Groupe de travail Réseau
RFC 610
NIC n° 21352
Traduction Claude Brière de L'Isle

Richard Winter, Jeffrey Hill, Warren Greiff
CCA, Computer Corporation of America
15 décembre 1973

Éléments nouveaux du concept de langage des données

Remerciement
Durant le cours du projet Datacomputer, de nombreuses personnes ont contribué au développement du langage des données. Les suggestions et critiques du Dr. Gordon Everest (Université du Minnesota), du Dr. Robert Taylor (Université du Massachusetts), du professeur Thomas Cheatham (Université Harvard) et du professeur George Mealy (Université Harvard) ont été particulièrement utiles. Au sein de CCA, plusieurs personnes ont participé, en plus des auteurs, à la conception du langage à divers stades du projet, et tout particulièrement Hal Murray, Bill Bush, David Shipman et Dale Stern.

Table des matières

1. Introduction

1.1 Le système Datacomputer

Le datacomputer est un système d'utilitaire de données à grande échelle, qui offre des services de mémorisation et de gestion de données aux autres ordinateurs.

Le datacomputer diffère des systèmes de gestion de données traditionnels de plusieurs façons.

D'abord, il est mis en œuvre sur un matériel dédié, et comporte un système de calcul séparé spécialisé pour la gestion des données.

Ensuite, le système est mis en œuvre à grande échelle. Les données sont destinées à être mémorisées sur des appareils de stockage de masse, avec des capacités de l'ordre du gigabit. Des fichiers de l'ordre de la centaine de millions de bits seront mis en ligne.

Troisièmement, il est destiné à prendre en charge le partage de données entre des systèmes d'exploitation fonctionnant dans des environnements divers. C'est à dire que les programmes entre lesquels se partage une base de données peuvent être écrits dans des langages différents, s'exécuter sur des matériels différents sous des systèmes d'exploitation différents, et accepter des utilisateurs finaux avec des exigences radicalement différentes. Pour permettre une telle utilisation partagée d'une base de données, des transformations doivent être réalisées entre les diverses représentations de matériel et les concepts de structuration des données.

Finalement, le système datacomputer est conçu pour fonctionner en douceur comme composant d'un système beaucoup plus large : un réseau informatique. Dans un réseau informatique, le datacomputer est un nœud spécialisé pour la gestion des données, et qui agit comme utilitaire de données pour les autres nœuds. L'Arpanet, pour lequel le datacomputer est en cours de développement, est un réseau international qui a plus de 60 nœuds. Parmi eux, certains sont actuellement spécialisés pour le traitement terminal, d'autres sont spécialisés dans le calcul (par exemple, ILLIAC IV), certains sont des nœuds de service tout venant (par exemple, MULTICS) et un (CCA) est spécialisé dans la gestion de données.

1.2 Datalanguage

Datalanguage est le langage dans lequel sont formulées toutes les demandes à l'ordinateur de données *(datacomputer)*. Il comporte des facilités pour la description et la création des données, pour la restitution ou la modification de données mémorisées, et pour l'accès à diverses facilités et services auxiliaires. En language de données, il est possible de spécifier toute opération que l'ordinateur des données est capable d'effectuer. Datalanguage est le seul langage accepté par l'ordinateur des données et c'est le moyen d'accès exclusif aux données et aux services.

1.3 Le présent effort de conception

Nous sommes actuellement engagés dans le développement de spécifications complètes du langage des données ; ceci est la seconde itération du processus de conception du langage.

Un premier essai plus modeste avait développé certains concepts et principes qui sont décrits dans le troisième document de travail de la présente série. Ils ont été utilisés comme base des mises en œuvre de logiciels qui ont résulté en une première capacité de service réseau. Un manuel d'utilisateur pour ce système a été publié comme document de travail numéro 7.

Il résulte de l'expérience acquise dans la mise en œuvre et du fonctionnement, à travers l'étude des exigences des utilisateurs et de travaux avec des utilisateurs potentiels, ainsi que de recherches sur d'autres travaux dans le domaine de la gestion des données, que quelques idées nouvelles ont été développées pour l'amélioration du langage des données. Elles ont été assimilées dans la conception du langage de cette nouvelle présentation.

Lorsque la conception du langage sera achevée, elle sera incorporée dans le logiciel existant (ce qui exigera des changements aux compilateurs de langage, mais aura peu d'impact sur le reste du système).

Les utilisateurs de Datacomputer auront accès au nouveau langage dans le courant de 1975.

1.4 Objet du présent document

Le présent document présente les concepts et les résultats préliminaires, plutôt que le concept achevé. Cette publication avancée est motivée par deux raisons.

La première est d'apporter des informations à ceux qui projettent d'utiliser l'ordinateur de données. Il pourront bénéficier de la connaissance de nos intentions pour leurs développements.

La seconde est de permettre aux concepteurs de système et de langage de commenter notre travail avant que la conception n'en soit figée.

1.5 Organisation du présent document

La suite du présent document est divisée en quatre sections.

La section 2 expose les considérations les plus globales de la conception du langage. Cela comporte nos vues sur le problème ; elles ont influencé notre travail jusqu'à présent et vont déterminer la plupart de nos actions pour achever ce dessein. Cette section donne les fondements pour la section 3, et révise certaines notions qui seront familières à ceux qui ont suivi de près nos travaux.

La section 3 expose certaines des questions spécifiques sur lesquelles nous avons travaillé. L'accent est mis sur les solutions et les options de solution.

Dans les sections 2 et 3, nous présentons notre travail "de bas en haut" : c'est à dire à partir de la réflexion sur la base des exigences connues et de notre conception des propriétés souhaitables du langage de données.

Nous avons aussi travaillé dans l'autre sens, en développant les primitives à partir desquelles se construit le langage. La section 4 présente notre travail dans ce domaine : un ordinateur de données modèle qui fournira finalement une définition sémantique précise du langage de données. La section 4 explique la partie du modèle qui est terminée, et la met en rapport avec nos autres travaux.

La section 5 expose les travaux qui restent à faire, à la fois sur le modèle et sur notre analyse "de bas en haut".

2. Considérations sur la conception du langage

2.1 Introduction

La gestion des données est la tâche qui consiste à gérer les données comme une ressource, sans considération du matériel et des programmes d'application. Elle peut être divisée en cinq sous tâches majeures :

(1) création de bases de données en mémoire,
(2) rendre les données disponibles (par exemple en satisfaisant les interrogations),
(3) maintenance des donnée lorsque des informations sont ajoutées, supprimées et modifiées,
(4) assurer l'intégrité des données (par exemple, par des systèmes de sauvegarde et de récupération, par des vérifications de cohérence internes),
(5) régulation d'accès, pour protéger les bases de données, le système, et la confidentialité des usagers.

Ce sont les fonctions majeures de l'ordinateur de données en rapport avec les données ; alors que le système va en fin de compte fournir d'autres services (tels que la comptabilité de l'usage, la surveillance des performances) celles-ci sont en fait auxiliaires et communes à toutes les facilités de service.

La présente section présente les considérations générales de la conception du langage de données, fondées sur nos observations du problème et de l'environnement dans lequel il doit être résolu. Le problème central est celui de la gestion des données, et l'ordinateur de données partage les mêmes objectifs que beaucoup des systèmes de données actuellement disponibles. Plusieurs aspects de l'ordinateur de données en font un ensemble unique de problèmes à résoudre.

2.2 Considérations sur le matériel

2.2.1 Une boîte séparée

L'ordinateur de données est un utilitaire complet de gestion de données dans un système clos, isolé. C'est à dire que le matériel, les données et le logiciel de gestion des données sont tenus à part de toutes les facilités de traitement d'objet général. Il y a une installation distincte dédiée à la gestion des données. Datalanguage est le seul moyen qu'ont les utilisateurs pour communiquer avec l'ordinateur de données et la seule activité de l'ordinateur de données est de traiter les demandes du langage de données.

Un matériel dédié donne un avantage évident : on peut le spécialiser à la gestion des données. Le ou les processeurs peuvent être modifiés pour avoir des "instructions" de gestion des données ; les fonctions communes de logiciel de niveau inférieur peuvent être incorporées dans le matériel.

Un avantage moins évident, mais peut-être plus significatif, est tiré de l'isolement lui-même. Le système peut être plus facilement protégé. Un ordinateur de données pleinement développé sur lequel il n'y a que des activités de maintenance peut fournir un environnement très soigneusement contrôlé. D'abord, il peut être rendu aussi physiquement sûr que souhaité. Ensuite, il n'a besoin d'exécuter que les logiciels système développés à CCA ; tous les programmes d'utilisateur sont dans un langage de haut niveau (le langage de données) qui est interprété efficacement par le système. Et donc, seul le logiciel système de l'ordinateur de données traite les données, et le système est moins vulnérable à la capture par un programme hostile. Donc, comme il y a le potentiel pour développer les services de confidentialité et d'intégrité des données qui ne sont pas disponibles sur les systèmes non spécialisés, on peut s'attendre à moins de difficultés pour développer les contrôles de confidentialité (y compris physiques) pour l'ordinateur de données que pour les systèmes qu'il sert.

2.2.2 Matériel de stockage de masse

L'ordinateur de données va mémoriser la plupart de ses données sur des appareils de mémorisation de masse, qui ont des caractéristiques d'accès distinctes. Deux exemples d'un tel matériel sont l'Unicon 690 de Precision Instruments et le système TBM de Ampex Corporation. Ils sont assez différents des disques, et diffèrent l'un de l'autre de façon significative.

Cependant, presque tous les utilisateurs vont ignorer les caractéristiques de ces appareils ; beaucoup ne vont même pas savoir que les données qu'ils utilisent sont sur l'ordinateur de données. Finalement, comme le développement du système progresse, les données peuvent être invisiblement envoyées d'un ordinateur de données à un autre, et par suite être mémorisées dans un format physique assez différent de celui utilisé à l'origine.

Dans un tel environnement, il est clair que les demandes de données devraient être établies en termes logiques, et non physiques.

2.3 Environnement du réseau

L'environnement du réseau génère des exigences supplémentaires pour le concept d'ordinateur de données.

2.3.1 Utilisation à distance

Comme l'ordinateur de données doit être accédé à distance, l'exigence de techniques efficaces de choix des données et de bons mécanismes pour l'expression des critères de choix est amplifiée. Cela à cause de l'étroitesse du chemin à travers lequel les utilisateurs du réseau communiquent avec l'ordinateur de données. Actuellement, un taux de transfert normal de process à process sur l'Arpanet est de 30 kilobits par seconde. Bien que cela puisse être augmenté grâce à l'optimisation des logiciels et des protocoles, et par des investissement supplémentaires en matériels et lignes de communications, il semble sûr de supposer que ce n'est pas près d'approcher des taux de transfert locaux (mesurés en mégabits par seconde).

Une demande normale réclame soit le transfert d'une partie d'un fichier à un site distant, soit une mise à jour sélective sur un fichier déjà mémorisé à l'ordinateur de données. Dans ces deux situations, de bons mécanismes pour spécifier les parties des données à transmettre ou changer vont réduire la quantité de données ordinairement transférées. Ceci est extrêmement important parce qu'avec le faible coût par bit de la mémorisation des données à l'ordinateur de données, les coûts de transmission vont représenter une part significative du coût total de l'utilisation de l'ordinateur de données.

2.3.2 Utilisation inter-traitement du système Datacomputer

L'utilisation efficace du réseau exige que des groupes de processus, distants les uns des autres, soient capables de coopérer pour accomplir une tâche donnée ou fournir un service donné. Par exemple, pour résoudre un problème donné qui implique la manipulation d'un dispositif, la restitution de données, l'interaction avec l'utilisateur d'un terminal, et les services généralisés d'un langage comme PL/I, il peut être plus économique d'avoir quatre processus qui coopèrent. L'un deux pourrait s'exécuter sur l'ILLIAC IV, un autre à l'ordinateur de données, un autre au MULTICS, et le dernier sur un TIP.

Bien qu'il y ait de la redondance à l'établissement de ces quatre processus et à leur inter communication, chacun accomplit sa tâche sur un système spécialisé dans cette tâche. Dans de nombreux cas, le résultat de l'utilisation d'un système spécialisé est un gain de plusieurs ordres de grandeur en économies ou en efficacité (par exemple, la mémorisation en ligne sur l'ordinateur de données a un coût en capital inférieur de deux ordres de grandeur aux coûts en ligne sur les systèmes conventionnels). Il en résulte qu'il y a une incitation considérable à examiner les solutions qui impliquent la coopération de processus sur des systèmes spécialisés.

Pour résumer : l'ordinateur de données doit être prêt à fonctionner comme un composant de petits réseaux de processus spécialisés, afin qu'il puisse être utilisé de façon efficace dans un réseau dans lequel il y a de nombreux nœuds spécialisés.

2.3.3 Traitement courant des données par le réseau

Un grand réseau peut prendre en charge suffisamment de matériels de gestion de données pour construire plus d'un ordinateur de données. Alors que ce matériel peut être combiné en un ordinateur de données encore plus grand, il y a des avantages à le configurer en deux systèmes (ou éventuellement plus). Chaque système devrait être assez grand pour faire des économies d'échelle dans la mémorisation des données et pour prendre en charge le logiciel de gestion des données. Des bases de données importantes peuvent être dupliquées, avec une copie dans chaque ordinateur de données ; si un ordinateur de données a une défaillance, ou est isolé par une défaillance du réseau, les données restent disponibles. Même si la duplication du fichier n'est pas garantie, la description peut en être conservée sur différents ordinateurs de données afin que les applications qui ont un besoin constant de mémorisation des données aient la garantie qu'au moins un des ordinateurs de données est disponible pour recevoir les entrées.

Ces sortes de protections contre les défaillances impliquent une coopération entre une paire d'ordinateurs de données ; dans un certain sens, elles exigent que les deux ordinateurs de données fonctionnent comme un seul système. Étant donné un système d'ordinateurs de données (qu'on peut penser comme un petit réseau d'ordinateurs de données) il est évidemment possible de faire l'expérience de la fourniture de services supplémentaires au niveau du réseau d'ordinateur de données. Par exemple, toutes les demandes pourraient être simplement adressées au réseau d'ordinateur de données ; le réseau d'ordinateur de données pourrait alors déterminer où est mémorisé (c'est-à-dire, sur quel ordinateur de données) chacun des fichiers référencés et comment satisfaire la demande au mieux.

Ici, deux sortes de coopération ont été mentionnées dans l'environnement de réseau : la coopération entre les processus pour résoudre un problème donné, et la coopération entre les ordinateurs de données pour fournir des optimisations globales au problème du traitement des données au niveau réseau. Ce sont seulement deux exemples, particulièrement intéressants parce qu'ils peuvent être mis en œuvre à court terme. Dans le réseau, des sortes de coopération beaucoup plus générales sont possibles, quoiqu'à un peu plus long terme. Par exemple, finalement, on peut vouloir que les ordinateurs de données fassent partie d'un système de gestion de données à l'échelle du réseau, dans lequel les données, les répertoires, les services, et les matériels seraient répartis de façon générale sur le réseau. Le système entier pourrait fonctionner comme un tout dans les circonstances appropriées. La plupart des demandes utiliseraient les données et les services de seulement quelques nœuds. Au sein du système à l'échelle du réseau, il y aurait plus d'un système de gestion des données, mais tous les systèmes s'interfaceraient à travers un langage commun. Comme les ordinateurs de données représentent la plus grande ressource de gestion de données dans le réseau, ils joueraient certainement un rôle important dans tout système à l'échelle du réseau. Le langage de l'ordinateur de données (langage de données) est certainement un choix pratique pour le langage commun à un tel système.

Et donc une exigence finale, encore que futuriste, imposée par le réseau à la conception d'un système d'ordinateurs de données, est qu'il soit un composant majeur convenable pour les systèmes de gestion de données à l'échelle du réseau. Si c'est faisable, on voudrait que le langage de données soit un candidat convenable comme langage commun d'un groupe de systèmes de gestion de données coopérant à l'échelle du réseau.

2.4 Différents modes d'utilisation de Datacomputer

Au sein de cet environnement de réseau, l'ordinateur de données va jouer plusieurs rôles. Quatre de ces rôles sont décrits dans cette section. Chacun d'eux impose des contraintes à la conception du langage de données. On peut les analyser selon les quatre avantages qui se chevauchent et qui sont présentées par l'ordinateur de données :

1. Services généralisés de gestion de données
2. Traitement de grands fichiers
3. Accès partagé
4. Mémorisation économique de gros volumes

Bien sûr, la principale raison pour utiliser l'ordinateur de données sera le service de gestion des données qu'il fournit. Cependant, pour certaines applications la taille va être le facteur dominant en ce que l'ordinateur de données va fournir un accès en ligne à des fichiers qui sont si grands que précédemment seul le stockage et le traitement hors ligne étaient possibles. La capacité à partager des données entre différents sites réseau avec des matériels très différents est une autre caractéristique que seul l'ordinateur de données peut apporter. Les économies d'échelle rendent l'ordinateur de données un substitut viable des bandes magnétiques dans de telles applications comme sauvegarde du système d'exploitation.

Naturellement, une combinaison des facteurs ci-dessus va se rencontrer dans la plupart des applications des ordinateurs de données. Les paragraphes suivants décrivent certains des modes d'interaction possibles avec l'ordinateur de données.

2.4.1 Prise en charge de grandes bases de données partagées

C'est l'application la plus significative de l'ordinateur de données, sous presque tous les aspects.

Des projets sont en cours pour mettre en ligne des bases de données de plus de cent milliards de bits sur l'ordinateur de données d'Arpanet. Parmi eux, il y a une base de données qui va finalement inclure 10 années d'observations météorologiques provenant de 5000 stations situées tout autour de la terre. Comme base de données en ligne, c'est d'une taille sans précédent. Elle présente un intérêt mondial et sera partagée par des utilisateurs qui travaillent sur des matériels très variés et dans toutes sortes de langages.

Parce que ces bases de données sont en ligne sur un réseau international, et parce qu'il est prévu qu'elles soient d'un intérêt considérable pour les chercheurs dans leurs champs respectifs, il semble évident qu'il y aura des schémas d'utilisation extrêmement variés. Une exigence forte est alors d'une approche souple et générale pour les traiter. Cette exigence de fournir aux différents utilisateurs d'une base de données des vues différentes des données et une préoccupation majeure de l'effort de conception du langage de données. Elles est exposée à part au paragraphe 2.5.

2.4.2 Extensions de systèmes locaux de gestion de données

On imagine que les systèmes locaux de traitement de données (systèmes de gestion de données, paquetages orientés applications, systèmes de traitement de texte, etc.) veulent tirer parti de l'ordinateur de données. Ils peuvent le faire à cause de l'économie de mémoire, à cause des services de gestion des données, ou parce qu'ils veulent tirer parti des données déjà mémorisées dans l'ordinateur de données. Dans tous les cas, de tels systèmes ont des propriétés distinctives comme utilisateurs d'ordinateur de données : (1) la plupart vont utiliser des données locales aussi bien que les données de l'ordinateur de données, (2) beaucoup seront concernés par la traduction de demandes locales en langage de données.

Par exemple, un système qui fait de la simple restitution de données et de l'analyse statistique pour des chercheurs en sciences sociales qui ne font pas de programmation peut vouloir utiliser une base de données du recensement mémorisée sur l'ordinateur de données. Un tel système peut effectuer toute une gamme de fonctions de restitution des données, et peut avoir besoin d'interactions sophistiquées avec l'ordinateur de données. Son schéma d'utilisation serait assez différent de celui d'un simple programme d'application dont la seule utilisation de l'ordinateur de données serait l'impression d'un rapport spécifique sur la base d'un seul fichier connu.

Ce système de sciences sociales utiliserait aussi des bases de données locales qu'il conserverait sur son propre site parce qu'elles sont petites et d'un accès plus efficace en local. On aimerait qu'il soit pratique de considérer les données de la même façon, qu'elles soient mémorisées en local ou sur l'ordinateur de données. Il y aura certainement des différences d'interface aux niveaux inférieurs du logiciel local ; il serait cependant agréable que les concepts et opérations locales puissent être facilement traduits en langage de données.

2.4.3 Utilisation de Datacomputer au niveau fichier

Dans ce mode d'utilisation, d'autres systèmes d'ordinateur tirent parti de la capacité de mémorisation en ligne de l'ordinateur de données. Pour ces systèmes, la mémorisation de l'ordinateur de données représente une nouvelle classe de mémoires moins chères et plus sûres que les bandes, presque aussi accessible qu'un disque local. Peut-être même peuvent-ils déplacer automatiquement les fichiers d'une mémorisation en ligne locale à l'ordinateur de données, donnant aux utilisateurs l'impression que tout est mémorisé en ligne localement.

La caractéristique distinctive de ce mode d'utilisation est que les opérations se font sur des fichiers entiers.

Un système qui fonctionne dans ce mode utilise seulement la capacité de mémoriser, restituer, ajouter, renommer, faire des répertoires et ainsi de suite. Une façon évidente de faire du traitement de niveau fichier quelque chose de facilement disponible pour la communauté de l'Internet est de faire usage du protocole de transfert de fichiers (du document n° 17 759 du Centre d'Informations du réseau) déjà utilisé pour le transfert de fichiers entre hôtes.

Bien qu'une telle utilisation de "tout le fichier" par l'ordinateur de données soit principalement motivée par les avantages de l'économie d'échelle, le partage des données au niveau fichier pourrait aussi être un sujet d'intérêt. Par exemple, les fichiers source de logiciels réseau communs peuvent résider sur l'ordinateur de données. Ces fichiers ont peu ou pas de structure, mais leur utilisation commune impose qu'ils soient disponibles en un lieu commun, toujours accessible. Cela tire parti de l'économie de l'ordinateur de données, plus que de n'importe quoi d'autre, car la plupart de ces services sont disponibles sur tout système de fichiers.

Ce mode d'utilisation n'est mentionné ici que parce qu'il tient compte d'un grand pourcentage des demandes du langage de données. Il n'exige que des capacités qui seraient présentes dans le langage de données dans tous les cas ; la seule exigence particulière est de s'assurer qu'il est facile et simple d'accomplir ces tâches.

2.4.4 Utilisation de Datacomputer pour l'archivage de fichiers

Ceci est une autre application orientée vers l'économie. L'idée de base est de mémoriser sur l'ordinateur de données tout ce dont vous n'allez avoir besoin que de façon occasionnelle. Cela pourrait inclure les fichiers de sauvegarde, les journaux d'audit, et les choses de ce genre.

Une idée intéressante en rapport avec l'archivage est l'archivage incrémentaire. Une pratique normale au regard de la sauvegarde de données mémorisées en ligne dans un système en temps partagé, est d'écrire toutes les pages qui sont différentes de ce qu'elles étaient dans le dernier dépôt. Il est alors possible de récupérer en restaurant le dernier dépôt complet, puis de restaurer tous les dépôts incrémentés jusqu'à la version désirée. Ce système offre de meilleurs coûts pour l'archivage et la mémorisation, et un coût plus élevé en récupération ; il est approprié lorsque la probabilité d'avoir besoin d'une récupération est faible. Le langage de données devrait alors être conçu pour permettre un archivage incrémentaire convenable.

Comme dans le cas de l'application précédente (système de fichiers), il est important de prendre l'archivage en considération au niveau des concepts à cause de sa fréquence et de son économie, et non parce qu'il exige nécessairement des particularités au niveau du langage. Il peut imposer que des mécanismes spécialisés pour l'archivage soient incorporés dans le système.

2.5 Partage de données

Le partage contrôlé des données est une préoccupation centrale de ce projet. Trois sous problèmes majeurs du partage des données sont : (1) l'utilisation en concurrence, (2) les concepts indépendants dans la même base de données, et (3) les variantes dans la représentation de la même base de données.

L'utilisation en concurrence d'une ressource par plusieurs processus indépendants est couramment mise en œuvre pour des données au niveau du fichier dans des systèmes dans lesquels les fichiers sont considérés comme des objets disjoints, sans relation. Elle est parfois mise en œuvre au niveau de la page.

Des travaux considérables ont déjà été menés sur ce problème au sein du projet d'ordinateur de données. Lorsque ce travail sera achevé, il aura un impact sur la conception du langage ; en tout état de cause, nous ne considérons cependant pas que cet aspect de l'utilisation en concurrence soit un problème de langage.

D'autres aspects du problème de l'utilisation en concurrence peuvent cependant exiger une participation plus consciente de la part de l'utilisateur. Ils se rapportent à la sémantique des collections d'objets de données, lorsque de telles collections s'étendent sur les frontières de fichiers connus du système d'exploitation interne. La question de ce qui constitue un conflit de mise à jour est ici plus complexe. Des questions en rapport avec celle là apparaissent sur la sauvegarde et la récupération. Si deux fichiers sont en rapport, peut-être n'y a-t il pas de sens à récupérer un état antérieur de l'un sans récupérer l'état correspondant de l'autre. Ces problèmes restent encore à creuser.

Un autre problème du partage des données est que tous les utilisateurs d'une base de données n'ont pas la même conception de cette base de données. Voici des exemples : (1) pour des raisons de confidentialité, certains utilisateurs devraient n'avoir accès qu'à une partie de la base de données (par exemple, les scientifiques qui font des études statistiques sur des dossiers médicaux n'ont pas besoin d'avoir accès aux noms et adresses), (2) pour l'indépendance des données de programme, les programmes de paye ne devraient accéder qu'aux données concernées par la rédaction des chèques de salaire, même si des inventaires des qualifications peuvent être stockés dans la même base de données, (3) pour un contrôle global d'efficacité, la simplicité des programmes d'application, et l'indépendance des données de programme, chaque programme d'application devrait "voir" une organisation des données qui est la meilleure pour sa tâche.

Pour pousser un peu plus loin l'analyse de l'exemple (3), considérons une base de données qui contient des informations sur des étudiants, des professeurs, des sujets et indique aussi quels étudiants ont quels professeurs sur quels sujets. Selon le problème à résoudre, un programme d'application peut avoir une forte exigence pour une des organisations suivantes :

(1) entrées de la forme (étudiant,professeur,sujet) sans souci des redondances. Dans cette organisation un objet d'un des trois types peut survenir de nombreuses fois.

(2) entrées de la forme

```
(étudiant,      (professeur,sujet),
                (professeur,sujet),
                .
                .
                .
                (professeur,sujet))
```

(3) entrées de la forme

```
(professeur,    sujet,(étudiant...étudiant),
                sujet,(étudiant...étudiant),
                sujet,(étudiant.. .étudiant))
```

et d'autres organisations sont certainement possibles.

Une approche de ce problème est de choisir une organisation pour les données mémorisées, puis d'avoir des programmes d'application qui écrivent les demandes qui organisent la sortie sous la forme qu'elles veulent. Le programmeur

d'application applique son ingéniosité à établir la demande de telle sorte que le processus de réorganisation soit combiné au processus de restitution, et le résultat est relativement efficace. Il y a d'importantes situations pratiques dans lesquelles cette approche est adéquate ; en fait il y a des situations dans lesquelles elle est souhaitable. En particulier, si l'efficacité ou le coût est une considération prépondérante, il peut être nécessaire pour chaque programmeur d'application de connaître tous les facteurs d'accès aux données et de leur organisation. Cela peut être le cas pour un fichier massif, dans lequel chaque restitution doit être adaptée à la stratégie et l'organisation de l'accès ; tout autre mode de fonctionnement résulterait en des coûts ou des temps de réponse inacceptables.

