S H I R K A

Un système de gestion de bases de connaissances centrées-objet

Manuel d'utilisation

François Rechenmann
INRIA Rhône-Alpes

Projet SHERPA - Unité de Recherche INRIA Rhone-Alpes / IMAG-LIFIA - mai 1993

Table des matières

Première partie : le modèle

1. Structure des schémas 11
1.1 Les Facettes 11
1.2. Classe d'un attribut 12
2. Classes et instances 14
3. Les facettes de typage $un et $liste-de 16
4. La facette de commentaire $com 17
5. Les facettes de restriction de type 18
5.1. La facette $domaine 18
5.2. La facette $sauf 18
5.3. La facette $intervalle 18
5.4. La facette $a-verifier 19
5.5. Les facettes $cardmin et $cardmax 20
6. Les facettes $valeur et $defaut 21
7. Les facettes de définition et de manipulation des variables 22
8. Facettes d'obtention de valeurs 23
8.1. La facette $sib-filtre et le filtrage 23
8.2. La facette $sib-exec et l'attachement procédural 26
8.3. Combinaison de moyens d'obtention de valeurs 29
9. L'héritage 32
9.1. Classes et sous-classes 32
9.2. Héritage multiple 34
9.3. La spécialisation 35
10. Détermination de valeurs d'attribut manquantes 37
10.1. Héritage des moyens d'obtention 37
10.2. Rôle des variables 38
10.3. Accès aux attributs imbriqués 39
10.4. Accès aux attributs hérités 41
11. Les facettes de réaction sur événement 42
11.1. Les facettes $si-succes et $si-echec 42
11.2. Les facettes $avant-ajout et $apres-ajout 43
11.3. Les facettes $avant-mod et $apres-mod 44
11.4. Les facettes $avant-sup et $apres-sup 44
12. Visualisation des instances 45
13. Le méta-niveau 49
13.1. Les méta-schémas 49
13.2. Le schéma objet 50
13.3. Le schéma attribut et ses spécialisations 50
13.4. Le schéma facette et ses spécialisations 51
13.5. Les schémas méthode et prédicat 52
13.6. Utilisation des méta-schémas 54
14. Un exemple : la base "Objets-du-plan" 55
14.1. Les méthodes, les prédicats et les fonctions Le-Lisp associées 55
14.2. Les classes d'objets du plan 57
14.3. Les instances 60

Deuxième partie : les commandes et les fonctions

1. Installation de Shirka 66
1.1 Installation directe 66
1.1 Installation complète 66
2. Lancer et quitter Shirka 68
3. L'aide en ligne et les facilités 69
3.1. Aide en ligne ? ?? et ??? 69
3.1.1. Liste des commandes disponibles 69
3.1.2. Informations sur une commande 69
3.1.3. Informations sur les paramètres d'une commande 70
3.1.4. Liste des commandes sous forme de symboles 70
3.2. Fin 71
3.3. Appel récursif de Shirka & 71
3.4. Valeur(s) de l'appel précédent = et << 71
3.5. Pas de valeur < 71
3.6. Poursuivre une instanciation > 72
3.7. Appel de lisp + ... 72
4. Les erreurs 73
5. Les commandes 74
5.1. Chargement d'une base : charger, verifier 75
5.2. Sauvegarde d'une base : sauver 75
5.3. Liste des attributs : l-att 76
5.4. Liste des instances d'une classe : l-inst 76
5.5. Liste des spécialisations d'une classe : l-spec 76
5.6. Création d'instances : cr-inst 77
5.7. Suppression d'une instance: sup-inst 78
5.8. Visualisation d'une instance : vi 79
5.9. Demande de valeur pour un attribut: val? 80
5.10. Ajout de valeur dans un attribut: aj-val 80
5.11. Modification de valeur dans un attribut: mod-val 81
5.12. Suppression de valeur dans un attribut: sup-val 81
5.13. Exécution d'une méthode : exec 82
5.14. Environnement de spécialisation et de classification: classif 83
5.14.1. Spécialisation d'une instance: spec 86
5.14.2. Classification d'une instance: class 87
5.14.3. Rattachement d'une instance: ratt 90
5.14.4. Explication globale de la classification d'une instance : expl-g 91
5.14.5. Explication d'une spécialisation ou d'une classification : expl 91
5.14.6. Affichage de la valeur d'un attribut d'une instance classée ou spécialisée : v? 91
5.14.7. Affichage des valeurs des attributs d'une instance classée ou spécialisée : vi-c 92
5.14.8. Modification de la valeur d'un attribut d'instance classée ou spécialisée : m-v 92
5.14.9. Suppression de valeur dans un attribut d'instance classée ou spécialisée : s-v 93
5.14.10. Ajouter une valeur dans un attribut d'instance classée ou spécialisée : a–v 94
5.14.11. Réinitialisation de la spécialisation et de la classification : init 94
5.15. Accès au tableur ShirCalc: tabl 95
5.16.Demande d'informations sur des classes : info 96
5.17.Demande d'informations sur des attributs d'une classe : info-att 96
6. Les fonctions 98
6.1. def-sh 98
6.2. #:shirka:noyau:schema 98
6.3. #:shirka:noyau:nom-sch 99
6.4. #:shirka:moteur:valeur? et val? 99
6.5. #:shirka:moteur:aj-valeur et affect 99

6.6. #:shirka:moteur:mod-valeur 100
6.7. #:shirka:moteur:sup-valeur 100
6.8. #:shirka:moteur:sup-inst 101
6.9. #:shirka:moteur:creer-instance 101
6.10. #:shirka:vi:visualiser 101
6.11. #:shirka:classif:class-inst 102
6.12. #:shirka:noyau:peres <schema> 102
6.13. #:shirka:noyau:ancetres <schema> 102
6.14. #:shirka:noyau:l-nom-att-inst-he <schema> 103
6.15. #:shirka:noyau:pos-nom-att <schema> <nom-att> 103
6.16. #:shirka:noyau:val-inst <instance> <pos-att> 103
6.17. #:shirka:noyau:val-facette <schema> <nom-att> <nom-fac> 104
6.18. #:shirka:noyau:val-fac-he <schema> <nom-att> <nom-fac> 104
6.19. #:shirka:noyau:l-spec <schema> 104
6.20. #:shirka:noyau:l-spec-t <schema> 105
6.21. #:shirka:noyau:l-spec-t-vraies <schema> 105
6.22. #:shirka:noyau:l-inst-t <schema> 105
6.23. #:shirka:moteur:admissible? <valeur> <nom-att> <schema> <instance> 106
6.24. #:shirka:top:dem-valeur <schema> <nom-att> <instance> <branger> <bcommande> 106
6.25. #:shirka:top:commande <nom-environnement> <prompt> 107
6.26. #:shirka:classif:spec-inst <instance> <nom-sch-dest> <binf> <bdial> 107
6.27. #:shirka:classif:ratt-inst <instance> <nom-sch-dest> 108
6.28. #:shirka:classif:fin-cl <liste-instances> 108

7. Le développement d'applications en Shirka 109
7.1. Shirka comme couche de base d'une application 109
7.2. Utilisation directe de Shirka 109
7.2.1. Définition de nouvelles commandes 109
7.2.2. Définition d'environnements 110

ANNEXES

Glossaire 115
Liste des facettes 117
Syntaxe 119
Liste des classes prédéfinies 119
Liste des attributs prédéfinis 119
Commandes sous forme de symboles 119
Shirka en Shirka 120
Schémas des commandes Shirka: 122
LISTE DES FONCTIONS (de…, (dmd…, (defvar… dans <SHIRKA>.ll 125
Commande "verifier" 135
Index 141
Commandes 141
Commandes sous forme de symboles 141
Fonctions 142

Shirka est un système de gestion de bases de connaissances centrées-objet. Il utilise un modèle de représentation des connaissances inspiré des "frames", dans lequel les connaissances sont organisées en unités interdépendantes. Une unité de représentation est appelée ***schéma***. Un schéma représente une classe d'objets ou un objet particulier appelé ***instance***. Le terme d'objet doit être pris dans son acception la plus large : ce peut être un objet physique ou conceptuel, un contexte ou une situation, ou toute autre entité sémantiquement définie. Une classe et ses instances possèdent des ***attributs***. Un attribut peut représenter une propriété, une caractéristique ou un lien vers d'autres classes ou représentants. Un attribut est à son tour décrit par une liste de ***facettes*** valuées ; les valeurs sont des schémas ou des références à des schémas. Les classes sont organisées en une structure de treillis dans laquelle une classe hérite des attributs des classes qui la dominent.

Utiliser Shirka, c'est définir des classes d'objets interdépendantes, puis créer des instances de ces classes, des objets particuliers, et poser des questions sur leurs attributs indéterminés. Shirka tente alors de répondre à ces questions en utilisant la connaissance introduite dans les classes lors de leur définition. Créer de nouvelles classes et créer des instances de classes existantes sont des travaux de natures et de difficultés différentes qui peuvent être supposés exécutés par des personnes distinctes, le concepteur de la base et l'utilisateur.

Par rapport à d'autres systèmes existants qui utilisent une représentation de connaissance centrée-objet, Shirka présente plusieurs avantages et originalités :

- Le modèle utilisé par Shirka est totalement uniforme. Tout schéma, attribut ou facette est une instance d'un schéma de classe de plus haut niveau. De même, les valeurs apparaissant dans une facette sont toujours des instances ou des références à des instances.
- L'inférence des valeurs d'attribut indéterminées ne fait pas appel à des règles, mais à des filtres décrits par des schémas. Il n'y a donc pas de combinaison de plusieurs représentations de connaissance.
- Par contre, plusieurs mécanismes d'inférence sont disponibles : héritage, attachement procédural, filtrage, valeur par défaut, spécialisation et classification. L'attachement procédural, qui consiste à associer une ou plusieurs procédures à un attribut d'une classe, utilise des descriptions externes, en termes de schémas, des procédures employées. Leur appel est donc entièrement contrôlé par Shirka.
- Le treillis des classes peut servir de support à un mécanisme de classification, qui cherche à déterminer les classes potentielles d'une instance donnée.
- Les attributs peuvent être mono ou multi-valués. Il est donc possible de traiter explicitement des listes de valeurs, élément par élément ou globalement.
- Shirka offre des moyens de gérer la cohérence entre les instances par l'intermédiaire de procédures attachées aux attributs et activées lors de certaines manipulations sur leurs valeurs, telles que l'ajout, la modification et la suppression. Un schéma de classe étant une instance d'une autre classe, cette gestion de cohérence peut être étendue aux classes elles-mêmes.
- Il est possible de définir, pour chaque attribut, les moyens de passer de la représentation interne à la représentation externe.

Shirka est bien adapté au développement d'applications pour lesquelles doivent coexister une base d'objets, une base de méthodes algorithmiques et une base de connaissances. Ces connaissances y sont en fait organisées autour des objets et des méthodes et permettent leur appariement : retenir la bonne méthode pour un objet dans un contexte donné.

Ce manuel présente la version 3 de Shirka. La première partie expose le modèle de représentation, les schémas. Elle définit les différentes facettes et leurs interprétations. Les exemples employés sont de compréhension immédiate et ne font pas référence à un domaine d'application particulier. Elle est suivie d'un exemple de base de connaissance complète. La deuxième partie présente les commandes accessibles à l'utilisateur, ainsi que l'ensemble des fonctions Le-Lisp utilisables lors de l'écriture des méthodes. Le manuel est complété par un glossaire et un rappel de la signification de chacune des facettes.

Remerciements

Le modèle et le logiciel Shirka® sont le résultat d'un travail collectif, à l'INRIA et au Laboratoire ARTEMIS à Grenoble.

Que soient ici remerciés : J.L. Aguirre, A. Bensaïd, D. Bloch, S. Demeyter, M.S. Doize, J. Euzenat, J.F. Gonguet, D. Granier, B. Pivot, M. Prokop, N. Rouibah, B. Rousseau, A. Sales-Simonet et P. Vignard.

Le logiciel Shirka, dans son état actuel de prototype, a été utilisé dans des projets de développement de systèmes à base de connaissances ou pour la réalisation d'outils adaptés. Les remarques, ainsi que les (nombreuses?) anomalies de fonctionnement notées à ces occasions, ont permis d'y apporter des améliorations notables. Que soient donc remerciés à ce titre : J.L. Gouzé et J. Grimm de l'unité de recherche INRIA de Sophia-Antipolis, A. Schmitt de la société ACT Informatique, S. Maure et C. Boucher de la société Apside Technologies, M. Prevosto de l'IFREMER, J.C. Protais de la société Sogitec, A. Messean et ses collègues à l'INRA de Jouy-en-Josas, A. Pavé et ses collègues au Laboratoire de Biométrie de l'Université Claude Bernard à Lyon et P. Puvilland du Bureau des Recherches Géologiques et Minières (BRGM) d'Orléans.

Pour tout renseignement ou information sur le logiciel Shirka, s'adresser à :

François RECHENMANN

Unité de Recherche INRIA Rhône-Alpes
IMAG-LIFIA
46, Avenue Félix Viallet
38031 GRENOBLE Cedex 1 (FRANCE)

E-mail: Francois.Rechenmann@imag.fr

Le-Lisp et Shirka sont des marques déposées par l'INRIA
Unix® est un nom déposé des laboratoires AT&T (Bell Labs.)
Macintosh™ est une marque déposée de Apple Corp.

Première partie

Le modèle

1. Structure des schémas

Un schéma est défini par son nom et la liste de ses attributs. Chaque attribut est à son tour décrit par un nom et une liste de facettes. Une facette possède un nom et une liste de valeurs (Figure 1).

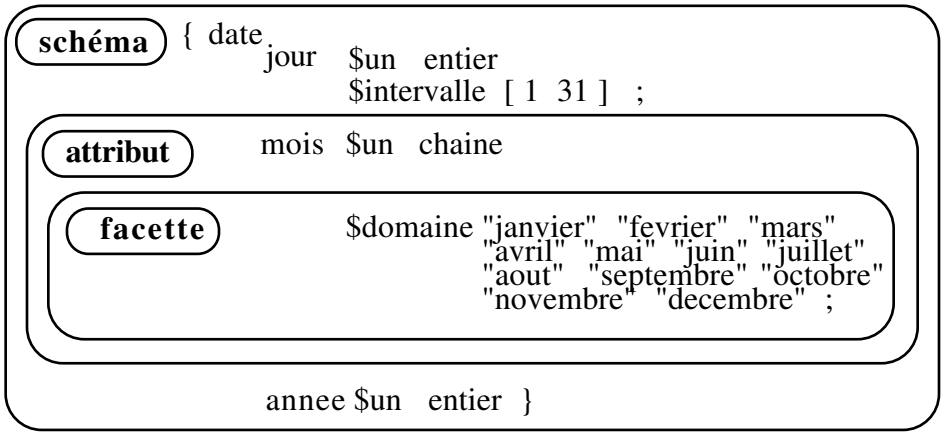


Figure 1 : structure d'un schéma.

Les noms de schémas et d'attributs sont choisis par l'utilisateur. Un même nom ne doit pas être donné à des schémas différents. Par contre, des attributs distincts peuvent avoir le même nom s'ils apparaissent dans des schémas qui n'appartiennent pas à une même famille (cf. §9).

1.1 Les Facettes

Les facettes sont choisies dans un ensemble prédéfini qui ne peut pas être étendu par l'utilisateur. Les facettes admises par la version 3 de Shirka sont les suivantes :

$un	**$liste-de**	**$com**	**$domaine**
$sauf	**$intervalle**	**$a-verifier**	**$cardmin**
$cardmax	**$valeur**	**$defaut**	**$var-nom**
$var<-	**$var->**	**$var-liste<-**	**$var-liste->**
$sib-exec	**$sib-filtre**	**$si-succes**	**$si-echec**
$avant-ajout	**$apres-ajout**	**$avant-mod**	**$apres-mod**
$avant-sup	**$apres-sup**	**$avant-vi**	**$apres-vi**
$pour-vi	**$en-vi**		

Les valeurs qui suivent ces noms sont toujours des schémas ou des références à des schémas par leur nom. Elles sont interprétées différemment suivant la facette où elles interviennent. Connaître et savoir utiliser Shirka c'est donc avant tout comprendre la signification de chacune des facettes disponibles, qui peuvent se classer en plusieurs catégories :

- facettes de typage (§3) :
  `$un  $liste-de`
- facette de commentaire (§4) :
  `$com`
- facettes de restriction de type (§5) :
  `$domaine  $sauf  $intervalle  $a-verifier  $cardmin  $cardmax`
- facette de définition de valeurs (§6) :
  `$valeur  $defaut`
- facettes de définition et de manipulation des variables (§7) :
  `$var-nom  $var->  $var<-  $var-liste<-  $var-liste->`
- facettes d'obtention des valeurs d'attribut (§8) :
  `$sib-filtre  $sib-exec`
- facettes de réaction sur événement (§11) :
  `$si-succes  $si-echec  $avant-ajout  $apres-ajout`
  `$avant-mod  $apres-mod  $avant-sup  $apres-sup`
- facettes de visualisation (§12) :
  `$avant-vi  $apres-vi  $en-vi  $pour-vi`

La signification de chacune des facettes est rappelée en annexe.

Dans un attribut, les facettes peuvent apparaître dans un ordre quelconque. Cependant, *il ne peut y apparaître deux facettes de même nom*, par exemple deux facettes **`$intervalle`** ou deux facettes **`$sib-filtre`**. Cette restriction ne pose aucun problème, car un nom de facette peut être suivi généralement d'un nombre non limité de valeurs.

1.2. Classe d'un attribut

Le nom d'un attribut peut être précédé du nom de sa classe. Par défaut, c'est-à-dire en l'absence de ce nom de classe, un attribut est considéré comme étant de la classe **`attribut`** (cf. §13.3). Certaines commandes et fonctions Shirka admettent comme paramètres la liste des classes d'attributs à considérer et la liste des classes d'attributs à ignorer. Il existe quelques sous-classes d'attribut prédéclarées dans Shirka (cf. §13.3).

Pour préciser la classe d'un attribut quand elle est différente de **`attribut`**, il suffit de faire précéder le nom de l'attribut du nom de cette classe, lui-même préfixé par le caractère **`&`**. Au préalable, cette classe d'attribut doit avoir été créée.

Exemple 1a :

```
{ personne
     sorte-de  =         objet ;
     age       $un       entier;
```

```
     &morpho   poids     $un  entier ;
     &morpho   taille    $un  entier ;
     nom                 $un  chaine }
```

avec la déclaration correspondante pour la classe d'attributs "morpho":

```
{ morpho
     sorte-de  =    attribut }
```

Dans cet exemple, les deux attributs **poids** et **taille** sont de la classe **morpho**. Il sera ainsi possible de lancer la classification d'une instance de **personne** en ne prenant en compte que les attributs de la classe **morpho** et en ignorant les autres, tel que **age** ou **nom** (cf. §5.14.2 dans la deuxième partie : *commandes* et *fonctions*).

2. Classes et instances

Un schéma peut décrire une classe d'objets ou un représentant particulier d'une classe. On parlera respectivement de *schéma de classe* ou de *schéma d'instance*, ou plus simplement de classe et d'instance. Il n'y a pas de différence structurelle entre un schéma de classe et un schéma d'instance. Cependant, s'il est vrai que tout schéma de classe est en fait lui-même une instance d'un schéma de plus haut niveau, appelé ***méta-schéma***, toute instance ne peut pas être systématiquement considérée comme un schéma de classe par un niveau inférieur. Ces distinctions seront éclaircies au §13. Un cas particulier de schémas (des sous-schémas pour être tout à fait rigoureux) dits "schémas non typés", du fait de l'absence de facette de typage (§3), se rapporte à la *spécialisation* et au *filtrage* (cf. §8 et §9).

Dans une base en exploitation, le nombre des instances est normalement très supérieur au nombre de classes. Les instances sont créées soit directement par l'utilisateur à l'aide de la commande **`cr-inst`** (2eme partie, §5.6), soit par le système lors d'un attachement procédural (1ere partie, §8.2), soit dans le corps d'une fonction Le-Lisp par appel à la fonction **`creer-instance`** (2eme partie, §6.9).

Alors que les attributs dans un schéma de classe sont décrits chacun par une liste de facettes, les attributs d'une instance ne sont précisés que par leurs valeurs. Les attributs d'une instance sont les mêmes que ceux apparaissant dans la description de sa classe.

Exemple 2a :

Le schéma suivant décrit la classe des dates au moyen de trois attributs : **`jour`**, **`mois`** et **`annee`**. Le jour est un entier compris entre 1 et 31, le mois est une chaîne de caractères dont la valeur peut être **`janvier`**, **`fevrier`**,..., **`novembre`** ou **`decembre`**, l'année est un entier quelconque.

```
{ date
     sorte-de  =             objet ;
     jour      $un           entier
               $intervalle   [ 1 31 ] ;
     mois      $un           chaine
               $domaine      "janvier" "fevrier" "mars" "avril"
                             "mai" "juin" "juillet" "aout" "septembre"
                             "octobre" "novembre" "decembre" ;
     annee     $un           entier }
```

Un représentant de cette classe, une date particulière, pourrait être :

```
{ date-23
     est-un    =    date ;
     jour      =    15 ;
     mois      =    "avril" ;
     annee     =    1987 }
```

La signification de l'attribut **`sorte-de`** sera expliquée au §9, où l'on verra qu'il s'agit en fait d'un *méta-attribut* (ie, défini dans un *méta-schéma*), ce qui explique qu'il se trouve instancié dans la définition du schéma **date**.

La syntaxe des schémas apparaît en partie sur cet exemple :
- les schémas sont encadrés par des accolades: **{** <schema> **}**
- les attributs sont séparés par un point-virgule **;**
- les intervalles sont délimités par les crochets: **[** <intervalle> **]**
- le signe **=** permet de définir une valeur d'attribut au niveau d'une instance
- les valeurs composant une liste sont séparées par un caractère blanc.

Toute instance est liée à une seule classe grâce à l'attribut **est-un**, possédé par tous les schémas de classe. Un nom peut lui avoir été fourni soit par l'utilisateur, soit par le système. *Une instance peut être incomplète : les attributs de sa classe n'ont pas tous une valeur*. Les valeurs manquantes peuvent éventuellement être obtenues par le moteur d'inférences de Shirka à partir des informations contenues dans le schéma de classe, en particulier des facettes **$valeur**, **$sib-filtre**, **$sib-exec**, **$defaut**, **$var<-** et **$var-liste<-**.

Les valeurs d'un attribut dans une instance sont
- soit d'un type simple :
  - **nombre** et ses spécialisations **entier** et **reel.** Une valeur entière ou réelle obéit à la syntaxe Le-Lisp.
  - **booleen**. Une valeur booléenne est **vrai** ou **faux**.
  - **chaine**. C'est une chaîne de caractères encadrés par des **" "** ; elle obéit à la syntaxe Le-Lisp.
  - **symbole**. Un symbole est une suite de caractères alphanumériques quelconques sans espacement et débutant par une lettre ; il n'est pas encadré par des **" "**.
- soit d'un type faisant référence à une classe. La valeur est soit un nom d'instance de cette classe, soit directement une instance.

Exemple 2b :

Dans le schéma de classe **personne** suivant, l'attribut **date-de-naissance** est du type **date**. Sa valeur dans l'instance **personne-12** est une date particulière spécifiée par son nom, éventuellement incomplète.

```
{ personne
    sorte-de           =    objet ;
    nom                $un  chaine ;
    age                $un  entier ;
    date-de-naissance  $un  date }

{ personne-12
    est-un             =    personne ;
    nom                =    "jean" ;
    date-de-naissance  =    date-1 }

{ date-1
    est-un    =    date ;
    mois      =    "mars" ;
    annee     =    1954 }
```

Les facettes permettant de définir le type d'un attribut sont les facettes:
$un et **$liste-de**

Ces facettes sont *obligatoires* dans cette phase de définition.

3. Les facettes de typage $un et $liste-de

Ces deux facettes sont obligatoires. Elles sont suivies de l'indication du type, qui peut être simple : **nombre**, **chaine**, **booleen**, **symbole** ou faire référence à un schéma de classe par son nom.

La différence entre **$un** et **$liste-de** est liée au nombre de valeurs possibles pour l'attribut. La facette **$un** précise que l'attribut ne peut avoir qu'une valeur, alors que la facette **$liste-de** précise que l'attribut peut posséder simultanément plusieurs valeurs, qui peuvent, dans certains cas (cf. §10.2.), être considérées comme constituant une liste.

Exemple 3a :

```
{ personne
     sorte-de            =              objet ;
     nom                 $un            chaine ;
     prenoms             $liste-de      chaine ;
     sexe-f              $un            booleen ;
     age                 $un            entier ;
     date-de-naissance   $un            date ;
     enfants             $liste-de      personne ;
     adresse             $un            adresse }

{ adresse
     sorte-de            =    objet ;
     numero              $un  entier ;
     rue                 $un  chaine ;
     ville               $un  chaine ;
     code-postal         $un  entier }

{ personne-23
     est-un              =    personne ;
     nom                 =    "dupond" ;
     prenoms             =    "pierre" "paul" ;
     sexe-f              =    faux ;
     date-de-naissance   =    date-8 ;
     enfants             =    personne-4 personne-56 ;
     adresse             =    { est-un      =    adresse ;
                                numero      =    12 ;
                                rue         =    "de la poste" ;
                                ville       =    "Trifouillis" } }
```

La valeur de l'attribut **adresse** dans l'instance **personne-23** est directement une instance, *et non pas une référence à une instance par son nom*, comme dans le cas de l'attribut **date-de-naissance**. Il s'agit d'une instance non nommée et donc *non partageable*.

4. La facette de commentaire $com

La facette **$com** permet d'associer un commentaire à un attribut. Le commentaire est une suite de chaînes de caractères entre **" "**, séparées par un espace. Il est exploité par Shirka pour la création interactive d'instances ; le commentaire peut s'afficher à l'écran à la demande de l'utilisateur. Il facilite également la maintenance des bases de connaissances en les documentant; en particulier, il est possible d'ajouter un commentaire à la classe elle-même, grâce à l'attribut **lui-meme** (cf §8.1).

Exemple 4a :

```
{ point
     sorte-de  =         objet ;
     lui-meme  $com      "le point est defini par le couple (x,y) " ;
     xcoord    $un       reel
               $com      "abscisse du point " "exprimee en mm." ;
     ycoord    $un       reel
               $com      "ordonnee du point " "exprimee en mm." }
```

5. Les facettes de restriction de type

Les types introduits par les facettes **$un** et **$liste-de** définissent un domaine pour les valeurs de l'attribut auquel elles sont attachées. Ce domaine peut être restreint par l'utilisation des facettes **$domaine**, **$sauf**, **$intervalle** et **$a-verifier**. Dans le cas de listes de valeurs, les facettes **$cardmin** et **$cardmax** apportent des restrictions sur le nombre de valeurs admises.

5.1. La facette $domaine

La facette **$domaine** est suivie de toutes les valeurs possibles de l'attribut auquel elle est associée. Ces valeurs sont des constantes de même type que l'attribut.

Exemple 5.1 :

```
{ carte-a-jouer
     sorte-de  =         objet ;
     hauteur   $un       objet-simple
               $domaine  as 2 3 4 5 6 7 8 9 10 valet dame roi ;
     couleur   $un       chaine
               $domaine "coeur" "carreau" "pique" "trefle" }
```

5.2. La facette $sauf

La facette **$sauf** est suivie de toutes les valeurs que l'attribut *ne peut pas* prendre. Ces valeurs sont des constantes de même type que celui de l'attribut.

Exemple 5.2 :

```
{ division-euclidienne
     sorte-de       =         methode ;
     dividende      $un       entier ;
     diviseur       $un       entier
                    $com      "un entier non nul"
                    $sauf     0 ;
     reste          $un       entier }
```

5.3. La facette $intervalle

La valeur de cette facette est une suite d'intervalles définis par leurs bornes minimum et maximum, entre crochets **[]**. Les différents intervalles doivent apparaître dans le sens des bornes croissantes. Les bornes sont des constantes du même type que celui de l'attribut où la facette apparaît. *Ce type doit être simple* :

- **nombre** (**entier** ou **reel**) ; les valeurs **+inf** (infini positif) et **-inf** (infini négatif) sont reconnues ;
- **chaine** (ordre alphabétique) ;
- **booleen** (faux < vrai) ;
- **symbole** (ordre alphanumérique).

Exemple 5.3 :

```
{ enfant
     sorte-de  =             personne ;
     age       $un           entier
               $intervalle   [ 0 15 ] }
```

5.4. La facette $a-verifier

Elle permet d'attacher à un attribut une restriction sous la forme d'un prédicat qui doit être vérifié pour toute valeur de l'attribut. Ce prédicat est décrit de façon externe par un schéma, spécialisation de la classe prédéfinie **predicat**. Il doit lui correspondre une fonction Le-Lisp qui, s'évaluant dans le bon contexte, renvoie **faux** ou **vrai** (en fait 'nil' ou 'non-nil'; **faux** et **vrai** seront toujours employés dans la suite du manuel). Le nom de cette fonction doit figurer comme valeur de l'attribut **nom-fct** hérité de la classe **predicat**. La spécification de ces prédicats, ainsi que l'écriture des fonctions correspondantes, sera exposée plus en détail au §8.2 qui traite du problème plus général de l'attachement procédural.

Exemple 5.4.a :

Une autre manière d'exprimer que l'âge d'une personne doit être compris entre deux bornes est de définir le prédicat **compris-entre** :

```
{ compris-entre
     sorte-de       =          predicat ;
     nom-fct        $valeur    compris-entre ;
     valeur         $un        entier ;
     borne-inf      $un        entier ;
     borne-sup      $un        entier }
```

et de placer derrière la facette **$a-verifier** de **age** une sous-classe non-nommée de ce prédicat :

```
{ enfant
     sorte-de  =              personne ;
     age       $un            entier
               $a-verifier    { compris-entre
                                   valeur         $var<-    age ;
                                   borne-inf      $valeur   0 ;
                                   borne-sup      $valeur   15 } }
```

La facette **$var<- age** signifie que la valeur de l'attribut **valeur** de **compris-entre** est la valeur de la variable **age**, elle-même liée à l'attribut **age**. La manipulation des variables est détaillée aux §7 et §10.2.

La fonction Le-Lisp, associée au prédicat **compris-entre** par l'attribut **nom-fct** de type **symbole** pourrait s'écrire :

```
(de compris-entre (inst)
     ;l'instance du schema compris-entre
     ;est transmise a la fonction
     (let      ((valeur  (val? 'valeur)))
               (and      (ge  valeur
                         (val? 'borne-inf))
                         (le  valeur
                         (val? 'borne-sup)))))
```

N.B. L'utilisation de la facette **$a-verifier** dans cet exemple présente l'avantage par rapport à la facette **$intervalle** de permettre d'envisager des bornes *variables*, valeurs d'autres attributs du schéma.

Lorsque les conditions à vérifier portent sur plusieurs attributs, elles doivent figurer dans la facette **$a-verifier** de l'attribut prédéfini **lui-meme**, qui possède par ailleurs d'autres usages (cf. §8.1).

Exemple 5.4.b :

```
{ carre
     sorte-de  =              rectangle ;
     lui-meme  $a-verifier    { egalite
                                   x    $var<-    largeur ;
                                   y    $var<-    longueur } ;
     largeur ;
     longueur }

{ egalite
     sorte-de  =         predicat ;
     nom-fct   $valeur   egalite ;
     x         $un       reel ;
     y         $un       reel ;
     epsilon   $un       reel
               $defaut   1.0e-8 }

(de egalite    (inst)
     (<=  (abs      (-   (val? 'x)
                         (val? 'y))
                    (val? 'epsilon)))
```

5.5. Les facettes $cardmin et $cardmax

Les facettes **$cardmin** et **$cardmax** permettent de préciser respectivement un nombre minimum et maximum de valeurs pour un attribut dont le type a été spécifié par la facette **$liste-de**. Ce nombre est une constante entière positive ou nulle. Toute tentative d'ajouter une valeur supplémentaire à un attribut **liste-de**, limité à [**$cardmax**] valeurs, se soldera par un échec. De même, toute tentative de supprimer une valeur d'une liste de [**$cardmin**] valeurs se soldera par un échec.

Exemple 5.5 :

```
{ personne
          sorte-de  =              objet ;
          nom       $un            chaine ;
          prenoms   $liste-de      chaine
                    $com           "pas plus de trois prenoms"
                    $cardmax  3
                    $cardmin  1 }
```

6. Les facettes $valeur et $defaut

La facette **$valeur** associe à un attribut une valeur, qui est valable pour toutes les instances de la classe décrite par le schéma où apparaît cet attribut. Cette valeur peut être multiple dans le cas où le type de l'attribut a été introduit avec la facette **$liste-de**. Dans tous les cas, la valeur précisée est une constante qui doit satisfaire toutes les contraintes connues sur l'attribut.

Exemple 6a :

```
{ gueridon
     sorte-de  =    table ;
     nb-pieds  $valeur   3 }
```

La facette **$defaut** a un rôle similaire, mais sa valeur n'est considérée dans une instance que dans la mesure où une autre valeur n'y figure pas. Plus généralement, une valeur par défaut peut être masquée, lors de l'héritage, par d'autres valeurs ou moyens d'obtenir des valeurs (cf. §9).

Exemple 6b :

```
{ table
     sorte-de  =              meuble ;
     nb-pieds  $un            entier
               $intervalle    [ 3 6 ]
               $defaut        4 }

{ table-de-ma-grand-mere
     est-un    =    table ;
     nb-pieds  =    3 }
```

7. Les facettes de définition et de manipulation des variables

Les variables permettent fondamentalement de relier des attributs. Dans les exemples d'utilisation de prédicats dans la facette **`$a-verifier`**, la facette **`$var<-`** spécifie que la valeur de l'attribut où elle apparaît peut être obtenue en recherchant la valeur de l'attribut dont elle porte le nom. Plus généralement, la facette **`$var-nom`** associe à un attribut une variable par laquelle il sera possible de faire référence à la valeur de ce dernier. Cette facette est souvent facultative: en effet, une variable d'un nom donné fait référence à l'attribut de même nom. Cependant, des ambiguïtés peuvent survenir si, dans un même schéma et par le jeu des imbrications, des attributs distincts portent le même nom.

La facette **`$var<-`** indique que la valeur de l'attribut est celle de la variable dont le nom suit. Symétriquement, la facette **`$var->`** indique que la valeur de l'attribut est à placer, quand elle est connue, dans la variable dont le nom suit.

Les facettes **`$var-liste<-`** et **`$var-liste->`** jouent le même rôle que les deux facettes précédentes, mais ne sont employées que pour faire référence à des attributs multivalués, c'est-à-dire des attributs dont le type est introduit par la facette **`$liste-de`**. Ces facettes précisent que les valeurs de ces attributs sont manipulées globalement, en tant que listes de valeurs, et non plus élément par élément.

De nombreux exemples d'utilisation des variables apparaîtront dans le chapitre suivant car elles sont indispensables à l'écriture des filtres et des attachements procéduraux.

8. Facettes d'obtention de valeurs

Il s'agit des facettes **$sib-filtre** et **$sib-exec** (**sib** pour "si-besoin"). La première permet de déterminer la ou les valeurs d'un attribut en les extrayant d'une ou de plusieurs instances satisfaisant un ensemble de conditions, sous la forme d'une suite de filtres. La facette **$sib-exec** réalise un attachement procédural qui permet de calculer la ou les valeurs de l'attribut auquel elle est attachée. Le calcul est effectué par une fonction Le-Lisp adéquate.

