Centre de Recherche en Informatique de Nancy
Institut National Polytechnique de Lorraine

Thèse

pour l'obtention du diplôme de

Doctorat en Informatique de l'Institut National Polytechnique de Lorraine

Antoine Menegaux

Ancien Elève de l'Ecole Polytechnique
Ingénieur à CR2A

SAMPI : Système d'Aide à la Maintenance des Programmes Interactif

Soutenue publiquement le : 10-11-88

Rapporteurs	P. Lescanne A. van Lamsweerde
Membres du jury	J.-P. Thomesse J.-P. Finance
Invités	D. Chandesris B. Lang J. Guyard

Centre de Recherche en Informatique de Nancy
Institut National Polytechnique de Lorraine

Thèse

pour l'obtention du diplôme de

Doctorat en Informatique de l'Institut National Polytechnique de Lorraine

Antoine Menegaux

Ancien Elève de l'Ecole Polytechnique
Ingénieur à CR2A

SAMPI :
Système d'Aide à la Maintenance des Programmes Interactif

1988

Soutenue publiquement le : 10-11-88

Rapporteurs	P. Lescanne A. van Lamsweerde
Membres du jury	J.-P. Thomesse J.-P. Finance
Invités	D. Chandesris B. Lang J. Guyard

* SAMPI: l'outil proposé veut aller au-delà de l'Alpha et de l'Oméga. Dans la numération grecque, α=1, β=2, γ=3, ..., χ=600, υ=700, ω=800, il manquait une lettre pour 900, aussi les Grecs utilisaient-ils l'ancienne lettre Sampi ϡ.

Remerciements

Je remercie :

Monsieur Pierre Lescanne, directeur de recherche au CNRS, responsable de l'équipe de recherche Eureca, au CRIN,

Monsieur Axel van Lamsweerde, professeur à l'Institut d'Informatique Notre-Dame de la Paix, à Namur,

qui ont bien voulu être les rapporteurs de ma thèse.

Je remercie :

Monsieur Jean-Pierre Thomesse, professeur à l'ENSEM, responsable de l'équipe de recherche Informatique Industrielle, au CRIN,

Monsieur Jean-Pierre Finance, professeur à l'Université Nancy I, directeur du CRIN,

qui ont encadré mes travaux de recherche.

Je remercie :

Monsieur Dominique Chandesris, conseiller attaché à la direction, à CR2A,

Monsieur Bernard Lang, directeur de recherche INRIA,

Monsieur Jacques Guyard, maître de conférences à l'Université Nancy I, membre de l'équipe de recherche Programmation, au CRIN,

qui ont accepté de faire partie de mon jury de thèse.

Je remercie :

Monsieur Michel Alard-Couluon, directeur, à CR2A,

Monsieur Jean-Marie Helmer, directeur technico-commercial, à CR2A,

pour l'intérêt qu'ils ont su porter à mes travaux.

Je remercie ici tous ceux qui m'ont assisté et soutenu durant mon travail de thèse. Je remercie aussi Fabrice Moitessier qui a bien voulu s'acquitter de la tâche ingrate de relecture du document.

Plan

Chapitre 1:

Le Problème et la Proposition

On cherche à définir un outil pour établir, dans le texte source d'un programme, des liens sémantiques entre des points *"physiquement"* distincts mais *"logiquement"* voisins. Le but est d'aider à l'**écriture** du programme – on peut établir un lien vers un modèle prédéfini, fiable, général, ... – et à la **lecture** du programme – on fixe dans les liens le cheminement logique de conception du programme.

1. Présentation

1.1. le problème

Le problème auquel on s'attache est le suivant: rédiger un programme, dans un langage informatique, s'élabore en plusieurs étapes:

- Dans un premier temps, de *saisie du programme*, on construit un nouveau programme. A cette étape on bénéficie d'une connaissance complète du programme qu'on construit: on sait à quel point du programme est déclarée une structure, à quel autre point elle est utilisée; on sait qu'on répète plusieurs fois un même schéma d'algorithme, instancié sur diverses structures de données; on sait qu'on satisfait des contraintes d'implantation, par exemple dans le séquencement de plusieurs instructions; etc...

- Dans un deuxième temps, de *correction du programme*, on relit le texte source saisi, et on le modifie dans un persepective d'évolution; ceci inclut la phase de mise au point et celle de maintenance du programme – on se place donc une semaine ou six mois plus tard.
  La difficulté à laquelle se heurte alors l'Informaticien est de *reconstruire*, par un effort de synthèse, le processus de rédaction du programme initial, qui tienne compte de toutes les informations cachées, de dépendances implicites entre morceaux du texte source.

1.2. les quatre soucis

Les quatre soucis auxquels on souhaite apporter une réponse sont:

Souci 1: lisibilité
Fixer dans la R.I. du texte les liens sémantiques qu'on a d'emblée reconnus facilite une relecture de ce texte – le lecteur ne doit pas reconstruire toutes les dépendances sémantiques puisque le support même du texte les contient déjà.

Souci 2: traçabilité
C'est une conséquence directe de la R.I. choisie: puisque tous les concepts sémantiquement liés sont réellement liés, on peut déduire assez finement les dépendances entre des zones distinctes du programme.

Souci 3: "compacité"
C'est éviter la dispersion de concepts qui ne traduisent qu'une seule notion sémantique. On peut remarquer que fréquemment l'expression informelle d'un problème est très succinte alors que sa traduction informatique est longue et complexe. Lier des notions sémantiques n'est pas nécessairement un processus linéaire, c'est-à-dire que ces notions peuvent aussi être emboîtées: cet emboîtement permet alors le regroupement d'une information disséminée.

Souci 4: réutilisabilité
Si l'on distingue clairement ce qui constitue la substantifique moëlle du programme de ce qui n'est que l'expression de choix techniques d'implantation, alors l'essentiel du programme est réutilisable dans de nouveaux contextes.

1.3. la réponse

La réponse est d'offrir à l'utilisateur un outil qui permet de construire des textes sources dans lesquels les dépendances sémantiques entre divers morceaux de texte sont *conservées* dans la Représentation Interne (R.I.) du programme. Cet outil est un éditeur, qu'on appelle:

l'éditeur à références concentrées.

1.4. la démarche

Le sujet étant posé, il s'agit de proposer un solution pour représenter, dans les textes sources, les liens de dépendance entre des points éloignés.

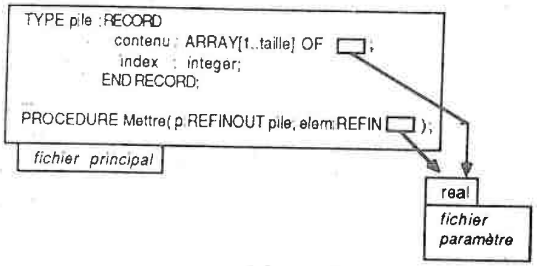
La difficulté à laquelle on est confronté est double:
- savoir attacher un lien de dépendance entre un point du texte et un autre texte;
- savoir fournir, à l'instanciation d'un modèle, un tel lien qui soit connu de l'objet instancié et non pas du modèle générique.
  Ce dernier point est nécessaire si l'on veut utiliser un même modèle avec des paramètres effectifs différents.

Le choix proposé, pour introduire cette structure de *graphe* dans les textes sources, a été de NOMMER les textes, de les EVALUER, et plus tard de s'y REFERER (cf. Chapitre 5.1, *«L'évaluation fonctionnelle»*).

2. L'éditeur à références concentrées

L'outil présenté est un éditeur à références concentrées. Le but poursuivi est de concentrer en un unique endroit tout choix de réalisation opéré à l'écriture d'un texte de programme.

A un premier niveau d'utilisation, le schéma est:



Par ce mécanisme, on peut paramétrer un texte par n'importe quel constituant du langage (ici: un type, bien que LTR3 ne soit pas un langage générique).

On a en plus la vision complète de l'objet réalisé: si l'on définit une pile de réels, on ne lira pas dans le texte source:

«pile de "element"; "element" = "real"»

mais directement:

«pile de "real"»

Ainsi la paramétrisation n'affecte pas la facilité de lecture puisque c'est l'objet final qu'on lit sous l'éditeur.

La première conséquence est alors qu'il faut, dans l'éditeur, définir la notion de *«zone d'affichage en lecture seulement»*. En effet à un endroit où apparaît "real" dans l'exemple précédent il ne faut pas avoir la possibilité de modifier le terme: autrement ce sont toutes les occurrences de "real" qui seraient simultanément modifiées – c'est bien dans ce cas non pas le «fichier principal» mais le «fichier paramètre» que l'on modifie.

La deuxième conséquence, réciproque de la précédente, est que l'éditeur doit offrir un confort suffisant pour déclarer une certaine zone de texte comme *«zone d'affichage en lecture seulement»*. Ainsi, pour passer d'une première rédaction de la pile comme:

«pile-de-réels»

à une deuxième:

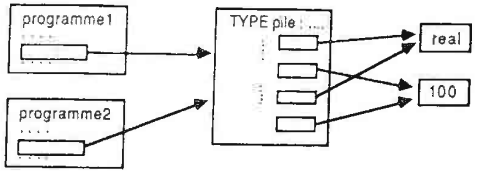
«pile-paramétrée-par-le-type-élément; élément = réel»

on n'a pas à réécrire la portion de programme qui définit la pile générique mais simplement à indiquer sous l'éditeur où sont les zones faisant référence à un même paramètre (ici: réel).

Ceci est certainement avantageux, car il est à mon sens plus facile de définir un modèle générique *par l'exemple*:

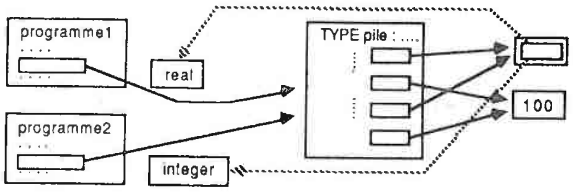
- on contruit un premier objet;
- on identifie ce qui correspond à des paramètres de l'objet;
- on construit le type paramétré, en réalisant alors l'objet comme une instance de ce type.

Au deuxième niveau, on peut considérer les objets introduits sous la forme d'une arborescence:



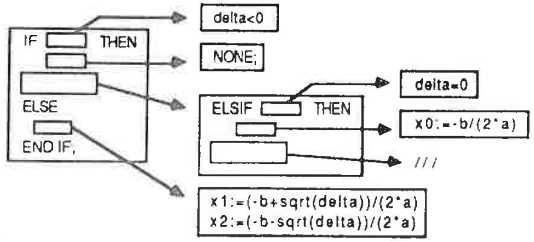
(pile de réels; de taille 100)

puis de graphe:



(le type des élements de la pile est indéfini;
la taille des piles est toujours 100)

On peut remarquer que ce mécanisme permet, en corollaire, la construction d'un éditeur syntaxique. Il suffit pour cela de définir toutes les structures syntaxiques du langage utilisé sous forme de type paramétré:



Note:
Chaque utilisation d'identificateur pourrait également être concentrée (a, b, delta, ...), mais le dessin deviendrait trop confus.
On notera que le dessin deviendrait confus mais non le texte, saisi ou lu. En effet la décomposition syntaxique est forcément arborescente; l'utilisation d'identificateurs en zone concentrée consiste en une redirection de la valeur du nœud vers une référence commune et ne constitue donc pas un élément de l'arbre syntaxique; c'est simplement un autre découpage du texte, selon une autre approche.

Un dernier point concerne les commentaires informels attachés au programme: ceux-ci peuvent également se référer à des zones concentrées du texte source, la cohérence entre les commentaires et le programme est ainsi mieux réalisée.

3. Présentation du document

A un premier niveau, on identifie une famille d'opérateurs primitifs pour représenter les «textes» : ils constituent la Syntaxe Concrète, présentée dans le **Chapitre 2**.

A un deuxième niveau, on introduit de nouveaux opérateurs, parce que les opérateurs précédents sont insuffisants pour traduire certaines dépendances qu'on trouve au sein des textes sources : pour l'essentiel, il s'agit des notions de *code Lisp inséré dans le texte* et de *liste de textes*. Ces opérateurs, ajoutés à la Syntaxe Concrète, constituent la Syntaxe Complétée, qu'on présente dans le **Chapitre 3**.

Enfin, à un troisième niveau, l'introduction de nouveaux opérateurs amène la question suivante : A-t-on bien exhaustivement balayé l'ensemble des types de dépendances qui apparaissent dans les textes sources? Pour éviter de devoir y répondre, on transforme la manière dont est défini le langage : on l'exprime sous la forme d'une Syntaxe Abstraite, et, par l'utilisation d'un traitement automatique, on s'autorise à facilement compléter le langage. Ceci fait l'objet du **Chapitre 4**.

Le **Chapitre 5** reprend, plus formellement, les présentations antérieures :
- on donne la sémantique opérationnelle du langage primitif, défini dans la Syntaxe Concrète;
- on donne une preuve de l'algorithme de modification incrémentale de la Syntaxe Abstraite.

Le **Chapitre 6** replace l'outil en comparaison des outils ou démarches existants.

En conclusion, on présente les perspectives de poursuite des travaux : il s'agirait déjà de soumettre l'outil à un test «en vraie grandeur» sur un exemple de taille significative; il s'agirait ensuite d'introduire la composante sémantique dans les «textes», morceaux de programme, qu'on est conduit à identifier. C'est le **Chapitre 7**.

Les deux derniers Chapitres, placés en Annexe, traitent des réalisations informatiques liées aux travaux.

Le **Chapitre 8** présente les deux éditeurs réalisés : l'*éditeur ligne*, qui supporte la totalité des fonctionnalités présentées mais est d'un emploi malaisé, et l'*éditeur page*, qui n'est que partiellement défini.

Le **Chapitre 9** regroupe trois aspects d'implantation qu'on a jugé intéressant de retenir:
- les contexte d'évaluation : on montre, par de nombreux exemples, comment sont évalués les opérateurs du langage;
- le manuel du concepteur de la syntaxe abstraite : il permet la définition de nouveaux termes du langage;
- le guide de l'implanteur de l'éditeur page : il expose les choix d'implantation retenus pour l'éditeur page.

4. Conclusion

On propose la définition d'une **Représentation Interne** *simple* et *extensible* pour la manipulation des objets d'un environnement de programmation – les documents de spécification, les programmes sources, les jeux de test, ... Ce que l'on souhaite offrir à l'utilisateur c'est un outil qui lui permette d'exprimer, à chaque étape de développement du logiciel, les décisions de conception qu'il est amené à faire:
- on le laisse s'exprimer, c'est-à-dire qu'à chaque instant il est libre d'adopter le choix qui lui convient;
- on lui donne les moyens de signifier ses intentions, qui sont alors conservées dans la forme interne des objets qu'il construit au même titre que les informations qu'il y attache.

Cet outil est un éditeur de texte; il réalise l'interaction de l'utilisateur avec ces objets.

La Représentation Interne se veut **simple**: elle se définit par un très petit nombre de concepts, ce qui en facilite et la compréhension et l'emploi. Mais de cette simplicité il résulte qu'on ne garantit ni la cohérence entre les objets construits ni la génération automatique de certains d'entre eux – par exemple celle des programmes sources.

Elle se veut aussi **extensible**: considérant qu'il serait très délicat d'établir une liste exhaustive des *types d'intentions* de l'utilisateur, on a cherché à développer une expression de cette Représentation Interne aussi indépendante que possible de l'état dans lequel on l'a arrêtée.

Chapitre 2:

Le Langage Primitif de Représentation Textuelle

C'est la première étape de définition des «textes». Elle autorise une structure modulaire du texte source de l'application à partir d'un nombre minimal de concepts: *la définition, l'utilisation, la référence*. On donne ici la syntaxe et la sémantique des termes introduits, et des exemples de mise en œuvre.

Chapitre 2.1:

Présentation de la Syntaxe Concrète

On présente les premières notions introduites, qui réalisent la *Syntaxe Concrète*:
- def = on définit un «texte» en le nommant,
- use = on utilise un «texte» par son nom,
- ref = on se réfère à un «texte» par son nom.

Un texte peut être vu pour sa valeur de représentation (use) ou sa valeur d'environnement (ref).

1. Les intentions

1.1. L'objectif

L'élément essentiel qui a guidé les travaux est la constatation suivante: un programme informatique, exprimé à l'aide d'un langage informatique, a le fâcheux trait de comporter un grand nombre de répétitions: répétitions de texte, de traitements, d'identificateurs. La définition même des langages modernes – typés, à structure de bloc – nécessite à l'introduction d'un concept une répétition; d'une part on *déclare* l'objet, d'autre part on l'*utilise*. Le lien sémantique entre la déclaration et l'utilisation d'un même concept est très généralement réalisé par la reconnaissance lexicale d'un identificateur; c'est le *«paradigme DESIGNER»*, une des *«étapes élémentaires»* dans l'analyse d'un programme [Gra 86].

Ces répétitions ne s'arrêtent pas aux seuls identificateurs: on peut avoir des répétitions sur les traitements, mais celles-ci sont alors plus difficiles à détecter, et plus encore à expliciter, parce que ces traitements ne travaillent pas nécessairement sur les mêmes structures de données. On introduit alors la notion de *généricité* qui permet de définir un traitement sur un type d'objet auquel on ne demande qu'un nombre minimal de propriétés. C'est ainsi qu'on procède dans la *«méthode Jackson»*: on définit le traitement, sans préciser la structure exacte des objets qu'on manipule; puis on instancie ces traitements par le choix d'une structure de données complètement définie [Cam 86]. C'est aussi un des points essentiels de la Méthode Orientée Objet proposée par G. Booch [Boo 86]. La généricité ne résoud cependant pas totalement le problème: on réintroduit en effet des répétitions par la définition des paramètres de généricité, formels et effectifs; on est en plus relativement limité dans le choix du langage de programmation.

La question est de savoir si l'on peut éviter les répétitions dans un texte source, le but n'étant pas de s'économiser des efforts inutiles mais de conserver son programme sous une forme lisible et maintenable.

1.2. La réponse

La réponse apportée consiste non à les supprimer mais à les transformer: la répétition d'une même notion est transformée en l'*utilisation* d'une même notion, laquelle est alors définie une unique fois. Par exemple l'identificateur qui désigne une variable du programme ne sera pas répété de la déclaration à l'utilisation de cette variable: l'une et l'autre utiliseront le même objet, et c'est cet objet qui définira l'identificateur.

Réaliser une telle «concentration» des définitions nécessite alors le développement de deux outils:
- un éditeur de texte structuré, qui sache manipuler aussi bien les caractères – les zones où aucune répétition n'est décelée – que les utilisations de zones communes – qui correspondent aux répétitions;
- un environnement de programmation, où sont gérés de façon cohérente les textes sources et les définitions de texte «concentré».

L'éditeur de texte précédant logiquement l'environnement de programmation (c'est l'éditeur de texte qui fournit à l'environnement les objets que manipulera ce dernier), c'est sur celui-ci que l'accent a principalement été mis.

1.3. L'éditeur syntaxique

Le grand nombre d'éditeurs syntaxiques actuellement existant sur le marché laisse à penser qu'on peut essayer d'utiliser l'un d'eux.
Ces éditeurs ont généralement les caractéristiques suivantes:
- on rédige la grammaire du langage de programmation qu'on veut manipuler, dans un langage particulier défini pour l'éditeur;
- on appelle un générateur qui, partant de l'expression de la grammaire du langage, va construire un éditeur spécifique à ce langage;
- on peut entreprendre une session sous l'éditeur: celui-ci assurera qu'on construira des programmes syntaxiquement corrects:
  - soit parce qu'on utilise un opérateur du langage,
  - soit parce qu'on instancie un atome du langage par une valeur lexicalement correcte.

De tels éditeurs réalisent donc la concentration d'informations, mais *uniquement* dans les définitions d'opérateurs. Cette concentration porte d'une part sur la "nature" – le phylum – des paramètres à fournir à l'utilisation d'un opérateur pour obtenir un texte syntaxiquement complet; d'autre part sur le schéma de décompilation de cet opérateur. Les atomes instanciables du langage doivent satisfaire certaines exigences d'ordre lexical mais n'ont généralement pas d'autres liens avec le reste du programme.

La solution envisagée consisterait donc à définir, dans l'optique d'une concentration d'un "morceau de texte", un nouvel opérateur du langage dont le schéma de décompilation serait ce "morceau de texte", et à utiliser ensuite cet opérateur à chaque occurrence de ce "morceau de texte".

1.4. L'état de la science dans le domaine

A ma connaissance, il n'existe pas à l'heure actuelle d'éditeur syntaxique supportant une *modification dynamique* de la syntaxe du langage édité. Ceci s'explique sans doute pour plusieurs raisons:
- *difficulté algorithmique*: la création d'un nouvel opérateur ne devrait pas être trop complexe, en revanche la modification ou la suppression d'un opérateur existant peuvent avoir de graves conséquences sur la totalité du programme en cours d'édition;
- *gestion des programmes*: si la syntaxe évolue, des programmes anciennement édités ne pourront pas forcément être réédités, ou tout au moins certaines branches de l'arbre syntaxique pourront être perdues;
- *position du problème*: à mon sens les développeurs d'éditeurs syntaxiques ont privilégié la phase d'utilisation de l'éditeur, en considérant que la phase de construction d'un éditeur était un préalable nécessaire mais non une finalité.

De cette constatation il résulte que la solution qu'on a précédemment envisagée ne peut pas raisonnablement être réalisée au sein d'un éditeur syntaxique existant. De là s'expliquent les choix qui sont présentés ensuite, qui cherchent à répondre au problème tel qu'il vient d'être présenté.

1.5. L'expression de besoin

On retient l'idée de la définition d'opérateurs. La contrainte qu'on s'impose est alors qu'étant placé sur un nœud de l'arbre syntaxique en cours d'édition on puisse définir un nouvel opérateur, dont le schéma de décompilation sera le texte associé au sous-arbre attaché à ce nœud et dont les paramètres seront l'ensemble des nœuds non encore instanciés de ce sous-arbre. Ceci se rattache directement à la notion de *méta-variable* telle qu'on la trouve classiquement dans un éditeur syntaxique; la différence est qu'on ne considère pas ici ce concept comme appartenant à l'éditeur – et ayant de ce fait une durée de vie limitée au temps de la session sous l'éditeur – mais qu'on le considère comme attaché au tampon («buffer») en cours d'édition: il sera donc conservé dans la Représentation Interne (R.I.) de ce tampon au même titre que les diverses instances d'opérateurs qui le composent.

Pour les opérateurs de liste, il faudra pouvoir définir un nouvel opérateur qui ne retienne qu'une partie de cette liste. Par exemple, il pourra s'agir de deux instructions. A un niveau de détail plus fin, considérant qu'un identificateur est une liste de caractères, il s'agira alors d'un certain nombre de caractères consécutifs.

Au cœur du problème se pose le choix de la R.I. L'éditeur qui va manipuler cette R.I. doit en effet avoir une vision uniforme de deux aspects bien distincts du programme:
– d'une part les formes instanciées d'opérateurs déjà définies: il s'agit typiquement d'un tampon d'édition – soit encore un sous-arbre de l'arbre syntaxique;
– d'autre part des définitions actuelles et à venir d'opérateurs, qui fournissent les schémas instanciables.
En effet, toute zone cohérente d'un tampon d'édition (c'est-à-dire toute zone qui ne chevauche pas deux branches distinctes de l'arbre syntaxique) peut être vue soit comme une zone du tampon soit comme l'instance d'un nouvel opérateur. Le passage de l'une à l'autre de ces deux visions doit être le plus aisé possible.

1.6. Le choix de la Représentation Interne

Pour spécifier la Représentation Interne (R.I.) on a défini un *Langage de Construction de Langage* (LCL). Ce langage est bien à différencier d'un *Langage de Description de Langage* (LDL) tel qu'on en trouve classiquement dans un éditeur syntaxique. Un LDL permet de définir les opérateurs et les phyla d'un langage, avec de plus ou moins riches compléments sémantiques. Mais le schéma traditionnel d'emploi d'un tel langage:
-> édition d'un programme en LDL
-> compilation
-> interprétation du code objet (c'est-à-dire session sous l'éditeur syntaxique)
ne permet de retour immédiat au source LDL initial. On peut cependant noter que l'étape de compilation permet d'assurer un grand nombre de contrôles concernant le bien-fondé de la syntaxe définie.
Le LCL à l'opposé n'offre guère de contrôles de cohérence mais il permet une *interprétation* du texte qui définit le langage, c'est-à-dire une construction *incrémentale* de la syntaxe du langage édité.

La différence porte donc plutôt sur la mise en œuvre du langage que sur le langage lui-même; mais l'une entraîne l'autre puisqu'on adopte une représentation «graphique» et non syntaxique des concepts des langages utilisés, ce qui nécessite alors une grande *simplicité* des notions introduites – afin d'éviter une trop importante diversité des formes graphiques définies.

Au départ, on définit les deux opérateurs principaux:
- def <nom> <rep>
  On définit un opérateur de nom <nom> de représentation <rep>: <rep> est le schéma de décompilation de cet opérateur.
- use <nom>
  On utilise l'opérateur de nom <nom>, s'il existe.

Un opérateur d'arité non nulle est une définition de texte def pour laquelle on trouve dans sa représentation des utilisations d'opérateurs use.

Comment va travailler l'éditeur? il évalue les termes du langage (LCL). Une définition est évaluée identiquement à elle-même. Pour une utilisation d'un nom <nom>, on commencera par chercher une définition du nom <nom>: si on en trouve une, alors l'utilisation sera *«visuellement»* remplacée par la représentation associée; si on n'en trouve pas, un affichage spécifique indiquera à l'utilisateur l'absence d'une définition visible de cet opérateur. *«Visuellement»* signifie qu'un outil interactif doit afficher à l'écran la forme décompilée des arbres syntaxiques manipulés, même si dans sa R.I. il ne conserve pas le texte affiché mais le nom de l'opérateur utilisé.

La notion de «recherche de définition» introduit celle de «contexte d'évaluation»: s'il faut rechercher une définition, où la rechercher? Pour cela on s'inspire des mécanismes utilisés dans les langages à structure de bloc.

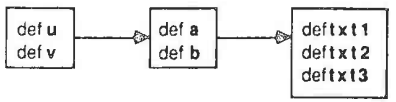
Une définition se présente en fait sous la forme:
- def <nom> <env> <rep>
  <env> est l'environnement des «définitions locales» de l'opérateur <nom>: on y trouve d'autres définitions def.

La recherche d'un opérateur <nom> s'effectue alors en construisant le contexte d'évaluation relatif à l'utilisation, puis en recherchant, dans l'ordre des environnements, la première définition de nom <nom>.

Par exemple:

```
def txt1
  env: def a
         env: def u
              def v
         rep: ...
       def b
         ...
  rep: ...
def txt2
  ...
def txt3
  ...
```

Dans la représentation de u, une utilisation de nom sera recherchée dans le contexte:



Au sommet on a alors l'«environnement global» qui est la liste de tous les opérateurs globalement définis, ou encore de tous les tampons en cours d'édition sous l'éditeur.

1.7. La réponse au problème

Par rapport au problème initial – la répétition de texte – on peut juger qu'on n'a fourni qu'une réponse partielle. En effet il peut exister des liens sémantiques entre deux éléments *différents* d'un programme, c'est-à-dire deux éléments qui n'ont pas le même schéma de décompilation. Un exemple typique est la variable et son type; ces deux objets sont différents à tout point de vue (en particulier ils ne sont pas reconnus par le même identificateur) et pourtant on ne saurait utiliser une variable en en ignorant le type.

Il est alors nécessaire d'introduire la possibilité de déclarer des *définitions groupées*. Une idée simple consiste à proposer un constructeur de structure ("record") grâce auquel on accède aux divers aspects d'un même concept – ses «propriétés» – au travers des champs de la structure. Par exemple une variable est une structure à deux champs: le champ «nom» et le champ «type»; on accède au nom d'une variable «var» par «var.nom» et à son type par «var.type». On peut trouver à ce choix un défaut majeur qui est qu'on perd la vision uniforme des objets manipulés; on a en effet dans ce cas deux types d'objets: les opérateurs simples et les structures.

Dans l'optique d'uniformiser les concepts, on modifie alors légèrement la notion de «structure»: il s'agit toujours d'une liste de champs, la différence étant qu'un champ de la strucure ne définit pas une «zone mémoire» de la structure de donnée mais une fonction. Accéder à un champ de l'objet du type «structure» c'est alors accéder à l'une des fonctions applicables sur cet objet. Si l'on veut maintenant définir une «zone mémoire» sur cet objet, il faut définir deux champs: un champ de lecture de la «zone» et un champ de mise à jour (écriture) de cette «zone»; il faut ensuite définir un «point d'accumulation» des données qui conserve la valeur persistante du champ: ce «point» n'est pas accessible de l'utilisateur.

1.8. la référence

Comment traduire dans le LCL ce nouveau concept? On a déjà la notion de définition groupée. Par exemple:

```
def txt
  env: def x
       def y
       def z
  rep: ...
```

peut être regardé comme un texte **txt** pour lequel la liste des fonctions applicables est:

```
def x   def y   def z
```

Il suffit alors de définir un opérateur du LCL qui permette l'accès à cette liste: c'est l'opérateur ref:

- ref <nom>
  On recherche l'opérateur de nom <nom>, et si on le trouve l'évaluation fournit l'environnement des définitions locales de l'opérateur <nom>.

Dans l'exemple:

```
use x   ref txt
```

on utilise l'opérateur **x**, sachant la référence à l'opérateur **txt**: on accède donc à la représentation de la définition **def x** définie localement à l'opérateur **txt**.
On notera que dans l'exemple on ne fait aucune hypothèse sur le texte **txt**: le texte en référence de l'utilisation est lui-même nommé et est donc aussi un paramètre de l'utilisation.

2. Eléments du langage

2.1. utilisation : use

On définit un texte par sa représentation. La représentation est constituée :
- de chaînes de caractères,
- d'utilisations d'autres textes.

Exemple:

```
def var = "XCB3"
def uti = (use var) ":=0;" "^M"
```

le texte var est défini comme étant la chaîne : "XCB3";
le texte uti est défini comme :
- l'utilisation du texte var
- les chaînes ":=0;" et "^M" (retour à la ligne).

Une utilisation de var : (use var) retourne le texte : "XCB3"
Une utilisation de uti : (use uti) retourne :
"XCB3" ":=0;" "^M" ou encore "XCB3:=0;<retour à la ligne>"

2.2. modularité : environnement des définitions (def)

Pour faciliter la gestion des textes définis (def), on introduit la notion d'environnement des définitions. Un texte est défini :
- par son nom,
- par des définitions internes de textes (l'environnement des définitions),
- par une représentation.