Cependant, la dépendance entre programme d'application et organisation des données ou stratégie d'accès n'est en général pas une bonne politique. Dans une base de données largement partagée, cela peut signifier un coût énorme dans le cas d'une réorganisation de la base de données, de changements du logiciel d'accès, ou même de changements du support de mémoire. Un tel changement peut exiger de reprogrammer dans des centaines de programmes d'application répartis dans tout le réseau.

En conséquence, on voit la nécessité d'un langage qui prenne en charge tout le spectre des modes de fonctionnement, y compris : (1) le programme d'application est complètement indépendant de la structure de mémorisation, de la technique d'accès, et de la stratégie de réorganisation, (2) les paramètres du programme d'application les contrôlent, (3) le programme d'application contrôle entièrement ces paramètres. Pour une base de données largement partagée, le mode (1) serait la politique préférée, sauf lorsque (a) le programmeur d'application pourrait faire un meilleur travail que le système pour prendre les décisions, et (b) la nécessité de cette augmentation d'efficacité surpasse les bénéfices de l'indépendance des données de programme.

En évaluant cette question pour une application particulière, il est important de réaliser le rôle de l'analyse globale d'efficacité. Lorsque il y a de nombreux usagers d'une base de données, le meilleur mode de fonctionnement dans un certain sens est ce qui minimise le coût total du traitement de toutes les requêtes et le coût total de mémorisation des données. Lorsque les applications vont et viennent, comme dans le monde réel où les besoins changent, les avantages du contrôle centralisé sont vraisemblablement surpassés par ceux de l'optimisation pour un programme d'application particulier.

Le troisième sous problème majeur survient en connexion avec les représentations de niveau d'élément. À cause de l'environnement dans lequel il s'exécute, chaque programme d'application a un ensemble préféré de concepts de formatage, indicateurs de longueur, conventions de bourrage et d'alignement, tailles de mots, représentations de caractères, et ainsi de suite. Encore une fois, il est de meilleure politique pour le programme d'application de se préoccuper seulement des représentations qu'il veut et non de la représentation des données mémorisées. Cependant, il va y avoir des cas dans lesquels l'efficacité pour une demande donnée surpasse tous les autres facteurs.

À ce niveau de représentation, il y a au moins une considération supplémentaire : la perte potentielle d'informations lors d'une conversion. Quiconque initie une conversion de type (et ce sera parfois l'ordinateur de données et parfois le programme d'application) doit aussi se charger d'assurer que les intentions de la demande sont préservées. Comme l'ordinateur de données doit toujours être responsable de la cohérence et de la signification d'une base de données partagées, il y a là quelques conflits à résoudre.

Pour résumer, il semble que le résultat d'un large partage des bases de données est qu'un plus grand système doit être envisagé lors du choix d'une politique de gestion des données pour une base de données particulière. Ce plus grand système, dans le cas de l'ordinateur de données, consiste en un réseau de programmes d'application répartis géographiquement en une base de données centralisée et en un système de gestion des données centralisé. L'exigence pour le langage des données est de fournir la souplesse dans la gestion de ce plus grand système. En particulier, il doit être possible de contrôler quand et où les conversions, les réorganisations de données, et les stratégies d'accès sont faites.

2.6 Besoin de communications de haut niveau

Toutes les considérations ci-dessus pointent sur la nécessité de communications de haut niveau entre l'ordinateur de données et ses usagers. La nature complexe et distincte du matériel de l'ordinateur de données rend impératif que les requêtes soient posées par l'ordinateur de données de telle sorte qu'il puisse prendre les décisions majeures concernant les stratégies d'accès à utiliser. En même temps, la grande quantité de données mémorisées et la demande de certains usagers de bandes passantes de transmission extrêmement élevées rend nécessaire de fournir un contrôle d'utilisateur de certains schémas de mémorisation et de transmission. Le fait que les bases de données seront utilisées par des applications qui désirent des vues différentes des mêmes données et avec des contraintes différentes signifie que l'ordinateur de données doit être capable de transposer la demande d'un usager sur les données d'autres utilisateurs. L'utilisation interprocessus de l'ordinateur de données signifie que le partage des données doit être complètement contrôlable pour éviter d'avoir besoin d'intervention humaine. De larges facilités pour assurer l'intégrité des données et contrôler l'accès doivent être fournies.

2.6.1 Description des données

L'exigence de base pour tous ces besoins est que les données mémorisées dans l'ordinateur de données soient complètement décrites dans des paramètres à la fois fonctionnels et physiques. Une description de haut niveau des données est particulièrement importante pour assurer le partage et le contrôle des données. L'ordinateur de données doit être capable de transposer entre différents matériels et différentes applications. Sous sa forme la plus triviale, cela signifie d'être capable de convertir des représentations de nombre à virgule flottante sur des machines différentes. À l'autre extrême, cela signifie

d'être capable de fournir des données matricielles pour le ILLIAC IV aussi bien que d'être capable de donner des réponses aux interrogations de programmes en langage naturel, toutes deux adressées à la même base de données météorologiques. Les descriptions de données doivent donner la capacité de spécifier les représentations au niveau binaire et les propriétés logiques et les relations des données.

2.6.2 Intégrité des données et contrôle d'accès

Dans l'environnement que nous avons décrit, les problèmes de la maintenance de l'intégrité des données et du contrôle de l'utilisation des données prennent une extrême importance. L'utilisation partagée des fichiers de l'ordinateur de données dépend de sa capacité à garantir que les restrictions à l'accès des données sont strictement mises en application. Comme les différents utilisateurs vont avoir des descriptions différentes, le mécanisme de contrôle d'accès doit être associé aux descriptions elles-mêmes. On peut contrôler l'accès aux données en contrôlant l'accès à leurs divers descripteurs. Un usager peut être contraint d'accéder à une base de données particulière par une seule description spécifique qui limite les données auxquelles il peut accéder. Dans un système où ceux qui mettent à jour une base de données peuvent ne pas se connaître les uns les autres, et avoir éventuellement des vues différentes des données, seul l'ordinateur de données peut assurer l'intégrité des données. Pour cette raison, toutes les restrictions sur les valeurs possibles des objets de données, et sur les relations possibles ou nécessaires entre les objets, doivent être déclarées dans la description des données.

2.6.3 Optimisation

Les décisions concernant la stratégie d'accès aux données doivent ordinairement être prises à l'ordinateur de données, où est disponible la connaissance des considérations physiques. Ces décisions ne peuvent pas être prises de façon intelligente si les demandes d'accès aux données ne sont pas faites à haut niveau.

Par exemple, comparons les deux situations suivantes : (1) une requête demande la sortie de toutes les observations météorologiques faites en Californie qui donnent certaines conditions de vent et de pression ; (2) une série de requêtes est envoyée, dont chacune demande une restitution des observations météorologiques de Californie : lorsque une requête trouve une observation avec les conditions de vent et de pression requises, elle transmet cette observation à un système distant. Les deux sessions arrivent au même résultat : la transmission d'un certain ensemble d'observations à un site de traitement distant. Dans la première session cependant, l'ordinateur de données reçoit au début une description des données qui sont nécessaires ; dans la seconde, il traite une série de requêtes, dont chacune est une surprise.

Dans le premier cas, un ordinateur de données intelligent a le choix de restituer toutes les données nécessaires en un accès à l'appareil de stockage de masse. Il peut alors mettre en mémoire tampon ces données jusqu'à ce que l'usager soit prêt à les accepter. Dans le second cas, l'ordinateur de données n'a pas les informations nécessaires pour une telle optimisation.

Le langage devrait permettre et encourager les usagers à fournir les informations nécessaires à l'optimisation. Ne pas le faire a un coût bien plus élevé avec le stockage de masse et de grands fichiers que dans les systèmes conventionnels.

2.7 Problèmes en rapport avec les applications

Dans les paragraphes précédents nous avons décrit un certain nombre de caractéristiques que doit fournit le système d'ordinateur de données. Dans celui-ci, on se concentre sur ce qui est nécessaire pour rendre ces caractéristiques directement utilisables par les usagers de l'ordinateur de données.

2.7.1 Interaction ordinateur de données-utilisateur

Une application interagit avec l'ordinateur de données dans une session. Une session consiste en une série de requêtes. Chaque session implique de se connecter à l'ordinateur de données via le réseau, d'établir les identités, et d'établir des chemins de transmission pour les données et pour le langage de données. Le langage de données est transmis en mode caractères (utilisant la norme réseau ASCII) sur la connexion de langage de données. Les messages d'erreur et d'état sont envoyés sur cette connexion au programme d'application.

La connexion de données (appelée un accès) est vue comme un flux binaire et donne sa propre description. Ces descriptions sont similaires à celles données pour les données mémorisées. Au minimum, cette description doit contenir assez d'informations pour que l'ordinateur de données analyse le flux binaire entrant. Il peut aussi contenir également des informations de validation des données. Pour mémoriser les données chez l'ordinateur de données, les données mémorisées doivent aussi avoir une description. L'utilisateur fournit la transposition entre les descriptions des données mémorisées et transmises.

–

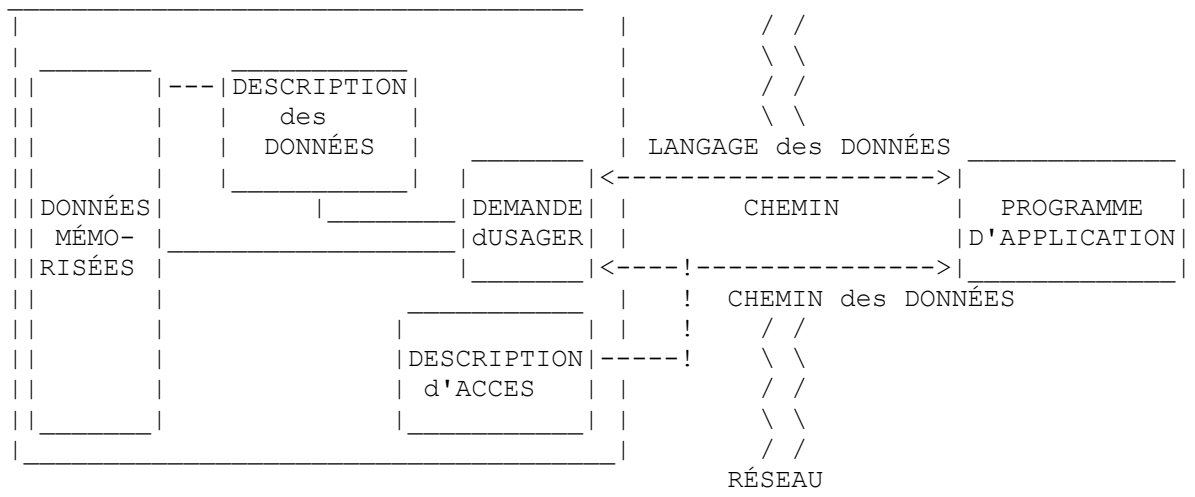
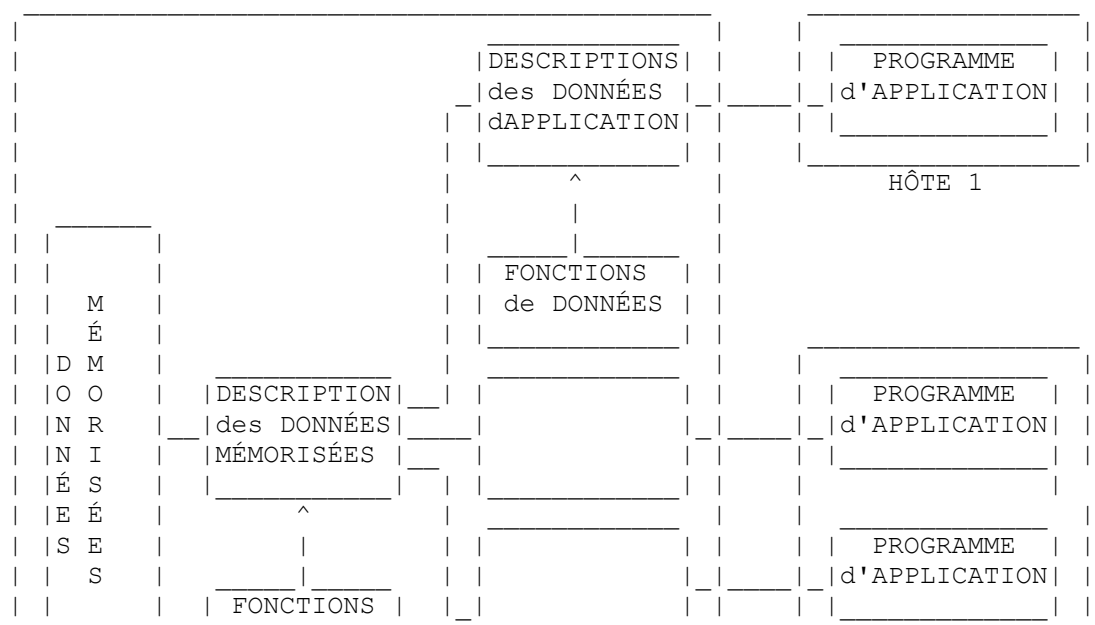


Figure 2-1 : Un modèle d'interaction Datacomputer/utilisateur

2.7.2 Caractéristiques d'application pour le partage de données

En utilisant les données mémorisées dans l'ordinateur de données, les usagers peuvent fournir une description des données qui sont personnalisées pour l'application. Cette description est transposée sur la description des données mémorisées. Ces descriptions peuvent être à des niveaux différents. C'est à dire que l'une peut simplement réarranger l'ordre de certains éléments, tandis qu'une autre pourrait invoquer une restructuration totale de la représentation mémorisée. Afin que chaque usager soit capable de bâtir sur les descriptions des autres, les entités de données devraient recevoir des types nommés. Ces définitions de type sont bien sûr à mémoriser avec les données qu'elles décrivent. De plus, certaines fonctions sont si étroitement liées aux données (en fait, elle peuvent être les données dans le cas d'une description virtuelle – voir la section 3) qu'elles doivent aussi résider dans l'ordinateur de données et leur lien avec les éléments de données devrait être maintenu par l'ordinateur de données. Par exemple, un usager peut décrire une base de données comme constituée de structures contenant des données des types "latitude" et "longitude". Il pourrait aussi décrire les fonctions pour comparer les données de ce type. D'autres usagers, non concernés par la structure du composant "latitude" lui-même, mais intéressés par l'utilisation de cette information simplement pour extraire d'autre champs qui les intéressent peuvent alors utiliser les définitions et fonctions fournies à l'usage de tous.

De plus, en adoptant cette stratégie, autant d'usagers que possible peuvent être rendus insensibles aux changements dans les fichiers qui sont à côté de leurs intérêts principaux. Par exemple, "latitudes" pourrait être changé d'une représentation binaire à une forme de caractère et si l'utilisation de ce champ était restreinte à ses définitions et fonctions associées, les systèmes d'application existants n'en seraient pas affectés. Les fonctions de conversion peuvent être définies pour éliminer l'impact sur les programme fonctionnant actuellement. La capacité de telles facilités de définition signifie que des groupes d'utilisateurs peuvent développer des fonctions et descriptions communes pour traiter des données partagées et que les conventions pour l'utilisation de données partagées peuvent être mises en application par l'ordinateur de données. Ces facilités sont discutées au paragraphe 3.15 "Extensibilité".



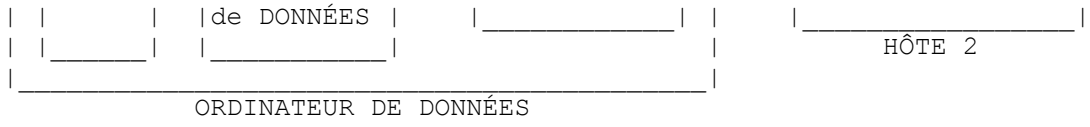


Figure 2-2 : Interaction de plusieurs utilisateurs avec l'ordinateur de données

2.7.3 Modèle de communication

Notre intention est que le langage de données, bien qu'à un haut niveau conceptuel, soit à un faible niveau de syntaxe. Le langage de données fournit un ensemble de fonctions primitives, et un ensemble de fonctions de niveau supérieur d'utilisation courante (voir la section 4 sur le modèle de langage de données). De plus, les utilisateurs peuvent définir leurs propres fonctions de sorte qu'ils puissent communiquer avec l'ordinateur de données à un niveau aussi proche conceptuellement de l'application que possible.

Il y a deux raisons pour que le langage de données soit à un faible niveau syntaxique. D'abord, il n'est pas souhaitable d'avoir des programmes qui composent les requêtes dans un format élaboré seulement pour être décomposées par l'ordinateur de données. Ensuite, en choisissant une syntaxe spécifique de haut niveau, l'ordinateur de données imposerait un ensemble de conventions et de terminologie qui ne correspondrait pas nécessairement à celui de la plupart des utilisateurs.

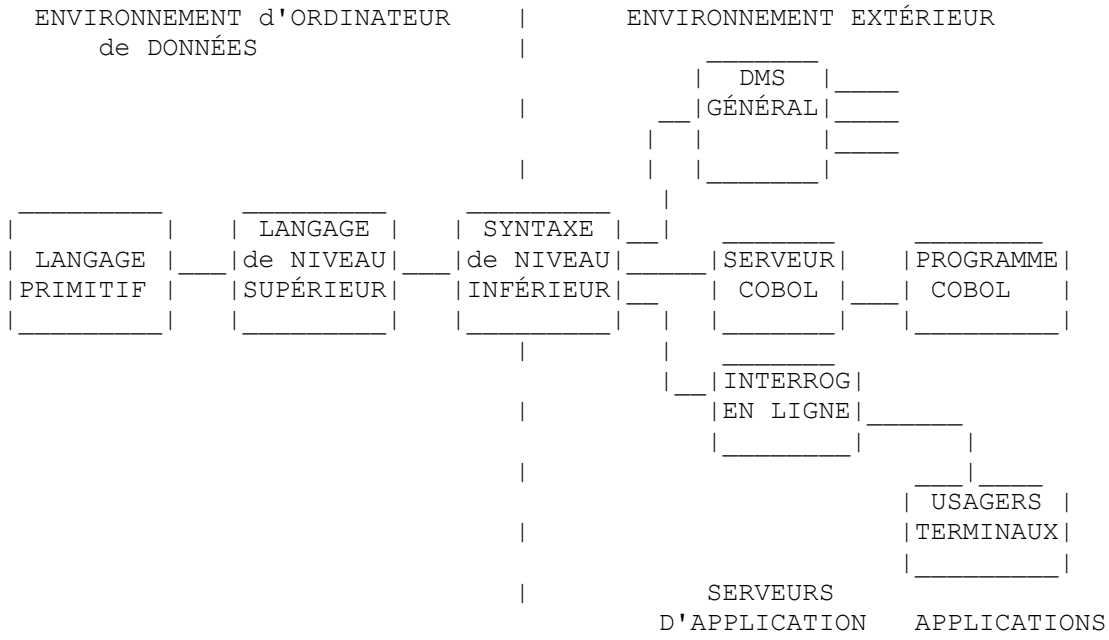


Figure 2-3 : Environnement de travail ordinateur de données/utilisateur

2.8 Résumé

Dans cette section, nous avons présenté les considérations majeures qui ont influencé l'effort actuel de conception du langage de données. L'ordinateur de données a beaucoup en commun avec la plupart des systèmes de gestion de données partagées à grande échelle, mais il a aussi un certain nombre de problèmes primordiaux qui sont particuliers au concept d'ordinateur de données. Le plus important d'entre eux est l'existence d'une boîte séparée contenant à la fois le matériel et le logiciel, le contrôle d'un appareil extrêmement important de mémorisation, et l'incorporation dans un environnement de réseau informatique. Le partage des données dans un tel environnement est une préoccupation centrale de la conception. Les facilités extensives de description des données et la communication de haut niveau entre usager et ordinateur de données sont nécessaires pour l'intégrité des données et pour l'optimisation des demandes d'utilisateur par l'ordinateur de données. De plus, l'utilisation prévue de l'ordinateur de données implique de satisfaire plusieurs contraintes opposées pour différents modes de fonctionnement. Une façon de satisfaire les divers besoins des usagers est de fournir des caractéristiques de langage de données telles que les usagers puissent développer leurs propres paquetages d'applications au sein du langage de données.

3. Principaux concepts de langage

Cette section expose les principales facilités du langage de données. Les détails spécifiques du langage ne sont pas présentées, cependant, l'exposé comporte les motivations qui sous-tendent l'inclusion des diverses caractéristiques du langage et définit aussi, de façon informelle, les termes utilisés.

3.1 Éléments de données de base

Les données de base sont le niveau atomique de toutes les constructions de données ; elles ne peuvent pas être décomposées. Toutes les structures de données de niveau supérieur sont fondamentalement composées d'éléments de données de base. De nombreux types d'éléments de données de base seront fournis. Le type d'un élément détermine les opérations qui peuvent être effectuées sur l'élément et la signification de ces opérations. Le langage des données fournira les types de primitives des éléments de données qui sont d'usage courant pour calculer les systèmes pour modéliser la réalité.

Les types de données de base suivants seront disponibles dans le langage de données : nombres à virgule fixe, nombres à virgule flottante, caractères, booléens, et bits. Ces types d'éléments sont "compris" par le système d'ordinateur de données dans la mesure où les opérations sont fondées sur le type d'un élément. Le langage des données va aussi comporter un type d'élément non interprété, pour les données qui seront seulement déplacées (y compris transmises) d'un endroit à un autre. Ce type de données ne sera compris que dans le sens trivial où l'ordinateur de données peut déterminer si deux éléments du type non interprété sont identiques. Les opérations standard sur les types de base des éléments seront disponibles. Les opérations seront incluses de telle sorte que l'usager de l'ordinateur de données puisse décrire une large gamme de fonctions de gestion de données. Elles ne sont pas incluses dans l'intention d'encourager l'utilisation de l'ordinateur de données pour la résolution de problèmes mathématiques.

3.2 Agrégats de données

Les agrégats de données sont des compositions d'éléments de données de base et éventuellement d'autres agrégats de données. Les types d'agrégats de données qui sont fournis permettent la construction de relations hiérarchiques des données. Les agrégats qui seront définitivement disponibles sont classés en struct *(structure)*, array *(dispositif)*, string *(chaîne)*, liste, et répertoire.

Une structure est un agrégat statique d'éléments de données (appelés composants). Une structure est statique dans ce sens que les composants d'une structure ne peuvent pas être ajoutés ou retranchés de la structure, ils sont inextricablement liés à la structure. Associé à chaque composant de la structure se trouve un nom par lequel ce composant peut être référencé par rapport à la structure. L'agrégat struct peut être utilisé pour modéliser ce qui est souvent vu comme un enregistrement, chaque composant étant un champ de cet enregistrement. Une structure peut aussi être utilisée pour grouper les composants d'un enregistrement qui sont plus fortement en relation, de façon conceptuelle, que d'autres composants et qui peuvent interopérer.

Les dispositifs permettent la répétition dans les structures de données. Un dispositif, comme une struct, est un agrégat statique d'éléments de données (appelés membres). Chaque membre d'un dispositif est du même type. Un indice est associé à chaque membre par lequel ce membre peut être référencé par rapport au dispositif. Les dispositifs peuvent être utilisés pour modéliser des répétitions de données dans un enregistrement (pour répéter des groupes).

Le concept de chaîne est en fait un hybride de données de base et d'agrégats de données. Les chaînes sont des agrégats en ce qu'elles sont des compositions (similaires à des dispositifs) de données plus primitives (par exemple, de caractères). Elles sont, cependant, généralement conçues comme de base en ce qu'elles sont principalement vues comme une unité plutôt que comme une collection d'éléments, où chaque élément a une importance individuelle. Aussi, la signification d'une chaîne est très dépendante de l'ordre des composants individuels. En termes plus concrets, il y a des opérations qui sont définies sur des types spécifiques de chaînes. Par exemple, les opérateurs logiques ("et", "ou", etc.) sont définis pour fonctionner sur des chaînes de bits. Cependant, il n'y a pas d'opération qui soit définie sur des dispositifs de bits, bien qu'il y ait des opérations définies à la fois sur des dispositifs, en général, et sur des bits. Les chaînes de caractères, bits, et données non interprétées seront disponibles dans le langage de données.

Les listes sont comme les dispositifs en ce qu'elles sont des collections de membres similaires. Cependant, les listes sont dynamiques plutôt que statiques. Les membres d'une liste peuvent être ajoutés et supprimés de la liste. Bien que les membres d'une liste soient ordonnés (en fait, plus d'un ordre peut être défini pour une liste) la liste n'est pas destinée à être référencée via un indice, comme c'est le cas avec un dispositif. Les membres d'une liste peuvent être référencés via une méthode de séquençage à travers la liste. Un membre d'une liste, ou un ensemble (voir la discussion sur les données virtuelles) de membres, peut aussi être référencé par une méthode d'identification par le contenu. La structure de la liste peut être utilisée pour modéliser la notion commune d'un fichier. Une utilisation restrictive des listes comme composants de structures donne un moyen d'action par rapport à la construction de relations dynamiques hiérarchisées au dessous du

niveau du fichier. Par exemple, les membres d'une liste peuvent être eux-mêmes, en partie, composés de listes, comme dans une liste de familles, où chaque famille contient une liste d'enfants ainsi que d'autres informations.

Les répertoires sont des agrégats de données dynamiques qui peuvent contenir tout type d'éléments de données. Les éléments de données contenus dans un répertoire sont appelés des "nœuds". Un nom est associé à chaque nœud d'un répertoire, par lequel ces éléments de données peuvent être référencés par rapport au répertoire. Comme avec les listes, les éléments peuvent être ajoutés de façon dynamique à un répertoire, et en être supprimés. La motivation principale de la fourniture de la capacité de répertoire est de permettre à l'usager de grouper des données en rapport conceptuel. Comme les répertoires n'ont pas besoin de contenir seulement des informations de type fichier, des données "auxiliaires" peuvent être conservées au titre du répertoire. Par exemple, des informations "constantes", comme des tableaux de gammes de salaires pour une base de données d'entreprise; ou des opérations et des types de données définies par l'utilisateur (voir ci-dessous) peuvent être conservées dans un répertoire avec les données qui peuvent utiliser ces informations. Les répertoires peuvent aussi faire eux-mêmes partie d'un répertoire, ce qui permet une hiérarchie des groupements de données.

Les répertoires seront aussi définis de telle sorte que les informations contrôlées par le système puissent être entretenues avec certains des éléments subordonnés (par exemple, l'heure de création, l'heure de mise à jour, les verrouillages pour la confidentialité, etc.). Il est aussi possible de permettre aux utilisateurs des données de définir et contrôler leurs propres informations qui seraient conservées avec les données. Au moins, la conception du langage de données va permettre un contrôle des paramètres des informations gérées par le système.