8.1. La facette $sib-filtre et le filtrage

Un filtre est un ensemble de conditions que doivent satisfaire des instances. Le filtre est utilisé pour retrouver ces instances. Il est décrit par un schéma non typé qui porte le nom du schéma de classe qui décrit les instances cherchées. Sa structure en est identique, mais chacun de ses attributs peut posséder des restrictions supplémentaires par rapport à celles figurant dans le schéma de classe.

Exemples 8.1.a :
Voici quelques exemples de filtres simples, tels qu'ils pourraient apparaître derrière une facette **$sib-filtre** :

- Ensemble des personnes (instances du schéma de classe **personne**), dont l'âge est compris entre 25 et 55 ans ou entre 60 et 70 ans, et dont le nom est **jean** ou **pierre** :

```
{ personne
    nom  $domaine      "jean" "pierre" ;
    age  $intervalle   [ 25 55 ] [ 60 70 ] }
```

- Ensemble des dates dont le jour est un 25 :

```
{ date
    jour      $domaine  25 }
```

- Ensemble des points du premier quadrant du plan :

```
{ point
    x-coord   $intervalle   [ 0.0 +inf ] ;
    y-coord   $intervalle   [ 0.0 +inf ] }
```

- Les parents de la personne **personne-23°:**

```
{ personne
    enfants   $a-verifier   { apparait-dans
                                element   $valeur       personne-23 ;
                                liste     $var-liste<-  enfants }°}
```

- La mère de **personne-23°:**

```
{ personne
    sexe-f    $domaine      vrai ;
    enfants   $a-verifier   { apparait-dans
                                element   $valeur       personne-23 ;
                                liste     $var-liste<-  enfants }°}
```

En pratique, les attributs d'un filtre font référence à des attributs du schéma de classe dans lequel il apparaît, et ce par l'intermédiaire des variables.

Exemple 8.1.b :

Soit la définition suivante de la classe des personnes :

```
{ personne
     sorte-de       =              objet ;
     a-pour-pere    $un            personne ;
     pere-de        $liste-de      personne }
```

et les instances incomplètes :

```
{ personne-1
     est-un         =    personne ;
     a-pour-pere    =    personne-3 }

{ personne-2
     est-un         =    personne ;
     a-pour-pere    =    personne-3 }

{ personne-3
     est-un         =    personne }
```

Il est alors possible de déterminer la valeur de l'attribut **pere-de** de l'instance **personne-3** si le schéma de classe **personne** est complété au niveau de cet attribut par la facette **$sib-filtre** :

```
{ personne
     sorte-de       =              objet ;
     lui-meme       $var-nom       lui ;
     a-pour-pere    $un            personne ;
     pere-de        $liste-de      personne
                    $sib-filtre    { personne
                                         lui-meme       $var-> pere-de ;
                                         a-pour-pere    $var<- lui }°}
```

L'attribut **lui-meme** est implicitement défini dans tout schéma de classe. Pour une instance de cette classe, sa valeur est l'instance elle-même.

Dans l'exemple, il lui est associé une variable nommée **lui**. En effet, par l'utilisation du filtre derrière la facette **$sib-filtre**, il apparaît deux attributs **lui-meme** dans le même schéma. La facette **$var-nom** permet de lever cette ambiguïté. Dans le filtre proprement dit, l'attribut **lui-meme** prend pour valeur la référence à chaque instance qui satisfait le filtre. La facette **$var->** indique que cette valeur doit devenir celle de l'attribut **pere-de** par l'intermédiaire de la variable de même nom. Comme l'attribut **pere-de** est multi-valué, le filtre ramène toutes les instances de **personne** le satisfaisant. L'attribut **lui-meme**, et donc la variable **lui**, prennent successivement ces instances comme valeurs, qui s'accumulent dans l'attribut **pere-de** de l'instance **personne-3**.

Les filtres peuvent être imbriqués dans les cas où la valeur d'un attribut d'un premier filtre est déterminée elle-même par un filtre.

Exemple 8.1.c :

```
{ homme
     sorte-de       =              personne ;
     lui-meme       $var-nom       lui ;
     a-pour-pere    $un            homme ;
     petits-fils    $liste-de      homme
                    $com           "les petits-fils d'un homme sont les "
                                   "hommes qui ont pour pere les "
                                   "hommes qui ont pour pere l'homme "
                                   "dont il est question "
                    $sib-filtre
                         { homme
                              lui-meme $var->    petits-fils ;
                              a-pour-pere
                                   $var-nom      pere
                                   $sib-filtre
                                        { homme  lui-meme $var-> pere ;
                                                 a-pour-pere $var<- lui}}}
```

Le filtre imbriqué qui suit la facette **$sib-filtre** du schéma **homme** permet de déterminer pour un homme donné la liste de ses petits-fils en tant que valeur de l'attribut **petits-fils**.

Ainsi, avec les instances suivantes de **homme** :

```
{ Pierre
     est-un         =    homme ;
     a-pour-pere    =    Paul }
{ Paul
     est-un         =    homme ;
     a-pour-pere    =    Jean }
{ Jacques
     est-un         =    homme ;
     a-pour-pere    =    Paul }
{ Jean
     est-un         =    homme }
```

la requête **val? Jean petits-fils** renvoie la liste : **Pierre Jacques**.

Plusieurs filtres peuvent apparaître derrière la facette **$sib-filtre**. Ils sont essayés séquentiellement. Si l'attribut auquel ils sont associés est mono-valué, c'est-à-dire typé par la facette **$un**, la première instance satisfaisant le filtre sera retenue et le processus de filtrage arrêté. Si l'attribut est multi-valué, toutes les instances satisfaisant les filtres sont retenues, dans la limite de cardinalité éventuellement spécifiée par la facette **$cardmax**. Tous les filtres sont alors mis à contribution. Toutes les instances ramenées sont distinctes.

Enfin, il existe un cas particulier de filtrage dans lequel le filtre se révèle complètement instancié, c'est-à-dire que tous ses attributs ont une valeur. Dans ce cas, il est inutile d'effectuer un balayage des instances afin de retenir celles qui satisfont le filtre, car la valeur cherchée peut en être directement extraite. Ces filtres n'ont d'intérêt que s'ils apparaissent groupés derrière une même facette **$sib-filtre**. Seuls les filtres instanciables sont alors susceptibles de fournir une valeur.

Exemple 8.1.d :

Dans l'exemple suivant, on veut déterminer le nombre de points-vacances de base d'un salarié, à partir du montant de son revenu mensuel, au moyen de l'application d'un barème comportant plusieurs tranches. Les filtres situés, dans la facette **$sib-filtre** de l'attribut **nb-points-base,** peuvent être instanciés sans balayage des instances de **tranche**. Seul le filtre pour lequel la valeur de la variable **revenu-mensuel** satisfait l'intervalle associé à l'attribut **revenu** ramène une valeur : le nombre de points correspondant à la tranche.

```
{ salarie
     sorte-de            =    personne ;
     revenu-mensuel      $un  entier ;
     nb-points-base      $un  entier
                         $sib-filtre
                              { tranche
                                   revenu    $var<-         revenu-mensuel
                                             $intervalle    [0 1500];
                                   points    $valeur        0
                                             $var->         nb-points-base }
                              { tranche
                                   revenu    $var<-         revenu-mensuel
                                             $intervalle    [1500 2400];
                                   points    $valeur        50
                                             $var->         nb-points-base }

                              -    -    -    -    -    -

                              { tranche
                                   revenu    $var<-         revenu-mensuel
                                             $intervalle    [7801 8700];
                                   points    $valeur        480
                                             $var->         nb-points-base }
                              { tranche
                                   revenu    $var<-         revenu-mensuel
                                             $intervalle    [8701 +inf];
                                   points    $valeur        580
                                             $var->         nb-points-base}}
```

Avec l'instance suivante :

```
{ dupond
     est-un              =    salarie ;
     revenu-mensuel      =    8200 }
```

la requête **val? dupond nb-points-base** renvoie la valeur 480.

8.2. La facette $sib-exec et l'attachement procédural

La facette **$sib-exec** permet d'associer à un attribut une ou plusieurs méthodes de calcul de sa ou ses valeurs. Le terme de calcul est général : il s'agit de faire appel à des fonctions dont l'évaluation détermine la valeur attendue.

Chaque méthode est décrite par un schéma de classe, appelé schéma **methode**, dans lequel l'attribut prédéfini **nom-fct** de type **symbole** a pour valeur le nom de la fonction Le-Lisp correspondante. Les autres attributs décrivent les paramètres de la méthode. C'est un sous-schéma non typé qui est placé derrière la facette **$sib-exec**. Les attributs de la méthode y sont liés à d'autres attributs par des variables. Certains d'entre eux

peuvent recevoir des restrictions supplémentaires ou voir leurs valeurs affectées par la facette **$valeur**. Plus loin, il apparaîtra qu'un filtre est en fait une spécialisation du schéma de classe dont il porte le nom.

Exemple 8.2.a :

```
{ rectangle
     sorte-de  =    parallelogramme ;
     longueur       $un  reel
                    $intervalle    [ 0.0 +inf ] ;
     largeur        $un  reel
                    $intervalle    [ 0.0 +inf ] ;
     surface        $un  reel
                    $sib-exec      { produit
                                       x    $var<-    longueur ;
                                       y    $var<-    largeur ;
                                       z    $var->    surface } }

{ produit
     sorte-de  =         methode ;
     nom-fct   $valeur   produit ;
     x         $un       reel ;
     y         $un       reel ;
     z         $un       reel }
```

La fonction Le-Lisp associée à une méthode ne possède qu'un seul paramètre, nommé "inst", instance du schéma de classe décrivant la méthode.

Le mécanisme d'appel est donc le suivant : une instance du schéma méthode est créée ; ensuite, les valeurs de ses attributs sont recherchées et placées dans l'instance. A ce niveau, au moins un attribut ne peut recevoir une valeur : c'est l'attribut correspondant au paramètre de sortie de la méthode.

C'est donc une instance incomplète qui est transmise à la fonction dont le nom apparaît comme valeur de l'attribut **nom-fct** du schéma **methode**. La fonction accède aux valeurs des attributs de l'instance par des fonctions Shirka, détaillées dans la deuxième partie de ce manuel, complète à son tour l'instance des valeurs de l'attribut de sortie et renvoie cette instance complétée. Ces valeurs sont propagées à l'attribut intéressé par les variables.

Exemple 8.2.b :

Code de la fonction Le-Lisp **produit** citée dans l'exemple précédent :

```
(de produit (inst)
     (affect  'z    (*   (val? 'x)
                         (val? 'y) ) ) )
```

Si l'instance suivante de **rectangle** est créée :

```
{ rectangle-2
     est-un    =    rectangle ;
     longueur  =    3.55 ;
     largeur   =    6.20 }
```

La requête **val? rectangle-2 surface** renverra la valeur 22.01. Pour cela, une instance incomplète de produit est créée avec un nom interne au système :

```
{ %produit-123-104
     est-un    =    %produit-123 ;
     x         =    3.55 ;
     y         =    6.20 }
```

Cette instance est transmise à la fonction **produit** qui la renvoie complétée :

```
{ %produit-123-104
     est-un    =    %produit-123 ;
     x         =    3.55 ;
     y         =    6.20 ;
     z         =    22.01 }
```

Chaque appel de la fonction attachée à un schéma méthode crée donc une instance de ce schéma. En fait, l'appel de la fonction est précédé d'un filtrage sur les instances existantes. En cas de succès de cette étape de filtrage, le résultat est directement extrait de l'instance retenue et la fonction n'est pas appelée.

Il est possible d'éviter cette étape de filtrage préalable en précisant dans l'attribut **sorte-de** que le schéma définissant la méthode est une spécialisation de **methode-sans** (filtrage) et non plus de **methode**. Ceci est parfois nécessaire si la fonction Le-Lisp correspondante provoque des effets de bord.

L'utilisation d'un prédicat dans la facette **$a-verifier** suit la même procédure, la différence étant, dans ce cas, l'absence d'un attribut résultat. La fonction renvoie faux ou vrai, directement au système Shirka. Si, préalablement, lors du filtrage, une instance est trouvée, le prédicat s'évalue à **vrai**. Là encore, le filtrage peut être évité en définissant le schéma associé comme spécialisation de **predicat-sans** au lieu de **predicat**.

Comme pour la facette **$sib-filtre**, il peut apparaître plusieurs filtres derrière un **$sib-exec**. Les méthodes associées aux filtres sont essayées séquentiellement jusqu'à ce qu'une valeur soit rendue. Cependant, contrairement à la facette **$sib-filtre**, même dans le cas d'attributs multi-valués, le premier succès arrête le processus.

Exemple 8.2.c :

```
{ difference
     sorte-de  =         methode ;
     nom-fct   $valeur   difference ;
     op1       $un       entier ;
     op2       $un       entier ;
     diff      $un       entier }

(de difference (inst)
     (affect 'diff (- (val? 'op1) (val? 'op2))))

{ demander
     sorte-de  =         methode-sans ;
     nom-fct   $valeur   demander ;
     question  $un       chaine ;
     reponse   $un       objet }

(de demander (inst)
     (with     ((prompt (val? 'question)))
               (affect 'reponse (read))))

{ extraire-convertir
     sorte-de  =              methode ;
```

```
     nom-fct   $valeur       extraire-convertir ;
     chaine    $un           chaine ;
     position  $un           entier
               $intervalle   [0 +inf] ;
     longueur  $un           entier
               $intervalle   [0 +inf] ;
     valeur    $un           entier }

(de extraire-convertir (inst)
     (let ((valeur 0)
          (code0 (chrnth 0 "0"))
          (chaine (val? 'chaine))
          (position (val? 'position)))
     (affect 'valeur
          (for ( i position 1
               (+ position (sub1 (val? 'longueur))) valeur)
               (setq valeur (+ (* 10 valeur)
               (- (chrnth i chaine) code0)))))))

{ somme
     sorte-de  =         methode ;
     nom-fct   $valeur   somme ;
     op1       $un       nombre ;
     op2       $un       nombre ;
     resultat  $un       nombre }

(de somme (inst)
     (affect 'resultat (+ (val? 'op1) (val? 'op2))))

{ personne
     sorte-de  =    objet ;
     nss       $un  chaine ;
     age       $un  entier
               $sib-exec
                    { difference
                         op1       $valeur   1989 ;
                         op2       $var<-    annee-naissance ;
                         diff      $var->    age }
                    { demander
                         question  $valeur   " Age de la personne ? " ;
                         reponse   $var->    age } ;
     annee-naissance
               $un entier
               $sib-exec
                    { somme
                         op1  $valeur  1900 ;
                         op2  $sib-exec
                              { extraire-convertir
                                   chaine    $var<-    nss ;
                                   position  $valeur   1 ;
                                   longueur  $valeur   2 ;
                                   valeur    $var->    op2 } ;
                         resultat  $var->    annee-naissance } }
```

8.3. Combinaison de moyens d'obtention de valeurs

Les facettes d'obtention de valeurs peuvent bien entendu se combiner. Les attributs d'un attachement procédural peuvent recevoir des valeurs par défaut, résulter d'un filtrage ou d'un attachement procédural, etc. L'obtention d'une valeur d'attribut d'instance peut nécessiter la détermination d'autres valeurs d'attribut, qui elles-mêmes nécessitent d'autres valeurs et ainsi de suite, jusqu'à ce que les valeurs soient trouvées directement

dans les instances. Ce cheminement rappelle le mode de raisonnement dit en "chaînage arrière" des systèmes à règles de production.

Exemple 8.3 :

```
{ point
     sorte-de            =          objet ;
     xcoord              $un        reel
                         $com       "abscisse du point" ;
     ycoord              $un        reel
                         $com "ordonnee du point" }

{ disque
     sorte-de            =               objet ;
     barycentre          $un             point ;
     rayon               $un             reel
                         $intervalle     [ 0.0 +inf ] ;
     surface             $un             reel
                         $sib-exec
                              { pi-r2
                                   r     $var<-    rayon ;
                                   s     $var->    surface } ;
     pts-interieurs      $liste-de       point
                         $com "liste des points connus situes a "
                              "l'interieur du disque"
                         $sib-filtre
                         { point
                              lui-meme
                                   $var-nom  pt
                                   $var->    pts-interieurs
                                   $a-verifier
                                   { inferieur
                                        a     $sib-exec
                                              { c-dist
                                                   pt1  $var<- pt ;
                                                   pt2  $var<- barycentre;
                                                   d    $var-> a } ;
                                        b     $var<-    rayon } } }
```

Le filtre associé à l'attribut **`pts-interieurs`** ramène la liste des points connus, instances de **`point`**, situés à une distance au centre du disque inférieure à son rayon. La distance entre un point filtré et le centre du disque est calculée par l'attachement procédural défini par le schéma **`c-dist`**.

```
{ c-dist
     sorte-de  =              methode ;
     nom-fct   $valeur        c-dist ;
     pt1       $un            point ;
     pt2       $un            point ;
     d         $un            reel
               $intervalle    [ 0.0 +inf ] }

{ inferieur
     sorte-de  =          predicat ;
     nom-fct   $valeur    inferieur ;
     a         $un        reel ;
     b         $un        reel }
```

```
(de c-dist (inst)
     (slet (   (pt1      (val? 'pt1))
               (pt2      (val? 'pt2))
               (x1       (#:shirka:moteur:valeur? 'xcoord pt1) )
               (x2       (#:shirka:moteur:valeur? 'xcoord pt2) )
               (y1       (#:shirka:moteur:valeur? 'ycoord pt1) )
               (y2       (#:shirka:moteur:valeur? 'ycoord pt2) ) )
     (affect   'd        (sqrt (+  (power (- x1 x2) 2)
                                   (power (- y1 y2) 2) ) ) ) ) )

(de inferieur (inst)
     (lt (val? 'a) (val? 'b) ) )

{ pi-r2
     sorte-de  =         methode ;
     nom-fct   $valeur   pi-r2 ;
     r         $un       reel ;
     s         $un       reel }

(de pi-r2 (inst)
          (affect   ( * (val? 'r) (val? 'r) 3.14159 )
                    's ) )
```

9. L'héritage

9.1. Classes et sous-classes

A une classe d'objets peuvent être attachées des sous-classes d'objets plus spécifiques, appelées *spécialisations* de cette classe.

La restriction d'une classe se traduit par l'écriture d'un schéma possédant les mêmes attributs que le schéma de la classe qu'il restreint et de nouveaux attributs propres.

Les attributs existants sont dits *hérités* de la classe supérieure. A son tour, cette classe admet des sous-classes et ainsi de suite. *L'héritage est transitif*. Un schéma de classe est relié au schéma de classe supérieur par l'attribut **`sorte-de`**, déjà rencontré dans les exemples des chapitres précédents. Il existe bien sûr une classe racine, qui n'est la restriction d'aucune autre, c'est la classe **`objet`**. Les classes qui sont des spécialisations directes d'**`objet`** fondent des familles.

Un attribut hérité d'une classe supérieure ne peut être que précisé et non redéfini. Préciser un attribut, ce peut être préciser son type, lui adjoindre de nouvelles restrictions ou de nouveaux moyens de détermination de ses valeurs, voire même fixer ces valeurs.

Il n'est pas possible de préciser le type d'un attribut s'il est simple, à l'exception du type **`nombre`**, pour lequel les types **`reel`** et **`entier`** sont considérés comme des spécialisations.

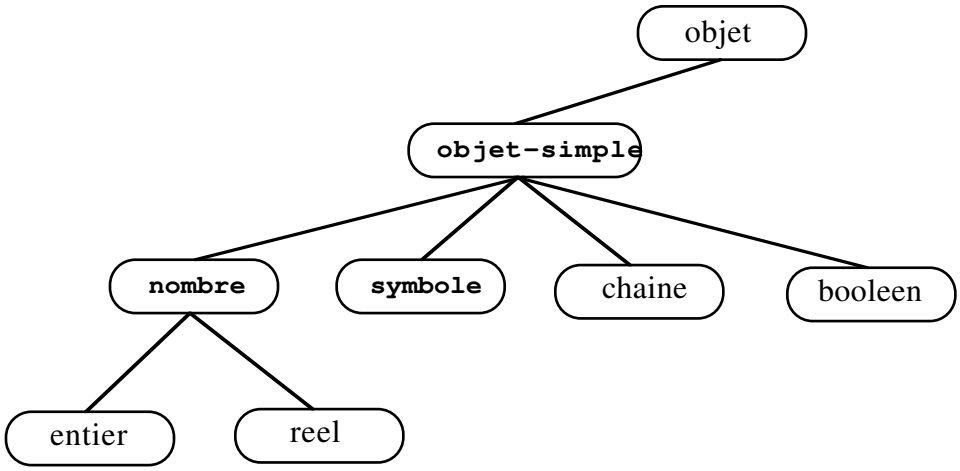


Figure 2 : la famille des objets simples.

Par contre, si le type d'un attribut dans une classe C1 est défini par référence à une autre classe C2, il est possible de restreindre ce type dans une sous-classe C1-1 de C1 en faisant référence, pour cet attribut et au moyen des facettes **`$un`** ou **`$liste-de`**, à une sous-classe de C2.

Exemple 9.1.a :

```
{ point-premier-quadrant
     sorte-de  =    point ;
     xcoord    $intervalle    [ 0.0     +inf ] ;
     ycoord    $intervalle    [ 0.0     +inf ] }
```

point-premier-quadrant définit la classe des points situés dans le premier quadrant du plan. C'est une spécialisation de la classe **point** à laquelle elle est reliée par le lien **sorte-de**.

```
{ disque-premier-quadrant
     sorte-de  =    disque ;
     centre    $un  point-premier-quadrant }
```

disque-premier-quadrant définit la classe des disques dont le centre est situé dans le premier quadrant. Son attribut **centre** est typé **point-premier-quadrant**, alors qu'il était typé **point** dans la classe supérieure **disque**. Les autres attributs, **surface**, **rayon**, **pts-interieurs**, sont automatiquement hérités de **disque**.

Les nouvelles restrictions peuvent également être décrites par les facettes : **$domaine**, **$sauf**, **$intervalle** et **$a-verifier**.

- Le domaine doit être inclus dans le domaine éventuellement hérité.
- Les valeurs placées derrière les facettes **$sauf** se cumulent.
- Les intervalles de la nouvelle classe doivent être compris dans les intervalles hérités.
- Les prédicats placés derrière la facette **$a-verifier** s'ajoutent aux prédicats hérités, si bien que toute valeur de l'attribut d'une instance de la nouvelle classe doit satisfaire les prédicats propres et les prédicats hérités.

En l'absence de nouvelles restrictions, les restrictions héritées s'appliquent directement.

Exemples 9.1.b :

```
{ personne
     sorte-de  =             objet ;
     nom       $un           chaine ;
     age       $un           entier
               $intervalle   [ 0 120 ] }

{ enfant
     sorte-de  =             personne ;
     age       $intervalle   [ 0 15 ] }

{ enfant-67
     est-un    =    enfant ;
     nom       =    "durand" ;
     age       =    12 }

{ date-hiver
     sorte-de  =         date ;
     mois      $domaine  "decembre" "janvier" "fevrier" "mars" }

{ date-hiver-1
     est-un    =    date-hiver ;
     jour      =    15 ;
     mois      =    "fevrier" ;
     annee     =    1986 }
```

Une instance d'une classe donnée est automatiquement considérée comme instance de toutes les classes supérieures. C'est ainsi que **enfant-67** est instance de la classe **enfant** et donc instance de la classe **personne**. L'inverse n'est évidemment pas vrai : une instance de **personne** n'est pas a priori instance d'**enfant**, mais peut le devenir si

certaines conditions sont vérifiées. Ce problème, dit de la spécialisation, sera examiné au §9.3.

La classe d'un attribut est également transmise par héritage et cette classe peut être spécialisée ; il est donc possible de spécifier comme classe d'un attribut dans une classe donnée une spécialisation de la classe de ce même attribut dans une sur-classe.

9.2. Héritage multiple

Une classe peut être définie comme la restriction simultanée de plusieurs classes de la même famille. Elle hérite alors des attributs de toutes ces classes, auxquels elle ajoute ses attributs propres. Les attributs hérités ne peuvent qu'être précisés au niveau de la sous-classe comme dans le cas du mono-héritage.

- Les intervalles spécifiés dans la sous-classe doivent appartenir, pour un attribut donné, à l'*intersection* de tous les intervalles des classes supérieures.
- Les domaines doivent être des sous-domaines de ceux associés à l'attribut dans les classes supérieures.
- Une valeur d'attribut doit vérifier les prédicats propres et tous les prédicats présents dans les classes supérieures. Elle ne peut être aucune des valeurs apparaissant dans les facettes **`$sauf`** attachées à l'attribut dans les classes supérieures.

L'attribut **`sorte-de`** d'un schéma de classe est donc multi-valué.

Exemple 9.2 :

```
{ salarie
     sorte-de           =             personne ;
     age                $intervalle   [ 18 65 ] ;
     revenu-mensuel     $un           entier ;
     activite           $un           activite }

{ etudiant
     sorte-de           =             personne ;
     age                $intervalle   [ 17 77 ] ;
     no-carte           $un           carte ;
     universites        $liste-de     universite }

{ etudiant-salarie
     sorte-de           =    etudiant salarie }

{ duduche
     est-un             =    etudiant-salarie ;
     nom                =    "Duduche" ;
     age                =    20 ;
     no-carte           =    carte-34 ;
     universites        =    grenoble-1 ;
     revenu-mensuel     =    4250 }
```

Dans l'instance **`duduche`**, l'attribut **`nom`** est hérité de **`personne`** ; la valeur de **`age`** est restreinte à l'intervalle [18 65] par les facettes **`$intervalle`** apparaissant dans **`salarie`** et **`etudiant`** ; les attributs **`no-carte`** et **`universites`** sont hérités de **`etudiant`**, **`revenu-mensuel`** de **`salarie`**. La valeur de l'attribut **`activite`** est indéterminée.

La figure 3 représente un exemple de demi-treillis faisant intervenir certaines classes des exemples précédents. Il y apparaît les classes prédéfinies de Shirka : **`objet`**, **`attribut`**, **`facette`** et **`schema`** qui seront explicitées lors de la présentation des méta-

schémas, ainsi que **predicat** et **methode**. Les classes **personne**, **enfant**, **etudiant**, **salarie** et **etudiant-salarie** constituent une famille.

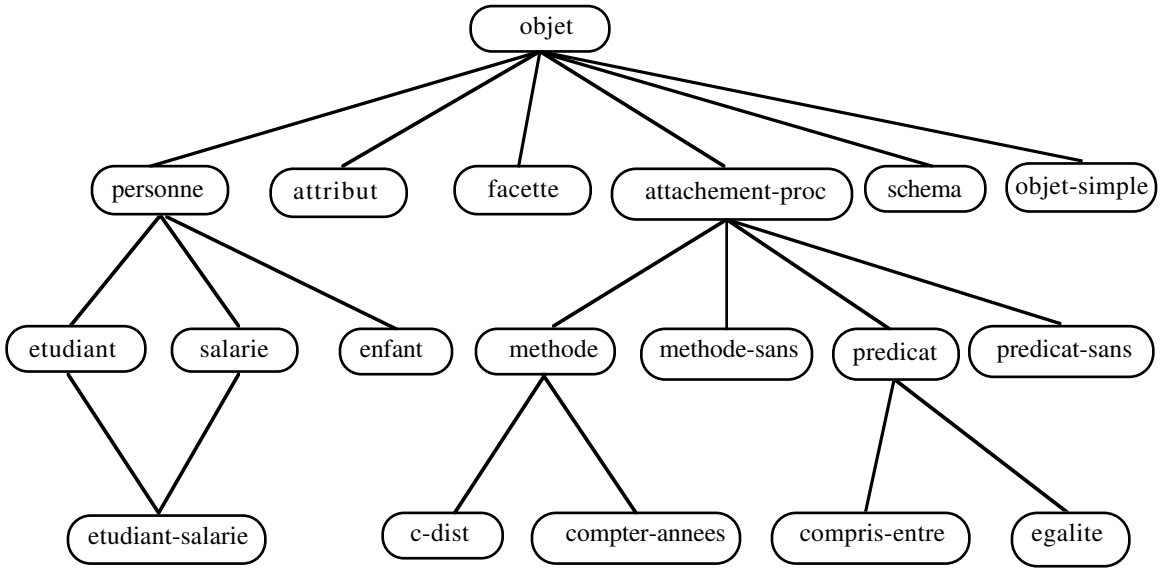


Figure 3 : exemple de treillis de classes.

9.3. La spécialisation

Dans Shirka, tous les attributs sont typés par les facettes **$un** ou **$liste-de**, seules facettes obligatoires à la définition d'un attribut. Une valeur n'est admissible pour un attribut d'une instance que si son type est compatible avec celui de l'attribut. Dans le cas des types simples (**nombre**, **chaine**, **booleen** et **symbole**), il doit y avoir égalité stricte entre le type attendu et le type de la valeur. Dans le cas des types construits, c'est-à-dire définis par référence à une classe, deux cas peuvent se présenter. Dans le premier cas, la valeur est une instance de la classe définissant le type ou de l'une de ses sous-classes, la valeur satisfait alors le type. Dans le deuxième cas, la valeur est une instance d'une des super-classes de la classe définissant le type. Il y alors tentative de spécialisation : cette instance peut-elle être considérée comme instance de la classe du type ?

La réponse à cette question est positive si l'instance peut être considérée comme instance successivement de chacune des classes comprises entre sa classe d'origine et la classe "cible". Pour qu'une instance d'une classe donnée puisse être considérée comme une instance d'une classe immédiatement inférieure, il faut considérer alors chacun des attributs de cette dernière :

- si l'attribut possède une valeur dans l'instance, elle doit satisfaire le type et les restrictions attachés à cet attribut dans la classe inférieure ;
- si l'attribut ne possède pas de valeur, deux cas peuvent se présenter :
  - si de nouvelles restrictions apparaissent dans la classe inférieure, la valeur de l'attribut doit être inférable et satisfaire ces restrictions ;
  - si aucune nouvelle restriction n'apparaît, la valeur peut rester indéterminée ;
- s'il apparaît un attribut propre à la classe inférieure, la valeur de cet attribut doit être inférable et admissible.

Exemple 9.3 :

```
{ pt1
     est-un    =    point ;
     xcoord    =    1.23 ;
     ycoord    =    0.65 }

{ d5
     est-un    =    disque-premier-quadrant ;
     centre    =    pt1 }
```

pt1 est une valeur admissible de l'attribut **centre** de **d5** s'il est possible de le faire passer de sa classe origine **point** à la classe cible **point-premier-quadrant**. De fait, les valeurs de **xcoord** et **ycoord** satisfont les contraintes exprimées par la facette **$intervalle** dans cette dernière classe et la spécialisation réussit.

Le mécanisme de spécialisation peut également être activé par l'utilisateur, grâce à la commande **spec** (2eme partie, au §5.14.1).

10. Détermination de valeurs d'attribut manquantes

Les instances de schémas de classe sont en général incomplètes : certains des attributs, propres ou hérités, n'ont pas reçu de valeur. C'est précisément le rôle des mécanismes d'inférence que de tenter de déterminer ces valeurs inconnues en utilisant les informations contenues dans les schémas de classe dont dépend l'instance.

> Exemple 10a :
>
> Dans l'exemple 3a, l'âge de **personne-23** est inconnu mais sa date de naissance est disponible. Il est alors possible, en utilisant la facette **$sib-exec** de l'attribut **age** de **personne**, de calculer la valeur de cet attribut. Bien entendu, si la date de naissance n'est pas connue, le système essayera de la déterminer. N'ayant aucun moyen d'y réussir, aucune facette d'obtention de la valeur n'apparaissant dans l'attribut **date**, **age** ne pourra pas non plus être calculé. La recherche des enfants de **personne-3** dans l'exemple 8.1.b constitue une autre illustration d'un mécanisme simple d'inférence.

10.1. Héritage des moyens d'obtention

Plus généralement, lors d'une interrogation du type : "Quelle est la valeur (ou les valeurs) de l'attribut **attribut** de l'instance **instance** ?", la valeur est tout d'abord cherchée dans l'instance elle-même. Si elle ne s'y trouve pas, le lien **est-un** est suivi pour retrouver la classe à laquelle cette instance appartient. Dans cette classe, les facettes **$valeur**, **$sib-filtre**, **$sib-exec** et **$defaut** sont examinées dans cet ordre :

- Si la facette **$valeur** est présente, les valeurs qui y apparaissent sont retenues et constituent la réponse à l'interrogation.
- Sinon, si la facette **$sib-filtre** est présente, son premier filtre est essayé. Seules les valeurs satisfaisant toutes les restrictions attachées à l'attribut dans la classe de l'instance sont retenues. Si le nombre de valeurs obtenues est insuffisant par rapport à ce que les facettes **$un**, **$liste-de**, **$cardmin** et **$cardmax** spécifient, le filtre suivant est essayé et ainsi de suite.
- Après les filtres de la facette **$sib-filtre**, les procédures des facettes **$sib-exec**, si elles sont présentes, sont activées. Enfin, **$defaut** peut encore fournir des valeurs.

Arrivé à ce stade, si le nombre de valeurs n'est toujours pas suffisant, le même processus se répète sur les classes immédiatement supérieures. Celles-ci sont explorées de la gauche vers la droite suivant l'ordre précisé dans l'attribut **sorte-de**.

> Exemple 10.1.a :
>
> Dans l'exemple 9.2, le moyen de déterminer l'**age** d'une instance de **etudiant-salarie** sera recherché d'abord dans **etudiant-salarie**, puis dans **etudiant**, puis **personne** et enfin dans **salarie**, pour terminer sur un échec puisqu'aucun moyen d'obtention n'apparaît dans ces classes.

Même si un moyen d'obtention a été trouvé dans une classe supérieure, ce sont les restrictions situées dans la classe de l'instance qui doivent être vérifiées lors des tests d'admissibilité des valeurs.

Enfin, les valeurs obtenues lors d'une interrogation, que celle-ci soit directement le fait de l'utilisateur ou qu'elle résulte d'une action du système, ne sont pas enregistrées dans l'instance. En effet, si c'était le cas, une interrogation ultérieure sur le même attribut de la même instance donnerait ces mêmes valeurs. Or, entre temps, l'état de la base a pu changer, si bien que la réponse à l'interrogation ne devra plus être la même.

Exemple 10.1.b :

Quand la date de référence dans l'exemple 3a change, la valeur à fournir en réponse à la question "âge de la personne **personne-23** ?" doit être recalculée. Si la réponse à cette même question posée antérieurement avait été enregistrée, l'âge aurait été figé dans l'instance **personne-23**.

10.2. Rôle des variables

Lors de l'activation d'un filtre, la distinction entre les facettes **$var<-** et **$var-liste<-** devient fondamentale. Dans la situation suivante :

```
nom-attribut1 $var-nom var1
```

var1 fait référence à une valeur de l'attribut **nom-attribut1**. Si cet attribut est multi-valué, **var1** peut prendre successivement chacune de ces valeurs. Si maintenant **var1** apparaît dans un filtre attaché à un autre attribut, ce filtre pourra être essayé avec chacune des valeurs de **var1** si la facette **$var<-** est utilisée.

```
{ filtre
     nom-attribut2 $var<-    var1 }
```

C'est par ce mécanisme de balayage des listes de valeurs que dans les exemples 8.1.b et 8.1.c, tous les fils et petits-fils de la personne peuvent être retrouvés.