Exemple:

```
def txt
   def var = "XCB3"
   def cst = "2.0"
= "IF " (use var) "<>0 THEN" "^M"
  (use var) ":=" (use cst) ";" "^M"
  "END IF;" "^M"
```

texte évalué	ou encore
"IF " "XCB3" "<>0 THEN" "^M"	IF XCB3<>0 THEN
"XCB3" ":=" "2.0" ";" "^M"	XCB3:=2.0;
"END IF;" "^M"	END IF;

le texte txt définit *localement* les textes var et cst, qu'il utilise ensuite dans sa représentation.
Le mécanisme d'évaluation *garantit* que les définitions internes aux textes sont placées *prioritairement* dans le contexte d'évaluation.
Dans la définition précédente de txt, on est donc assuré que les utilisations des textes var et cst font bien référence aux définitions internes du texte txt – il n'y aura aucun effet de bord indésirable à ce niveau.

2.3. référence : ref

Une référence à un texte ne s'intéresse pas à la représentation du texte nommé en référence mais à l'environnement de définition de ce texte :
- la valeur rendue par l'**utilisation** d'un texte est sa représentation évaluée;
- la valeur rendue par la **référence** à un texte est son environnement des définitions évalué.

Ce sera la seule différence – tout à fait d'importance cependant – entre utilisation (use) et référence (ref). En particulier, la recherche des textes ou la construction du contexte d'évaluation seront semblables dans les deux cas.
On remarquera qu'il s'agit très précisément de la dualité d'interprétation d'une classe d'un Langage Orienté Objet : une classe est un type instanciable (ici : par utilisation) ou un module exportant des propriétés (ici : par référence).

Par exemple:

```
def uti
   def var = "XCB3"
   def cst = "2.0"
= (use var) ":=" (use cst) ";"
```

```
(use uti)
"XCB3" ":=" "2.0" ";"

(ref uti)
env uti
   def var = "XCB3"
   def cst = "2.0"
```

le texte uti vu comme un type: (use uti) retourne une suite de chaînes de caractères;
le texte uti vu comme un module: (ref uti) retourne un environnement de définitions.

Un exemple de référence:

```
def txt
   ref uti
= "IF " (use var) "<>" (use cst) " THEN" "^M"
  (use uti) "^M"
  "END IF;" "^M"
```

```
texte évalué
"IF " "XCB3" "<>" "2.0" " THEN" "^M"
"XCB3" ":=" "2.0" ";" "^M"
"END IF;" "^M"
```

```
ou encore
IF XCB3<>2.0 THEN
XCB3:=2.0;
END IF;
```

l'environnement des définitions de txt est simplement une référence au texte uti, l'environnement des définitions de txt est donc exactement l'environnement des définitions de uti – de ce fait, le texte txt évalué a la visibilité des textes définis localement au texte uti, soit ici les textes var et cst.

2.4. environnement local

Dans la suite, on désigne par le terme **emploi** aussi bien l'utilisation d'un texte (use) que la référence à un texte (ref).
Pour faciliter l'écriture des emplois de texte, on introduit la notion d'environnement local d'emploi: c'est un environnement – des définitions ou des références – attaché à l'emploi d'un texte, et placé *en priorité* dans le contexte d'évaluation de cet emploi.

Un exemple d'utilisation de texte avec un environnement local d'utilisation:

```
def t = (use X) ":=0"
def u = (use t
           (def X = "X0"))
```

```
évaluation de u
(use X) ":=0"  sachant  (def X = "X0")
soit : "X0" ":=0"
```

2.5. résumé : la syntaxe concrète

syntaxe "informelle"

```
(1) env ::= ε | trm env
(2) trm ::= def | ref
(3)     def ::= <nom> env rep
(4)     ref ::= <nom> env
(5) rep ::= ε | atm rep
(6) atm ::= stg | use
(7)     stg ::= <string>
(8)     use ::= <nom> env
```

(1) un environnement est une liste de termes
(2) un terme est une définition def ou une référence ref
(3) une définition est: un nom <nom>, un environnement, une représentation
(4) une référence est: un nom <nom>, un environnement
(5) une représentation est une liste d'atomes
(6) un atome est une *string* stg ou une utilisation use
(7) une *string* stg est une chaîne de caractères <string>
(8) une utilisation est: un nom <nom>, un environnement

syntaxe "Lisp"

```
(1) env ::= '(' {trm} ')'
(2) trm ::= def | ref
(3)     def ::= '(' 'def' <nom> env rep ')'
(4)     ref ::= '(' 'ref' <nom> env ')'
(5) rep ::= '(' {atm} ')'
(6) atm ::= stg | use
(7)     stg ::= <string>
(8)     use ::= '(' 'use' <nom> env ')'
```

Le parenthésage permet de définir le plus sommairement possible une syntaxe qui est non ambiguë. Dans les cas d'emploi (ref ou use), on se dispensera de préfixer la liste (par 'ref' ou 'use' respectivement) s'il n'y a pas d'ambiguïté.

Notations

Pour faciliter la lecture des exemples, on adopte les notations suivantes sur les valeurs de représentation:

- on n'indique pas les guillemets pour chaque chaîne de caractères figurant dans une représentation;
- de ce fait, on distingue une *utilisation* en encadrant le symbole utilisé par des accolades;
- un environnement local d'utilisation est "récursivement" représenté selon ces règles.

Par exemple (1):

```
(def affect ()
   ((var) ":=" (exp) ";" "^M"))
```

est représenté:

```
(def affect ()
   ("{var}:={exp};"))
```

(en particulier, s'il n'y a pas nécessisté de le conserver, on oublie le retour à la ligne "^M").

Par exemple (2):

```
(def aux ()
   ((txt1 ((def X () ("0"))) " + " (txt2 ((comm))))))
```

donne:

```
(def aux ()
   ("{txt1 ((def X () ("0")))} + {txt2 ((comm))}"))
```

Chapitre 2.3:

Exemple de structuration des données

On présente ici un exemple de structuration de l'information pour la réalisation d'une petite partie d'une application – on s'intéresse ici à l'aspect *«programming in the small»*. L'exemple appartient au domaine de l'informatique de gestion, avec ses deux principales difficultés:

- le programme manipule plusieurs structures de données complexes, qui sont différentes mais qui répondent à des schémas de conception voisins;
- comme il arrive fréquemment, et c'est le cas ici, le langage utilisé est assez pauvre, et offre en particulier peu d'outils de structuration (on utilise ici le langage BAL, développé par BULL sur Micral, à mi-chemin entre BASIC et FORTRAN).

On cherche dans cet exemple à illustrer certaines fonctionnalités de l'éditeur, et en particulier la capacité d'évolution du programme construit; dans la présentation on distingue une première étape où l'on identifie le modèle générique d'une seconde où l'on doit prendre en compte de nouveaux cas.

1. Présentation générale

On veut réaliser un programme gérant diverses structures de données. Ces données se composent d'éléments, qu'on souhaite voir triés selon un certain ordre. On doit pouvoir insérer, supprimer ou modifier ces éléments.

La condition de tri peut d'emblée suggérer l'emploi d'une table d'indirection pour l'accès aux éléments triés. Par ce moyen, on évite de recopier les éléments à chaque nouveau tri (ce qui serait coûteux en temps dans le cas général), et on se ramène à modifier la table d'indirection. On introduit donc:
- un «rang physique»: ce sont les vrais numéros sous lesquels sont stockés les éléments;
- un «rang logique»: ce sont les numéros d'ordre de ces mêmes éléments triés.

Par exemple:

rang physique	*donnée*
1	"PREMIER"
2	"DEUXIEME"
5	"CINQUIEME"
8	"HUITIEME"

rang logique	*donnée*
RG(1)=5	"CINQUIEME"
RG(2)=2	"DEUXIEME"
RG(3)=8	"HUITIEME"
RG(4)=1	"PREMIER"

On utilisera généralement un tableau RG pour réaliser la table d'indirection:
RG(*rang_logique*) = *rang_physique*

Une première analyse du problème vise à extraire de l'énoncé les propriétés des objets qu'on va définir, et de cette analyse déduire la spécification formelle de ces objets. Dans le cas présent, on obtient (...) l'interface de l'objet construit:

```
TYPE  type_element : ...;

PROCEDURE  Lire_Objet        ( m : IN  integer;  x : OUT type_element );
PROCEDURE  Ecrire_Objet      ( m : IN  integer;  x : IN  type_element );
PROCEDURE  Inserer_Objet     ( m : IN  integer;  x : IN  type_element;
                               men : OUT boolean );
PROCEDURE  Supprimer_Objet   ( m : IN  integer;
                               men : OUT boolean );
PROCEDURE  Rechercher_Objet  ( x : IN  type_element;  m : OUT integer;
                               men : OUT boolean );
```

remarques:
- la variable men retourne un compte-rendu d'erreur;
- la table d'indirection RG n'est pas passée en paramètre des procédures parce qu'on la considère comme globale au module.

Il reste à réaliser toutes ces procédures définies comme propriétés du type des *éléments*.
Si l'on ne s'intéresse qu'à l'aspect méthodologique du problème, on met simplement quelques points de suspension, indiquant par là que ce n'est pas l'objet de notre propos. Il faut cependant reconnaître que la spécification formelle telle que décrite ci-dessus, si elle demande de la part du programmeur un effort de conception supplémentaire, ne représente en revanche en taille de code écrit qu'une petite partie, purement déclarative. De plus, le respect d'une cohérence d'ensemble dans les évolutions possibles de cette partie de code est en général assez bien assuré par un compilateur, puisqu'il s'agit justement de déclarations. Ceci serait en fait à moduler en fonction du choix du langage de programmation arrêté, mais on peut prétendre à l'emploi de langages ou d'outils suffisamment structurés pour réaliser dès la phase de compilation des contrôles de sémantique statique (exemple: avec le langage C, la commande *make*).
C'est donc à cette zone de points de suspension qu'on s'attachera, puisqu'elle représente la majeure partie du texte source qu'on va écrire.

Remarque:
On peut distinguer deux catégories de procédures dans l'exemple:
- *les plus simples*: ce sont Lire_Objet et Ecrire_Objet, qui travaillent uniquement sur les éléments;
- *les plus complexes*: ce sont les autres, qui doivent mettre à jour ou consulter la table d'indirection des rangs.

Dans la suite on présente rapidement les deux premières, et on s'attardera davantage sur la dernière, à savoir la procédure de recherche Rechercher_Objet.

2. Présentation détaillée

On considère les cinq procédures sur les Objets:

Lire_Objet
reçoit: **M** = le rang logique de lecture
renvoie: **X** = l'élément lu

Ecrire_Objet
reçoit: **M** = le rang logique d'écriture
X = l'élément à écrire

Inserer_Objet
reçoit: **M** = le rang logique d'insertion
X = l'élément à insérer
renvoie: **men** = C.R.
(0 si tout va bien, 1 en cas d'erreur – pas de modification)

Supprimer_Objet
reçoit: **M** = le rang logique de lecture
renvoie: **men** = C.R.

Rechercher_Objet
reçoit: **X** = l'élément recherché
XC = partie utile de l'élément utilisée pour la recherche
renvoie: **men** = C.R.
M = le rang logique de cet élément, quand celui-ci existe (men=0)
M = le rang logique d'insertion de l'élément, si celui-ci n'a pas été trouvé (men=1)

Toutes ces procédures connaissent:
RG = le rang physique des éléments logiquement triés:
RG(*rang_logique*) = *rang_physique*
LM = le nombre d'éléments existants
= le nombre d'éléments significatifs des tableaux

On considère ensuite les quatre structures de données différentes (qu'on appelle dans la suite les *contextes* d'utilisation):

contexte [1]
éléments: **X1** = chaîne de 50 caractères
X1C = partie utile de **X1** utilisée pour le tri
= les 40 premiers caractères de **X1**
rang: **RG1** = tableau
table: **fichier1**, ouvert sous le numéro logique **1**
bornes: **O < M ≤ LM1**

contexte [2]
éléments: **X2** = chaîne de 10 caractères
X2C = partie utile de **X2** utilisée pour le tri = **X2**
rang: rang habituel des entiers
table: **TX2** = tableau d'éléments de type **X2** indicé par **M**
bornes: **O < M ≤ LM2**

contexte [3]
éléments: **X3** = chaîne de 50 caractères
X3C = partie utile de **X3** utilisée pour le tri (40 car.)
X3NB = conversion en entier du premier caractère de **X3**
rang: **RG3** = tableau
table: **fichier3**, ouvert sous le numéro logique **3**
bornes: **LM3** = tableau
LM3(1) = rang -1 du premier élément **X3** commençant par le caractère '**A**'
LM3(26) = rang -1 du premier élément **X3** commençant par le caractère '**Z**'
LM3(27) = rang -1 du premier élément **X3** commençant par le caractère '**fin**'
= nombre d'éléments existants

contexte [4]
éléments: **X4** = chaîne de 50 caractères
X4C1 = partie de **X4** utilisée pour le tri (4 car.)
X4C2 = idem, si égalité sur **X4C1** (40 car.)
X4C3 = idem, si égalité sur **X4C1** et **X4C2** (6 car.)
rang: **RG4** = tableau
table: **fichier4**, ouvert sous le numéro logique **4**
bornes: **O < M ≤ LM4**

Le **contexte [1]** représente le cas général:
- les éléments sont lus dans un fichier,
- une table des rangs permet une lecture par indirection
- les bornes des rangs des éléments significatifs sont **0** et **LM**.

Pour le **contexte [2]**, la lecture est réalisée non dans un fichier mais dans un tableau.

Pour le **contexte [3]**, ce sont les bornes qui diffèrent, par l'introduction du tableau **LM3** (ceci pour permettre un accès beaucoup plus rapide à l'information contenue dans le fichier).

Pour le **contexte [4]**, c'est le critère de tri qui diffère (on introduit un ordre lexicographique sur trois composantes des éléments à comparer).

Remarque:
On utilise la notation **X**... pour le paramètre du type des *éléments*, et **Y**... pour nommer une variable locale ayant mêmes propriétés.
On trouvera en annexe le texte des quatre procédures de recherche, ainsi que leur forme «textuelle».

3. Première étape : le modèle générique

Dans cette première étape, on ne retient que les deux premiers types de structures de données: **[1]** et **[2]**. Reconnaissant, malgré la distance entre ces deux types, qu'on va devoir écrire deux fois le même "genre" de modules, on peut souhaiter paramétrer un module générique unique puis instancier chaque cas avec ses particularités.

3.1. les structures de données

On travaille toujours sur des *listes triées*. La différence qu'on observe entre **[1]** et **[2]** est que le premier utilise un fichier et le second un tableau. Cette différence pouvant vraisemblablement se reproduire, on définit en premier ces deux choix de représentation des données dans un texte commun: le texte comm.

```
(def comm
   ((def fic
| fic = fichier :                                                    |
| le rang physique RGPHY reçoit la conversion du rang logique CONV-RGLOG  |
        ((def lect ()
| lect : on recherche, dans le fichier ouvert sous le numéro logique NUM-LOG |
| et au rang physique RGPHY, la valeur stockée, retournée dans VAL        |
              ("{RGPHY}={CONV-RGLOG}
                SEARCH={NUM-LOG},{RGPHY}:{VAL}"))
         (def ecr ()
| ecr : on modifie, dans le fichier ouvert sous le numéro logique NUM-LOG et |
| au rang physique RGPHY, la valeur stockée par une nouvelle valeur VAL   |
              ("{RGPHY}={CONV-RGLOG}
                MODIF={NUM-LOG},{RGPHY}:{VAL}"))
         (def ins ()
| ins : on insère, dans le fichier ouvert sous le numéro logique NUM-LOG et à |
| un nouveau rang physique RGPHY, la valeur VAL                           |
              ("{RGPHY}={CONV-RGLOG}
                INSERT={NUM-LOG},{RGPHY}:{VAL}"))
         (def supp ()
| supp : on supprime, dans le fichier ouvert sous le numéro logique NUM-  |
| LOG la valeur stockée au rang physique RGPHY                            |
              ("{RGPHY}={CONV-RGLOG}
                DELETE={NUM-LOG},{RGPHY}"))))
    (def tab
| tab = tableau :                                                        |
        ((def lect ()
| lect : on retourne dans VAL la valeur du rang logique VAL-RGLOG        |
              ("{VAL}={VAL-RGLOG}"))
         (def ecr ()
| ecr : on modifie la valeur au rang logique VAL-RGLOG par la valeur VAL |
              ("{VAL-RGLOG}={VAL}"))))))
```

On dégage ainsi les propriétés de comm:

- (ref fic ((comm)))
  - paramètres:
    - RGPHY : rang physique
    - CONV-RGLOG : conversion du rang logique en rang physique
    - NUM-LOG : numéro logique du fichier ouvert
    - VAL : valeur du type des *éléments* (lue, écrite, insérée)

  - propriétés:
    - lect : lecture
    - ecr : écriture
    - ins : insertion
    - supp : suppression

- (ref tab ((comm)))
  - paramètres:
    - VAL-RGLOG : conversion du rang logique en une valeur
    - VAL : valeur (lue, écrite)
  - propriétés:
    - lect : lecture
    - ecr : écriture

3.2. Les contextes d'utilisation

Puisqu'on a deux types de structures, on a deux contextes d'utilisation du schéma-type comm.

Premier cas:

```
(def ctx1
   ((def NOM () ("1"))
    (def RANG
       ((def RGPHY () ("H"))
        (def CONV-RGLOG () ("RG1({RGLOG})"))
        (def NUM-LOG () ("1"))
        (ref fic ((comm)))))
    (def OBJ
       ((def val () ("X1"))
        (def aux () ("Y1"))))))
```

Deuxième cas:

```
(def ctx2
   ((def NOM () ("2"))
    (def RANG
       ((def CONV-RGLOG () ("TX2({RGLOG})"))
        (ref tab ((comm)))))
    (def OBJ
       ((def val () ("X2"))
        (def aux () ("Y2"))))))
```

Le contexte **ctx1** présente trois champs:

- **NOM** : donne un nom à ce contexte (ici: 1)
- **RANG** : définit les propriétés applicables dans le contexte, par référence au schéma des fichiers (fic)
- **OBJ** : définit les objets sur lesquels portent ces propriétés (val = variable courante, aux = variable auxillaire)

Du fait de l'instanciation partielle des paramètres de fic, depuis **ctx1** on "voit" les propriétés:

```
RANG :
   lect
     (" H=RG1({RGLOG})       ; RGPHY = "H"      CONV-RGLOG = "RG1({RGLOG})"
       SEARCH=1,H:{VAL}")    ; NUM-LOG = "1"    RGPHY = "H"
   ecr
     (" H=RG1({RGLOG})       ; idem
       MODIF=1,H:{VAL}")
   ins
     (" H=RG1({RGLOG})       ; idem
       INSERT=1,H:{VAL}")
   supp
     (" H=RG1({RGLOG})       ; idem
       DELETE=1,H")
```

On peut noter par exemple que l'instanciation de CONV-RGLOG introduit un nouveau paramètre RGLOG, et que la "non-instanciation" du paramètre VAL fait qu'on conserve ce paramètre.

De la même façon, **ctx2** présente les propriétés:

- **NOM**
- **RANG**

```
   lect
     (" {VAL}=TX2({RGLOG})")    ; VAL-RGLOG = "TX2({RGLOG})"
   ecr
     (" TX2({RGLOG})={VAL}")    ; idem
```

- **OBJ**

3.3. Les traitements simples

On peut maintenant construire les procédures génériques Lire_Objet et Ecrire_Objet:

```
(def Lire ()
  (" * Lire_Objet: M:IN integer; {val({OBJ})}:OUT type_element
    100   {lect  ((RANG)
                  (def RGLOG () ("M"))
                  (def VAL    () ({val({OBJ})})))}
          RETURN"))

(def Ecrire ()
  (" * Ecrire_Objet: M:IN integer; {val ({OBJ})}:IN type_element
    110   {ecr   ( (RANG)
                  (def RGLOG () ("M"))
                  (def VAL    () ({val({OBJ})})))}
          RETURN"))
```

Lire et Ecrire sont paramétrés par:

- un texte RANG, qui doit présenter respectivement les propriétés lect et ecr, lesquelles sont elles-mêmes paramétrées par un texte RGLOG et un texte VAL;
- un texte OBJ, qui doit présenter la propriété val.

Le choix des noms de textes permet de facilement générer le texte final:

Cas [1]

```
(Lire   ((ctx1)))
(Ecrire   ((ctx1)))
```

Cas [2]

```
(Lire   ((ctx2)))
(Ecrire   ((ctx2)))
```

Dans les deux cas, on utilise le schéma Lire ou Ecrire en fournissant le *contexte* de décompilation, **ctx1** ou **ctx2**.
On obtient (en annexe, on donne les étapes de cette évaluation):

Cas [1]

```
* Lire_Objet: M:IN integer; X1:OUT type_element
100   H=RG1(M)
      SEARCH=1,H:X1
      RETURN

* Ecrire_Objet: M:IN integer; X1:IN type_element
110   H=RG1(M)
      MODIF=1,H:X1
      RETURN
```

Cas [2]

```
* Lire_Objet: M:IN integer; X2:OUT type_element
100   X2=TX2(M)
      RETURN

* Ecrire_Objet: M:IN integer; X2:IN type_element
110   TX2(M)=X2
      RETURN
```

3.4. les traitements complexes

Il s'agit de définir les trois autres procédures: Inserer_Objet, Supprimer_Objet et Rechercher_Objet. Pour ne pas allonger démesurément l'exemple, on ne traitera ici que la dernière, Rechercher_Objet.
Le traitement étant plus complexe, on introduit un intermédiaire dans la définition du texte: le texte schema, qui donne le schéma de décompilation, paramétré localement par des textes:

```
(def Rechercher
   ( (def schema
       ...représentation...)
     ...paramètres  locaux...))
```

Pour *utiliser* le texte, il faut alors accéder à la valeur schema du texte Rechercher:

```
(schema ((Rechercher)))
```

Le schéma

Le schéma de recherche est le suivant (sans s'effrayer de la densité de GOTO qui apparaissent):

```
(def schema
  ((def fin ()
     ("MEN={CR}
       RETURN")))
   (" 140   {CLE}=0
     145   IF {CLE}={borne((RANG))} GOTO 147
           {CLE}={CLE}+1
           {lect  ((RANG)
                   (def RGLOG () ("{CLE}"))
                   (def VAL   () ("{aux{(OBJ)}}")))}
           IF  {test((aux((OBJ))))}<{test((val((OBJ))))}  GOTO  145
           IF  {test((aux((OBJ))))}>{test((val((OBJ))))}  GOTO  148
           {fin  ((def  CR  ()  ("0")))}
     147   {CLE}={CLE}+1
     148   {fin  ((def  CR  ()  ("1")))}"))
```

dans schema:

- on utilise le texte local fin:
  - paramétré par CR,
  - utilisé avec, pour CR, la valeur "0" ou "1".

- les paramètres sont:
  - CLE : c'est le rang logique utilisé pour l'itération;
  - RANG : qui offre les propriétés:
    - borne : borne supérieure des clés,
    - lect : lecture, paramétrée comme précédemment par RGLOG et VAL.
  - OBJ : qui offre les propriétés:
    - val : la variable courante du type des *éléments*,
    - aux : une variable auxiliaire de ce type,
    - val et aux offrant eux-même la nouvelle propriéte: test.

Les contextes

On reprend donc légèrement la définition du *contexte* **ctx1**:

```
(def ctx1
  ((def NOM () ("1"))
   (def RANG
     ((def borne () ("LM1"))
      ...))
   (def OBJ
     ((def val
        ((def test () ("X1C")))
        ("X1"))
      (def aux
        ((def test () ("Y1C")))
        ("Y1")))
```


Pour **ctx2**, on définit:

```
(def borne () ("LM2"))
```

et la valeur de test étant, par définition, la valeur totale:

```
(def val
  ((def test () ("{val}")))
  ("X2"))
```

et de même pour **aux**.

Enfin, on reprend la définition de Rechercher:
le paramètre CLE est pris, par défaut, égal à "M":

```
(def Rechercher
  ((def schema ...)
   (def CLE () ("M"))))
```

L'utilisation du texte

En se plaçant dans le *contexte* **ctx1**:

```
(schema
  ((ctx1)
   (Rechercher)))
```

on obtient (...) le texte attendu:

```
140   M=0
145   IF M=LM1 GOTO 147
      M=M+1
      H=RG1(M)
      SEARCH=1,H:Y1
      IF Y1C<X1C GOTO 145
      IF Y1C>X1C GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```

Dans le *contexte* **ctx2**, on a l'utilisation similaire:

```
(schema
  ((ctx2)
   (Rechercher)))
```

4. Deuxième étape: les nouveaux cas

On a défini:
- des structures de données communes: fic (fichier) et tab (tableau);
- des contextes particuliers: ctx1 et ctx2;
- des modèles de décompilation: Lire, Ecrire et Rechercher.

Il s'agit maintenant de réutiliser ces modèles dans deux nouvelles situations **[3]** et **[4]**. Les procédures Lire_Objet et Ecrire_Objet étant suffisamment simples, on peut réutiliser les schémas Lire et Ecrire dans les cas **[3]** et **[4]**. On passe donc tout de suite au cas de la procédure Rechercher_Objet, modélisée par le texte Rechercher.

4.1. le troisième contexte

Le texte attendu est le suivant:

```
140   M=LM3(X3NB-64)
145   IF M=LM3(X3NB-63) GOTO 147
      M=M+1
      H=RG3(M)
      SEARCH=3,H:Y3
      IF Y3C<X3C GOTO 145
      IF Y3C>X3C GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```

Le troisième cas ressemble beaucoup au premier: la différence est qu'une recherche ne nécessite plus le balayage du fichier à partir de son origine et jusqu'à sa fin (entre "0" et "{borne({RANG})}"), mais un balayage plus fin entre deux valeurs du tableau des bornes LM3.

On dégage donc deux nouveaux concepts, qui n'apparaissaient pas dans les deux cas précédents: il s'agit des paramètres CLE-ORIGINE et CLE-MAX. Le schéma de recherche devient alors:

```
(def schema
   ((def fin ...))
   ("140   {CLE}={CLE-ORIGINE}
     145   IF {CLE}={CLE-MAX} GOTO 147
           ..."))
```

Sachant que pour les contextes **ctx1** et **ctx2** ces deux paramètres n'ont pas été identifiés, on fournit une valeur d'instanciation *par défaut* dans l'environnement de Rechercher:

```
(def Rechercher
   ((def schema ...))
    (def CLE-ORIGINE () ("0 "))
    (def CLE-MAX () ("{borne({RANG})}"))
    (def CLE () ("M"))))
```


c'est-à-dire qu'on *ne modifie pas* les contextes déjà construits **ctx1** et **ctx2** – et dans ces deux cas l'évaluation est *inchangée*.

Construction du contexte

On doit construire pour **[3]** un contexte légèrement différent:

```
(def ctx3
   ((def NOM () ("3"))
    (def RANG
       ((def borne () ("LM3({CLE-LOG})"))
        ...))
    (def OBJ
       ((def val
          ((def cle () ("X3NB"))
           (def test () ("X3C")))
          ("X3"))
        (def aux
           ...)))))
```

Les modifications sont les suivantes:
- la propriété borne du RANG de **ctx3** est paramétrée par un texte CLE-LOG;
- les objets OBJ de **ctx3** présentent une nouvelle propriété: cle, qui est la clé relative à l'objet par laquelle on accèdera à la borne.

Utilisation de la recherche

A l'utilisation, on *surcharge* les définitions de CLE-ORIGINE et CLE-MAX:

```
(schema
   ((ctx3)
    (def CLE-ORIGINE ()
       ("{borne
            ({def  CLE-LOG  ()  ({cle({val({OBJ})})}-64)})}"))
    (def CLE-MAX ()
       ("{borne
            ({def  CLE-LOG  ()  ({cle({val({OBJ})})}-63)})}"))
    (Rechercher)))
```

qui se lit:
- on se place dans le contexte **ctx3**;
- on définit les textes **CLE-ORIGINE** et **CLE-MAX**, qui prennent en compte les spécificités du cas **[3]**;
- on se réfère enfin au modèle **Rechercher**, pour lequel les paramètres instanciés par défaut reçoivent ici une valeur particulière.

Remarque:
On utilise ici une propriété bien particulière de l'évaluateur, qui est de réaliser la recherche d'un définition de texte *dans l'ordre* des déclarations: en cas de redéfinition, on a la garantie que c'est la première définition qui est choisie.