Les répertoires sont le type le plus général et dynamique de données agrégées. Le nom et la description (voir ci-dessous) des nœuds de répertoire existent tous deux avec les nœuds eux-mêmes, plutôt qu'au titre de la description du répertoire. Aussi, le niveau d'incorporation d'un répertoire est il dynamique car les répertoires peuvent être ajoutés de façon dynamique aux répertoires. Les répertoires sont le seul agrégat pour lequel c'est vrai.

Le langage de données va aussi fournir des variations spécifiques et utiles aux agrégats de données ci-dessus. Des structures seront disponibles pour permettre des composants facultatifs. Dans ce cas, l'existence d'un composant se fonderait sur le contenu d'autres composants. Il serait aussi possible de permettre que leur existence se fonde sur des informations trouvées à un niveau supérieur de la hiérarchie des données. De même, les composants avec le type "non résolu" seront fournis. C'est à dire que le composant peut être d'un nombre fixé de types. Le type du composant serait fondé sur le contenu d'autres composants de la structure. Il est aussi souhaitable de permettre que le type ou l'existence d'un composant soit fondé sur des informations autres que le contenu d'autres composants. Par exemple, le type d'un composant pourrait être fondé sur le type d'un autre composant. En général, on aimerait que le langage de données permette que les attributs (voir ci-dessous) d'un élément soient une fonction des attributs d'autres éléments.

On aimerait aussi fournir des listes mixtes. Les listes mixtes sont des listes qui contiennent plus d'un type de membres. Dans ce cas, les membres devraient se définir par eux-mêmes, c'est à dire que le type de tout membre devrait être tel que le degré d'information qui le définit puisse être trouvé.

Similaires aux composants dont le type est non résolu sont les dispositifs de longueur non résolue. Dans ce cas, les informations qui définissent la longueur du dispositif doivent être portées avec le dispositif lui-même ou peut-être avec d'autres composants d'un agrégat qui met en application le dispositif.

Dans tous les cas ci-dessus, le type d'un élément est non résolu à certain degré et les informations qui résolvent totalement le type sont portées avec l'élément. Il est possible que dans certains cas, ou peut-être tous, le système d'ordinateur de données pourrait être chargé de la maintenance de ces informations, les rendant invisibles à l'utilisateur des données.

3.3 Capacités relationnelles générales

Les agrégats de données décrits ci-dessus permettent la modélisation des diverses relations entre les données. Toutes les relations qui peuvent être construites sont hiérarchiques.

Deux approches peuvent être suivies pour donner la capacité de modéliser des relations non hiérarchiques. De nouveaux types d'agrégats de données peuvent être produits qui élargissent la gamme des relations de données qui peuvent être exprimées dans le langage de données. Ou bien, un type de données de base de "pointeur" peut être introduit pour servir de primitive à partir de laquelle des relations peuvent être représentées. Le pointeur serait un type de données qui établit une sorte de correspondance d'un élément à un autre. C'est-à-dire qu'il serait une méthode pour trouver un élément, l'autre étant donné. La fourniture de la capacité d'avoir des éléments du type pointeur n'exige pas l'introduction du concept d'adresse ce qui dans notre esprit serait une étape dangereuse. Par exemple, un élément défini comme pointant sur un enregistrement dans un fichier personnel pourrait contenir un numéro de sécurité sociale qui est contenu dans chaque enregistrement du fichier et identifie de façon univoque cet enregistrement. En général un pointeur est un élément d'information qui peut être utilisé pour identifier de façon univoque un autre élément.

Bien que l'approche du pointeur donne la plus grande souplesse, elle le fait au prix de la relégation de la plus grande partie du travail chez l'utilisateur ainsi qu'en limitant sévèrement le contrôle du système d'ordinateur de données sur les données. Une solution hybride est possible, dans laquelle sont fournis de nouveaux types d'agrégats de données ainsi qu'une forme restreinte de type de données de pointeur. Bien que l'approche à suivre soit toujours à l'étude, la conception du langage de données inclura une méthode pour exprimer les structures de données non hiérarchiques.

3.4 Rangement des données

Les listes sont généralement vues comme ordonnées. Il est cependant possible qu'une liste puisse être utilisée pour modéliser une collection dynamique d'éléments similaires qui ne sont pas vus comme ordonnés. Le cas non ordonné est important, en ce que, connaissant cette information, l'ordinateur de données peut être plus efficace car de nouveaux membres peuvent être ajoutés chaque fois que c'est utile.

Une liste peut être ordonnée d'un certain nombre de façons. Par exemple, l'ordre d'une liste peut être fondé sur le contenu de ses membres. Dans le cas le plus simple, cela implique le contenu d'un élément de données de base. Par exemple, une liste de structures contenant des informations sur les employés d'une compagnie peut être ordonnée sur le composant qui contient le numéro de sécurité sociale de l'employé. Des critères de rangement plus complexes sont possibles. Par exemple, la même liste pourrait être dans l'ordre alphabétique du nom de famille de l'employé. Dans ce cas, la relation d'ordre est une fonction de deux éléments, le nom de famille et le prénom. L'utilisateur peut aussi vouloir définir son propre schéma de rangement, même pour les rangements fondés sur des éléments de données de base. Un rangement pourrait être fondé sur l'intitulé du travail de l'employé et pourrait même utiliser des données auxiliaires (c'est-à-dire, des données externes à la liste). Il est aussi possible de tenir une liste dans l'ordre des insertions. Dans le cas le plus général, l'utilisateur peut définir de façon dynamique cet ordre par la spécification de l'endroit où un élément doit être placé au titre de sa demande d'insertion. Dans tous les cas ci-dessus, les données peuvent être conservées en ordre ascendant ou descendant.
En plus de tenir une liste dans un certain ordre, il est possible de définir un ou plusieurs ordres "imposés" sur une liste. Ces ordres doivent être fondés sur le contenu des membres d'une liste. Cette situation est similaire au concept de données virtuelles (voir ci-dessous) en ce que la liste n'est pas tenue physiquement dans un ordre donné, mais restituée comme si elle l'était. Les rangements de ce type peuvent être formés de façon dynamique (voir la discussion sur "set" an paragraphe 3.8 sur les données virtuelles). Les ordres imposés peuvent être réalisés par la maintenance de structures auxiliaires (voir la discussion du paragraphe 3.9 "Représentation interne") ou par l'utilisation d'une stratégie de tri sur les restitutions. Un gros travail a été accompli à l'égard de la mise en œuvre effective de la maintenance et de l'imposition des ordres sur les listes. Ce travail est décrit dans le document de travail n° 2.

3.5 Intégrité des données

Une caractéristique importante de tout système de gestion de données est sa capacité à assurer l'intégrité des données. Les données ont besoin d'être protégées contre les manipulation erronées des personnes et contre les défaillances du système.

Le langage de données fournira des vérifications automatiques de validité. De nombreuses nuances doivent être fournies pour que les compromis appropriés puissent se faire entre le degré d'assurance et le coût de validation. L'utilisateur du langage de données sera capable de demander une validation constante, où les vérifications de validité sont faites chaque fois que les données sont mises à jour ; une validation sur accès, où les vérifications de validité sont effectuées lorsque les données sont référencées mais avant qu'elles ne soient restituées ; une validation programmée à intervalles réguliers, où les données sont vérifiés à intervalle régulier ; une validation en arrière plan, où le système va faire les vérifications pendant ses temps morts ; et la validation à la demande. La validation constante et la validation sur accès sont en fait des cas particuliers du concept plus général de validation déclanchée par un événement. Dans ce cas, l'utilisateur spécifie un événement qui va causer l'invocation des procédures de validation des données. Cette caractéristique peut être utilisée pour accomplir des choses comme la validation à la suite d'un "lot" de mises à jour. De même, certains mécanismes pour spécifier des combinaisons de ces types seraient utiles.

Pour que certaines des techniques de validation des données soient efficaces, il peut être nécessaire de conserver quelques informations de "tenue de compte" de la validation des données avec les données elles-mêmes. Par exemple, des informations qui peuvent être utilisées pour déterminer si un éléments a été vérifié depuis sa dernière mise à jour peuvent être utilisées pour causer une validation sur accès si il n'y a pas eu de validation en arrière plan récente. L'ordinateur de données peut fournir une maintenance automatique facultative de ces sortes d'informations particulières.

Afin que le système d'ordinateur de données s'assure de la validité des données, l'utilisateur doit définir ce qui est valide. Deux types de validation peuvent être demandés. Dans le premier cas, l'utilisateur peut dire à l'ordinateur de données qu'un élément de données spécifique ne peut prendre qu'une valeur parmi un ensemble spécifique. Par exemple, le composant couleur d'une structure peut seulement supposer les valeurs 'rouge', 'vert', ou 'bleu'. L'autre cas est lorsque une certaine relation doit exister entre les membres d'un agrégat. Par exemple, si le composant "sexe" d'une structure est "male" la valeur du composant "enceinte" doit être 0.

La validation des données n'est que la moitié du tableau de l'intégrité des données. L'intégrité des données implique des

méthodes de restauration des données endommagées. Cela exige d'entretenir des informations redondantes. On fournira des dispositifs pour rendre le système d'ordinateur de données responsable de la maintenance de données redondantes et éventuellement même de restauration automatique des données endommagées. Nous avons exposé à la Section 2 l'utilisation éventuelle de l'ordinateur de données pour la sauvegarde de fichiers. Tous les dispositifs qui sont fournis à cette fin seront aussi disponibles comme méthode de conservation des informations de sauvegarde pour la restauration des fichiers qui résident sur l'ordinateur de données.

3.6 Confidentialité

Le langage de données devra aussi fournir des capacités extensives de protection de la confidentialité. Sous sa forme la plus simple, un verrou de confidentialité est fourni au niveau du fichier. Le verrou est ouvert avec une clé sous forme d'un mot de passe. Associé à cette clé est un ensemble de privilèges (lecture, mise à jour, etc.). Deux degrés de généralité sont recherchés. La confidentialité devrait être disponible à tous les niveaux des données. Donc, des groupes de données en rapport, y compris des groupes de fichiers, pourraient être rendus privés par la création de répertoires privés. Des champs d'enregistrements spécifiques pourraient également être rendus privés en ayant des composants privés dans la structure alors que d'autres composants de la structure sont visibles à une classe d'utilisateurs plus large (ou différente). On aimerait aussi que l'utilisateur soit capable de définir son propre mécanisme. De cette façon, des mécanismes très personnalisés, complexes, et donc sûrs, peuvent être définis. Des dispositifs tels que "chacun peut voir son propre salaire" seraient aussi possibles.

3.7 Conversion

De nombreux types de données sont en rapport, en ce que certaines, ou toutes, les valeurs possibles d'un type de données ont une traduction "évidente" en valeurs d'un autre type. Par exemple, le caractère "6" a une traduction naturelle en l'entier 6, ou la chaîne de six caractères "abc " (avec trois blancs en queue) a une traduction naturelle en la chaîne de quatre caractères "abc " (un blanc en queue). Le langage de données fournira des capacités de conversion pour les traductions standard, d'utilisation courante. Ces conversions peuvent être invoquées explicitement par l'utilisateur ou invoquées implicitement lorsque les données d'un type sont nécessaires pour une opération mais que des données d'un autre type sont fournies. Dans le cas d'une invocation implicite de conversion de données, l'utilisateur aura le contrôle sur le fait que la conversion intervienne pour un élément de données particulier. Plus généralement nous aimerions fournir une facilité par laquelle l'utilisateur puisse spécifier les conditions qui déterminent quand un élément doit être converti. Aussi, l'utilisateur devrait être capable de définir ses propres opérations de conversion, soit pour une conversion entre des types qui ne sont pas fournis par le système d'ordinateur de données, soit en outrepassant l'opération de conversion standard pour certains ou tous les éléments d'un type donné.

3.8 Données virtuelles et dérivées

Souvent, des informations importantes pour les utilisateurs des données sont incorporées dans les données elles-mêmes plutôt qu'entretenues explicitement. Par exemple, la valeur du dollar dans un intérêt individuel pour un fichier des actionnaires d'une compagnie. Comme la valeur de la compagnie change fréquemment, il n'est pas pratique de conserver cette information avec chaque enregistrement. Il est utile d'être capable d'utiliser le fichier comme si les informations de ce type faisaient partie de chaque enregistrement. Lorsque on se réfère au champ Valeur du dollar d'un enregistrement, le système d'ordinateur de données va automatiquement utiliser les informations de l'enregistrement, comme un pourcentage des titres de propriété de la compagnie, éventuellement en conjonction avec des informations qui ne font pas partie de l'enregistrement mais sont conservées ailleurs, comme les immobilisations de la compagnie, pour calculer la valeur en dollars. De cette façon, l'utilisateur des données n'a pas besoin de se soucier du fait que ces informations ne sont pas réellement conservées dans l'enregistrement.

Un "ensemble" *(set)*, qui est un type spécifique de conteneur virtuel en langage de données, mérite une mention particulière. Un "ensemble" est une liste virtuelle. Par exemple, supposons qu'il y ait une liste réelle de gens qui représentent un échantillon d'une population. Par "données réelles" on veut dire des données physiquement mémorisées dans l'ordinateur de données. Un "ensemble" pourrait être défini comme contenant tous les membres de cette liste qui possèdent une automobile. Le concept de "ensemble" donne un dispositif puissant pour voir des données comme appartenant à plus d'une collection sans duplication physique. Les "ensembles" sont aussi utiles en ce qu'ils peuvent être formés de façon dynamique. Dans une liste réelle, les "ensembles" fondés sur cette liste peuvent être créés sans avoir été préalablement décrits.

Comme mentionné plus haut, les données virtuelles peuvent être très économiques. Ces économies peuvent devenir très importantes par rapport à l'utilisation des ensembles. Les économies ne se trouvent pas seulement par rapport aux exigences de stockage mais aussi par rapport à l'efficacité du traitement. Les temps de traitement peuvent être réduits du fait que les calculs ne seront faits que lors de l'accès aux données. La capacité à obtenir un fonctionnement efficace grâce à l'optimisation augmente lorsque des données virtuelles sont définies par d'autres données virtuelles. Pour des ensembles, de grosses économies peuvent être réalisées par une "optimisation" directe des calculs incorporés.

Les idées ci-dessus seront éclairées par des exemples. Nous avons créé un ensemble des possesseurs d'automobile, A, un ensemble des propriétaires fonciers, HA, peut être défini sur la base de A. Les membres de HA peuvent être sortis de façon très efficace, en une seule étape, en restituant les gens qui sont à la fois des propriétaires d'automobile et des propriétaires fonciers. C'est plus efficace que de sortir réellement l'ensemble A puis de l'utiliser pour créer HA. Ceci est vrai lorsque un des éléments d'information ou les deux (propriétaire d'automobile et propriétaire foncier) sont indexés (voir la discussion du paragraphe 3.9 sur la représentation interne) aussi bien que lorsque ni l'un ni l'autre ne sont indexés.

Les mêmes gains sont réalisés lorsque des opérations sont nécessaires sur des données virtuelles. Par exemple, si un ensemble, H, a été défini comme l'ensemble des propriétaires fonciers sur la base de la liste originale des gens, l'ensemble HA pourrait être défini comme l'intersection (voir la discussion du paragraphe 3.13 sur les opérateurs) de A et de H. Dans ce cas aussi, HA peut être calculé en une étape. L'utilisation des ensembles permet à l'utilisateur de demander des manipulations de données sous une forme proche de sa vision conceptuelle, laissant le problème du traitement effectif de sa demande à l'ordinateur de données.

Un autre utilisation des données virtuelles est la réalisation du partage de données. Un élément pourrait être défini, de façon virtuelle, comme le contenu d'un autre élément. Si aucune restriction n'est établie sur ce que peut être cet élément, nous sommes capables de définir deux chemins d'accès aux mêmes données. Et donc, les données peuvent être subordonnées à deux ou plusieurs structures agrégées. Dit d'une autre façon, il y a deux chemins ou plus d'accès aux données. Cette capacité peut être utilisée pour modéliser des données qui font partie de plus d'une relation de données. Par exemple, deux fichiers pourraient avoir les mêmes enregistrements sans qu'on conserve de copies dupliquées.

Il est aussi possible, via le partage des données, de regarder les données de différents points de vue. Les données partagées se comportent différemment selon la façon (et en fin de compte, par qui) on y accède. Bien que la capacité à avoir plusieurs chemins d'accès aux mêmes données et la capacité d'avoir des données qui sont calculées sur l'accès fassent toutes deux partie de la capacité générale des données virtuelles, le langage de données les fournira probablement comme des dispositifs distincts, car elles ont des caractéristiques d'utilisation différentes.

Les données dérivées sont similaires aux données virtuelles en ce qu'elles sont des données redondantes qui peuvent être calculées à partir d'autres informations. À la différence des données virtuelles, elles sont conservées physiquement. L'utilisateur peut choisir entre des données virtuelles et des données dérivées par suite de compromis fondés sur le coût estimé du calcul, de la fréquence des mises à jour, du coût estimé du stockage, et de la fréquence d'accès. Par exemple, supposons qu'un fichier contienne une liste de budgets de divers projets dans un département. Le budget du département peut être calculé comme une fonction des projets de budgets individuels. Cette information pourrait être définie comme des données dérivées car on peut supposer qu'elles ne seront pas mises à jour fréquemment (par exemple, une fois par an) alors qu'on peut supposer qu'on va y accéder relativement souvent.

Les options qui seront fournies donneront à l'utilisateur le contrôle sur le moment où calculer les données dérivées. Ces options seront similaires à celles fournies pour le contrôle des opérations de validité des données. Les concepts de validation des données et de données dérivés sont similaires en ce que certaines opérations doivent être effectuées sur les données en question. Dans le cas de la validation des données, les informations dérivées sont la condition des données.

3.9 Représentation interne

Jusqu'à présent, nous n'avons exposé que les aspects de haut niveau des données, le niveau logique. Comme les données doivent, à tout moment, résider sur un appareil physique, une représentation des données doit être choisie. Dans certains cas, il est approprié de laisser ce choix au système d'ordinateur de données. Par exemple, la représentation des informations qui sont utilisées dans le processus de transmission d'autres données, mais qui résident elles-mêmes seulement dans l'ordinateur de données, peut n'avoir aucune importance pour l'utilisateur.

Cependant, il est important que l'utilisateur soit capable de contrôler le choix de la représentation. Dans toute application qui exige surtout la transmission des données plutôt que leur interprétation par l'ordinateur de données, les données devraient être conservées sous une forme cohérente avec le système qui communique avec l'ordinateur de données. Par rapport aux types de données de base, le langage de données va fournir la plupart des représentations couramment utilisées dans les systèmes avec lesquels il interagit. Pour certains types (par exemple, la virgule fixe) cela sera réalisé en fournissant une description paramétrique (par exemple, une convention de signe, de taille) de la représentation. Dans d'autres cas (par exemple, de virgule flottante) des représentations spécifiques seront offertes (par exemple, le système 360 de virgule flottante courte, le système 360 de virgule flottante longue, la virgule flottante pdp-10, etc.).

Un autre aspect du problème de la représentation interne concerne les structures agrégées. La méthode choisie pour représenter les structures agrégées peut largement affecter le coût de manipulation des données. L'utilisateur doit avoir le contrôle sur cette représentation car lui seul a une idée de la façon dont les données vont être utilisées. Le langage de données va fournir diverses options de représentation qui vont permettre une mise en œuvre efficace des structures de

données. Cela inclut la disponibilité de structures auxiliaires, conservées automatiquement par le système d'ordinateur de données. Ces structures peuvent être utilisées pour effectuer une restitution efficace de sous-ensembles de collections de données, sur la base du contenu des membres (c'est-à-dire, du concept bien connu d'indices), une maintenance efficace de l'ordre d'une collection de données, une maintenance des informations redondantes pour les besoins de l'intégrité des données, et un traitement efficace des données partagées dont les caractéristiques comportementales dépendent du chemin d'accès. Il sera utile de noter ici que l'effort de conception du langage de données va tenter de fournir des méthodes par lesquelles l'utilisateur des données puisse décrire l'utilisation espérée de ses données, afin que les détails de la représentation interne puissent être abandonnés à l'ordinateur de données.

3.10 Attributs et classes de données

Le type d'un élément détermine les opérations qui sont valides sur cet élément et ce qu'elles signifient. Les "attributs de données" sont des raffinements du type des données. Les attributs des données affectent la signification des opérations. Par exemple, on voudrait fournir l'option de définir des éléments à virgule fixe à mesurer. Le facteur d'échelle, dans ce cas, serait un attribut de données à virgule fixe. Il affecte la signification des opérations sur ces données. Le concept d'attribut est utile en ce qu'il permet des informations concernant la manipulation d'un élément à associer à l'élément plutôt que l'invocation de toutes les opérations sur cet élément.

Le concept d'attribut peut être appliqué à un agrégat aussi bien qu'aux données de base. Par exemple, un attribut d'une liste pourrait définir où un nouveau membre sera inséré. Les options pourraient être de l'insérer au début de la liste, à la fin de la liste, ou dans un ordre fondé sur le contenu du membre. L'ajout d'un nouveau membre à une liste avec un des attributs ci-dessus pourrait être fait en produisant une simple demande d'insertion sans avoir à spécifier où le nouveau membre est à insérer.

Le concept de "classe de données" est en fait l'inverse du concept d'attribut de données. Une classe de données est une collection de types de données. Le concept de classe de données permet la définition des opérations, indépendamment du type spécifique d'un élément de données. Par exemple, en définissant la classe des données arithmétiques comme étant composée des types de données à virgule fixe et à virgule flottante, les opérateurs de comparaison ("égal", "inférieur à", etc.) peuvent être définis comme opérant sur des données arithmétiques, indépendamment du fait qu'elles ont une virgule fixe ou flottante. Aussi le concept d'agrégat de données peut être vu comme des répertoires, listes, etc. qui mettent en application une classe. Comme il y a des opérations définies sur des données arithmétiques, il y a aussi des opérations définies sur des agrégats arbitraires.

La relation inverse entre les classes de données et les attributs de données est très forte. Par exemple, le concept de liste peut être vu comme une classe de données, mettant en application tous les types de listes (par exemple, des listes d'entiers, des listes de chaînes de caractères, etc.) indépendamment des types de leurs membres. Le type des membres d'une liste (par exemple, entier, chaîne de caractères, etc.) est alors vu comme un attribut. Les attributs et classes de données sont aussi des concepts relatifs. Alors que le concept de liste peut être vu comme une classe de données, il peut aussi être vu comme un attribut, par rapport au concept d'agrégat de données.

3.11 Description des données

Une "description de données" est une déclaration des propriétés (voir au paragraphe 3.10 la discussion sur les attributs) d'un élément de données. Des exemples de propriétés qui sont enregistrées dans une description sont le nom d'un élément, sa taille, son type de données, sa représentation interne, des informations de confidentialité, etc.

Le langage de données contiendra des mécanismes pour spécifier les descriptions des données. Ces descriptions seront traitées par l'ordinateur de données, et utilisées chaque fois que l'élément de données est référencé. L'utilisateur ne sera capable de créer physiquement des données qu'en spécifiant d'abord leur description. Les propriétés d'une description peuvent être divisées en groupes selon leur fonction. Certaines ont la fonction de spécifier les détails de représentation, ce qui n'intéresse pas la plupart des utilisateurs, alors que d'autres, comme le nom, sont d'un intérêt presque universel.

Toutes les données d'utilisateur font partie d'une structure (d'utilisateur ou de système) de données plus large. Les structures qui contiennent les données établissent un chemin d'accès aux données. Dans le processus du suivi de ce chemin, le système d'ordinateur de données doit accumuler une description complète de l'élément de données. Par exemple, la description d'un élément de données d'un répertoire peut se trouver associée avec ce nœud du répertoire. Les membres d'une liste ou d'une matrice sont décrits au titre de la description de la liste ou de la matrice. On doit parler de deux exceptions vraisemblables. D'abord, alors que les aspects de données peuvent (à la demande de l'utilisateur) être laissés au système, ces aspects sont quand même décrits ; ils le sont par le système. Comme exposé plus haut, certaines données seront, dans une certaine mesure, auto décrites (par exemple, les membres de listes mixtes). Cependant, elles sont pleinement décrites dans certaines structures englobantes, en ce qu'une méthode pour déterminer la pleine description est décrite.

Il vaut de noter ici que plus tôt on trouve une description complète dans le chemin d'accès, plus efficace sera vraisemblablement l'ordinateur de données dans le traitement des demandes qui manipulent un élément de données.

Cependant, la capacité à avoir des données dont la description complète n'existe pas à de hauts niveaux du chemin d'accès donne une plus grande souplesse à la définition des structures de données.

3.12 Référence des données

Les données ne peuvent être manipulées que si elles peuvent être référencées. De la même façon que des données n'existent que si elles sont décrites, elle ne peuvent exister que si il y a un chemin d'accès aux données. La méthode de référence aux données est de définir le chemin d'accès aux données. Comme mentionné ci-dessus, il y a une méthode de référence à tout élément relatif à l'agrégat de données qui le contient. Les nœuds de répertoires et les composants des structures sont référencés via le nom associé au nœud ou composant. Les membres de matrices sont référencés via l'indice associé au membre. Les membres des listes sont référencés via une méthode qui spécifie la position du membre ou en identifiant de façon univoque le membre par son contenu. Pour référencer tout élément de données arbitraire, le chemin d'accès doit être pleinement défini par une définition explicite ou implicite de chaque liaison dans la chaîne. Dans le cas de données virtuelles, il y a une liaison implicite supplémentaire dans la chaîne, qui est celle de la méthode employée pour obtenir les données provenant des autres éléments de données. Noter aussi que si des pointeurs sont fournis (voir l'exposé sur les capacités relationnelles générales) ils peuvent aussi servir de liaison dans la chaîne d'accès à un élément.

La conception du langage de données va faciliter la résolution du problème (et en réduire le coût) du référencement des éléments de données en fournissant les méthodes par lesquelles une partie du chemin d'accès peut être implicitement définie. Par exemple, le langage de données va fournir un concept de "contexte". Durant le cours de l'interaction avec l'ordinateur de données, des niveaux de contexte peuvent être établis de telle sorte que les données puissent être référencées directement, dans le contexte. Par exemple, à l'initialisation d'une session, l'utilisateur peut (en fait, sera probablement obligé de) définir un répertoire qui sera le contexte de cette session. Tous les éléments subordonnés à ce répertoire peuvent être référencés directement dans ce contexte. Une autre caractéristique sera la qualification partielle. Chaque niveau de structure n'a pas besoin d'être mentionné pour référencer un élément incorporé dans un empilement profond de structures. Seuls les niveaux intermédiaires qui suffisent à identifier l'élément de façon univoque doivent être spécifiés.

3.13 Opérations

Ce paragraphe expose les fonctions incorporées du langage de données dont l'importance est centrale pour la manipulation des données : les fonctions qui opèrent sur des éléments, les fonctions qui opèrent sur des agrégats, les fonctions primitives et les fonctions de haut niveau.

Parmi les primitives qui opèrent sur les éléments, les plus intéressantes sont les fonctions d'allocation, de comparaisons, de logique, d'arithmétique et de conversion.

La primitive d'allocation transfère une valeur d'un élément à un autre ; ces éléments doivent être du même type. Lorsque ils sont de types différents, une conversion doit être effectuée, ou alors une forme non primitive d'allocation est impliquée.