Si la situation se reproduit plusieurs fois dans le même filtre, le nombre d'essais possibles est égal au produit du nombre de valeurs possibles pour chaque variable. La valeur de la première variable par ordre d'apparition dans le filtre est fixée, puis la seconde et les suivantes. Au deuxième essai, c'est la valeur de la dernière variable qui est modifiée. Quand ses valeurs ont toutes été utilisées, la valeur de l'avant-dernière variable est modifiée et la valeur de la dernière est de nouveau la première de la liste des valeurs de l'attribut qu'elle référence. De façon générale, les listes sont parcourues comme dans des boucles imbriquées, les listes associées aux dernières variables du filtre étant parcourues plus vite que celles des variables apparaissant en tête.

Exemple 10.2.a :

```
{ schema1
     a1   $liste-de      entier ;
     a2   $un            chaine ;
     a3   $liste-de      entier ;
     a4   $liste-de      chaine
          $sib-filtre    { schema2
                              b1   $var<-    a1 ;
                              b2   $var<-    a2 ;
                              b3   $var<-    a3 ;
                              b4   $var->    a4 } }
```

```
{ schema2
     b1   $un  entier ;
     b2   $un  chaine ;
     b3   $un  entier ;
     b4   $un  chaine }
```

et soit l'instance incomplète de **schema1** :

```
{ instance1
     est-un    =    schema1 ;
     a1        =    1 2 3 ;
     a2        =    "a" ;
     a3        =    4 5 }
```

dont la valeur de **a4** est à déterminer. Le filtre associé à **a4** dans le schéma **schema1** pourra être essayé 3 * 1 * 2 = 6 fois avec successivement les configurations suivantes :

```
     b1        1    1    2    2    3    3
     b2        "a"  "a"  "a"  "a"  "a"  "a"
     b3        4    5    4    5    4    5
```

Si **var1** apparait dans le filtre dans la facette **$var-liste<-**, elle ne peut prendre qu'une seule valeur, la liste des valeurs de l'attribut **nom-attribut1**.

```
{ filtre
     nom-attribut2 $var-liste<-  var1 }
```

Exemple 10.2.b :

Si dans le schéma **schema1** de l'exemple précédent, le filtre **schema2** s'écrit maintenant :

```
{ schema2
     b1   $var-liste<-  a1 ;
     b2   $var<-        a2 ;
     b3   $var<-        a3 ;
     b4   $var->        a4 }
```

il sera essayé 2 fois avec les valeurs successives suivantes :

```
     b1        ( 1 2 3 )      ( 1 2 3 )
     b2        "a"            "a"
     b3        4              5
```

10.3. Accès aux attributs imbriqués

Il est souvent nécessaire d'accéder à des attributs n'apparaissant pas explicitement dans une description.

Exemple 10.3.a :

```
{ personne
     sorte-de       =    objet ;
     nom            $un  chaine ;
     age            $un  entier ;
     date-naiss     $un  date }
```

Il est possible d'associer à l'attribut **age** une méthode de calcul à partir de la date de naissance, valeur de l'attribut **date-naiss**. Cette méthode sera précisée par une facette **$sib-exec** :

```
{ personne
     sorte-de       =          objet ;
     nom            $un        chaine ;
     age            $un        entier
                    $sib-exec      { compter-annees
                                        date-origine   $var<- date-naiss ;
                                        nb-annees      $var-> age } ;
     date-naiss     $un        date }

{ compter-annees
     sorte-de       =          methode ;
     nom-fct        =          compter-annees ;
     date-origine   $un        date ;
     date-courante  $un        date
                    $defaut    date-87 ;
     nb-annees      $un        entier }

{ date-87
     est-un         =     date ;
     annee          =     1987 }

( de compter-annees ( instance )
     ( affect  nb-annees
          ( - (#:shirka:moteur:valeur? (val?  date-courante)  annee)
          (#:shirka:moteur:valeur?  (val?  date-origine )  annee ))))
```

Les appels imbriqués de **valeur** sont nécessaires pour accéder à l'attribut **annee** des schémas **date** correspondant à la date origine et à la date courante.

La solution de l'exemple consiste donc à résoudre ces références imbriquées à l'intérieur de la fonction réalisant la méthode de calcul. Elle présente plusieurs inconvénients :

- elle réduit la lisibilité de la description,
- elle complique l'écriture de la fonction Le-Lisp,
- elle oblige à particulariser la fonction et elle a donc tendance à augmenter le nombre de schémas méthodes et de fonctions associées.

L'autre solution consiste à définir une spécialisation locale du schéma dont les attributs doivent être référencés. Cette spécialisation s'introduit dans la facette **$un** ou **$liste-de**.

Exemple 10.3.b :

```
{ personne
     sorte-de       =     objet ;
     nom            $un   chaine ;
     age            $un   entier
                    $sib-exec
                         { difference
                              op1       $valeur   1987 ;
                              op2       $var<-    annee-orig ;
                              diff      $var->    age } ;
     date-naiss     $un   { date
                              annee     $var-nom  annee-orig } }
```

```
{ difference
     sorte-de  =    methode ;
     nom-fct   =    difference ;
     op1       $un  entier ;
     op2       $un  entier ;
     diff      $un  entier }
( de difference ( instance )
     ( affect 'diff     (-  (val?'op1)
                        (val? 'op2))))
```

Cette solution permet l'emploi de la méthode **difference**, plus générique.

L'accès à des attributs par le biais de spécialisations locales peut être réalisé également lors d'un filtrage (facette **$sib-filtre**) et lors de tout attachement procédural.

10.4. Accès aux attributs hérités

Les variables ne peuvent faire référence qu'à des valeurs d'attributs syntaxiquement présents dans un schéma. Par conséquent, pour référencer des attributs hérités, il est nécessaire de les faire figurer dans le schéma correspondant. Pour cela, la simple écriture de leur nom suffit.

Exemple 10.4 :

```
{ parallelogramme
     sorte-de       =              figure-plane ;
     angle          $un            reel
                    $com           "angle aigu exprime en degres"
                    $intervalle    [ 0  90 ] ;
     largeur        $un            reel ;
     longueur       $un            reel }

{ losange
     sorte-de       =              parallelogramme ;
     lui-meme       $a-verifier    { egalite
                                        a    $var<-    largeur ;
                                        b    $var<-    longueur } ;
     largeur ;
     longueur }
```

11. Les facettes de réaction sur événement

Toutes les facettes de réaction sur événement réalisent un attachement procédural dont le fonctionnement est conforme à celui de la facette **$sib-exec**, à la différence près que les procédures activées ne doivent pas rendre de résultat. Seuls des effets de bord sont escomptés. Les facettes **$var->** et **$var-liste->** ne peuvent donc pas apparaître dans les filtres de ces facettes. L'appel à la fonction Le-Lisp associée n'est pas précédé d'une phase de filtrage.

Les événements susceptibles de faire appel aux procédures de ces facettes sont le succès ou l'échec de l'obtention d'une valeur pour un attribut d'instance et l'ajout, la modification ou la suppression d'une valeur d'attribut d'instance. Dans ces trois derniers cas, les facettes correspondantes peuvent servir à assurer la cohérence dans et entre les instances.

11.1. Les facettes $si-succes et $si-echec

Les procédures de ces facettes sont appelées respectivement en cas d'échec et de succès lors de l'obtention de la valeur d'un attribut d'une instance en utilisant les informations contenues dans les schémas de classe dominant l'instance. Si la valeur est (*resp*. n'est pas) directement trouvée dans l'instance, la facette **$si-succes** (*resp*. **$si-echec**) n'est pas activée. En cas de succès (*resp*. d'échec) de l'obtention de la valeur au niveau d'un schéma de classe, la recherche de la facette **$si-succes** (*resp*. **$si-echec**) à activer se fait à partir de ce schéma.

La facette **$si-succes** est souvent utilisée pour ranger la valeur obtenue dans l'instance, de sorte que l'interrogation suivante l'y retrouve sans inférence.

Exemple 11.1 :

```
{ personne
     sorte-de       =              objet ;
     lui-meme       $var-nom       lui ;
     nom            $un            chaine ;
     age            $un            entier
                    $sib-exec      { difference
                                        op1       $valeur   1987 ;
                                        op2       $var<-    annee-orig ;
                                        diff      $var->    age } ;
                    $si-succes     { ranger-valeur
                                        instance       $var<-  lui ;
                                        nom-attrib     $valeur age ;
                                        valeur         $var<-  age } ;
     date-naiss     $un            { date
                                        annee     $var-nom annee-orig } }

{ ranger-valeur
     sorte-de       =         methode ;
     nom-fct        $valeur   ranger ;
     instance       $un       objet ;
     nom-attrib     $un       symbole ;
     valeur         $un       objet }
```

```
(de ranger (inst)
     (#:shirka:moteur:aj-valeur
               (#:shirka:noyau:schema  (val? 'instance) )
                    (val? 'nom-attrib)
                    (val? 'valeur)
                    t ))
```

Soit l'instance :

```
{ personne-45
     est-un         =    personne ;
     date-naiss     =    { est-un       =    date ;
                              annee     =    1978 } }
```

La question "age de personne-45?", c'est-à-dire la requête **`val? personne-45 age`** déclenche la méthode **`difference`**. La valeur obtenue est rangée dans l'attribut **`age`** de l'instance **`personne-45`** par suite de l'activation de la méthode **`ranger-valeur`** de la facette **`$si-succes`**. Si la question est posée une seconde fois, la même réponse est évidemment obtenue, mais sans faire appel à la méthode **`difference`** puisque la valeur est maintenant directement trouvée dans l'attribut **`age`** de l'instance.

La facette **`$si-succes`** n'est pas non plus activée. Il est clair que la réponse à cette question ne changerait pas si la valeur de la date de naissance était modifiée. Par contre, en l'absence de la facette **`$si-succes`**, la valeur n'ayant pas été rangée dans l'attribut **`age`** de l'instance, des questions successives entraîneraient à chaque fois le calcul de l'âge. La valeur de celui-ci tiendrait alors compte des modifications éventuelles de la date de naissance.

11.2. Les facettes $avant-ajout et $apres-ajout

Ces facettes sont activées lors de l'ajout d'une valeur à un attribut d'instance, que cet ajout soit demandé par l'utilisateur par la commande **`aj-val`** ou déclenché par une méthode faisant appel à la fonction prédéfinie **`aj-val`** dans un attachement procédural. Si l'attribut est mono-valué (**`$un`**), l'ajout consiste à lui donner une valeur. La valeur ajoutée doit être admissible, c'est-à-dire satisfaire toutes les contraintes et restrictions attachées à l'attribut. De plus, dans le cas d'un attribut multi-valué, la nouvelle liste de valeurs doit elle-même être admissible, en particulier le nombre de ses éléments doit respecter la contrainte éventuellement introduite par une facette **`$cardmax`**.

La facette **`$avant-ajout`** est activée avant l'opération d'ajout, si celle-ci est possible. La ou les méthodes qui lui sont associées feront donc référence, par une variable, à la valeur de l'attribut avant l'ajout.

La facette **`$apres-ajout`** est activée après l'opération d'ajout, si celle-ci a été effective, et les méthodes associées feront référence à la valeur complétée par l'opération d'ajout. La facette n'est pas activée si l'opération d'ajout échoue parce que la valeur ajoutée n'est pas admissible.

11.3. Les facettes $avant-mod et $apres-mod

Elles sont activées en cas de modification, dans une instance, de la valeur de l'attribut auquel elles sont associées. La modification peut être demandée par l'utilisateur en appelant la commande **mod-val** ou bien être déclenchée par une méthode faisant appel à la fonction pré-définie **mod-val** dans un attachement procédural. Dans le cas d'un attribut multi-valué, il faut préciser, outre la nouvelle valeur, la valeur de la liste à remplacer.

La valeur de l'attribut peut être simple ou faire référence à une instance. Dans ce dernier cas, les deux facettes de réaction sur modification sont activées soit si la référence est modifiée, soit si une modification est apportée à l'instance référencée, c'est-à-dire si la valeur de l'un de ses attributs est modifiée. Symétriquement, lorsqu'une instance est modifiée, Shirka examine toutes les instances qui y font référence par un attribut. Si cet attribut possède au niveau de la classe une facette **$avant-mod** ou **$apres-mod**, les procédures correspondantes sont appelées.

La facette **$avant-mod** est activée avant que la modification ne soit effectuée, donc si l'une des méthodes qui lui est attachée fait référence à la valeur de l'attribut, elle travaillera sur la valeur non modifiée.

La facette **$apres-mod** est activée une fois la modification faite et suppose que celle-ci était admissible.

11.4. Les facettes $avant-sup et $apres-sup

Elles sont respectivement appelées avant et après la suppression d'une valeur dans un attribut d'instance. Pour un attribut multi-valué, la valeur ne peut être supprimée que si la liste des valeurs après suppression respecte les contraintes attachées à l'attribut, en particulier le nombre minimum d'éléments, éventuellement fixé par une facette **$cardmin**.

Si la valeur de l'attribut est une référence à une instance, les facettes de réaction sur suppression sont activées soit si la référence est supprimée, soit si l'instance référencée est détruite. Symétriquement, lors de la suppression d'une instance, Shirka examine toutes les instances qui y faisaient référence par un attribut. Si cet attribut possède au niveau de la classe une facette **$avant-sup** ou **$apres-sup**, les procédures correspondantes sont activées.

12. Visualisation des instances

Les instances possèdent une représentation externe par défaut. Cette représentation externe peut être redéfinie, totalement ou partiellement, par le concepteur de la base de connaissances, éventuellement classe par classe. Il est ainsi possible de préciser dans un schéma de classe la manière de passer, pour ses instances, d'une représentation interne à la représentation externe en utilisant les facettes de visualisation.

En l'absence de facettes spécifiques, une instance est visualisée sur l'écran, ou dans un fichier, suivant un format standard à la demande de l'utilisateur par la commande **vi**. Cette commande permet de spécifier le nombre de niveaux de visualisation requis. Les facettes **$avant-vi**, **$pour-vi**, **$apres-vi** et **$en-vi** permettent de moduler ce format standard en précisant, pour chaque attribut, des méthodes de représentation externe de sa ou ses valeurs.

La visualisation d'une instance se fait donc *attribut par attribut* dans l'ordre issu de la description des schémas de classe dominant l'instance. Par défaut, les conventions appliquées correspondent au format d'entrée utilisé dans ce manuel et accepté par la commande **charger** :

- une instance est encadrée par les accolades **{** et **}** ;
- les attributs sont visualisés séquentiellement par leur nom suivi du signe **=**, de leur(s) valeur(s) et d'un point-virgule **;**. Seuls les attributs définis par le concepteur et l'attribut **est-un** sont représentés ;
- si la valeur d'un attribut est indéterminée dans l'instance et n'est pas inférable dans le cas où le booléen correspondant de la commande **vi** est vrai, il n'apparaît pas ;
- si la valeur d'un attribut est d'un type simple, le format de visualisation correspond au format Le-Lisp ;
- si la valeur d'un attribut est une instance, celle-ci est visualisée encadrée par les accolades **{** et **}** ;
- si la valeur d'un attribut est une référence à une instance, le nom de cette dernière est visualisé. Cependant, si le nombre de niveaux de visualisation demandé le permet, c'est l'instance référencée elle-même qui est visualisée entre **{ }**.

Exemple 12a :

Visualisation de l'instance **duduche** de la classe **etudiant-salarie** :

```
{ duduche
     est-un          =    etudiant-salarie ;
     nom             =    "Duduche" ;
     age             =    20 ;
     no-carte        =    carte-34 ;
     universites     =    grenoble-1 ;
     salaire         =    4250 }
```

Visualisation de l'instance **personne-23** de la classe **personne** au niveau 0 :

```
{ personne-23
     est-un               =    personne ;
     nom                  =    "dupond"
     prenoms              =    "pierre" "paul" ;
     sexe-f               =    faux ;
```

```
     date-de-naissance  =    date-8 ;
     enfants            =    personne-4 personne-56 ;
     adresse            =    { est-un          =    adresse ;
                               numero         =    12 ;
                               rue            =    "de la poste" ;
                               localite       =    "Trifouillis" } }
```

au niveau 1 :

```
{ personne-23
     est-un             =    personne ;
     nom                =    "dupond"
     prenoms            =    "pierre" "paul" ;
     sexe-f             =    faux ;
     date-de-naissance  =    { date-8
                               est-un    =    date ;
                               jour      =    22 ;
                               mois      =    "fevrier" ;
                               annee     =    1952 } ;
     enfants            =    { personne-4
                               est-un             =    personne ;
                               prenoms            =    "pierre" ;
                               sexe-f             =    faux ;
                               age                =    6 ;
                               date-de-naissance  =    date-9 }
                             { personne-56
                               est-un             =    personne ;
                               prenoms            =    "lucie" ;
                               sexe-f             =    vrai ;
                               date-de-naissance  =    date-34 }
     adresse            =    { est-un      =    adresse ;
                               numero    =    12 ;
                               rue       =    "de la poste" ;
                               localite  =    "Trifouillis" } }
```

Au niveau 2, les instances **`date-9`** et **`date-34`** seront à leur tour détaillées.

La facette **`$pour-vi`** permet d'associer à un attribut une ou plusieurs méthodes de visualisation qui se substituent à la méthode par défaut de Shirka. Il est par conséquent possible de visualiser des valeurs de type simple, suivant un autre format que celui de Le-Lisp, ou encore de visualiser une instance, valeur d'attribut, suivant un format tout à fait spécifique.

La facette **`$avant-vi`** permet de spécifier des actions de visualisation devant se dérouler *avant* la méthode de visualisation proprement dite de la valeur de l'attribut. Par exemple, elle permet de redéfinir le caractère **`{`** de début d'instance, mais aussi de commander des passages à la ligne et des indentations.

La facette **`$apres-vi`** permet symétriquement de spécifier des actions de visualisation devant se dérouler *après* la méthode de visualisation proprement dite.

La facette **`$en-vi`** permet enfin de spécifier des actions exécutées *en cours* de visualisation d'une liste de valeurs d'un attribut multi-valué. Elle permet par exemple d'introduire un séparateur de liste autre que le caractère blanc.

Exemple 12.b :

```
{ boucle-pour
     sorte-de       =          instruction ;
     lui-meme       $a-verifier
                         { boucle-conforme
                              val-initiale   $var<-    val-initiale ;
                              increment      $var<-    increment ;
                              val-finale     $var<-    val-finale } ;
     index          $un             symbole ;
     val-initiale   $un             entier ;
     increment      $un             entier ;
     val-finale     $un             entier ;
     instructions   $liste-de       instruction }

{ boucle-conforme
     sorte-de       =          predicat ;
     nom-fct        $valeur    bcle-cf ;
     val-initiale   $un        entier ;
     increment      $un        entier ;
     val-finale     $un        entier }

(de  bcle-cf (inst)
     (>=       (*   (val? 'increment)
                    (-   (val? 'val-initiale)
                         (val? 'val-finale)))
               0))

{ boucle-pour-Pascal
     sorte-de       =               boucle-pour ;
     index          $avant-vi       { v-for }
                    $apres-vi       { v-:= } ;
     increment      $avant-vi       { v-by } ;
     val-finale     $avant-vi       { v-to
                                         incr      $var<-    increment }
                    $apres-vi       { v-begin } ;
     instructions   $en-vi          { v-; }
                    $apres-vi       { v-end } }

{ v-to
     sorte-de  =          methode ;
     nom-fct   $valeur    v-to ;
     incr      $un        entier }

(de v-to (inst)
          (prin     (if (>= (val? 'incr) 0)
                    " to "
                    " downto ")))
```

Les autres méthodes **v-for**, **v-:=**, **v-by**, **v-;** , **v-begin** et **v-end** ; impriment simplement une chaîne de caractères, respectivement **for** , **:=** , **by** , **;** , **begin** et **end**. Une autre solution peut consister à ne définir qu'une seule méthode d'impression, à laquelle les chaînes sont passées en argument.

L'instance suivante :

```
{ boucle-pour-12
     est-un         =    boucle-pour ;
     index          =    i ;
     val-initiale   =    2 ;
     increment      =    -3 ;
     val-finale     =    -8 }
```

sera visualisée telle quelle, mais si elle était instance de **boucle-pour-Pascal**, elle serait visualisée ainsi :

```
for i:= 2 by -3 downto -8 begin end
```

Bien entendu, les exemples donnés ci-dessus sont élémentaires. Rien n'empêche de programmer des méthodes de visualisation tirant parti des fonctions de gestion du terminal virtuel de Le-Lisp, ou des fonctions graphiques élaborées de la machine cible (Macintosh, …).

13. Le méta-niveau

Dans tous les chapitres précédents, la distinction entre schémas de classe et schémas d'instance est omniprésente. En fait, dans Shirka, un schéma de classe est lui-même une instance d'un "super-schéma" de classe, encore appelé *méta-schéma*.

13.1. Les méta-schémas

En tant que schémas de classe, les méta-schémas sont organisés en famille, dont l'élément supérieur est le schéma **schema**, seul schéma à être instance de lui-même. Généralement, un schéma de classe introduit par le concepteur d'une base est défini comme une instance de **schema**. Il y apparaît donc simultanément l'attribut **sorte-de** et implicitement l'attribut **est-un** dont la valeur est dans ce cas **schema**. Il est aussi possible de définir de nouveaux méta-schémas, spécialisations de **schema**.

La classe **schema** est ainsi définie :

```
{ schema
     est-un          =    schema ;
     nom             =    schema ;
     sorte-de        =    objet ;
     list-att        =    a-sorte-de a-liste-att a-spec a-inst ;
     inst            =    schema objet attribut facette
                          attachement-proc methode methode-sans
                          predicat predicat-sans }
```

où **a-sorte-de**, **a-liste-att**, **a-spec** et **a-inst** sont des instances de la classe **attribut** décrivant les attributs **sorte-de**, **list-att** (liste des attributs), **spec** (liste des spécialisations) et **inst** (liste des instances). Initialement, l'attribut **spec** ne possède pas de valeur. Pour bien comprendre cette définition, il faut remarquer que **schema** est instance de lui-même et spécialisation d'**objet**, dont il hérite les attributs **est-un**, **util-par** et **nom**. Initialement, l'attribut **util-par** ne possède pas de valeur.

Lors de l'instanciation d'un méta-schéma, **schema** ou l'une de ses spécialisations, les différents attributs reçoivent une valeur, y compris l'attribut **list-att**, dont la valeur est la liste des attributs de la classe ainsi définie. Pour des raisons de lisibilité, les éléments de cette liste apparaissent directement dans la représentation externe des schémas de classe utilisée dans ce manuel et non pas sous la forme :

```
list-att =    attribut-1 attribut-2 . . . attribut-n ;
```

comme pour les autres attributs. Dans cette représentation externe, la classe **schema** s'écrit ainsi :

```
{ schema
     sorte-de        =              objet ;
     sorte-de        $liste-de      schema ;
     list-att        $liste-de      attribut ;
     spec            $liste-de      schema ;
     inst            $liste-de      objet }
```

13.2. Le schéma objet

Pour les schémas de classe, le schéma situé au sommet du treillis est pré-défini dans Shirka et s'appelle **objet**. C'est une instance du méta-schéma **schema**. Initialement, sa description est la suivante :

```
{ objet
     est-un    =    schema ;
     nom       =    objet ;
     spec      =    objet-simple schema attribut facette attachement-proc
                    intervalle erreur ;
     list-att  =    a-est-un a-util-par a-nom-inst }
```

Les attributs **util-par**, **inst** et **sorte-de** n'y ont pas de valeur. La classe **objet** possède initialement sept spécialisations qui seront précisées plus loin. Les instances d'**objet** ne comportent que trois attributs : **est-un**, **util-par** et **nom**. L'attribut **util-par** permet de relier une instance à toutes les instances qui y font référence par un attribut. Sa valeur est exploitée lors de la propagation des modifications et des suppressions provoquant l'activation des facettes correspondantes de réaction sur événement (cf. §11.3 et §11.4). Les spécialisations d'**objet** apportent leurs attributs propres qui s'ajoutent à ces trois premiers, présents par héritage dans tous les schémas de classe.

13.3. Le schéma attribut et ses spécialisations

Les attributs et les facettes sont des instances respectivement des classes **attribut** et **facette**, elles-mêmes spécialisations de la classe **objet**.

attribut possède sept attributs propres :

- son nom **nom-att** ,
- la liste de ses facettes **liste-fac**,
- sa **nature** (mono ou multi-valué),
- son **type**,
- la donnée de son **domaine**,
- l'**intervalle** de valeurs admissibles,
- son domaine de valeurs non admissibles (**sauf**).

Les cinq derniers attributs permettent de mémoriser dans un attribut particulier, instance d'**attribut**, des informations qui théoriquement devraient être déterminées chaque fois qu'elles sont requises.

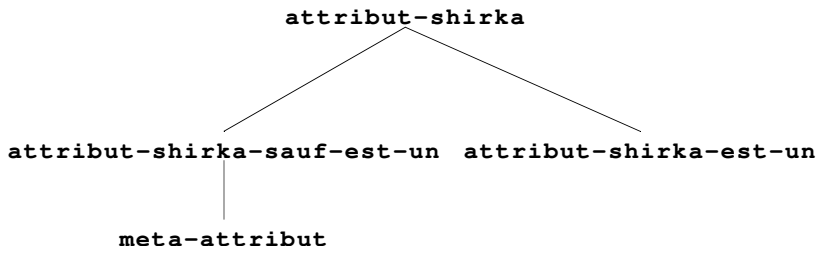
```
{ attribut
     sorte-de       =              objet ;
     nom-att        $un            symbole ;
     list-fac       $liste-de      facette ;
     nature         $un            symbole
                    $domaine       $un $liste-de ;
     type           $un            symbole ;
     domaine        $liste-de      objet ;
     intervalle     $liste-de      intervalle ;
     sauf           $liste-de      objet }

{ intervalle
     sorte-de       =    objet ;
     borne-inf      $un  objet-simple ;
     borne-sup      $un  objet-simple }
```

A titre d'exemple, l'attribut **a-sorte-de** est défini par l'instance suivante :

```
{ a-sorte-de
    est-un          =    attribut ;
    nom-att         =    sorte-de ;
    list-fac        =    {    est-un         =    $un ;
                              list-val       =    schema } }
```

Le schéma **attribut** possède des spécialisations, et l'utilisateur peut en créer de nouvelles en vue de préciser la classe d'un attribut dans un schéma de classe de sa base (cf. exemple §1.2).

L'arbre suivant présente l'organisation des classes d'attribut prédéfinies:



attribut-shirka regroupe les attributs **est-un**, **util-par** et **nom-inst**
attribut-shirka-sauf-est-un regroupe les attributs **util-par** et **nom-inst**
attribut-shirka-est-un ne contient que l'attribut **est-un**
meta-attribut regroupe les attributs **list-att**, **spec**, **inst** et **sorte-de**

Pour définir une nouvelle classe d'attribut, il suffit d'en donner la définition sous forme d'un schéma de classe, par exemple au début du fichier contenant la base de connaissances, et en tous cas avant l'utilisation du nom de cette classe devant un nom d'attribut.

13.4. Le schéma facette et ses spécialisations

La classe **facette** possède deux attributs propres :

- le nom **nom-fac**,
- la liste de ses valeurs **liste-val**.

```
{ facette
    sorte-de        =              objet ;
    nom-fac         $un            symbole ;
    list-val        $liste-de      objet }
```

La figure 4 montre les différentes spécialisations de la classe **facette**. Au premier niveau, les facettes sont distinguées suivant qu'elles admettent une seule valeur, plusieurs valeurs et une seule ou plusieurs, suivant les cas d'utilisation. Les spécialisations de la classe **f-fil-att** définissent toutes les facettes de filtrage et d'attachement procédural, y compris les facettes de visualisation.

Il est ainsi facile de créer une nouvelle facette dans Shirka. Il suffit d'ajouter sa définition sous forme de schéma dans le fichier **Installer-Shirka.1**, puis de relancer l'installation de Shirka. L'interprétation de cette nouvelle facette est bien entendu à la charge de fonctions spécifiques, à développer en faisant appel aux fonctions Shirka:
val-facette et **val-fac-he**, décrites en §6.16 et §6.17.

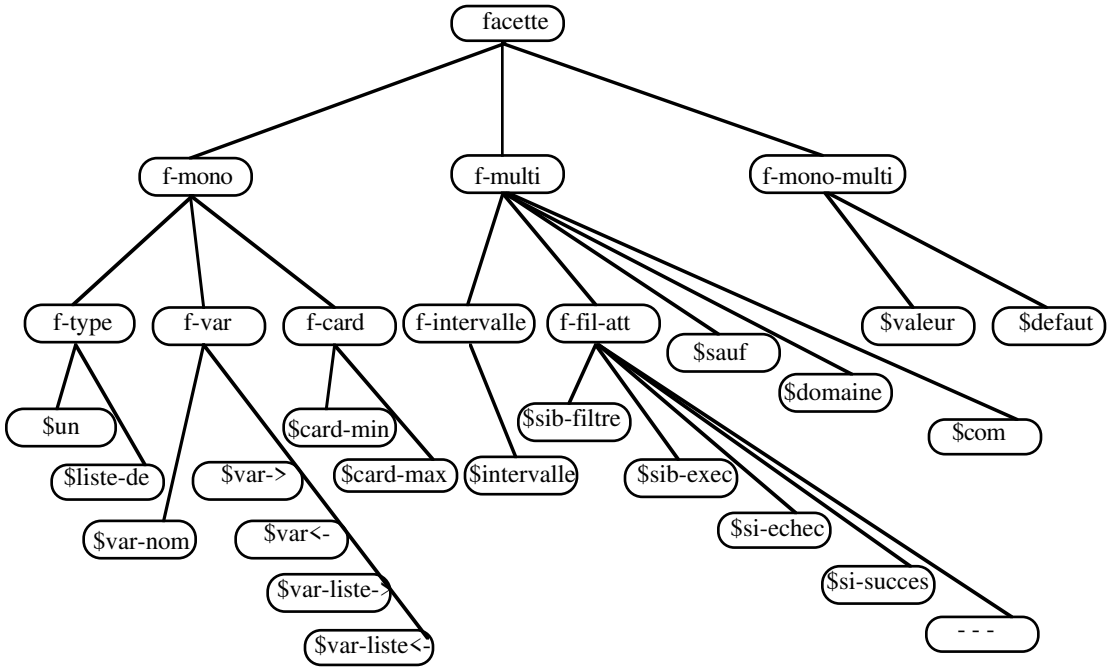


Figure 4 : la famille facette

Toute facette particulière, telle que **$un**, **$sib-exec**, **$com** ou **$intervalle**, est définie comme une classe terminale de la famille facette.

```
{ $un
     sorte-de       =          f-type ;
     nom-fac        $valeur    $un ;
     list-val       $un        objet }

{ $sib-exec
     sorte-de       =          f-fil-att ;
     nom-fac        $valeur    $sib-exec }

{ $com
     sorte-de       =               f-multi ;
     nom-fac        $valeur         $com ;
     list-val       $liste-de       chaine }

{ $intervalle
     sorte-de       =               f-intervalle ;
     nom-fac        $valeur         $intervalle ;
     list-val       $liste-de       intervalle }
```

13.5. Les schémas méthode et prédicat

La classe **attachement-proc** est également prédéfinie en tant que spécialisation d'**objet**. Elle possède un seul attribut propre **nom-fct**, dont la valeur précise le nom de la fonction Le-Lisp associée. Ses spécialisations sont **methode** et **predicat**, avec ou sans filtrage. Tous les prédicats et méthodes introduits par le concepteur d'une base doivent donc être définis en tant que spécialisations de ces classes.

```
{ attachement-proc
     sorte-de       =    objet ;
     nom-fct        $un  symbole }
```

```
{ methode
    sorte-de       =    attachement-proc }

{ methode-sans
    sorte-de       =    attachement-proc }

{ predicat
    sorte-de       =    attachement-proc }

{ predicat-sans
    sorte-de       =    attachement-proc }
```

La figure 5 résume les relations qu'entretiennent méta-schémas, schémas de classe et schémas d'instances. Les schémas d'un même plan sont de même nature. Les liens entre plans sont des liens **`est-un`** d'instance à classe. Par exemple, **`duduche`** est une instance d'**`etudiant-salarie`**, **`objet`** est instance de **`schema`**. Les liens en gras dans un même plan sont des liens **`sorte-de`** de classe à sous-classe. Par exemple, **`personne`** est une sous classe d'**`objet`**. A remarquer : **`schema`** est instance de lui-même et **`sorte-de objet`** ; **`objet`** est bien sûr instance de **`schema`**.

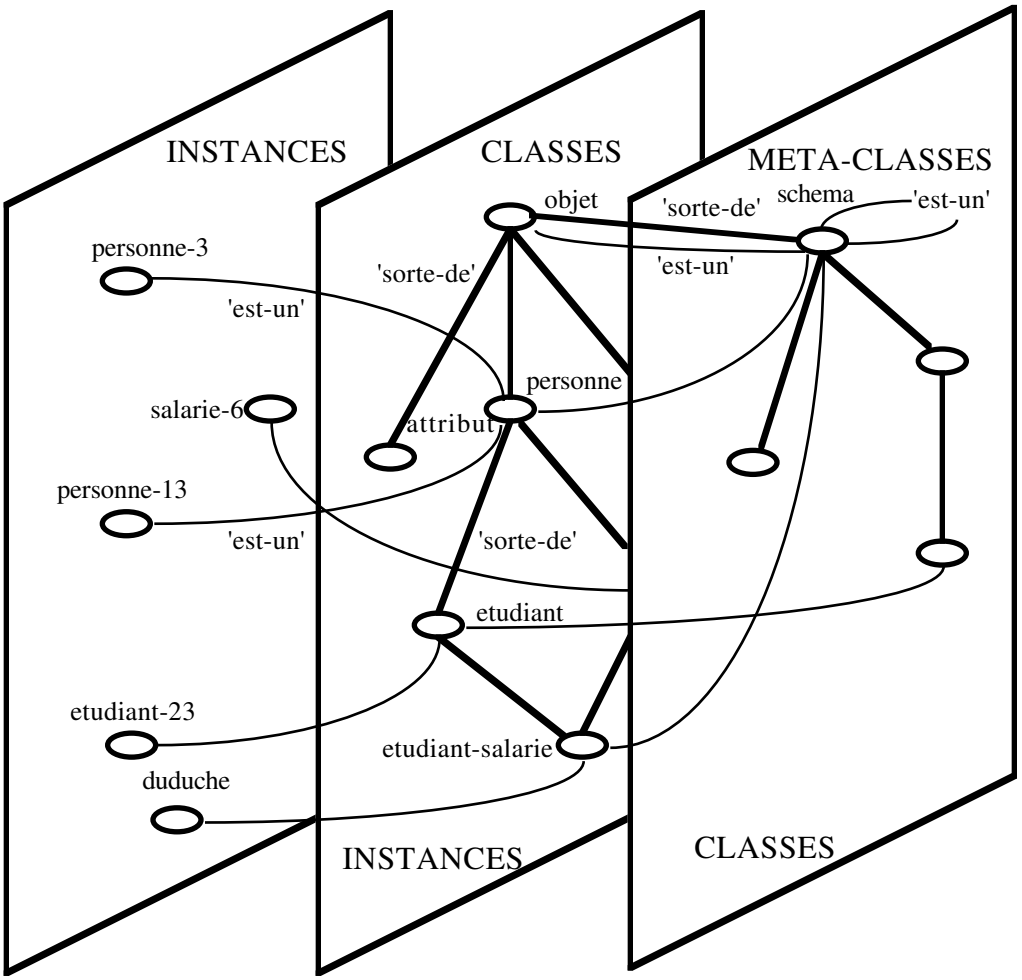


Figure 5 : instances, classes et méta-classes.