4.2. le quatrième contexte

Le texte attendu est le suivant:

```
140   M=0
145   IF M=LM4 GOTO 147
      M=M+1
      H=RG4(M)
      SEARCH=4,H:Y4
      IF Y4C1<X4C1 GOTO 145
      IF Y4C1>X4C1 GOTO 148
      IF Y4C2<X4C2 GOTO 145
      IF Y4C2>X4C2 GOTO 148
      IF Y4C3<X4C3 GOTO 145
      IF Y4C3>X4C3 GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```

Ce dernier cas reprend dans son principe les idées du précédent. La différence est qu'il faut ici *répéter* trois fois le test, qui n'apparaît qu'une fois dans le modèle initial.

Construction du contexte

Le contexte **ctx4** reflète l'idée d'un test triple:

```
(def ctx4
   ((def NOM () ("4"))
    (def RANG ...)
    (def OBJ
       ((def val
           ((def test1 () ("X4C1"))
            (def test2 () ("X4C2"))
            (def test3 () ("X4C3")))
           ("X4"))
        (def aux
            ...)))))
```

chaque valeur d'OBJ a maintenant trois propriétés de test: test1, test2 et test3.

Utilisation de la recherche

On extrait du schema de Rechercher la partie de test, qu'on appelle proc-test:

```
(def Rechercher
   ((def schema
       ((def fin ...))
       ("...
         {proc-test}
         ..."))
    ;; suite
```



```
    ;; suite
    (def proc-test ()
       ("IF {test-aux}<{test-val} GOTO 145
         IF {test-aux}>{test-val} GOTO 148
         {suite}"))
    (def test-val  ()  ("{test((val((OBJ)))))}"))
    (def test-aux  ()  ("{test((aux((OBJ)))))}"))
    (def suite () ())
    ...))
```

Le texte proc-test est paramétré par test-val, test-aux et suite, dont on donne une valeur *par défaut* qui correspond à celle qu'on avait initialement – en particulier la suite est vide: on n'attend pas, dans proc-test, d'autres instructions derrière les deux conditions.

Dans le contexte **ctx4**, on utilise alors la recherche:

```
(schema
   ((ctx4)
    (def test-val  ()  ("{test1((val((OBJ)))))}"))
    (def test-aux  ()  ("{test1((aux((OBJ)))))}"))
    (def suite ()
       ("{proc-test
          ((def test-val  ()  ("{test2((val((OBJ)))))}"))
           (def test-aux  ()  ("{test2((aux((OBJ)))))}"))
           (def suite ()
              ("{proc-test
                 ((def test-val  ()  ("{test3((val((OBJ)))))}"))
                  (def test-aux  ()  ("{test3((aux((OBJ)))))}"))
                  (def suite  ()  ()))}"))))}"))
    (Rechercher)))
```

4.3. remarque

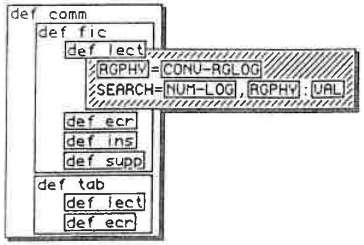
La technique de définition des valeurs par défaut des paramètres semble assez facilement automatisable: la seule contrainte est de définir un *environnement d'accueil* pour le texte qu'on utilise (ici Rechercher vis-à-vis de schema), pour y placer ces valeurs par défaut. Dans l'exemple, on a adopté la solution "riche" qui était de définir un *environnement d'accueil* spécialement dédié au texte Recherche. La solution "économique" serait de définir un *module* Bas-Niveau, dans lequel les paramètres par défaut seraient mis en commun entre toutes les propriétés du module:

```
(def Bas-Niveau
   ((def Lire ...)
    (def Ecrire ...)
    ...
    (def Rechercher ...)
    ...paramètres locaux du module Bas-Niveau...))
```

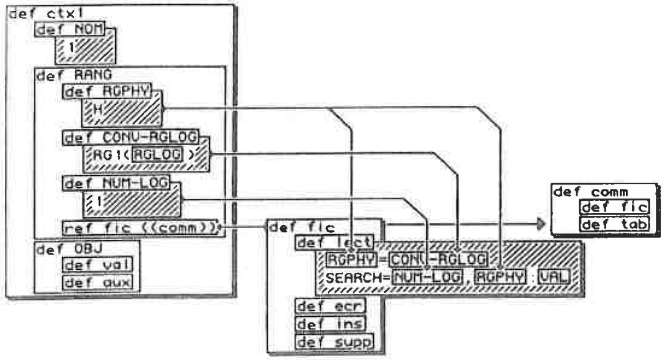
L'instanciation par défaut serait alors relative au module, et non plus à telle propriété donnée du module.

Annexe 1 : exemple d'évaluation

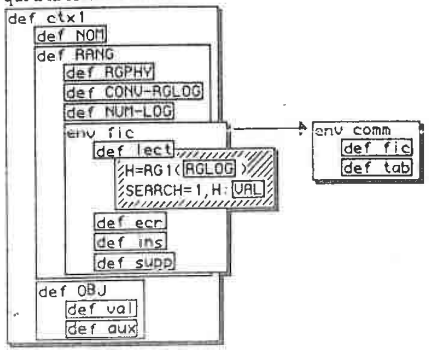
On représente ici graphiquement le texte comm, en détaillant la valeur de représentation du texte lect, défini localement à fic:



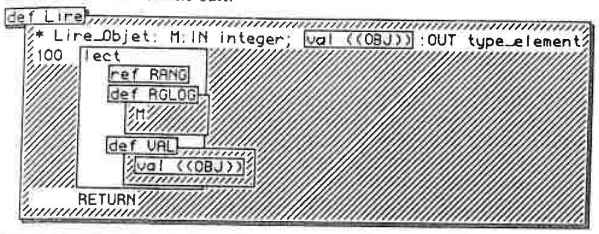
Du fait des instanciations de certains paramètres du texte lect, *depuis* le texte ctx1, on "voit" l'environnement suivant:



qui a la forme évaluée:



Le texte Lire est comme suit:

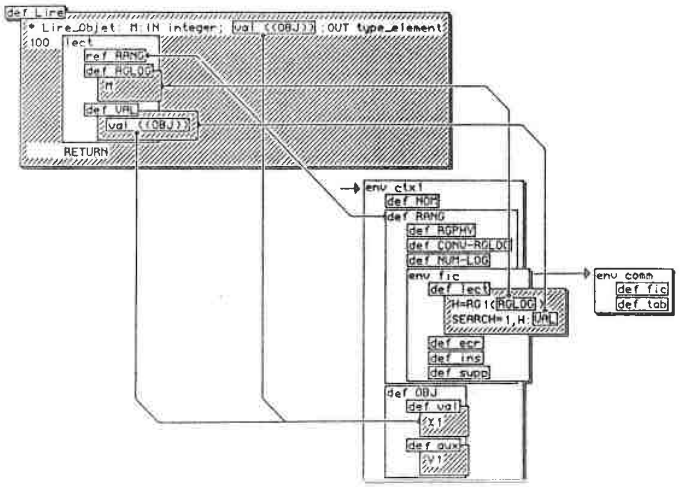


(en particulier, l'utilisation du texte lect fait intervenir un environnement local d'emploi, qui comporte la référence au texte RANG et la définition des textes RGLOG et VAL).

L'utilisation du texte Lire, *dans le contexte* ctx1, c'est-à-dire l'expression:

```
(use Lire
     ((ref  ctx1)))
```

conduit à l'évaluation suivante:



Remarques sur l'évaluation

- Les textes NOM, RANG et OBJ sont visibles tout au long de l'évaluation de la représentation de Lire.

- L'évaluation de l'utilisation du texte lect s'effectue en deux temps:
  - évaluation de la référence au texte RANG: le texte est visible puisque la totalité de l'évaluation est faite dans le contexte ctx1;
  - évaluation de l'utilisation lect: le texte est visible, du fait de la référence évaluée précédente, et les paramètres non encore instanciés de lect sont instanciés par les définitions des textes RGLOG et VAL, localement à l'utilisation.

- Le résultat retourné par l'évaluation est:

```
def Lire
  * Lire_Objet: M:IN integer; X1:OUT type_element
100  H=RG1(M)
     SEARCH=1,H:X1
     RETURN
```

Annexe 2 : les procédures de recherche

RECHERCHE 1

```
140   M=0
145   IF M=LM1 GOTO 147
      M=M+1
      H=RG1(M)
      SEARCH=1,H:Y1
      IF Y1C<X1C GOTO 145
      IF Y1C>X1C GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```

RECHERCHE 2

```
140   M=0
145   IF M=LM2 GOTO 147
      M=M+1
      Y2=TX2(M)
      IF Y2C<X2C GOTO 145
      IF Y2C>X2C GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```


RECHERCHE 3

```
140   M=LM3(X3NB-64)
145   IF M=LM3(X3NB-63) GOTO 147
      M=M+1
      H=RG3(M)
      SEARCH=3,H:Y3
      IF Y3C<X3C GOTO 145
      IF Y3C>X3C GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```

RECHERCHE 4

```
140   M=0
145   IF M=LM4 GOTO 147
      M=M+1
      H=RG4(M)
      SEARCH=4,H:Y4
      IF Y4C1<X4C1 GOTO 145
      IF Y4C1>X4C1 GOTO 148
      IF Y4C2<X4C2 GOTO 145
      IF Y4C2>X4C2 GOTO 148
      IF Y4C3<X4C3 GOTO 145
      IF Y4C3>X4C3 GOTO 148
      MEN=0
      RETURN
147   M=M+1
148   MEN=1
      RETURN
```

Annexe 3 : la représentation textuelle

Les contextes

ctx1

```
(def ctx1
   ((def NOM () ("1"))
    (def RANG
       ((def borne () ("LM1"))
        (def RGPHY () ("H"))
        (def CONV-RGLOG () ("RG1({RGLOG})"))
        (def NUM-LOG () ("1"))
        (ref fic ((comm)))))
    (def OBJ
       ((def val
          ((def test () ("X1C")))
          ("X1"))
        (def aux
          ((def test () ("Y1C")))
          ("Y1"))))))
```

ctx2

```
(def ctx2
   ((def NOM () ("2"))
    (def RANG
       ((def borne () ("LM2"))
        (def CONV-RGLOG () ("TX2({RGLOG})"))
        (ref tab ((comm)))))
    (def OBJ
       ((def val
          ((def test () ("{val}")))
          ("X2"))
        (def aux
          ((def test () ("{aux}")))
          ("Y2"))))))
```

ctx3

```
(def ctx3
   ((def NOM () ("3"))
    (def RANG
       ((def borne () ("LM3({CLE-LOG})"))
        (def RGPHY () ("H"))
        (def CONV-RGLOG () ("RG3({RGLOG})"))
        (def NUM-LOG () ("3"))
        (ref fic ((comm)))))
    ;; suite
```



```
    ;; suite
    (def OBJ
       ((def val
          ((def cle () ("X3NB"))
           (def test () ("X3C")) )
          ("X3"))
        (def aux
          ((def cle () ("Y3NB"))
           (def test () ("Y3C")) )
          ("Y3"))))))
```

ctx4

```
(def ctx4
   ((def NOM () ("4"))
    (def RANG
       ((def borne () ("LM4"))
        (def RGPHY () ("H"))
        (def CONV-RGLOG () ("RG4({RGLOG})"))
        (def NUM-LOG () ("4"))
        (ref fic ((comm)))))
    (def OBJ
       ((def val
          ((def test1 () ("X4C1"))
           (def test2 () ("X4C2"))
           (def test3 () ("X4C3")))
          ("X4"))
        (def aux
          ((def test1 () ("Y4C1"))
           (def test2 () ("Y4C2"))
           (def test3 () ("Y4C3")))
          ("Y4"))))))
```

La recherche générique

recherche

```
(def Rechercher
   ((def schema
      ((def fin ()
         ("MEN={CR}
           RETURN")))
       ("140 {CLE}={CLE-ORIGINE}
         145   IF {CLE}={CLE-MAX} GOTO 147
               {CLE}={CLE}+1
               {lect ((RANG)
                      (def RGLOG () ("{CLE}"))
                      (def VAL   () ("{aux((OBJ))}"))))}
               {proc-test}
               {fin ((def CR () ("0")))}
         147   {CLE}={CLE}+1
         148   {fin ((def CR () ("1")))}"))
    ;; suite
```

```
;; suite
(def proc-test ()
   ("IF {test-aux}<{test-val} GOTO 145
     IF {test-aux}>{test-val} GOTO 148
     {suite}"))
(def test-val  ()  ("{test((val((OBJ)))))}"))
(def test-aux  ()  ("{test((aux((OBJ)))))}"))
(def suite () ())
(def CLE-ORIGINE () ("0"))
(def CLE-MAX  () ("{borne((RANG))}"))
(def CLE  () ("M"))))
```

Les recherches contextuelles

```
rech1
    (schema
      ((ctx1)
       (Rechercher)))
rech2
    (schema
      ((ctx2)
       (Rechercher)))
rech3
    (schema
      ((ctx3)
       (def CLE-ORIGINE ()
          ("{borne
              ((def  CLE-LOG  ()  ("{cle((val((OBJ)))))-64"))))}"))
       (def CLE-MAX ()
          ("{borne
              ((def  CLE-LOG  ()  ("{cle((val((OBJ)))))-63"))))}"))
       (Rechercher)))
rech4
    (schema
      ((ctx4)
       (def test-val  ()  ("{test1((val((OBJ)))))}"))
       (def test-aux  ()  ("{test1((aux((OBJ)))))}"))
       (def suite ()
          ("{proc-test
              ((def test-val  ()  ("{test2((val((OBJ)))))}"))
               (def test-aux  ()  ("{test2((aux((OBJ)))))}"))
               (def suite ()
                  ("{proc-test
                      ((def test-val  ()  ("{test3((val((OBJ)))))}"))
                       (def test-aux  ()  ("{test3((aux((OBJ)))))}"))
                       (def suite  ()  ()))}"))}"))
       (Rechercher)))
```


Chapitre 2.4:

Exemple de structuration des traitements

On donne dans cette partie trois petits exemples de structuration des traitements d'un programme par les «textes»:

- *La pile*: on utilise le langage (pseudo-) générique LTR3; l'exemple montre combien l'expression de la généricité par les concepts du langage alourdit nettement la rédaction du programme.
- *Les lecteurs-écrivains*: l'exemple appartient au domaine du parallélisme; il montre les subtiles contraintes à respecter dans ce domaine.
- *La racine carrée*: on pose ici le problème de la modification du programme, *à sémantique constante*; cet aspect n'est pas supporté par l'outil qu'on propose.

1. La pile

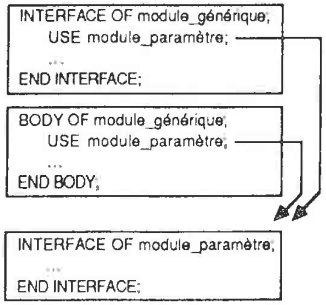
Le domaine d'intérêt des travaux touchant au Génie Logiciel, il paraît bien naturel de présenter l'exemple de la pile. On s'intéresse cependant plus ici au mode *d'implantation* de la pile qu'à sa qualité de type abstrait de données générique.

1.1. généricité en LTR3

L'exemple est présenté dans le langage LTR3, qui n'est pas générique; mais parce qu'il est modulaire, on peut assez facilement introduire une pseudo-généricité qui semble bien suffisante. La technique consiste à définir le "module générique" en le paramétrant par un module de définition des paramètres: chaque instanciation s'effectue alors en deux temps:
- construction du corps du module des paramètres;
- recopie du module générique.

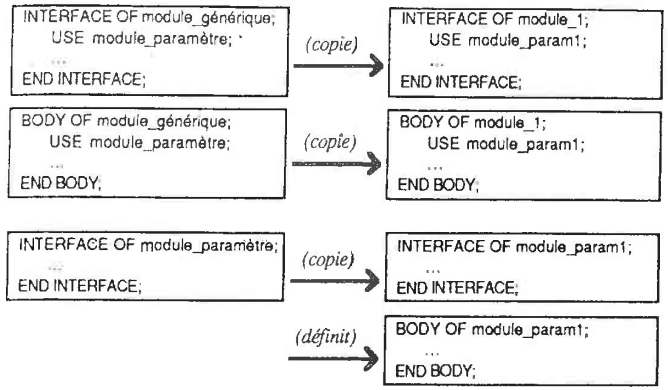
Par exemple:

Le module générique



- On définit *complètement* le module générique;
- On "spécifie" la module des paramètres en définissant son interface – l'utilisateur connaîtra donc les paramètres à fournir pour une instanciation.

Une instanciation



1.2. la forme générique de la pile

La "spécification" de l'interface

On identifie les propriétés exportées par le type pile:

```
INTERFACE OF pile_générique;
    USE pile_paramètres;
    TYPE pile: RECORD rep:pileRep; END RECORD;
    PROCEDURE Initialiser( p:INOUT pile);
    PROCEDURE Mettre( p:INOUT pile; e:IN element);
    PROCEDURE Prendre( p:INOUT pile; e:IN element);
END INTERFACE;
```

On peut remarquer en particulier que (1) le type des éléments n'est pas spécifié (ce sera un paramètre de généricité) et (2) la représentation du type pile n'est pas non plus donnée (le type est défini comme un *type dérivé* de la représentation du type pileRep, lequel sera aussi un paramètre de généricité).

La "spécification" du corps

On identifie des «traitements de bas niveau», indépendants du choix de la représentation:

```
BODY OF pile_générique;

    USE pile_paramètres;

    PROCEDURE Initialiser( p:INOUT pile);
    BEGIN
        annuler-pointeur(p.rep);
    END;

...suite
```

```
;;suite

    PROCEDURE Mettre( p:INOUT pile;  e:IN element);
    BEGIN
        incrémenter-pointeur(p.rep);
        écrire-valeur(p.rep,  e);
    END;

    PROCEDURE Prendre( p:INOUT pile;  e:IN element);
    BEGIN
        lire-valeur(p.rep,  e);
        décrémenter-pointeur(p.rep);
    END;

END BODY;
```

Entre Mettre et Prendre on observe la classique symétrie inverse dans l'ordre des «traitements de bas niveau».

La "spécification" du module des paramètres

On donne ici la déclaration du type pileRep qui doit définir la représentation de la pile et les différentes procédures dont il faudra écrire le corps à l'instanciation du module:

```
INTERFACE OF pile_paramétres;
    TYPE element;
    TYPE pileRep;
    PROCEDURE annuler-pointeur( p:INOUT pileRep);
    PROCEDURE incrémenter-pointeur( p:INOUT pileRep);
    PROCEDURE décrémenter-pointeur( p:INOUT pileRep);
    PROCEDURE écrire-valeur( p:INOUT pileRep;  e:IN element);
    PROCEDURE lire-valeur( p:INOUT pileRep;  e:OUT element);
END INTERFACE;
```

On attend en plus la définition du type des éléments element.

1.3. les deux instanciations.

Dans la première instanciation, on réalise la pile par l'utilisation d'un tableau et d'un indice qui sert de pointeur de pile. On réalise la seconde par une liste chaînée.

La définition du type est "logiquement" rattachée au corps du module mais "réellement" donnée dans son interface – le langage l'impose pour simplifier le travail du compilateur.

1ère instanciation

```
TYPE pileRep:
    RECORD
        a : ARRAY[1..100] OF element;
        b : integer;
    END RECORD;
```

2ème instanciation

```
TYPE pileRep:  REFERENCE listeRep;
TYPE listeRep: RECORD
                   a : element;
                   b : pileRep;
               END RECORD;
```


Le corps du module des paramètres définit les procédures d'interface:

1ère instanciation

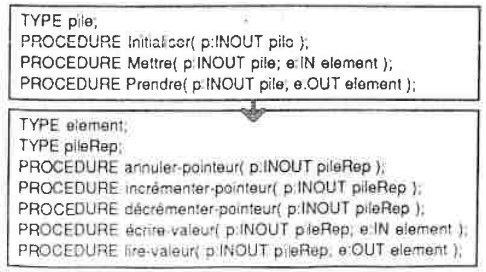
```
PROCEDURE annuler-pointeur
                ( p:INOUT pileRep);
BEGIN
    p.b := 0;
END;

PROCEDURE incrémenter-pointeur
                ( p:INOUT pileRep);
BEGIN
    p.b := p.b + 1;
END;

PROCEDURE décrémenter-pointeur
                ( p:INOUT pileRep);
BEGIN
    p.b := p.b - 1;
END;

PROCEDURE écrire-valeur
        ( p:INOUT pileRep;  e:IN element);
BEGIN
    p.a [p.b] := e;
END;

PROCEDURE lire-valeur
        ( p:INOUT pileRep;  e:OUT element);
BEGIN
    e := p.a [p.b];
END;
```

2ème instanciation

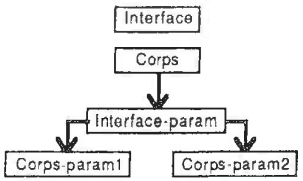
```
PROCEDURE annuler-pointeur
                ( p:INOUT pileRep);
BEGIN
    p := nil;
END;

PROCEDURE incrémenter-pointeur
                ( p: INOUT pileRep);
VAR  q:pileRep;
BEGIN
    NEW q;
    q.b := p;
    p := q;
END;

PROCEDURE décrémenter-pointeur
                ( p:INOUT pileRep);
BEGIN
    p := p.b;
END;

PROCEDURE écrire-valeur
        ( p:INOUT pileRep;  e:IN element);
BEGIN
    p.a := e;
END;

PROCEDURE lire-valeur
        ( p:INOUT pileRep;  e:OUT element);
BEGIN
    e := p.a;
END;
```

1.4. conclusion

La paramétisation de la pile par le choix de représentation conduit à une décomposition du module pile à deux niveaux:



Si on néglige la recopie des modules génériques dans le cas de LTR3, on a de toute manière à manipuler cinq unités de programme:



Au regard de la très faible complexité du problème, on peut juger cette décomposition insatisfaisante: on a de fait beaucoup compliqué un problème très simple – sans parler du temps d'exécution supplémentaire qu'introduit cette décomposition, par de nombreux empilements d'appels de procédures.

L'idée donc est de maintenir le décomposition à deux niveaux, mais de ne pas utiliser le langage, parce que les concepts qu'il propose sont d'un trop haut niveau par rapport à la dimension du problème auquel on s'attache.
Ceci a deux avantages:

- On maintient la définition des «traitements de bas niveau», ce qui permet:
  - de distinguer ce qui est du domaine de l'algorithmique de la pile de ce qui est spécifique au choix d'implantation;
  - de conserver une proximité entre des définitions voisines – par exemple, entre incrémenter-pointeur et décrémenter-pointeur, on a logiquement une grande ressemblance, dont on peut vérifier *de visu* qu'elle se retrouve dans le code si ces définitions apparaissent "à proximité" l'une de l'autre.
- On offre à l'utilisateur un "environnement de travail" plus accueillant: au lieu d'aller chercher dans des zones séparées du programme les définitions qui l'intéressent, il a une vue *complète* du module instancié; il pourra toujours le regarder pour sa structure, mais il peut aussi le voir dans son unité.

1.5. la représentation «textuelle»

La forme générique

Elle donne la structure des algorithmes qui réalisent les procédures de la pile.

```
BODY OF pile_générique;
    PROCEDURE Initialiser( p:INOUT pile);
    BEGIN
        {annuler-pointeur}
    END;
    PROCEDURE Mettre( p:INOUT pile;  e:IN element);
    BEGIN
        {incrémenter-pointeur}
        {écrire-valeur}
    END;
    PROCEDURE Prendre( p:INOUT pile,  e:IN element);
    BEGIN
        {lire-valeur}
        {décrémenter-pointeur}
    END;
END BODY;
```


Les instanciations «textuelles»

1ère instanciation

```
(def annuler-pointeur ()
    ("p.b := 0;"))
(def incrémenter-pointeur ()
    ("p.b := p.b + 1;"))

(def décrémenter-pointeur ()
    ("p.b := p.b - 1;"))
(def écrire-valeur ()
    ("p.a [p.b] := e;"))
(def lire-valeur ()
    ("e := p.a [p.b];"))
```

2ème instanciation

```
(def annuler-pointeur ()
    ("p := nil;"))
(def incrémenter-pointeur ()
    (("VAR  q:pile;")
     "NEW q;
      q.b := p;
      p := q;"))
(def décrémenter-pointeur ()
    ("p := p.b;"))
(def écrire-valeur ()
    ("p.a := e;"))
(def lire-valeur ()
    ("e := p.a;"))
```

Les formes «visuelles»

1ère instanciation

```
BODY OF pile_1;

    PROCEDURE Initialiser( p:INOUT pile);
    BEGIN
      p.b := 0;
    END;

    PROCEDURE Mettre
          ( p:INOUT pile;  e:IN element);
    BEGIN
      p.b := p.b + 1 ;
      p.a [p.b] := e;
    END;

    PROCEDURE Prendre
          ( p:INOUT pile;  e:OUT element);
    BEGIN
      e := p.a [p.b];
      p.b := p.b - 1 ;
    END;

END BODY;
```

2ème instanciation

```
BODY OF pile_2;

    PROCEDURE Initialiser( p:INOUT pile);
    BEGIN
      p := nil;
    END;

    PROCEDURE Mettre
          ( p:INOUT pile;  e:IN element);
    VAR  q:pile;
    BEGIN
      NEW q;
      q.b := p;
      p := q;
      p.a := e;
    END;

    PROCEDURE Prendre
          ( p:INOUT pile;  e:OUT element);
    BEGIN
      e := p.a;
      p := p.b;
    END;

END BODY;
```

L'instanciation est réalisée par le choix, sous l'éditeur, de l'une des deux implantations proposées: on choisit donc d'évaluer la forme générique précédente dans un contexte qui contient les définitions des textes annuler-pointeur, incrémenter-pointeur, décrémenter-pointeur, écrire-valeur et lire-valeur, celles de gauche ou de droite.

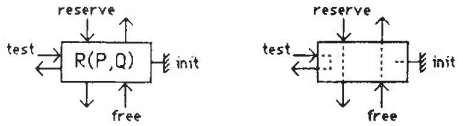
2. Les lecteurs-écrivains

Après «l'exemple d'école» des Types Abstraits de données: la pile, on présente «l'exemple d'école» du parallélisme: il s'agit de l'exemple des lecteurs-écrivains, qu'on traite ici avec une priorité égale pour les lecteurs et les écrivains.

2.1. la spécification du problème

L'exemple appartenant au domaine du parallélisme, on utilisera les réseaux de Petri pour le spécifier.

Pour construire le réseau des lecteurs-écrivains on utilise la *ressource double*:



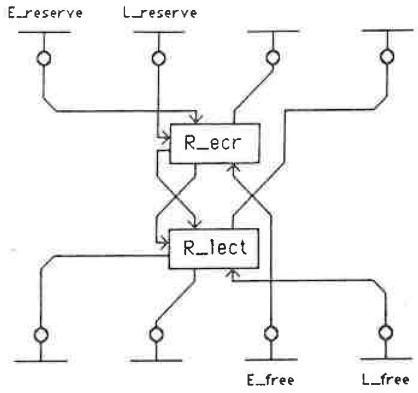
ressource double *flux de contrôle*

ressource double **R(P,Q)**, **Q≤P**:
reserve : réserve un point d'entrée de la ressource: au maximum **P** réservations;
free : libère un point d'entrée de la ressource: au minimum **0** réservation;
test : teste si le nombre de points d'entrée réservés est inférieur strictement à **Q**;
init : initialise la ressource.

Une tâche est bloquée:
- sur **reserve** : quand **P** points d'entrée sont occupés;
- sur **free** : quand aucun point d'entrée n'est occupé;
- sur **test** : quand au moins **Q** points d'entrée (**Q≤P**) sont occupés.

Les lecteurs-écrivains

On utilise deux ressources doubles:
R_ecr = R(1,1) *la ressource des écrivains*: les écrivains sont en exclusion mutuelle; les écrivains sont en exclusion avec les lecteurs.
R_lect = R(∞,1) *la ressource des lecteurs*: les lecteurs peuvent travailler simultanément; les lecteurs sont en exclusion avec les écrivains.



E_reserve (réservation d'un écrivain)
- réserve la ressource d'écriture,
- teste s'il n'y a pas de lecteur, testé dans la ressource de lecture.

L_reserve (réservation d'un lecteur)
- teste s'il n'y a pas d'écrivain, testé dans la ressource d'écriture,
- signale sa présence, par une réservation de la ressource de lecture.

E_free (libération d'un écrivain)
- libère la ressource d'écriture.

L_free (libération d'un lecteur)
- signale son départ, par une libération de la ressource d'écriture.

2.2. la réalisation du problème

Le schéma qui spécifie le comportement du module des lecteurs-écrivains peut se construire par des traitements séquentiels, la synchronisation se faisant par des objets communs – ici les ressources doubles R_ecr et R_lect. Le module peut donc être réalisé assez simplement:

Interface des lecteurs-écrivains

```
INTERFACE OF lecteurs-écrivains;

    PROCEDURE Initialiser;
    PROCEDURE E_reserve;
    PROCEDURE L_reserve;
    PROCEDURE E_free;
    PROCEDURE L_free;

END INTERFACE;
```

Corps des lecteurs-écrivains

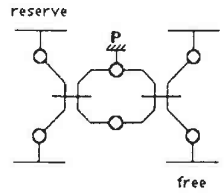
```
BODY OF lecteurs-écrivains;
    USE ressources-doubles;
    VAR  R_ecr, R_lect : ressource-double;

    PROCEDURE Initialiser;
    BEGIN
        Initialiser( R_ecr, 1, 1 );
        Initialiser( R_lect, 10000, 1 );  % 10000 = ∞
    END;

    PROCEDURE E_reserve;
    BEGIN
        reserve(R_ecr);
        test(R_lect);
    END;

    PROCEDURE L_reserve;
    BEGIN
        test(R_ecr);
        reserve(R_lect);
    END;

    PROCEDURE E_free;
    BEGIN
        free(R_ecr);
    END;

    PROCEDURE L_free;
    BEGIN
        free(R_lect);
    END;

END BODY;
```

Interface des ressources-doubles

```
INTERFACE OF ressources-doubles;

    TYPE  ressource-double;
    PROCEDURE Initialiser(R:INOUT ressource-double;
                          max_reserve,max_free:IN  integer);
    PROCEDURE reserve( R:INOUT ressource-double);
    PROCEDURE test( R:INOUT ressource-double);
    PROCEDURE free( R:INOUT ressource-double);

END INTERFACE;
```

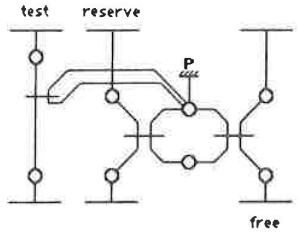

2.3. la spécification des ressources doubles

La ressource
Une ressource (simple) se représente par le réseau:

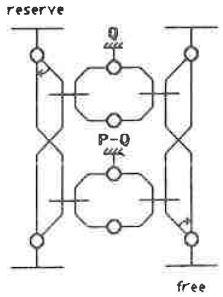


A l'initialisation, on a **P** jetons sur la place marquée **P**.

Le test sur la ressource
Le test "emprunte" un jeton dans la réserve de la ressource: il doit y en avoir un, et quand il y en a un il est replacé aussitôt après qu'on l'a pris.

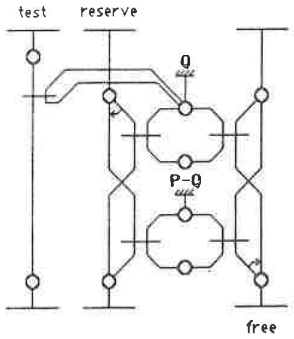


La ressource double
Dans la ressource double on partage la réserve des jetons de la ressource entre deux places:



Les flèches donnent l'ordre de priorité de déclenchement des transitions – dans les deux cas où il pourrait y avoir un choix. Par reserve, on vide d'abord la réserve des Q jetons, puis celle des P-Q jetons restants. Par free, on réapprovisionne d'abord cette dernière, puis la première.

Le test sur la ressource double
Ainsi, la place initialement marquée **Q** est vide *si et seulement si* au moins **Q** réservations ont eu lieu sans libérations correspondantes.



Le test porte uniquement sur la place marquée **Q**: la tâche qui teste la ressource est donc bloquée dès que **Q** points d'entrée de la ressource sont occupés.

2.4. la réalisation des ressources doubles

La réalisation des ressources demande de définir le type ressource-double et les procédures reserve, test et free.