Les opérateurs de comparaison acceptent une paire d'éléments du même type, et retournent un objet booléen qui indique si une condition donnée est obtenue ou non. Le type détermine combien de conditions différentes peuvent être comparées. Une paire d'éléments numériques peut être comparée pour voir lequel est le plus grand, alors qu'une paire d'éléments non interprétés ne peut être comparée que pour égalité. En général, un concept de "plus grand que" n'est incorporé pour un type de données que si il constitue un concept très largement appliqué. Les opérateurs de comparaison sont utilisés dans la construction de conditions d'inclusion lors de la définition de sous ensembles d'agrégats de données.

Le résultat d'une opération de comparaison est un élément booléen : un de ceux dont la valeur est soit VRAI, soit FAUX. Les primitives logiques sont fournies et des fonctions booléennes généralisées peuvent être construites à partir d'elles. Avec les opérateurs logiques et de comparaison, des conditions complexes pour l'inclusion des objets dans des ensembles peuvent être spécifiées.

Les opérateurs arithmétiques seront disponibles pour la manipulation de données numériques. Ici, on ne s'intéresse pas au calcul généralisé, mais à des applications d'arithmétique dans des choix de données, d'allocation d'espace, de calcul en indice, de contrôle d'itération, etc.

La conversion est une partie importante de la traduction généralisée de données, et nous nous intéressons à la fourniture d'une facilité de conversion incorporée substantielle. En particulier, nous voulons fournir un sous-programme système efficace pour chaque fonction de conversion "standard" ou largement utilisée. Sont d'une importance particulières les conversions de et vers les données de chaînes de caractères, en représentation de chaîne de caractères, par exemple, d'éléments numériques, il y a de nombreux formats possibles qui correspondent à un seul type de données. La conversion entre jeux de caractères et le traitement des bourrages et des troncatures sont vus comme des problèmes de conversion.

Il y a deux classes principales d'opérateurs de primitives définis sur les agrégats : ceux qui se rapportent à la référence aux

données (voir au paragraphe précédent) et ceux qui ajoutent et suppriment des composants. Changer un composant existant est réalisé par une allocation, et c'est une opération sur le composant, et non sur l'agrégat.

L'ajout et la suppression de composants ne sont définis que pour les agrégats qui ne sont pas par nature de composition statique. Donc, on peut ajouter un composant à une LISTE, mais pas à une MATRICE. Pour spécifier la suppression, il est nécessaire de spécifier quel composant est à supprimer, et à partir de quel agrégat (dans le cas où il est partagé). L'ajout exige la spécification du nouveau composant, de l'agrégat, et parfois des informations auxiliaires. Par exemple, certains types d'agrégat vont permettre l'ajout de nouveaux composants n'importe où dans la structure ; dans celle-ci, une position doit être indiquée par rapport à tous les composants existants.

Il est souvent souhaitable d'opérer sur certains des membres d'une liste, ou de traiter un groupe de membres comme une liste de plein droit. Par exemple, il peut arriver couramment de transmettre à un programme distant pour analyse l'histoire médicale des patients qui développent des maladies cardiaques avant l'âge de 30 ans. Cela peut être seulement quelques uns d'une longue liste de patients.

Dans ce cas, l'opération à effectuer est la transmission au système distant ; cette opération est effectuée sur plusieurs membres de la liste des patients. Ceux qui sont à transmettre sont vus comme un "ensemble" ; l'ensemble est spécifié comme contenant tous les membres d'une liste donnée qui satisfait deux conditions : (1) âge inférieur à 30 ans et (2) avoir un affection cardiaque.

Les ensembles peuvent être définis explicitement, ou implicitement, simplement avec les mécanismes de référence appropriés. La "définition" d'un ensemble est distincte de "l'identification comme membre", qui est distincte de "l'accès à la qualité de membre". La définition implique de spécifier les candidats d'un ensemble et de spécifier une règle selon laquelle les membres de l'ensemble peuvent se distinguer des non membres ; par exemple, une condition d'inclusion telle que "moins de 30 ans avec affection cardiaque". L'identification implique une application effective de la règle à tous les candidats à l'adhésion. Lorsque l'adhésion a été identifiée, elle peut être comptée, mais il n'y a pas nécessairement d'accès aux données elles-mêmes. Lorsque on accède à un membre, on peut opérer sur son contenu.

Les primitives pour accomplir chacune de ces opérations sur un ensemble seront fournies ; cependant, il sera normalement optimal pour l'ordinateur de données de déterminer quand chaque étape doit être effectuée. Pour permettre aux utilisateurs de fonctionner à un niveau auquel l'ordinateur de données ait une efficacité optimale, des opérateurs de niveau supérieur seront fournis sur les ensembles. Certains d'entre eux sont des opérateurs logiques, comme l'union et l'intersection. Elles entrent et sortent des ensembles. Un opérateur qui complète un ensemble est aussi disponible (dans la mesure où la définition établit tous les candidats possibles, un ensemble a toujours un complément bien défini).

Ces opérateurs de niveau supérieur peuvent être appliqués à tout ensemble défini ; les membres de l'ensemble n'ont pas besoin d'être identifiés ni atteints. Le système va effectuer de telles opérations sans réellement accéder aux membres si il le peut.

Certains des autres opérateurs sur des ensembles comptent les membres, partagent un ensemble en un ensemble d'ensembles, unifient un ensemble d'ensembles en un ensemble. Un ensemble peut être utilisé pour faire référence à un autre ensemble, pourvu qu'il y ait une façon bien définie d'identifier les membres du second ensemble à partir du premier. Par exemple, un ensemble C peut contenir tous les enfants qui travaillent mal en classe. Un ensemble F peut être défini, où les membres de F sont des enregistrements sur les familles qui ont un enfant dans l'ensemble C.

Certaines autres opérations utiles sur les ensembles sont : ajouter tous les membres d'un ensemble à un agrégat, supprimer tous les membres d'un ensemble (il est fréquent qu'un changement aussi massif puisse être effectué de façon bien plus efficace qu'en demandant individuellement les mêmes changements) changer tous les membres d'un ensemble d'une certaine façon.

Un ensemble peut être mis en liste en accédant réellement à chaque membre et en le collectant physiquement.

Certaines des opérations sur les listes sont : l'enchaînement de listes en listes plus grandes, la division d'une liste en des listes plus petites, le tri d'une liste, la fusion d'une paire de listes ordonnées (en préservant l'ordre).

Il n'est pas prévu de faire une énumération complète des opérations de haut niveau, mais plutôt de suggérer. Nous prévoyons d'incorporer des fonctions de niveau élevé pour des opérations qui sont utilisées de façon très courante, et peuvent être mises en œuvre dans le système significativement mieux que par les utilisateurs à l'aide du langage. Pour la plupart des fonctions mentionnées ici, des connaissances considérables sont accumulées sur les bonnes mises en œuvre. En particulier, les techniques utilisées pour inverser l'accès de fichier fournissent de nombreuses opérations d'ensembles à effectuer sans accès réel aux données.

3.14 Contrôle

Les caractéristiques de contrôle du langage de données sont aux opérations de base ce que les agrégats de données sont aux éléments de données de base. Les caractéristiques de contrôle sont utilisées pour créer des demandes complexes à partir des demandes de base fournies par le langage de données.

Les demandes conditionnelles permettent à l'usager d'altérer le flux de demandes normales en spécifiant que certaines demandes sont à exécuter dans certaines conditions. Le langage de données général va fournir la capacité de choisir au plus une parmi un certain nombre de demandes à faire sur la base d'un ensemble de conditions ou de la valeur d'un certain élément. Dans sa forme la plus simple, la condition permet une exécution facultative d'une demande donnée.

Les demandes itératives causent l'exécution d'une demande (appelée le corps) un nombre fixe ou variable de fois ou jusqu'à satisfaction d'une certaine condition. Le langage de données fournira des demandes itératives qui permettront d'effectuer des manipulations similaires sur tous les membres d'une structure agrégée aussi bien que le type standard de demande itérative fondée sur le comptage. En fournissant la capacité d'exprimer directement des manipulations sur des agrégats qui exigent le traitement de tous les éléments subordonnés à l'agrégat, l'ordinateur de données peut être plus efficace dans le traitement des demandes des utilisateurs. Par exemple, un processus, défini par l'usager, de conversion qui opère sur des chaînes de caractères, peut être mis en œuvre beaucoup plus efficacement si l'ordinateur de données est explicitement informé de ce que le processus exige un traitement séquentiel des caractères. Le langage de données va aussi traiter l'itération en parallèle. Par exemple, l'usager sera capable de spécifier des opérations qui exigent le séquençage à travers deux listes en parallèle ou plus. Cela sera fait si le contenu d'un fichier doit être mis à jour sur la base d'un fichier d'informations de corrections.

Les demandes composées sont des collections de demandes qui se comportent comme une seule. Elles sont principalement fournies pour permettre un traitement conditionnel ou itératif sur plus d'une déclaration. Les demandes composées fournissent aussi des points de référence de demandes qui peuvent être utilisés pour contrôler le flux de traitement des demandes. C'est à dire que les demandes composées peuvent être "nommées". L'utilisateur de langage de données sera capable de spécifier des informations de contrôle qui vont causer l'excitation conditionnelle d'une demande composée. En permettant de la nommer, l'utilisateur peut causer l'excitation d'un nombre quelconque de demandes composées entrées précédemment.

Nous ne prévoyons pas de fournir la capacité traditionnelle "goto". En n'incluant pas une demande "goto", les chances d'un fonctionnement efficace (via l'optimisation) de l'ordinateur de données sont accrues. On espère aussi, de cette façon, forcer les utilisateurs du langage de données à spécifier les manipulations de données de façon claire.

Deux formes de la demande composée seront fournies, ordonnée et non ordonnée. Dans le cas non ordonné, l'utilisateur informe l'ordinateur de données que les demandes peuvent être effectuées dans n'importe quel ordre. Cela devrait permettre à l'ordinateur de données de travailler plus efficacement et pourrait même permettre en traitement en parallèle.

Durant une session de l'ordinateur de données il est vraisemblable qu'un utilisateur va éprouver le besoin de données temporaires. C'est à dire, de données dont il faut se souvenir, à court terme, d'informations qui sont nécessaires pour le traitement des demandes. Ce court terme peut être une session ou une petite partie d'une session. Le langage de données fournira une facilité de données temporaires. Les données temporaires seront faciles à créer, utiliser et éliminer. Cela sera réalisé en permettant au système de prendre (facultativement) des décisions concernant les données. Par exemple, la représentation d'un élément entier temporaire ne sera souvent d'aucun intérêt pour l'utilisateur. Certaines caractéristiques qui sont fournies pour des données permanentes seront réputées non pertinentes par rapport aux données temporaires.

Les données temporaires seront associées à une collection de demandes dans ce que nous appellerons un bloc. Un bloc ne sera pas différent d'une demande composée sauf que ces données sont définies avec la demande qui le compose et sont automatiquement créées à l'entrée du bloc et détruites à la sortie du bloc.

3.15 Extensibilité

Le but du langage de données est de fournir des facilités de structure de données à deux niveaux. À un niveau, l'utilisateur peut tirer parti des capacités de données de haut niveau qui feront le plus gros du travail de gestion de ses données automatiquement et permettront à l'ordinateur de données de fonctionner plus efficacement dans certains cas, car il a reçu le contrôle des données. À un autre niveau, cependant, les caractéristiques qui sont fournies vont permettre à l'utilisateur de décrire son application en termes de concepts de primitives. De cette façon, l'utilisateur d'ordinateur de données peut composer une grande variété de constructions de données et a une grande souplesse par rapport aux manipulations qu'il peut effectuer sur ses données. En interagissant avec l'ordinateur de données au niveau des primitives, l'utilisateur peut aussi exercer un contrôle sur les méthodes employées par l'ordinateur de données dont il peut résulter une utilisation plus efficace des ressources pour des applications non standard. Le langage des données fournira des caractéristiques qui permettront à l'utilisateur de créer un environnement dans lequel le système d'ordinateur de données paraît fournir des caractéristique taillées sur mesure pour son application.

Les caractéristiques de contrôle exposées ci-dessus permettent à l'usager d'étendre les opérations disponibles sur les données par une composition appropriée des opérations. Le langage de données va fournir une méthode pour définir une demande composite comme étant une nouvelle demande (appelée une "fonction"). De cette façon, une nouvelle opération peut être définie une fois sur des données spécifiques puis utilisée de façon répétée. Afin que l'utilisateur puisse définir des opérations générales, le langage de données fournira des fonctions qui peuvent être paramétrées. C'est à dire, des fonctions qui seront non seulement capables de fonctionner sur des données spécifiques mais pourront être définies comme travaillant sur toutes données d'un type spécifique. Cette capacité ne sera pas limitée aux types de données de base (par exemple, les entiers) ou même à des types agrégés spécifiques (par exemple, une matrice d'entiers) mais incluront aussi la capacité de définir des fonctions qui opèrent sur des classes de données. Par exemple, on pourra définir des fonctions qui opèrent sur des listes indépendamment du type des membres de la liste. La capacité d'étendre et modifier les fonctions existantes ainsi que de créer de nouvelles fonctions sera aussi fournie. Cela inclut d'étendre les types de données pour lesquels une fonction est définie ou de modifier le comportement d'une fonction pour certains types de données.

Comme avec les opérations, les agrégats de données exposés ci-dessus permettent à l'utilisateur d'étendre les types de données primitifs par une composition appropriée. Par exemple, une matrice à deux dimensions d'entiers peut être créée en construisant une matrice de matrices d'entiers. La situation pour les types de données est analogue à celle des opérations. Le langage de données donnera la possibilité de définir une composition de données comme nouveau type de données. La capacité à définir des structures générales de données sera aussi fournie essentiellement en paramétrant la définition des nouvelles données. Cela va permettre de définir le concept général de matrice à deux dimensions comme une matrice de matrices. Une fois défini, on peut créer des matrices à deux dimensions d'entiers, des matrices bidimensionnelles de booléens, etc. Comme avec les fonctions, il est aussi nécessaire d'étendre ou modifier les types de données existants. On peut vouloir étendre les attributs qui s'appliquent à un type de données particulier, ou ajouter de nouveaux attributs, ou ajouter de nouveaux choix pour les attributs existants.

Les caractéristiques de contrôle peuvent aussi être étendues. Des caractéristiques de contrôle particulières peuvent être nécessaires pour traiter une structure de données d'une façon particulière pour un processus de structure de données définie par l'utilisateur. Par exemple, si une structure de données de type arborescence a été définie en termes de listes de listes, l'utilisateur peut vouloir définie une fonction de contrôle qui cause une opération spécifiée sur chaque élément d'une arborescence spécifiée. Comme avec les types et fonctions de données, il faut être capable de modifier et étendre les caractéristiques de contrôle existantes aussi bien que la capacité à en créer de nouvelles.

Le langage de données donnera la possibilité de traiter les descriptions et opérations de données à peu près de la même façon que sont traitées les données. On peut décrire et manipuler des descriptions et opérations de la même façon qu'on peut décrire et manipuler des données. Il est impossible de parler des types de données sans prendre en compte les opérations, et il est également impossible de parler des opérations sans comprendre les types de données sur lesquels elles opèrent. Afin que l'usager soit capable d'affecter le comportement du système d'ordinateur de données, la conception du langage de données inclura une définition du cycle de fonctionnement de l'ordinateur de données. Des définitions précises de tous les aspects des données (attributs, classes, relations des agrégats avec leurs éléments subordonnés, etc.) en termes d'interaction avec les opérations du langage de données seront données. De cette façon, l'ordinateur de données peut offrir des outils qui donneront à son utilisateur la capacité d'être un participant actif à la conception du langage de données qu'il utilise.

4. Modèle pour la sémantique du langage des données

Pour les besoins de la définition et de l'expérimentation de la sémantique du langage et des techniques de traitement du langage, nous avons développé un modèle d'ordinateur de données.
Les principaux éléments du modèle sont les suivants :

(1) un ensemble de fonctions primitives
(2) un environnement dans lequel les objets de données peuvent être créés, manipulés et supprimés, en utilisant les primitives
r3) une structure pour la représentation des collections des valeurs des données, leur description, leurs relations, et leur nom.
(4) un interpréteur qui exécute les primitives
(5) un compilateur qui entre les demandes dans un langage très simple, effectue les opérations de liaison et de macro expansion, et génère les appels aux primitives sémantiques internes.

Si nos efforts de modélisation réussissent, le modèle évoluera jusqu'à ce qu'il accepte un langage comme le langage de données dont les propriétés ont été décrites dans les sections 2 et 3 du présent document. Le processus de rédaction de la spécification finale exigera simplement de réconcilier les détails non modélisés avec la structure modélisée. Un détail relativement gros qui peut ne jamais coller au modèle est la syntaxe ; dans ce cas, la conciliation sera plus sollicitée ; cependant, nous croyons fortement que la structure sémantique devrait déterminer la syntaxe plutôt que le contraire, de sorte que nous serons dans la position propre à régler le problème.

En construisant un modèle pour chacun des éléments énumérés ci-dessus, nous "mettons en œuvre" le langage comme nous l'avons conçu, au sens très large. En effet, nous travaillons en laboratoire, plutôt que strictement sur le papier. Comme nous ne sommes pas concernés par les performances ou la facilité d'usage de l'ordinateur de données que nous construisons en laboratoire, nous sommes capables de construire sans être impliqués par les soucis qui consomment la plus grande partie du temps d'une mise en œuvre. Cependant, comme nous construisons et bricolons, plutôt que de simplement travailler sur le papier, nous avons quelques uns des avantages qui n'arrivent normalement qu'avec l'expérience de la mise en œuvre de ses idées.

Le modèle d'ordinateur de données est un programme, développé à ECL, qui utilise le langage EL1. Nous nous intéressons présentement au processus de développement du programme, et pas de le faire tourner. Notre principale exigence est d'avoir, avant l'existence du langage de données, une notation bien définie et souple dans laquelle spécifier les structures de données, les définitions et les exemples de fonctions. EL1 convient pour cela. Avoir un programme qui fonctionne réellement et agit comme un simple ordinateur de données est en fait un sous-produit de la spécification de la sémantique dans un langage de programmation. Il n'est pas nécessaire que le programme fonctionne, mais il fournit des caractéristiques intéressantes. Il améliore l'effet de "laboratoire", en faisant des choses comme compiler automatiquement les chaînes de primitives, en affichant l'état de l'environnement dans des exemples compliqués, en découvrant automatiquement les incohérences (sous la forme de bogues), et ainsi de suite.

Il y a deux raisons majeures pour que EL1 soit une notation convenable pour spécifier la sémantique du langage de données. L'une est que ces langages ont un certain nombre de points communs, à la fois dans les concepts et dans les buts de la description des données. (En partie, parce que EL1 lui-même a été une bonne source d'idées pour attaquer le problème du langage de données). Les deux langages mettent l'accent sur les opérations sur les données, indépendamment de la représentation sous-jacente. La seconde raison pour laquelle EL1 est une façon convenable pour spécifier le langage de données est que EL1 est extensible ; en fait, de nombreuses fonctions primitives pourraient être incorporées directement dans EL1 en utilisant les facilités d'extension. Parfois, nous avons choisi d'incorporer moins que ce que nous aurions pu, pour exposer les problèmes qui nous intéressent.

Jusqu'à présent, le modèle a été principalement utile pour exposer les problèmes conceptuels et les relations entre les décisions de conception. Aussi, parce qu'il comporte tant d'éléments du système complet (compilateur, interpréteur, environnement, etc.) il encourage à une analyse très complète de toute proposition.

En présentant le modèle dans cette section, nous avons choisi de mettre l'accent sur les idées et les exemples, plutôt que sur les définitions formelles en EL1. Cela parce que les idées sont plus permanentes et pertinentes pour l'instant (les formalismes changent assez fréquemment) et parce que nous pensons que les gens ne lisent les définitions formelles que pour voir les idées derrière elles. Les définitions formelles peuvent être intéressantes en elles-mêmes lorsque le langage est complet ; pour l'instant elles ne présentent probablement d'intérêt que pour nous.

La section est organisée en un grand nombre de paragraphes. Les premiers traitent des concepts de base des objets de données, des descriptions, et des relations entre les objets. Nous exposons ensuite les fonctions primitives de sémantique et présentons des définitions informelles et des exemples dans les paragraphes 4.7 et 4.8. Le paragraphe 4.9 est un bref exposé du cycle de compilation, interprétation et exécution. Le paragraphe 4.10 donne un exemple très élaboré de la façon dont les fonctions primitives peuvent être combinées pour faire quelque chose d'intéressant : une restitution sélective par contenu. Les deux derniers paragraphes reviennent à l'exposé des fonctions de haut niveau et à des conclusions.

4.1 Objets

Un "objet" a un nom, une description, et une valeur. Il peut être mis en relation avec d'autres objets.

Le "nom" est un symbole, qui peut être utilisé pour accéder à l'objet à partir des fonctions du langage.

La "description" est une spécification des propriétés de l'objet, dont beaucoup se rapportent à la signification ou à la représentation de la valeur.

La "valeur" est l'information sur l'intérêt ultime de l'objet.

Les relations entre les objets sont hiérarchiques. Chaque objet peut être mis en relation directe avec au plus quatre autres objets, désignés comme son "parent", son "fils", son "frère de gauche", et son "frère de droite".

Ce concept spécifique de relations est tout ce qui a été incorporé au modèle à ce jour. Un de nos principaux objectifs à l'avenir est de faire des expériences avec des relations plus générales entre les objets.

4.2 Descriptions

Une description a les composants "nom", "type" et "paramètre dépendant du type". Elle peut être mise en relation hiérarchique avec d'autres descriptions, selon un schéma similaire à celui décrit pour les objets en 4.1.

Le "nom" a un rôle dans le référencement, comme dans le cas des objets.

"Type" est une idée intuitive indéfinie pour laquelle nous pensons développer une signification précise au sein du langage de données (voir au paragraphe 3.10 quelques unes des idées sur ce sujet). Dans les termes du présent modèle, cela signifie simplement un des éléments suivants : LIST *(liste)*, STRUCT *(structure)*, STRING *(chaîne)*, BOOL *(booléen)*, DESC *(description)*, DIR *(répertoire)*, FUNC *(fonction)*, OPD *(descripteur d'opération)*. Chacun d'eux se réfère à une sorte de valeur correspondant aux idées courantes en programmation (à l'exception de OPD, qui est expliqué au paragraphe 4.7) et sur lesquelles certaines opérations sont définies.

Des exemples de "paramètres dépendants du type" sont les deux éléments nécessaires pour définir une CHAÎNE : taille d'option et taille. Une CHAÎNE est une séquence de caractères ; la taille de la CHAÎNE est le nombre de caractères qu'elle contient. Si une CHAÎNE a une taille fixe, la taille d'option est FIXE et taille est le nombre de caractères qu'elle contient toujours. Si une CHAÎNE a une taille variable, la taille d'option est VARIABLE, et taille est son maximum (en fait elle pourrait aussi avoir un minimum dans un schéma plus raffiné).

Lorsque la description d'un objet a un type de CHAÎNE, il est couramment dit que l'objet est une CHAÎNE.

4.3 Valeurs

La valeur est la donnée elle-même.

Un objet de type BOOL ne peut avoir que la valeur VRAI ou la valeur FAUX.

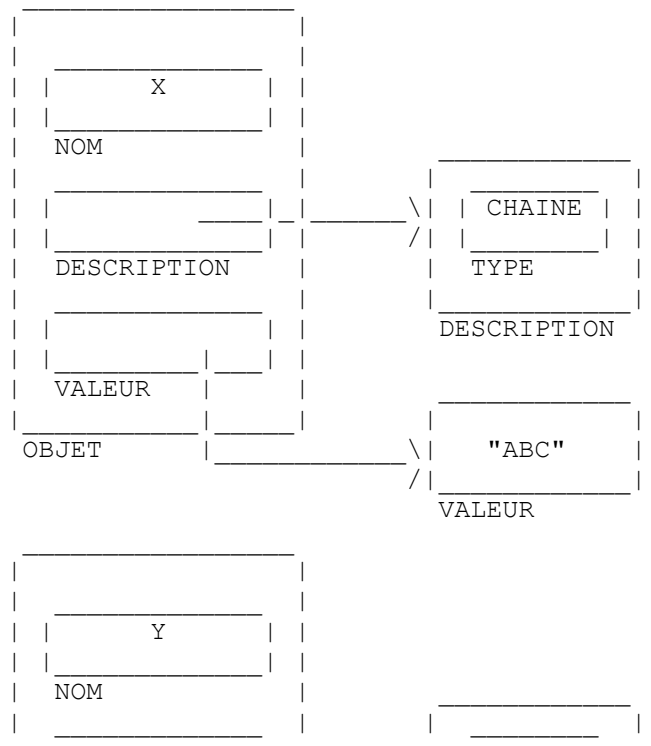
Un objet de type CHAÎNE a des valeurs telles que "ABC", "JEAN", ou 'ROUEN'.

Chaque valeur a une représentation, en bits. Donc un BOOL est représenté par un seul bit, qui sera un "un" pour représenter VRAI et un "zéro" pour représenter FAUX.

4.4 Quelques exemples

Voici quelques exemples de structures qui impliquent des objets, des descriptions, et des valeurs. Dans ces explications et croquis, l'objectif est de faire passer quelques idées sur ces structures de primitives ; des détails considérables sont omis dans les croquis dans un souci de clarté.

La Figure 4-1 montre deux objets. X est du type chaîne et a une valeur 'ABC'. Y est de type booléen et a une valeur de VRAI.



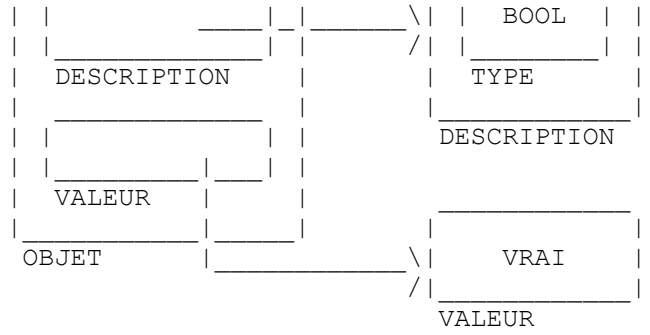


Figure 4-1 : deux objets élémentaires

La Figure 4-2 illustre un objet de type dir (un "répertoire") et les objets qui s'y rapportent. Le répertoire a pour nom SMITH. Deux objets ont été entrés dans ce répertoire, nommés X et Y.