13.6. Utilisation des méta-schémas

L'introduction de méta-schémas présente de nombreux avantages pour le concepteur d'une base de connaissances Shirka. Puisque tout schéma de classe est en fait une instance de **schema** ou de l'une de ses spécialisations, tout ce qui a été dit jusqu'ici sur les instances est applicable aux classes. Il est ainsi intéressant de définir des spécialisations du méta-schéma **schema** afin, par exemple, de :

- définir une représentation externe des schémas de classe spécifique, différente de la représentation utilisée dans ce manuel ;
- grouper des schémas de classe comme instances d'un même méta-schéma particulier de façon à appliquer efficacement sur ces classes des filtres en tous points semblables aux filtres présentés dans ce manuel, sur des instances simples.

14. Un exemple : la base “Objets-du-plan”

14.1. Les méthodes, les prédicats et les fonctions Le-Lisp associées

```
{ c-pente
     sorte-de  =         methode ;
     nom-fct   $valeur   c-pente ;
     x-pt1     $un       reel ;
     y-pt1     $un       reel ;
     x-pt2     $un       reel ;
     y-pt2     $un       reel ;
     p         $un       reel }

(de c-pente (inst)
     ( affect 'p ( divide
                   ( - (val? 'y-pt2)  (val? 'y-pt1) )
                   ( - (val? 'x-pt2)  (val? 'x-pt1) ) ) ) )


{ c-dist
     sorte-de  =         methode ;
     nom-fct   $valeur   c-dist ;
     pt1       $un       point ;
     pt2       $un       point ;
     d         $un       reel }

(de c-dist (inst)
     (slet (   (pt1      (val? 'pt1))
               (pt2      (val? 'pt2))
               (x1       (#:shirka:moteur:valeur? 'xcoord pt1) )
               (x2       (#:shirka:moteur:valeur? 'xcoord pt2) )
               (y1       (#:shirka:moteur:valeur? 'ycoord pt1) )
               (y2       (#:shirka:moteur:valeur? 'ycoord pt2) ) )
     (affect   'd        (sqrt (+  (power (- x1 x2) 2)
                                   (power (- y1 y2) 2) ) ) ) ) )


{ pi-r2
     sorte-de  =         methode ;
     nom-fct   $valeur   pi-r2 ;
     r         $un       reel ;
     s         $un       reel }

(de pi-r2      (inst)
               (affect 's ( * (val? 'r) (val? 'r) 3.14159 ) ) )


{ milieu-segment
     sorte-de  =         methode ;
     nom-fct   $valeur   milieu-segment ;
     a         $un       point ;
     b         $un       point ;
     m         $un       point
               $com      "milieu du segment [a b] " }
```

```
(de milieu-segment (inst)
     (affect 'm (  #:shirka:moteur:creer-instance
                   'point
                   '( (xcoord
                        (/ ( +    ( #:shirka:moteur:valeur? 'xcoord
                                    (val? 'a) )
                                    ( #:shirka:moteur:valeur? 'xcoord
                                    (val? 'b) ) )
                        2.0) )
                        (ycoord
                        (/ ( +    ( #:shirka:moteur:valeur? 'ycoord
                                    (val? 'a) )
                                    ( #:shirka:moteur:valeur? 'ycoord
                                    (val? 'b) ) )
                        2.0) ) )
                   nil) ) )


{ produit
     sorte-de  =         methode ;
     nom-fct   $valeur   produit ;
     a         $un       reel ;
     b         $un       reel ;
     p         $un       reel }

(de produit (inst)
     (affect 'p ( * (val? 'a) (val? 'b) ) ) )


{ egalite
     sorte-de       =         predicat ;
     nom-fct        $valeur   egalite ;
     a              $un       reel ;
     b              $un       reel ;
     precision      $un       reel
                    $defaut   1.0e-8 }

(de egalite (inst)
     (<=  (abs (-   (val? 'a)
                    (val? 'b) ) )
          (val? 'precision) ) )


{ inferieur
     sorte-de  =         predicat ;
     nom-fct   $valeur   inferieur ;
     a         $un       reel ;
     b         $un       reel }

(de inferieur (inst)
     ( <= (val? 'a)  (val? 'b)))


{ apparait
     sorte-de  =              predicat ;
     nom-fct   $valeur        apparait ;
     element   $un            objet ;
     liste     $liste-de      objet }

(de apparait (inst)
     (member (val? 'element) (list (val? 'liste) ) ) )
```

14.2. Les classes d'objets du plan

```
{ objet-du-plan
     sorte-de  =    objet }

{ point
     sorte-de  =         objet-du-plan ;
     xcoord    $un       reel
               $com      "abscisse du point" ;
     ycoord    $un       reel
               $com      "ordonnee du point" }

{ point-premier-quadrant
     sorte-de  =    point ;
     xcoord    $intervalle    [ 0.0 +inf ] ;
     ycoord    $intervalle    [ 0.0 +inf ] }

{ figure-plane
     sorte-de       =              objet-du-plan ;
     surface        $un            reel ;
     barycentre     $un            point ;
     contient       $liste-de      figure-plane }

{ segment
     sorte-de       =    figure-plane ;
     origine        $un  { point
                              xcoord    $var-nom  x-orig ;
                              ycoord    $var-nom  y-orig } ;
     extremite      $un  { point
                              xcoord    $var-nom  x-extr ;
                              ycoord    $var-nom  y-extr } ;
     surface        $valeur   0.0 ;
     barycentre     $sib-exec
                         { milieu-segment
                              a         $var<-    origine ;
                              b         $var<-    extremite ;
                              m         $var->    barycentre } ;
     pente          $un       reel
                    $sib-exec
                         { c-pente
                              x-pt1     $var<-    x-orig ;
                              y-pt1     $var<-    y-orig ;
                              x-pt2     $var<-    x-extr ;
                              y-pt2     $var<-    y-extr ;
                              p         $var->    pente } ;
     longueur       $un  reel
                    $sib-exec
                         { c-dist
                              pt1       $var<-    origine ;
                              pt2       $var<-    extremite ;
                              d         $var->    longueur } }
```

```
{ disque
    sorte-de       =              figure-plane ;
    surface        $sib-exec
                        { pi-r2
                             r     $var<-    rayon ;
                             s     $var->    surface } ;
    barycentre     $var-nom       centre ;
    rayon          $un            reel
                   $intervalle    [ 0.0   +inf ] ;
    pts-interieurs      $liste-de      point
                   $sib-filtre
                        { point
                             lui-meme
                                  $var-nom  pt-c
                                  $var->    pts-interieurs
                                  $a-verifier
                                  { inferieur
                                       a     $sib-exec
                                             { c-dist
                                                  pt1 $var<- centre;
                                                  pt2 $var<- pt-c ;
                                                  d     $var-> a } ;
                                       b     $var<-    rayon } } }

{ disque-premier-quadrant
    sorte-de       =    disque ;
    barycentre     $un  point-premier-quadrant }

{ parallelogramme
    sorte-de  =              figure-plane ;
    angle     $un            reel
              $com           "angle aigu des cotes exprime en degres"
              $intervalle    [ 0  90 ] ;
    largeur   $un            reel ;
    longueur  $un            reel }

{ rectangle
    sorte-de  =              parallelogramme ;
    angle     $valeur        90 ;
    surface   $sib-exec
                   { produit
                        a     $var<-    largeur ;
                        b     $var<-    longueur ;
                        p     $var->    surface } ;
    largeur ;
    longueur }

{ losange
    sorte-de  =              parallelogramme ;
    lui-meme  $a-verifier
                   { egalite
                        a     $var<-    largeur ;
                        b     $var<-    longueur } ;
    largeur ;
    longueur }

{ carre
    sorte-de  =    rectangle      losange }
```

```
{ figure-coloree
     sorte-de  =              figure-plane ;
     lui-meme  $var-nom       fg-c ;
     couleur   $un            symbole
               $var-nom       couleur-fg-c
               $domaine       vert  jaune  rouge  bleu  noir
               $defaut        noir
               $sib-filtre
                    { figure-coloree
                         lui-meme
                              $a-verifier
                                   { apparait
                                        element   $var<- fg-c ;
                                        liste     $var<- contient } ;
                         contient ;
                         couleur   $var->    couleur-fg-c } }

{ carre-colore
     sorte-de  =         carre     figure-coloree }

{ carre-rouge
     sorte-de  =         carre-colore ;
     couleur   $valeur   rouge }
```

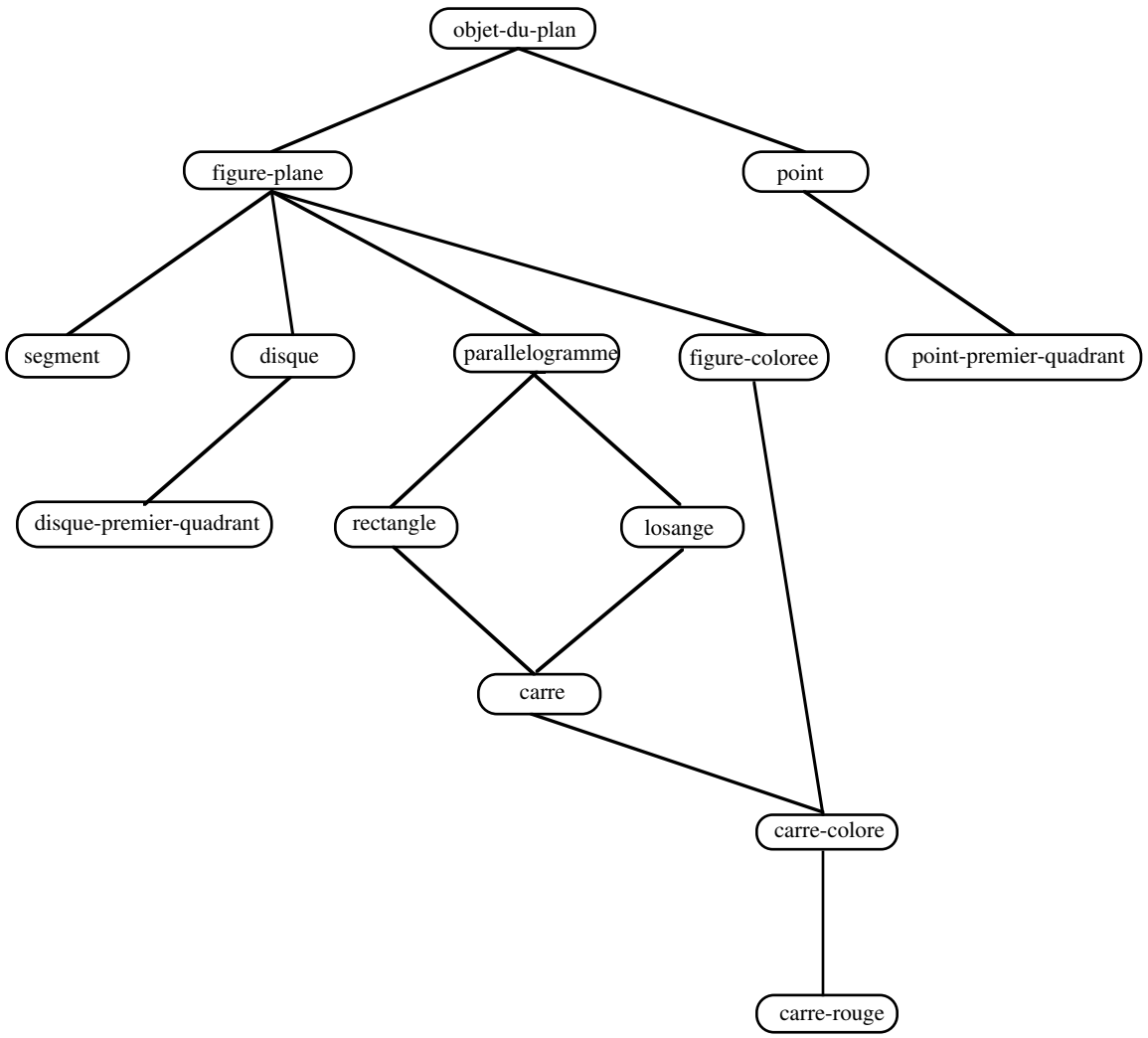


Figure 5 : treillis des classes de la base "Objets-du-plan".

14.3. Les instances

```
{ pt1
     est-un    =    point ;
     xcoord    =    1.5 ;
     ycoord    =    2.34 }

{ pt2
     est-un    =    point ;
     xcoord    =    4.75 ;
     ycoord    =    -2.8 }

{ pt3
     est-un    =    point ;
     xcoord    =     5.5 ;
     ycoord    =     3.0 }

{ pt4
     est-un    =    point ;
     xcoord    =     4.0 ;
     ycoord    =     1.2 }

{ pt5
     est-un    =    point ;
     xcoord    =     5.50 ;
     ycoord    =     4.50 }

{ d1
     est-un         =    disque ;
     barycentre     =    pt1 ;
     rayon          =    6.22 }

{ s1
     est-un         =    segment ;
     origine        =    pt1 ;
     extremite      =    pt2 }

{ s2
     est-un         =    segment ;
     origine        =    pt4 ;
     extremite      =    pt3 }

{ s3
     est-un         =    segment ;
     origine        =    pt1 ;
     extremite      =    pt4 }

{ s4
     est-un         =    segment ;
     origine        =    pt3 ;
     extremite      =    pt5 }

{ c1
     est-un                   =    carre ;
     point-inferieur-gauche   =    pt3 ;
     largeur                  =     1.50 ;
     longueur                 =     1.50 }
```

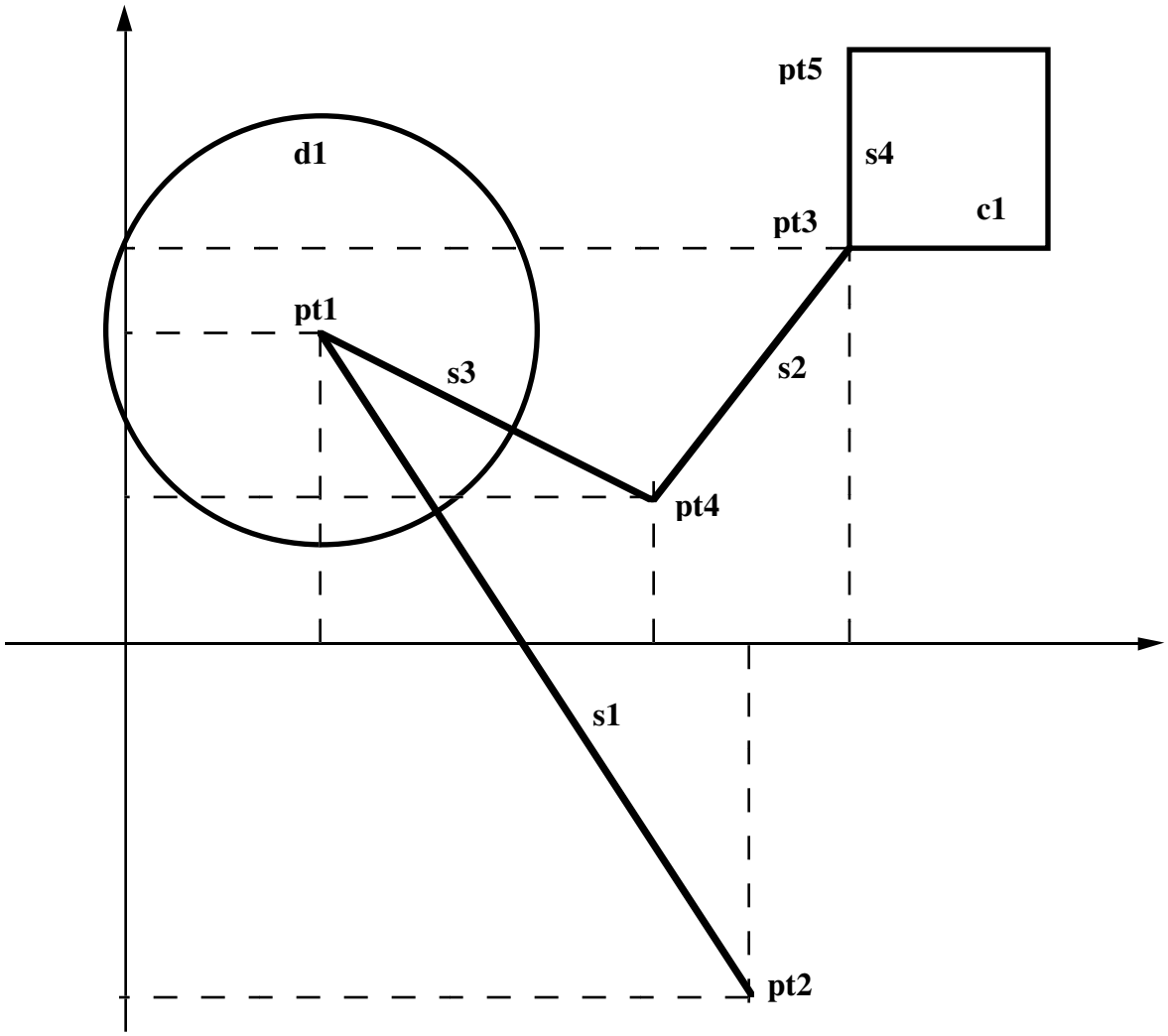


Figure 6 : les instances.

Deuxième partie

Les commandes et les fonctions

Cette deuxième partie présente les commandes utilisables lors du dialogue avec Shirka. Sont également décrites les fonctions Le-Lisp associées à l'écriture d'attachements procéduraux ou à l'emploi de Shirka en tant que sous-système d'une application.

Dans les exemples suivants, ce qui est écrit par l'utilisateur apparaît en **caractères gras**.

1. Installation de Shirka

Toutes les opérations décrites ci-dessous se déroulent évidemment sous l'environnement Le-Lisp natif (Le-Lisp V15.25 actuellement), ou sous Le-Lisp/Aida (V1.65), distribué par la société ILOG. La version de référence actuelle est installée et maintenue sur stations SUN.

1.1 Installation directe

L'installation directe de Shirka suppose l'existence de deux répertoires (ou *directories* dans la terminologie 'Unix') ou dossiers (dans la terminologie 'Macintosh'), sur un disque dur. Le premier reçoit le code source Le-Lisp, le second reçoit la base initiale Shirka et les bases de connaissances des diverses applications. Les noms complets de ces deux répertoires apparaissent en valeur des variables **`#:shirka:*code*`** et **`#:shirka:*bases*`** dans les premières lignes du fichier **`Shirka`**. Ces noms doivent se terminer par le séparateur de répertoires, par exemple :

`/`	pour l'environnement Unix
`:`	pour l'environnement Macintosh

La première étape de cette installation consiste donc à modifier ces valeurs dans le fichier **`Shirka`**, afin qu'elles correspondent aux noms exacts des répertoires de l'installation locale. Il est tout à fait possible de ne prévoir qu'un seul nom de répertoire, qui devient une "racine" commune d'installation; il suffit pour cela de donner la même valeur à ces deux variables.

Dans le cas où l'installation du vérifieur-formateur de bases Shirka est également prévue, il est nécessaire de définir la variable **`#:shirka:*verif*`** qui indique le nom complet du répertoire qui recevra les bases Shirka formatées (cf. la commande "**`verifier`**"). Dans le cas contraire, il suffit de commenter les lignes qui concernent cette option.

La base d'objets initiale de Shirka (préchargée en vecteurs Le-Lisp), est placée dans le fichier **`Base-Shirka`**, du répertoire **`#:shirka:*bases*`** des bases de connaissances.

1.1 Installation complète

Lorsque l'utilisateur de Shirka veut modifier la base pré-installée **`Base-Shirka`** il doit utiliser un autre module, nommé **`Installer-Shirka`**. Le positionnement des variables globales **`#:shirka:*code*`** et **`#:shirka:*bases*`** est à effectuer comme précédemment pour configurer votre installation personnelle.

La seconde étape de l'installation consiste à charger **`Installer-Shirka`** en donnant son nom complet ; par exemple dans une version Macintosh :

```
?(load "TeleMaque:Shirka:Installer-Shirka")
Shirka s installe...
C est fini.
= TeleMaque:Shirka:Installer-Shirka
?
```

Cette seconde étape ne doit être faite qu'une seule fois, lorsque vous créez de nouvelles commandes (§7). Elle a pour résultat de régénérer le fichier **Base-Shirka**. Cette procédure est également utile si cette base venait à être détruite par erreur ; il suffirait alors de recommencer l'opération "Installer-Shirka".

2. Lancer et quitter Shirka

Avant d'utiliser Shirka, il est nécessaire de charger son code Le-Lisp. Il suffit pour cela de demander le chargement du *fichier* **Shirka** en précisant son nom complet (ou simplement "Shirka" si l'on s'est déjà placé dans le bon répertoire, avec une commande !**cd** au "top-level" Le-Lisp, pour un système Unix). Le lancement de Shirka, après ce chargement, s'effectue alors en appelant la *fonction* **shirka** sans argument. Le symbole d'invite (ou 'prompt') **Shirka:** apparaît alors à l'écran.

Par exemple :

```
? !cd <repertoire-installation-Shirka>
?(load "Shirka")     (ou: ?^L Shirka )
Shirka se charge ...
... patience ...
Voila, c est fait.
Taper (shirka) pour demarrer.
= Shirka
?(shirka)
Shirka:
```

Pour quitter Shirka, il suffit de taper le tiret **-**.

```
Shirka:-
= au-revoir
?
```

Il est tout à fait possible d'activer et de désactiver Shirka aussi souvent que nécessaire, respectivement par appel de la fonction **shirka** au "top-level" et par la frappe du tiret **-** sous Shirka.

3. L'aide en ligne et les facilités

Toutes les commandes Shirka sont décrites par des schémas dont les attributs sont les paramètres de la commande. Appeler une commande, c'est donc instancier le schéma de classe correspondant et transmettre l'instance à la fonction associée à la commande.

L'aide en ligne permet d'obtenir des informations sur :

- **?** la liste des commandes disponibles
  ou le commentaire et le type de l'attribut
- **??** le domaine, l'intervalle, la valeur du sauf et la valeur par défaut
- **???** la liste des commandes sous forme de symboles

Les facilités suivantes sont également disponibles :

- **-** pas de valeur ou fin de liste de valeurs
  ou sortie d'un environnement
  ou sortie de Shirka
- **&** appel de Shirka
- **=** valeur de l'appel précédent
- **<<** cet attribut et les suivants reçoivent leurs valeurs de l'appel précédent
- **<** cet attribut et les suivants ne reçoivent pas de valeur
- **>** poursuite de l'instanciation (lors de la création d'une instance)
- **+** appel Lisp

3.1. Aide en ligne ? ?? et ???

L'aide en ligne permet d'obtenir :

- la liste des commandes disponibles,
- des informations sur les paramètres d'une commande,
- la liste des commandes sous forme de symboles.

Il n'est pas nécessaire d'utiliser constamment l'aide interactive. Il est possible de fournir spontanément les valeurs des différents paramètres les unes à la suite des autres, sur une même ligne ou sur plusieurs. A tout moment, l'information sur le paramètre suivant apparaît après l'émission d'un retour chariot.

3.1.1. Liste des commandes disponibles

Pour obtenir la liste des commandes disponibles, il suffit de taper le point d'interrogation **?** derrière le symbole d'invite.

```
Shirka°: ?
Commandes disponibles°: charger verifier sauver l-att l-inst l-spec
cr-inst sup-inst vi val? aj-val mod-val sup-val exec classif tabl info
info-att
Shirka°:
```

3.1.2. Informations sur une commande

En faisant suivre immédiatement le caractère **?** du nom d'une commande, le commentaire d'information associé à celle-ci apparaît.

```
Shirka°: ?sup-inst
Suppression d'une instance
Shirka°: ?sauver
Sauvegarde des instances d'une liste de classes
Shirka°:
```

3.1.3. Informations sur les paramètres d'une commande

Pour appeler une commande, il faut taper son nom suivi d'un retour chariot. Le nom du premier paramètre de la commande apparaît alors. A ce moment, la frappe du point d'interrogation **?** provoque l'affichage à l'écran d'informations sur ce paramètre : un commentaire explicatif, suivi du type des valeurs de ce paramètre ainsi que l'indication **un** ou **liste-de**, qui précise si le paramètre admet une seule valeur ou plusieurs. Le type peut être l'un quelconque des types manipulés par Shirka. Ce peut donc être un type simple (**nombre**, **chaine**, **booleen** ou **symbole**) ou construit (**objet**, **schema**, etc.).

```
Shirka:vi
instance? ?
nom de l instance: un symbole
instance? d1
niveaux?
```

De même, après le symbole d'invite de la classification, le point d'interrogation **?** permet d'obtenir la liste des commandes disponibles dans cet environnement.

En tapant un double point d'interrogation **??**, la valeur par défaut éventuelle du paramètre est affichée, ainsi que le domaine, l'intervalle de valeurs admissibles et la liste de valeurs non admissibles, s'il en possède.

```
niveaux? ?
nombre de niveaux de visualisation: un entier
niveaux? ??
par defaut: 0
niveaux? -
avec-inference? ?
vrai si les valeurs visualisees doivent
etre inferees, faux sinon (par defaut): un booleen
avec-inference? -
{ d1
     est-un = disque ;
     barycentre = pt1 ;
     rayon = 2.000000E0 ; }
-> d1
Shirka:
```

3.1.4. Liste des commandes sous forme de symboles

En tapant un triple point d'interrogation **???**, on obtient un rappel de la liste des commandes sous forme de symboles.

```
Shirka°: l-inst
classe? ???
?  commentaire et type de l'attribut
?? domaine, intervalle, sauf et valeur par defaut
-  pas de valeur ou fin de liste de valeurs
=  valeur de l'appel precedent
<  cet attribut et les suivants ne recoivent pas de valeur
>  poursuite de l'instantiation
<< cet attribut et les  suivants recoivent
   leurs valeurs de l'appel precedent
&  empilement d'appel
+  appel Lisp
classe? segment
-> s1
Shirka°:
```

3.2. Fin

Le tiret **–** indique de façon générale la fin d'un environ :
- quitter Shirka (§ 2),
- quitter un environnement tel que la classification (§ 5.14),
- fin de la liste des valeurs d'un paramètre.

Si le paramètre est mono-valué, la frappe de sa valeur suivie d'un retour-chariot fait passer au paramètre suivant. Dans le cas d'un paramètre multi-valué, toutes les valeurs entrées sont enregistrées jusqu'à l'émission d'un tiret **–**, qui marque la fin de la liste.

La donnée de la valeur peut, dans tous les cas, se réduire à ce tiret : le paramètre prend alors sa valeur par défaut, si cette dernière existe. Dans le cas contraire, il y a un risque d'erreur à l'exécution de la commande.

3.3. Appel récursif de Shirka &

A n'importe quel moment, et en particulier à l'intérieur d'un appel de commande, il est possible d'appeler une autre commande en provoquant l'apparition du symbole d'invite Shirka. Il suffit pour cela de taper le "et-commercial" **&**. Dès que cette nouvelle commande a été exécutée, le dialogue se poursuit à l'endroit où il avait été interrompu.

```
Shirka:sup-val
instance? d1
attribut? &
     Shirka:1-att disque
     -> est-un util-par nom-inst surface barycentre contient rayon
     pts-interieurs
attribut? barycentre
valeur? ?
valeur a supprimer: un objet
valeur? pt1
-> pt1
```

3.4. Valeur(s) de l'appel précédent = et <<

La frappe du caractère **=**, lorsque la valeur d'un paramètre est attendue, signifie que c'est la valeur de ce même paramètre lors de l'appel précédent de la même commande qui doit être retenue. Le signe **<<** signifie que tous les paramètres suivants prennent les mêmes valeurs que lors de l'appel précédent de la commande.

```
Shirka°: sup-val = rayon
valeur? -
->  6.220000E0
Shirka°: sup-val <<
-> echec
Shirka°:
```

3.5. Pas de valeur <

Le signe **<** indique que le paramètre ou l'attribut courant et tous les suivants ne reçoivent pas de valeur explicite de la part de l'utilisateur. Par contre, si des valeurs par défaut sont définies pour ces attributs, elles sont utilisées. Le signe < est utilisable lors de tout appel de commande, mais aussi lors des instanciations interactives avec la commande **cr-inst** (§ 5.6) et des spécialisations et classifications interactives avec les commandes **spec** et **class** (§5.4.1 et §5.4.2).

```
Shirka°: vi d1 <          (Shirka ne posera plus de questions après cette
commande)
{ d1
est-un = disque ;
barycentre = pt1 ;
rayon = 6.220000E0 ;
}
-> d1
Shirka°:
```

3.6. Poursuivre une instanciation >

Ce symbole n'est utilisable que dans la commande **cr-inst** (§ 5.6).

Le signe **>** permet, dans le cas d'un attribut dont le type est une référence à un schéma de classe, de créer une instance de cette classe. Cette instance deviendra la valeur de l'attribut. Cette création a lieu comme si la commande **cr-inst** était appelée à nouveau.

```
Shirka°: cr-inst segment
Classe: segment
Nom de l'instance? s10
barycentre? -
contient? -
origine? >
Classe: point
Nom de l'instance? -
         xcoord? 7.3
         ycoord? 1.8
extremite? <
-> s10
Shirka°:
```

3.7. Appel de lisp +

La frappe du caractère + provoque un appel à l'interpréteur Le-Lisp. L'utilisateur se retrouve au top-level et peut alors demander une évaluation. L'affichage du résultat est suivi du retour à l'environnement Shirka.

```
Shirka°: +
Lisp°: (plist 's10)
(nb-indet 0 schema #[segment () s10 () () #[point (s10) %point-144
 7.300000E0  1.800000E0] () () ()])
Shirka°:
```

4. Les erreurs

Si la valeur fournie pour un paramètre donné ne satisfait pas le type de ce dernier, le message d'erreur **valeur non admissible** apparaît à l'écran. La valeur fournie est ignorée et une autre valeur est attendue.

```
Shirka:aj-val d1
attribut? "barycentre"
valeur non admissible
attribut? ?
nom de l attribut: un symbole
attribut? barycentre
valeur? ?
valeur a ajouter: un objet
valeur? pt1
en-fin? -
-> pt1
Shirka:
```

Dans le cas d'une liste, la valeur erronée est également ignorée. Si les valeurs de la liste sont tapées les unes à la suite des autres, il y a alors un risque de construire une liste avec des valeurs prenant la place des valeurs reconnues non admissibles. Les cardinalités minimum et maximum éventuelles sont également testées pour des attributs multi-valués. Leur non-respect entraîne l'apparition du même message **valeur non admissible**.

D'autres erreurs se manifestent par le fait que la commande renvoie **-> echec**.

```
Shirka:l-inst d1
-> echec
```

5. Les commandes

La liste des commandes disponibles apparaît à l'écran après avoir tapé le point d'interrogation **?** derrière le symbole d'invite Shirka :

charger charger une base de schémas et de fonctions Le-Lisp (suffixe ".sh") ;

verifier vérifier syntaxiquement (et formater) une base de schémas Shirka ;

sauver sauvegarder les instances d'une liste de classes dans un fichier Lisp ;

l-att afficher la liste de tous les attributs, propres et hérités, instanciables et non instanciables, d'une classe ;

l-inst afficher la liste des instances d'une classe ;

l-spec afficher la liste des spécialisations d'une classe ;

cr-inst créer une instance de façon interactive ;

sup-inst supprimer une instance ;

vi visualiser à l'écran une instance à plusieurs niveaux d'imbrication des valeurs de ses attributs ;

val? chercher à déterminer la valeur d'un attribut, en utilisant le cas échéant les divers mécanismes d'inférence ;

aj-val ajouter une valeur à un attribut d'instance ;

mod-val modifier une valeur dans un attribut en la remplaçant par une autre ;

sup-val supprimer une valeur dans un attribut ;

exec lancer l'exécution d'une méthode, après avoir demandé les valeurs de ses arguments, c'est-à-dire des attributs du schéma associé ;

classif donne accès à l'environnement d'identification et de classification ;

tabl donne accès au tableur ShirCalc, si celui-ci a été chargé ;

info afficher à l'écran les commentaires associés à une liste de classes ;

info-att afficher à l'écran des informations sur des attributs d'une classe ;

5.1. Chargement d'une base : charger, verifier

☞ **charger <base>**

permet de charger en mémoire une base Shirka.

☞ **verifier <base>**

permet de vérifier syntaxiquement les schémas d'une base Shirka.

- <**base**> est le nom d'une base, c'est-à-dire d'un fichier texte dans lequel figurent des définitions de schémas et de fonctions Lisp. Ce fichier doit obligatoirement se situer dans le catalogue destiné à recevoir les bases de connaissances et dont le nom a été précisé lors de l'installation (§1);

Les schémas doivent respecter la syntaxe utilisée dans ce manuel, à savoir :
- les schémas eux-mêmes, quelle que soit leur nature (classe, instance, filtre, attachement procédural, spécialisation locale) sont encadrés par des accolades **{** et **}** ;
- la fin d'attribut est indiquée par un point-virgule **;**
- les intervalles sont encadrés par des crochets **[** et **]**.

Les fonctions Lisp sont les fonctions correspondant aux attachements procéduraux.

La commande **verifier** appelle un "vérifieur-formateur" qui analyse la base en une passe ; les erreurs détectées (ainsi qu'une copie formatée de <**base**>) sont envoyées dans un catalogue nommé par la variable **#:shirka:*verif*,** initialisée à l'installation (§1). Pour plus de détails, cf. Annexe "Vérifieur-Formateur de bases Shirka".

```
Shirka:charger figures-du-plan.sh
Erreur: la base TeleMaque:Bases Shirka:figures-du-plan.sh n existe pas.
-> echec
Shirka:charger Objets-du-plan.sh
Chargement de la base TeleMaque:Bases Shirka:objets-du-plan.sh
-> TeleMaque:Bases Shirka:objets-du-plan.sh
Shirka:
```

5.2. Sauvegarde d'une base : sauver

☞ **sauver <classes> <base>**

permet de sauvegarder une base.

- <**classes**> est la liste des classes pour lesquelles les instances doivent être sauvegardées;
- <**base**> est le nom de la base qui doit les recevoir. Le contenu de cette base, si elle existe, est alors écrasé. Le fichier correspondant se situera dans le catalogue ou le dossier destiné à recevoir les bases de connaissances et dont le nom a été précisé lors de l'installation (§1);

```
Shirka°: sauver
classe? segment point -
nom-base? segments-points.sh
```

```
-> TeleMaque:Shirka:Bases Shirka:segments-points.sh
Shirka°:
```

5.3. Liste des attributs : l-att

☞ **l-att <classe> <classes-att> <sauf-classes-att>**

affiche à l'écran la liste des attributs de cette classe: attributs propres et hérités, attributs instanciables et non instanciables (les attributs non instanciables sont les attributs auxquels une facette **$valeur** a été attachée, dans cette classe ou dans une classe supérieure). Les attributs apparaissent dans l'ordre d'instanciation.

- <**classe**> est un symbole, nom d'un schéma de classe;
- <**classes-att**> liste des classes d'attribut à considérer;
- <**sauf-classes-att**> liste des classes d'attribut à ne pas considérer;

```
Shirka:l-att
classe? objet
-> est-un util-par nom-inst
Shirka:l-att schema
-> est-un util-par nom-inst list-att sorte-de spec inst
Shirka:
```

5.4. Liste des instances d'une classe : l-inst

☞ **l-inst <classe>**

affiche à l'écran la liste des instances d'une classe.