Le type

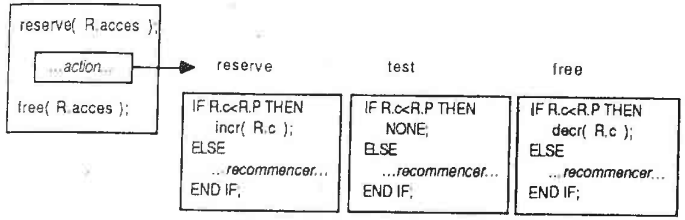
Un aspect du problème, qui n'est pas apparu sur le réseau de Petri, est l'accès concurrent des procédures aux «variables d'état» de la ressource. Dans un réseau de Petri, le déclenchement d'une transition est *«instantané»*: la mise à jour du nombre de jetons dans les places concernées est indivisible; dans une implantation, chaque étape de la mise à jour est différenciée – on a *plusieurs* instructions – et la mise à jour complète peut être interrompue – du moins si le moniteur est préemptif. Pour réaliser cette indivisibilité du déclenchement on utilise donc une ressource simple, à un point d'entrée unique.

```
TYPE ressource: RECORD
                    P,Q : integer;
                    c : integer;
                    acces : resource(1);
                END RECORD;
```

- P,Q sont les paramètres des ressources précédents;
- c va être le nombre de points d'entrée occupés (initialement c=0);
- acces est une ressource simple (type resource) à un point d'entrée qui permet l'accès exclusif aux autres champs de la structure.

Les procédures

Les procédures ont alors toutes un schéma de construction voisin: pour une ressource double R, on a:



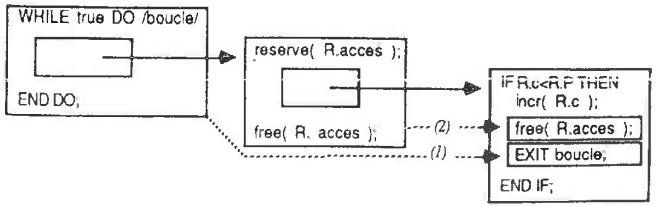
- à gauche: on réalise l'accès exclusif,
- à droite: on a les traitements spécifiques des trois procédures.

On pourrait être *"tenté"* de réaliser *...recommencer...* en "entourant" le traitement d'une boucle infinie, qui teste la condition jusqu'à ce qu'elle devienne vraie.
Par exemple:

```
reserve
WHILE true DO /boucle/
   IF R.c<R.P THEN
      incr( R.c );
      EXIT boucle;
   END IF;
END DO;
```

On voit tout de suite que la solution ne convient pas, puisqu'on est à l'intérieur du bloc d'accès exclusif de la ressource et donc qu'aucune tâche ne pourra jamais modifier les valeurs des champs de la ressource, ce qui rendrait éventuellement la condition vraie.

D'une façon naturelle, on est alors conduit à placer la boucle infinie "au-dessus" du bloc d'accès exclusif. A chaque itération, on libère puis on réserve à nouveau la ressource d'accès exclusif; une autre tâche pourra donc se glisser dans l'intervalle (on réalise ici une attente active: pour un fonctionnement correct il faut supposer que la tâche n'est pas *super-prioritaire*, parce qu'elle ne serait alors jamais interrompue):

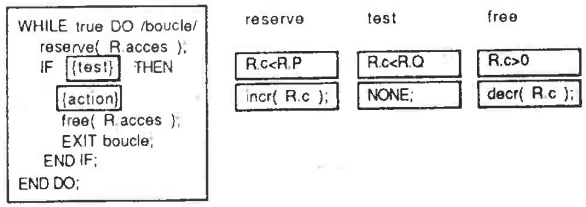


(1) Dans le *contexte* de la boucle /boucle/, on peut quitter la boucle par une phrase EXIT.
(2) Dans le *contexte* du bloc de réservation, une «sortie brutale» (phrases EXIT ou RETURN) est précédée d'une libération de la ressource.

La représentation «textuelle»

A un niveau de détail fin, on pourrait gérer le *contexte* dans lequel s'effectue la «sortie brutale» d'une boucle: cela nécessite inévitablement de travailler sur les arbres syntaxiques du langage, pour connaître la sémantique des "boûts de code" qu'on manipule (cf. Chapitre 3.3, *«L'édition syntaxique»*).
A un niveau plus grossier d'analyse du programme, on ne peut pas déduire des contraintes d'ordre contextuel; en revanche on peut offrir à l'utilisateur le moyen de les exprimer. Cela signifie qu'on ne propose pas un ***mécanisme de déduction*** qui construise ou complète le programme mais plutôt un ***mécanisme de déclaration*** qui demande à l'utilisateur de réaliser lui-même les déductions mais qui permet aussi d'en conserver la trace.

La boucle qu'on retrouve dans tous les cas permet de décomposer la vue du corps d'une procédure en deux parties:
– la partie "parallèle": on gère l'accès exclusif;
– la partie "algorithmique": on y place les traitements réalisés, quand on est déchargé des aspects relatifs au parallélisme.



Conclusion

L'exemple est simple; il présente une petite difficulté qu'on résoud en prenant soin, à la rédaction du programme, de ne pas oublier le contexte dans lequel on est placé – ici dans une zone d'exclusion mutuelle.
La représentation «textuelle» fournit *in extenso* le texte source qu'on est assez naturellement conduit à construire. Cependant elle permet aussi de fixer, dans la Représentation Interne du texte source, la contrainte contextuelle qu'on a respectée; à la relecture du programme, et à plus forte raison à sa correction on se rappellera cette contrainte, parce qu'elle est contenue dans le support même de l'information.

Corps des ressources-doubles

```
BODY OF ressources-doubles;

    PROCEDURE initialiser(R:INOUT ressource-double;
                          max_reserve,max_free:IN  integer);
    BEGIN
        R.P  := max_reserve;
        R.Q  := max_free;
        R.c  := 0;
        free( R.acces );
    END;

    PROCEDURE reserve( R:INOUT ressource-double);
    BEGIN
        WHILE true DO /boucle/
            reserve( R.acces );
            IF R.C<R.P THEN
                incr( R.c );
                free( R.acces );
                EXIT boucle;
            END IF;
            free( R.acces );
        END DO;
    END;

    PROCEDURE test( R:INOUT ressource-double);
    BEGIN
        WHILE true DO /boucle/
            reserve( R.acces );
            IF R.C<R.Q THEN
                NONE;
                free( R.acces );
                EXIT boucle;
            END IF;
            free( R.acces );
        END DO;
    END;

    PROCEDURE free( R:INOUT ressource-double);
    BEGIN
        WHILE true DO /boucle/
            reserve( R.acces );
            IF R.C>0 THEN
                decr( R.c );
                free( R.acces );
                EXIT boucle;
            END IF;
            free( R.acces );
        END DO;
    END;

END BODY;
```

3. La racine carrée

L'exemple peut être vu selon deux jours:
- on construit progressivement le programme, en utilisant des unités prédéfinies de bibliothèque;
- on modifie le programme construit: la question est de savoir si la sémantique est conservée après les modifications.

3.1 construction progressive

Le langage *cible* est le langage Lisp.
On veut définir la fonction sqrt qui calcule la racine carrée d'un entier par la méthode classique: on construit une suite (a_n) telle que:

$$a_{n+1} = 1/2\ (\ a_n + N/a_n\ )$$

(qui calcule la racine carrée de N).

On utilise d'abord le schéma des fonctions Lisp, le texte programme:

```
(def progamme ()
     ("(de {NOM} ({PARAM})
          {CORPS})"))
```

Le texte programme est paramétré par:
- NOM : le nom de la fonction,
- PARAM : le paramètre de la fonction,
- CORPS : le nom de la fonction.

Pour construire son programme, on définit donc un nouveau texte, PGME, qui utilise le schéma prédéfini de déclaration d'une fonction programme. On donne dans la suite pour les premières étapes de construction les deux formes:
- textuelle: les textes qu'on définit;
- visuelle: ce qu'on voit (à l'écran).

La construction

forme textuelle

```
(def PGME
     ()
     ("{programme}"))
```

forme visuelle

```
(de {NOM} ({PARAM})
    {CORPS})
```

On définit localement dans le texte construit PGME le nom et le paramètre de la fonction:

```
(def PGME
     ((def NOM () ("sqrt"))
      (def PARAM () ("num")))
     ("{programme}"))
```

```
(de sqrt (num)
    {CORPS})
```


On développe ensuite le corps, en utilisant le schéma prédéfini d'itération par approximations successives succ-approx, paramétré par:
- INITIALISATION : la valeur d'initialisation du résultat,
- TEST : la tolérance du calcul,
- APPROXIMATION : le calcul d'approximation.

```
(def PGME
     ((def NOM () ("sqrt"))
      (def PARAM () ("num"))
      (def CORPS
           ()
           ("{succ-approx}")))
     ("{programme}"))
```

```
(de sqrt (num)
    (prog (result)
          (setq result {INITIALISATION})
      LP  (cond ({TEST} (return result)))
          (setq result {APPROXIMATION})
          (go LP)))
```

On développe ensuite dans le corps chaque paramètre:

- *(1)* INITIALISATION : égale 1

- *(2)* TEST : on utilise «l'égalité à epsilon près»:

```
(def egal-epsilon ()
     ("(< (abs (- {ARG1} {ARG2})) {EPSILON})"))
```

avec les paramètres:
- ARG1 = "(* result result)"
- ARG2 = "num"
- EPSILON = "0.001"

On construit donc le texte:

```
(def TEST
     ((def ARG1 () ("(* result result)"))
      (def ARG2 () ("num"))
      (def EPSILON () ("0.001")))
     ("{egal-epsilon}"))
```

qui donne:

```
(< (abs (- (* result result) num)) 0.001)
```

- *(3)* APPROXIMATION : on utilise le schéma de la moyenne

```
(def moyenne ()
     ("(/ (+ {ARG1} {ARG2}) 2)"))
```

avec les paramètres:
- ARG1 = "result"
- ARG2 = "(/ num result)"

On construit alors:

```
(def APPROXIMATION
     ((def ARG1 () ("result"))
      (def ARG2 () ("(/ num result)")))
     ("{moyenne}"))
```

qui donne:

```
(/ (+ result (/ num result)) 2)
```

On donne à la fin du paragraphe une vue synthétique de la bibliothèque des unités prédéfinies et du programme construit.

Conclusion

La construction progressive présente deux avantages:

- Dans une démarche descendante, elle facilite **l'écriture du programme**. L'utilisateur en effet n'a pas à «réinventer la science» chaque fois qu'il écrit un programme: il *réutilise* un élément prédéfini de la bibliothèque. On gagne donc sur le plan de l'effort de développement demandé au programmeur; on gagne surtout en fiabilité du programme – les éléments prédéfinis peuvent avoir été dûment validés, alors que le programme construit ne le sera vraisemblablement pas. Par exemple, si l'on regarde le texte succ-approx d'approximation successive:
  - la définition des paramètres permet d'*isoler* les éléments dont on se servira pour prouver la terminaison de la boucle;
  - plus modestement le schéma aide le programmeur; sans ce schéma il aurait pu oublier par exemple d'initialiser la variable de boucle: avec le texte prédéfini qui lui est rendu il est nécessairement porté à s'interroger sur la valeur à donner à l'initialisation.

- Dans une démarche ascendante, elle facilite **la relecture du programme**. Si l'on s'éloigne de la «situation idéale» présentée précédemment où tous les besoins exprimés sont satisfaits dans la bibliothèque, on peut néanmoins offrir à l'utilisateur la possibilité de se replacer dans cette situation. Il s'agira alors pour lui:
  - d'écrire son programme,
  - puis de définir les "bouts de programme" qu'il reconnaît comme formant des "bouts d'algorithme" qui gagnent à être identifiés.

  Si l'utilisateur est placé dans un "environnement nu" – sans bibliothèque prédéfinie – son programme n'est alors pas «le fichier de texte fic.ll» qui contient la totalité de l'information mais une collection structurée de «textes» qu'il utilise simultanément. L'intérêt d'une telle approche est de permettre à l'utilisateur d'écrire son programme (il n'est pas arrêté dans l'affinage du programme parce que telle unité serait introuvable dans la bibliothèque) tout en lui donnant le moyen d'exprimer, dans le texte source, sa démarche de conception d'une solution.

La bibliothèque

```
(def programme ()
   ("(de {NOM} ({PARAM})
       {CORPS})"))

(def succ-approx ()
   ("(prog (result)
           (setq result {INITIALISATION})
         LP (cond ({TEST} (return result)))
           (setq result {APPROXIMATION})
           (go  LP))"))

(def egal-epsilon ()
   ("(< (abs (- {ARG1} {ARG2})) {EPSILON})"))

(def moyenne ()
   ("(/ (+ {ARG1} {ARG2}) 2)"))
```

La forme «textuelle» du programme

```
(def PGME
  ((def NOM () ("sqrt"))
   (def PARAM () ("num"))
   (def CORPS
     ((def INITIALISATION () ("1"))
      (def TEST
        ((def ARG1 () ("(* result result)"))
         (def ARG2 () ("num"))
         (def EPSILON () ("0.001")))
        ({egal-epsilon}))
      (def APPROXIMATION
        ((def ARG1 () ("result"))
         (def ARG2 () ("(/ num result)")))
        ("{moyenne}")))
     ("{succ-approx}")))
  ("{programme}"))
```

La forme «visuelle» du programme

```
(de sqrt (num)
  (prog (result)
        (setq result 1)
    LP (cond ((< (abs (- (* result result) num)) 0.001) (return result)))
       (setq result (/ (+ result (/ num result)) 2))
       (go  LP)))
```

3.2. la modification du programme

Dans cette seconde partie, on s'intéresse à la modification du programme. La modification d'un programme se justifie pour deux raisons:
- soit le programme était faux: on corrige l'erreur;
- soit le programme est correct, mais alors on veut le faire évoluer, pour accroître ses performances ou l'adapter à un nouveau «milieu», ou encore faire varier ses fonctionnalités.

La question qu'on doit se poser est de savoir si la modification *préserve* la sémantique du programme. Dans le premier cas, il est à souhaiter que la réponse soit non; dans le second cas, on aimerait répondre par l'affirmative, et si possible d'une façon *automatique*. L'automatisation d'un tel contrôle est un problème résolument complexe, elle n'est pas supportée par l'outil qu'on propose. Dans la suite on montre certaines difficultés qu'elle pose – sur la très simple fonction de calcul d'une racine carrée – en présentant deux outils qui pourraient en faciliter la résolution.

3.2.1. position du problème

La fonction précédemment construite est:

```
(de sqrt (num)
  (prog (result)
        (setq result 1)
     LP (cond ((< (abs (- (* result result) num)) 0.001) (return result)))
        (setq result (/ (+ result (/ num result)) 2))
        (go LP)))
```

On aimerait éviter la redondance du calcul multiplicatif – multiplication ou division – dans le test et l'approximation de l'algorithme. Pour ce faire on effectue le regroupement en deux étapes:

(1) remplacer (- (* result result) num)
par (- result (/ num result))

(2) regrouper les deux termes (/ num result)
par l'utilisation d'une variable auxiliaire X

3.2.2. la modification locale

R. Dybvig et B. Smith présentent dans [DyS 85] un *«éditeur sémantique»*. L'éditeur travaille sur une forme "lispienne" du langage FP défini par J. Backus [Bac 78]:
- il connaît la syntaxe FP – représentée par des fonctions Lisp;
- il connaît un certain nombre de règles d'équivalence entre les termes de FP.

L'outil permet la saisie d'un terme de FP, puis sa transformation, via les règles d'équivalence orientées en règles de réécriture, à *sémantique constante*.

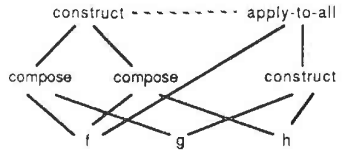
Une des originalités de l'approche est de gérer la relation d'équivalence que définissent les règles: un programme conserve dans sa Représentation Interne les classes d'équivalence des termes qui le composent. On a donc deux interactions possibles sous l'éditeur:
- soit visualiser le programme: on voit la forme décompilée correspondant au «chemin sélectionné» dans le graphe de congruence;
- soit modifier le chemin, par ajout d'un nouveau lien de congruence entre les termes ou par le choix d'un nouveau chemin du graphe construit.

Pour éclairer, le propos on a par exemple la règle d'équivalence:

```
(construct (compose f g) (compose f h))
    ≡ (apply-to-all f (construct g h))
```

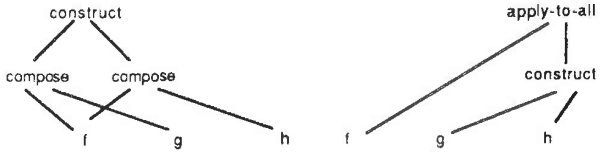
(construct, apply-to-all, compose sont des *opérateurs* du langage FP).

Le programme est alors représenté par le graphe:



La flèche représente l'instanciation classique d'un nœud d'opérateur par un terme tel qu'on le trouve dans un arbre syntaxique.
Le pointillé symbolise la relation d'équivalence entre les termes.

Un «chemin sélectionné» sera par exemple le sous-graphe (orienté):



qui est un programme *sémantiquement équivalent* au sous-graphe dont la "racine" est apply-to-all.

L'exemple

On veut remplacer le test:
(< (abs (- (* result result) num)) 0.001)
par le test:
(< (abs (- result (/ num result)) 0.001)
dont on "remarque" qu'il a une sémantique voisine.

En adoptant les notations mathématiques plus concises et en élargissant le problème, on veut remplacer:
$|r1 \times r2 - n| < \varepsilon$
par le test:
$|r1 - n/r2| < \varepsilon$

On transforme le premier terme (entre crochets on indique quelles règles d'équivalence on utilise):

```
   |r1×r2-n| < ε
= |r1×((r1×r2)/r1-n/r1)|  <  ε   [ x-y ≡ z×(x/z-y/z)  ]
= |r1×((r1/r1)×r2-n/r1)|  <  ε   [ (x×y)/z ≡ (x/z)×y ]
= |r1×(1×r2-n/r1)|        <  ε   [ x/x ≡ 1 ]
= |r1×(r2-n/r1)|          <  ε   [ 1×x ≡ x ]
= |r1|×|r2-n/r1|          <  ε   [ |x×y| ≡ |x|×|y| ]
= |r2-n/r1|               <  ε/|r1|   [ x×y<z ≡ y<z/x si x>0 ]
```

On s'arrête ici, parce qu'on ne peut plus transformer le terme dans la direction qu'on suit. Or celui-ci est *intuitivement* équivalent à:

$|r2-n/r1| < \varepsilon$

En effet dans le cadre qui nous intéresse la majoration est «paramétrée» par un ε *assez petit*, donc si l'on suppose que |r1| est minoré alors majorer par ε/|r1| ou majorer par ε est *à peu près* équivalent. Il faudrait ici appliquer une règle de réécriture *dépendante du problème* («problem-dependant»): dans l'absolu elle serait sémantiquement fausse, mais pour ce qui nous intéresse elle serait presque vraie.

Le cas pose problème: si l'on s'impose des transformations à sémantique constante, on est arrêté à cette étape; si l'on s'autorise des approximations, la relation de congruence sur les termes devient obsolète.

3.2.3. la modification globale

R. Waters présente un autre *éditeur sémantique*, «l'Apprenti du Programmeur» [Wat 82] [Wat 86]. Comme tout bon Apprenti, celui-ci se charge des «basses besognes» en laissant la «partie noble» du travail à la charge du Programmeur.

Le système fonctionne à plusieurs niveaux:

- L'interface utilisateur cherche à faciliter le travail de saisie du programmeur:
  - les commandes sont entrées en pseudo-anglais,
  - chaque commande retourne immédiatement en écho le texte du programme en Lisp.

  Le programmeur conçoit donc dans sa "langue naturelle" mais comprend ce qu'il compose dans la "langue informatique".
- Pour la construction d'un algorithme, l'utilisateur invoque des schémas prédéfinis («clichés»); pour les points de détail du programme il peut directement entrer du code Lisp.
- Le système quant à lui se place en amont du programe Lisp: il travaille sur un graphe de flot de données (data-flow) et assure la correspondance entre ce graphe, les instructions de l'utilisateur et l'écho en langage Lisp.

Par nature, le système ne garantit aucune conservation de la sémantique du programme lors d'une transformation: l'Apprenti n'est pas là pour enseigner au Programmeur les bonnes règles de la programmation mais pour bénéficier de la connaissance experte de ce dernier. Sa tâche est plutôt de contrôler que le Programmeur n'introduit pas de contradictions notoires, et tout particulièrement, pour ce qui concerne les variables:

- leur zone de visibilité (syntaxe): il a ure la déc ration des variables, ou encore sait reconnaître les variables libres d'une fonction (l'occurrence d'utilisation d'une variable globale);
- leurs zones de pertinence (sémantique): il peut reconnaître les zones du programme où une variable a une valeur pertinente ou même constante.

L'exemple

Par une *"transformation syntaxique"*, on a transformé la programme en:

```
(de sqrt (num)
  (prog (result)
        (setq result 1)
     LP (cond ((< (abs (- result (/ num result))) 0.001) (return result)))
        (setq result (/ (+ result (/ num result)) 2))
        (go LP)))
```

A l'étape suivante, on regroupe les deux divisions (/ num result) par l'emploi d'une variable locale; cette seule formulation d'«expression de besoin» par l'utilisateur fournit alors le programme:

```
(de sqrt (num)
  (prog (result X)
        (setq result 1)
     LP (setq X (/ num result))
        (cond ((< (abs (- result X)) 0.001) (return result)))
        (setq result (/ (+ result X) 2))
        (go LP)))
```

Que garantit le système?

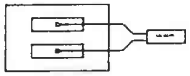
- il assure la déclaration de la variable locale X – ce qu'on peut vérifier *de visu* grâce à l'écho rendu;
- il vérifie que les deux divisions de départ se calculent bien sur les mêmes données – et donc qu'on peut bien les regrouper;
- il place l'affectation de la variable auxilliaire X "au bon endroit" – ici, à l'entrée de la boucle d'itération.

Les deux derniers points sont les deux aspects intéressants du système. Ils sont le fait de la Représentation Interne du programme par un graphe de flot de données: on n'a pas la vue séquentielle habituelle d'un langage de programmation qui rend très ardue la réalisation des vérifications présentées. Une limitation sévère du système en est la conséquence: à chaque "bout de programme algorithmiquement significatif" doit être associé un graphe de flot de données; si ce "bout" est dans la bibliothèque des objets prédéfinis on réutilise le graphe associé; autrement il faut le définir, ce qui alourdit beaucoup l'étape de programmation.

3.2.4. le graphe de flot de données

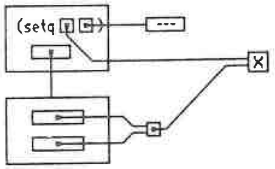
En s'affranchissant de la question des déclarations de variable, on veut résoudre le problème suivant:

On part d'un «texte troué» sur lequel on a reconnu qu'à deux points distincts on élaborait le même calcul:



Les deux «trous» sont donc remplis par le même texte qui réalise le calcul.

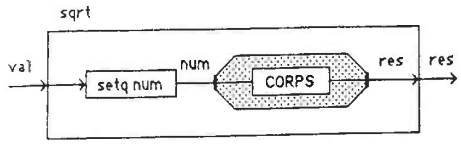
On veut savoir si l'on peut remplacer ce texte par le suivant:



- On introduit une variable auxiliaire X;
- On affecte la valeur du calcul élaboré à cette variable
- On effectue ensuite le traitement, et les élaborations du calcul sont remplacées par l'élaboration de la valeur de la variable X.

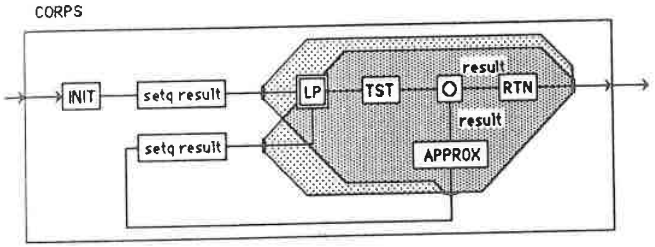
Pour réaliser une telle transformation en toute quiétude il faut donc démontrer qu'à nul point du «texte troué» on n'affecte des objets susceptibles d'affecter l'élaboration du calcul: un problème non simple.
La représentation «textuelle» qu'on propose ne sait pas répondre à la question. En revanche les graphes de flot de données semblent assez bien adaptés à la situation. L'exemple précédent se réalise par les considérations qui suivent.

La fonction sqrt, de paramètre num, n'affecte pas la valeur du paramètre: num est globalement constant dans le corps de la fonction.



: zone de valeur pertinente et constante de "num"
val : valeur du paramètre effectif à l'appel de "sqrt"
res : valeur résultat de l'appel de "sqrt"

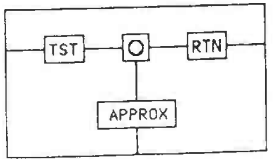
Le CORPS est construit avec le schéma prédéfini succ-approx:



: zone de valeur pertinente et constante de "result" à l'initialisation ou à l'itération de la boucle
: zone d'intersection

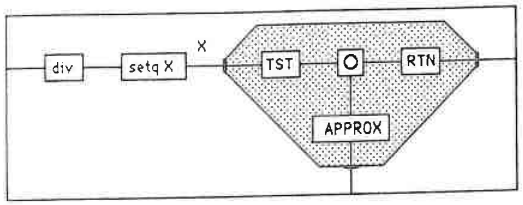
Pour le regroupement des deux divisions, on observe les points suivants:
- elles sont syntaxiquement égales, préliminaire indispensable;
- elles apparaissent dans les "boîtes" TST et APPROX;
- elles utilisent les valeurs des variables num et result.

L'intersection des «zones de valeur pertinente et constante» des variables num et result fournit la nouvelle "boîte":



qui contient justement les "boîtes" TST et APPROX.

Il suffit alors de placer l'affectation de X en tête de cette "boîte":



: zone de valeur pertinente et constante de "X"

div : la division

(et on remplace TST et APPROX dans l'élaboration de la division par l'accès à la valeur de X).

Chapitre 2.5:

Exemple de structurations connexes

On présente ici un exemple qui allie la structuration des données et la structuration des traitements. On y souligne les liens étroits qui sont décelables entre des points distincts du programme mais intuitivement voisins.

Le programme analysé se développe selon deux axes:
- l'axe des *«structures de données»*: on introduit ici trois structures de données "similaires";
- l'axe des *«traitement»*: à chaque structure de données est associée une liste de traitements.

On cherche à montrer ici qu'un certain traitement d'une certaine structure de données n'est autre chose que la forme doublement instanciée d'un modèle générique commun à tous les cas. Cette double instanciation, mal supportée par un langage de progammation, pourrait se traduire, *assez* aisément, dans l'éditeur proposé.

I. Présentation

L'exemple qu'on traite est le programme Lisp qui implante les fonctions de modification incrémentale de la syntaxe abstraite (cf Chapitre 5.5, *«Construction de la Syntaxe Abstraite»*). Deux remarques s'imposent:

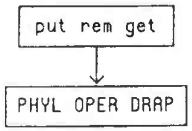
- Le choix de l'exemple n'est pas innocent: c'est une partie du programme écrit dont la rédaction a été particulièrement soignée; c'est-à-dire que l'homogénéité du programme est à la mesure de l'effort de spécification-conception préliminaire. Cette constatation n'est pas une surprise: si l'on veut obtenir des programmes *bien construits* le prix à payer est un supplément de travail non négligeable dans les premières étapes du développement – définition du problème et définition de la solution.

- Le type de problème abordé est purement algorithmique; le modèle mathématique sous-jacent est presque apparent dans le programme. Les symétries qu'on observe dans le programme sont donc à rapprocher de celles qu'on trouve très fréquemment en mathématiques.

Je ne pense pas qu'il faille en conclure que l'exemple est mal choisi: on n'a pas toujours de répétition dans le texte source d'un programme; mais quand on en a, et c'est le cas spécialement dans les parties algorithmiques, il me semble intéressant de les mettre en relief. C'est la situation dans laquelle on se place.
Pour terminer ce préambule, je voudrais juste signaler que le programme n'a pas été remanié pour étayer l'argumentation. S'il présente de nombreuses symétries c'est parce qu'elles se sont imposées d'une façon naturelle à la rédaction du programme.

1.1. présentation générale

L'analyse du problème conduit à décomposer le programme en deux niveaux:



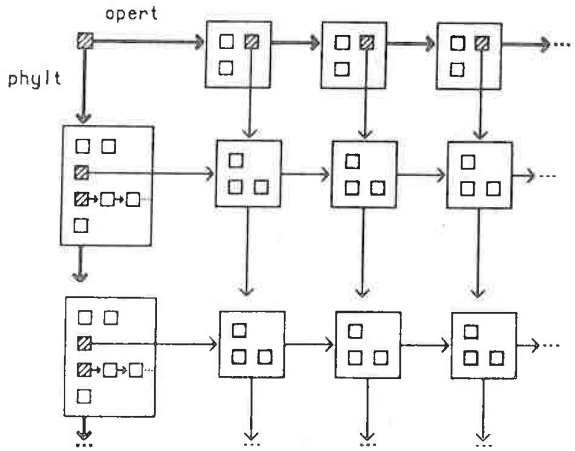
- en haut: les «fonctions utilisateur»
  put = placer
  rem = supprimer
  get = obtenir des informations
- en bas: les «types de données»
  PHYL = phylum
  OPER = opérateur
  DRAP = drapeau : phylum x opérateur

L'implantation présentée diffère légèrement de celle qu'on trouve formellement définie dans la partie Technique (cf. ibid.) : on n'introduit pas de matrice des phyla, et à chaque phylum on attache alors la liste de ses phyla pères. Ceci permet de n'avoir qu'une seule structure de données, ce qui simplifie le présentation.

Note:
Relativement au coût des algorithmes, on perd en efficacité sur la fonction ascendance: celle-ci construit l'ascendance d'un phylum donné en parcourant le graphe comme s'il s'agissait d'un arbre. Elle passera donc plusieurs fois sur les mêmes nœuds si deux chemins différents y conduisent.
Pour la *syntaxe initiale*, la seule situation où on passe deux fois sur un même nœud est celle (1) où le nœud d'origine est le phylum LSP et (2) où le nœud doublement visité est le phylum SEX. Du fait de la faible connexité du graphe, l'algorithme en pratique est donc plus efficace – quoiqu'en théorie il le soit moins.