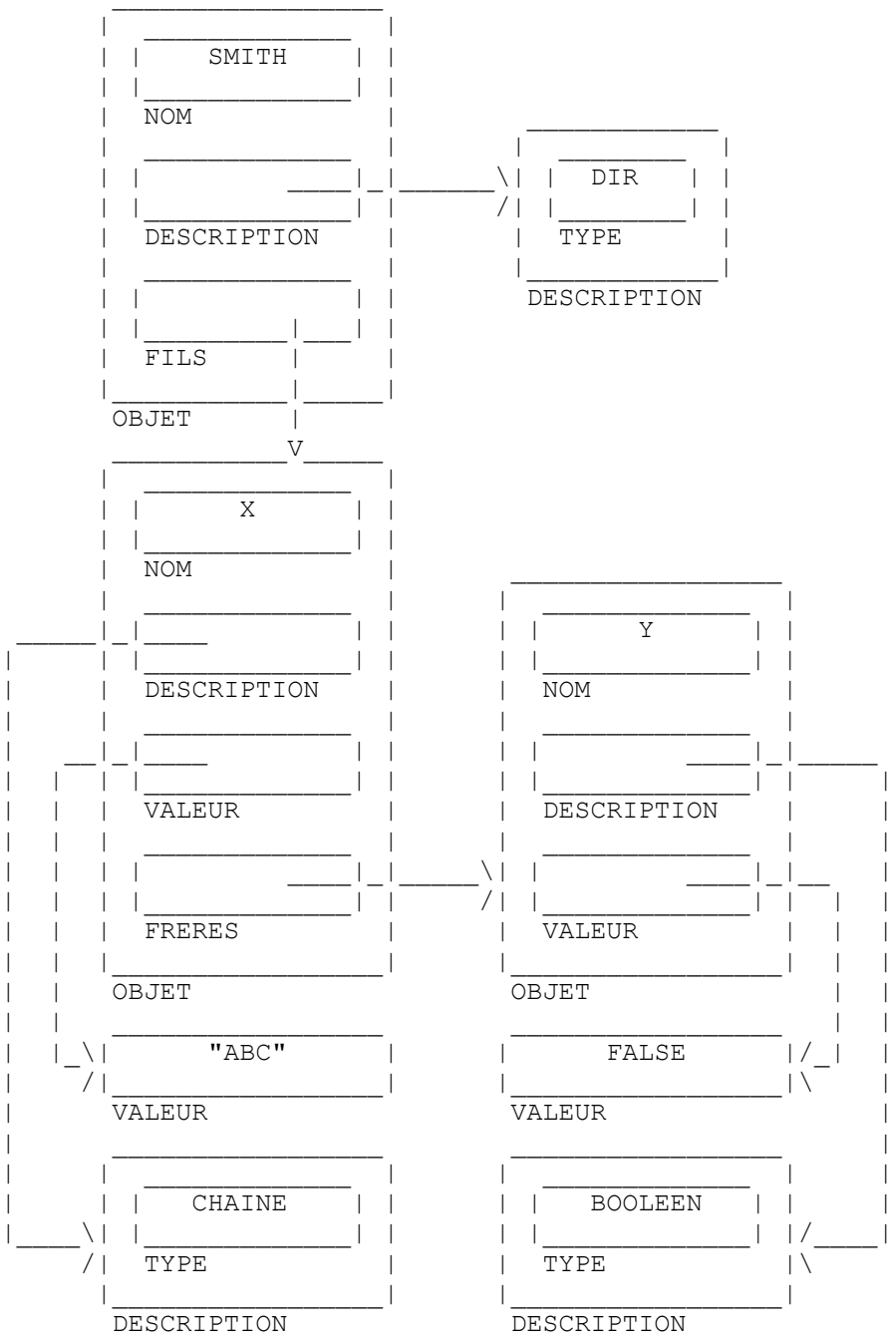


Figure 4-2 : un répertoire avec deux membres

L'idée d'un type "dir" est similaire à celle d'un répertoire de fichiers dans la plupart des systèmes. Un répertoire est un endroit où on peut mémoriser des objets désignés, les ajouter et les supprimer librement. Les entrées dans le répertoire sont tous les objets dont le parent est ce répertoire. La Figure 4-3 montre un groupe d'objets d'une structure plus rigide. Nous avons ici R, une structure, et A et B, une paire de chaînes. Noter que les boîtes étiquetées "objet" dans la Figure 4-3 portent précisément les mêmes relations les unes avec les autres que celles étiquetées "objet" dans la Figure 4-2. Cependant, il y a deux conditions qui tiennent pour 4-3 mais pas pour 4-2 : (1) la valeur de R contient les valeurs de A et B, et (2) les descriptions de R, A et B sont toutes en relation.

Les structures ont les propriétés suivantes : (1) le nom et la description de chaque composant dans la structure sont établis lors de la création de la structure, et (2) dans une valeur de la structure, l'ordre d'occurrence des valeurs des composants est fixé.

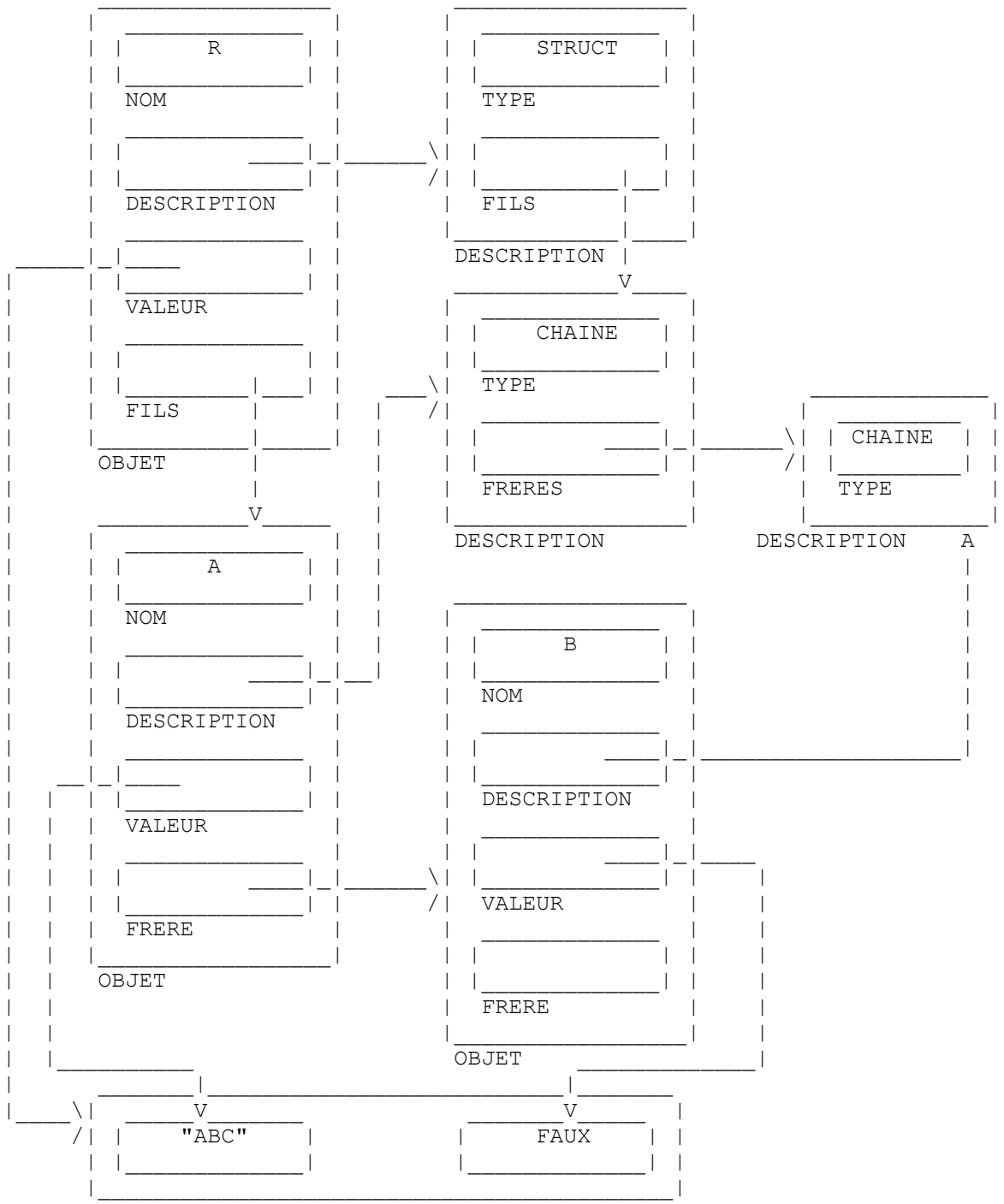


Figure 4-3 : une STRUCTURE avec deux membres

La Figure 4-4 montre une liste nommée L. Une structure d'objets similaire est impliquée ici, mais à cause de la régularité de la structure, toutes les boîtes étiquetées "objet" ne sont pas réellement présentes.

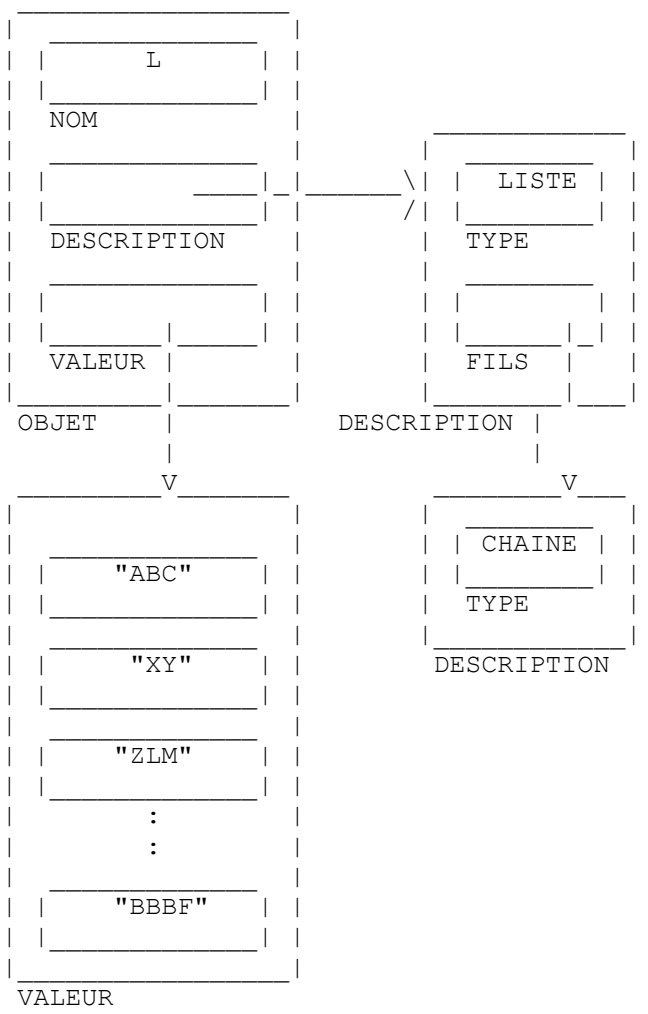


Figure 4-4 : une LISTE

L a un nombre de composants variable qui satisfont tous à la description subordonnée à la description de L.

On pourrait imaginer une boîte "objet" pour chaque chaîne dans L. Chacune de ces boîtes pointerait sur sa chaîne respective et sur la description commune de ces chaînes. Au lieu de cela, nous créons de telles boîtes au fur et à mesure des besoins.

4.5 Définitions de types

Ci-après figurent des définitions plus précises de types, dans les termes du présent modèle. Ils servent à établir plus fermement la sémantique de notre structure d'objets, des descriptions et des valeurs ; cependant, on ne devrait pas les voir comme fournissant une définition pour la spécification complète du langage.

Un objet de type CHAÎNE a une valeur qui est une séquence de caractères (figure 4-1).

Un objet de type BOOL a une valeur qui est une valeur de vérité (VRAI ou FAUX - figure 4-1).

Un objet de type DIR a des objets subordonnés, ayant chacun sa propre description et valeur. Les objets subordonnés peuvent être ajoutés et supprimés à volonté (figure 4-2).

Un objet de type STRUCTURE a des objets subordonnés, dont chacun a une description qui est subordonnée à la description de la STRUCTURE, et une valeur contenue dans la valeur de la STRUCTURE. Le nombre, l'ordre et la description des composants est fixée lors de la création de la STRUCTURE (figure 4-3).

Un objet de type LISTE peut être vu comme ayant des objets subordonnés imaginaires, dont l'existence est simulée par l'utilisation des techniques appropriées pour le traitement de LISTE. Chacun d'eux a la même description, qui est subordonnée à la description de la LISTE. Chacun a une valeur distincte, contenue dans la valeur de la LISTE. En fait, seuls existent l'objet LISTE, la LISTE et les descriptions de composants et les valeurs (figure 4-4).

Un objet de type DESC a une description comme valeur. Cette valeur est la même sorte d'entité qui sert pour la description des autres objets.

Un objet de type FONC une invocation de fonction comme valeur. On sera à même d'en dire plus sur elles après avoir exposé les fonctions.

Un objet de type OPD a un descripteur d'opération comme valeur (voir le paragraphe 4.7 pour des précisions).

4.6 Environnement d'objet

Il y a trois catégories d'objets dans le modèle d'ordinateur de données. Ce sont p/objets, t/objets, et i/objets.

Les p/objets sont des objets permanents créés explicitement avec des fonctions de langage. Ils correspondent à l'idée de données mémorisées dans l'ordinateur de données réel. Il a trois objets spéciaux. Ils ne sont spéciaux qu'en ce qu'ils sont créés au titre de l'initialisation de l'environnement, plutôt que comme résultat de l'exécution d'une fonction du langage. Ils sont nommés STAR, BLOCK et TOP/LEVEL. Tous trois sont du type DIR.

Un objet est un p/objet si il est subordonné à STAR ; c'est un t/objet si il est subordonné à BLOC. TOP/LEVEL est subordonné à BLOC. (Voir les figures 4-5 et 4-6).

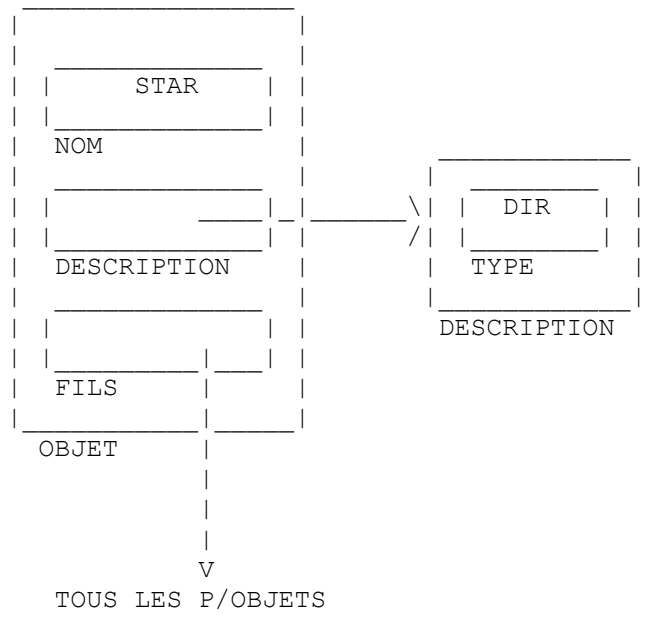
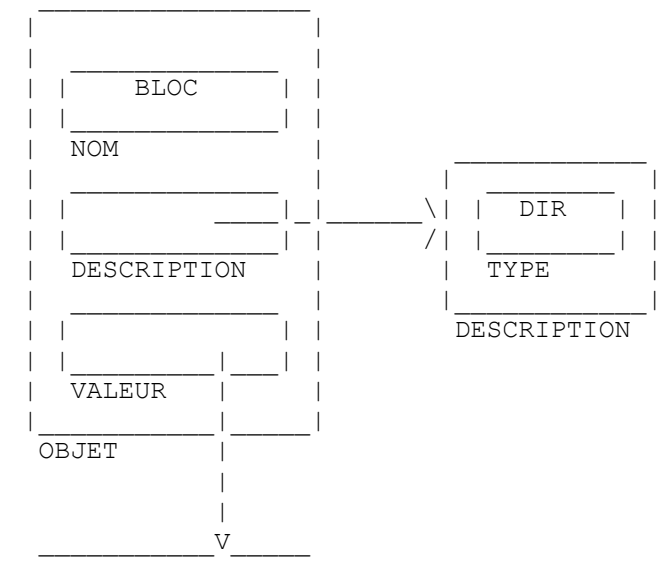


Figure 4-5 : STAR et p/objets

Les t/objets sont des objets temporaires, aussi créés explicitement avec des fonctions du langage. Cependant, ils correspondent à des éléments temporaires définis par l'utilisateur, à la fois local pour les demandes et de "niveau supérieur" (c'est-à-dire, non local pour aucune demande, mais existant jusqu'à suppression ou déconnexion.)



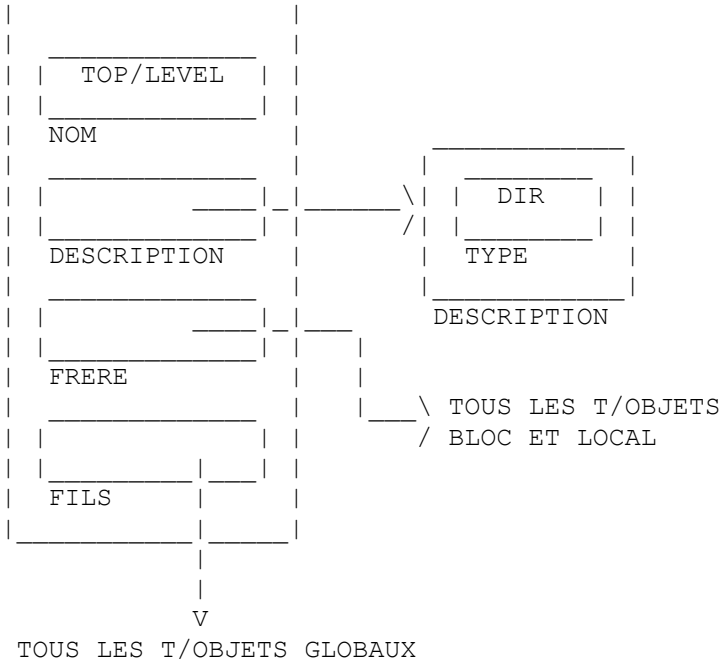


Figure 4-6 : BLOC, TOP/LEVEL et t/objets

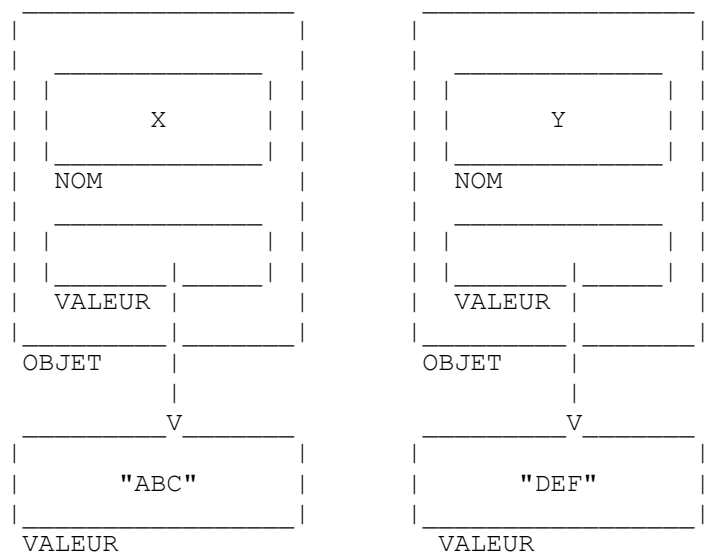
Les I/objets sont des objets internes, définis par le système, dont la création et la suppression est implicite dans l'exécution de certaines fonctions de langage.

Les I/objets découlent directement des invocations de fonctions (objets de type FUNC), et sont toujours locaux à l'exécution de telles invocations de fonction. Ils correspondent aux notions de (1) littéral, et (2) compilateur - ou interpréteur – généré de façon temporaire.

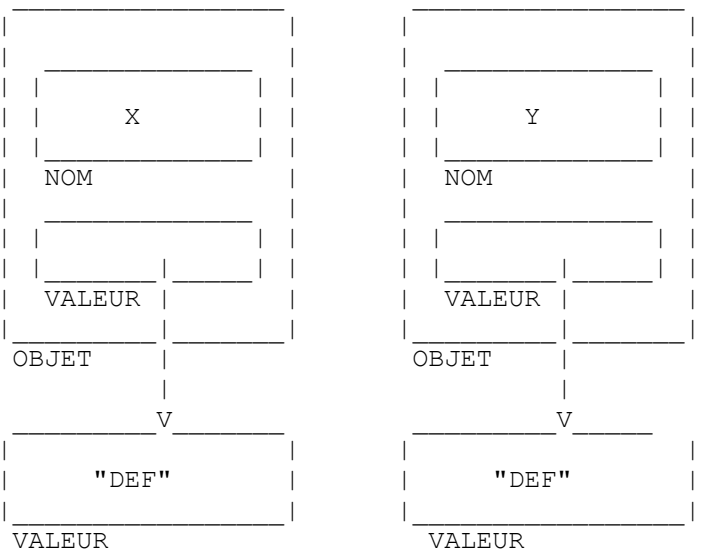
4.7 Fonctions primitives du langage

Nous exposons ici les fonctions primitives du langage actuellement mises en œuvre dans le modèle et qui seront vraisemblablement du plus grand intérêt. On met l'accent, dans ce paragraphe, sur la mise en relation des fonctions les unes avec les autres. Le paragraphe 4.8 contient de plus amples détails et exemples.

"Assign" fonctionne sur une paire d'objets, appelés la cible et la source. La valeur de la source est copiée dans la valeur de la cible. La Figure 4-7 montre une paire d'objets, X et Y, avant et après l'exécution d'une tâche ayant X pour cible et Y pour source. Présentement, la tâche n'est définie que pour les objets de type BOOL et les objets de type CHAINE. Les objets impliqués doivent avoir des descriptions identiques.



Avant la tâche



Après la tâche

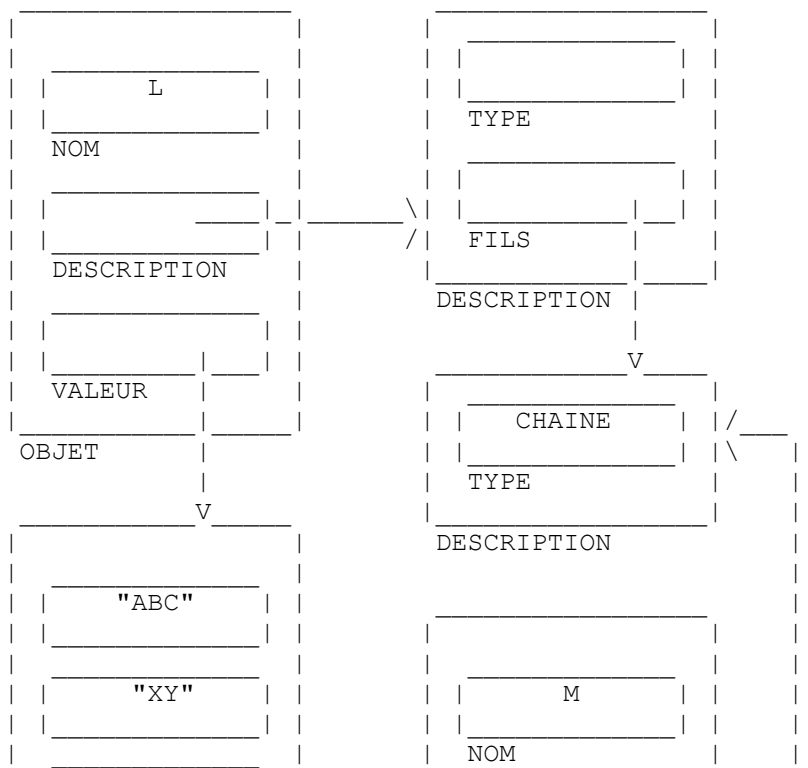
Figure 4-7 : Effet d'allocation

On définit une classe de fonctions primitives pour manipuler les LISTES. On les appelle "listop". Toutes les listop entrent un objet particulier appelé un "descripteur_d'opération" ou OPD.

Pour accomplir une opération complète sur une LISTE, une séquence de listop doit être exécutée. Il a des restrictions sémantiques sur la composition d'une telle séquences, et il vaut mieux considérer la séquence entière comme une grosse opération. L'état d'une telle opération est conservé dans la OPD.

Revenons à la figure 4-4. Il y a une boîte marquée "objet" ; cette boîte représente la liste comme un tout. Pour travailler sur tout membre, on a besoin d'une boîte objet pour le représenter. La Figure 4-8 montre la structure avec une boîte d'objet additionnel ; la nouvelle boîte représente un membre à un moment donné. Sa valeur est un des composants de la valeur de la LISTE ; sa description est subordonnée à la description de la LISTE. En 4-8, le nom de cet objet est M.

En 4-8 nous avons assez de structure pour fournir une description et une valeur pour M, et cela est suffisant pour permettre l'exécution des opérations sur M comme élément. Cependant, il n'y a pas de lien direct entre l'objet M et l'objet L. La structure est complétée par l'addition d'une OPD, montrée à la figure 4-9.