- <**classe**> est le nom de cette classe;

La liste obtenue comprend les instances propres de la classe, ainsi que les instances de toutes ses sous-classes. En effet, une instance d'une classe donnée est aussi instance de toutes les classes que sa classe spécialise.

```
Shirka:l-inst objet-du-plan
-> s1 d1 pt1 pt2
Shirka:
```

5.5. Liste des spécialisations d'une classe : l-spec

☞ **l-spec <classe>**

affiche la liste des spécialisations, ou sous-classes, d'une classe.

- **<classe>** est le nom de cette classe;

La liste obtenue comprend toutes les spécialisations, directes et indirectes.

```
Shirka:l-spec figure-plane
-> segment
     disque
     parallelogramme
          rectangle
               carre
                    carre-colore
                         carre-rouge
          losange
figure-coloree
Shirka:l-spec carre
-> carre-colore
     carre-rouge
Shirka:
```

5.6. Création d'instances : cr-inst

☞ **cr-inst <classe> <classes-att> <sauf-classes-att>**

permet de créer des instances de classe de façon interactive.

- <**classe**> est un symbole, nom de la classe à instancier;
- <**classes-att**> liste des classes d'attribut à considérer;
- <**sauf-classes-att**> liste des classes d'attribut à ne pas considérer;

La liste des attributs dont la valeur est demandée à l'utilisateur tient compte de la liste <**classes-att**> ou de la liste <**sauf-classes-att**>. Ces deux paramètres sont exclusifs: un seul doit être valué. Si aucun des deux ne l'est, tous les attributs instanciables sont demandés.

Si <**classe**> possède des spécialisations, Shirka demande si c'est l'une de celles-ci qu'il faut instancier. La liste des sous-classes s'affiche si le point d'interrogation **?** est tapé. Pour ne pas préciser une sous-classe, il suffit de taper le tiret **-**.

Le nom de l'instance à créer est demandé. La liste des noms des instances existantes pour la famille à laquelle appartient la classe est fournie si le symbole ?? est renvoyé. De plus, il est possible de laisser au système le soin de nommer l'instance en tapant le tiret **-**.

Ensuite, Shirka demande à l'utilisateur la valeur de chacun des attributs instanciables de la classe, dans le même ordre que celui présenté par la commande **l-att**. Il est alors possible de répondre par :

- la ou les valeurs. Si l'attribut est multi-valué, la donnée de la liste doit se terminer par un tiret. Si une valeur ne respecte pas le type ou les contraintes (**domaine**, **sauf**, **intervalle**, **a-verifier**) attachées à l'attribut, le message **valeur non admissible** apparaît et la valeur est ignorée.
- un point d'interrogation **? :** le type de l'attribut apparaît, accompagné d'un éventuel commentaire.
- un double point d'interrogation **??** : si les facettes correspondantes sont attachées à l'attribut, directement ou par héritage, alors sont affichées la valeur par défaut, le domaine, les intervalles de valeurs admissibles et la liste de valeurs non admissibles s'il y a lieu.
- le signe **<°** : l'attribut courant et tous les suivants ne reçoivent pas de valeur.
- le signe **>** permet, dans le cas d'un attribut dont le type est une référence à un schéma de classe, de créer une instance de cette classe. Cette instance deviendra

la valeur de l'attribut. Cette création a lieu comme si la commande **cr-inst** était appelée à nouveau.

```
Shirka:cr-inst
nom-classe? disque
Classe: disque
Nom de l instance? d2
surface? -
barycentre? ?
un point
barycentre? >
     Classe: point
     Nom de l instance? ??
     Instances existantes:
     pt1 pt2
     Nom de l instance? pt3
          xcoord? 5.0
          ycoord? 2.5
contient? -
rayon? 3.2
pts-interieurs? -
-> d2
Shirka:
```

Il est possible de fournir comme valeur d'attribut une instance qui n'a pas encore été créée. La valeur est malgré tout enregistrée, mais le test de sa validité est différé. Quand l'instance est effectivement créée, tous les tests en attente sont effectués. Quand un test est négatif, la valeur est retirée de l'attribut où elle apparaissait.

```
Shirka:cr-inst segment
Classe: segment
Nom de l instance? s2
barycentre? -
contient? -
origine? pt3
extremite? pt4
Schema pt4 indefini
Attention: l admissibilite de la valeur pt4
ne peut etre verifiee pour l attribut
extremite de l instance s2 de la classe segment
pente? <
-> s2
Shirka:cr-inst point
Classe: point
Nom de l instance? pt4
xcoord? <
La valeur pt4 de l attribut extremite de l instance s2 de la classe
segment vient d etre verifiee admissible.
-> pt4
Shirka:
```

5.7. Suppression d'une instance: sup-inst

☞ **sup-inst <instance>**

permet de supprimer une instance.

- <**instance**> est le nom de l'instance à supprimer;

```
Shirka:l-inst point
-> pt1 pt2 %point-127
Shirka:sup-inst %point-127
-> %point-127
Shirka:
```

5.8. Visualisation d'une instance : vi

☞ **vi <instance> <vu-comme> <classes-att> <sauf-classes-att> [<niveaux>] [<inference>]**

permet de visualiser à l'écran une instance, en faisant appel le cas échéant aux attachements procéduraux placés derrière les facettes de visualisation de son schéma de classe, directement ou par héritage.

- <**instance**> est le nom de l'instance à visualiser;
- <**vu-comme**> est le nom d'une classe sûre ou possible pour l'instance après classification ou spécialisation. L'inférence de la valeur de l'attribut se déroule comme si l'instance appartenait à cette classe. En fait, l'instance continue d'appartenir à sa classe de rattachement;
- <**classes-att**> est la liste des classes d'attributs à considérer;
- <**sauf-classes-att**> est la liste des classes d'attributs à ne pas considérer;
- <**niveaux**> est le nombre de niveaux de visualisation. Au niveau 1, les valeurs, qui sont des références à d'autres instances par leur nom, sont affichées suivies du contenu de l'instance correspondante au niveau 0. Au niveau 2, ces mêmes valeurs sont affichées suivies du contenu lui-même visualisé au niveau 1, et ainsi de suite;
  ◊ Par défaut, ce nombre est égal à 0, ce qui signifie que les valeurs des attributs de l'instance sont affichées telles quelles.
- <**inference**> est un booléen qui indique si les valeurs des attributs doivent être inférées si elles n'apparaissent pas directement dans l'instance;
  ◊ Par défaut, les valeurs ne sont pas inférées.

```
Shirka:vi d1 <
{ d1
est-un = disque ;
barycentre = pt1 ;
rayon = 6.220000E0 ; }
-> d1
Shirka:vi d1
niveaux? 1 <
{ d1
est-un = disque ;
barycentre = { pt1
          est-un = point ;
          xcoord = 1.500000E0 ;
          ycoord = 2.340000E0 ; } ;
rayon = 6.220000E0 ; }
-> d1
Shirka:vi s1 1
avec-inference? vrai
{ s1
est-un = segment ;
surface = 0.0 ;
barycentre = { %point-127
          est-un = point ;
          xcoord = 3.125000E0 ;
```

```
          ycoord = -2.300000E-1 ; } ;
origine =     { pt1
          est-un = point ;
          xcoord = 1.500000E0 ;
          ycoord = 2.340000E0 ; } ;
extremite =   { pt2
          est-un = point ;
          xcoord = 4.750000E0 ;
          ycoord = -2.800000E0 ; } ;
pente = -1.581538E0 ;
longueur = 6.081291E0 ; }
-> s1
Shirka:
```

5.9. Demande de valeur pour un attribut: val?

☞ **val? <instance> <attribut> <vu-comme>**

permet de lancer l'inférence de la ou des valeurs d'un attribut. Pour obtenir cette valeur, Shirka consulte d'abord l'instance elle-même. Si la valeur n'y figure pas, Shirka essaie les différents moyens d'inférence (valeur de classe, attachement procédural, filtrage, défaut) spécifiés par les facettes correspondantes dans les schémas de classe définissant l'instance.

- <**instance**> est le nom de l'instance;
- <**attribut**> est le nom de l'attribut dans l'instance;
- <**vu-comme**> est le nom d'une classe sûre ou possible pour l'instance après classification ou spécialisation. L'inférence de la valeur de l'attribut se déroule comme si l'instance appartenait à cette classe. En fait, l'instance continue d'appartenir à sa classe de rattachement;

```
Shirka:val? s1 barycentre
-> %point-127
Shirka:vi %point-127 <
{ %point-127
est-un = point ;
xcoord = 3.125000E0 ;
ycoord = -2.300000E-1 ; }
-> %point-127
Shirka:
```

5.10. Ajout de valeur dans un attribut: aj-val

☞ **aj-val <instance> <attribut> <valeur> [<position>]**

permet d'ajouter une valeur dans un attribut (mono ou multi-valué) d'une instance.

Remarque: si la valeur ajoutée est une référence à une instance inexistante, la valeur est malgré tout enregistrée mais le test de validité est différé.

- <**instance**> est le nom de l'instance;
- <**attribut**> est le nom de l'attribut de cette instance;
- <**valeur**> est la valeur à ajouter pour cet attribut dans l'instance;
- <**position**> booléen qui précise *uniquement dans le cas d'un attribut multi-valué*, si l'ajout doit se faire en tête ou en fin de liste;
  - ◊ Par défaut, l'ajout se fait en fin de liste.

```
Shirka:cr-inst point pt4
Classe: point
xcoord? <
-> pt4
Shirka:aj-val
instance? pt4
attribut? xcoord
valeur? ?
valeur a ajouter: un objet
valeur? 0.0
en-fin? -
->  0.0
Shirka:aj-val pt4 ycoord 0.0 -
->  0.0
Shirka:
```

Des erreurs se produisent:
- si l'attribut est inconnu,
- si l'attribut est mono-valué et qu'il possède déjà une valeur,
- si d'éventuelles limites imposées par les facettes **$cardmin** et **$cardmax** ne sont pas respectées,
- de façon générale si la valeur n'est pas admissible.

5.11. Modification de valeur dans un attribut: mod-val

☞ **mod-val <instance> <attribut> <valeur-n> <valeur-a>**

permet, dans un attribut, de remplacer une valeur par une autre.

- <**instance**> est le nom de l'instance;
- <**attribut**> est le nom de l'attribut;
- <**valeur-n**> est la nouvelle valeur;
- <**valeur-a**> est l'ancienne valeur, à remplacer ; elle n'a besoin d'être précisée que dans le cas d'un attribut multi-valué;

```
Shirka:mod-val pt4 xcoord
valeur? ?
nouvelle valeur: un objet
valeur? 1.2
ancienne-valeur? -
-> 1.2
```

5.12. Suppression de valeur dans un attribut: sup-val

☞ **sup-val <instance> <attribut> <valeur>**

permet de supprimer une valeur dans un attribut.

- <**instance**> est le nom de l'instance;
- <**attribut**> est le nom de l'attribut;
- <**valeur**> est la valeur à supprimer ; elle n'a besoin d'être fournie que si l'attribut est multi-valué;

```
Shirka:sup-val pt4
```

```
attribut? ycoord
valeur? ?
valeur a supprimer: un objet
valeur? -
-> 0.0
```

5.13. Exécution d'une méthode : exec

☞ **exec <methode>**

permet de lancer l'exécution d'une méthode ou d'un prédicat après avoir demandé les valeurs de ses arguments, c'est-à-dire des attributs du schéma associé. Cette commande permet donc de tester les méthodes des attachements procéduraux, hors de leur contexte.

- <**methode**> est le nom d'un schéma méthode ou prédicat;

Lorsque Shirka demande les valeurs des attributs du schéma méthode ou prédicat afin de l'instancier, l'utilisateur a les mêmes possibilités de réponses que dans le cas de la commande **cr-inst** (§5.6). Dans le cas d'une méthode, il ne faut évidemment pas fournir la valeur de l'attribut associé au résultat de l'exécution. L'instance partielle est transmise à la fonction Lisp dont le nom figure comme valeur de l'attribut **nom-fct** du schéma méthode. La fonction complète éventuellement l'instance par la valeur de l'attribut résultat. L'ensemble de l'instance est enfin affiché à l'écran, suivi de la valeur de la fonction. S'il s'agit d'un prédicat et qu'il n'est pas vérifié, le message échec apparaît.

```
Shirka°: exec
methode? milieu-segment
a? pt1
b? >
Classe: point
Nom de l'instance? -
     xcoord? 1.5
     ycoord? -2.34
m? -
{ %milieu-segment-133
est-un = milieu-segment ;
a = pt1 ;
b = { %point-136
     est-un = point ;
     xcoord = 1.500000E0 ;
     ycoord = -2.340000E0 ;
} ;
m = { %point-137
     est-un = point ;
     xcoord = 1.500000E0 ;
     ycoord = 0.0 ;
} ;
}
-> %milieu-segment-133
Shirka°: exec pi-r2
r? 4.0
s? -
{ %pi-r2-138
     est-un = pi-r2 ;
     r = 4.000000E0 ;
     s = 5.026544E1 ;
}
-> %pi-r2-138
Shirka°:
```

5.14. Environnement de spécialisation et de classification: classif

Cet environnement regroupe l'ensemble des commandes de spécialisation et de classification d'instances.

☞ **classif**

permet d'entrer dans l'environnement de spécialisation et de classification d'instances. Cette commande ne possède aucun paramètre.

dans classif active, les commandes suivantes, dont la liste peut être obtenue en tapant le caractère **?**, sont accessibles :

spec	**spéc-**ialisation d'une instance d'une classe donnée vers une classe plus spécifique de la même famille
class	**class-**ification d'une instance à l'intérieur d'une famille
ratt	**ratt-**achement effectif d'une instance à une classe donnée
expl-g	**expl-**ication **g-**lobale de la classification d'une instance
expl	**expl-**ications détaillées sur la spécialisation ou la classification d'une instance
v?	affichage de **v-**aleur d'un attribut d'une instance classée ou spécialisée
vi-c	**vi-**sualiser des valeurs des attributs d'une instance **c-**lassée ou spécialisée
m-v	**m-**odification de **v-**aleur d'un attribut d'instance classée ou spécialisée
s-v	**s-**uppression de **v-**aleur dans un attribut d'instance classée ou spécialisée
a-v	**a-**jout de **v-**aleur dans un attribut d'instance classée ou spécialisée
init	(ré)**init-**ialisation de la spécialisation et de la classification.

Le symbole d'invite **Classif:** rappelle que l'environnement est actif. Les conventions générales d'interaction restent valables: le caractère **-** en fait sortir.

```
Shirka:classif
     Classif:?
     Commandes disponibles: spec class ratt expl-g expl v? vi-c m-v s-v
     a-v init
     Classif:-
Shirka:
```

Les exemples d'appels de commandes de spécialisation et de classification qui suivent utilisent la base **courbe-de-reponse.sh**, dont les schémas décrivent différents types de courbes de réponse temporelle :

```
{ courbe-de-reponse
    sorte-de            =         objet ;
    optimum             $un       symbole
                        $domaine  neant minimum maximum
                        $com "existence et nature d un optimum" }

{ courbe-monotone
    sorte-de            =         courbe-de-reponse ;
    optimum             $domaine  neant ;
    croissante          $un       booleen ;
    nb-pts-inflexion    $un       entier
                        $com      "nombre de points d inflexion" ;
    concavite           $un       symbole
                        $domaine  positive negative nulle alternee }

{ monotone-croissante
    sorte-de            =         courbe-monotone ;
    croissante          $domaine  vrai ;
    asymptote-sup       $un       booleen
                        $com      "vrai si la courbe possede "
                                  "une asymptote superieure" }

{ monotone-cr-sans-pt-inflexion
    sorte-de            =         monotone-croissante ;
    nb-pts-inflexion    $domaine  0 }

{ lineaire-cr
    sorte-de            =         monotone-cr-sans-pt-inflexion ;
    concavite           $domaine  nulle ;
    asymptote-sup       $domaine  faux }

{ exponentielle
    sorte-de            =         monotone-cr-sans-pt-inflexion ;
    concavite           $domaine  positive ;
    asymptote-sup       $domaine  faux }

{ puissance
    sorte-de            =         monotone-cr-sans-pt-inflexion ;
    concavite           $domaine  negative ;
    asymptote-sup       $domaine  faux }

{ monod
    sorte-de            =         monotone-cr-sans-pt-inflexion ;
    concavite           $domaine  alternee ;
    asymptote-sup       $domaine  vrai }

{ mono-moleculaire
    sorte-de            =         monotone-cr-sans-pt-inflexion ;
    concavite           $domaine  negative ;
    asymptote-sup       $domaine  vrai }

{ monotone-cr-avec-pt-inflexion
    sorte-de            =         monotone-croissante ;
    nb-pts-inflexion    $intervalle   [ 1 +inf ] ;
    symetrique/pt-inflexion
                        $un       booleen
                        $com      "vrai si la courbe est "
                                  "symetrique par "
                                  "rapport a son point "
                                  "d inflexion" }
```

```
{ logistique
     sorte-de           =         monotone-cr-avec-pt-inflexion;
     concavite          $domaine  alternee ;
     nb-pts-inflexion   $domaine  1 ;
     asymptote-sup      $domaine  vrai ;
     symetrique/pt-inflexion
                        $domaine  vrai }

{ gompertz
     sorte-de           =         monotone-cr-avec-pt-inflexion ;
     concavite          $domaine  alternee ;
     nb-pts-inflexion   $domaine  1 ;
     asymptote-sup      $domaine  vrai ;
     symetrique/pt-inflexion
                        $domaine  faux }

{ monotone-decroissante
     sorte-de           =         courbe-monotone ;
     croissante         $domaine  faux;
     asymptote-inf      $un       booleen
                        $com      "vrai si la courbe possede "
                                  "une asymptote inferieure" }

{ exp-decroissante
     sorte-de           =         monotone-decroissante ;
     nb-pts-inflexion   $domaine  0 ;
     concavite          $domaine  positive ;
     asymptote-inf      $domaine  vrai }

{ lineaire-decr
     sorte-de           =         monotone-decroissante ;
     nb-pts-inflexion   $domaine  0 ;
     concavite          $domaine  nulle ;
     asymptote-inf      $domaine  faux }

{ courbe-non-monotone
     sorte-de           =         courbe-de-reponse ;
     optimum            $domaine  minimum  maximum ;
     symetrique/optimum $un       booleen
                        $com      "vrai si la courbe est symetrique "
                                  "par rapport a son optimum" }
```

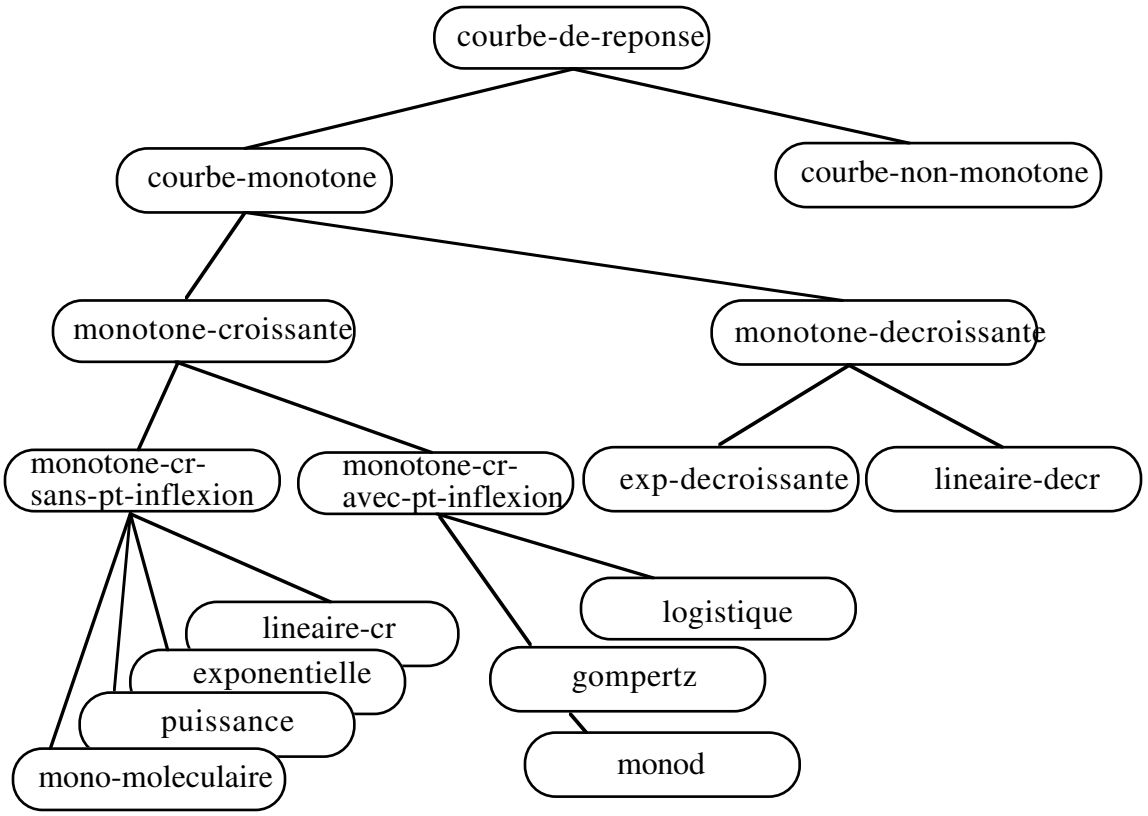


Figure 7 : treillis des courbes de réponse.

5.14.1. Spécialisation d'une instance: spec

☞ **spec <instance> <classe> <classes-att> <sauf-classes-att> [<inference>]**

permet de spécialiser une instance vers une classe de la même famille.

- <**instance**> est le nom de l'instance à spécialiser;
- <**classe**> est le nom de la classe vers laquelle la spécialisation est demandée;
- <**classes-att**> est la liste des classes d'attributs à considérer;
- <**sauf-classes-att**> est la liste des classes d'attributs à ne pas considérer;
- <**inference**> est un booléen qui précise si les valeurs d'attributs nécessaires doivent être inférées (valeur **vrai**) ou demandées à l'utilisateur (valeur **faux**);
  ◊ Par défaut, la valeur est **vrai**.

La liste des attributs impliqués dans la spécialisation tient compte de la liste <**classes-att**> ou de la liste <**sauf-classes-att**>. Ces deux paramètres sont exclusifs: un seul doit être valué. Si aucun des deux ne l'est, tous les attributs sont pris en compte lors de la spécialisation.

Dans le cas où une valeur d'attribut est demandée à l'utilisateur, les possibilités de réponse sont identiques à une création interactive d'instance par la commande **cr-inst**, à la différence près que la frappe du **??** fait apparaître l'union des domaines et intervalles de l'attribut sur l'ensemble de la famille concernée. La spécialisation proprement dite fonctionne suivant les principes expliqués dans la première partie, au §9.3.

```
Shirka:charger courbes-de-reponse.sh
Chargement de la base courbe.sh
-> TeleMaque:Bases Shirka:courbe.sh
Shirka:classif
     Classif:cr-inst courbe-de-reponse
     Classe: courbe-de-reponse
     Sous-classe ? -
     ? ??
     Sous-classes connues:
     courbe-monotone monotone-croissante monotone-cr-sans-pt-inflexion
     lineaire-cr exponentielle puissance monod mono-mol
     monotone-cr-avec-pt-inflexion logistique gompertz monotone-
     decroissante
     exp-decroissante lineaire-decr courbe-non-monotone
     ? -
     Nom de l instance? cr1
     optimum? -
     -> cr1
     Classif:spec cr1
     classe? ?
     nom de la classe a laquelle l instance devrait pouvoir se
     rattacher: un symbole
     classe? exponentielle
     avec-inference? faux
     avec-dialogue ? -
     optimum? ??
     domaine: neant minimum maximum
     optimum? neant
     croissante? vrai
     nb-pts-inflexion? ?
     un entier
     nb-pts-inflexion? ??
     domaine: 0 1
     intervalle: [1 +inf]
     nb-pts-inflexion? 0
     concavite? ??
     domaine: positive negative nulle alternee
     concavite? positive
     asymptote-sup? ?
     un booleen
     asymptote-sup? faux
     -> sur
     Classif:
```

5.14.2. Classification d'une instance: class

☞ **class <instance> <classe> <classes-att> <sauf-classes-att> [<guide>] [<inference>]**

permet de classer une instance.

- <**instance**> est le nom de l'instance à classer;
- <**classe**> est le nom de la classe qui constitue l'élément maximal du sous-treillis intéressé;
- <**classes-att**> est la liste des classes d'attributs à considérer;
- <**sauf-classes-att**> est la liste des classes d'attributs à ne pas considérer;
- <**guide**> est un booléen qui précise si le mode guidé est requis ou non.

- Dans le mode guidé, les domaines ou intervalles affichés à la suite d'une demande **??** correspondent aux seules valeurs possibles compte-tenu des résultats déjà obtenus par la classification, au moment de la demande.
- En mode non guidé, les domaines et intervalles contiennent l'union des valeurs possibles rencontrées dans la famille concernée.

◊ Par défaut, le mode guidé est requis.

- **<inference>** est un booléen qui précise, comme dans la commande **spec-inst** si les valeurs d'attributs nécessaires doivent être inférées ou demandées à l'utilisateur.

◊ Par défaut, les valeurs sont inférées.

La liste des attributs impliqués dans la classification tient compte de la liste **<classes-att>** ou de la liste **<sauf-classes-att>**. Ces deux paramètres sont exclusifs: un seul doit être valué. Si aucun des deux ne l'est, tous les attributs sont pris en compte lors de la classification.

Le résultat fourni par cette commande consiste en une partition du sous-treillis des classes en trois sous-ensembles :

- les classes sûres auxquelles il est certain que l'instance puisse être rattachée,
- les classes impossibles auxquelles il est certain que l'instance ne puisse pas être rattachée,
- et les classes possibles pour lesquelles il n'est pas possible de trancher par manque d'informations sur des valeurs d'attributs.

Un nom de classe sûre ou possible peut être utilisé comme valeur du paramètre **vu-comme** dans les commandes **vi** et **val?**. Dans le premier cas, l'instance sera visualisée, éventuellement avec inférence, comme si l'instance appartenait à la classe **sûre** ou **possible**. Dans le deuxième cas, la valeur de l'attribut est inférée comme si l'instance appartenait à la classe **sûre** ou **possible.** Dans les deux cas, la classe de rattachement de l'instance n'est pas modifiée.

```
Classif:cr-inst courbe-de-reponse - cr2
Classe: courbe-de-reponse
optimum? -
-> cr2
Classif:class cr2
classe? ?
nom de la classe origine a partir
de laquelle on veut classer l instance: un symbole
classe? -
avec-inference? faux
(vrai (par defaut) si les valeurs des attributs
sont inferees, puis demandees si l'inference a echoue;
faux si elles ne sont qu'inferees;
sans signification si avec-inference est faux°: un booleen
avec dialogue ? -
mode-guide? ?
mode guide si vrai (par defaut): un booleen
guide? -
optimum? neant
croissante? vrai
nb-pts-inflexion? 1
concavite? alternee
asymptote-sup? vrai
symetrique/pt-inflexion? ?
un booleen
symetrique/pt-inflexion? faux
Les classes sures:
-> courbe-de-reponse courbe-monotone monotone-croissante
```

```
monotone-cr-avec-pt-inflexion gompertz
Les classes impossibles:
-> courbe-non-monotone monotone-decroissante
monotone-cr-sans-pt-inflexion logistique
Classif:
```

L'algorithme de classification ne peut s'appliquer qu'à l'intérieur d'une même famille. Il est cependant possible au cours d'une classification de faire appel récursivement à l'algorithme sur une autre famille. Prenons immédiatement un exemple :

Soit le problème de déterminer si une situation biologique, à modéliser à l'aide d'un système d'équations dynamiques, est d'un type connu pour lequel on sait proposer un ou plusieurs modèles mathématiques adéquats. Il est possible de construire une famille **situation-biologique**, dont les classes décrivent les différentes situations répertoriées. Les schémas de ces classes possèdent un attribut tel que la courbe de réponse observée à travers les données expérimentales disponibles. Une solution simple consiste à définir cet attribut de type symbole et d'associer dans les sous-classes des domaines pertinents (cf. figure 8). Cette solution est peu satisfaisante dans la mesure où, lors du dialogue avec l'utilisateur, à la question **courbe-de-reponse?**, ce dernier devra choisir un symbole dans l'ensemble des valeurs possibles :

```
courbe-de-reponse? ??
domaine: lineaire-cr exponentielle puissance monod mono-moleculaire
logistique gompertz exp-decroissante lineaire-decr
```

Une meilleure solution est d'utiliser la famille **courbe-de-reponse** précédemment définie afin d'aider l'utilisateur à répondre à la question. Pour ce faire, il suffit dans Shirka de taper le caractère *****. Shirka recherche alors s'il existe une classe de même nom que l'attribut, ici **courbe-de-reponse**, et lance une classification à partir de cette classe. L'utilisateur peut alors choisir sa réponse, ou ses réponses dans le cas d'un attribut multi-valué, uniquement parmi l'ensemble des classes sûres et possibles que cette classification a retenues. Bien entendu, en l'absence du caractère *****, la classification continue de façon classique.

Il est tout à fait possible d'envisager ainsi plusieurs familles, ou *îlots de connaissances*, dans lesquelles la classification se déplace. Il n'y a pas de limite à l'imbrication des classifications, ni au nombre d'attributs dans une famille donnée pour lesquels une (sous-)classification peut être lancée.

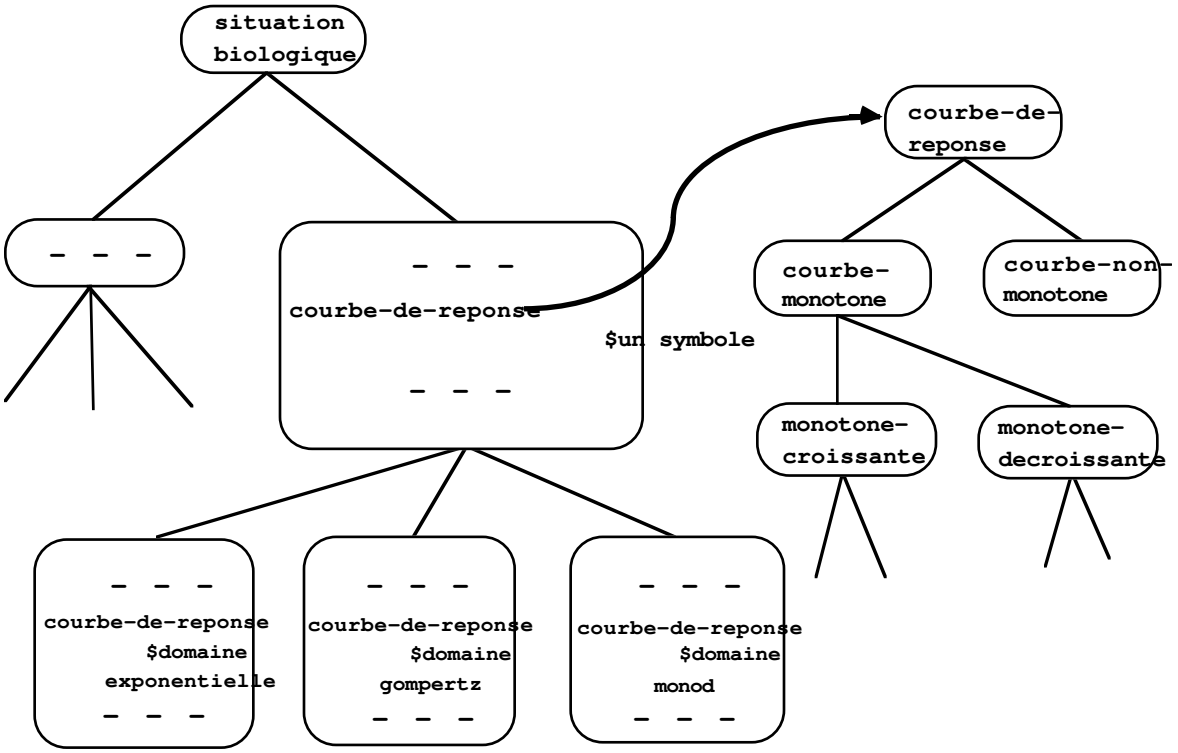


Figure 8 : application récursive de la classification.

5.14.3. Rattachement d'une instance: ratt

Ni la commande **spec**, ni la commande **class** ne changent la classe de l'instance à spécialiser ou à classer. Ce changement de classe reste à la charge de l'utilisateur qui doit s'assurer qu'une instance donnée n'appartient jamais simultanément à deux (et a fortiori plusieurs) classes qui ne sont pas en relation de spécialisation.

☞ **ratt <instance> <classe>**

permet d'effectuer le rattachement d'instance.

- <**instance**> est le nom de l'instance à déplacer;
- <**classe**> est le nom de la nouvelle classe de rattachement;

Avant d'effectuer le rattachement, Shirka vérifie qu'il est licite. Il est nécessaire que la classe à laquelle le rattachement est demandé soit **s re** ou **possible** pour l'instance concernée. Si les commandes **spec-inst** ou **class** ont été appliquées sur l'instance, leurs résultats sont directement exploités. Dans le cas contraire, une phase de spécialisation est préalablement effectuée.

```
Classif:ratt cr2 logistique
Rattachement refuse.
L instance cr2 ne peut pas appartenir a la classe logistique
La valeur de l attribut symetrique/pt-inflexion est faux
alors que l admissibilite est pour: vrai
-> echec
Classif: ratt cr2 gompertz
-> gompertz
Classif:
```

5.14.4. Explication globale de la classification d'une instance : expl-g

☞ **expl-g <instance> <classe>**

permet d'afficher à l'écran les noms des classes d'une même famille, accompagnés de la mention sure, possible ou impossible relativement à une instance donnée, préalablement classée. Elle fournit donc une explication globale de la classification d'une instance. Cette explication peut ensuite être affinée à l'aide de la commande **expl**.

- <**instance**> est le nom de l'instance classée pour laquelle l'explication globale est demandée;
- <**classe**> est le nom de la classe à partir de laquelle l'explication globale est demandée. Cette classe et ses sous-classes sont affichées à l'écran avec la mention sure, possible ou impossible, suivant que l'instance considérée peut, pourrait ou ne peut pas appartenir à la classe affichée. Quand une classe est impossible, ses sous-classes n'apparaissent pas;
  - ◊ Par défaut, c'est la classe "chef de famille" qui est considérée.

```
Shirka°: expl-g cr2 -
courbe-de-reponse (sur)
     courbe-monotone (sur)
          monotone-croissante (sur)
               monotone-cr-sans-pt-inflexion (impossible)
               monotone-cr-avec-pt-inflexion (sur)
               logistique (impossible)
               gompertz (sur)
          monotone-decroissante (impossible)
     courbe-non-monotone (impossible)
-> cr2
Shirka°:
```

5.14.5. Explication d'une spécialisation ou d'une classification : expl

☞ **expl <instance> <classe>**

permet d'obtenir l'explication de la spécialisation ou de la classification d'une instance, c'est-à-dire de préciser la nature de la classe spécifiée : **sûre**, **possible** ou **impossible**.