1.2. les données

La structure de données est une *matrice de drapeaux*, dont les lignes sont les phyla et les colonnes les opérateurs:

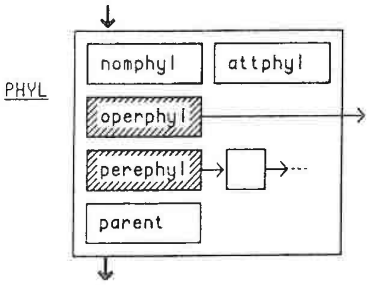


les variables globales

Elles permettent de parcourir une ligne ou une colonne de la matrice:

pyhlt : parcours de la liste des phyla
opert : parcours de la liste des opérateurs

le phylum **PHYL**



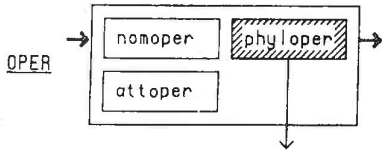
• champs du phylum
- **nomphyl** : nom du phylum
- **attphyl** : attributs du phylum
- **operphyl** : la ligne attachée au phylum de la matrice
- **perephyl** : liste des phyla pères du phylum
- **parent** : champ entier: *«degré de connexité»* du nœud "phylum" par rapport à un autre phylum donné.

• construction du phylum
makephyl (nomphyl attphyl)
: construit le phylum (nomphyl attphyl () () 0)

• propriétés du phylum
- **nomphyl** : nom du phylum
- **attphyl** : attributs du phylum
- **operphyl** : liste des drapeaux, *le long de* la liste des opérateurs
- **perephyl** : liste des phyla pères du phylum
- **parent** : test (vrai si le champ parent est non nul)
- **valparent** : valeur du champ parent
- **setparent** : incrémente de 1 le champ parent
- **resetparent** : décrémente de 1 le champ parent

Les quatre premiers champs sont aussi des propriétés du type. Le cinquième est en revanche plus complexe.

l'opérateur **OPER**



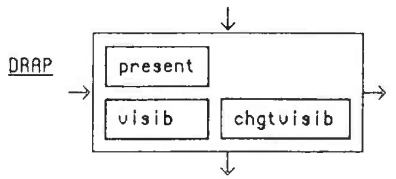
• champs de l'opérateur
- **nomoper** : nom de l'opérateur
- **attoper** : attributs de l'opérateur
- **phyloper** : la colonne attachée à l'opérateur de la matrice

• construction de l'opérateur
makeoper (nomoper attoper)
: construit l'opérateur (nomoper attoper ())

• propriétés de l'opérateur
- **nomoper** : nom de l'opérateur
- **attoper** : attributs de l'opérateur
- **phyloper** : liste des drapeaux, *le long de* la liste des phyla

Les champs sont ici identifiés aux propriétés.

le drapeau **DRAP**



• champs du drapeau
- **present** : champ booléen: *«appartenance en propre»* d'un opérateur à un phylum
- **visib** : champ entier: *«degré d'appartenance»* d'un opérateur à un phylum
- **chgtvisib** : champ booléen: on y conserve l'information «l'opérateur est nouvellement visible» ou «l'opérateur était anciennement visible» en vue de l'analyse des attributs

(ce dernier champ est introduit du fait de la séparation des traitements qui modifient la structure de la matrice de ceux qui réalisent l'analyse des attributs)

- construction du drapeau
  makedrap : construit le drapeau (() 0 ())

- propriétés du drapeau
  present : test (vrai si le champ present est vrai)
  setpresent : met le champ present à vrai
  resetpresent : met le champ present à faux
  visib : test (vrai si le champ visib est non nul)
  valvisib : valeur du champ visib
  nouvvisib : test (vrai si le champ chgtvisib est vrai)
  ancivisib : idem
  incrvisib : incrémente le champ visib d'un entier en paramètre
  + met à jour la champ chgtvisib si «l'opérateur est nouvellement visible», c'est-à-dire si le champ visib était nul avant l'appel de la fonction
  decrvisib : décrémente le champ visib d'un entier en paramètre
  + idem inversé
  resetnouvvisib : met le champ chgtvisib à faux
  resetancivisib : idem

L'accès aux champs est ici "filtré" par diverses fonctions. En particulier le champ chgtvisib est regardé relativement au traitement antérieur:
- on se demandera si «l'opérateur est nouvellement visible» (nouvvisib) après avoir incrémenté le drapeau (incrvisib);
- on se demandera si «l'opérateur était anciennement visible» (ancivisib) après avoir décrémenté le drapeau (decrvisib);

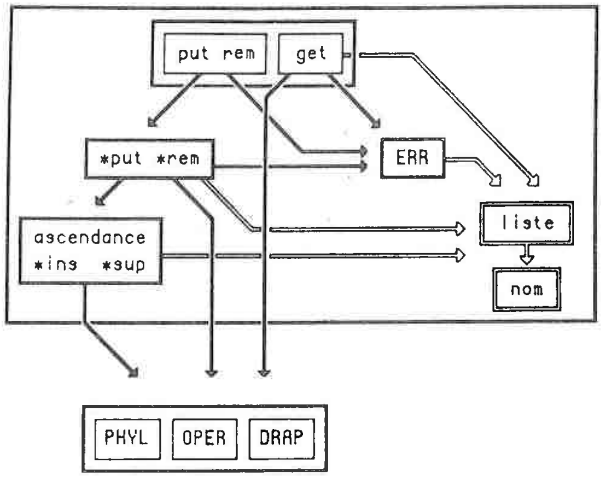
1.3. les traitements

On morcelle les traitements pour en faciliter la réalisation:

schéma page suivante

- les «fonctions utilisateur»
  put = putphyl, putoper, putoperphyl, putperephyl, putfilsphyl
  rem = remphyl, remoper, remoperphyl, remperephyl, remfilsphyl
  get = getphyl, getoper, getoperphyl, getperephyl, getfilsphyl
  Elles font le lien entre l'utilisateur qui nomme les objets (phylum ou opérateur) et les objets eux-mêmes (contrôles de définition; contrôles de non redéfinition).

- les fonctions de traitements
  ***put** = *putphyl, *putoper, *putoperphyl, *putperephyl
  ***rem** = *remphyl, *remoper, *remoperphyl, *remperephyl
  Elles réalisent les traitements sur les objets (phylum ou opérateur).



graphe d'appel

- les fonctions d'erreur
  ERR = CONTROLE-def, CONTROLE-indef, ERREUR
  Elles filtrent les cas d'erreur.

- les fonctions auxiliaires
  ascendance
  ***ins** = *insereoperphyl
  ***sup** = *supprimeoperphyl
  Elles correspondent à un «regroupement de code» – le traitement étant effectué à plusieurs points distincts du programme, on définit une fonction.

- le module générique
  liste = recherche, ins, sup
  Elles définissent la structure générique des listes:
  recherche : recherche d'un élément de la liste d'après son nom
  ins : insertion d'un élément dans la liste
  sup : suppression d'un élément de la liste
  Le module est paramétré par la fonction nom (le nom d'un élément de la liste).

2. La macro-génération

La macro-génération est une technique «vieille comme le monde», à la fois largement utilisée et largement critiquée. On l'utilise dans deux optiques:

- Dans un souci d'*efficacité*: la génération de "code en ligne" est la plus simple des techniques d'optimisation. A l'interprétation d'un programme Lisp, on aimera bien ne pas trop attendre; pour la compilation, on aimera éviter les appels incompilables à la fonction eval.
- Dans un souci d'*abstraction*: on exprime le problème dans ses termes, et on génère du code dans les termes du langage. Dans un langage comme COBOL, où l'on cherche à écrire un programme comme on écrirait un texte en anglais, la technique est presque un concept primitif du langage.

Le gros avantage de la macro-génération en est sa simplicité. Mais cette simplicité ne doit pas masquer certains dangers qui lui sont liés:

- On ne contrôle plus la taille du code généré: même si l'on tient compte de l'augmentation régulière de la puissance des ordinateurs, il faut malgré tout rester dans des limites raisonnables.
- On ne connaît pas *a priori* l'allure du code généré: la macro-génération peut construire un code totalement aberrant, ou subtilement aberrant; par exemple en C, la macro-définition:

```
#define max1(a,b) ((a<b)? (b):(a))
```

offre toutes les garanties qu'on ne retrouverait pas dans la suivante:

```
#define max2(a,b) ((a<b)? b:a)
```

les deux définitions sont pourtant syntaxiquement très voisines.

On trouve dans [KoW 87] une syntaxe pour les macro-définitions en Lisp qui cherche à en faciliter la compréhension; au lieu de proposer classiquement une fonction qui permette de *tester* sur quelques exemples la macro-génération, on propose un moyen d'expression qui fait ressembler la macro-définition à sa forme macro-générée. Dans [TrY 80] on cherche à réduire l'initiative laissée au programmeur à l'utilisation d'une macro-définition (COBOL): c'est la macro-définition syntaxique, où les paramètres effectifs doivent répondre à certaines contraintes d'ordre syntaxique qu'on impose à la définition.

2.1. les données

On a présenté précédemment les trois types abstraits de données identifiés:
- PHYL : phylum
- OPER : opérateur
- DRAP : drapeau phylum x opérateur

2.1.1. la référence uniforme

On a vu que pour un phylum:
- les quatre premiers champs sont aussi des propriétés du type,
- le cinquième est d'accès plus complexe.

Dans une *implantation efficace* du type, on voudra accéder directement aux premiers champs et utiliser des fonctions pour les propriétés du dernier. Il se pose alors la question de *référence uniforme*: on aimerait nommer de la même façon des concepts qui ont une sémantique semblable mais une implantation différente.

Par exemple:
(1) PHYL_NOMOPER
(2) NOMOPER(PHYL)
devraient s'interpréter identiquement. On utiliserait alors systématiquement la syntaxe (2), parce qu'elle permet de ne pas distinguer – sur le plan des choix d'implantation – le champ NOMOPER du champ PARENT:
NOMOPER(PHYL) deviendrait PHYL_NOMOPER,
PARENT(PHYL) resterait un appel fonctionnel.

- Dans un langage comme Alphard, la chose est vraie.

- Dans un langage comme Ada, ceci est faux; mais les *outils* développés autour du langage le rendent vrai – un optimiseur global, placé en aval du compilateur, reconnaît dans NOMOPER une "fausse fonction" qu'il génère par du code en ligne.

- Dans un langage comme Lisp, ceci sera vrai à l'initiative du programmeur: il peut définir la fonction NOMOPER par une macro-définition qui remplace alors les appels de la fonction par du code en ligne (Note: on pourrait se placer dans cette situation en Ada avec l'utilisation d'un pragma – le pragma INLINE).

- Dans d'autres langages, on utilise une définition «textuelle»: la propriété de référence uniforme, mal supportée par ces langages, est introduite en amont du compilateur (c'est-à-dire qu'on fournit au compilateur un texte macro-généré).

2.1.2. les similitudes de traitements

On s'intéresse ici aux propriétés qui définissent de "vraies fonctions".
La définition d'un Type Abstrait de données n'est jamais très éloignée de celle de Type Abstrait Algébrique (TAA): on aimerait alors exprimer dans le texte les propriétés de consistance et de complétude qu'on peut vérifier dans le formalisme des TAA.

On regarde le type des drapeaux, pour lequel aucune propriété n'est une "fausse fonction". Le type est déclaré en Ada:

```
type TYPE_DRAP is record
                    CHP_PRESENT  : BOOLEAN;
                    CHP_VISIB    : INTEGER;
                    CHP_CHGTVISIB : BOOLEAN;
                  end record;
```

Les sous-programmes qui opèrent sur le champ CHP_PRESENT sont:

```
function PRESENT (DRAP:in TYPE_DRAP) return BOOLEAN is
begin
    return DRAP.CHP_PRESENT;
end;

procedure SETPRESENT (DRAP:in TYPE_DRAP) is
begin
    DRAP.CHP_PRESENT := TRUE;
end;

procedure RESETPRESENT (DRAP:in TYPE_DRAP) is
begin
    DRAP.CHP_PRESENT := FALSE;
end;
```

Les grandes similitudes entre ces trois sous-programmes sont la traduction dans le langage de l'effort du programmeur visant à obtenir la consistance et la complétude dans la définition du type. Un fois encore on ne va pas présenter un moyen de s'assurer ces propriétés mais seulement de les exprimer et de les conserver dans le programme.

Au vu des trois sous-programmes, on "devine" les étapes de construction.
Par exemple, la procedure SETPRESENT se construit:

-- schéma général des sous-programmes

```
[][(DRAP:[] TYPE_DRAP)[] is
begin
    []
end;
```

-- particularisation des procédures

```
procedure [] (DRAP:[in out] TYPE_DRAP)[] is
begin
    DRAP.CHP_PRESENT := [];
end;
```

-- particularisation de la procédure SETPRESENT

```
procedure SETPRESENT (DRAP:in out TYPE_DRAP) is
begin
    DRAP.CHP_PRESENT := TRUE;
end;
```

On donne en annexe l'arbre de construction des sous-programmes relatifs au champ CHP_PRESENT ainsi que sa représentation «textuelle».

On pourrait définir des «textes» à un niveaux de détail plus fin:
- Partant du type TYPE_DRAP et de l'objet DRAP, on déduit les propriétés applicables – dont celle qui nous intéresse ici: l'accès au champ CHP_PRESENT.
- La déclaration de la variable DRAP, paramètre des sous-programmes, est générée automatiquement.
- L'accès au champ CHP_PRESENT détermine le type de l'expression:
  - retournée par la fonction PRESENT: type BOOLEAN,
  - affectée dans les procédures SETPRESENT et RESETPRESENT: les constantes TRUE et FALSE respectivement.

On reconnaîtrait dans les autres propriétés du type TYPE_DRAP les mêmes similitudes, relatives cette fois aux champs CHP_VISIB et CHP_CHGTVISIB.

2.1.3. la généricité

On introduit la structure générique des listes:

- le paramètre de généricité est:
  nom : fonction d'accès au nom d'un élément de la liste

- les propriétés sont
  recherche : recherche d'un élément dans la liste d'après son nom
  ins : insertion d'un élément dans la liste
  sup : suppression d'un élément de la liste

- l'implantation intuitive est:
  - une liste d'éléments,
  - l'élément de tête est statiquement défini: ceci est nécessaire pour réaliser des «effets de bord»,
  - *qui n'est pas un tableau*: on peut supprimer n'importe quel élément de la liste (et pas nécessairement le dernier = la pile).

La représentation qui s'impose de manière évidente est celle d'une liste chaînée.
Comment alors traduire cette structure sous la forme d'un paquetage générique?

Première réponse

La première approche consiste à considérer la liste comme un type de structure ("record") dont les champs sont les propriétés de la liste:

informellement:

```
type LISTE is record
                RECHERCHE : function ...;
                INS       : procedure ...;
                SUP       : procedure ...;
              end record;
```

en Ada:

```
package LISTE is
    function RECHERCHE ...;
    procedure INS ...;
    procedure SUP ...;
end LISTE;
```

On doit alors paramétrer le paquetage par:

- les "vrais" paramètres:
  TYPE_ELT : le type des éléments de la liste
  NOM : la fonction de nommage des éléments

- les "faux" paramètres:
  TYPE_REFELT : le type qui construit une liste chaînée d'éléments
  VAL, NXT, NULNXT, SETNXT, RESETNXT : les sous-programmes de manipulation des listes chaînées: accès à l'élément, accès au suivant, test (vrai si l'élément suivant est nul – null), création d'un nouvel élément suivant, suppression de l'élément suivant.

Le grand nombre de paramètres de généricité soulève deux problèmes:
- d'une part à chaque instanciation il faudra définir toute une panoplie de sous-programmes pour instancier les paramètres formels, ce qui est encombrant et n'introduit aucune nouvelle abstraction intéressante – parce que les paramètres demandés "collent" de trop près la structure de données sous-jacente;
- d'autre part le module générique a une sémantique floue: la moindre erreur de sémantique dans l'instanciation des paramètres rendra le comportement des procédures génériques tout à fait imprévisible – le module devient finalement inutilisable.

Deuxième réponse

Une deuxième possibilité est de réaliser un type générique dans le paquetage générique de la liste. C'est le type construit dans le paquetage qui sera utilisé par les instances.

en Ada:

```
generic
    type TYPE_ELT;
    with function NOM (X:in TYPE_ELT) return STRING;
package LISTE is
    type TYPE_LISTE is ...;
    function RECHERCHE ...;
    procedure INS ...;
    procedure SUP ...;
end LISTE;
```

Les paramètres du paquetage se limitent alors uniquement aux "vrais" paramètres de généricité: **TYPE_ELT** et la fonction **NOM**. On retrouve dans cette approche le traditionnel exemple de la pile paramétré par le type des éléments.

Dans le cas présent ce choix ne convient pas. En effet, les types de données manipulés sont:

```
type TYPE_PHYL;
type TYPE_OPER;
type TYPE_DRAP;

type TYPE_REFPHYL is access TYPE_PHYL;
type TYPE_REFOPER is access TYPE_OPER;
type TYPE_REFDRAP is access TYPE_DRAP;

type TYPE_PHYL is record
                    CHP_NOMPHYL  : TYPE_NOM;
                    CHP_ATTPHYL  : TYPE_ATTPHYL;
                    CHP_OPERPHYL : TYPE_REFDRAP;
                    CHP_PEREPHYL : TYPE_REFPHYL;
                    CHP_PARENT   : INTEGER;
                end record;

type TYPE_OPER is record
                    CHP_NOMOPER  : TYPE_NOM;
                    CHP_ATTOPER  : TYPE_ATTPHYL;
                    CHP_PHYLOPER : TYPE_REFDRAP;
                end record;

type TYPE_DRAP is record
                    CHP_PRESENT  : BOOLEAN;
                    CHP_VISIB    : INTEGER;
                    CHP_CHGTVISIB : BOOLEAN;
                end record;
```

Le type TYPE_LISTE des éléments chaînés se retrouve ici:
- pour le type TYPE_PHYL : dans le type TYPE_REFPHYL,
- pour le type TYPE_OPER : dans le type TYPE_REFOPER,
- pour le type TYPE_DRAP : dans le type TYPE_REFDRAP.

Il se trouve (par hasard) que la définition récursive du type TYPE_PHYL, par le champ CHP_PEREPHYL qui est une référence sur le type, interdit de définir le type TYPE_PHYL puis d'instancier le type des listes dans le paquetage LISTE.

Plus généralement on peut trouver l'approche peu naturelle: en effet elle demande de définir le *cas général* d'abord, puis de réaliser les cas particuliers à partir de ce cas général – on définit le type générique, puis on l'instancie. La démarche naturelle de paramétrisation est plutôt inverse: on réalise plusieurs objets, on les reconnaît bâtis sur un même modèle, on définit alors le modèle dont chaque réalisation devient une instance. C'est du reste la démarche qu'on a adopté ici, et qui a conduit à se placer dans une situation qu'on ne peut pas inverser.

2.2. les traitements

On regarde ici la première fonction "intéressante": *putphyl:

```
(de *putphyl (nomphyl attphyl)
    (let ((phyl (makephyl nomphyl attphyl)))
        (ins phylt phyl)
        (mapv '(lambda (oper)
                    (let ((drap (makedrap)))
                        (ins (phyloper oper) drap)
                        (ins (operphyl phyl) drap)))
              opert))
    (*putphyl-ana nomphyl attphyl))
```

On trouve dans cette fonction les principales facultés d'abstraction du langage Lisp:
- occurrences de *variables libres* phylt et opert: elles permettent de réaliser les «effets de bord;
- utilisation des *propriétés* des objets «de bas niveau»: makephyl, phyloper, ...; on nomme la propriété sans savoir quelle est son implantation – il s'agit peut-être de l'appel d'un traitement, peut-être aussi d'un accès (macro-généré) à un élément de la structure de données;
- utilisation de *fonctions génériques*, ici ins: parce que le langage est a-typé, la définition d'une fonction générique est particulièrement simple (et particulièrement peu contrôlée aussi);
- appel d'un traitement sur un *paramètre fonctionnel*: mapv sur la lambda-expression.

Les premiers points illustrent les possibilités d'abstraction de données du langage; le dernier concerne les traitements: c'est celui-là qu'on regarde maintenant.

2.2.1. dérivation de la fonction mapv

Mapv n'est pas (dans LeLisp V15.2) une fonction prédéfinie. C'est l'application de mapc (prédéfinie) sur le type liste – les listes pour lesquelles le premier élément n'est pas significatif. Ceci suggère *évidemment* l'emploi d'une macro-définition:

```
    (mapv fct ref1 ... refN)
;donne:
    (mapc fct (cdr ref1) ... (cdr refN))
```

De même si mapc n'était pas prédéfinie, on introduirait la macro-définition:

```
    (mapc fct lst1 ... lstN)
;donne:
    (let ((aux1 lst1) ... (auxN lstN))
        (while (and aux1 ... auxN)
               (fct (nextl aux1) ... (nextl auxN))))
```

On pourrait ensuite affiner nextl, si la fonction n'était pas prédéfinie:

```
    (nextl arg)
;donne:
    (prog1
        (car arg)
        (setq arg (cdr arg)))
```


On affinerait ensuite prog1, etc..., jusqu'à arriver aux symboles primitifs implantés effectivement par le langage.

On voit donc que le langage permet une «expression de haut niveau» du problème (mapc) et une implantation sur les éléments primitifs «de bas niveau» (while, next, ...) par des schémas de transformation qui restent transparents à l'utilisateur – c'est l'interprète Lisp qui se charge de réécrire le programme.
Pourquoi présenter cet aspect du langage sous la rubrique «macro-génération»? c'est que volontairement on se place dans l'optique d'une exécution **efficace** des programmes: dans l'absolu, parce qu'il est toujours désagréable d'attendre inutilement; pour les domaines spécifiques où l'efficacité est un impératif crucial – le Temps Réel.
Si l'on se plaçait dans une autre optique, on pourrait se limiter à toujours programmer en Lisp, dont la puissance d'expression est suffisante pour couvrir la totalité des techniques de programmation offertes dans les langages – H. Abelson et G. Sussman donnent dans [AbS 83] un panorama complet de la programmation par le biais du langage Scheme, un dialecte de Lisp.

2.2.2. l'expression dans un langage de haut niveau

Un langage de «haut niveau», comme Ada, est assez rebelle à l'expression de l'affinage d'un algorithme dans les termes du langage.

Pour représenter la fonctions mapv, on pourrait chercher à mettre encore en pratique la généricité d'Ada, mais ceci semble mal adapté; il faudrait:
- paramétrer la procédure générique mapv par le type d'objet (PHYL, OPER, DRAP) assorti des "faux sous-programmes" en paramètre et par le paramètre fonctionnel fct, lequel aurait *a priori* un nombre quelconque d'arguments;
- à chaque utilisation de mapv instancier tous les paramètres (les "vrais" et les "faux").

Plus simplement on recopie le code et de ce fait on disperse la connaissance.

Dans l'exemple, la relative simplicité des fonctions Lisp permet de facilement identifier les étapes de dérivation du programme Lisp dans le texte du programme Ada:

schéma page suivante

La traduction *automatique* d'un programme Lisp dans un langage «de haut niveau» ne va pas sans poser de nombreux problèmes:
- dans l'exemple le programme Ada obtenu est "peu naturel": le nombre d'imbrications de blocs de déclaration («declare») correspond mal aux schémas de présentation habituels – on définirait plutôt ici toutes les variables locales en tête de procédure;
- certaines transformations introduisent des variables auxiliaires (ici: la variable refoper), ce qui serait du reste encore plus vrai avec un langage moins «puissant» (Pascal par exemple): de la difficulté de générer automatiquement des noms significatifs;
- il y a beaucoup de non-dits dans le dessin – par exemple la fameuse fonction générique ins – qui devraient être explicités pour une élaboration automatique de la traduction.

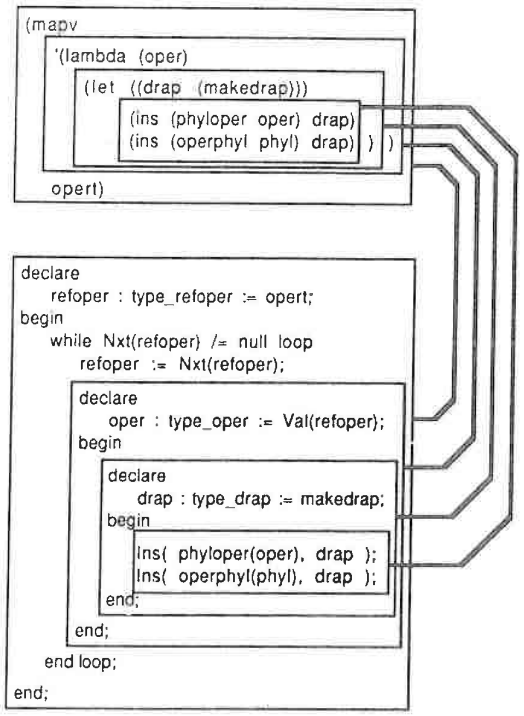


schéma d'identification des programmes Lisp et Ada

En inversant les rôles des partenaires – Lisp et Ada – on perd en *automatisation* du processus de traduction mais on gagne en *faisabilité*. Le programme est écrit en Ada, les occurrences d'apparition de la fonction mapv sont les occurrences d'utilisation du «texte» mapv: par affinage, via les «textes», du programme Ada, on reconstruit formellement la fonction Lisp, mais en travaillant dans le langage «exigeant» – «exigeant», si l'on admet que Lisp est plutôt un langage «accomodant».

3. Les symétries du programme

Comme il a déjà été dit, la relative proximité du modèle mathématique (les matrices) fait apparaître des symétries dans le programme.
On donne en annexe les fonctions de traitement (*put et *rem) et les fonctions auxiliaires (ascendance, *ins, *sup).

3.1. les données

Les types PHYL et OPER, des phyla et des opérateurs, travaillent sur une même matrice: il est donc assez logique de retrouver des similitudes entre les traitements relatifs aux phyla et ceux relatifs aux opérateurs.

On donne les deux fonctions où la similitude est la plus manifeste:

```
(de *putphyl (nomphyl attphyl)
    (let ((phyl (makephyl nomphyl attphyl)))
         (ins phylt phyl)
         (mapv '(lambda (oper)
                   (let ((drap (makedrap)))
                        (ins (phyloper oper) drap)
                        (ins (operphyl phyl) drap)))
               opert))
    (*putphyl-ana nomphyl attphyl))

(de *putoper (nomoper attoper)
    (let ((oper (makeoper nomoper attoper)))
         (ins opert oper)
         (mapv '(lambda (phyl)
                   (let ((drap (makedrap)))
                        (ins (operphyl phyl) drap)
                        (ins (phyloper oper) drap)))
               phylt))
    (*putoper-ana nomoper attoper))
```

Il suffit de remplacer *lexicalement* oper en phyl et phyl en oper pour changer de fonction. Le remplacement lexical, assez facile à faire sous un bon éditeur de texte, devient terriblement fastidieux si l'on veut l'exprimer dans le langage. En effet tous les termes utilisés dans les fonctions deviennent des paramètres de généricité: la fonction est totalement *"creuse"*, et il sera quasiment impossible de la modifier.

Les autres similitudes sont plus floues, mais non moins décelables: il serait encore plus difficile de les expliciter syntaxiquement; il sera plus subtile de les traduire lexicalement («textuellement»), mais raisonnablement pas impossible.

3.2. les traitements

Les traitements *put et *rem sont inverses l'un de l'autre, par définition. On va donc encore assez logiquement observer une symétrie entre ces traitements – sur un même type de données PHYL ou OPER.

On donne ici aussi les deux fonctions où la similitude est la plus visible:

```
(de *putoper (nomoper attoper)
    (let ((oper (makeoper nomoper attoper)))
         (ins opert oper)
         (mapv '(lambda (phyl)
                   (let ((drap (makedrap)))
                        (ins (operphyl phyl) drap)
                        (ins (phyloper oper) drap)))
               phylt))
    (*putoper-ana nomoper attoper))

(de *remoper (oper)
    (let ((numoper (num opert oper)))
         (numsup opert numoper)
         (mapv '(lambda (phyl)
                   (numsup (operphyl phyl) numoper))
               phylt))
    (*remoper-ana oper))
```

Une écriture «plus orthodoxe» de *remoper serait:

```
(de *remoper (oper)
    (sup opert oper)
    (mapv '(lambda (phyl)
              (sup (operphyl phyl) oper))
          phylt))
    (*remoper-ana oper))
```

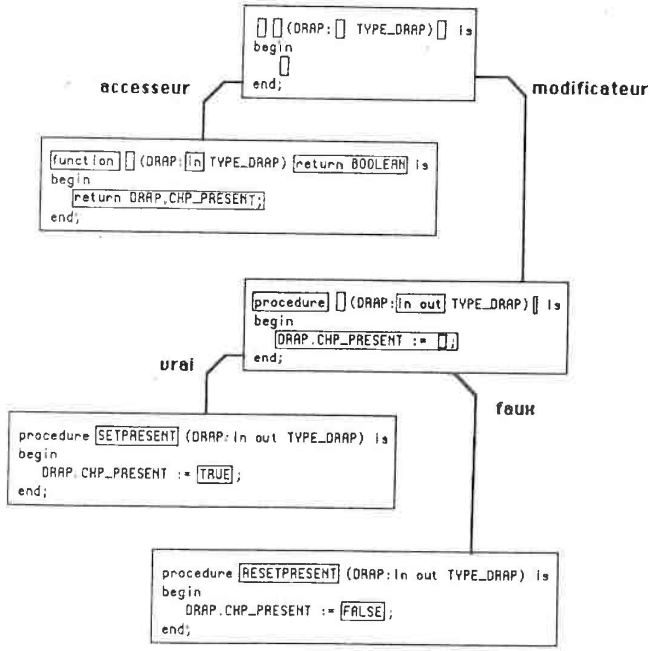
qui rend le parallèle plus facile entre les deux fonctions – on remplace, *subtilement*, ins par sup ou réciproquement.
Les fonctions num... qui ont été introduites permettent d'améliorer l'efficacité de l'algorithme: au lieu de rechercher un élément de la liste d'après son nom, on retient son numéro d'ordre dans la liste (numoper), puis on accède directement à l'élément par ce numéro.

Pour garantir le passage de *remphyl à *remoper, il faut donc:
- maintenir le remplacement de ins par sup,
- introduire un remplacement *contextuel*: dans le contexte de la déclaration de la variable locale num, l'appel de la fonction sup est «textuellement» remplacé par numsup, et le paramètre oper par numoper.