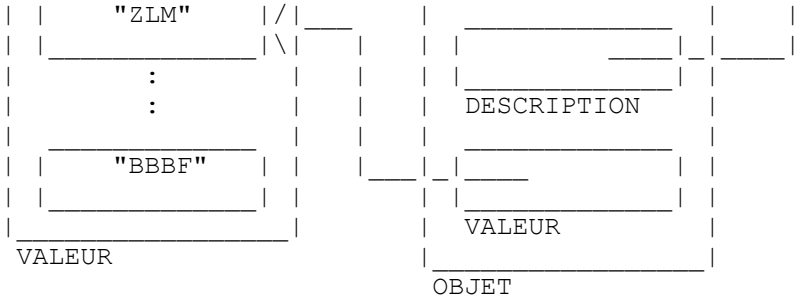


Figure 4-8 : LISTE et objets membres

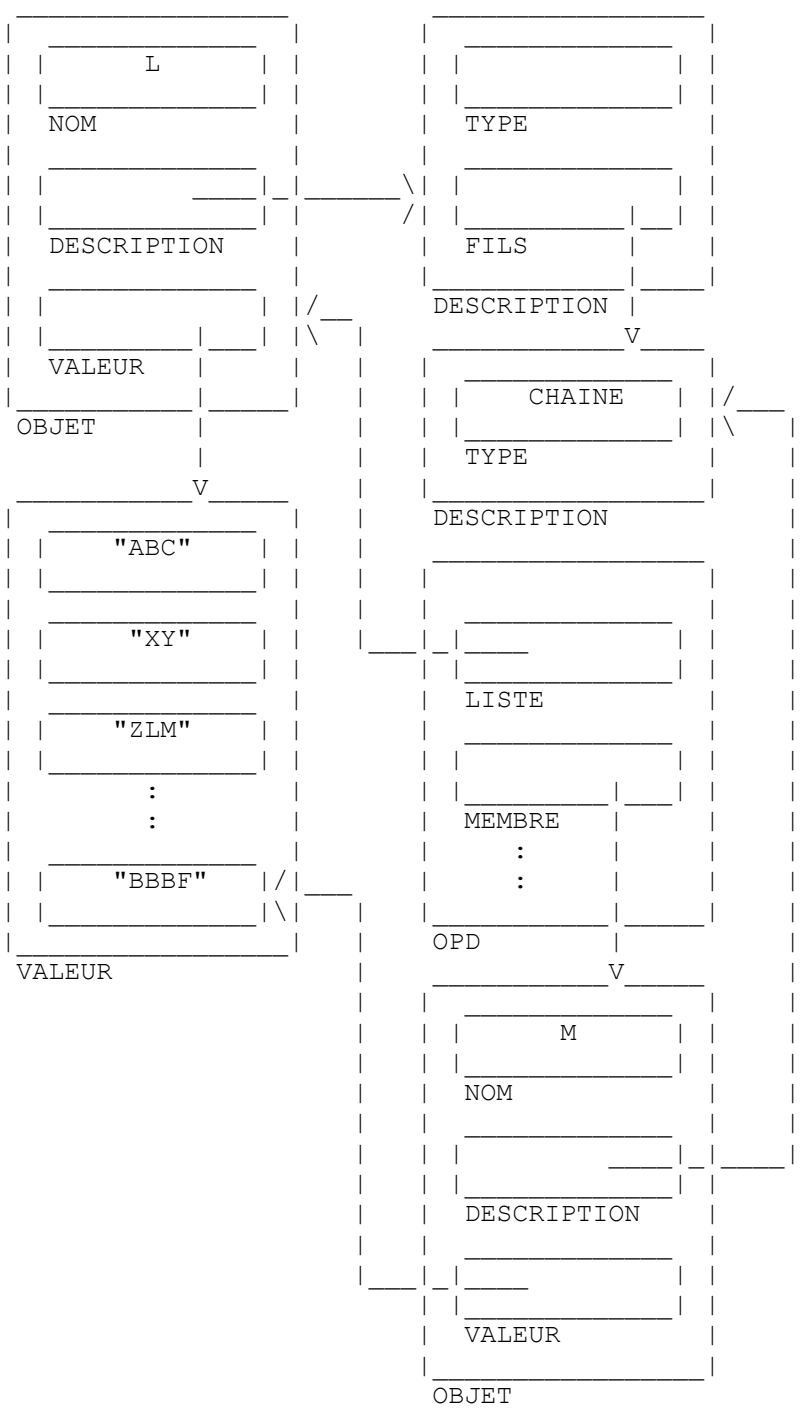


Figure 4-9 : OPD, LISTE et objets membres

L'OPD établit la relation de l'objet, et contient des informations sur la séquence des opérations de liste (listop) primitives en cours. Lorsque des informations suffisantes sont conservées dans l'OPD, on a en 4-9 une structure qui est adéquate pour la

maintenance de l'intégrité de la LISTE et du fonctionnement global de la liste. En plus des pointeurs LISTE et membre, l'OPD contient des informations qui indiquent : (1) quelles sous opérations sont activées pour la séquence, (2) la sous opération en cours, (3) le nombre d'instances de membres de la LISTE actuelle, (4) un indicateur de fin de liste. Les sous opérations sont add/member *(ajouter un membre)*, delete/member *(supprimer un membre)*, change/member *(changer un membre)* et get/member *(obtenir un membre)*. Toutes s'appliquent aux membres actuels. Seules les sous opérations qui ont été activées au début d'une séquence peuvent être exécutées durant cette séquence ; finalement, la connaissance à l'avance des intentions que cela implique va fournir des informations importantes pour le contrôle et l'optimisation de la synchronisation.

À présent, un OPD met en rapport un seul objet membre à un seul objet de LISTE. Cela impose une importante restriction sur les séquences de classe d'opérations qui peuvent être exprimées. Toute transformation de LISTE qui exige un accès simultané à plus d'un membre doit être représentée comme plus d'une séquence. (et nous ne savons pas encore résoudre les problèmes impliqués par l'exécution simultanée de telles séquences, même lorsque les deux sont contrôlées par un seul processus.)

Toute transformation d'une LISTE peut encore être réalisée par la mémorisation des résultats intermédiaires dans des objets temporaires ; cependant, il est certainement plus souhaitable d'incorporer l'idée de membres actuels multiples dans la sémantique du langage, que d'utiliser de tels moyens temporaires. Une importante extension future de listop traitera de ce problème.

Il y a six listop : listop/begin, listop/end, which/membre, end/of/list, open/membre et close/membre.

Listop/begin et listop/end effectuent les fonctions évidentes de commencer et terminer une séquence de listops. Listop/begin entre des objets LISTE et MEMBRE, un OPD, et une spécification de sous opérations à activer. Elle initialise l'OPD, y compris l'établissement des liaisons avec les objets LISTE et MEMBRE. Après l'établissement de la relation OPD-LISTE-MEMBRE, il est seulement nécessaire de fournir l'OPD et les paramètres auxiliaires comme entrée à une listop dans la séquence. Tout le reste peut être déduit de l'OPD.

Listop/end libère l'OPD ainsi que toutes les ressources acquises par listop/begin.

Which/member établit le membre en cours pour toute sous opération. Ce sera le premier membre de la LISTE, le dernier membre de la LISTE, ou le membre suivant de la LISTE. Cette listop identifie simplement sur quel membre l'opération est à effectuer ; elle ne rend pas le contenu du membre accessible.

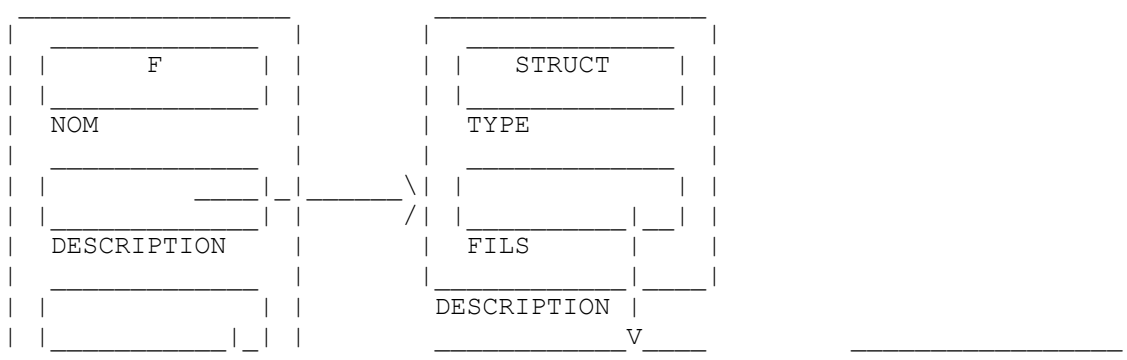
Open/member et close/member viennent en soutien à une sous opération. La sous opération est indiquée comme un argument de open/member. Open/member établit toujours un pointeur de l'objet membre à la valeur du membre ; close/member libère toujours ce pointeur. De plus, chacune de ces listops peut faire une action, selon la sous opération.

Les détails de l'action vont dépendre de la représentation de la LISTE dans la mémoire, de la taille d'un membre de la liste, et des choix faits par la mise en œuvre.

Ente l'exécution de open/member et de close/member, les données sont accessibles. Elles peuvent toujours être lues ; dans le cas des sous opérations add/member et change/member, on peut aussi écrire dedans.

End/of/list vérifie un fanion dans l'OPD et retourne un objet de type booléen. La valeur de l'objet est la même que la valeur du fanion ; elle est VRAIE si un get/membre, change/membre ou delete/membre ne serait pas réussi à cause du déplacement d'un which/member "au delà de la fin". Cette listop est fournie de sorte qu'il soit possible d'écrire des procédures qui terminent conditionnellement quand tous les membres ont été traités.

Get/struct/member donne la capacité de traiter les STRUCT. Un objet STRUCT donné qui pointe sur la valeur STRUCT va établir un pointeur d'un objet MEMBRE donné sur la valeur du membre. (Le pointeur qu'il établit est représenté par une ligne en pointillés à la figure 4-10).



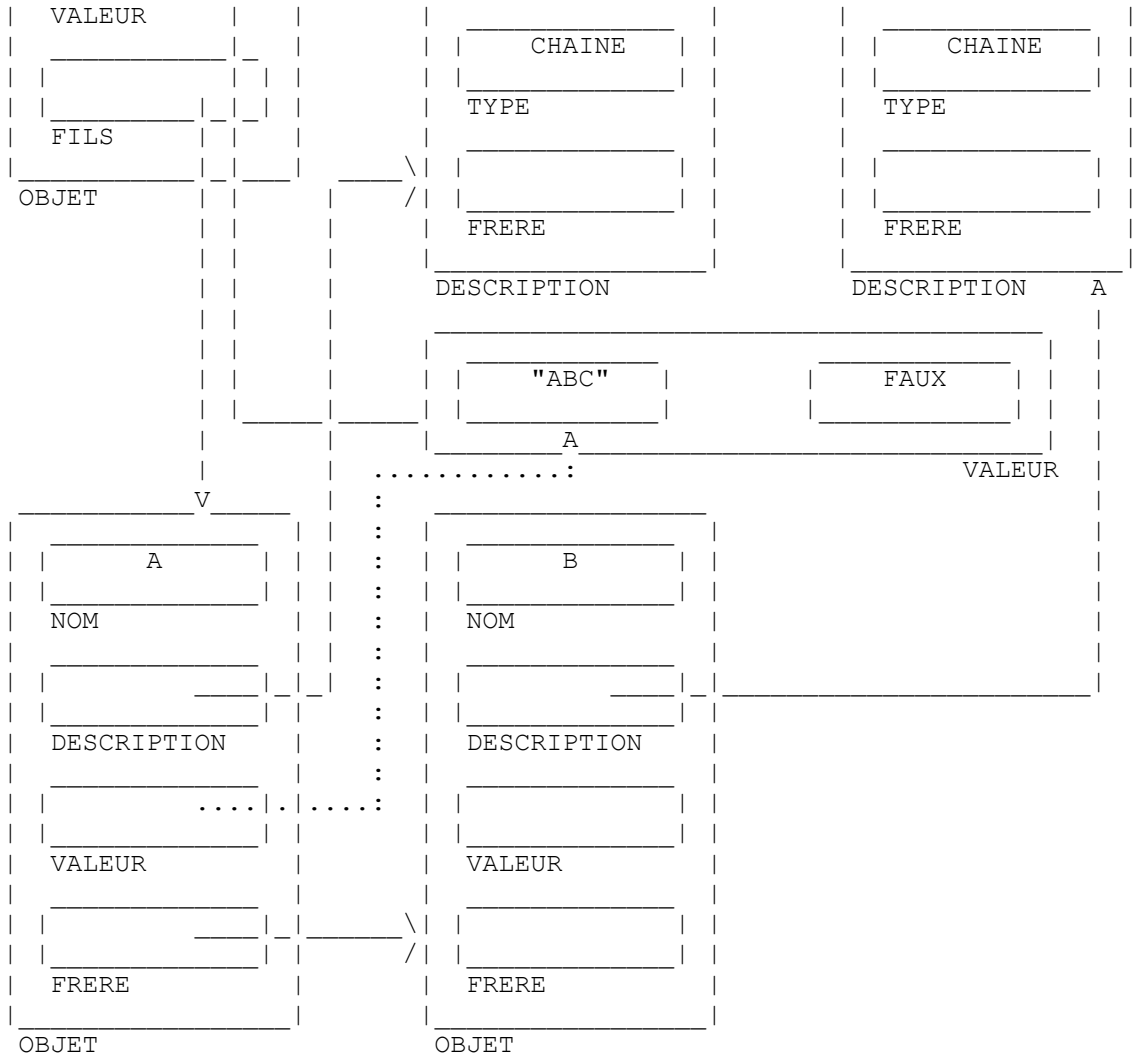


Figure 4-10 : Effet de GET/STRUCT/MEMBRE

Les primitives discutées jusqu'à maintenant (assign, listops, et get/struct/member) fournissent les facilités de base pour travailler sur les structures de LISTE, STRUCT et les composants élémentaires. En utilisant seulement ceux là, il est possible de transférer le contenu d'une structure hiérarchique à une autre, d'ajouter des structures, ou de supprimer des portions de structures, et ainsi de suite. Pour effectuer des opérations plus intéressantes, des facilités de contrôle et de sélection sont nécessaires.

Une facilité de contrôle rudimentaire est fournie par les primitives if/then *(si/alors)*, if/then/else *(si/alors/autrement)*, till *(jusqu'à)* et while *(tandis que)*. Elles évaluent toutes une invocation de fonction primitive, qui doit retourner une valeur booléenne. Une action est faite sur la base de cette valeur booléenne.

Soit A et B des invocations de fonction. If/then(A,B) va exécuter B si A retourne VRAI. If/then/else(A,B,C) va exécuter B si A retourne VRAI ; il va exécuter C si A retourne FAUX. Les opérateurs while et till font des itérations, exécutant d'abord A puis B. While termine la boucle lorsque A retourne FAUX ; till termine la boucle lorsque A retourne VRAI. Si cela arrive la première fois, B n'est jamais exécuté.

Jusqu'à présent, nous avons mentionné une fonction qui retourne une valeur booléenne : listop, end/of/list. Deux autres classes de fonctions qui ont cette propriété sont les booléens et les comparaisons. Il y a trois primitive booléennes (et, ou, non) et six primitives de comparaisons (égal, moins/que, plus/que, non/égal, inférieur/ou/égal, supérieur/ou/égal – seul égal est mis en œuvre au moment de la présente publication).

Les entrées et sorties booléennes BOOL ; les comparaisons entrent des paires d'objets élémentaires qui ont les mêmes BOOL de description et de résultat. Les expressions composées de booléens et de comparaisons sur les contenus d'élément sont un des principaux outils utilisés pour référencer les données de façon sélective dans les systèmes de gestion de données.

Avec les booléens, les comparaisons, et les primitives identifiées plus tôt, nous pouvons effectuer des "restitutions"

sélectives. C'est à dire que nous pouvons transférer à la LISTE B tous les éléments qui dans la LISTE A ont une valeur de 'ABC'. En fait, nous avons (sémantiquement) une capacité générale à effectuer des restitutions fondées sur le contenu ainsi que des mises à jour sur des structures hiérarchiques arbitraires. On peut même programmer quelque chose d'aussi complexe que le traitement d'une liste de transactions par rapport à une liste maître, qui est une des applications typiques du traitement de données d'affaires.

Bien sûr, on ne s'attend pas à ce que les utilisateurs du langage de données expriment des demandes au niveau des listops. De plus, les listops définies ici ne sont pas un moyen très efficace pour effectuer certaines des tâches que nous avons mentionnées. Pour obtenir de bonnes solutions, nous avons à la fois besoin d'opérateurs de niveau supérieur et d'autres primitives qui utilisent d'autres techniques de traitement.

En plus de celles déjà discutées, le modèle contient des fonctions pour :
(1) référencer un objet par son nom qualifié,
(2) générer une constante,
(3) générer des descriptions de données,
(4) écrire des fonctions et blocs composés avec des variables locales,
(5) créer des objets.

Les facilités pour générer des constantes et des descriptions de données (qui sont un cas particulier de constantes) sont marginales, et n'ont pas de caractéristiques particulièrement intéressantes. Evidemment, les description de données seront un problème important dans nos efforts ultérieurs de modélisation.

Les objets qui référencent les fonctions permettent de faire référence aux t/objects et aux p/objects (ces termes sont définis en 4.6). Un p/object est référencé en donnant le nom de chemin de STAR jusqu'à lui. Un t/object est référencé en donnant le nom du chemin depuis le répertoire de bloc dans lequel il est défini comme étant.

Les fonctions composées permettent qu'une invocation de séquence de fonctions soit traitée syntaxiquement comme une seule invocation. Donc, par exemple, dans if/then(A,B), B est fréquemment l'invocation d'une fonction composée, qui à son tour invoque une séquence d'autres fonctions.

Create prend deux entrées : un objet supérieur et une description. L'objet supérieur doit être un répertoire. Le nouvel objet est créé comme fils le plus à gauche du répertoire ; son nom est déterminé par la description.

4.8 Détails des fonctions primitives du langage

Ce paragraphe donne des spécifications pour les primitives exposées au paragraphe précédent. On omet ici aussi les détails qui ne présentent pas un intérêt général ; l'objectif étant de fournir assez d'informations pour que le lecteur puisse comprendre les exemples.

La plupart des primitives surviennent à deux niveaux dans le modèle. Les primitives internes sont appelées i/functions et les primitives externes, ou de langage, sont appelées l/functions. Les relations entre les deux types sont expliquées au paragraphe 4.9. Celui-ci traite des i/functions.

Les L/functions ont en entrée et en sortie des "_forms_",qui sont des structures arborescentes sont les nœuds d'extrémité sont des atomes. Les atomes sont des choses comme les noms de fonction, les noms d'objet, les constantes de chaîne littérale, des valeurs et délimiteurs de vérité. Les invocations des i/functions sont aussi exprimées comme des formes.

Toute forme peut être évaluée, donnant un objet. Une forme qui est une invocation de i/function donne la valeur retournée par la i/function, c'est-à-dire une autre forme. En général, la forme retournée par une invocation de i/function va, lorsque elle est évaluée, donner un objet de langage de données (c'est-à-dire, la sorte d'objet qui a été représenté par une "boîte d'objet" dans les dessins).

4.8.1 Fonctions de reconnaissance de nom

Elles retournent une forme qui s'évalue comme objet.

L/TOBJ
L'entrée doit désigner un objet temporaire subordonné à un répertoire de niveau supérieur ou de bloc.

L/POBJ
L'entrée doit désigner un objet permanent (c'est-à-dire un objet subordonné à STAR).

Les invocations normales sont L/POBJ(X.Y.Z) et L/TOBJ(A).

4.8.2 Générateurs de constantes

Chacune de ces fonction entre un symbole atomique qui donne une valeur d'une constante à créer. Chacune retourne une forme qui va s'évaluer comme un objet qui a la valeur spécifiée et une description appropriée.

LC/STRING – une invocation normale est LC/STRING('ABC')

LC/BOOL – une invocation normale est LC/BOOL(TRUE)

4.8.3 Fonctions d'éléments élémentaires

Ces formes d'entré et sortie s'évaluent en objets élémentaires (objets qui peuvent n'avoir pas d'objet subordonné -- en fait, des objets dont la valeur est considérée comme atomique). Finalement, tous les opérateurs de comparaison seront mis en œuvre.

L/ASSIGN
Les entrées doivent s'évaluer comme des chaînes ou comme des booléens. Les résultats sont une forme qui transfère la valeur du second en celle du premier. L'invocation typique : L/ASSIGN(L/TOBJ(A),LC/STRING('XYZ')) va formes en sortie après évaluation une copie de "XYZ" dans la valeur de A.

L/EQUAL
Entre une paire de formes qui s'évaluent en objets, qui doivent avoir des descriptions identiques et être des booléens ou des chaînes. Il retourne une forme qui s'évalue en un objet de type booléen. La valeur de cet objet est VRAI si les entrées ont des descriptions et valeurs identiques ; et FAUX autrement. L'invocation typique est :

```
L/EQUAL(L/TOBJ(X),LC/STRING('DEF'))
```

L/AND, L/OR, L/NOT
Ce sont les opérateurs booléens standard. Les entrées sont des formes qui s'évaluent en booléens ; les sorties sont des formes qui s'évaluent en booléens. L/AND et L/OR prennent deux entrées ; L/NOT une seule. L'invocation typique est :

```
L/AND( L/EQUAL(L/TOBJ(X),LC/STRING('DEF')), L/EQUAL(T/TOBJ(Y),LC/STRING('GHI')) )
```

La forme retournée va, lorsque elle est évaluée, retourner VRAI si les deux X ont la valeur 'DEF' et Y a la valeur 'GHI'.

4.8.4 Fonctions de description des données

Elles retournent toutes une forme qui s'évalue en une description (c'est-à-dire une qui est représentée dans nos dessins par une boîte marquée "description").

LD/STRING
Elle prend en entrée trois paramètres qui spécifient le nom, l'option de taille et la taille de la chaîne. L'invocation typique est :

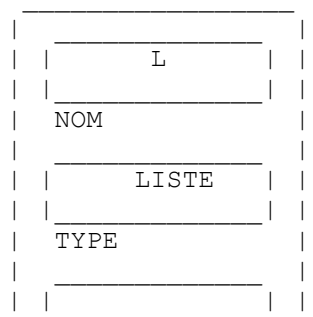
```
LD/STRING(X,FIXED,3)
```

Cette invocation retourne une forme qui s'évalue en la description d'une chaîne de longueur fixe de 3 caractères nommée X.

LD/LIST
Entre deux formes. La première est le nom de la liste et la seconde s'évalue comme une description du membre de la liste. L'invocation typique est :

```
LD/LIST(L,LD/STRING(M,FIXED,3))
```

Elle crée la structure montrée à la figure 4-11, et retourne une forme qui s'évalue comme la description représentée par la boîte supérieure.



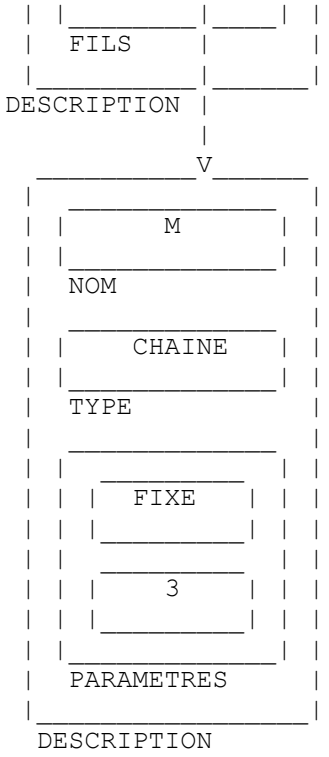
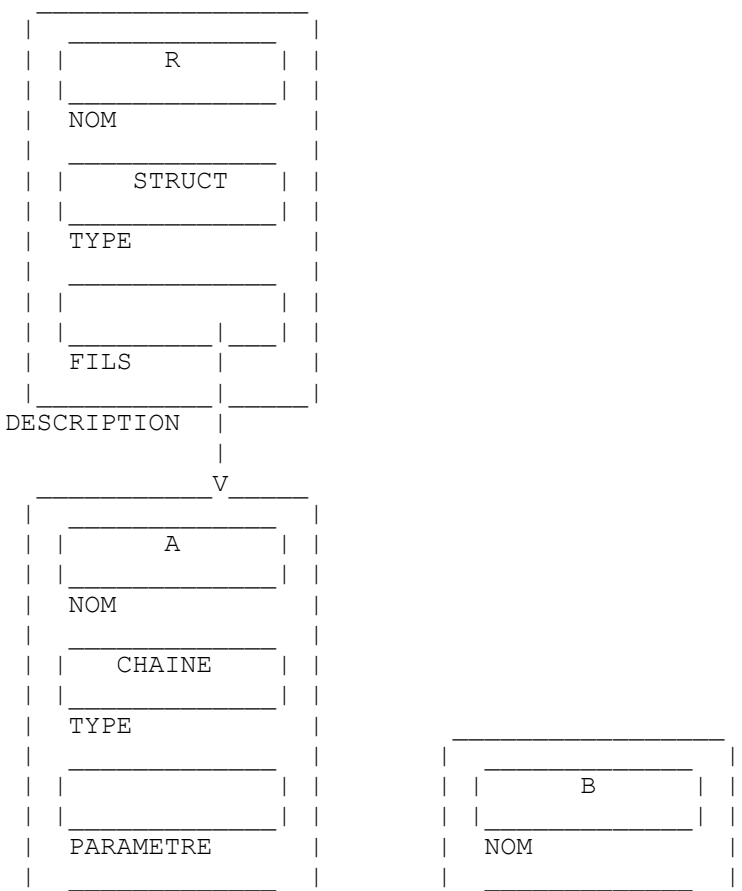


Figure 4-11 : Descriptions de LISTE et des membres

LD/STRUCT
Entre une forme à utiliser comme le nom pour la structure et une ou plusieurs formes qui s'évaluent en descriptions ; celles-ci sont prises comme descriptions des membres. L'invocation typique :

LD/STRUCT(R, LD/STRING(A,FIXED,3) LD/BOOL(B))

produit la structure montrée en 4-12 ; elle retourne une forme qui s'évalue en la boîte supérieure.



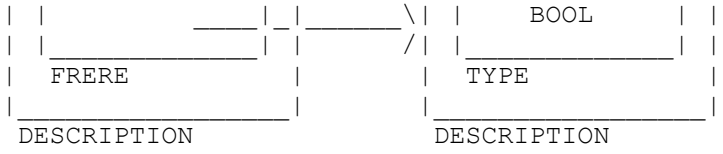


Figure 4-12 : Descriptions de STRUCT et des membres

LD/BOOL, LB/DIR, LD/OPD, LD/FUNC, LD/DESC
Chacune entre un nom et produit une seule description ; chacune retourne une forme qui s'évalue en la description produite. L'invocation typique est : LD/BOOL(X).

4.8.5 Création de données

L/CREATE
Entre deux formes et les évalue. La première doit donner un objet de type DIR ; la seconde doit donner une description pour l'objet à créer. Elle crée l'objet et retourne une forme, qui, lorsque il est évalué, va générer une valeur pour le nouvel objet. Un simple exemple : L/CREATE(L/TOBJ(X),LD/B0OL(Y)).

La Figure 4-13 montre le répertoire X avant l'exécution de l'invocation ci-dessus. Il ne contient que un OPD. Après exécution, le répertoire apparaît comme dans 4-14. La création d'une valeur pour Y survient lorsque la forme retournée par L/CREATE est évaluée (traité au paragraphe 4.9).