- <**instance**> est le nom d'une instance sur laquelle ont été appliquées les commandes **spec** ou **class**;
- <**classe**> est le nom d'une classe;

```
Classif:expl
instance? cr2
classe? logistique
L'instance cr2 ne peut pas appartenir a la classe logistique
La valeur de l attribut symetrique/pt-inflexion est faux
alors que l admissibilite est pour: vrai
Classif:
```

5.14.6. Affichage de la valeur d'un attribut d'une instance classée ou spécialisée : v?

☞ **v? <attribut> <instance>**

permet d'afficher la valeur d'un attribut d'une instance. Cette commande ne concerne que des instances préalablement soumises à classification ou spécialisation à l'aide des commandes **class** et **spec**.

- <**attribut**> est le nom de l'attribut;
- <**instance**> est le nom de l'instance;

```
     Classif° : v?
     attribut? concavite
     instance? cr2
-> alternee
     Classif° : v? optimum =
-> neant
     Classif° :
```

5.14.7. Affichage des valeurs des attributs d'une instance classée ou spécialisée : vi-c

☞ **vi-c <instance>**

permet d'afficher les valeurs des attributs d'une telle instance. Dans le cas d'une classification interactive, ces valeurs sont bien entendu les réponses fournies par l'utilisateur. Cette commande ne concerne que des instances préalablement soumises à classification ou spécialisation à l'aide des commandes **class** et **spec**.

- <**instance**> est le nom de l'instance;

```
     Classif° : vi-c
     instance? cr2
optimum = neant ;
croissante = vrai ;
nb-pts-inflexion = 1 ;
concavite = alternee ;
asymptote-sup = vrai ;
symetrique/pt-inflexion = faux ;
-> cr2
     Classif° :
```

5.14.8. Modification de la valeur d'un attribut d'instance classée ou spécialisée : m-v

☞ **m-v <attribut> <valeur> [<ancienne-valeur>] <instance>**

permet de modifier une valeur dans un attribut d'une telle instance. Il est alors possible de relancer la classification ou la spécialisation sur cette même instance et d'obtenir de nouveaux résultats. Cette commande ne concerne que des instances préalablement soumises à classification ou spécialisation à l'aide des commandes **class** et **spec**.

- <**attribut**> est le nom de l'attribut;
- <**valeur**> est la nouvelle valeur;
- <**ancienne-valeur**> est la valeur à modifier. Elle ne doit être précisée que dans le cas où l'attribut est multi-valué;
- <**instance**> est le nom de l'instance;

```
     Classif° : m-v
     attribut? symetrique/pt-inflexion
```

```
     valeur? ?
nouvelle valeur de l'attribut°: un objet
     valeur? vrai
     ancienne-valeur? -
     instance? cr2
-> vrai
     Classif°: class <<
 Les classes sures:
-> courbe-de-reponse courbe-monotone monotone-croissante
monotone-cr-avec-pt-inflexion logistique
 Les classes impossibles:
-> courbe-non-monotone monotone-decroissante
monotone-cr-sans-pt-inflexion gompertz
     Classif°:
```

5.14.9. Suppression de valeur dans un attribut d'instance classée ou spécialisée : s-v

☞ **s-v <attribut> [<valeur>] <instance>**

permet de supprimer une valeur dans un attribut d'une telle instance. Il est alors possible de relancer la classification ou la spécialisation sur cette même instance et d'obtenir de nouveaux résultats. Cette commande ne concerne que des instances préalablement soumises à classification ou spécialisation à l'aide des commandes **class** et **spec**.

- <**attribut**> est le nom de l'attribut;
- <**valeur**> est la valeur à supprimer, elle ne doit être précisée que dans le cas où l'attribut est multi-valué;
- <**instance**> est le nom de l'instance;

```
     Classif°: s-v
     attribut? symetrique/pt-inflexion
     valeur? -
     instance? cr2
-> vrai
     Classif°: class <<
 Les classes sures:
-> courbe-de-reponse courbe-monotone monotone-croissante
 Les classes possibles:
-> monotone-cr-avec-pt-inflexion logistique gompertz
 Les classes impossibles:
-> courbe-non-monotone monotone-decroissante
monotone-cr-sans-pt-inflexion
     Classif°: expl
     instance? cr2
     classe? logistique
L'instance cr2 pourrait appartenir a la classe
monotone-cr-avec-pt-inflexion
La valeur de l'attribut symetrique/pt-inflexion est inverifiable
L'instance cr2 pourrait appartenir a la classe logistique
La valeur de l'attribut symetrique/pt-inflexion est indefinie
alors que l'admissibilite est pour°: vrai
     Classif°:
```

5.14.10. Ajouter une valeur dans un attribut d'instance classée ou spécialisée : a–v

☞ **a-v <attribut> <valeur> <instance>**

permet d'ajouter une valeur dans un attribut d'une telle instance. Il est alors possible de relancer la classification ou la spécialisation sur cette même instance et d'obtenir de nouveaux résultats. Cette commande ne concerne que des instances préalablement soumises à classification ou spécialisation à l'aide des commandes **class** et **spec**.

- <**attribut**> est le nom de l'attribut;
- <**valeur**> est la valeur à ajouter;
- <**instance**> est le nom de l'instance;

```
     Classif°: a-v
     attribut? symetrique/pt-inflexion
     valeur? faux
     instance? cr2
-> faux
     Classif°: class <<
 Les classes sures:
-> courbe-de-reponse courbe-monotone monotone-croissante
monotone-cr-avec-pt-inflexion gompertz
 Les classes impossibles:
-> courbe-non-monotone monotone-decroissante
monotone-cr-sans-pt-inflexion logistique
     Classif°:
```

5.14.11. Réinitialisation de la spécialisation et de la classification : init

☞ **init [<instances>]**

permet d'effacer des informations accumulées par les commandes **spec** et **class** lors de leur exécution. Il est en particulier nécessaire de l'appeler quand, après avoir tenté de spécialiser ou classer une instance, l'utilisateur désire modifier les valeurs d'attributs de cette dernière avant de recommencer. Dans le cas d'une classification interactive, les réponses fournies par l'utilisateur sont oubliées; une autre demande de classification relancera un dialogue complet.

- <**instances**> est la liste des noms des instances pour lesquelles les informations doivent être effacées. Si la liste est vide, toutes les instances soumises à classification ou spécialisation sont concernées par l'initialisation;

```
     Classif°: init
     instances? cr2 -
-> cr2
     Classif°: class
     instance? cr2 <
     optimum? neant
     croissante? faux
     nb-pts-inflexion? 0
     concavite? ??
domaine: positive negative nulle alternee
     concavite? nulle
     asymptote-inf? faux
 Les classes sures:
-> courbe-de-reponse courbe-monotone monotone-decroissante
lineaire-decr
```

```
 Les classes impossibles:
-> courbe-non-monotone monotone-croissante exp-decroissante
     Classif°:
```

5.15. Accès au tableur ShirCalc: tabl

L'accès au tableur ShirCalc est conditionné par le chargement préalable de ce dernier. Au lieu de charger Shirka, il faut charger **ShirCalc**.

```
?^LShirCalc
Shirka se charge ...
... patience ...
Voila, c est fait.
Le tableur se charge...
C est fait.
= shircalc
(shirka)
Shirka:charger objets-du-plan.sh
Chargement de la base objets-du-plan.sh
-> TeleMaque:Bases Shirka:objets-du-plan.sh
```

☞ **tabl <classe> [<mat riel>] [<lignes>] [<colonnes>]**

permet d'appeler le tableur. Le tableur visualise en lignes les instances de cette classe, les colonnes correspondant à ses attributs. Il autorise toutes les manipulations telles que création et suppression d'instance, inférence, ajout, modification et suppression de valeurs d'attributs ainsi que des permutations d'instances et d'attributs afin d'améliorer la lisibilité du tableau. Les déplacements dans ce dernier sont commandés par les touches du curseur au clavier.

- <**classe**> est le nom d'une classe;
- <**mat riel**> est le type de matériel utilisé;
  ◊ Par défaut, il s'agit de l'Apple Macintosh, mais les terminaux VT100 sont également connus;
- <**lignes**> permet de préciser le nombre de lignes du terminal;
  ◊ Par défaut, le nombre de lignes est 24.
- <**colonnes**> permet de préciser le nombre de colonnes du terminal;
  ◊ Par défaut, le nombre de colonnes est 80.

```
Shirka:tabl disque <
```

Le tiret **–** fait sortir du tableur, le point d'interrogation **?** permet d'obtenir la liste des commandes disponibles :

- **e** échange deux colonnes, c'est-à-dire deux attributs. Après avoir positionné le curseur sur chacun des attributs à permuter, la commande **e** doit être tapée.
- **l** liste des valeurs d'un attributs multi-valué. Ces valeurs défilent dans la cellule en utilisant les flèches gauche et droite du curseur. Pour arrêter, il faut retaper **l**
- **d** descend dans le développement d'un attribut, dont le type est défini par une classe. La valeur de cet attribut est affiché comme instance de cette classe.
- **r** remonte au niveau du développement précédent.
- **p** permute deux instances. Après avoir positionné le curseur sur chacune des instances à permuter, la commande **p** doit être tapée.
- **a** ajoute une valeur à un attribut d'instance. L'ajout de la valeur est validée par un retour-chariot.
- **s** supprime la valeur de la cellule.

- **m** modifie la valeur de la cellule. La nouvelle valeur est validée par un retour-chariot.
- **i** infère la valeur de la cellule. Cette valeur disparaît à la frappe d'un autre caractère.
- **c** crée une nouvelle instance vide au dessous de la ligne où est positionné le curseur.
- **t** "tue" une instance, c'est-à-dire la supprime.

5.16.Demande d'informations sur des classes : `info`

☞ **info <classes>**

permet d'afficher à l'écran les textes associés à la facette de commentaire **$com** de l'attribut **lui-meme** des schémas de classes de la liste fournie.

- <**classes**> est la liste des noms de classes pour lesquelles les informations sont demandées;

```
Shirka°: info
classes? exec info -
Classe°: exec
Execution d'une methode

Classe°: info
Demande d'informations sur des classes

Shirka°:
```

5.17.Demande d'informations sur des attributs d'une classe : info-att

☞ **info-att <classe> [<attributs>]**

permet d'afficher à l'écran des informations sur des attributs d'une classe :
le texte du commentaire associé à la facette **$com**, la facette **$un** ou **$liste-de** suivi du type,
le domaine ou les intervalles de valeurs admissibles,
la liste des valeurs exclues par la facette **$sauf**,
la méthode d'inférence utilisée.

Cette commande est applicable à tout schéma de classe, y compris ceux qui décrivent les commandes.

- <**classe**> est le nom de la classe;
- <**attributs**> est la liste des noms des attributs de cette classe pour lesquels les informations sont demandées;
  ◊ Par défaut, tous les attributs de la classe, propres et hérités, sont considérés.

```
Shirka°: info-att
classe? exec
attributs? -
Classe°: exec
Execution d'une methode
Attributs°:
nom-fct
     $un symbole
     valeur°: #:shirka:top:executer
methode
     nom de la methode a executer
     $un symbole
Shirka°: info-att disque
attributs? barycentre
attributs? -
Classe°: disque
Attributs°:
barycentre
     $un point
Shirka°: info-att disque -
Classe°: disque
Attributs°:
surface
     $un reel
     inferable par attachement procedural
barycentre
     $un point
contient
     $liste-de figure-plane
rayon
     $un reel
     intervalle°: [ 0.0 +inf]
pts-interieurs
     $liste-de point
     inferable par filtrage
Shirka°: info-att
classe? info-att
attributs? attributs -
Classe°: info-att
Demande d'informations sur des attributs d'une classe
Attributs°:
attributs
     noms des attributs
     par defaut, tous les attributs
     $liste-de symbole
Shirka°:
```

6. Les fonctions

La connaissance de quelques fonctions Le-Lisp de Shirka est nécessaire pour l'écriture d'attachements procéduraux, méthodes ou prédicats. Certaines de ces fonctions, **val?**, **valeur** et **aj-val**, sont apparues dans les exemples de la première partie. D'autres fonctions sont utiles, en particulier si Shirka est vu comme un sous-système d'une application plus vaste.

6.1. def-sh

☞ **def-sh <description>**

permet de créer des schémas directement dans une fonction Le-Lisp ; elle assure également la compatibilité avec les versions de Shirka qui ne reconnaissaient pas la syntaxe allégée de ce manuel.

- <**description**> est une description complètement parenthésée d'un schéma dans laquelle les parenthèses encadrent les schémas, les attributs, les facettes et les intervalles, mais pas les listes de valeurs;

```
(def-sh  '(point-premier-quadrant
          (sorte-de (= point))
          (xcoord   ($intervalle (0.0 +inf)))
          (ycoord   ($intervalle (0.0 +inf)))))
= #[schema () point-premier-quadrant (#[attribut () () xcoord (#[
$intervalle () () (#[intervalle () () 0.0 +inf])]) $un reel () (( 0.0
+inf)) ()] #[attribut () () ycoord (#[$intervalle () () (#[intervalle (
) ()  0.0 +inf])]) $un reel () (( 0.0 +inf)) ()]) (point) () ()]
```

6.2. #:shirka:noyau:schema

☞ **#:shirka:noyau:schema <nom>**

permet de ramener le schéma lui-même, en représentation interne Shirka.

- <**nom**> est le nom d'un schéma (d'une classe, d'une instance);

```
(#:shirka:noyau:schema 'point)
= #[schema () point (#[attribut () () xcoord (#[$com () () (
abscisse du point)] #[$un () () reel]) $un reel () () ()] #[attribut ()
() ycoord (#[$com () () (ordonnee du point)] #[$un () () reel]) $un
reel () () ()]) (objet-du-plan) (%point-113 %point-114 %point-119
point-premier-quadrant) (pt1 pt2)]
```

6.3. #:shirka:noyau:nom-sch

☞ **#:shirka:noyau:nom-sch <representation>**

permet de ramener le nom du schéma. C'est la fonction réciproque de la précédente.

- <**representation**> est la représentation interne d'un schéma ; elle ramène son nom.

```
(#:shirka:noyau:nom-sch (#:shirka:noyau:schema 'point))
= point
```

6.4. #:shirka:moteur:valeur? et val?

☞ **#:shirka:moteur:valeur? <attribut> <instance>**

permet de retourner la valeur d'un attribut d'une instance. La fonction :**valeur?** appelle le moteur d'inférence.

- <**attribut**> est le nom d'un attribut dont la valeur est requise;
- <**instance**> est le nom de l'instance;

☞ **val?**

La macro **val?** est destinée à faciliter l'écriture des fonctions des attachements procéduraux. En effet, ces fonctions doivent avoir pour paramètre l'instance du schéma correspondant. Elles accèdent aux valeurs des attributs de cette instance par la macro **val?** avec pour argument le nom de l'attribut "quoté". La fonction **valeur?** peut également être employée.

```
(#:shirka:moteur:valeur? 'longueur 's1 )
= 6.081291E0
```

6.5. #:shirka:moteur:aj-valeur et affect

☞ **#:shirka:moteur:aj-valeur <instance> <attribut> <valeur> <position>**

demande l'ajout d'une valeur dans un attribut et retourne cette valeur si celle-ci a pu être affectée, ou **nil** sinon (par exemple en cas de non admissibilité de la valeur).

- <**instance**> est une instance (et non pas son nom);
- <**attribut**> est un nom d'attribut;
- <**valeur**> est une valeur pour cet attribut;
- <**position**> est **nil** si la valeur doit être ajoutée en début de liste dans le cas d'un attribut multi-valué, **t** si en fin de liste;

☞ **affect**

La macro **affect** est destinée à faciliter l'écriture des fonctions d'attachements procéduraux. En effet, ces fonctions doivent compléter l'instance qui leur est passée en paramètre en assignant une valeur à l'attribut résultat. La fonction **aj-valeur** peut être employée à ces fins, mais la macro **affect** est plus simple: elle admet deux arguments, le nom de l'attribut "quoté" et la valeur à lui affecter.

```
(#:shirka:moteur:aj-valeur (#:shirka:noyau:schema 's1)
                   'longueur
                   3.45
                   nil)
= 3.450000E0
```

6.6. #:shirka:moteur:mod-valeur

☞ **#:shirka:moteur:mod-valeur <instance> <attribut> <valeur-a> <valeur-n>**

demande une modification de valeur d'attribut et retourne cette nouvelle valeur si celle-ci a pu être affectée à l'attribut, ou **nil** sinon (par exemple dans le cas où la nouvelle valeur n'est pas admissible ou que la valeur à remplacer n'apparaît pas dans l'attribut).

- <**instance**> est une instance (et non pas son nom);
- <**attribut**> est un nom d'attribut;
- <**valeur-a**> est la valeur de cet attribut à modifier;
- <**valeur-n**> est la nouvelle valeur;

```
(#:shirka:moteur:mod-valeur (#:shirka:noyau:schema 's1)
               'longueur
               3.45
               6.08
               nil)
= 6.08
```

6.7. #:shirka:moteur:sup-valeur

☞ **#:shirka:moteur:sup-valeur <instance> <attribut> <valeur>**

demande une suppression de valeur dans un attribut et retourne cette valeur si celle-ci a pu être supprimée, ou **nil** sinon (c'est-à-dire quand la valeur n'est pas présente dans l'attribut).

- <**instance**> est une instance (et non pas son nom);
- <**attribut**> est un nom d'attribut;
- <**valeur**> est la valeur de cet attribut qui doit être supprimée;

```
(#:shirka:moteur:sup-valeur (#:shirka:noyau:schema 's1)
               'longueur
               6.08
               )
= 6.08
```

6.8. #:shirka:moteur:sup-inst

☞ **#:shirka:moteur:sup-inst <instance> <bsup-spec>**

supprime une instance, et retourne son nom.

- **<instance>** est l'instance à supprimer;
- **<bsup-spec>** est un booléen indiquant si les sous-classes et leurs instances doivent supprimées si **<instance>** est une classe.

```
(#:shirka:moteur:sup-inst (#:shirka:noyau:schema 'd2) nil)
= d2
```

6.9. #:shirka:moteur:creer-instance

☞ **#:shirka:moteur:creer-instance <classe> <valeurs> <instance>**

demande de créer une instance d'une classe donnée à l'intérieur d'une fonction Lisp.

- **<classe>** est le nom de la classe;
- **<valeurs>** est la liste des valeurs des attributs sous la forme d'une A-liste **(°nom°d attribut . valeur)**, où **valeur** est évaluée à l'intérieur de :**creer-instance**;
- **<instance>** est le nom de l'instance;

Si ce dernier paramètre reçoit **nil**, le nom de l'instance est généré par le système. La fonction ramène **nil** si la création n'a pas pu s'effectuer, parce que la classe est inconnue, l'un des attributs est inconnu dans la classe, la valeur d'un attribut n'est pas admissible ou encore parce que l'instance existe déjà. Sinon, la fonction ramène le nom de l'instance créée.

```
(#:shirka:moteur:creer-instance 'point
                '((xcoord . 0.0) (ycoord . 0.0))
                'pt4)
= pt4
```

6.10. #:shirka:vi:visualiser

☞ **#:shirka:vi:visualiser <instance> <niveaux> <inference>**

permet de visualiser une instance à différents niveaux, en exploitant éventuellement les facettes **$pour-vi**, **$avant-vi**, **$en-vi** et **$apres-vi**.

- **<instance>** est l'instance à visualiser (et non pas son nom);
- **<niveaux>** indique le nombre de niveaux;
- **<inference>** est un booléen indiquant si les valeurs des attributs doivent être inférées avant visualisation;

```
(#:shirka:vi:visualiser (#:shirka:noyau:schema 'pt1) 1 nil)
{ pt1
est-un = point ;
```

```
xcoord =  1.500000E0 ;
ycoord =  2.340000E0 ;
} = pt1
```

6.11. #:shirka:classif:class-inst

☞ **#:shirka:classif:class-inst <instance> <classe> <pour-types> <sauf-types> <guide> <inference>**

lance l'algorithme de classification.

- <**instance**> est l'instance à classer;
- <**classe**> est le nom de la classe à partir de laquelle la classification doit commencer;
- <**pour-types**> liste des classes d'attribut à considérer;
- <**sauf-types**> liste des classes d'attribut à ne pas considérer;
- <**guide**> est un booléen qui précise le mode (vrai si guidé, faux sinon);
- <**inference**> est un booléen qui indique si la classification se fait avec inférence des valeurs des attributs (vrai) ou si ces valeurs sont demandées à l'utilisateur (faux);

Elle retourne une liste composée des trois listes des noms des classes **sûres**, **possibles** et **impossibles**.

```
(#:shirka:classif:class-inst
     ( #:shirka:noyau:schema 'cr1)
     'courbe-de-reponse
     vrai
     faux)
```

6.12. #:shirka:noyau:peres <schema>

☞ **#:shirka:noyau:peres <schema>**

retourne la liste des noms des superclasses directes d'une classe donnée.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:peres (#:shirka:noyau:schema 'carre))
= (rectangle losange)
```

6.13. #:shirka:noyau:ancetres <schema>

☞ **#:shirka:noyau:ancetres <schema>**

retourne la liste des noms de toutes les superclasses, y compris donc **objet**, d'une classe donnée.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:ancetres (#:shirka:noyau:schema 'carre))
```

```
= (rectangle losange parallelogramme figure-plane objet-du-plan objet
parallelogramme)
```

6.14. #:shirka:noyau:l-nom-att-inst-he <schema>

☞ **#:shirka:noyau:l-nom-att-inst-he <schema>**

retourne la liste des noms des attributs instanciables, propres et hérités, d'une classe donnée. Chacun de ces noms porte en valeur de propriété position un nombre entier qui est la position de cet attribut dans le vecteur Lisp représentant l'instance et en valeur de propriété nature l'indication $un ou $liste-de suivant que cet attribut est mono ou multi-valué.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:l-nom-att-inst-he (#:shirka:noyau:schema 'carre))
= (est-un util-par nom-inst largeur longueur surface barycentre
contient)
```

6.15. #:shirka:noyau:pos-nom-att <schema> <nom-att>

☞ **#:shirka:noyau:pos-nom-att <schema> <nom-att>**

retourne la position, sous forme d'un nombre entier, d'un attribut dans les vecteurs Le-Lisp représentant les instances d'une classe donnée. L'attribut peut-être interne ou hérité.

- <**schema**> est le schéma de la classe concernée;
- <**nom-att**> est le nom de l'attribut;

```
(#:shirka:noyau:pos-nom-att (#:shirka:noyau:schema 'segment) 'extremite)
= 6
```

6.16. #:shirka:noyau:val-inst <instance> <pos-att>

☞ **#:shirka:noyau:val-inst <instance> <pos-att>**

retourne la valeur d'un attribut connu par sa position dans une instance donnée. Cette fonction ne met pas en œuvre le moteur d'inférence.

- <**instance**> est l'instance concernée (et non pas son nom);
- <**pos-att**> est la position, sous forme d'un nombre entier, de l'attribut dont la valeur est demandée. Cette position peut-être obtenue à l'aide des fonctions Shirka :**pos-nom-att** ou :**l-nom-att-inst-he**;

```
(#:shirka:noyau:val-inst (#:shirka:noyau:schema 'S1) 6)
= pt2
```

6.17. #:shirka:noyau:val-facette <schema> <nom-att> <nom-fac>

☞ **#:shirka:noyau:val-facette <schema> <nom-att> <nom-fac>**

retourne la valeur d'une facette connue par son nom, pour un attribut connu par son nom et dans une classe donnée, sans mettre en œuvre le mécanisme d'héritage. Si la facette n'a pas été redéfinie, retourne **nil**.

- <**schema**> est le schéma de la classe concernée;
- <**nom-att**> est le nom de l'attribut;
- <**nom-fac**> est le nom de la facette;

6.18. #:shirka:noyau:val-fac-he <schema> <nom-att> <nom-fac>

☞ **#:shirka:noyau:val-fac-he <schema> <nom-att> <nom-fac>**

retourne la valeur d'une facette connue par son nom d'un attribut connu par son nom dans une classe donnée, en tenant compte de l'héritage. Si la facette a été redéfinie, retourne la valeur de cette facette sans tenir compte de l'héritage.

- <**schema**> est le schéma de la classe concernée;
- <**nom-att**> est le nom de l'attribut;
- <**nom-fac**> est le nom de la facette;

```
(#:shirka:noyau:val-fac-he (#:shirka:noyau:schema 'carre) 'surface '$sib-
exec)
= (%produit-123)

(#:shirka:noyau:schema '%produit-123)
= #[schema () %produit-123 (#[attribut () () a (#[$var<- () () largeur]
) $un reel () () ()] #[attribut () () b (#[$var<- () () longueur]) $un
reel () () ()] #[attribut () () p (#[$var-> () () surface]) $un reel ()
() ()]) (produit) () ()]
```

6.19. #:shirka:noyau:l-spec <schema>

☞ **#:shirka:noyau:l-spec <schema>**

retourne la liste des noms des classes qui spécialisent directement une classe donnée.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:l-spec (#:shirka:noyau:schema 'parallelogramme))
= (rectangle losange)
```

6.20. #:shirka:noyau:l-spec-t <schema>

☞ **#:shirka:noyau:l-spec-t <schema>**

retourne la liste des noms de toutes les spécialisations d'une classe donnée.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:l-spec-t (#:shirka:noyau:schema 'parallelogramme))
= (rectangle carre carre-colore carre-rouge losange)
```

6.21. #:shirka:noyau:l-spec-t-vraies <schema>

☞ **#:shirka:noyau:l-spec-t-vraies <schema>**

retourne la liste des noms de toutes les spécialisations d'une classe donnée, à l'exception des filtres, des attachements procéduraux et des spécialisations locales.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:l-spec (#:shirka:noyau:schema 'figure-coloree))
= (%figure-coloree-125 carre-colore)

(#:shirka:noyau:l-spec-t (#:shirka:noyau:schema 'figure-coloree))
= (%figure-coloree-125 carre-colore carre-rouge)

(#:shirka:noyau:l-spec-t-vraies (#:shirka:noyau:schema 'figure-coloree))
= (carre-colore carre-rouge)
```

6.22. #:shirka:noyau:l-inst-t <schema>

☞ **#:shirka:noyau:l-inst-t <schema>**

retourne la liste des noms des instances d'une classe donnée et de ses sous-classes.

- <**schema**> est le schéma de la classe concernée;

```
(#:shirka:noyau:l-inst-t (#:shirka:noyau:schema 'figure-plane))
= (s1 d1)
```

6.23. #:shirka:moteur:admissible? <valeur> <nom-att> <schema> <instance>

☞ **#:shirka:moteur:admissible? <valeur> <nom-att> <schema> <instance>**

teste l'admissibilité d'une valeur d'attribut pour une instance donnée.
La fonction retourne la valeur si elle est admissible, *nil* dans le cas contraire.

- <**valeur**> est la valeur à tester. Ce peut être une liste de valeurs, auquel cas toutes les valeurs sont testées, ou une valeur simple;
- <**nom-att**> est le nom de l'attribut;
- <**schema**> est le schéma de la classe à laquelle appartient l'instance;
- <**instance**> est l'instance concernée;

```
(#:shirka:moteur:admissible? "12.56" 'surface (#:shirka:noyau:schema
'disque) (#:shirka:noyau:schema 'd1))
= ()

(#:shirka:moteur:admissible? 12.56 'surface (#:shirka:noyau:schema
'disque) (#:shirka:noyau:schema 'd1))
= 1.256000e+1
```

6.24. #:shirka:top:dem-valeur <schema> <nom-att> <instance> <branger> <bcommande>

☞ **#:shirka:top:dem-valeur <schema> <nom-att> <instance> <branger> <bcommande>**

permet de demander à l'utilisateur la valeur d'un attribut d'une instance et de la ranger éventuellement dans cette instance. La fonction retourne la valeur si elle est admissible et **nil** dans le cas contraire.

- <**schema**> est le schéma de la classe à laquelle appartient l'instance;
- <**nom-att**> est le nom de l'attribut;
- <**instance**> est l'instance concernée;
- <**branger**> doit s'évaluer à :
  - vrai (**non nil**) si la valeur fournie par l'utilisateur doit être rangée dans l'instance,
  - faux (**nil**) sinon;
- <**bcommande**> doit être faux (nil) ;

```
(#:shirka:top:dem-valeur (#:shirka:noyau:schema 'disque) 'surface
(#:shirka:noyau:schema 'd1) t nil)
surface? 12.56
= 1.256000e+1
```

6.25. #:shirka:top:commande <nom-environnement> <prompt>

☞ **#:shirka:top:commande <nom-environnement> <prompt>**

active une boucle de lecture des commandes de l'utilisateur. Cette fonction sert essentiellement lors du développement d'environnements spécialisés dans Shirka.

- <**nom-environnement**> est le nom de l'environnement;
- <**prompt**> est une chaîne de caractères qui servira de message d'invite, lors du dialogue dans cet environnement;

```
(#:shirka:top:commande 'shirka "Shirka°: ")
Shirka°:
Shirka°: ?
Commandes disponibles°: charger sauver l-att l-inst l-spec cr-inst
sup-inst vi val? aj-val mod-val sup-val exec info info-att classif tabl
l-super
Shirka°: -
** toplevel°: echappement indefini°: retour
```

6.26. #:shirka:classif:spec-inst <instance> <nom-sch-dest> <binf> <bdial>

☞ **#:shirka:classif:spec-inst <instance> <nom-sch-dest> <binf> <bdial>**

demande la spécialisation d'une instance vers une classe "cible", généralement une sous-classe de sa classe d'appartenance. Cette fonction retourne le symbole **sur**, **possible** ou **impossible** suivant que l'instance peut appartenir, pourrait appartenir ou ne peut pas appartenir à la classe "cible".

- <**instance**> est l'instance à spécialiser;
- <**nom-sch-dest**> est le nom de la classe "cible";
- <**binf**> est vrai si les valeurs des attributs inconnues dans l'instance doivent être inférées, faux si ces valeurs ne sont que demandées à l'utilisateur;
- <**bdial**> est vrai si les valeurs sont demandées à l'utilisateur quand l'inférence n'a pas rendu de résultat. La valeur de <**bdial**> n'a de sens que lorsque <**binf**> est vrai;

```
(#:shirka:classif:spec-inst (#:shirka:noyau:schema 'cr1) 'logistique nil
t)
optimum? neant
croissante? vrai
nb-pts-inflexion? 1
concavite? ??
domaine: positive negative nulle alternee
concavite? alternee
asymptote-sup? vrai
symetrique/pt-inflexion? vrai
= sur
```

6.27. #:shirka:classif:ratt-inst <instance> <nom-sch-dest>

☞ **#:shirka:classif:ratt-inst <instance> <nom-sch-dest>**

permet de rattacher une instance à une classe, dans la mesure où cette classe est sûre ou possible pour cette instance. La fonction retourne **<nom-sch-dest>** si le rattachement a été effectué, () sinon.

- <**instance**> est l'instance à rattacher;
- <**nom-sch-dest**> est le nom de la classe à laquelle l'instance doit être rattachée;

```
(#:shirka:classif:ratt-inst (#:shirka:noyau:schema 'cr1) 'logistique)
= logistique
(#:shirka:noyau:l-inst (#:shirka:noyau:schema 'logistique))
= (cr1)
```

6.28. #:shirka:classif:fin-cl <liste-instances>

☞ **#:shirka:classif:fin-cl <liste-instances>**

permet d'effacer toutes les informations associées aux tentatives de spécialisation et de classification d'une liste d'instances. La fonction retourne la liste des instances traitées.

- <**liste-instances**> est une liste de noms d'instances pour lesquelles les informations associées doivent être effacées. Si cette liste est vide, toutes les instances précédemment spécialisées ou classées sont traitées;

```
(#:shirka:classif:fin-cl '(cr1))
= (cr1)
```

7. Le développement d'applications en Shirka

Il y a fondamentalement deux modes d'utilisation du système Shirka. Il peut d'une part servir de couche de base d'un programme d'application qui exploite ses fonctions primitives, et d'autre part être utilisé directement à travers son langage de commandes.

7.1. Shirka comme couche de base d'une application

Shirka peut être employé en tant que couche de base d'une application développée en Le-Lisp. Le programmeur fait appel aux fonctions primitives de Shirka, documentées au chapitre 6 (2eme partie). L'interface utilisateur est celle de l'application. Shirka n'est alors pas visible par l'utilisateur, l'application n'exploitant que ses capacités d'inférence.

7.2. Utilisation directe de Shirka

L'utilisation directe de Shirka passe par la création, sous un éditeur quelconque, d'un ou de plusieurs fichiers contenant des descriptions de schémas de classes et d'instances et des textes de fonctions Le-Lisp pour les attachements procéduraux. L'utilisateur doit charger cette base à l'aide de la commande **charger** (§5.1, 2eme partie). Il peut alors créer de nouvelles instances, demander la valeur de leurs attributs et les spécialiser ou encore les classer. Il utilise ainsi directement l'interface Shirka à travers les commandes décrites au chapitre 5 (2eme partie).

Il est également possible de particulariser l'interface utilisateur de Shirka en créant de nouvelles commandes et de nouveaux environnements.

7.2.1. Définition de nouvelles commandes

La définition d'une nouvelle commande dans Shirka comporte deux phases : l'écriture du schéma décrivant la commande et son insertion dans la base de schémas initiale, puis l'écriture de la fonction Le-Lisp réalisant les actions de cette commande.

Les schémas associés aux commandes sont tous des spécialisations du schéma prédéfini **shirka**, lui-même spécialisation du schéma **methode**. Ils possèdent en particulier l'attribut **nom-fct** dont la valeur est le nom de la fonction Le-Lisp associée. Les autres attributs décrivent les paramètres de la commande. Il est recommandé de commenter chacun de ces attributs ainsi que l'attribut **lui-meme**. Ces commentaires seront en effet exploités par l'interface Shirka pour l'aide en ligne. Les schémas décrivant les commandes de base de Shirka apparaissent en annexe. Les fonctions associées constituent le fichier **fcts-commandes**. A titre d'exemple, le schéma suivant définit une nouvelle commande Shirka qui affiche la liste de toutes les super-classes d'une classe, donnée par son nom :

```
{ l-super
     sorte-de       =         shirka ;
     lui-meme       $com      "liste de toutes les super-classes "
                              "d'une classe donnee" ;
     nom-fct        $valeur   fct-l-super ;
     classe         $un       symbole
                    $com      "nom de la classe" }
```

La fonction associée **fct-l-super** utilise la primitive **#:shirka:noyau:ancetres** décrite en 6.13 (2e partie) :

```
(de fct-l-super (inst)
     (impr-res
          (#:shirka:noyau:ancetres
               (#:shirka:noyau:schema
                    (val? 'classe)))))
```

L'insertion de la nouvelle commande dans la base initiale de Shirka nécessite de recréer le fichier **Base-Shirka** en relançant l'installation de Shirka (chapitre 1, 2e partie) après avoir ajouté dans le fichier **installer1** ou **installer2** le texte du schéma **commande** correspondant. La fonction Le-Lisp doit être ajoutée au fichier **fcts-commandes** ou mieux dans un fichier séparé, qui doit bien entendu être chargé en temps utile.

```
(shirka)
Shirka°: ?
Commandes disponibles°: charger sauver l-att l-inst l-spec cr-inst
sup-inst vi val? aj-val mod-val sup-val exec info info-att classif tabl
l-super
Shirka°: ?l-super
liste de toutes les super-classes d'une classe donnee
Shirka°: charger objets-du-plan.sh
Chargement de la base objets-du-plan.sh
-> TeleMaque:Bases Shirka:objets-du-plan.sh
Shirka°: l-super
classe? ?
nom de la classe°: un symbole
classe? rectangle
-> parallelogramme figure-plane objet-du-plan objet
Shirka°: l-super carre-rouge
-> carre-colore carre figure-coloree rectangle losange parallelogramme
figure-plane objet-du-plan objet parallelogramme figure-plane
Shirka°:
```

7.2.2. Définition d'environnements

L'interface utilisateur de Shirka connait la notion d'environnement. Un exemple est donné par la commande **classif** qui permet d'accéder à l'environnement de classification, qui possède ses propres commandes en sus des commandes du premier niveau. Il est donc possible de créer des environnements spécialisés par application. Dans chacun de ces environnements l'utilisateur ne voit que les commandes de l'application, quoique les commandes Shirka soient également utilisables.