On peut noter que dans ces manipulations «textuelles» du programme on est très loin du langage et de la sémantique des termes qu'il définit, et pourtant on est très près du sens intuitif qu'on donne au texte du programme.

Annexe 1 : Construction du champ CHP_PRESENT

Arbre de construction



Représentation «textuelle»

```
(def sous-programme ()
   ("{TYP-SS-PGME} {NOM-SS-PGME}
                    (DRAP:{SNS-PARAM} TYPE_DRAP) {CPT-SS-PGME} is
     begin
        {CORPS}
     end;"))

(def accesseur ()
   ((def TYP-SS-PGME () ("function"))
    (def SNS-SS-PGME () ("in"))
    (def CPT-SS-PGME () ("return BOOLEAN"))
    (def CORPS () ("return DRAP.CHP_PRESENT;")))
   ("{sous-programme}"))

(def modificateur ()
   ((def TYP-SS-PGME () ("procedure"))
    (def SNS-SS-PGME () ("in out"))
    (def CPT-SS-PGME () (""))
    (def CORPS () ("DRAP.CHP_PRESENT := {VAL};")))
   ("{sous-programme}"))

(def test ()
   ((def NOM-SS-PGME () ("PRESENT")))
   ("{accesseur}"))

(def set ()
   ((def NOM-SS-PGME () ("SETPRESENT"))
    (def VAL () ("TRUE")))
   ("{modificateur}"))

(def reset ()
   ((def NOM-SS-PGME () ("RESETPRESENT"))
    (def VAL () ("FALSE")))
   ("{modificateur}"))
```


Annexe 2 : Les fonctions de traitement en Lisp

```
;==================
    (de *putphyl (nomphyl attphyl)
        (let ((phyl (makephyl nomphyl attphyl)))
             (ins phylt phyl)
             (mapv '(lambda (oper)
                        (let ((drap (makedrap)))
                             (ins (phyloper oper) drap)
                             (ins (operphyl phyl) drap)))
                   opert))
        (*putphyl-ana nomphyl attphyl))
;------------------
    (de *putoper (nomoper attoper)
        (let ((oper (makeoper nomoper attoper)))
             (ins opert oper)
             (mapv '(lambda (phyl)
                        (let ((drap (makedrap)))
                             (ins (operphyl phyl) drap)
                             (ins (phyloper oper) drap)))
                   phylt))
        (*putoper-ana nomoper attoper))
;------------------
    (de *putoperphyl (phyl oper)
        (let ((drap (drap (operphyl phyl) (nomoper oper))))
             (ascendance phyl)
             (*insereoperphyl oper 1)
             (setpresent drap))
        (*putoperphyl-ana phyl oper))
;------------------
    (de *putperephyl (phyl pere)
        (ascendance pere)
        (when (parent phyl)
              (ERREUR "le phylum " (nomphyl phyl)
                      " est parent du phylum " (nomphyl pere)))
        (mapv '(lambda (oper drapoper)
                   (when (visib drapoper)
                         (*insereoperphyl oper (valvisib drapoper))))
              opert (operphyl phyl))
        (ins (perephyl phyl) pere)
        (*putperephyl-ana phyl oper))
;------------------
    (de *insereoperphyl (oper valvisib)
        (mapv '(lambda (phyl drapphyl)
                   (resetnouvvisib drapphyl)
                   (when (parent phyl)
                         (incrvisib drapphyl valvisib (valparent phyl))))
              phylt (phyloper oper))
        (*insereoperphyl-ana oper))
;==================
```

```
;====================
    (de *remphyl (phyl)
        (mapv '(lambda (pere)
                   (*remperephyl phyl pere))
              (perephyl phyl))
        (mapv '(lambda (fils)
                   (when (recherche (perephyl fils) (nomphyl phyl))
                         (*remperephyl fils phyl)))
              phylt)
        (let ((numphyl (num phylt phyl)))
            (numsup phylt phyl)
            (mapv '(lambda (oper)
                       (numsup (phyloper oper) numphyl))
                  opert))
        (*remphyl-ana phyl))
;--------------------
    (de *remoper (oper)
        (let ((numoper (num opert oper)))
            (numsup opert numoper)
            (mapv '(lambda (phyl)
                       (numsup (operphyl phyl) numoper))
                  phylt))
        (*remoper-ana oper))
;--------------------
    (de *remoperphyl (phyl oper)
        (let ((drap (drap (operphyl phyl) (nomoper oper))))
            (ascendance phyl)
            (*supprimeoperphyl oper 1)
            (resetpresent drap))
        (*remoperphyl-ana phyl oper))
;--------------------
    (de *remperephyl (phyl pere)
        (ascendance pere)
        (mapv '(lambda (oper drapoper)
                   (when (visib drapoper)
                         (*supprimeoperphyl oper (valvisib drapoper))))
              opert (operphyl phyl))
        (sup (perephyl phyl) pere)
        (*remperephyl-ana phyl oper))
;--------------------
    (de *supprimeoperphyl (oper valvisib)
        (mapv '(lambda (phyl drapphyl)
                   (resetancivisib drapphyl)
                   (when (parent phyl)
                         (decrvisib drapphyl valvisib (valparent phyl))))
              phylt (phyloper oper))
        (*supprimeoperphyl-ana oper))
;====================

    (de ascendance (phyl)
        (mapv 'resetparent phyl)
        (ascendance-int phyl))
    (de ascendance-int (phyl)
        (setparent phyl)
        (mapv 'ascendance-int (perephyl phyl)))
```


Chapitre 3:

Le Langage Complété pour la Structuration des Textes

Après la présentation de la *Syntaxe Concrète*, qui permet la manipulation de «textes» individuels, on donne celle de la *Syntaxe Complétée*, dont l'introduction permet *l'exécution de code Lisp* et la gestion de *listes de «textes»*.
On illustre par des exemples les nouvelles fonctionnalités qui en résultent.

Chapitre 3.1:

Présentation de la Syntaxe Complétée

On introduit de nouveaux opérateurs des «textes»:
- lsp = on peut insérer du code Lisp dans les «arbres de textes»,
- lst = on peut gérer des listes de «textes»,
- deft = on peut typer les «textes».

Ces nouveaux opérateurs permettent alors de donner une *dynamique* aux «textes». La *Syntaxe Concrète* précédente à laquelle on adjoint ces nouveaux opérateurs devient alors la *Syntaxe Complétée*.

1. Comportement dynamique des textes

1.1. modification textuelle

La définition d'un texte, puis son utilisation, permettent de *concentrer*, à la déclaration de ce texte, l'ensemble de l'information qu'il représente. Or un emploi un tant soit peu intéressant de ce mécanisme demande de pouvoir définir des textes incomplets: on conçoit que des textes "entièrement définis" ne correspondent qu'à des cas bien particuliers d'utilisation: ce peuvent être les mots-clés du langage utilisé, des identificateurs, certains schémas répétitifs qu'impose le langage, ... On est très vite limité par l'emploi exclusif de textes "entièrement définis" puisqu'alors à l'utilisation d'un texte on ne peut d'aucune manière bénéficier d'informations spécifiques au contexte d'utilisation dans lequel on se situe.

Une première réponse apportée est la *paramétrisation* des textes. Comme il est décrit dans la partie traitant du contexte d'évaluation des textes, le choix retenu a été de définir la paramétrisation comme l'absence de définition. Cela signifie que la notion de texte paramétré est elle-même fonction du contexte dans lequel on se situe, ou même de la vision qu'on a de ce contexte.
Sur la figure qui suit on montre un exemple où une simple paramétrisation est malgré tout insuffisante pour définir un unique texte pour la déclaration des deux types homme et femme:

```
TYPE type_sexe : (masculin,feminin);
TYPE type_info (sexe:type_sexe) : ...;
TYPE homme : RECORD
               identif  : type_info(masculin);
               conjoint : type_info(feminin);
             END RECORD;
TYPE femme : RECORD
               identif  : type_info(feminin);
               conjoint : type_info(masculin);
             END RECORD;
```

On reprend le texte de la figure précédente, en introduisant, dans le texte du programme, la notion de type paramétré:

```
TYPE type_sexe : (masculin,feminin);
FUNCTION oppose (sexe:type_sexe)
 BEGIN
   CASE sexe
    WHEN masculin : oppose:=feminin;
    WHEN feminin  : oppose:=masculin;
   END CASE;
 END;
TYPE type_info (sexe:type_sexe) : ...;
TYPE humain (sexe:type_sexe) :
      RECORD
        identif  : type_info(sexe);
        conjoint : type_info(oppose(sexe));
      END RECORD;
TYPE homme : humain(masculin);
TYPE femme : humain(feminin);
```


Il faut noter qu'on demande alors beaucoup de bienveillance de la part du langage utilisé:
- on peut paramétrer un type par un objet de n'importe quel type (ici: type_sexe);
- on peut instancier un paramètre formel par un autre paramètre formel (ici: sexe);
- on peut appeler, à la définition d'un type, une fonction définie par l'utilisateur (ici: oppose);
- on peut définir des types dérivés (ici: homme et femme).

On peut noter aussi que le texte obtenu:
- est plus long,
- n'est pas tellement plus général,
- mais est certainement plus délicat à relire.

Il serait donc souhaitable de pouvoir définir des "points de dialogue" entre les textes définis et l'évaluation des textes. A ces points, la représentation serait alors calculée en partie par l'application d'instructions intelligibles de l'évaluateur.

Par exemple:

```
def humain
= "TYPE " (nom) ":" "^M"
  "RECORD" "^M"
  " identif  : type_info(" (sexe) ");" "^M"
  " conjoint : type_info("
        (si (sexe) = "masculin"
            alors "feminin"
            sinon "masculin")  ");" "^M"
  "END RECORD;" "^M"

def homme
= (use humain
     def nom = "homme"
     def sexe = "masculin")
def femme
= (use humain
     def nom = "femme"
     def sexe = "feminin")
```

1.2. contrôle non textuel

La manipulation symbolique des textes offre, en plus d'une facilité accrue de construction et de maintenance du programme, la possibilité de réaliser un certain nombre de contrôles non textuels: ce seront des comportements dont aucune trace ne restera visible dans le texte source du programme, et qui sont pourtant directement déduits de la texture du programme.

Pour illustrer le propos, on peut prendre l'exemple du contrôle de type; réaliser le contrôle de type nécessite une connaissance au moins parcellaire du *contexte* où l'on se trouve: il faut savoir reconnaître ce qu'est l'«emploi d'une variable», il faut aussi reconnaître les déclarations de types – problèmes liés à la syntaxe du langage utilisé –, et la portée des déclarations – problème lié à la sémantique statique du langage.

Dans l'exemple de la figure A, ce contrôle est réalisé dès la phase d'édition: on diminue donc la distance entre la détection d'une erreur et le moyen de la corriger.

figure a: On déclare symboliquement les variables "A", "B" et "C" de type respectif "integer", "integer" et "real".

```
def var
   def decla = "VAR " (nom) ":" (type) ";" "^M"
   def util = (nom)

def varA
   def var = "A"
   def type = "integer"
   (ref var)

def varB
   def var = "B"
   def type = "integer"
   (ref var)

def varC
   def var = "C"
   def type = "real"
   (ref var)
```

figure b: On déclare un texte qui réalise l'affectation affect:
- il est paramétré par les deux textes var1 et var2;
- il contrôle que les textes var1 et var2 ont une "propriété" type identique – que les deux variables sont de même type.

```
def affect
= (util(var1)) ":="
     (si (type(var1)) = (type(var2))
         alors
         sinon "(erreur de type)")
     (util(var2)) ";" "^M"
```

figure c:
1: On utilise affect avec les variables "A" et "B": aucune erreur n'est détectée.
2: On utilise affect avec les variables "A" et "C": l'affichage signale la détection d'une erreur de type.
3: On utilise affect en oubliant un paramètre (var2): l'affichage signale ici aussi la détection d'une erreur – le message pourrait être plus explicite en prévoyant le cas dans la définition de affect (pour la gestion des définitions partielles, cf. Chapitre 4.2, *«les difficultés»*).

```
1: (use affect                donne
      def var1 (ref varA)        A:=B;
      def var2 (ref varB))

2: (use affect                donne
      def var1 (ref varA)        A:=(erreur de type)C;
      def var2 (ref varC))

3: (use affect                donne
      def var1 (ref varA))       A:=(erreur de type)(util);
```


On notera que le système ne fait que *signaler*, par affichage, les cas d'erreur, mais ne les refuse pas. On est en effet dans la phase d'édition: nombreux seront les cas où, provisoirement au moins, on aura un texte sémantiquement incorrect, mais qui devrait être rapidement corrigé. On ne peut pas alors contraindre l'utilisateur à respecter l'ordre des déclarations que requiert le compilateur, ce qui serait inutilement pénible, et encore moins le contraindre à réaliser des modifications simultanées, surtout quand ce n'est pas possible – par exemple ici, il n'est pas possible de changer simultanément les types de "A" et "B" pour le type "real".

Des exemples autres que le contrôle de type sont plus difficiles à présenter; plus justement, toute la sémantique statique du langage (typage, déclarations des procédures, ...) pourrait être prise en compte: je veux parler ici de contrôles de plus hauts niveaux. Il faudrait s'inspirer des outils de spécification et de validation d'algorithmes, pour réaliser des contrôles de terminaison de boucle, préservation d'invariant, initialisation des variables ...

1.3. insertion de code Lisp

Les paragraphes précédents ont souligné les avantages qu'on retirerait d'un évaluateur qui sache reconnaître, en plus des définitions et emplois de textes, les instructions d'un langage à définir. On pourrait donc envisager d'enrichir la syntaxe des textes présentée au chapitre précédent, en définissant:
- un phrase conditionnelle,
- une phrase de sélection,
- une phrase d'itération ...

Plus simplement, on autorise l'insertion de code Lisp. Ceci est bien sûr très lié au fait que l'évaluateur est écrit en Lisp, mais il fallait bien choisir un langage.

L'avantage de Lisp est que le langage est interprété, qu'il a des outils puissants associés (LeLisp V15.2), et qu'il est bien adapté à la manipulation de données "changeantes" – les structures de données peuvent contenir des traitements. Ce choix permet surtout de se dispenser de la définition d'un nouveau langage, qui entraîne toujours avec elle divers soucis:
- analyse lexicale,
- cohérence de la grammaire,
- complétude de la grammaire,
- difficulté de dialogue avec les outils existants,
- bibliothèque des fonctions prédéfinies à construire.

1.4. syntaxe Lisp

On définit ici la nouvelle syntaxe de description des textes, dans laquelle apparaissent les expressions Lisp. Dans la présentation des grammaires, on a omis les parenthèses qu'impose Lisp pour faciliter la lecture.

1.4.1. rappel des syntaxes

On reprend les syntaxes décrites plus en détail au chapitre traitant du contexte d'évaluation (cf. Chapitre 9.1, *«Contexte d'évaluation»*).

syntaxe concrète

On apporte une petite modification au niveau des chaînes de caractères:

```
(1)  env ::= ε | trm env
(2)  trm ::= def | ref
(3)       def  ::= <nom> env rep
(4)       ref  ::= <nom> env
(5)  rep ::= ε | atm rep
(6)  atm ::= stg | use | <string>
(7)       stg  ::= <nom>
(8)       use ::= <nom> env
```

(1) un environnement est une liste de termes
(2) un terme est une définition def ou une référence ref
(3) une définition est: un nom <nom>, un environnement, une représentation
(4) une référence est: un nom <nom>, un environnement
(5) une représentation est une liste d'atomes
* (6) un atome est une *string* stg ou une utilisation use ou une chaîne de caractères <string>
* (7) une *string* stg est: un nom <nom>
(8) une utilisation est: un nom <nom>, un environnement

On introduit une distinction entre
- stg: un nom,
- STRING: une chaîne de caractères,

pour plus de symétrie entre les définitions des environnements et des représentations. stg ne sera peut-être pas très utile en pratique.

syntaxe évaluée

C'est la *syntaxe évaluée* des textes – ou encore la syntaxe des textes évalués, c'est-à-dire la syntaxe des expressions retournées par une évaluation:

```
(1e) env ::= ε | trm env
(2e) trm ::= def | ref
(3e)      def  ::= <singleton>
(4e)      ref  ::= <singleton>
(5e) rep ::= ε | atm rep
(6e) atm ::= <string> | rep
```


(1e) un environnement est une liste de termes
(2e) un terme est une définition def ou une référence ref
(3e) une définition est atome du langage (<singleton>)
(4e) une référence est atome du langage (<singleton>)
(5e) une représentation est une liste d'atomes
(6e) un atome est une chaîne de caractères <string> ou une représentation:
- une chaîne de caractère s'évalue identiquement à elle-même;
- une *string* stg s'évalue en convertissant le nom de stg en une chaîne de caractères – par exemple (stg txt) s'évalue: "txt";
- une utilisation retourne la représentation évaluée du texte;
- une utilisation indéfinie – le texte n'a pas été trouvé – est convertie en une chaîne de caractères, avec un "format d'erreur" conventionnel – par exemple: (use txt) donne: "{txt}".

1.4.2. syntaxe commune

Les syntaxes non évaluée et évaluée se ressemblant beaucoup, on est conduit à les identifier, ce qui simplifie les notions introduites sans les appauvrir ni les surcharger.

```
(1)  env ::= ε | trm env
(2)  trm ::= def | ref | env
(3)       def  ::= <nom> env rep
(4)       ref  ::= <nom> env
(5)  rep ::= ε | atm rep
(6)  atm ::= stg | use | <string> | rep
(7)       stg  ::= <nom>
(8)       use ::= <nom> env
```

(1) un environnement est une liste de termes
* (2) un terme est une définition def ou une référence ref ou un environnement env – et ceci même *avant* l'évaluation
(3) une définition est: un nom <nom>, un environnement, une représentation
(4) une référence est: un nom <nom>, un environnement
(5) une représentation est une liste d'atomes
* (6) un atome est une *string* stg ou une utilisation use ou une chaîne de caractères <string> ou une représentation rep – et ceci même *avant* l'évaluation
(7) une *string* stg est: un nom <nom>
(8) une utilisation est: un nom <nom>, un environnement

On enrichit un peu la syntaxe qui se définit avant l'évaluation; on enrichit surtout celle qui se définit après l'évaluation:

- def et ref ne sont plus des atomes (<singleton>), mais sont définis comme ils l'étaient *avant* l'évaluation. Ceci n'est qu'une question de point de vue: def vu comme un atome possédait quand même un champ env et un champ rep, mais qui étaient ceux de la syntaxe *non évaluée*; dans la syntaxe *évaluée*, ces champs étaient donc non significatifs, quoique présents – puisqu'une définition est évaluée identiquement à elle-même. En confondant, *syntaxiquement*, les notions d'environnement ou de représentation évalués et non évalués, on "démasque" les champs de def, ce qui ne prête pas à conséquence. Il reste qu'une définition sera toujours invariante par évaluation, et en particulier que les champs env et rep seront les formes *non évaluées* des champs env et rep de définition de def.
  La même remarque s'applique pour ref.

- Pour les représentations, on a syntaxiquement encore, beaucoup plus de possibilités sur la syntaxe évaluée qu'on en avait auparavant. Ici aussi c'est une question de point de vue: l'évaluation ne retournant que des chaînes de caractères (<string>) ou des représentations rep, certaines dérivations de la grammaire ne seront jamais réalisées.

Le choix de restreindre ainsi le nombre de dérivations possibles de la syntaxe évaluée sur les représentations n'est pas tout à fait innocent. On peut en effet juger qu'un emploi "transparent" des textes ne montre à l'utilisateur que les représentations évaluées: la syntaxe non évaluée et les environnements devraient être gérés automatiquement pour assurer toujours une lecture agréable du texte construit. Dans cette optique la syntaxe doit ressembler de très près à celle que reconnaît un écran Vidéo, à savoir des chaînes de caractères.
On conserve le concept de représentation rep pour un simple problème technique de tabulation: la syntaxe évaluée des représentations fournit donc bien pratiquement des suites de caractères.

1.4.3. syntaxe Lisp

On a repris la syntaxe précédente, en plaçant le Lisp lsp partout où on peut l'attendre:
- ni sur env, ni sur rep qui sont des listes respectivement de termes trm et d'atomes atm,
- ni sur def, ref, stg, use qui sont les "atomes" du langage,
- sur trm et atm qui sont les concepts du langage où on attend "diverses choses", sans impératif absolu.

Les deux dernières formules décrivent une expression Lisp:

```
(1)  env ::= ε | trm env
(2)  trm ::= def | ref | env | lsp
(3)       def  ::= <nom> env rep
(4)       ref  ::= <nom> env
(5)  rep ::= ε | atm rep
(6)  atm ::= stg | use | <string> | rep | lsp
(7)       stg  ::= <nom>
(8)       use ::= <nom> env
(9)  lsp  ::= ε | sex lsp
(10) sex ::= trm | atm | <liste> | <atome>
```

(9) un texte Lisp lsp est une liste de S-expressions (S-ex)
(10) une S-expression sex est un terme trm ou un atome atm, ou bien encore l'un des deux terminaux qui tiennent compte de la syntaxe des S-expressions de Lisp: une liste (<liste>) ou un atome (<atome>)

Note: syntaxe Lisp des S-expressions:

```
S-ex ::= list | atome
liste ::= '(' { S-ex } ')'
atome ::= IDENT
```

soit:
- une S-expression est une liste ou un atome,
- un atome est un identificateur Lisp (ou une chaîne de caractères, un nombre,...).
- une liste est: «ouvrez la parenthèse», des S-ex, «fermez la parenthèse», d'où une densité élevée de parenthèses dans les programmes Lisp.

On reprend sur les exemples précédents:

Le type humain, paramétré par le type type_sexe:

```
def humain
= "TYPE " (nom) ";" "^M"
  "RECORD" "^M"
  " identif  : type_info(" (sexe) ");" "^M"
  " conjoint : type_info("
  (lsp
     (if (= (use sexe) "masculin")
         "feminin"
         "masculin")) ");" "^M"
  "END RECORD;" "^M"
```

Le contrôle de type sur l'affectation:

```
def affect
= (util1(var1)) ":="
  (lsp
     (if (= (use type(var1)) (use type(var2)))
         ()
         "(erreur de type)"))
  (util2(var2)) ";" "^M"
```

Note:
On remarquera sur les exemples que:
- en dehors des expressions Lisp, on se dispense de préfixer une utilisation de texte par 'use', parce qu'il n'y a pas d'ambiguïté;
- à l'intérieur d'une expression Lisp, on la préfixe.

L'ambiguïté qui doit être levée dans le second cas tient à ce que, dans une expression Lisp, il est plus simple de reconnaître un des préfixes introduits ('def', 'ref', ...) que de reconnaître le nom d'une fonction Lisp (il y en a beaucoup): de ce fait, c'est la solution simple qui a été adoptée.

2. La syntaxe complétée

On présente plusieurs opérateurs qu'on ajoute à la *syntaxe concrète*; on construit ainsi la *syntaxe complétée*, qui à son tour pourra être complétée par de nouvelles définitions.
- l'opérateur fct: il simplifie l'insertion d'une expression Lisp;
- les opérateurs lst-init, lst, tst, lst-env: ils permettent la manipulation de *listes* de textes;
- les opérateurs lst2-init, lst2, tst2, atm1, atm2: ils permettent le parcours simultané de deux listes de textes;
- les opérateurs deft, reft, uset: ils servent à *typer* les textes, à leur définition et à leur emploi.

2.1. opérateur de S-expression simple

SYNTAXE

```
lsp ::= lsp | fct
fct ::= sex
```

SEMANTIQUE

fct simplifie la syntaxe d'emploi d'une S-ex Lisp, quand on souhaite ne faire figurer qu'une seule S-ex.

EXEMPLE

```
(lsp
  (if (= (use x) 0)
      "zero"
      "non nul"))
```

```
(fct if (= (use x) 0)
        "zero"
        "non nul")
```

2.2. opérateurs de listes

SYNTAXE

```
atm ::= atm | lst-init | lst tst
lst-init ::= env rep
lst      ::= ε | atm lst
tst      ::= ε | sex tst

env ::= env lst-env
lst-env ::= ε | env lst-env
```

SEMANTIQUE

opérateur lst-init
Il permet d'initialiser le parcours d'une liste d'environnements. Le champ d'environnement est un environnement prioritairement placé dans le contexte d'évaluation pour la construction de la liste des environnements à parcourir. Le champ de représentation est la représentation qui sera évaluée pour chaque nouvelle construction d'un contexte – pour la construction du contexte, voir l'opérateur lst-env.

opérateur lst
Il permet, *à l'intérieur* d'un opérateur lst-init, de spécifier une autre représentation à évaluer pour le parcours du reste de la liste d'environnements. La liste sera parcourue jusqu'à épuisement à l'intérieur de l'opérateur lst, et ne sera alors plus parcourue par l'opérateur lst-init, ou par d'autres opérateurs lst englobants.
Note: à l'entrée dans lst, on commence par avancer d'un rang dans la liste des environnements.

opérateur tst
Il permet de tester le nombre d'environnements restant dans la liste à parcourir, et de spécifier une certaine représentation à évaluer en fonction de ce nombre. La syntaxe est plus sévère qu'il n'y paraît. La syntaxe d'emploi correcte, à respecter mais non contrôlée, est celle du selectq de Lisp (le 'case of' de Pascal):

```
forme entrée :
   (tst
     (0  (rep ...))
     (1  (rep ...))
         ...
     (T  (rep ...)))

forme Lisp :
   (selectq (nombre d'environnements restant dans la liste)
     (0  (rep ...))
     (1  (rep ...))
         ...
     (T  (rep ...)))
```

Note: dans le nombre d'environnements restants, on compte l'environnement dans lequel on est en train d'évaluer.

opérateur lst-env
Il permet de définir, "proprement", une liste d'environnements. Le parcours de la liste s'effectue en deux étapes:

- première étape: *initialisation.*
  On construit le contexte formé de l'environnement de lst-init puis du contexte courant d'évaluation. Le premier environnement rencontré dans le contexte évalué est alors l'environnement de liste:
  - s'il s'agit d'un opérateur lst-env, on aura bien une liste d'environnements;
  - s'il s'agit d'un environnement ordinaire, cet environnement sera vu comme une liste d'environnements composée d'un seul élément.

  Cette possibilité permet de préserver la compatibilité entre des textes qui utilisaient un certain texte txt qui n'était pas un texte de liste, et des textes qui utilisent ce même texte txt qui a été transformé en un texte de liste.

- deuxième étape: *parcours.*
  Pour chaque élément de la liste d'environnements:
  - on évalue cet environnement dans le contexte ctx auquel est attachée la liste d'environnements;
  - on évalue la représentation de lst-init ou de lst dans le contexte formé de cet environnement évalué et du contexte ctx;
  - on avance d'un pas dans la liste et on réitère le procédé, jusqu'à épuisement de la liste.

EXEMPLE

lst-param: le texte est paramétré par une liste de paramètres l-param; c'est la décompilation des indices de tableau.

```
def lst-param
= (lst-init
     (env (l-param))
     (rep "[" (nom) (lst "," (nom)) "]")))
```

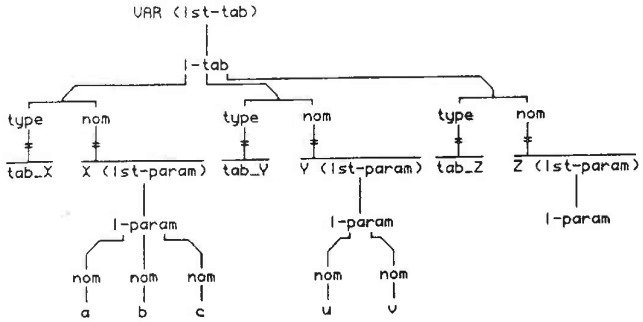
lst-tab: le texte est paramétré par une liste de déclaration l-tab; c'est la décompilation des déclarations de variables, dans une syntaxe du type de la syntaxe de Pascal.

```
def lst-tab
= "VAR" "^M"
  (lst-init
     (env (l-tab))
     (rep "   " (nom) ":" (type) ";" "^M"))
```

decla: le texte définit trois variables "X", "Y" et "Z", elles-mêmes définies comme des tableaux – "Z" est un tableau sans indice: le texte l-param a un environnement vide.

```
(def decla
   (def l-tab
      (lst-env
         (env
            (def nom
               (def l-param
                  (lst-env
                     (env  (def nom = "a"))
                     (env  (def nom = "b"))
                     (env  (def nom = "c"))))
               = "X" (lst-param))
            (def type
               = "tab_X"))
         (env
            (def nom
               (def l-param
                  (lst-env
                     (env  (def nom = "u"))
                     (env  (def nom = "v"))))
               = "Y" (lst-param))
            (def type
               = "tab_Y"))
         (env
            (def nom
               (def l-param)
               = "Z" (lst-param))
            (def type
               = "tab_Z"))))
  = (lst-tab))
```


On peut ainsi représenter l'arbre abstrait associé à la définition de ces trois variables:



La forme évaluée de l'utilisation du texte decla est alors:

```
VAR
  X[a,b,c]: tab_X;
  Y[u,v]: tab_Y;
  Z: tab_Z;
```

2.3. opérateurs de listes doubles

SYNTAXE

```
atm  ::= atm | lst2-init | lst2 | tst2 | atm1 | atm2
     lst2-init ::= env env env rep
     lst2      ::= ε | atm lst2
     tst2      ::= ε | sex tst2
     atm1      ::= ε | atm atm1
     atm2      ::= ε | atm atm2
```

SEMANTIQUE

opérateur lst2-init

Il permet le parcours simultané de deux listes:
- le premier champ d'environnement définit la première liste d'environnements;
- le deuxième définit la deuxième liste;
- le troisième définit les cas d'erreur; il doit nécessairement contenir la définition des textes:
  - miss1: on a épuisé la première liste, sans avoir épuisé la deuxième;
  - miss2: on a épuisé la deuxième liste, sans avoir épuisé la première.

De plus, il sert d'environnement local pour l'évaluation des représentations: les définitions présentes dans cet environnement pourront donc être utilisées dans le champ rep.

Le champ de représentation est, comme pour lst-init, la représentation qui sera évaluée pour chaque paire d'environnements lors du parcours des listes.

opérateur lst2
Il se définit comme lst, mais en parcourant les deux listes d'environnements.

opérateur tst2
Il se définit comme tst, le test portant sur la première liste.

opérateur atm1
Il permet d'exprimer, dans une représentation, qu'on désire voir évaluer ce qui suit dans le contexte construit avec la première liste d'environnements. Les atomes atm qui n'apparaissent pas sous atm1 (ou atm2) sont évalués dans le contexte de départ.

opérateur atm2
C'est le symétrique de atm1, avec la deuxième liste.