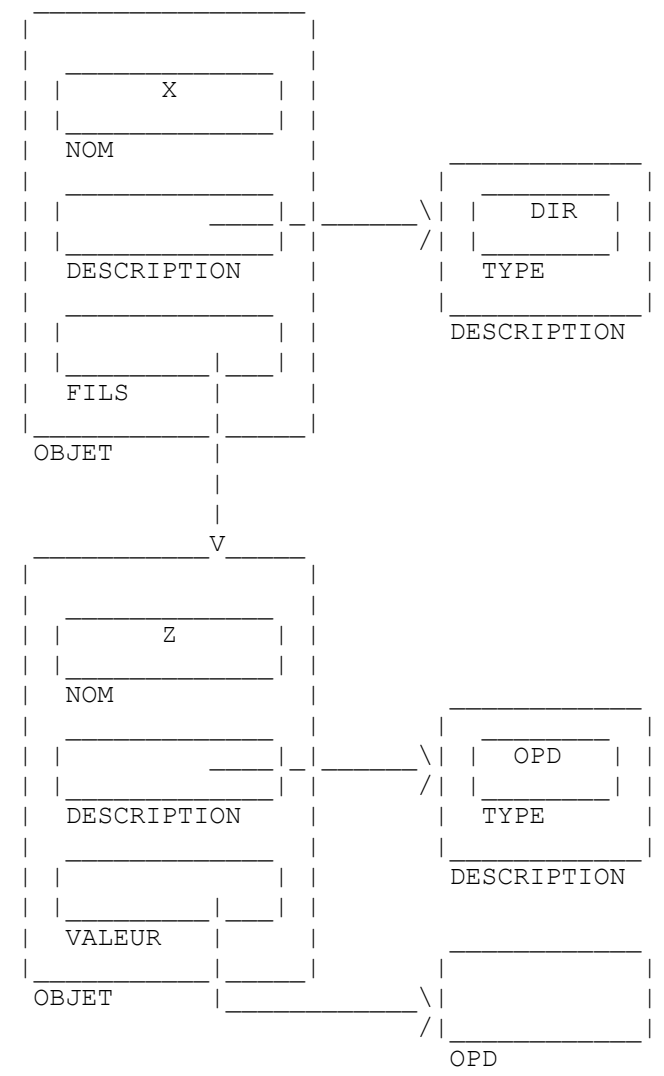


Figure 4-13 : X et Z avant la création de Y

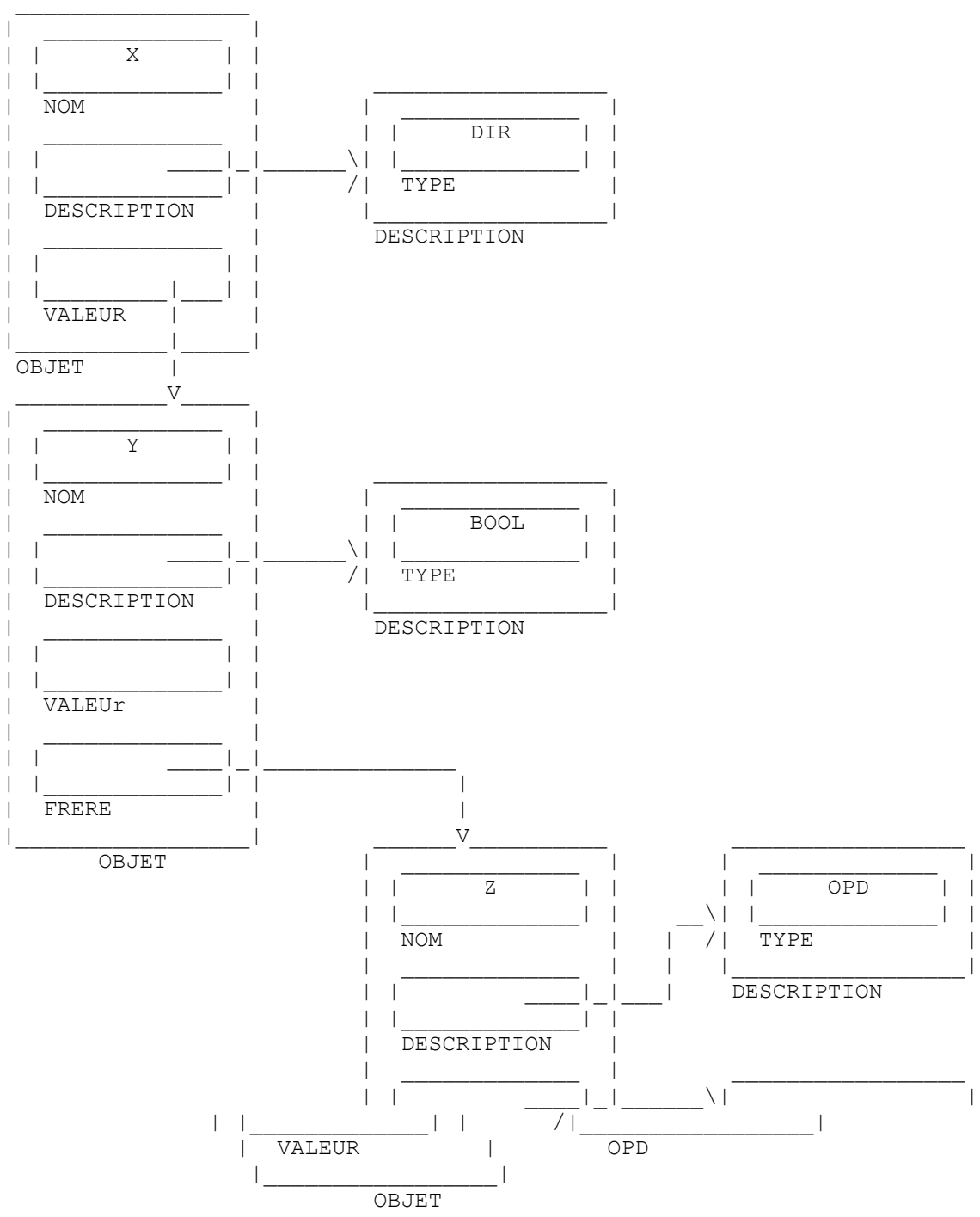


Figure 4-14 : X, Y, et Z après L/CREATE

4.8.6 Contrôle

L/IF/THEN, L/IF/THEN/ELSE
Utilisé pour demander une évaluation conditionnelle d'une forme. l'invocation typique est :

L/IF/THEN(L/EQUAL(L/TOBJ(A),LC/STRING('ABC'),L/ASSIGN(L/TOBJ(B),LC/STRING('DE')))

La forme retournée va faire ce qui suit lorsque elle est évaluée : si A a la valeur "ABC", alors mémoriser "DE" dans la valeur de B.

L/WHILE, L/TILL
Ces primitives font une itération conditionnelle, comme expliqué à la section précédente. Des exemples figurent plus loin.

L/CF
Fonction composée *(Compound function)* : Elle entre une ou plusieurs formes et retourne une a forme qui, lorsque elle est évaluée, va évaluer chaque entrée à la suite. L'invocation typique est :

L/CF(L/ASSIGN(L/TOBJ(R.A),LC/STRING('XX')),L/ASSIGN(L/TOBJ(R.B),LC/STRING('YY')))

Lorsque le résultat de L/CF est évalué, il va allouer de nouvelles valeurs à R.A et R.B.

4.8.7 Listops

Ces primitives sont exécutées à la suite afin d'effectuer des opérations sur les listes. À l'exception de L/END/OF/LIST, ces fonctions sortent des formes qui sont évaluées pour leur seul effet ; c'est à dire que les formes en sortie ne retournent pas de valeurs par elles-mêmes.

L/LISTOP/BEGIN
Les formes en entrées s'évaluent comme : (1) une LISTE, (2) un objet qui représente le membre actuel de la LISTE, (3) un OPD. Aussi, une liste de formes atomiques dont les valeurs sont prises comme sous opérations à activer. L'invocation normale est :

L/LISTOP/BEGIN(L/POBJ(F),L/TOBJ(R),L/TOBJ(OPF),ADD,DELETE)

Cela retourne une forme qui va initialiser une séquence de listops à effectuer sur F. L'invocateur a précédemment créé R et OPF. Il a l'intention d'ajouter et supprimer des membres de la liste.

Toutes les invocations suivantes dans cette séquence de listops ont seulement besoin de spécifier le OPD et les paramètres auxiliaires.

L/LISTOP/END
Entre une forme qui s'évalue en un OPD. Donne en résultat une forme qui, quand elle est évaluée; supprime l'OPD et casse les relations entre OPD, LISTE et les objets membres.

L/WHICH/MEMBER
Entre deux formes. La première s'évalue comme un OPD ; la seconde est PREMIERE, DERNIERE, ou SUIVANTE. La forme en sortie, lorsque elle est évalue, va établir un nouveau membre actuel pour la prochaine sous opération.

Note : Cela ne rend pas accessible la valeur du membre, cela l'identifie simplement en réglant le numéro d'instance dans l'OPD. L'invocation normale est :

L/WHICH/MEMBER(L/TOBJ(OPF),NEXT)

Lorsque un which/member cause l'avance au delà de la fin de la liste, un fanion est mis dans l'OPD.

L/END/OF/LIST
Entre une forme qui s'évalue en OPD. Sort une forme qui, à l'évaluation, retourne un booléen. Il a la valeur VRAI si le fanion de fin de liste de l'OPD est mis.

L/OPEN/MEMBER
Entre une forme qui s'évalue en un OPD et une forme qui doit être ADD, DELETE, GET, ou CHANGE. Sort une forme qui, à l'évaluation, va initier la sous opération demandée sur le membre actuel de la liste. La sous opération établit toujours le pointeur de l'objet membre sur l'instance de valeur du membre actuel. De plus, dans le cas de ADD, cette valeur doit être créée. L'invocation normale est :

L/OPEN/MEMBER (L/TOBJ (OPF) ,ADD)

L/CLOSE/MEMBER
Entre une forme qui s'évalue en OPD. Sort une forme qui, quand elle est évaluée, va terminer la sous opération en cours. L'invocation normale est :

L/CLOSE/MEMBER(L/TOBJ(OPF))

Met toujours un terme au pointeur de l'objet membre sur la valeur de membre. De plus, dans le cas de DELETE, retire la valeur de membre de la LISTE. Dans le cas de ADD, elle entre la valeur du membre dans la LISTE. Elle fait du membre ajouté le membre actuel, de sorte qu'une séquence de ADD exécutée sans faire intervenir which/members va ajouter les nouveaux membres en séquence.

Un exemple élaboré, qui implique des listops et plusieurs autres primitives, est donné au paragraphe 4.10.

4.9 Cycle d'exécution

Le modèle d'ordinateur de données a un cycle d'exécution en deux parties : il compile d'abord les demandes, puis les interprète. Une "demande" est une invocation de l/function ; "compilation" est le résultat agrégé de l'exécution de toutes les

invocations de l/function impliquées dans la demande (ce sont normalement de nombreuses invocations, car il y a habituellement plusieurs niveaux d'invocations incorporés, avec les résultats des invocations internes délivrés comme arguments au prochain niveau d'invocations). Normalement, le processus d'exécution d'une l/function implique une simple expansion de macro, précédée par des liens, des vérifications et (éventuellement) une optimisation.

La forme compilée consiste entièrement en symboles atomiques et en invocations de i/function. Les i/function sont des primitives internes qui entrent et sortent des objets de langage de données (les entités représentées par les boîtes marquées "objet" dans les dessins).

Chacune des l/functions discutées se compile en une seule i/function ; et donc l'aspect d'expansion de macro de la compilation est en fait trivial. Cependant, cela ne sera pas vrai en général ; c'est seulement ces _primitive_ l/functions qui le rendent vrai maintenant.

La décision d'utiliser un cycle de compiler et interpréter appelle quelques explications. La façon de comprendre ceci est de le voir en termes de fonctions qui seront effectuées dans un système strictement interprétatif. Il sera encore nécessaire d'effectuer des vérifications globales sur la validité de la demande avant l'exécution d'aucune de ses parties. Cela parce que l'exécution partielle d'une demande incorrecte peut laisser une base de données dans un état incohérent ; si c'est une grande base de données, ou si elle est complexe, le coût de récupération sera considérable. Donc il est rentable de faire autant de vérifications que possible ; lorsque le système est complètement développé, cela inclura certains éléments de simple prédiction du flux d'exécution ; dans tous les cas, cela implique beaucoup plus qu'une vérification syntaxique.

Comme toutes ces vérifications globales doivent être effectuées avant l'exécution réelle, elles ne font pas en fait partie de l'exécution elle-même, pour n'importe quelle forme. En les effectuant au titre d'un processus séparé de compilation, nous formalisons simplement une modularité qui existe effectivement déjà.

Il restera encore cependant les cas dans lesquels les fonctions de vérification, de liaison et d'optimisation doivent être exécutées durant l'interprétation, si elle le sont. Cela va survenir lorsque les informations nécessaires ne sont pas disponibles tant qu'on n'a pas accédé aux données. Lorsque on le peut, on traitera de telles occurrences en concevant la plupart des fonctions de telle sorte qu'elles puissent être exécutées au titre de l'une ou l'autre "moitié" du cycle.

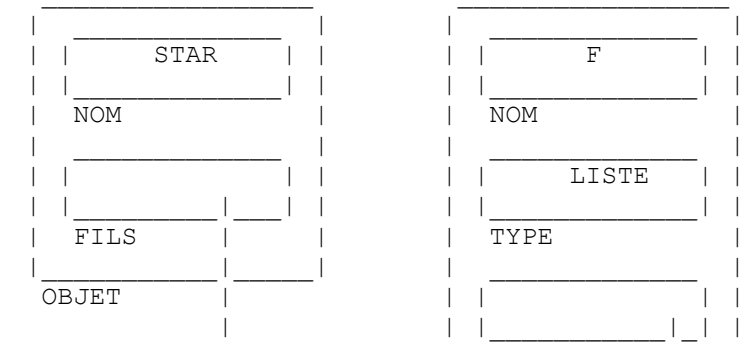
Avec le développement du modèle, nous espérons obtenir une meilleure solution de ce problème ; il est certainement raisonnable de terminer avec une structure dans laquelle il y a de nombreux cycles de compilation et d'interprétation, peut-être en formant une structure dans laquelle survient l'incorporation de cycles au sein de cycles.

4.10 Exemples d'opérations sur des LISTE

Nous développons ici un exemple d'une opération sur une LISTE en utilisant la primitive l/functions. On montre d'abord les invocations de la fonction nécessaires pour créer une LISTE nommée F et en lui donnant quelques valeurs de membres. On copie alors certains membres choisis dans une seconde LISTE G.
Pour créer F :

```
L/CREATE("STAR",LD/LIST(F,
                LD/STRUCT(R,
                    LD/STRING(A,FIXED,2),
                    LD/STRING(B,FIXED,2))))
```

Cela crée F comme membre du répertoire permanent STAR (voir au paragraphe 4.6 les détails sur STAR). Le symbole STAR a un statut particulier dans le "langage", en ce qu'il est un des quelques symboles atomiques à s'évaluer directement en objet. (On se rappelle que la plupart des objets permanents sont référencés à travers une invocation à L/POBJ ; réserver le symbole STAR est équivalent à réserver STAR comme nom et d'écrire L/POBJ(STAR). La solution choisie ici est plus facile à écrire.) L'exécution de cette invocation construit la structure montrée en 4-15 (sauf pour STAR, qui existe avant l'invocation). La valeur créée initialement pour F est une LISTE vide -- une LISTE de zéro membre.



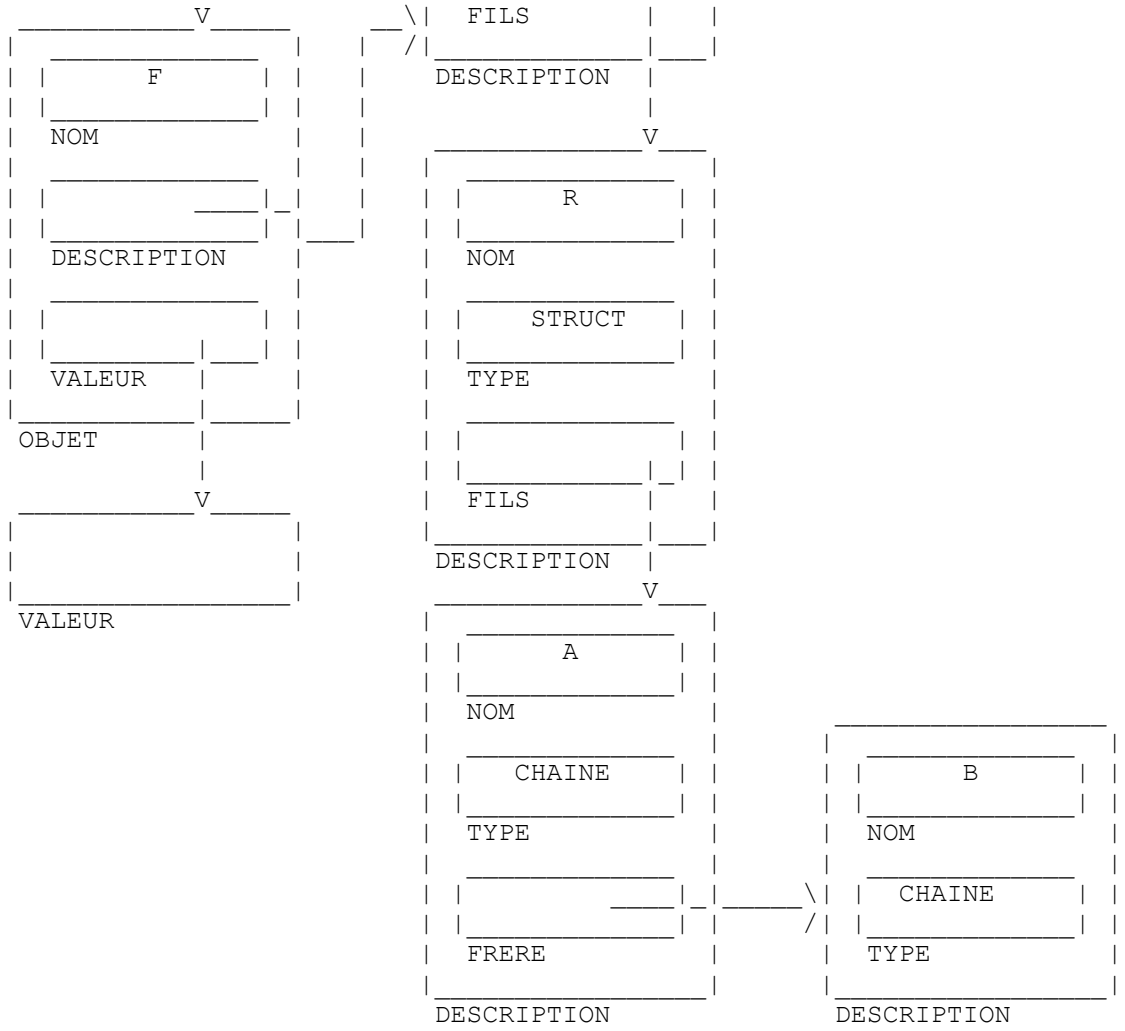


Figure 4-15 : F immédiatement après la création

Pour ajouter des membres à F, nous devons utiliser des listops, et pour cela nous devons créer deux objets de plus : un objet pour représenter le membre actuel et un descripteur d'opération (OPD). Ce sont des objets temporaires plutôt que permanents ; ils sont aussi de "niveau supérieur" (c'est-à-dire, pas locaux par rapport à une demande). Les objets temporaires, de niveau supérieur sont créés somme membres du répertoire TOP/LEVEL. Les invocations pour les créer sont :

```
L/CREATE(L/TOBJ(TOP/LEVEL),
        LD/STRUCT(M,
            LD/STRING(A,FIXED,2),
            LD/STRING(B,FIXED,2)))
L/CREATE(L/TOBJ(TOP/LEVEL),LD/OPD(OPF))
```

On crée M pour représenter le membre actuel ; sa description est la même que l'entrée pour un membre de F (voir l'invocation qui a créé F). La façon appropriée pour réaliser cela est d'utiliser un mécanisme qui partage la description réelle de membre de LISTE avec M ; cependant, ce mécanisme n'existe pas encore dans notre modèle.

Nous souhaitons maintenant ajouter des données à F ; chaque membre sera une STRUCT contenant deux CHAINES de deux caractères.

Pour commencer la séquence listop :

```
L/LISTOP/BEGIN(L/POBJ(F),L/TOBJ(M),L/TOBJ(OPF),ADD)
```

Cette invocation établit la structure montrée à la figure 4-16. Elle initialise l'OPD, le faisant pointer sur F et M et enregistre que seule la sous opération ADD est activée.

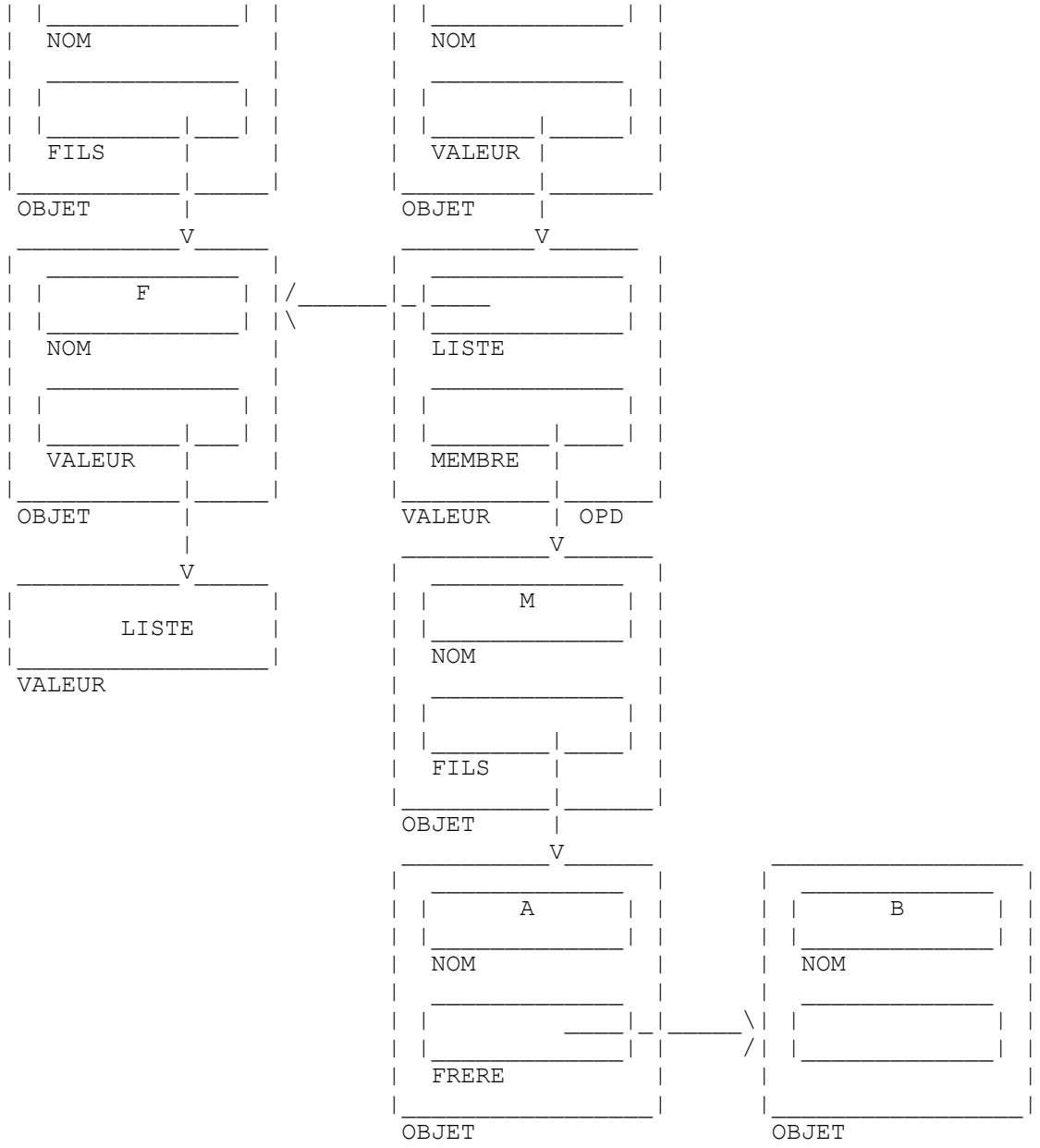


Figure 4-16 : F, OPF et M après L/BEGIN/LISTOP

Nous devons ensuite établir un membre actuel. Nous voulons ajouter des membres à la fin (dans ce cas, les ajouter n'importe où aurait le même effet, car la LISTE est vide) ce qui est fait en faisant de LAST le membre actuel.

L/WHICH/MEMBER(L/TOBJ(OP1),LAST)

Maintenant, pour ajouter un nouveau membre à F, nous pouvons exécuter ce qui suit :

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('AB'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('CD'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER crée une valeur de STRUCT pour M. Elle n'affecte pas la valeur de F. Chaque membre de la valeur STRUCT est initialisé lorsque la STRUCTURE est créée. Le résultat est montré en 4-17 ; noter que les valeurs de membre de STRUCT ne sont pas encore en relation avec les objets M.A et M.B.

La Figure 4-18 montre les changements accomplis par le premier L/ASSIGN ; le pointeur de l'objet M.A sur la valeur a été établi par un GET/STRUCT/MEMBER compilé par L/TOBJ(M.A). La valeur a été remplie par l'opérateur assign. Le second assign a un effet similaire, remplissant la seconde valeur. L'invocation de L/CLOSE/MEMBER prend la valeur montrée pour M en 4-18 (avec la seconde valeur de membre remplie) et lui ajoute la valeur de F. Le résultat est montré en 4-19 ; à comparer avec 4-16.