Pour créer un tel environnement, il est nécessaire de définir une nouvelle commande portant le nom de cet environnement. A titre d'exemple, voici le texte de définition de la commande **classif** qui permet dans Shirka d'accéder à l'environnement de classification :

```
{classif
     sorte-de       =         shirka ;
     lui-meme       $com      "Environnement de classification" ;
     nom-fct        $valeur   #:shirka:top:classer }
```

La fonction associée, ici :**classer**, doit être écrite suivant le modèle suivant :

```
;definition du sous-systeme Classif de Shirka
(de°:classer (inst)
     (untilexit retour
```

```
          (tag erreur
               ;decalage de la marge gauche
               (with ((lmargin (+ (lmargin) 8)))
                    (:commande 'dial-cl "Classif°: "))))))
```

La fonction **commande** admet deux arguments :

- le nom d'une sous-classe de **methode** - ici **dial-cl** -
- et une chaîne de caractères – ici **Classif°:** – servant de symbole d'invite dans le nouvel environnement. La sous-classe de **methode** doit évidemment être créée.

```
{dial-cl
     sorte-de  =    methode }
```

Toutes les commandes de l'environnement se créent alors comme précédemment, mais en tant que spécialisation de cette sous-classe. L'utilisateur de l'application devra charger et appeler Shirka, puis taper le nom de l'environnement. La frappe du signe **?** sera alors suivie de la liste des commandes de cet environnement, les autres commandes de Shirka et éventuellement des autres environnements, restant utilisables.

Il est cependant possible d'aller plus loin et de permettre à l'utilisateur d'appeler directement l'application et de se retrouver immédiatement dans l'environnement correspondant. Il suffit de définir une nouvelle fonction d'après le modèle suivant :

- **nom-applic** est le nom de l'application,
- **nom-com** est le nom de la commande de plus haut niveau de l'environnement (dans l'exemple de la classification, c'est **classif**),
- **invite** est le symbole d'invite dans l'environnement :

```
(de nom-applic ()
     (untilexit retour
          (tag erreur
               (#:shirka:top:commande 'nom-com "invite°: ")))
'au-revoir)
```

Cette fonction, comme les autres fonctions écrites à l'occasion de l'application, doivent être chargées avec Shirka. L'appel de l'application se fait alors en tapant au "top-level" Le-Lisp l'appel à la fonction **nom-applic**.

Annexes

GLOSSAIRE

Attachement procédural

L'attachement procédural consiste à attacher, par des facettes particulières, à un attribut d'une classe une procédure, ici une fonction Le-Lisp, qui est appelée à l'occasion d'événements tels que :

- demande (facette **$sib-exec**),
- ajout (**$avant-ajout**, **$apres-ajout**),
- modification (**$avant-mod**, **$apres-mod**)
- suppression de valeur (**$avant-sup**, **apres-sup**),
- succès ou échec d'obtention d'une valeur (**$si-succes**, **$si-echec**)
- test d'admissibilité d'une valeur (**$a-verifier**).

Attribut

Tout schéma, de classe et d'instance, est composé d'un nom et d'une liste d'attributs. Les attributs décrivent les propriétés et les composants d'un objet. Dans une classe, un attribut est décrit par une liste de facettes. Dans une instance, il est décrit par une ou plusieurs valeurs.

Classe

Une classe d'objets est définie par un schéma de classe qui décrit de façon générique les objets de cette classe.

Classification

La classification est un mécanisme d'inférence qui permet, après avoir créé une instance d'une classe donnée, de déterminer à quelles sous-classes l'instance est susceptible d'être rattachée.

Facette

Les facettes décrivent les attributs d'une classe. Il existe plusieurs catégories de facettes.

- Les facettes de typage (**$un** et **$liste-de**) assignent un type à l'attribut.
- Les facettes de restriction (**$domaine**, **$sauf**, **$intervalle**, **$a-verifier**, **$cardmin**, **$cardmax**) permettent d'introduire des contraintes que doivent respecter les valeurs d'un attribut pour être admissibles.
- La facette **$com** sert à commenter un attribut.
- Les facettes d'inférence (**$valeur**, **$defaut**, **$sib-exec**, **$sib-filtre**) sont exploitées lors de l'obtention d'une valeur d'attribut indéterminé.
- Les facettes de réaction (**$si-succes**, **$si-echec**, **$avant-ajout**, **$apres-ajout**, **$avant-mod**, **$apres-mod**, **$avant-sup**, **$apres-sup**) permettent l'exécution de fonctions quand des événements se produisent sur les attributs auxquels elles sont attachées.
- Les facettes de manipulation de variables (**$var-nom**, **$var<-**, **$var->**, **$var-liste<-**, **$var-liste->**) servent à référencer des attributs dans les attachements procéduraux, les filtres et les spécialisations locales.
- Enfin, les facettes de visualisation (**$avant-vi**, **$apres-vi**, **$pour-vi**, **$en-vi**) sont utilisées pour visualiser une instance suivant une représentation externe propre à la classe.

Famille

On appelle famille dans Shirka, un ensemble de classes formant un sous-treillis dont l'élément maximal est une spécialisation directe de **objet**. Une base de connaissances est généralement constituée de plusieurs familles. Les classes définies par spécialisation simultanée de classes appartenant à des familles différentes sont dénuées de sens. De même, les algorithmes de spécialisation et de classification ne peuvent s'appliquer qu'à l'intérieur d'une même famille.

Filtre, filtrage

Un filtre est un schéma, spécialisation du schéma de classe dont il porte le nom. Le filtrage est le mécanisme d'inférence qui consiste à rechercher les instances de cette classe qui satisfont le filtre et à en extraire, par des facettes **$var->** ou **$var-liste->**, les valeurs de l'attribut auquel le filtre est attaché par la facette **$sib-filtre**.

Héritage

Un schéma de classe hérite des attributs des schémas de classe dont il est une spécialisation à travers le lien **sorte-de**. Pour chaque attribut, il hérite des facettes de l'attribut de même nom suivant une stratégie propre à chaque catégorie de facette.

Inférence

Dans Shirka, l'inférence consiste à déterminer les valeurs d'attribut manquantes dans des instances en utilisant la connaissance contenue dans les classes, en particulier les facettes **$valeur**, **$sib-exec**, **$sib-filtre** et **$defaut**.

Instance

Un schéma d'instance décrit un objet particulier d'une classe. Les attributs de cette dernière y reçoivent des valeurs.

Instancier, instanciation

Instancier un schéma de classe, c'est créer une instance de la classe et tenter de déterminer la valeur de chacun de ses attributs par inférence.

Méthode

Une méthode est définie par un schéma de classe, spécialisation de la classe **methode** ou **methode-sans** et dont elle hérite l'attribut **nom-fct**. Une spécialisation de ce schéma portant le même nom peut alors apparaître derrière les facettes d'attachement procédural. La valeur de l'attribut **nom-fct** permet d'y associer une fonction Le-Lisp. S'il s'agit d'une spécialisation de **methode-sans**, l'appel de cette fonction n'est pas précédé d'une étape de filtrage.

Prédicat

Un prédicat est défini par un schéma de classe, spécialisation de la classe **predicat** ou **predicat-sans** et dont il hérite l'attribut **nom-fct**. Une spécialisation de ce schéma portant le même nom peut alors apparaître derrière la facette **$a-verifier**. La valeur de l'attribut **nom-fct** permet d'y associer une fonction Le-Lisp, qui doit renvoyer faux ou vrai (nil ou non nil). S'il s'agit d'une spécialisation de **predicat-sans**, l'appel de cette fonction n'est pas précédé d'une étape de filtrage.

Restriction

La restriction d'une classe se traduit par l'écriture d'un schéma comportant par héritage les mêmes attributs que celui de la classe qu'il restreint, plus de nouveaux attributs propres. Les attributs hérités ne peuvent être que précisés et non redéfinis.

Schéma

Le schéma est l'unité de description de base de Shirka. C'est un schéma de classe s'il décrit une classe d'objets. C'est un schéma d'instance s'il décrit un objet particulier d'une classe.

Schéma méthode

Schéma définissant une *méthode*.

Schéma prédicat

Schéma définissant un *prédicat*.

Spécialisation

Ce terme désigne à la fois une classe qui en spécialise une ou plusieurs autres par restriction et l'action qui consiste à déterminer si une instance d'une classe donnée peut être considérée comme instance de l'une de ses sous-classes.

Spécialisation locale

Les spécialisations locales peuvent apparaître derrière les facettes **$un** et **$liste-de**. Comme leur nom l'indique, elles constituent une spécialisation de la classe dont elles portent le nom. Une spécialisation locale permet d'introduire des restrictions supplémentaires que doivent satisfaire les instances en tant que valeurs de l'attribut auquel elles sont attachées. En pratique, elles servent souvent à faire référence à des attributs imbriqués par l'intermédiaire de la facette **$var-nom**.

Treillis, demi-treillis

Un treillis est un graphe partiellement ordonné possédant un sommet maximal et un sommet minimal; un demi-treillis supérieur possède un sommet maximal, mais éventuellement plusieurs sommets minimaux. Dans Shirka, la relation de spécialisation (attribut **sorte-de**) structure l'ensemble des classes en un demi-treillis supérieur de sommet maximal **objet**. Toutes les classes sans spécialisation constituent autant de sommets minimaux.

Valeur admissible

Une valeur d'attribut est admissible si elle satisfait toutes les conditions associées à cet attribut par les facettes **$un**, **$liste-de**, **$intervalle**, **$domaine**, **$sauf**, **$a-verifier**, **$cardmin** et **$cardmax**. Eventuellement, l'admissibilité d'une valeur passe par une étape de spécialisation.

Variable

Les variables permettent de faire référence à des attributs d'un schéma de classe dans les attachements procéduraux, les filtres ou les spécialisations locales. Sauf ambiguïté, un attribut peut être référencé par la variable du même nom. En cas d'ambiguïté, il faut recourir à la facette **$var-nom**.

LISTE DES FACETTES

$a-verifier

Facette d'attachement procédural. Elle est suivie d'un ou de plusieurs schémas prédicat. Ces prédicats doivent être tous vérifiés pour qu'une valeur de l'attribut auquel est associée la facette soit admissible.

$apres-ajout

Facette d'attachement procédural. Les méthodes qui la suivent sont déclenchées en cas d'ajout de valeur à l'attribut, après que cet ajout ait été réalisé.

$apres-mod

Facette d'attachement procédural. Les méthodes qui la suivent sont déclenchées en cas de modification de la valeur de l'attribut, après que cette modification ait été réalisée.

$apres-sup

Facette d'attachement procédural. Les méthodes qui la suivent sont déclenchées en cas de suppression de la valeur de l'attribut, après que cette suppression ait été réalisée.

$apres-vi

Facette de visualisation. Les méthodes qui la suivent sont déclenchées, lors de la visualisation d'une instance demandée par la commande ou la fonction **vi**, après que l'attribut auquel elle est attachée ait été lui-même visualisé.

$avant-ajout

Facette d'attachement procédural. Les méthodes qui la suivent sont déclenchées en cas d'ajout de valeur à l'attribut, avant que cet ajout ait été réalisé.

$avant-mod

Facette d'attachement procédural. Les méthodes qui la suivent sont déclenchées en cas de modification de la valeur de l'attribut, avant que cette modification ait été réalisée.

$avant-sup

Facette d'attachement procédural. Les méthodes qui la suivent sont déclenchées en cas de suppression de la valeur de l'attribut, avant que cette suppression ait été réalisée.

$avant-vi

Facette de visualisation. Les méthodes qui la suivent sont déclenchées, lors de la visualisation d'une instance demandée par la commande ou la fonction **vi**, avant que l'attribut auquel elle est attachée ne soit lui-même visualisé.

$cardmax

Facette de restriction. Elle permet de préciser le nombre maximum de valeurs admises pour un attribut multi-valué.

$cardmin

Facette de restriction. Elle permet de préciser le nombre minimum de valeurs requises pour un attribut multi-valué.

$com

Facette de commentaire. Elle est suivie d'une liste de chaînes de caractères exploitées lors de l'instanciation interactive de la classe (commande **cr-inst**). Les commentaires facilitent la maintenance des bases de connaissance.

$defaut

Facette d'inférence. La valeur qui la suit est retenue comme valeur de l'attribut si aucun autre moyen d'inférence n'a fonctionné auparavant.

$domaine

Facette de restriction. Elle est suivie d'une liste des valeurs possibles de l'attribut.

$en-vi

Facette de visualisation. Les méthodes qui la suivent sont déclenchées, lors de la visualisation d'une instance demandée par la commande ou la fonction **vi**, entre les visualisations de valeurs consécutives de l'attribut multi-valué auquel elle est attachée.

$intervalle

Facette de restriction. Elle est suivie d'un ou de plusieurs intervalles donnés par un ou plusieurs couples de bornes inférieure et supérieure. La valeur de l'attribut correspondant doit être située dans l'un de ces intervalles. Cette facette ne peut être associée qu'à un attribut de type simple.

$liste-de

Facette de typage. Elle joue le même rôle que la facette **$un**, mais indique un attribut multi-valué.

$pour-vi

Facette de visualisation. Les méthodes qui la suivent sont déclenchées, lors de la visualisation d'une instance demandée par la commande ou la fonction **vi**, pour visualiser l'attribut auquel elle est attachée.

$sauf

Facette de restriction. Elle est suivie d'une liste de valeurs que l'attribut correspondant ne peut pas prendre.

$si-echec

Facette d'attachement procédural. Les méthodes qui la suivent sont appelées en cas de échec de l'obtention de la valeur de l'attribut par inférence.

$si-succes

Facette d'attachement procédural. Les méthodes qui la suivent sont appelées en cas de succès de l'obtention de la valeur de l'attribut par inférence.

$sib-exec

Facette d'inférence et d'attachement procédural. Elle est suivie d'un ou de plusieurs schémas **methode**, avec ou sans filtrage. Ces schémas sont associés à des fonctions Le-Lisp. Lors de la recherche de la valeur de l'attribut, un schéma **methode** est instancié et son instance est transmise à la fonction qui la complète. L'instance est renvoyée à Shirka qui propage le résultat.

$sib-filtre

Facette d'inférence. Elle est suivie d'un ou plusieurs filtres. Un filtre décrit les instances recherchées. La valeur de l'attribut est extraite des instances qui satisfont le filtre (cf. *filtrage*).

$un

Facette de typage. Elle est suivie du nom d'un type simple (**entier**, **reel**, **chaine**, **booleen** ou **symbole**), ou du nom d'une classe. Dans ce dernier cas, les valeurs de l'attribut correspondant sont des instances de cette classe.

$valeur

Facette d'inférence. Elle fournit une valeur d'attribut valable pour toutes les instances de la classe.

$var<-

Indique que l'attribut peut recevoir la valeur de la variable dont le nom suit.

$var->

Indique que la valeur de l'attribut doit être transmise à la variable dont le nom suit.

$var-liste<-

Semblable à **$var<-** mais précise, dans le cas d'une variable associée à un attribut multi-valué, que la valeur reçue de la variable est une liste considérée dans son ensemble.

$var-liste->

Semblable à **$var->** mais précise, dans le cas d'un attribut multi-valué, que la valeur transmise à la variable est la liste dans son ensemble.

$var-nom

Permet d'associer explicitement une variable à un attribut et de lever ainsi les ambiguïtés liées au fait que par défaut une variable fait référence à l'attribut de même nom et que plusieurs attributs de même nom peuvent apparaître dans un même schéma.

SYNTAXE

Les schémas doivent respecter la syntaxe utilisée dans ce manuel, à savoir :

- les schémas eux-mêmes, quel que soit leur niveau d'imbrication et leur nature (classe, instance, filtre, attachement procédural, spécialisation locale) sont encadrés par des accolades **{** et **}** ;
- la fin d'attribut est indiquée par un point-virgule **;**
- et les intervalles sont encadrés par des crochets **[** et **]**.

Les fonctions Le-Lisp sont les fonctions correspondant aux attachements procéduraux.

Liste des classes prédéfinies

```
* objet
    + attribut
    + facette
    + schema
    + attachement-proc
      •    pr dicat
      •    pr dicat-sans
      •    m thode
      •    m thode-sans
```

Liste des attributs prédéfinis

```
* sorte-de
* lui-meme
* nom-fct
* est-un
```

Commandes sous forme de symboles

<	cet attribut et les suivants ne reçoivent pas de valeur
<<	cet attribut et les suivants reçoivent leurs valeurs de l'appel précédent
?	commentaire et type de l'attribut
??	domaine, intervalle, sauf et valeur par défaut
???	liste des réponses possibles
-	pas de valeur ou fin de liste de valeurs
>	poursuite de l'instanciation
=	valeur de l'appel précédent
+	appel top-level Le-Lisp
&	empilement d'appel

Shirka en Shirka

```
{ schema
     est-un          =    schema ;
     nom             =    schema ;
     sorte-de        =    objet ;
     list-att        =    a-sorte-de a-liste-att a-spec a-inst ;
     inst            =    schema objet attribut facette
                          attachement-proc methode methode-sans
                          predicat predicat-sans }

{ schema
     sorte-de        =              objet ;
     sorte-de        $liste-de      schema ;
     list-att        $liste-de      attribut ;
     spec            $liste-de      schema ;
     inst            $liste-de      objet }

{ Objet
     est-un     =    schema ;
     nom        =    objet ;
     spec       =    attribut facette attachement-proc schema ;
     list-att   =    a-est-un a-util-par a-nom }

{ attribut
     sorte-de        =              objet ;
     nom-att         $un            symbole ;
     list-fac        $liste-de      facette ;
     nature          $un            symbole
                     $domaine       $un $liste-de ;
     type            $un            symbole ;
     domaine         $liste-de      objet ;
     intervalle      $liste-de      intervalle ;
     sauf            $liste-de      objet }

{ intervalle
     sorte-de        =    objet ;
     borne-inf       $un  symbole ;
     borne-sup       $un  symbole }

{ a-sorte-de
     est-un          =    attribut ;
     nom-att         =    sorte-de ;
     list-fac        =    { est-un  =    $un ;
                          list-val  =    schema } }

{ facette
     sorte-de        =              objet ;
     nom-fac         $un            symbole ;
     list-val        $liste-de      objet }

{ $un
     sorte-de  =         facette ;
     nom-fac   $valeur   $un }

{ $sib-exec
     sorte-de  =         facette ;
     nom-fac   $valeur   $sib-exec }
```

```
{ $com
     sorte-de       =              facette ;
     nom-fac        $valeur        $com ;
     list-val       $liste-de      chaine }

{ attachement-proc
     sorte-de  =    objet ;
     nom-fct   $un  symbole }

{ methode
     sorte-de  =    attachement-proc }

{ methode-sans
     sorte-de  =    attachement-proc }

{ predicat
     sorte-de  =    attachement-proc }

{ predicat-sans
     sorte-de  =    attachement-proc }
```

Schémas des commandes Shirka:

```
{charger
  sorte-de    = shirka ;
  lui-meme    $com  "Chargement d'une base de connaissances" ;
  nom-fct     $valeur     #:shirka:top:charger-fic  ;
  nom-base    $un  symbole
              $com  "nom de la base a charger"
}

{verifier
  sorte-de    = shirka ;
  lui-meme    $com  "Verification et formatage d'une base Shirka" ;
  nom-fct     $valeur     #:shirka:verif:af-shirka  ;
  nom-base    $un  symbole
              $com  "nom de la base a verifier et formater"
}

{sauver
  sorte-de    =     shirka ;
  lui-meme    $com "Sauvegarde des instances d'une liste de classes" ;
  nom-fct     $valeur     #:shirka:top:sauver-inst  ;
  classe      $liste-de  symole
              $com  "noms des classes des instances a sauver" ;
  nom-base    $un   symbole
              $com "nom de la base de sauvegarde"
}

{l-att
  sorte-de    =     shirka ;
  lui-meme    $com  "Liste des attributs d'une classe" ;
  nom-fct     $valeur     #:shirka:top:liste-att  ;
  classe      $un  symbole
              $com   "nom de la classe dont on veut "
                     "connaitre  les  attributs";
  classes-att
              $liste-de   symbole
              $com  "liste des classes d'attributs a considerer" ;
  sauf-classes-att
              $liste-de   symbole
              $com  "liste des classes d'attributs a ne pas considerer"
}

{l-inst
  sorte-de    =     shirka ;
  lui-meme    $com  "Liste des instances d'une classe" ;
  nom-fct     $valeur     #:shirka:top:liste-inst  ;
  classe      $un  symbole
              $com  "nom de la classe"
}

{l-spec
  sorte-de    =     shirka ;
  lui-meme    $com  "Liste des specialisations d'une classe" ;
  nom-fct     $valeur  #:shirka:top:liste-spec  ;
  classe      $un  symbole
              $com  "nom de la classe" ;
  niveaux     $un   entier
              $intervalle [  0  9999  ]
              $com  "nombre  de  niveaux  de  specialisation"
}
```

```
{cr-inst
   sorte-de    =     shirka ;
   lui-meme    $com  "Creation d'une instance" ;
   nom-fct     $valeur     #:shirka:top:creer-instance  ;
   nom-classe
               $un  symbole
               $com  "nom de la classe a instancier" ;
   classes-att
               $liste-de  symbole
               $com  "liste des classes d'attributs a considerer" ;
   sauf-classes-att
               $liste-de  symbole
               $com "liste des classes d'attributs a ne pas considerer"
}

{  sup-inst
   sorte-de    =     shirka ;
   lui-meme    $com  "Suppression d'une instance" ;
   nom-fct     $valeur     #:shirka:top:supprimer-inst  ;
   instance    $un  symbole
               $com  "nom de l'instance a supprimer"
}

{  vi
   sorte-de    =     shirka ;
   lui-meme    $com  "Visualisation d'une instance" ;
   nom-fct     $valeur     #:shirka:top:visu-inst  ;
   instance    $un   symbole
               $com  "nom de l'instance" ;
   vu-comme    $un   symbole
               $com  "classe temporaire de rattachement"
   classes-att
               $liste-de   symbole
               $com  "liste des classes d'attributs a considerer" ;
   sauf-classes-att
               $liste-de   symbole
               $com  "liste des classes d'attributs a ne pas considerer";
   avec-inference
               $un   booleen
               $defaut     faux
               $com  "vrai si les valeurs visualisees doivent"
                     "etre inferees, faux sinon (par defaut)" ;
   niveaux     $un entier
               $defaut     0
               $com  "nombre de niveaux de visualisation"
}

{  val?
   sorte-de    = shirka ;
   lui-meme    $com  "Obtention de la valeur d'un attribut") ;
   nom-fct     $valeur     #:shirka:top:obtenir-valeur  ;
   instance    $un   symbole
               $com  "nom de l'instance" ;
   attribut    $un  symbole
               $com  "nom de l'attribut" ;
   vu-comme    $un  symbole
               $com  "classe temporaire de rattachement"
}
```

```
{ manip-val
  sorte-de    =     shirka ;
  instance    $un  symbole
              $com  "nom de l'instance" ;
  attribut    $un   symbole
              $com  "nom de l'attribut" ;
  valeur      $un   objet ;
  l-spec      $valeur     aj-val  mod-val  sup-val
}

{ exec
  sorte-de    =     shirka ;
  lui-meme    $com  "Execution d'une methode" ;
  nom-fct     $valeur     #:shirka:top:executer  ;
  methode     $un  symbole
              $com  "nom de la methode a executer"
}

{ info
  sorte-de    =     shirka ;
  lui-meme    $com  "Demande d'informations sur des classes" ;
  nom-fct     $valeur     #:shirka:top:info  ;
  classes     $liste-de   symbole
              $com  "liste des noms de classe"
}

{ info-att
  sorte-de    =     shirka ;
  lui-meme    $com  "Demande d'informations sur des attributs "
                    "d'une classe" ;
  nom-fct     $valeur     #:shirka:top:info-att  ;
  classe      $un   symbole
              $com  "nom de la classe" ;
  attributs   $liste-de   symbole
              $com  "noms des attributs; par defaut, tous les attributs"
}

{ classif
  sorte-de    =     shirka ;
  lui-meme    $com  "Environnement de classification" ;
  nom-fct     $valeur     #:shirka:top:classer
}
```