EXEMPLE

- On définit le texte affiche, qui permet l'affichage des indices d'un tableau "A". Pour cet affichage, on contrôle le type de chacun des indices utilisés par rapport au type déclaré.
  Le paramètre l-int fait référence à la liste des types d'indices déclarés.
  Le paramètre l-ind fait référence à la liste des types d'indices utilisés.
  Dans l'environnement local, on définit:
  - les deux textes nécessaires miss1 et miss2,
  - deux textes auxiliaires test et erreur.

```
(def affiche
= "A" "["
  (lst2-init
     (env (l-int))
     (env (l-ind))
     (env
        (def test
           = (atm2 (exp))
             (fct unless (= (atm1 (type))
                            (atm2 (type)))
                  (rep "(erreur de type: " (atm1 (type)) ")")))
        (def miss1
           (def occ = (exp))
           = "(trop de parametres)" (erreur))
        (def miss2
           (def occ = (type))
           = "(manque des parametres)" (erreur))
        (def erreur
           = (occ) (lst "," (occ))))
     = (test) (lst2 "," (test)) "]")
```

On remarquera que pour erreur on n'utilise plus l'opérateur lst2 mais l'opérateur lst: l'évaluation de lst2-init ou de lst2 garantit que, si l'on épuise une des listes et non l'autre, alors on se retrouve "à l'intérieur" d'une évaluation virtuelle de l'opérateur lst-init où la liste parcourue est la liste qu'on n'a pas encore épuisée, à la hauteur où on est dans cette liste au moment où l'erreur est détectée.

- On définit la liste l-int des types des indices, et le texte decla de déclaration de la variable "A".

```
(def l-int
   (lst-env
      (env  (def type = "jour"))
      (env  (def type = "1..31"))))

(def decla
   = "A : ARRAY "
     (lst-init
        (env (l-int))
        (rep "[" (type) (lst "," (type)) "]"))
     " OF horaire;")
```

- On définit la liste l-ind des expressions exp et types type des indices utilisés.

```
(def l-ind
   (lst-env
      (env  (def exp  = "dimanche")
            (def type = "jour"))
      (env  (def exp  = "20")
            (def type = "1..31"))))
```

- On représente l'évaluation des textes decla et affiche, en tenant compte de toutes les déclarations faites jusqu'ici.

```
(use decla) :
      A : ARRAY [jour,1..31] OF horaire;

(use affiche) :
      A[dimanche,20]
```

- On présente divers cas d'utilisation de affiche, en simplifiant la présentation. La colonne de gauche montre ce qu'on s'attend à obtenir, considérant qu'on donne, à chaque ligne, une nouvelle définition de l-ind. La colonne de droite montre ce qu'on obtient à l'utilisation du texte affiche.

```
A[dimanche,20]          A[dimanche,20]
A[lundi]                A[lundi(manque des parametres)1..31]
A[1,1]                  A[(erreur de type: jour)1,1]
A[mardi,3,3,3]          A[mardi,3(trop de parametres)3,3]
A[hier,aujourd_hui]     A[(erreur de type: jour)hier,
                          (erreur de type: 1..31)aujourd_hui]
A[]                     A[(manque des parametres)jour,1..31]
```

On notera que, dans tous les cas d'erreur, il n'est indiqué qu'un message d'avertissement: l'évaluation de affiche signale les cas d'erreur, mais ne les refuse pas.

2.4. opérateurs de typage des textes

SYNTAXE

```
trm  ::= trm deft reft
     deft ::= <nom> <nom> env rep
     reft ::= <nom> <nom> env

atm  ::= atm uset
     uset ::= <nom> <nom> env
```

SEMANTIQUE

Les textes sont typés. A la définition, ou à l'emploi d'un texte, on fournit deux identificateurs:
- le premier est le nom du type,
- le deuxième est le nom de l'objet.

Une utilisation ou une référence typée à un texte typé nécessite que les deux noms de types soient égaux. Pour les autres cas:
- un emploi typé d'un texte non typé échoue – le texte n'est pas trouvé;
- un emploi non typé d'un texte typé réussit – le texte est trouvé;
- un emploi non typé d'un texte non typé réussit.

On notera donc que c'est à l'emploi, et non à la définition, qu'on décide si l'on souhaite réaliser un emploi typé ou non du texte.

EXEMPLE

```
deft typeX init
   = "X:=0"
deft typeX incr
   = "X:=X+1"
def z
   def init
      = "Y:=0"
   deft typeY incr
      = "Y:=Y+1"                    donne
   def a = (use init)          :    "Y:=0"
   def b = (use incr)          :    "Y:=Y+1"
   def c = (uset typeX init)   :    "X:=0"
   def d = (uset typeX incr)   :    "X:=X+1"
```

utilisation de a: l'utilisation non typée de init fournit la première définition visible de init – celle de "Y".

utilisation de b: de la même façon, et bien que le texte incr défini pour "Y" soit typé, on prend cette définition de incr.

utilisation de c: le texte init défini pour "Y" n'étant pas typé, il ne peut être pris en compte: on prend alors la définition relative à "X" – parce qu'aussi elle est du type attendu typeX.

utilisation de d: le texte incr défini pour "Y" n'est pas du type attendu typeX, et c'est encore celui défini pour "X" qui est pris.

2.5. résumé : la syntaxe complétée

On regroupe ici la *syntaxe concrète* et les opérateurs qu'on lui a ajouté.

```
(1)   env ::= ε | trm env | lst-env
§2.2  lst-env ::= ε | env lst-env
(2)   trm ::= def | ref | env | lsp | deft | reft
(3)       def  ::= <nom> env rep
(4)       ref  ::= <nom> env
§2.4      deft ::= <nom> <nom> env rep
§2.4      reft ::= <nom> <nom> env
(5)   rep ::= ε | atm rep
(6)   atm ::= stg | use | <string> | rep | lsp | lst-init | lst | tst
                | lst2-init | lst2 | tst2 | atm1 | atm2 | uset
(7)       stg  ::= <nom>
(8)       use  ::= <nom> env
§2.2      lst-init  ::= env rep
§2.2      lst       ::= ε | atm lst
§2.2      tst       ::= ε | sex tst
§2.3      lst2-init ::= env env env rep
§2.3      lst2      ::= ε | atm lst2
§2.3      tst2      ::= ε | sex tst2
§2.3      atm1      ::= ε | atm atm1
§2.3      atm2      ::= ε | atm atm2
§2.4      uset      ::= <nom> <nom> env
(9)   lsp ::= ε | sex lsp | fct
§2.1      fct  ::= sex
(10)  sex ::= trm | atm | <liste> | <atome>
```

Les numéros sont ceux de la *syntaxe Lisp*, une version déjà complétée de la *syntaxe concrète*, présentée au § 1.4.3.
Les numéros de paragraphes reportent aux paragraphes qui présentent les nouveaux opérateurs de la *syntaxe complétée*.

Chapitre 3.2:

Etude quantitative de l'évolution des programmes

Dans cette présente partie on illustre le phénomène d'évolution des programmes, et de l'expansion souvent inattendue et incontrôlée de certaines zones de code.
L'étude compare l'emploi des macro-définitions Lisp sur un exemple de 500 lignes de programme avec l'emploi des «textes»; d'où il ressort que dans ce second cas l'effort de généralisation est demandé plus tôt au programmeur mais qu'il est aussi plus durable.

1. Présentation

L'exemple choisi est le texte Lisp de l'évaluateur des textes. L'exemple est loin d'être totalement traité. En fait, on n'a retenu qu'une petite partie du texte, laquelle a l'avantage de mieux souligner les différences de comportement des «textes sources» lors de l'évolution du programme.

L'exemple traite des attributs d'opérateur de la syntaxe abstraite (cf. Chapitre 9.2, *«La Syntaxe Abstraite : Manuel du concepteur»*). Ce choix autorise à facilement introduire la notion d'évolution du programme: en effet, chaque nouvel opérateur défini correspond à un enrichissement de la sémantique du programme, et par là même à une évolution de celui-ci. L'exemple, qu'on peut juger un peu artificiel, a cependant l'avantage de permettre l'observation *in vitro* du phénomène d'évolution, qu'il serait nécessaire bien sûr d'adapter à l'évolution de zones de programme moins franchement découpées.

1.1. attributs

Les attributs d'opérateur sont au nombre de sept. Tous les attributs ne sont en fait pas traités:

- attribut d'*évaluation*: Cet attribut est spécifique de chaque opérateur; c'est justement par sa définition qu'on caractérise un nouvel opérateur, aussi n'y a-t-il que de faibles possibilités de mise en commun du code Lisp correspondant. Pour cette raison, et afin de ne pas gonfler les résultats par des valeurs qui resteraient constantes quelles que soient les représentations adoptées, cet attribut n'a pas été retenu.

- attribut de *recherche*: Ce "pseudo-attribut", qui ne s'applique qu'aux opérateurs ayant trait aux environnements de définition (env, isp, def, ref, lst-env), ne donnerait qu'une information partielle et n'a pas non plus été retenu.

- attribut de *complétion*: Celui-ci est représenté. Il est assez typique d'une partie de code "sans surprise", où la définition même de l'opérateur induit nécessairement celle de l'attribut. Cependant un procédé classique d'édition impose une définition nouvelle, sans un réel contrôle de cohérence, de cet attribut.

- attribut d'*impression complète*: Celui-ci est également représenté. Ici aussi la définition de l'attribut est relativement systématique, malgré quelques subtiles différences.

- attribut d'*impression concise*: Cet attribut étant dans sa forme très similaire au précédent, il n'a pas été traité.

- attribut d'*entrée* dans les champs de l'opérateur: Cet attribut est également traité. Pour chaque opérateur, il évolue très peu, mais suffisamment pour ne pouvoir donner qu'une unique définition.

- attribut de *sortie*: Semblable au précédent, il n'est pas traité non plus.

- attribut d'*impression évaluée*: Cet attribut, qui est davantage rattaché au phylum de l'opérateur qu'à l'opérateur lui-même, introduirait des effets parasites qui risqueraient de nuire à une bonne interprétation des résultats. Aussi n'est-il pas traité.

1.2. représentations

Les représentations des «textes sources» sont les suivantes:

- syntaxe expansée: C'est le produit final des autres représentations, c'est-à-dire le texte au complet.

- syntaxe avec les macros Lisp: C'est la syntaxe précédente, dans laquelle on a introduit des macros Lisp qui allègent le texte. Ces macros étant expansées une unique fois, on a bien après un premier passage l'équivalent de la syntaxe expansée, ceci pour ce qui concerne la place en mémoire occupée par le programme.

- syntaxe avec les macros Lisp, non compris les termes qui varient linéairement avec le nombre d'opérateurs définis: C'est la syntaxe précédente, dont on déduit les termes qui font "faussement" croître la taille du texte. Il s'agit: des en-têtes des fonctions Lisp, dont l'aspect systématique fait qu'elles ne contiennent aucune information réellement spécifique; des appels de macros, qu'on peut aussi considérer comme peu significatifs.

- représentation par les définitions de textes: Cette dernière représentation est celle qui nous intéresse. Elle est indépendante des macros Lisp définies auparavant. C'est ici au niveau de l'édition qu'on regroupe les termes apparemment voisins.
  La syntaxe précédente, où on néglige la composante linéaire, est introduite pour établir une meilleure comparaison avec la représentation par des textes. En effet cette dernière ne tient pas compte de la croissance linéaire du texte, parce qu'il s'agit dans ce cas de paramètres définis par ailleurs. Il faut donc bien, dans le cas de l'emploi de macros Lisp, supprimer ces termes pour mieux juger des différences entre ces deux formes de représentation.

1.3. remarque

En conclusion, on peut remarquer que la partie du programme traitée semble être justement celle pour laquelle la technique de représentation par des textes semble appropriée.
A ceci on peut donner deux réponses. La première est que, s'agissant d'un exemple qui cherche à valider l'emploi d'une telle technique, il est raisonnable de présenter un exemple qui "marche bien". La seconde, plus fondamentalement, est que cette technique ne doit peut-être pas s'appliquer à la totalité d'un texte source: elle concerne de façon privilégiée ces zones de code où, tant sur le plan conceptuel qu'algorithmique, l'intérêt est très limité parce que leur contenu est répétitif et systématique. Malheureusement dans tout programme on a toujours de ces semblables zones, qui sont souvent vite réalisées et ont une tendance fâcheuse à croître rapidement. On obtient alors un programme dont le cœur du programme est une partie très restreinte et très soignée, mais autour duquel gravitent de nombreuses zones obscures, répétitives et volumineuses, et qui seront justement ces endroits d'où naîtront les problèmes.

2. Résultats

2.1. le «texte utile»

figure 1: taille du «texte utile»
figure 2: pente d'évolution du «texte utile»

Le «texte utile» est, d'une façon informelle, le texte du programme qui reflète les aspects *spécifiques* de chaque opérateur – par opposition aux parties du texte répétées avec de faibles changements, qui représentent la «composante linéaire» du texte.

Une première comparaison concerne les courbes (a), (b) et (c). La courbe (b) évolue légèrement moins vite que (a), ce qui signifie qu'en introduisant les macros Lisp, on réduit la taille du code, ce qui est heureux. La courbe (c) évolue moins vite que (b), ce qui n'est pas étonnant puisque les points de (c) représentent des points de (b) auxquels on soustrait une certaine quantité. Il est intéressant de constater cependant que, une fois supprimée la «composante linéaire», on observe des plateaux sur la courbe (c), qui signifient qu'à certaines étapes de l'évolution du programme le «texte utile» conserve une forme stable.

Une deuxième comparaison est à faire entre les courbes (c) et (d). Le fait que (d) de termine à un point nettement inférieur à celui de (c) n'est pas révélateur d'une plus grande faculté de concision, puisqu'on compare ici deux objets structurellement très différents – (c) est du texte source classique, donc une suite linéaire de lignes de code, alors que (d) est la "représentation plane" d'un arbre, soit une information structurée. En revanche on peut constater que (d) croît très vite, dès le début, et se stabilise ensuite. Ce point me paraît plus important, puisqu'il signifie que, avec le choix d'une représentation par les définitions de texte, on s'impose au départ d'être plus prolixe, mais on atteint alors un état beaucoup plus stable vis-à-vis de l'évolution du programme. C'est-à-dire qu'après avoir passé en revue la plupart des cas particuliers – jusqu'à l'opérateur Ist environ – on réutilise systématiquement le travail antérieur, d'une façon ou d'une autre.

2.2. le texte à croissance linéaire

figure 3: taille du texte à croissance linéaire
figure 4: pente d'évolution du texte à croissance linéaire

Comme il a été dit, il s'agit pour le texte construit avec les macros Lisp des en-têtes de fonctions et des appels de macros. Pour la représentation par les définitions de texte, il s'agit des paramètres de définition des opérateurs, et de l'utilisation (au sens de l'utilisation de texte) des définitions de texte.

La comparaison paraît être ici très nettement en faveur de l'emploi de macros Lisp. Il faut cependant bien interpréter cette «composante linéaire» dans les deux cas. Avec les macros Lisp, courbe (b), il s'agit des en-têtes de fonction et des appels de macros, soit des éléments sémantiquement peu parlants. Ils ne sont pourtant pas vides de sens, c'est-à-dire qu'ils peuvent aussi subir des évolutions avec le programme – par exemple on modifie les paramètres d'une macro. Or ces éléments sont généreusement répartis dans le texte du programme, et c'est donc pas à pas qu'il faudra les retrouver lors de toute modification. A l'inverse, avec les définitions de texte, courbe (a), il

s'agit pour l'essentiel de paramètres: leurs définitions peuvent être "physiquement" regroupées à un point précis du programme, et, même si la taille du code correspondant est plus grande qu'avec les macros Lisp, elles demeurent bien moins dispersées.

Il faudrait encore remarquer ici qu'on compare deux informations bien différentes: des bouts de code dispersés ou une information structurée qui s'apparente davantage dans ce cas à un tableau de déclarations.

2.3. le «texte utile» normalisé

figure 5: taille du «texte utile», ramenée à l'unité

On reprend ici les données de la figure 1, mais les courbes sont ici normalisées pour mieux observer le comportement relatif de chaque cas.

On peut constater que les courbes (a) et (b) sont très voisines, ce qui signifie que la «composante linéaire» n'est pas un facteur déterminant, et très proches de la diagonale, ce qui signifie que, malgré l'introduction des macros Lisp, la taille du code n'est pas loin de croître régulièrement avec la complexité du programme.

On voit en revanche que la courbe (c) est très nettement au-dessus de la diagonale, ce qui signifie que l'effort à fournir est demandé plus tôt mais qu'il est également plus durable.

3. Conclusion

Pour conclure, il n'est pas inutile de parler d'un aspect important de la programmation, tant à la construction qu'à la maintenance des programmes. Il s'agit de la **lisibilité** des textes sources.

Dans l'exemple présenté, l'emploi des macros Lisp est resté dans la limite du raisonnable, c'est-à-dire du lisible. On aurait pu bien sûr imaginer une unique macro Lisp qui tiendrait compte de tous les aspects spécifiques de chaque opérateur, mais on obtiendrait alors certainement un texte très difficile à relire. LeLisp d'ailleurs offre en standard une fonction spéciale qui permet d'expanser une macro sans l'évaluer, tant il est vrai que le résultat souhaité à l'expansion d'une macro n'est pas toujours celui qui est obtenu.

Avec des définitions de texte, la "macro Lisp" est écrite peu à peu, et toujours en gardant *en vue* la forme expansée de cette macro, ce qui est un gain de temps et de sécurité appréciable, surtout si l'on veut appliquer la technique à un programme de taille importante. Ici, 500 lignes représentent, à l'échelle d'une macro, une taille importante.

Carthago delenda est.

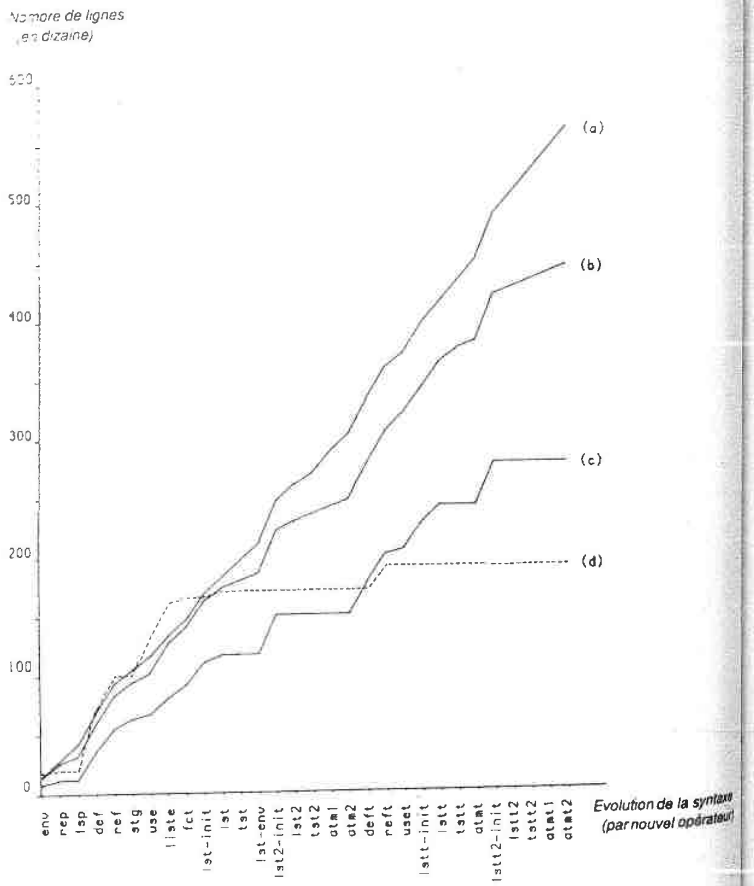


figure 1 : taille du «texte utile»

a : syntaxe expansée
b : syntaxe avec les macros Lisp
c : syntaxe avec les macros Lisp, non compris les termes qui varient linéairement avec le nombre d'opérateurs définis
d : représentation par les définitions de textes

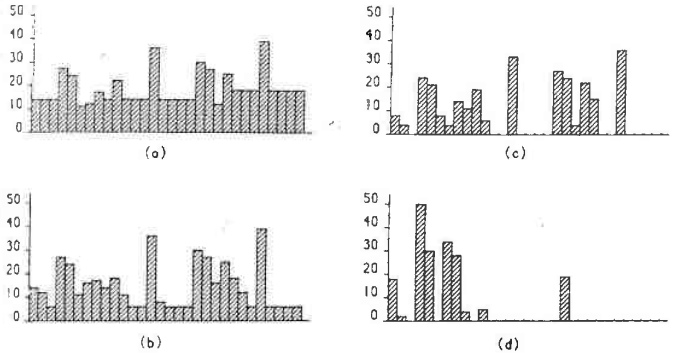


figure 2 : pente d'évolution du «texte utile»

a : syntaxe expansée
b : syntaxe avec les macros Lisp
c : syntaxe avec les macros Lisp, non compris les termes qui varient linéairement avec le nombre d'opérateurs définis
d : représentation par les définitions de textes

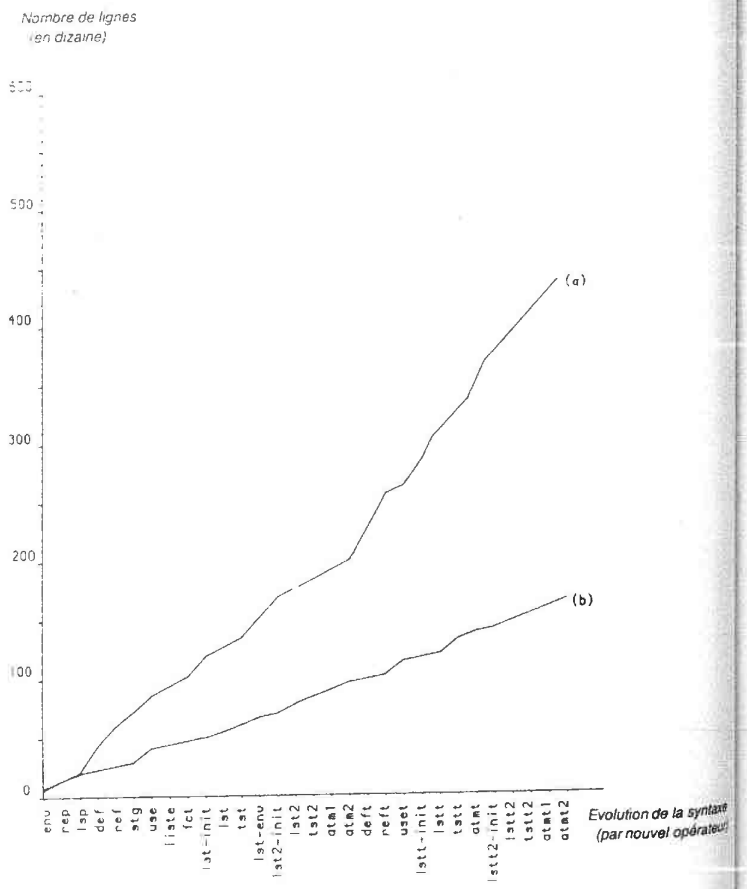


figure 3 : taille du texte à croissance linéaire

a : représentation par les définitions de textes

b : syntaxe avec les macros Lisp, non compris les termes qui varient linéairement avec le nombre d'opérateurs définis

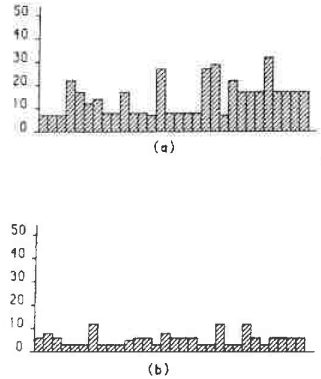


figure 4 : pente d'évolution du texte à croissance linéaire

a : représentation par les définitions de textes

b : syntaxe avec les macros Lisp, non compris les termes qui varient linéairement avec le nombre d'opérateurs définis

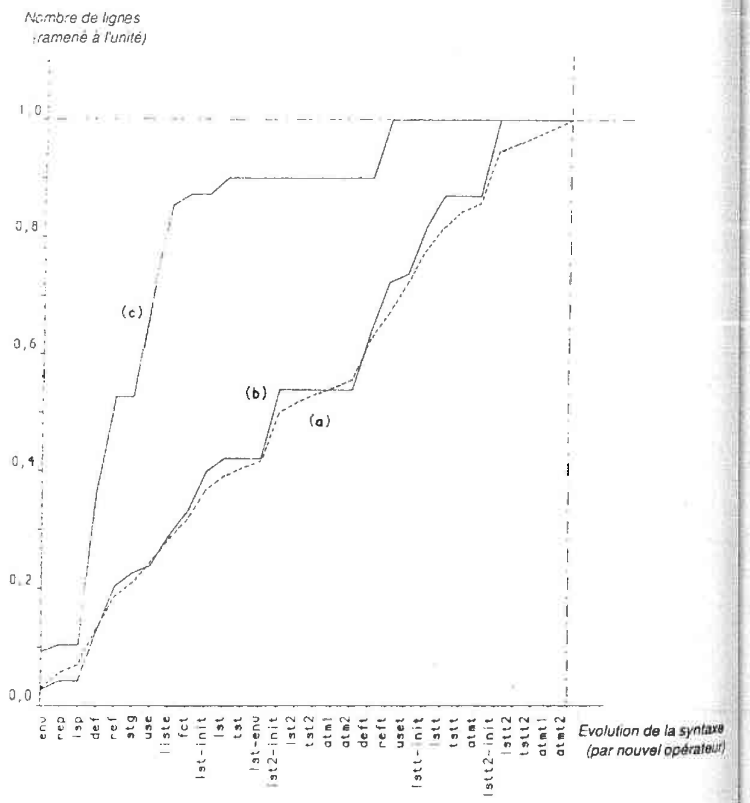


figure 5 : taille du «texte utile», ramenée à l'unité
a : syntaxe avec les macros Lisp
b : syntaxe avec les macros Lisp, non compris les termes qui varient linéairement avec le nombre d'opérateurs définis
c : représentation par les définitions de textes

Chapitre 3.3:

L'édition syntaxique

L'édition syntaxique est un sujet d'étude qui a déjà été largement exploré et dont les techniques de mise en œuvre sont passées au milieu «industriel» – Mentor [MMV 85], CPS [ReT 85], ALOE [MeN 81], ... On montre ici comment elle s'introduit dans le «monde des textes», et quelles propriétés particulières on offre dans ce cadre.

Il s'agit d'un exemple assez complet d'utilisation de la Syntaxe Complétée, qui illustre les fonctionnalités propres à l'outil proposé. Elles sont le fait de l'utilisation d'une représentation «textuelle» des opérateurs du langage de programmation utilisé sous l'éditeur:

- on peut mettre à profit le *comportement dynamique* des «textes» pour assurer sur le programme en cours d'édition des contrôles non textuels – contrôle de type, ...;
- l'homogénéité des représentations de la grammaire du langage utilisé et de l'arbre syntaxique du programme construit permet l'introduction de nouveaux "opérateurs", soit pour prendre en compte une information de contexte, soit pour définir un certain schéma d'instructions du langage;
- la distance relative entre la grammaire du langage et le programme permet d'utiliser divers schémas de décompilation, et en particulier un schéma de décompilation qui fournisse non plus du texte source mais le résultat de interprétation du programme.

En conclusion, on suggère un amélioration du formalisme d'expression des «textes» qui permettrait de traduire les spécificités d'un langage comme des propriétés d'Objets qui définiraient ce langage.

1. Définition d'un langage

On illustre la démarche par un exemple. Les figures auxquelles il est fait référence se trouvent en annexe.

1.1. la grammaire du langage
(figure 1.1)

On donne la grammaire du langage de programmation choisi pour l'exemple, dans une notation voisine de la notation BNF, et un exemple de programme «syntaxiquement correct».

Par exemple: l'affectation

<affect> ::= <VAR> := <EXP>

Le terme <affect> se dérive en:
- le terme <VAR> (terminal instanciable du langage),
- le symbole ":=",
- le terme <EXP> (terminal instanciable).

1.2. la syntaxe abstraite et les schémas de décompilation
(figure 1.2)

syntaxe abstraite
(figure 1.2.a)

On définit la syntaxe abstraite du langage qui va permettre de définir un éditeur spécifique à ce langage. La syntaxe d'expression choisie est voisine de celle définie par le langage Metal, utilisé par Mentor: les phyla sont des identificateurs écrits en majuscule, les opérateurs en minuscule. Les phyla «prédéfinis» correspondent à des terminaux instanciables pour lesquels on ne précise pas davantage la définition.

Par exemple:

```
INSTR ::= affect affiche cond
```

Le phylum INSTR (des instructions) regroupe les opérateurs affect (affectation), affiche (affichage) et cond (phrase conditionnelle).

```
affect -> VAR EXP
```

L'opérateur affect est défini par deux opérateurs, lesquels doivent appartenir aux phyla VAR (les variables) et EXP (les expressions) respectivement.

schémas de décompilation
(figure 1.2.b)

La définition précédente de la syntaxe abstraite est insuffisante pour réaliser l'affichage d'un programme; on donne donc, en plus de celle-ci, les schémas de décompilation des opérateurs.

Par exemple:

```
affect(VAR,EXP) =  VAR := EXP
```

L'opérateur affect, paramétré par deux termes symbolisés ici par les méta-variables VAR et EXP, se décompile en:
- la forme décompilée du paramètre VAR,
- la symbole ":=",
- la forme décompilée du paramètre EXP.

arbre syntaxique
(figure 1.2.c)

On donne l'arbre syntaxique associé au programme précédent.

Par exemple:

```
INSTR = affect
        VAR = "x"
        EXP = "y"
```

On choisit dans le phylum INSTR l'opérateur affect, en fournissant les "paramètres":
- VAR, un terminal instanciable auquel on donne la valeur "x",
- EXP, avec la valeur "y".

1.3. la représentation textuelle
(figure 1.3)

grammaire
(figure 1.3.a)

On place dans l'«environnement global» la syntaxe du langage – c'est-à-dire au vu de tous les textes. On peut noter que:
- les opérateurs du langage donnent lieu à la définition d'un texte de même nom;
- les phyla du langage, qui particularisent les paramètres des opérateurs, apparaissent comme des utilisations indéfinies de textes, dans la représentation de la définition des opérateurs qu'ils concernent.