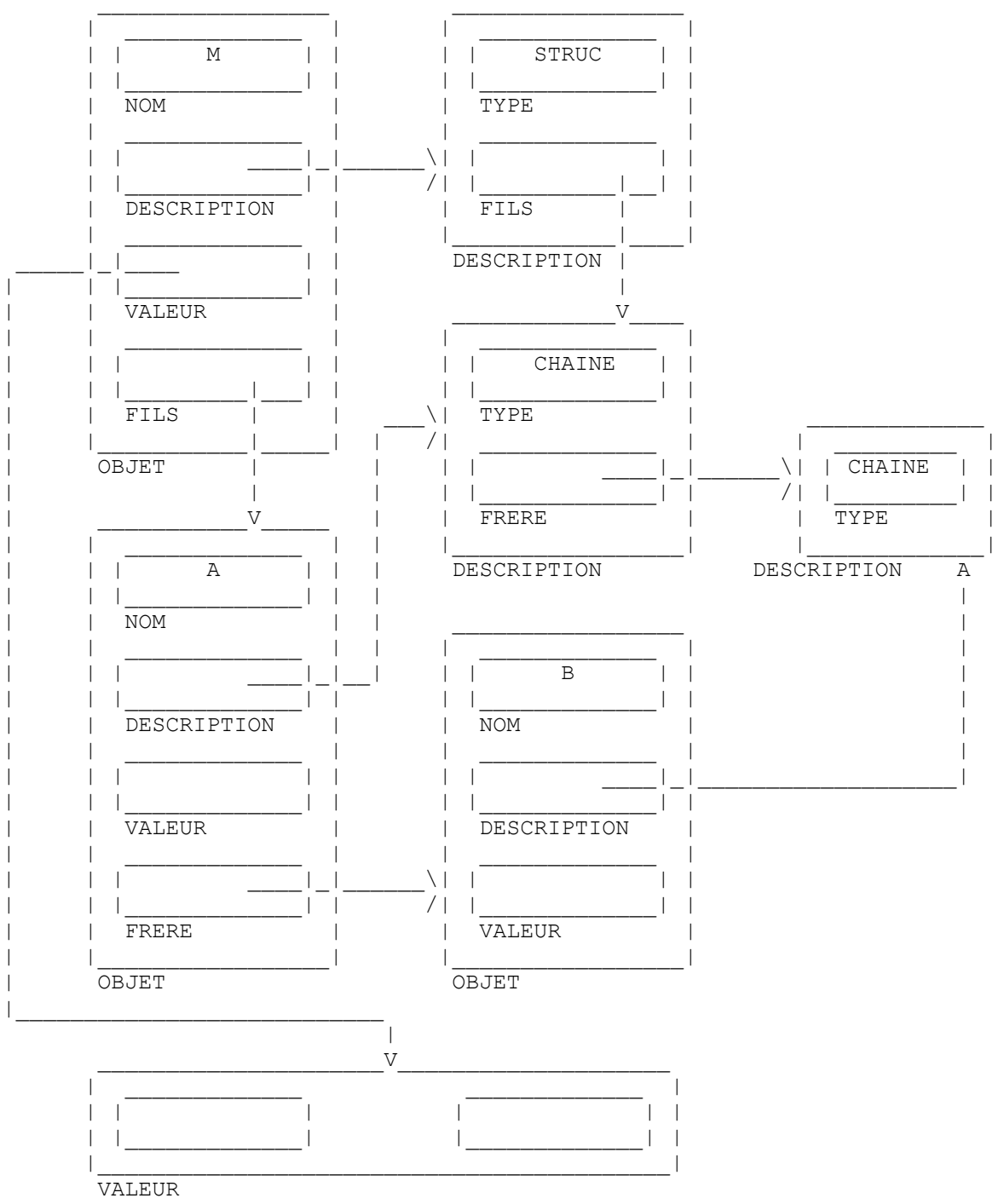
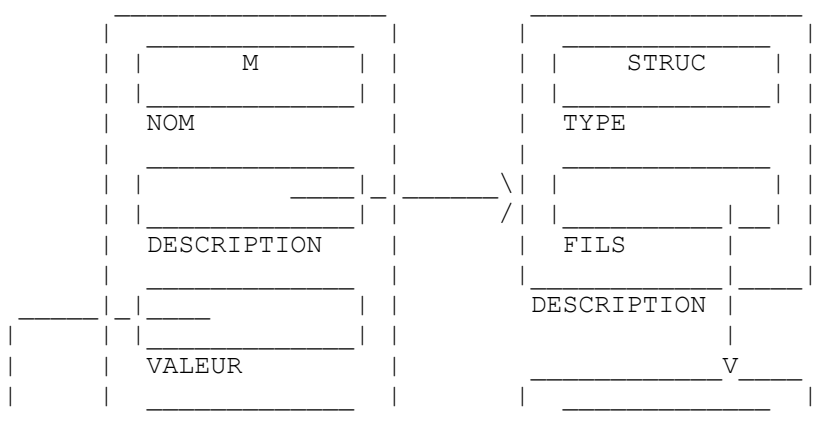


Figure 4-17 : Après L/OPEN/MEMBER



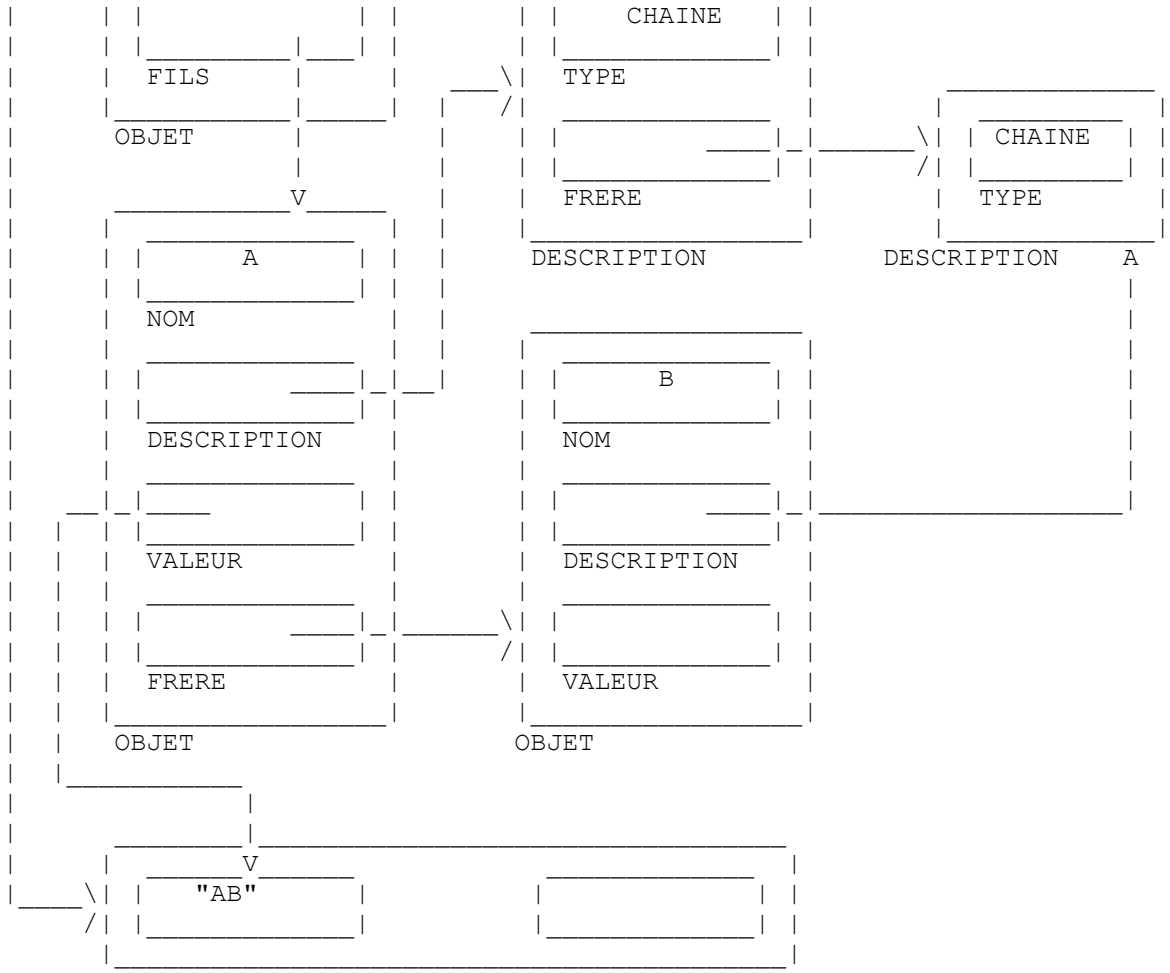
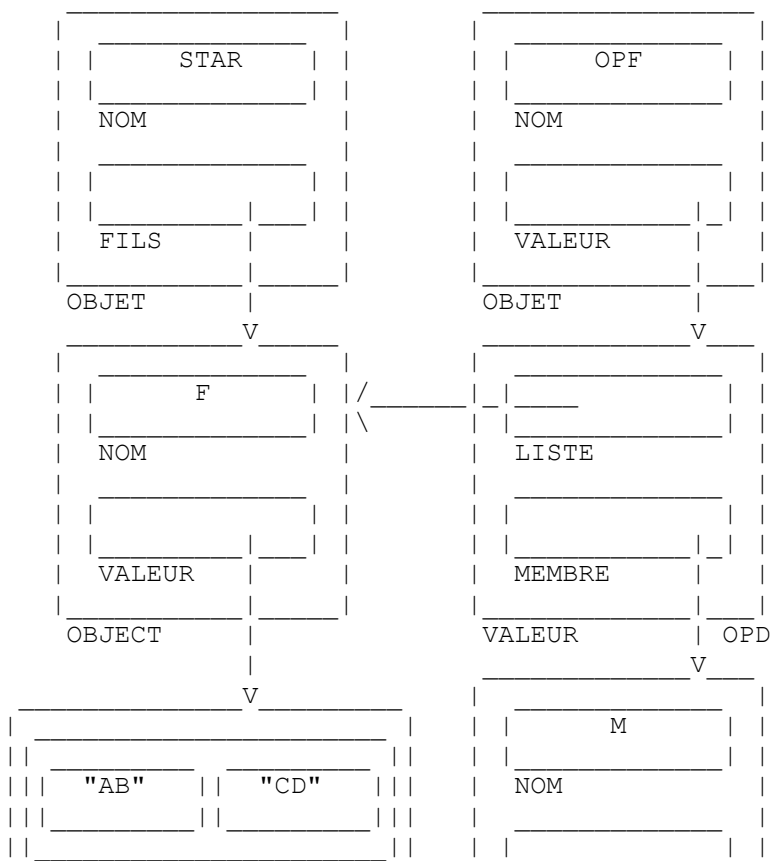


Figure 4-18 : VALEUR après le premier L/ASSIGN



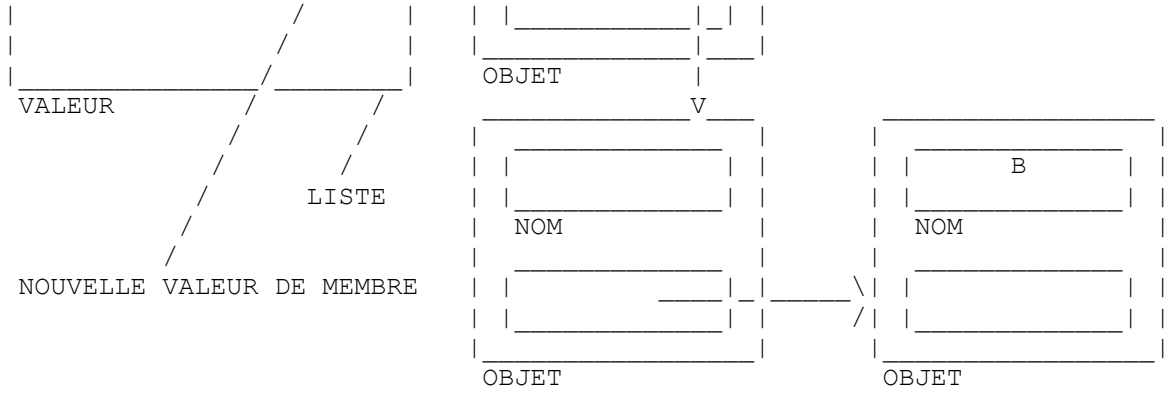


Figure 4-19 : Après L/CLOSE/MEMBER

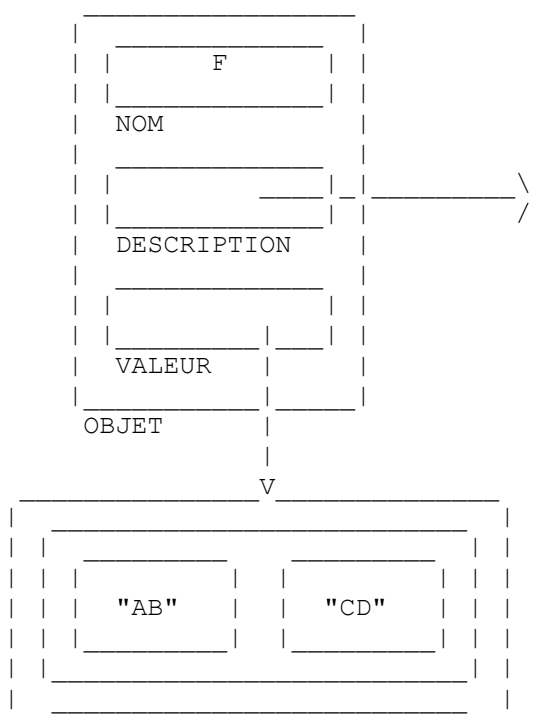
En exécutant des groupes similaire de quatre primitives, et en faisant varier seulement les valeurs des constantes, on peut construire la LISTE F montrée en 4-20. Les invocations requises sont montrées ci-dessous :

```
L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('FF'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('GH'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('AB'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('IJ'))
L/CLOSE/MEMBER(L/TOBJ(OPF))

L/OPEN/MEMBER(L/TOBJ(OPF),ADD)
L/ASSIGN(L/TOBJ(M.A),LC/STRING('CD'))
L/ASSIGN(L/TOBJ(M.B),LC/STRING('LM'))
L/CLOSE/MEMBER(L/TOBJ(OPF))
```

La sous opération add a pour effet de faire des membres qui viennent d'être ajoutés le membre actuel ; et donc aucune invocation L/WHICH/MEMBER n'est nécessaire dans cette séquence.

Pour terminer la séquence de listops :

```
L/END/LISTOP(L/TOBJ(OPF))
```



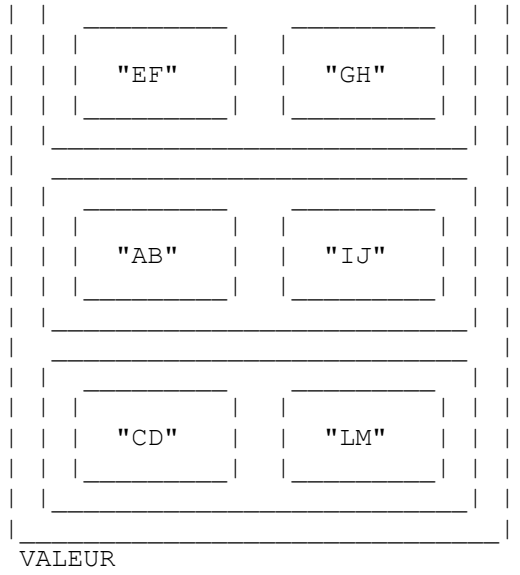


Figure 4-20 : Après L/END/LISTOP

Un exercice un tout petit peu plus intéressant est de construire des invocations qui créent une LISTE nommée G, ayant la même description que F, et de copier alors dans G tous les membres de F qui ont A égal à 'AB'.

Nous devons d'abord créer G, un OPD et un objet pour représenter le membre actuel.

```
L/CREATE("STAR",LD/LIST(G,
                LD/STRUCT(R,
                    LD/STRING(A,STRING,2),
                    LD/STRING(B,STRING,2)))
L/CREATE(L/TOBJ(TOP/LEVEL),LD/OPD(OPG))
L/CREATE(L/TOBJ(TOP/LEVEL) ,LD/STRUCT(GM,
                    LD/STRING(A,STRING,2),
                    LD/STRING(B,STRING,2)))
```

Nous avons maintenant besoin d'initier deux séquences de listops, une sur G et une sur F.

```
L/BEGIN/LISTOP(L/POBJ(F),L/TOBJ(M),
            L/TOBJ(OPF),GET)
L/BEGIN/LISTOP(L/POBJ(G),L/TOBJ(GM),
            L/TOBJ(OPG),ADD)
L/WHICH/MEMBER(L/TOBJ(OPF),FIRST)
L/WHICH/MEMBER(L/TOBJ(OPG),LAST)
```

Nous allons maintenant prendre à la suite les membres de F ; chaque fois que le membre actuel A est égal à 'AB', nous allons ajouter un membre à G. On copie ensuite les valeurs du membre actuel de F dans le membre de G qui vient d'être ajouté. Lorsque le membre actuel ne satisfait pas à ce critère, on ne fait rien avec lui.

D'abord, pour écrire une boucle qui va s'exécuter jusqu'à ce qu'on arrive à la fin de F :

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),x)
```

Tout ce qu'on mettra dans cette invocation à la place de "x" va s'exécuter de façon répétée jusqu'à l'établissement du fanion Fin de liste dans l'OPF.

On doit remplacer "x" par une seule invocation de fonction afin de donner à L/TILL ce qu'il cherche. Cependant, on va exécuter "x" une fois pour chaque membre de F, et on aura besoin d'exécuter plusieurs listops à chaque fois. La solution est d'utiliser L/CF, la fonction "compound-function " :

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),L/CF(y))
```

On peut maintenant remplacer "y" par une séquence d'invocations de fonctions.

À chaque fois qu'on fait l'itération, on a besoin de traiter un nouveau membre de F ; au départ, on est réglé pour obtenir le premier membre. La séquence suivante est alors nécessaire :

```
L/CF(   L/OPEN/MEMBER(L/TOBJ(OPF),GET),
    z
    L/CLOSE/MEMBER(L/TOBJ(OPF)),
    L/WHICH/MEMBER(L/TOBJ(OPF),NEXT) )
```

Ce qui figure ci-dessus est une fonction composée qui va ouvrir le membre actuel de F, lui faire quelque chose (représenté ci-dessus par "z") le fermer, et demander le membre suivant.

Nous voulons remplacer "z" par l'invocation d'une fonction qui vérifie le contenu de A dans le membre actuel de F, et soit ne fait rien, soit ajoute un membre à G, en copiant les valeurs du membre actuel de F. Si "w" représente l'action d'ajouter un membre à G et de copier les valeurs, nous pouvons alors exprimer cela par :

```
L/IF(L/EQUAL(L/TOBJ(M.A),LC/STRING('AB')),w)
```

Une façon convenable d'exprimer "ajouter un membre et copier les valeurs" est :

```
L/CF(L/OPEN/MEMBER(L/TOBJ(OPG),ADD),
     L/ASSIGN(L/TOBJ(GM.A),L/TOBJ(M.A)),
     L/ASSIGN(L/TOBJ(GM.B),L/TOBJ(M.B)),
     L/CLOSE/MEMBER(L/TOBJ(OPG))
```

Ceci est assez similaire à l'exemple précédent de sorte qu'aucune explication ne devrait être nécessaire.

Quand on met tout cela ensemble, on obtient :

```
L/TILL(L/END/OF/LIST(L/TOBJ(OPF)),
 L/CF(  L/OPEN/MEMBER(L/TOBJ(OPF),GET),
                                          L/IF(L/EQUAL(L/TOBJ(A),LC/STRING('AB')),
L/CF(   L/OPEN/MEMBER(L/TOBJ(OPG),ADD),
              L/ASSIGN(L/TOBJ(GM.A),L/TOBJ(M.A)),
              L/ASSIGN(L/TOBJ(GM.B),L/TOBJ(M.B)),
              L/CLOSE/MEMBER/L/TOBJ(OPG)) ) )
     L/CLOSE/MEMBER(L/TOBJ(OPF)),
     L/WHICH/MEMBER(L/TOBJ(OPF),NEXT) ) )
```

Pour conclure l'opération, on exécute :

```
L/LISTOP/END(L/TOBJ(OPG))
L/LISTOP/END(L/TOBJ(OPF))
```

Le résultat est une LISTE G dont le premier membre a la valeur ('AB','CD') et dont le second membre a la valeur ('AB','IJ'). Avec quelques variations à l'exemple ci-dessus, on peut effectuer quelques nouvelles opérations LIST.

4.11 Fonctions de niveau supérieur

Bien que ces i/functions primitives soient utiles, on ne s'attend pas à ce que les utilisateurs opèrent en langage de données à ce bas niveau. Nous voulons mettre ces primitives à la disposition des utilisateurs de telle façon qu'ils puissent traiter les cas exceptionnels, et qu'ils puissent construire leur propres fonctions de haut niveau pour les applications atypiques. Normalement, ils devraient opérer au moins au niveau de la construction suivante (qui est légale dans le langage de données réel actuellement mis en œuvre) :

```
FOR G.R,F.R WITH A EQ 'AB'
     G.R=F.R
     END
```

Cette expression relativement concise accomplit le même résultat que la construction élaborée de i/functions donnée à la fin du paragraphe précédent. On peut définir des i/functions très similaires aux fonctions sémantiques utilisées dans le logiciel actuel et écrire la demande ci-dessus sous la forme :L/FOR(L/POBJ(G),R

```
     L/POBJ(F),R,L/WITH(L/EQUAL(L/TOBJ(A),
                    LC/STRING('AB')))
```

Les différences entre l'invocation de la i/function et la demande en langage de données ci-dessus sont principalement syntaxiques.

Pour concevoir des fonctions comme L/FOR et L/WITH, le problème central est en rapport avec le choix des bonnes restrictions. On ne peut pas avoir toutes les généralités disponibles au niveau de la primitive. Certains choix importants pour ces fonctions particulières sont : (1) traiter des entrées et sorties multiples, (2) lorsque des "FOR" sont incorporés, comment les "FOR" externes restreignent t-ils les options disponibles aux "FOR" internes ? (3) quelle est la généralité des fonctions de sélection ? peuvent-elles générer des FOR à leur tour ? (4) quelles sont les options par rapport à l'endroit où devrait commencer le traitement ? (met on à jour, remplace t-on, ou ajoute t-on à la ou aux listes de sortie ?)
Finalement, ce problème se rapporte à celui plus général du traitement des ensembles, qui sont une généralisation de l'idée de collection de membres dans une LISTE qui ont des propriétés communes. FOR est seulement un des nombreux opérateurs qui peuvent entrer des ensembles.

4.12 Conclusion

Le présent modèle, bien qu'embryonnaire, contient déjà assez de primitives et de types de données pour permettre la définition et la manipulation généralisée de structures de données hiérarchiques. Les opérations de gestion de données courantes, telles que la restitution par contenus et la mise à jour sélective peuvent être exprimées.

L'utilisation de ce modèle dans le développement de ces primitives a résulté en spécifications précises, bien définies et cohérentes en interne pour les éléments de langage et les fonctions. Le travail dans l'environnement de laboratoire fourni par le modèle semble être un avantage substantiel.

5. Travaux à venir

Dans cette section, on passe en revue ce qui a été réalisé jusqu'à présent dans la conception, et on décrit les travaux qui restent à faire avant l'achèvement de la conception du langage de données.

5.1 Résumé

Le plus important parmi nos réalisations est que nous estimons avoir délimité les problèmes et présenté les grandes lignes d'une solution au développement d'un langage pour le système d'ordinateur de données. Les éléments clés de notre approche sont la primauté de la description des données en capturant tous les aspects des données, la séparation des caractéristiques logiques et physiques de la description des données, la capacité des utilisateurs à définir des vues différentes des mêmes données, la capacité à associer les fonctions à différentes utilisations des éléments des données, un essai de capturer les aspects communs des données à tous les niveaux possibles, et la capacité pour les utilisateurs de communiquer avec l'ordinateur de données au plus haut niveau que permet l'application.

5.2 Sujets de recherches à venir

Bien qu'il reste encore du travail pour pouvoir présenter un projet définitif du langage de données, on peut mettre en évidence certaines questions particulières qui appellent des investigations plus approfondies.

Jusqu'à présent, seules les structures de données hiérarchisées (c'est à dire celles qui peuvent être modélisées par un contenu physique) ont été développées jusqu'à un certain point. Nous avons aussi l'intention de rechercher et de fournir d'autres types de structures de données. Nous sommes certains que le cadre de notre langage ne repose pas sur des hypothèses qui interdiraient de tels ajouts.

Notre travail actuel sur la régulation d'accès est centré sur l'utilisation de descriptions multiples pour les données. Nous devons travailler encore les aspects à la fois techniques et administratifs de la régulation d'accès. Les problèmes du chiffrement des données aussi bien pour la transmission que la mémorisation doivent aussi être creusés.

Un autre question qui nécessite de pousser des recherches est celle des exigences du protocole sur le traitement des interactions avec l'ordinateur de données.

La séparation de la description en modules indépendants doit faire encore l'objet de recherches. En particulier, on doit chercher dans les travaux déjà réalisés sur des spécifications séparées de descriptions logiques, des descriptions physiques, et de la transposition entre les deux.

5.3 Syntaxe du langage des données

Nous n'avons pas encore proposé une syntaxe pour le langage de données que nous sommes en train de développer. Les parties les plus difficiles du problème sont certainement les questions de sémantique et les questions concrètes. Nous sommes confiants dans le fait que les diverses formes syntaxiques peuvent être choisies et mises en œuvre sans difficultés excessives. Il peut être préférable de développer différentes formes syntaxiques pour le langage pour différents types d'utilisateurs ou même pour les diverses sous parties du langage lui-même. Comme exposé à la section 2, la syntaxe d'utilisateur pour l'ordinateur de données est supposée être à un niveau bas. Il serait facile aux programmes de générer les demandes du langage de données dans cette syntaxe.

5.4 Travaux à venir sur le modèle du langage des données

Le modèle fournit d'excellentes fondations sur lesquelles construire un langage avec les facilités décrites à la section 3. Il reste encore beaucoup de travail à faire.

Pour encore un moment, l'accent sera mis sur les ensembles, les opérateurs de haut niveau, l'extension du langage et la description des données.

On prévoit de modéliser les ensembles comme un nouveau type de données, dont la valeur est normalement partagée avec d'autres objets. Encore un peu de travail sur la liaison et le partage des valeurs sera nécessaire pour prendre cela en charge.

Les ensembles peuvent être considérés comme un cas particulier de relations généralisées, ce qui viendra un peu plus tard.

Les opérateurs de haut niveau comme "FOR" seront construits à partir des primitives existantes, et seront finalement définis comme ayant un effet mais plusieurs expansions possibles. L'expansion va dépendre de la représentation des données et de la présence de structures auxiliaires.

D'autres expansions seront possibles lorsque la description des données aura été répartie entre divers modules. Ceci aussi exige des recherches approfondies.

Nous pensons que le problème de l'extension du langage est beaucoup plus facile à entreprendre dans l'environnement fourni par le modèle de l'ordinateur de données. En particulier, on prévoit que l'environnement de laboratoire va nous aider à évaluer les interactions complexes et les effets envahissants des opérateurs dans le langage qui étend le langage.

Le travail sur la description des données dans le court terme va se concentrer sur l'isolement des attributs, la représentation d'une structure variable dans la description, la description des descriptions et le développement d'un ensemble suffisant de types de données incorporés.

Plus tard, nous espérons modéliser la sémantique des pointeurs comme type de données, lorsque la représentation du pointeur et la sémantique de l'espace d'adresse dans lequel il pointe seront spécifiées dans la description du pointeur.

Un grand nombre de questions de niveau inférieur seront posées, y compris certains des problèmes découverts aujourd'hui dans la modélisation. Certaines d'entre elles sont abordées dans la section 4.

5.5 Prise en charge d'applications

Le langage de données que nous sommes en train de concevoir est destiné à fournir des services aux sous systèmes qui résolvent une large classe de problèmes qui se rapportent à la gestion des données. On a comme exemples de tels sous systèmes les générateurs de rapports, les systèmes d'interrogation en ligne pour non programmeurs, les systèmes de traitement de documents, les systèmes de traitement de transactions, les systèmes de collecte de données en temps réel, et les systèmes de bibliothèque et de bibliographie. Il y en a bien d'autres.

L'idée est que de tels systèmes vont fonctionner sur d'autres machines, faire référence ou mémoriser des données dans l'ordinateur de données, et faire un usage intense du langage de données. Un tel système ne sera pas écrit entièrement en langage de données, mais une grosse composante de sa fonction serait exprimée en requêtes du langage de données ; certains modules de contrôle vont construire les requêtes et effectuer les fonctions qui ne sont pas en langage de données.

Bien que nous ayons l'expérience de telles applications dans d'autres environnements, que nous ayons parlé à des utilisateurs potentiels, il nous faudra encore du travail pour déterminer si notre langage est réellement adéquat pour eux. C'est à dire que nous ne nous attaquons pas directement au problème de construire un système d'interrogation en ligne adapté à l'utilisateur humain ; nous essayons de fournir les outils qui rendront facile sa construction. Il y a un besoin bien défini d'analyse des outils qui seront vraisemblablement les bons. Bien sûr, l'essai ultime sera l'utilisateur réel, mais nous voulons évacuer autant de problèmes que possible avant la mise en œuvre.

Un composant important de la prise en charge d'applications est que les programmes utilisateurs seront fréquemment écrits dans des langages de haut niveau tels que le FORTRAN, le COBOL ou PL/1. Nous devons examiner la capacité du langage de données à prendre en charge de tels utilisateurs avant que la conception ne soit figée.

5.6 Plans pour le futur

Cet article a posé les fondations d'un nouveau concept de langage de données.
La Section 3 donne les grandes lignes d'un concept de langage de données, qui va être habillé durant les mois à venir. Après la production d'une spécification détaillée, nous prévoyons une relecture intensive, des révisions, et l'incorporation dans un plan de mise en œuvre. La mise en œuvre interviendra par étapes, compatibles avec les plans établis pour le développement d'un service d'ordinateur de données avec des capacités de gestion des données.