LISTE DES FONCTIONS (de…, (dmd…, (defvar… dans <SHIRKA>.ll

Classement : ordre d'apparition dans le code LeLisp (sauf les 'dmd' listés a part)
Version Shirka : /softs/SHIRKA/shirka-iace/
-- in -- : nom du fichier courant pour la recherche

```
-- in --  DB-filtre1.ll
(defvar  #:sys-package:colon  '#:shirka:moteur:filtrage)
(defvar  #:exp:trace  nil)
(de  :FILTRAGE  (nom-inst-f  nom-filtre)
(de  :FILTRAGE-BOITEUX  (nom-inst-f  nom-filtre)
(de  :ACTION-ECHEC-FILTRAGE  (nom-inst-f  nom-filtre)
(de  :ACTION-SUCCES-FILTRAGE  (nom-inst-f  nom-filtre  filt-h-niv  list-inst
(de  :FILTRER-INST  (nom-inst-f  nom-filtre)
(de  :A-VERIF-LUI-MEME  (nom-filtre)
(de  :MEC-ATT-FIXES-1  (nom-filtre  l-att-fixes  nom-inst-f  nom-inst-trav)
(de  :MEC-ATT-FIXES-2  (l-att-fixes  nom-inst-f  nom-filtre  nom-inst-trav)
(de  :MEC-ATT  (  nom-filtre  val-att-t  val-att-f  nom-att  )
(de  :MEC-AUTRES  (l-att-var  nom-inst-f  nom-filtre  nom-inst-trav)
(de  :MAJ-INST-TRAV   (nom-filtre  inst-cour)
(de  :RENVOYER-VAL  (nom-filtre  nom-inst-f)
(de  :APPARTIENT?  (nom-att  nom-filtre)
(de  :ADMIS?  (  val-a-tester  nom-filtre  nom-att)

(dmd  :INFERER?  (nom-filtre  nom-att  val-att-reseau)
(dmd  :EXISTE-FILTRE?  (val-att-reseau)
(dmd  :REF-INTERNE?  (nom-filtre  nom-att  val-att-reseau)
(dmd  :FILTRE-PLUS-HAUT-NIVEAU?  (nom-filtre)
(dmd  :RAZ-MARQUEUR  (nom-filtre  nom-inst-f)

-- in --  DB-filtre2.ll
(defvar  #:sys-package:colon  '#:shirka:moteur:filtrage)
(DE  :ADMIS-CONST-A-VERIF?  (val-a-tester  nom-filtre  nom-att  )
(DE  :INFERENCE-FILT  (val-att-reseau  nom-filtre  nom-inst-f  nom-att)
(DE  :ERREUR  ()  (PRINT  'ERRRRRRRRREUR....))
(DEFVAR  %var-fixe-simple  0)
(DEFVAR  %var-fixe-app  1)
(DEFVAR  %var-semifixe-simple  2)
(DEFVAR  %var-semifixe-app  3)
(DEFVAR  %var-result  4)
(DEFVAR  %var-result-imb  5)
(DEFVAR  %var-interne  6)
(DE  :INSTANCIER-SEMI-FIXES  (nom-inst-filtre  nom-filtre)
(DE  :OBT-VAL-ATT-SEMI-FIXES  (val-att-reseau  nom-inst-filtre  nom-filtre
(DE  :ESSAI-VAR  (val-att-reseau  nom-filtre  nom-att)
(DE  :ESSAI-SIB-EX  (val-att-reseau  nom-filtre  nom-att)
(DE  :SIBEX1  (l-sib-exec  nom-filtre)
(DE  :ESSAI-SIB-CHOIX  (val-att-reseau  nom-filtre  nom-att)
(DE  :ESSAI-DEFAUT  (val-att-reseau  nom-filtre  nom-att)
(DE  :FILT-CONST-ANC  (nom-instance  nom-schema)
(DE  :FILTRAGE-CONSTANT  (  nom-instance  nom-schema  )
(DE  :FILT-CONST-INST  (  nom-instance  nom-filtre  )
(DE  :ADMIS-SPEC-LOC?  (  sch-spec  nom-instance)
(DE  :ADMIS-CONST-SPEC?  (val-a-tester  sch-loc  nom-att  )
(DE  :SUP-INST-FILT-ANC  (nom-instance  nom-schema)
(DE  :SUP-INST-FILT  (nom-instance  nom-schema  )
(dmd  :RECH-INST-FILT  (nom-inst  nom-filtre)
(dmd  :CHERCHER-FILTRES  (  nom-schema  )
(dmd  :OBT-SCH-CIBLE  (nom-filtre)
(dmd  :OBT-TYPE-ATT-CIBLE  (nom-filtre  nom-schema)
```

```
-- in --   FR-fcts-commandes.ll
(defvar  #:sys-package:colon  '#:shirka:top)
(de  :aj-val  (inst)
(de  :lister-noms  (liste)
(de  :impr-res  (valeur)
(de  :sup-val  (inst)
(de  :mod-val  (inst)
(de  :creer-instance  (inst)
(de  :visu-inst  (inst)
(de  :impr-l-spec-t  (liste-spec  nb-niveaux  niveau-courant  marge-g)
(de  :liste-spec  (inst)
(de  :liste-inst  (inst)
(de  :obtenir-valeur  (inst)
(de  :supprimer-inst  (inst)
(de  :liste-att  (inst)
(de  :classer  (inst)
(de  :specialiser-inst  (inst)
(de  :classer-inst  (inst)
(de  :rattacher-inst  (inst)
(de  :nettoyage  (inst)
(de  :explication  (inst)
(de  :impr-l-spec-t-expl  (liste-spec  nom-instance  marge-g)
(de  #:shirka:top:expl-globale  (inst)
(de  :v?  (inst)
(de  :vi-c  (inst)
(de  :a-v  (inst)
(de  :m-v  (inst)
(de  :s-v  (inst)
(de  :executer  (inst)
(de  :info  (inst)
(de  :info-att  (inst)
(de  :tableur  (inst)
(de  #:shirka:top:charger-fic  (inst)
(de  #:shirka:top:trad-sch  (caract  bsch-inst)
(de  #:shirka:top:facette?  (nom-fac)
(de  #:shirka:top:erreur-trad  (numero)
(de  :sauver-inst  (inst)
(de  :demander  (inst)
(de  :ranger-valeur  (inst)

-- in --   FR-interf-util.ll
(defvar  #:sys-package:colon  '#:shirka:top)
(de  shirka  ()
(de  :creer  (schema  pour-classes  sauf-classes  util-par  bprem-niv)
(de  :dem-nom-ssclasse  (schema)
(de  :dem-nom-inst  (nom-sch  bprem-niv)
(de  :dem-valeur  (schema  nom-att  instance  pour-classes  sauf-classes
(de  :lecture  ()
(de  :commande  (nom-sous-systeme  nom-prompt)
(de  :dem-nom-com  (nom-sous-systeme)
(de  :msg  (message)
(de  :aff-nom-classe  (nom-classe)

-- in --   FR-meta.ll
(defvar  #:sys-package:colon  '#:shirka:comp)
(defvar  %prem-att  3)
(defvar  %est-un  0)     ;lien est-un d'instance a classe
(defvar  %util-par  1)  ;lien util-par
(defvar  %nom-sch  2)   ;nom d'un schema de classe
(defvar  %nom-att  3)   ;nom d'attribut
(defvar  %nom-inst  2)  ;nom d'une instance
(defvar  %list-att  3)  ;liste des attributs d'un schema
(defvar  %list-fac  4)  ;liste des facettes d'un attribut
```

```
(defvar %nat 5)        ;nature d'un attribut: $un ou $liste-de
(defvar %type 6)       ;type d'un attribut
(defvar %dom  7)       ;domaine d'un attribut
(defvar %int  8)       ;intervalle d'un attribut
(defvar %sauf 9)       ;sauf d'un attribut
(defvar %list-val 3) ;liste des valeurs d'une facette
(defvar %sorte-de 4) ;lien sorte-de de classe a classe
(defvar %spec 5)       ;liste des specialisations d'une classe
(defvar %inst 6)       ;liste des instances d'une classe
(defvar  %liste-toutes-facettes
(defvar  %liste-F-multi
(defvar  %liste-F-var
(defvar  %liste-F-type
(defvar  %liste-F-card
(defvar  %liste-F-mono-multi
(defvar  %liste-F-intervalle
(defvar  %liste-F-fil-att
(de #:exp:init-exp () )
(defvar %taille-att 10)
(defvar %taille-fac 4)   ;car nom n'est pas instanciable dans les
(de :ext-int ((nom-sch . attributs))
(de :traiter-liste-reprise (nom-sch)
(de :verif-insts (l-nom-inst schema)
(de :verif-inst (nom-inst schema l-att)
(de :sup-filt-spec-loc (schema)
(de :trad-l-att (meta-sch schema attributs)
(de :sort-sh (fn l)
(de :ffusion-sh (l1 l2)
(de :trad-l-fac (bliste-de facettes schema)
(de :compar-fac (fac1 fac2)
(de :trad-l-val (bliste-de nom-fac valeurs schema)
(de :detacher-filt (filtres)
(de :detacher-spec-loc (spec-loc)
(de :sch-modif (schema nouv-nom)
(de :detacher-inst (inst nom-type)

-- in --  FR-moteur.ll
(defvar #:sys-package:colon '#:shirka:moteur)
(de :admissible? (valeur nom-att schema instance )
(de :test-admissible (valeur nom-att schema instance)
(de :a-verifier? (valeur nom-att schema instance)
(de :a-verifier-anc (valeur nom-att schema nom-inst)
(de :verif-lui-meme (schema instance)
(de :reflexe   (schema nom-att nom-inst nom-reflexe)
(de :aj-valeur (instance nom-att valeur bfin)
(de :creer-est-un (instance nom-sch)
(de :creer-sorte-de (schema peres)
(de :sup-valeur (instance nom-att valeur)
(de :mod-valeur (instance nom-att valeur2 valeur1)
(de :sup-inst (instance bsup-spec)
(de :creer-instance (nom-classe liste-valeurs nom-inst)
(de :spec-inst? (instance nom-sch-dest)
(dmd :aj-util-par (instance nom-objet)
(dmd :eq-val (val1 val2 type)
(dmd val?    (nom-att)
(dmd affect (nom-att valeur)

-- in --  FR-noyau.ll
(defvar #:sys-package:colon '#:shirka:noyau)
(de :rec (l-vec nom rang)
(de :rec-fac (l-fac nom-fac)
(de :classe-de (nom-inst)
(de :l-nom-att (schema)
```

```
(de :l-nom-att-class (schema classes-att sauf-classes-att)
(de :considerer-classe-att (type-att pour-types sauf-types)
(de :classe-att? (type-att liste-classes-att)
(de :facette (schema nom-att nom-fac)
(de :val-fac (attribut nom-fac)
(de :val-facette (schema nom-att nom-fac)
(de :ramener-anc (schema)
(de :demarquer (schema)
(de :attrib-he (schema nom-att)
(de :obtenir-att (schema nom-att)
(de :val-fac-he (schema nom-att nom-fac)
(de :ramener-fac (schema nom-att nom-fac)
(de :val-fac-he-sch (schema nom-att nom-fac)
(de :ramener-fac-sch (schema nom-att nom-fac)
(de :obtenir-nature (schema nom-att)
(de :ramener-nat (schema nom-att)
(de :nature (schema nom-att)
(de :union (l1 l2)
(de :ramener-att (schema)
(de :l-nom-att-he-class (schema classes-att sauf-classes-att)
(de :ramener-att-class (schema classes-att sauf-classes-att)
(de :sorte-de? (nom-sch-inf nom-sch-sup)
(de :l-nom-att-inst-he (schema)
(de :ramener-att-inst (schema)
(de :l-nom-att-inst-he-class (schema pour-classes sauf-classes)
(de :ramener-att-inst-class (schema pour-classes sauf-classes)
(de :pos-nom-att (schema nom-att)
;(de nature-att (nom-att)
(de #:shirka:noyau:classe-att (schema nom-att)
(de #:shirka:noyau:classe-att-he (schema nom-att)
(de :chercher-classe-att (schema nom-att)
(de :demarquer-bis (schema)
(de :sch-classe? (schema)
(de :l-sch-entre (nom-sch-inf nom-sch-sup)
(de :app-inter? (valeur type l-inter)
(de :app-dom? (valeur type domaine)
(de :app-sauf? (valeur type sauf)
(de :egal-inst? (inst1 inst2)
(de :ramener-int (schema nom-att type)
(de :comb-inter (inter1 inter2 type)
(de :ramener-dom (schema nom-att type)
(de :comb-dom (domaine1 domaine2 type)
(de :ramener-sauf (schema nom-att type)
(de :nouv-att? (schema nom-att)
(de :exist-type (schema nom-att)
(de :l-spec-t (schema)
(de :l-spec-t1 (schema)
(de :l-spec-t-vraies (schema)
(de :l-spec-t-vraies1 (schema)
(de :l-inst-t (schema)
(de :l-inst-t1 (schema)
(de :demarquer-fils (schema)
(de :make-inst (nom-schema nom-inst)
(dmd :meta? (nom-att)
(dmd :schema (nom-sch)
(dmd :nom-sch (schema)
(dmd :l-att (schema)
(dmd :attrib (schema nom-att)
(dmd :nom-att (attribut)
(dmd :l-fac (attribut)
(dmd :fac (attribut nom-fac)
(dmd :peres (schema)
(dmd :ancetres (schema)
```

```
(dmd :l-nom-att-he (schema)
(dmd :position (nom-att)
(dmd :val-inst (instance pos-att)
(dmd :att-inst? (schema nom-att)
(dmd :type-simple? (type)
(dmd :tr-inter (inter)
(dmd :intervalle (schema nom-att type)
(dmd :domaine (schema nom-att type)
(dmd :sauf (schema nom-att type)
(dmd :l-spec (schema)
(dmd :l-inst (schema)

-- in -- FR-vi.ll
(defvar #:sys-package:colon '#:shirka:vi)
(de :visualiser (instance pour-classes sauf-classes niveau binf)
(de :appel-meth-vi (nom-inst schema nom-att nom-fac)
(de :vi-att (instance pour-classes sauf-classes schema nom-att niveau
(de :vi-valeurs (nom-inst pour-classes sauf-classes schema nom-att
(de :vi-val (schema pour-classes sauf-classes nom-att valeur type nivea

-- in -- Installer-Shirka.1.ll
(defvar #:sys-package:colon '#:shirka:installer)
(de :sort-sh (fn l)
(de :ffusion-sh (l1 l2)
(defvar %prem-att 3)
(defvar %est-un 0)    ;lien est-un d'instance a classe
(defvar %util-par 1) ;lien util-par
(defvar %nom-sch 2)  ;nom d'un schema de classe
(defvar %nom-att 3)  ;nom d'attribut
(defvar %nom-inst 2) ;nom d'une instance
(defvar %list-att 3) ;liste des attributs d'un schema
(defvar %list-fac 4) ;liste des facettes d'un attribut
(defvar %nat 5)       ;nature d'un attribut: $un ou $liste-de
(defvar %type 6)      ;type d'un attribut
(defvar %dom  7)      ;domaine d'un attribut
(defvar %int  8)      ;intervalle d'un attribut
(defvar %sauf 9)      ;sauf d'un attribut
(defvar %list-val 3) ;liste des valeurs d'une facette
(defvar %sorte-de 4) ;lien sorte-de de classe a classe
(defvar %spec 5)      ;liste des specialisations d'une classe
(defvar %inst 6)      ;liste des instances d'une classe
(de :def-sch (nom description)
(defvar %taille-att 10)
(defvar %taille-fac 4)  ;car nom n'est pas instanciable dans les
(de :creer-est-un (instance nom-sch)
(de :creer-sorte-de (schema pere)
(de :ext-int ((nom-sch . attributs))
(de :trad-l-att (meta-sch schema attributs)
(de :trad-l-fac (bliste-de facettes schema)
(de :compar-fac (fac1 fac2)
(de :l-spec-t (schema)
(de :l-spec-t1 (schema)
(de :demarquer-fils (schema)
(de :trad-l-val (bliste-de nom-fac valeurs schema)

-- in -- Installer-Shirka.2.ll
(defvar #:sys-package:colon '#:shirka:installer)

-- in -- JLA-def-sh.ll
(defvar #:sys-package:colon '#:shirka:comp)
(de :gen-symbole (nom)
(defvar %l-fac-filtre
(defvar %l-sch-tr nil)
```

```
(defvar %nom-schema-en-cours nil)
(de def-sh (schema)
(de :ram-nouv-att (schema)
(de :compilation (schema)
(de :compilation-f (nom-filtres)
(de :const-reseau (filtre)
(de :const-vect (attribut filtre)
(de :ext-part-const (attribut)
(de :ext-part-var (nom-att nom-filtre)
(defvar %const 0)
(defvar %var 1)
(defvar %sib-filt 2)
(defvar %sib-exec 3)
(defvar %sib-choix 4)
(defvar %defaut 5)
(defvar %a-verif 6)
(defvar %type-att 7)
(defvar %pos-att 8)
(dmd :val-reseau (nom-filtre)
(dmd :val-att-reseau (nom-filtre nom-att)
(dmd :val-fac-att-reseau (val-att-reseau nom-fac)
(dmd :val-fac-reseau (nom-filtre nom-att nom-fac)

-- in -- JLA-erreurs.ll
(de #:shirka:erreur:traiter-err (schema-erreur err-def-sh)

-- in -- JLA-moteur-inf.ll
(defvar #:sys-package:colon '#:shirka:moteur)
(defvar #:exp:trace nil)
(de :valeur? (%%nom-att nom-inst)
(de :moteur (nom-att schema instance nature)
(de :essayer-peres (nom-att schema instance nature)
(de :trouver-val (attribut schema instance nature)
(de :reusite (val schema)
(de :obt-val-ref? (ref)
(de :obt-val-ref1? (ref)
(de :test-adm-lval (valeurs nom-att schema instance)
(de :sib-exec (nom-filtres schema nom-att nom-inst)
(de :traiter-se-sans-filtr (filtre schema nom-fct)
(de :traiter-se-simple (filtre schema nom-fct)
(de :traiter-se-complexe (filtre schema nom-fct)
(de :obt-val (l-inst filtre nom-fct)
(de :lancer-filtrage (filtre inst-filtre)
(de :instantier (schema)
(de :instancier (schema)
(de :cherch-val-renv (nom-filtre inst-filtre)
(de :obt-marqueur-se (filtre)
(de :exploser-inst (instance l-att-inst)
(de :exploser-att (l-inst valeurs index)
(de :const-list (nb-copies inst)
(de :attach-proc (nom-filtres)
(de :essayer-attach-proc (filtre)
(de :a-verif (nom-filtres)
(de :essayer-a-verif (filtre)
(dmd :lier (nom-sch nom-inst)
(dmd :make-w-inst (nom-sch)
(dmd :delier (nom-sch)
(dmd :obt-ref-var (nom-att schema)
(dmd :ref-simple? (ref)
(dmd :instance? (nom-sch)
(dmd :aj-inst (schema nom-inst)
(dmd :obt-att-dem (nom-filtre)
```

```
-- in --   JLA-sib-filtre.ll
(defvar  #:sys-package:colon  '#:shirka:moteur:filtrage)
(de  :sib-filtre  (l-nom-filt  schema  nature)
(de  :sib-filtre1  (l-nom-filt  nature)
(de  :inst-filt  (nom-filt)
(de  :obt-marqueur  (nom-filt)
(de  :creer-inst-vide  (nom-sch  nb-att)
(de  :inst-att-fixes  (nom-inst  nom-filt  ordre-att)
(de  :obt-val-att-fixes  (val-att-reseau  nom-inst  nom-filt  nom-att)
(de  :res-ref-var  (val-fac-var  nom-inst  nom-filt  nom-att)
(de  :ref-var-type  (type  val-fac-var)
(de  :obt-val-ref  (ref)
(de  :obt-val-ref-f  (nom-inst-f  ref)
(de  :essayer-def  (nom-filt  nom-att)
(de  :f-imbr  (f-imb)
(de  :valeur-att?  (nom-inst-tr  nom-att)
(de  :valeur-att-f?  (nom-inst-f  nom-att)
(de  :filtre?  (schema)
(de  :pos-att-filtre?  (nom-f  nom-att)
(de  :aj-valeur-inst-filtre  (val-att  nom-att  nom-inst-f)
(dmd  :ref-fixe?  (val-fac-var)
(dmd  :inst-cour  (nom-sch)
(dmd  :val-const?  (val-fac-const)
(dmd  :val-att?  (nom-inst  pos-att)
(dmd  :aj-val-inst  (val-att  pos-att  nom-inst)

-- in --   JLA-tr-var.ll
(defvar  #:sys-package:colon  '#:shirka:comp)
(de  :tr-var  (schema)
(de  :ram-var-def  (l-attribut)
(de  :spec-loc  (att)
(de  :tr-var-ass  (nom-sch  attribut)
(de  :tr-var-att  (nom-sch  nom-att  att-ref  type-var)
(de  :res-ref  (att-ref  type-var)
(de  :chercher-ref  (nom-sch  att-ref)
(de  :constr-ref  (schema  ref  type-var)
(de  :obt-nature  (ref)
(de  :tr-var-dem  (nom-sch  attribut)
(de  :niv-ref  (nom-sch)
(de  :ref-int?  (nom-att  l-var-ass)
(de  :tr-var-fac-filtre  (attribut)
(de  :typer-se  (nom-f)
(de  :det-ref-c  (lass)
(de  :constr-dep  (nom-filtre)
(de  :constr-dep-att  (nom-att  l-dep)
(de  :elim-doubles  (liste)
(de  :ord-dep  (liste-dep)
(de  :obt-nouv-dep  (elem  sources)
(de  :elim-sources  (l-elem  sources)
(de  :obt-sources  (l-dep)
(de  :obt-ord-inst-att  (schema)
(de  :obt-ord-inst-att-he  (schema)
(dmd  :obt-var-def  (att)
(dmd  :spec-loc?  (nom-sch)

-- in --   P3A-comd2.ll
(defvar  #:sys-package:colon  '#:shirka:tabl)
(de  :SCROLL_HOR  (droite)
(de  :SCROLL_VERT  (bas)
(de  :SUBST_ATT2  ()
(de  :SUBST_INS  ()
(de  :SCROLL_CASE  (  liste_inferee  )
(de  :CURSEUR  (mode)
```

```
(de  :DEPL_CUR_2  ()

-- in --   P3A-interf.ll
(defvar  #:sys-package:colon  '#:shirka:tabl)
(de  :DESCENDRE  ()
(de  :REMONTER  ()
(de  :T_LECTURE  (type  icol  ilig  nbr)
(de  :CONV_ENT  (l)
(de  :CONV_REEL  (l)
(de  :GEST_COMMANDES  (cle)
(de  :HELP  ()

-- in --   P3A-tabl.ll
(defvar  #:sys-package:colon  '#:shirka:tabl)
(de  :TAB_2  (nom_sch)
(de  :INIT_GENERALE   ()
(defvar  %MAXCOL   (1-  (DIV  (-  %XMAX  14)  13)))
(defvar  %MAXLGN1  (1-  (DIV  (-  %YMAX  11)  3)))
(defvar  %MAXLGN2  (1-  (DIV  (-  %YMAX  3)  5)))
(defvar  %sch_cour  nil)
(defvar  %sch_source  nil)
(defvar  %list_sch  nil)
(defvar  %explore_en_bas  t)
(defvar  %oldlig  nil)
(defvar  %oldcol  nil)
(de  :INIT_%ECRAN2  ()
(de  :tr-spec-loc  (nom-classe)
(de  :INIT_TAB2  ()
(defvar  %colonne  0)
(defvar  %ligne  0)
(de  :TABL2  ()
(de  :TABLEUR  (inst)
(de  :CALC_LIGNE2  (num)

-- in --   P3A-term-mac.ll
(de  getfnum  (nom)
(de  tyattrib  (i)
(de  tydelln  ()
(defvar  %left  28)
(defvar  %right  29)
(defvar  %up  30)
(defvar  %down  31)
(de  tyi-moi  ()  (tyi))
(de  VISUAL_CURSOR  (mode)  nil)
(de  CLIGNO_CURSOR  (mode)  nil)
(de  CLIGNO_TEXTE   (mode)  nil)
(de  SOULIGN_TEXTE  (mode)  nil)

-- in --   P3A-util.ll
(defvar  #:sys-package:colon  '#:shirka:tabl)
(de  :ECHANGE  ( tabl l m )
(de  :NIVEAU_2  ( l )
(de  :FORMAT  (str nbr)
(de  :INIT_VAR_STR  ()
(de  :GET_KEY  ()
(de  :INIT_DES_ATT  ()
(de  :CREAT_LATT  (sch)
(de  :INIT_DES_VIS  ()
(de  :INIT_DES_INS  ()
(de  :CREAT_LINS  (sch)
(de  :TABL_ATT_TYP  (sch)
(de  :COUPLES_ATT_TYP  (schema  l_att)
(de  :TABL_COUR  ()
```

```
(de :ECH_ATT (sch n p)
(de :ECH_INS (sch n p)
(de :MAJ_LIGNE_1 (sch inst_suiv)
(de :MAJ_COLONNE_1 (sch att_suiv)
(de :EMPILE_IMB (sch num_inst inst)
(de :DEPILE_IMB (sch num_inst)
(de :INDICE_LIGNE_1 (sch)
(de :INDICE_COLONNE_1 (sch)
(de :VISUAL_INST (sch num_inst)
(de :DECALER ( i j )
(de :SCHS_FILS (sch)
(de :SCHS_PERES (sch)
(de :SCH_LGN (ligne)
(de :VAL_ATTCOUR ()
(de :ATTCOUR (col lgn)
(de :LIGNE_COURANTE ()

-- in -- PP-classif1.ll
(defvar #:sys-package:colon '#:shirka:classif)
(defvar %pour-types nil)
(defvar %sauf-types nil)
(defvar %binf nil)
(defvar %bdial nil)
(de :instpp-de? (instance classe pour-types sauf-types bguide binf
(de :allonge (nom-inst nom-sch nom-att classe)
(de :crebid-cl-inst (instance)
(de :assoc-schema (instance)
(de :spec-inst (instance nom-sch-dest pour-types sauf-types binf bdial)
(de :specialiser-vers (nom-inst nom-sch-dest
(de :trait-sur (nom-inst nom-sch)
(de :trait-poss (nom-inst nom-sch)
(de :trait-imposs (nom-inst nom-sch)
(de :marquer-sur-anc (schema nom-inst)
(de :sur-peres? (nom-sch nom-inst)
(de :filles (schema)
(de :class-inst (instance nom-sch-tete pour-types sauf-types bguide bin
(de :impression ((li-sur li-pos li-imp))
(de :chef-fam (nom-sch)
(de :chef (nom-sch)
(de :prop-inst (nom-inst nom-sch)
(de :expl-cl (nom-inst nom-sch)
(de :rech-expl (nom-inst nom-inst-bid lis-classes)
(de :trait-expl (nom-inst nom-inst-bid nom-sch)
(de :affexpl (nom-inst nom-inst-bid nom-sch)
(de :ecr-int (inter)
(de :ecr-dom (dom)
(de :ecr-expl (nom-inst nom-sch nom-att val prop dom)
(de :fin-cl (liste-instances)
(de :sup-prop (nom-sch l-inst-bid)
(de :sup-instance (instance)
(de :ratt-inst (instance nom-sch-dest)
(de :cr-inst-temp (instance nom-sch-dest nom-inst-temp)
(de :dem-val (schema classe nom-att instance bguide branger)
(de :traiter-ilot (nom-ilot bliste)
(de :lire-reponses (liste-classes)
(dmd :nom-inst (instance)

-- in -- PP-classif2.ll
(defvar #:sys-package:colon '#:shirka:classif)
(de :verif-val (nom-inst nom-sch nom-att valeur classe dom)
(de :obtien-val (nom-inst nom-sch nom-att classe binf bdial)
(de :infer-val (schema instance  classe attribut nature %%nom-att)
(de :comb-i-d (i d type)
```

```
(de :dom-adm (schema nom-att classe)
(de :pintervalle (schema nom-att type)
(de :c-inter (inter type)
(de :pramener-int (schema nom-att type)
(de :pcomb-inter (inter1 inter2 type)
(de :pdomaine (schema nom-att type)
(de :pramener-dom (schema nom-att type)
(de :c-dom (l-domaine type)
(de :pcomb-dom (domaine1 domaine2 type)
(de :simpl-inter (l-inter type)
(de :simpl-int (l-inter type)          ;l-inter trie par min croissants
(de :tri-int (l-inter type)
(de :inf-min-ent (int1 int2)
(de :inf-min-reel (int1 int2)
(de :inf-min-chaine (int1 int2)
(de :super (x y type)      ; x > y
(de :app-inter? (valeur type l-inter)
(de :union-int-dom (schema nom-inst nom-att bguide)
(de :verif-lui-meme (schema instance)
(de :instancier (schema)
(de :v? (nom-inst nom-att)
(de :vi-c (nom-inst)
(de :a-v (nom-inst nom-att valeur)
(de :m-v (nom-inst nom-att valeur nouv-valeur)
(de :s-v (nom-inst nom-att valeur)

-- in --  Shirka
(defvar install-disk "/softs/SHIRKA/" )
(defvar #:shirka:*code*   (catenate install-disk "shirka-iace/"))
(defvar #:shirka:*bases* (catenate install-disk "shirka-iace/BasesShirk
;;;(defvar #:shirka:*verif* (catenate #:shirka:*bases* "basesverif/"))
(defvar #:shirka:*verif* "/tmp/SHIRKA/basesverif/"))
(defvar #:ivan:*home* (catenate install-disk "IVAN/"))
;;;(defvar #:shirka:verif:format-v 'a4-portrait)
(defvar #:shirka:verif:format-v 'a4-paysage)

-- in --  kashir.ll
(defvar %liste-reprise nil)
(de #:exp:expl (inst)
```

Commande “verifier”

☞ **verifier <base>**

permet de vérifier syntaxiquement les schémas Shirka et d'opérer un pré-traitement de macro-commandes insérées dans la base. Cette commande est utile principalement lorsqu’on importe une nouvelle base, donc non “fiable” a fortiori ou encore une base qui vient d’être éditée pour la première fois. La commande de chargement n’effectuant presque pas de tests de validité syntaxique, il est donc intéressant de lancer une telle vérification avant de charger ce type de bases.

- **<base>** est le nom d'une base de schémas, c’est-à-dire d’un fichier texte dans lequel figurent des définitions de schémas et (éventuellement) des fonctions Lisp. Ce fichier doit obligatoirement se situer dans le catalogue (ou "dossier") destiné à recevoir les bases Shirka et dont l'accès a été précisé lors de l’installation (§1).

La commande **verifier** appelle un "vérifieur-formateur", qui analyse en une passe une base Shirka; les erreurs syntaxiques détectées sont indiquées dans un fichier nommé "**liste**", placé dans le catalogue nommé par la variable **#:shirka:*verif***, initialisée lors de l’installation (§1). Une copie formatée de **<base>** est écrite dans ce catalogue.

Le formatage est prédéfini suivant deux présentations: feuille A4 standard, nommée “a4-portrait”, et la présentation “à l'italienne”, nommée 'a4-paysage. Cette dernière option permet de visualiser plus clairement des bases possédant de nombreuses imbrications de schémas. Dans l'environnement Macintosh, il est très facile de passer de l'un à l'autre mode: il suffit de choisir l'option d'impression correspondante (option "Format d'impression" dans le menu "Fichier"). Dans l'environnement Unix, il faut disposer d'une imprimante 120 colonnes (et au-delà), ou bien utiliser une option d’impression capable de gérer ce format (par exemple option ‘p’ de la commande ‘a2ps’ qui sait générer du Postscript). Par défaut, le formatage est initialisé dans le mode dit "A4-portrait"; pour passer dans l'autre mode, il suffit de commuter dans le fichier d'installation "**Shirka**", l'affectation de la variable globale **#:shirka:verif:format-v** associée au formatage actif. D'autres modes peuvent être installés, en intervenant directement dans la fonction Le-Lisp dédiée à cette initialisation: "**PF-formats.ll**".

Insertion dynamique de fichiers:

Comme indiqué ci-dessus, le vérifieur contient un interpréteur de macros-commandes, non présentes dans la syntaxe standard de Shirka. Dans la version actuelle, seule la commande d'inclusion est présente: elle permet d'indiquer au pré-processeur qu'il lui faut inclure des lignes source contenant des schémas (voire des définitions de fonctions lisp, avec une réserve indiquée plus loin, dans les **remarques** de la liste), à l'endroit où l'utilisateur a placé cette commande d'insertion.

Syntaxes:

```
;<INCLUDE>  "accès-COMPLET-sur-fichier-a-insérer-ici"
;<include>  " -idem- "
```

Exemple (version Mac):

```
; Inserer ici la fonction 1:
;<include>  "mon-disque:fichiers-externes:fonc-1.ll"
; Inserer les schemas communs aux bases:
;<include>  "mon-disque:Shirka:Bases-Shirka:schemas-communs"
```

ATTENTION : il est impératif de placer la commande **<INCLUDE>** *immédiatement après* le point-virgule (sans blancs!), sinon l'automate de décodage considère qu'il s'agit d'une *ligne de commentaires*. Les lignes "**include**" sont retranscrites pour mémoire dans le fichier résultat, mais sans les délimiteurs **< >**.

Commentaires:

• l'imbrication maximale des commandes d'insertion de fichiers est de 7;

• le fichier résultat, formaté et placé dans le catalogue des bases vérifiées, contient la TOTALITE des lignes ainsi concaténées et peut par conséquent être passé au chargeur Shirka.

Un exemple complet:

Voici la trace de deux appels de la commande '**verifier**' sur un même exemple contenant deux schémas Shirka et un appel **include**, adressant un fichier contenant une seule fonction Le-Lisp, placée dans le même dossier que les bases Shirka.

La première vérification s'est effectuée avec l'option de formatage <**A4-portrait**>, la deuxième, après un changement manuel sous Le-Lisp de la variable globale **#:shirka:verif:format-v**, montre l'option <**A4-paysage**>.

La liste de vérification syntaxique contient trois messages "ATTENTION", qui ne sont pas comptés comme erreur par le vérifieur. Ils indiquent, dans cet exemple, que l'automate de lecture a rencontré du code source Le-Lisp et que des parenthèses fermantes en excès ont été détectées. L'ensemble des messages est fourni en annexe.

1/ la base shirka :

```
; base avec-du-lisp.sh : source original dans "Bases-shirka:"
;
{ tache-scai sorte-de   = objet;
est-un     = tache;
}
;
;<include>  "OUROBOROS:Shirka:Bases-shirka:erreur-G.ll"

{T-laisse-passer
sorte-de   =         tache-scai;
est-un     =         tache-term;
att1 $un entier;
att2 $liste-de reel;
&entree l-seq-e      $liste-de entier
$var-nom    l-s-e;
&entree l-res-e      $liste-de entier
$var-nom    l-r-e;
&sortie l-seq-s      $liste-de entier
$var-nom    l-s-s;
&sortie l-res-s      $liste-de entier
$var-nom    l-r-s;
proc       =   { sorte-de = proc-laisse-passer ;
ent-l-s   $var<-   l-s-e ;
sort-l-s  $var->   l-s-s ;
ent-l-r   $var<-   l-r-e ;
sort-l-r  $var->   l-r-s }   }
```

Commentaires:

On remarquera le format complètement libre en entrée: aucun blanc n'est présent en début de ligne source, aucune tabulation n'est employée.

La police de caractères employée est une police à chasse constante, définie dans le fichier d'installation de LeLisp.

Le formateur ne fonctionne correctement qu'avec ce type de polices, puisqu'il insère des blancs pour effectuer ses tabulations. La relecture, au moyen d'un éditeur de texte, d'une base ainsi formatée, en utilisant une police proportionnelle (type "Times") ne donnera pas un résultat très esthétique!

2/ la liste produite après le lancement de `verifier` :

NB: la création de la liste est en accord avec le format demandé: pour cet exemple, le format `a4-portrait`; les lignes sources qui dépassent la marge droite sont repliées dans une (ou plusieurs) lignes de continuation, repérées par l'absence de compteur de ligne.

```
     1 : ; base avec-du-lisp.sh
     2 : ;
     3 : { tache-scai sorte-de   = objet;
     4 : est-un     = tache;
     5 : }
     6 : ;
     7 : ;<include> "OUROBOROS:Shirka:Bases-shirka:erreur-G.ll"
     8 : (de :erreur-G (message err-triplet)
*** ATTENTION : ( [ debut lisp...]
     9 :       (let ((nerrs-Gasel (getprop ':erreur-G '#/*)) )
    10 :       #| illustration d'un bug vicieux produit par l'analyse de la
    11 :         sequence <diese>/<cara> : cette syntaxe n'est pas (encore)
    12 :         prevue dans la lexico, ainsi que les commentaires <dieze-barre>
    13 :         Si <cara> est une parenthese, ou si le commentaire contient des
    14 :         parentheses non equilibrees, alors ...bug!
    15 :      |#
    16 :          (ifn message nerrs-Gasel
    17 :               (incr nerrs-Gasel)
    18 :               (putprop '(1 2 3 (on empile)) 'nerrs)
    19 :               (:listing "***" 'err)
    20 :               (:listing (catenate "*** ERREUR : "
    21 :                           (string (explode err-triplet))) 'err)
    22 :               (:listing (catenate "***            " message ".") 'err)))
    23 :      )
*** ATTENTION : [...fin lisp ] )
    24 : ))
         !!
*** ATTENTION : ( ')' superflues!)
    25 : ; parentheses droites en trop : l'analyseur doit les tolerer...
    26 : ; fin erreur-G
    27 :
    28 :
    29 :
    30 :
    31 : {T-laisse-passer
    32 : sorte-de   =        tache-scai;
    33 : est-un     =        tache-term;
    34 : att1 $un entier;
    35 : att2 $liste-de reel;
    36 : &entree l-seq-e     $liste-de entier
    37 : $var-nom    l-s-e;
    38 : &entree l-res-e     $liste-de entier
    39 : $var-nom    l-r-e;
    40 : &sortie l-seq-s     $liste-de entier
    41 : $var-nom    l-s-s;
    42 : &sortie l-res-s     $liste-de entier
    43 : $var-nom    l-r-s;
    44 : proc       =   { sorte-de = proc-laisse-passer ;
    45 : ent-l-s   $var<-   l-s-e ;
    46 : sort-l-s $var->   l-s-s ;
    47 : ent-l-r   $var<-   l-r-e ;
    48 : sort-l-r $var->   l-r-s }   }
    49 :
    50 :
    51 :
    52 :
```

```
   53 :
   54 :
   55 :
   56 :
   57 :

Nombre de schemas et sous-schemas analyses: 2


       : ...STOP... 0 erreur(s)
```

REMARQUES

• la liste est une image fidèle du (des) fichier(s) source (aux repliements de ligne près);

• les messages d'erreur sont précédés par une ligne de marqueurs '!' qui se positionnent sur les unités syntaxiques fautives (mais des tabulations dans le source peuvent perturber cet alignement). Il y a trois types d'erreurs, associés à un "<message>":

1/ **ATTENTION** "**message d'information**";
aucune influence sur la lecture;
2/ **ERREUR** "**erreur lexicale ou syntaxique**";
la vérification reprend sur une entité "fiable" comme ';' (ou '}')
pour dépiler correctement l'attribut (schéma) fautif;
3/ **FATALE** "**erreur fatale dans l'analyse, ou erreur interne**";
la lecture de la base s'arrête immédiatement)

• les lignes vides (par ex. [49..57] dans la liste) sont ignorées par l'analyseur, ainsi que les commentaires Lisp placés entre les définitions de schémas;

• le commentaire placé dans la fonction lisp (entre **#|** **|#**) indique une possibilité d'erreur de comptage des parenthèses, dans un cas particulier. Cette configuration est liée au non traitement des *#-Macros* de Le-Lisp (en dehors des noms de package **#:**). L'automate lexical, dans sa version actuelle, n'effectue en effet qu'une lecture simplifiée du code lisp puisque son rôle n'est pas d'évaluer celui-ci; il est alors possible que des configurations lexicales complexes entrainent un décalage dans le comptage des parenthèses, seul indicateur fiable sur lequel l'automate se fonde pour détecter la fin du code. La liste contiendra alors une indication erronée:

```
*** ATTENTION : [...fin lisp ] )
```

… suivie évidemment par d'autres erreurs. L'utilisateur saura alors qu'il ne doit pas essayer d'insérer ces fichiers avec une commande **include** mais qu'il devra charger à part (en amont ou en aval), sous l'environ lisp, ces fonctions "à problème".

3/ la base formatée, mode A4-portrait, avec inclusion du code Le-Lisp :

```
;  base  avec-du-lisp.sh  (a4-portrait)
;

{   tache-scai
      sorte-de      = objet ;
      est-un        = tache ;

}
;
;include"OUROBOROS:Shirka:Bases-shirka:erreur-G.ll"

(de  :erreur-G  (message  err-triplet)
      (let  ((nerrs-Gasel  (getprop  ':erreur-G  '#/*))  )
      #|  illustration d'un bug vicieux produit par l'analyse de la
        sequence <diese>/<cara> : cette syntaxe n'est pas (encore)
        prevue dans la lexico, ainsi que les commentaires <dieze-barre>...
        Si <cara> est une parenthese, ou si le commentaire contient des
        parentheses non equilibrees, alors ...bug!
     |#
         (ifn message nerrs-Gasel
              (incr nerrs-Gasel)
              (putprop '(1 2 3 (on empile)) 'nerrs)
              (:listing "***" 'err)
              (:listing (catenate "*** ERREUR : "
                          (string (explode err-triplet))) 'err)
              (:listing (catenate "***            " message ".") 'err)))
    )
;  parentheses droites en trop : l'analyseur doit les tolerer...
;  fin erreur-G

{   T-laisse-passer
      sorte-de      = tache-scai ;
      est-un        = tache-term ;
      att1          $un             entier ;
      att2          $liste-de       reel ;
      &entree       l-seq-e
                    $liste-de       entier
                    $var-nom        l-s-e ;
      &entree       l-res-e
                    $liste-de       entier
                    $var-nom        l-r-e ;
      &sortie       l-seq-s
                    $liste-de       entier
                    $var-nom        l-s-s ;
      &sortie       l-res-s
                    $liste-de       entier
                    $var-nom        l-r-s ;
     proc          =
      {   sorte-de        = proc-laisse-passer ;
            ent-l-s       $var<-          l-s-e ;
            sort-l-s      $var->          l-s-s ;
            ent-l-r       $var<-          l-r-e ;
            sort-l-r      $var->          l-r-s
      }

}
```

Commentaires:

• Le code Lisp n'est pas reformaté: le vérifieur retranscrit simplement les lignes telles qu'il les lit dans le fichier d'insertion.

Index

Commandes

- 71
?? 70
??? 70
a-v <attribut> <valeur> <instance> 94
aj-val <instance> <attribut> <valeur> [<position>] 80
Appel de lisp + 72
Appel récursif de Shirka & 71
charger <base> 75
class <instance> <classe> <classes-att> <sauf-classes-att> [<guide>] [<inference>] 87
classif 83
cr-inst <classe> <classes-att> <sauf-classes-att> 77
def-sh <description> 98
exec <methode> 82
expl <instance> <classe> 91
expl-g <instance> <classe> 91
info <classes> 96
info-att <classe> [<attributs>] 96
init [<instances>] 94
l-att <classe> <classes-att> <sauf-classes-att> 76
l-inst <classe> 76
l-spec <classe> 76
m-v <attribut> <valeur> [<ancienne-valeur>] <instance>. 92
mod-val <instance> <attribut> <valeur-n> <valeur-a> 81
Poursuivre une instanciation > 72
ratt <instance> <classe> 90
s-v <attribut> [<valeur>] <instance> 93
sauver <classes> <base> 75
spec <instance> <classe> <classes-att> <sauf-classes-att> [<inference>] 86
spécialisation 92
sup-inst <instance> 78
sup-val <instance> <attribut> <valeur> 81
tabl <classe> [<matériel>] [<lignes>] [<colonnes>] 95
v? <attribut> <instance> 91
val? <instance> <attribut> <vu-comme> 80
Valeur(s) de l'appel précédent = et << 71
verifier <base> 75; 135
vi <instance> <vu-comme> <classes-att> <sauf-classes-att> [<niveaux>] [<inference>] 79
vi-c <instance> 92

Fonctions

def-sh 98
#:shirka:noyau:schema 98
#:shirka:noyau:nom-sch 99
#:shirka:moteur:valeur? et val? 99
#:shirka:moteur:aj-valeur et affect 99
#:shirka:moteur:mod-valeur 100
#:shirka:moteur:sup-valeur 100
#:shirka:moteur:sup-inst 101
#:shirka:moteur:creer-instance 101
#:shirka:vi:visualiser 101
#:shirka:classif:class-inst 102
#:shirka:noyau:peres <schema> 102
#:shirka:noyau:ancetres <schema> 102
#:shirka:noyau:l-nom-att-inst-he <schema> 103
#:shirka:noyau:pos-nom-att <schema> <nom-att> 103
#:shirka:noyau:val-inst <instance> <pos-att> 103
#:shirka:noyau:val-facette <schema> <nom-att> <nom-fac> 104
#:shirka:noyau:val-fac-he <schema> <nom-att> <nom-fac> 104
#:shirka:noyau:l-spec <schema> 104
#:shirka:noyau:l-spec-t <schema> 105
#:shirka:noyau:l-spec-t-vraies <schema> 105
#:shirka:noyau:l-inst-t <schema> 105
#:shirka:moteur:admissible? <valeur> <nom-att> <schema> <instance> 106
.#:shirka:top:dem-valeur <schema> <nom-att> <instance> <branger> ommande> 106
#:shirka:top:commande <nom-environnement> <prompt> 107
#:shirka:classif:spec-inst <instance> <nom-sch-dest> <binf> <bdial> 107
#:shirka:classif:ratt-inst <instance> <nom-sch-dest> 108
#:shirka:classif:fin-cl <liste-instances> 108