Par exemple:

```
(def affect ()
   ((VAR) ":=" (EXP) "^M"))  ; Note : "^M" = «retour à la ligne»
```

On définit le texte affect:
- qui utilise deux textes VAR et EXP (à définir par ailleurs),
- dont la valeur de représentation correspond à ce qui précédemment s'appelait le schéma de décompilation de l'opérateur.

arbre syntaxique
(figure 1.3.b)

On place un dernier "opérateur" dans l'environnement global: il s'agit du programme en cours d'édition; on le définit ici dans un texte nommé PGME qui sera associé à un nouveau tampon de l'éditeur.

Par exemple:

```
(def INSTR
   ((def VAR () ("x"))
    (def EXP () ("y")))
   ((affect)))
```

On définit le texte INSTR par:
- sa valeur de représentation: il utilise le texte affect;
- sa valeur d'environnement: il définit les paramètres de l'"opérateur" affect, c'est-à-dire ici les textes VAR et EXP.

1.4. la concentration des variables
(figure 1.4)

Dans l'exemple on reconnaît plusieurs occurrences d'identificateurs sémantiquement liés: d'une part "x" et d'autre part "y". Dans ces deux cas il s'agit bien d'un même concept – une variable – qu'on utilise plusieurs fois: une fois pour la déclaration, les autres fois comme utilisation d'une variable déclarée. Comme il s'agit ici de la valeur de terminaux instanciables du langage, il n'y a pas de liens particuliers entre ces diverses occurrences de "x" ou de "y".
On définit alors deux nouveaux "opérateurs", d'arité nulle, nom-x et nom-y, qu'on place également dans l'environnement global. Le tampon d'édition PGME utilise alors ces définitions au lieu de donner explicitement leur représentation (qui est respectivement "x" et "y").

Par exemple:

```
(def nom-x () ("x"))

(def INSTR
   ((def VAR () ((nom-x)))
    (def EXP () ((nom-y))))
   ((affect)))
```

Le texte VAR n'est plus défini comme «la chaîne de caractères "x"» mais comme l'«utilisation de l'"opérateur" nom-x»: on a bien *concentré* dans la définition de ces nouveaux textes les occurrences d'utilisation de ces identificateurs.

1.5. la concentration des propriétés
(figure 1.5)

On observe qu'une variable a en fait une sémantique plus riche que sa seule représentation: son utilisation est toujours étroitement liée au *type* sur lequel elle a été définie. Plus généralement certains concepts du programme seront définis par un regroupement de propriétés; on utilisera alors l'environnement de définition pour lier ces propriétés, la référence à un environnement pour y accéder.

Par exemple, on définit ici:

```
(def var-x
   ((def nom () ("x"))
    (def typ () ("integer"))
    (def init () ("0"))))

(def INSTR
   ((def VAR () ((nom((var-x)))))
    (def EXP () ((nom((var-y))))))
   ((affect)))
```

La variable var-x est définie par ses trois propriétés: son nom (nom), son type (typ) et sa valeur d'initialisation (init). Le texte VAR utilise la propriété nom de l'objet var-x.

1.6. la paramétrisation des propriétés
(figure 1.6)

En fait dans le choix précédent de représentation on reste trop près du langage et de ses spécificités: ici on *sait* que dans le langage on représente de la même façon une variable utilisée soit pour accéder à sa valeur soit pour en affecter la valeur.
Le problème des répétitions dans le texte n'est encore pleinement résolu, mais les répétitions sont maintenant plus diffuses. Par exemple une déclaration de variable utilise toujours les propriétés nom et typ de la variable déclarée; l'affectation ne s'intéresse pas tant au nom d'une variable qu'à sa possible propriété de se voir affecter une valeur; etc...

On définit donc un *modèle* pour les variables:

```
(def var
   ((def lect () ((NOM)))
    (def ecr () ((NOM)))
    (def init () ((INIT)))))
```

Le texte var:
- est paramétré par trois textes: NOM, TYP, INIT,
- définit pour les variables les propriétés: lect, ecr, init.

Chaque variable est alors déclarée en référence à ce modèle, et les opérateurs de la grammaire peuvent exploiter ce point – ils savent d'une part quelles sont les propriétés des «variables», ils savent d'autre part quels paramètres elles ont définis à leur déclaration.

Par exemple:

```
(def VAR-x
   ((def NOM () ("x"))
    (def TYP () ("integer"))
    (def INIT () ("0"))
    (var)))

(def INSTR
   ((def VAR ((var-x)) ((ecr)))
    (def EXP ((var-y)) ((lect))))
   ((affect)))
```

La variable "x": VAR-x est construite sur le *modèle* var.
A l'utilisation de affect, on utilise les propriétés des variables:
- la propriété d'écriture ecr de la variable VAR-x,
- le propriété de lecture lect de la variable VAR-y.

1.7. le poids sémantique des propriétés
(figure 1.7)

L'introduction des "propriétés" que doivent satisfaire certains paramètres autorise l'établissement de contrôles sémantiques. On montre ici par quel moyen on peut effectuer le contrôle de type dès la phase d'édition. On le réalise sans pénalisation pour le programmeur, puisqu'il se traduit par d'éventuels messages de mise en garde mais qu'il ne contraint pas le programmeur à corriger son erreur.

Par exemple:

```
(def affect ()
   ((VAR) ":="
       (lsp
          (unless (egal (use TYP((VAR))) (use TYP((EXP))))
                  "{erreur de type}"))
       (EXP) "^M"))
```

qui retourne l'affichage:

```
x:={erreur de type}y
```

On notera que le programme – le texte PGME – n'a pas été retouché: le contrôle de type est réalisé dans la grammaire et non dans le programme. On choisit donc lors d'une session sous l'éditeur d'activer ou non le contrôle de type par le choix de la grammaire de décompilation – ou plus précisément par le choix de la liste des textes qui définissent les opérateurs de la grammaire.

2. Information de contexte

On présente plusieurs exemples d'"opérateurs" dont le comportement est lié au *contexte* dans lequel on les utilise – il s'agit soit d'opérateurs du langage soit de textes qu'on aura identifiés comme représentant des concepts significatifs du langage.

2.1. la décompilation contextuelle

Les opérateurs du langage sont définis comme des textes placés dans l'«environnement global»; la prise en compte d'information de contexte se traduit donc simplement par la redéfinition de l'opérateur concerné dans un environnement local.

2.1.1. le «else pendant»

L'exemple classique d'un opérateur dont le schéma de décompilation dépend du contexte est celui du «else pendant» de Pascal. On définit ici les opérateurs de phrase conditionnelle suivants:

• l'opérateur cond1

```
(def cond1 ()
   ("if " (TEST) " then" "^M"
    "  " (ALORS) "^M"
    (lsp
       (if  (egal (use SINON) "")
            (rep  "else")
            (rep  "else" "^M"
                  "   " (SINON))))))
```

On affiche systématiquement le mot-clé "else", qu'il y ait ou non une «partie SINON».

• l'opérateur cond2

```
(def cond2 ()
   ("if " (TEST) " then" "^M"
    (lsp
       (if  (egal (use SINON) "")
            (rep  "  " (ALORS))
            (rep  "  " (ALORS
                        ((def cond () ((cond1))))) "^M"
                  "else" "^M"
                  "   " (SINON))))))
```

Dans cond2:
- si la «partie SINON» est vide: on affiche simplement la «partie ALORS» (en particulier on n'affiche pas de "else");
- si elle est non vide:
  - on affiche la «partie ALORS», en redéfinissant l'opérateur cond sur l'opérateur cond1,
  - on affiche ensuite la «partie SINON» simplement.

• l'opérateur cond

```
(def cond ()
   ((cond2)))
```

L'opérateur cond est l'opérateur utilisé dans l'arbre syntaxique du programme:
- initialement on l'identifie à cond2: si la «partie SINON» est vide, on ne veut pas voir s'afficher un "else" inutile;
- quand on l'utilise, la «partie SINON» conserve la propriété de ne pas afficher les "else" inutiles; en revanche la «partie ALORS», qui redéfinit l'opérateur cond sur cond1, va forcer cet affichage.

Par exemple:

forme «textuelle» *forme «visuelle»*

On n'affiche aucun "else":

```
(def UTIL1                                  if x=y then
  ((def TEST () ("x=y"))                      if y=z then
   (def ALORS                                   x:=y
     ((def TEST () ("y=z"))
      (def ALORS () ("x:=y"))
      (def SINON () ()))
     ((cond)))
   (def SINON () ()))
  ((cond)))
```

Sur la «branche ALORS», l'existence d'une «partie SINON» force l'affichage du "else" (la règle de syntaxe de Pascal rapporte la «partie SINON» introduite par "else" à la phrase conditionnelle la plus proche pour laquelle ne se rapporte encore aucune «partie SINON»):

```
(def UTIL2                                  if x=y then
  ((def TEST () ("x=y"))                      if y=z then
   (def ALORS                                   x:=y
     ((def TEST () ("y=z"))                   else
      (def ALORS () ("x:=y"))               else
      (def SINON () ()))                      y:=x
     ((cond)))
   (def SINON () ("y:=x")))
  ((cond)))
```

Dans la «partie SINON de la partie ALORS» il n'y a pas nécessité d'affichage du "else":

```
(def UTIL3                                  if x=y then
  ((def TEST () ("x=y"))                      if y=z then
   (def ALORS                                   x:=y
     ((def TEST () ("y=z"))                   else
      (def ALORS () ("x:=y"))                   if x=z then
      (def SINON                                  write('Erreur')
        ((def TEST () ("x=z"))
         (def ALORS () ("write('Erreur')"))
         (def SINON () ()))
        ((cond)))
     ((cond)))
   (def SINON () ()))
  ((cond)))
```


Dans cette même partie on affiche le "else": cette propriété est "héritée" sur la «branche SINON» du deuxième niveau de la «branche ALORS» du premier niveau:

```
(def UTIL4                                  if x=y then
  ((def TEST () ("x=y"))                      if y=z then
   (def ALORS                                   x:=y
     ((def TEST () ("y=z"))                   else
      (def ALORS () ("x:=y"))                   if x=z then
      (def SINON                                  write('Erreur')
        ((def TEST () ("x=z"))                  else
         (def ALORS () ("write('Erreur')"))   else
         (def SINON () ()))                     y:=x
        ((cond)))
     ((cond)))
   (def SINON () ("y:=x")))
  ((cond)))
```

Il ne faut cependant pas oublier de redéfinir l'opérateur cond par sa forme initiale (sur cond2) dans une sous-branche où l'ambiguïté disparaît. Par exemple, l'opérateur repeat:

```
(def repeat ()
   ("repeat" "^M"
    " " (INSTR
          ((def cond () ((cond2))))) "^M"
    "until " (EXP-COND)))
```

Le parenthésage implicite défini par l'opérateur repeat fait "oublier" le contexte d'apparition de cet opérateur.

2.1.2. groupement d'instructions

On peut aussi avoir des ambiguïtés d'affichage introduites à la définition d'"opérateurs" propres à l'utilisateur. Par exemple, on définit l'"opérateur":

```
(def lecture ()
   ("carcou := carlu;" "^M"
    "read(carlu)"))
```

qui est utilisé dans le programme comme une instruction indivisible.

Selon la position de l'"opérateur" dans le programme, l'affichage sera différent. Par exemple:

```
[1] begin
      ...
      carcou := carlu;
      read(carlu);
      Traitement(carcou);
      ...
    end;
```

```
[2] while condition do
        begin
            carcou := carlu;
            read(carlu);
        end;
```

dans le cas **[1]**, les deux instructions sont affichées "en ligne"; dans le cas **[2]** on les encadre par les parenthèses begin-end.

On peut systématiquement se placer dans le cas **[2]**, mais ceci risque d'entraver une bonne lisibilité du programme. Pour conserver les deux formes possibles d'affichage on introduit alors un opérateur de groupement d'instructions qui se chargera de la prise en compte du contexte d'utilisation de l'opérateur.

```
(def liste-instr1 ()
   ("begin" "^M"
    " " (lst-init ((INSTR*))
          ((INSTR) ";" "^M"))
    "end;" "^M"))

(def liste-instr2 ()
   ((lst-init ((INSTR*))
       ((INSTR) ";" "^M"))))

(def liste-instr ()
   ((liste-instr2)))
```

L'"opérateur" liste-instr est défini par défaut sur liste-instr2: on n'affiche pas les parenthèses begin-end. Certains opérateurs redéfinissent alors liste-instr. Par exemple:

```
(def while
   ("while " (EXP-COND) " do" "^M"
    " " (INSTR
          ((def liste-instr () ((liste-instr1)))))))
```

L'"opérateur" lecture est défini sur liste-instr: il laisse à ce dernier le soin de décider l'affichage ou non des parenthèses begin-end.

```
(def lecture
   ((def INSTR*
       (lst-env
        ((def INSTR () ("carcou := carlu")))
        ((def INSTR () ("read(carlu)")))))
   ((liste-instr))))
```


2.2. les opérateurs contextuels

Une grammaire est très généralement exprimée sous la forme d'une grammaire «indépendante du contexte» ("context-free"). En pratique, la grammaire d'un langage n'a quasiment jamais cette propriété. Mais on en fait pourtant largement usage, pour la réalisation des outils qui travaillent sur une grammaire (analyseur syntaxique, compilateur, éditeur syntaxique, ...).

Des exemples d'opérateurs contextuels:

- Dans la définition d'un sous-programme: la «phrase RETURN» qui est la phrase par laquelle on indique qu'on termine l'exécution du sous-programme et quelle est la valeur retournée dans le cas d'une fonction; l'«appel récursif» qui se représente par un appel de sous-programme mais qui a une sémantique beaucoup plus riche.

- Dans une boucle: la «phrase EXIT» qui indique qu'on sort de la boucle – éventuellement en la nommant.

- Dans un bloc de traitement: la «levée d'exception» qui signale une exception à l'appelant – ce dernier point est vrai en LTR3, où un identificateur d'exception suit les règles classiques de visibilité; il est faux en Ada: l'exception "remonte" dans la pile des environnements d'appel, sans barrière de visibilité.

le cas de la fonction

grammaire
(figure 2.1)

Pour simplifier la présentation, on n'introduit pas la notion de liste. De ce fait:
- une fonction est définie avec un unique paramètre,
- le corps de la fonction est composé d'une unique instruction,
- etc...
La prise en compte des listes compliquerait l'exemple mais ne le contredirait pas.

- L'opérateur fonction
  Il est défini par:

```
(def fonction
   ((def return ...«phrase RETURN»...)
    (def call-rec ...«appel récursif»...))
   (...déclaration de la fonction...))
```

  c'est-à-dire que:
  - sa valeur (de représentation) est la déclaration de la fonction,
  - il définit localement les opérateurs return et call-rec: ceux-ci seront alors accessibles dans la déclaration de la fonction, et uniquement là.

- L'opérateur call-fct
  C'est l'appel d'une fonction, formellement paramétré par:
  - BLOC-FCT: un «bloc de fonction», duquel on attend la propriété NOM,
  - EXP: une expression qui représente le paramètre effectif de l'appel.

la fonction compte
(figure 2.2)

La définition d'une fonction introduit celle de «bloc de fonction»: c'est le bloc des déclarations de la fonction, qui définit:
- le nom de la fonction et le type de valeur retournée,
- les paramètres formels (leur nom, leur type, leur sens),
- les déclarations locales.

Partant, on pourra ensuite utiliser ces déclarations "assez profondément" dans l'arbre syntaxique du corps de la fonction. La difficulté est alors de savoir nommer ces déclarations "globales" – "globales" vis-à-vis du corps de la fonction.

Pour particulariser les déclarations "globales" de la fonction, on structure la définition «textuelle» de la fonction en plaçant les paramètres de définition d'une fonction *à l'intérieur* d'un texte de nom bloc-fct:

```
(def fct-compte
   ((fonction)
    (def bloc-fct
       ((def NOM ...)
        (def TYP ...)
        (def PARAM ...)
        ...déclarations locales...
        (def INSTR ...))))
   ((fonction)))
```

On définit le texte fct-compte: la fonction compte:
- en faisant référence à l'opérateur fonction; il donne la visibilité des opérateurs return et call-rec;
- en fournissant, dans le texte bloc-fct, les paramètres:
  - NOM: le nom de la fonction,
  - TYP: le type de la valeur retournée,
  - PARAM: la liste des paramètres formels (ici: le paramètre formel), définis par:
    - NOM: le nom du paramètre formel,
    - TYP: son type,
    - SNS: son sens (IN, OUT, INOUT, ...),
  - des définitions locales de textes,
  - INSTR: la liste des instructions du corps (ici: l'instruction du corps).

Dans le cas de la fonction compte, on définit localement le texte param-arbre:
- vis-à-vis du corps de la fonction il s'agit d'une variable locale;
- vis-à-vis de l'en-tête de la fonction, le paramètre PARAM est défini par *référence* au texte param-arbre: il s'agit donc d'un paramètre formel.

Un appel de fonction consiste alors en la fourniture d'un «bloc de fonction». Par exemple:

```
(def TEST
   ((def BLOC-FCT
       ((bloc-fct   ((fct-feuille)))))
    (def EXP ...))
   ((call-fct)))
```

on fournit le «bloc de fonction» de la fonction feuille.

Un appel récursif se ramène à fournir le «bloc de fonction» implicitement visible dans la déclaration de la fonction – le sien:

```
(def call-rec
   ((def BLOC-FCT
       ((bloc-fct))))
   ((call-fct)))
```

(le paramètre EXP est fourni par défaut: call-rec est lui-même paramétré par EXP).

les fonctions sur les arbres
(figure 2.3)

On reconnaît dans la définition «textuelle» des fonctions sur les arbres le même schéma de construction du texte – renforcé ici par le fait que le paramètre formel est toujours un arbre en entrée (mode "IN"): mais ceci est en fait un "hasard".

2.3. les schémas contextuels

On reprend ici l'exemple du bloc de réservation, présenté auparavant (cf. Chapitre 2.4, *«Exemple de structuration des traitements»*). Le bloc de réservation est un "opérateur" défini par l'utilisation qui va *connoter* les phrases de «sortie brutale»: la «phrase EXIT» ou la «phrase RETURN» (on ne traite pas cette dernière).

- le texte while

```
(def while
   ((def exit1 ()
       ("EXIT " (NOM((bloc-while))) ";" "^M")))
   ("while " (EXP-COND) " do /* (NOM((bloc-while))) "/" "^M"
    " " (lst-init ((INSTR*))
          ((INSTR
             ((def exit () ((exit1)))))))
    "END DO;" "^M"))
```

pour accéder au nom de la boucle on introduit un «bloc de boucle» bloc-while; les instructions sont décompilées en définissant le texte exit par sa forme simple exit1.

- le texte reserve

```
(def reserve
   ((def exit2 ()
       ("free(" (NOM((bloc-reserve))) ");" "^M"
        (exit1))))
   ("reserve(" (NOM((bloc-reserve))) ");" "^M"
    " " (lst-init ((INSTR*))
          ((INSTR
             ((def exit () ((exit2)))))))
    "free(" (NOM((bloc-reserve))) ");" "^M"))
```

on introduit un «bloc de réservation» bloc-reserve pour nommer la ressource concernée. Les instructions sont alors décompilées en définissant le texte exit par sa forme complétée exit2.

Note: en toute rigueur il faudrait dans reserve introduire un test qui vérifie s'il y a lieu de définir un texte exit, c'est-à-dire qui puisse savoir si l'on est placé *à l'intérieur* d'une boucle (texte while).

• le "corps" du bloc de réservation

```
(def INSTR
   ((def EXP-COND () ("R.c<R.P"))
    (def ALORS
       (lst-env
       ((def INSTR () ("incr(R.c);" "^M")))
       ((def INSTR () ((exit))))))
    (def SINON
        (lst-env)))
   ((cond)))
```

on utilise l'"opérateur" exit, en "oubliant" qu'on est placé à l'intérieur d'un bloc de réservation, et qu'il faut donc libérer la ressource: c'est le texte exit qui se le rappellera.

3. Exécution

Le procédé de décompilation des arbres syntaxiques – l'évaluation d'un texte – et le langage – Lisp – suggèrent l'introduction d'une nouvelle forme de décompilation: l'exécution du programme.
On reprend ici l'exemple du langage (§ 1) et celui des fonctions (§ 2) en vue d'une interprétation effective du programme en cours d'édition.

3.1. l'exécution

précisions de l'exemple
(figure 3.1)

Si l'on souhaite réaliser une exécution symbolique du programme, il faut davantage préciser l'exemple. Cela touche deux points:
• on définit «textuellement» les types utilisés dans le programme (type-int et type-real), qui présentent maintenant les deux propriétés:
  - nom : le nom du type,
  - conv : le schéma de conversion d'une chaîne de caractères (formellement nommée STG) en une valeur du type;
• le type d'une variable (champ TYP) est défini par référence à un texte de type – type-int ou type-real; les "valeurs immédiates" "0" ou "1.0" sont définies comme la conversion d'une chaîne de caractères en une valeur du type (utilisation de la propriété conv du type).

Les points précédents ne font que mieux préciser la définition du langage et son utilisation. La seule véritable modification intervient dans l'introduction d'un nouveau champ à la déclaration d'une variable: le champ VAL, dont la représentation a une valeur indéfinie; il est utilisé à l'exécution pour stocker la valeur de la variable.

les nouveaux termes du langage

Pour éviter des insertions fréquentes de code Lisp par l'entremise de l'opérateur lsp, on introduit trois nouveaux opérateurs: exec, clean et funclean. Pour alléger encore un peu plus la syntaxe, et illustrer la possibilité de définition d'un phylum, on introduit un nouveau phylum: EXEC.

Syntaxe

```
EXEC ::= exec clean funclean
    exec      -> SEX*...
    clean     -> NOM ENV REP
    funclean -> NOM ENV REP
```

• phylum EXEC
  C'est le phylum des exécutions, fils du phylum des représentations REP: là où est attendue une représentation on pourra donc trouver un opérateur d'exécution – et non réciproquement.

• opérateur exec
exec est à EXEC ce que lsp que est à LSP: l'évaluation de l'opérateur correspond à l'évaluation séquentielle des S-expressions qui le composent (c'est "progn" en Lisp).

• opérateur clean
clean correspond à l'utilisation "propre" use: son évaluation est élaborée d'une manière identique à celle de use, mais la valeur retournée est "nettoyée" des préfixes de représentation rep.
Par exemple: use NOM retourne (rep (rep "nom-x"))
clean NOM retourne "nom-x"

• opérateur funclean
funclean est l'application:
- sur l'argument évalué du champ REP
- de la lambda-expression trouvée sous le nom NOM dans l'environnement local ENV.
On peut donc écrire ceci:
(funclean NOM ENV REP) ≡ (funcall (clean NOM ENV) REP)
L'argument REP est évalué en-dehors de la lambda-expression. A l'application de la lambda-expression il est construit une "sorte" de fermeture lexicale grâce à l'environnement ENV.
Pour que l'évaluation fournisse un résultat correct, il faut bien sûr que l'utilisation du texte NOM retourne une lambda-expression: ceci n'est pas testé à l'évaluation.

la grammaire d'exécution
(figure 3.2)

On redéfinit les opérateurs du langage en modifiant leur valeur de représentation. Celle-ci retourne maintenant un résultat qui appartient au «monde Lisp» (en particulier conv-int et conv-real sont les conversions lexicales de chaînes de caractères en entiers et réels respectivement).
La "décompilation" du programme dans cette grammaire fournit le résultat de son évaluation symbolique. Ici, il s'affiche:

```
'Erreur'
'fin'
```


3.2. le prototypage

On reprend l'exemple des fonctions.

précision de l'exemple
(figure 3.3)

Ici aussi on précise mieux l'exemple:
- les types,
- les emplois de type,
- la somme: on définit l'opérateur somme.

la grammaire d'exécution
(figure 3.4)

On construit la grammaire d'exécution:
- les types et opérateurs simples sont construits comme dans le précédent exemple;
- l'appel de fonction est réalisé par un «changement de contexte» au plein sens du terme: on modifie le «contexte d'évaluation» pour y placer prioritairement le «bloc de fonction» de la fonction appelée;
- l'appel récursif nécessite la définition de deux nouvelles propriétés sur les variables: empile et depile, pour gérer la pile des appels récursifs.

Dans l'exemple, la fonction compte utilise les fonctions sur les arbres feuille, fg et fd, pour lesquelles on n'a pas encore écrit le corps. Au lieu d'en écrire un, et fixer alors dans le programme un certain choix de représentation du type arbre, on remplace le corps de ces fonctions par du code d'exécution – du code Lisp, introduit par l'opérateur exec.

le type arbre
(figure 3.5)

On représente très simplement un arbre par une liste à deux champs. Ce choix n'offre aucune garantie vis-à-vis du programme (typage, facilité de traduction dans le langage, etc...). Mais il permet à la fonction appelante compte d'avoir l'illusion que le type arbre a déjà été réalisé – les appels à ces fonctions retournent des valeurs qui semblent traduire un "bon comportement".

4. Définition de propriétés

Dans les exemples précédents, on peut juger que les choix de définition des opérateurs de la grammaire contraignent à l'écriture de programmes "très proches" du langage. Par exemple l'appel d'une fonction correspond à l'utilisation de l'opérateur call-fct et la donnée de paramètres; le programme appelant connaît donc la nature syntaxique de l'élément appelé, alors qu'il n'y en a pas vraiment la nécessité.

On s'est déjà un peu extrait des spécificités du langage pour les variables:
- dans sa définition, une variable est un NOM, un TYP, éventuellement une valeur INIT;
- dans son utilisation c'est lect, ecr, ou init, qui sont des propriétés indépendantes de la définition «textuelle» d'une variable.

On peut étendre la démarche aux fonctions:
- dans sa définition, une fonction est un certain nombre de paramètres, et un certain schéma de déclaration de fonction;
- dans son utilisation c'est l'élaboration d'une valeur (à rapprocher de la propriété lect des variables), avec la définition d'un paramètre effectif.

Au lieu d'appeler une fonction "en le disant explicitement":

```
(def EXP
   ((def BLOC-FCT
       ((bloc-fct((feuille)))))
    (def EXP ...))
   ((call-fct)))
```

on utilise le texte appelé, par sa propriété d'«utilisation»:

```
(def EXP
   ((def EXP ...))
   ((feuille)))
```

Par ce schéma, on ignore alors quel choix de représentation a été fait pour le concept «feuille». L'"appel" pourra fournir:

```
feuille(a)   ou   a.feuille
```

selon la nature de la propriété d'«utilisation» du texte feuille.

La difficulté à masquer des définitions de texte conduit à introduire une *barrière de visibilité* à la définition d'une fonction. Par exemple, on définit:

```
(def compte
   ((def  util  ()  ((call-fct)))
    (def  private  ...))
   ((fonction)))
```

Le texte compte offre les *deux* propriétés:
- util: l'utilisation de la fonction,
- private: une propriété que, *«conventionnellement»*, on n'utilise pas – cette propriété n'est utile que pour la déclaration de variables de bloc.

Dans sa partie privée, il définit la fonction:

```
(def  private
   ((decl-fonction)
    (def  bloc-fct
       ((def NOM ...)
        ...
        (def  INSTR  ...))))
   ((decl-fonction)))
```

Le schéma général de définition d'une fonction est donc:

```
compte = fonction
       -> util = ...
          private
              -> decl-fonction
                 bloc-fct
                     -> NOM = ...
                        ...
                        INSTR = ...
```

Il est relativement simple: il s'agit d'un arbre de profondeur 2. Mais si l'on demande à l'utilisateur de construire et de maintenir lui-même cet arbre, alors il se révèlera certainement complexe – ou tout du moins pénible d'emploi. Il faudrait donc offrir à l'utilisateur un moyen d'exprimer en des termes simples des schémas de traitements pour le parcours ou la modification de *l'arbre des textes*. Il lui permettrait par exemple de se positionner directement sur les éléments significatifs de l'arbre – ici le premier élément significatif est le champ NOM; tout ce qui précède répond au modèle de définition d'une fonction, mais ne l'intéresse pas.

Annexe

figure 1.1: le langage

grammaire

Notations:
- ::= : le terme de gauche se dérive en l'expression de droite
- | : indique un choix dans la dérivation
- {...} : répétition de l'expression entre accolades, 0, 1 ou plusieurs fois
- [...] : expressions optionnelles

```
<pgme>       ::= <pgme-decla> <pgme-instr>
<pgme-decla> ::= VAR { <decla> }
<pgme-instr> ::= { <instr> }

<decla>   ::= <NOM> : <TYP>
<instr>   ::= <affect> | <affiche> | <cond>
<affect>  ::= <VAR> := <EXP>
<affiche> ::= write( <EXP> )
<cond>    ::= if <exp-cond>
              then { <instr> }
              [ else { <instr> } ]
              endif
<exp-cond> ::= <EXP> = <EXP> | <EXP> <> <EXP>

<NOM> <TYP> <VAR> <EXP> ::= terminaux instanciables
```

exemple

```
VAR x:integer
    y:real
x:=0
y:=1.0
if x=y
then
  write('Ok')
else
  x:=y
  write('Erreur')
endif
write('fin')
```


figure 1.2: la représentation syntaxique

figure 1.2.a: syntaxe abstraite

::= : le phylum (à gauche) contient les opérateurs (à droite)
-> : l'opérateur (à gauche) est défini par les phyla (à droite)
cas particulier des listes: *... signifie que l'opérateur est défini par une *liste* d'opérateurs éléments du phylum

```
PGME ::= pgme
  pgme -> PGME-DECLA PGME-INSTR
PGME-DECLA ::= pgme-decla
  pgme-decla -> DECLA*...
PGME-INSTR ::= pgme-instr
  pgme-instr -> INSTR*...

DECLA ::= decla
  decla -> NOM TYP
INSTR ::= affect affiche cond
  affect  -> VAR EXP
  affiche -> EXP
  cond    -> EXP-COND THEN-CLAUSE ELSE-CLAUSE

EXP-COND ::= exp-egal exp-diff
  exp-egal -> EXP EXP
  exp-diff -> EXP EXP
THEN-CLAUSE ::= then-clause
  then-clause -> INSTR*...
ELSE-CLAUSE ::= else-clause
  else-clause -> INSTR*...

NOM TYP VAR EXP ::= prédéfini
```

figure 1.2.b: schémas de décompilation

Chaque opérateur est défini par:
- la liste des paramètres formels (qu'on nomme par les noms de phyla associés),
- le schéma de décompilation (à droite de =).

cas particulier des listes: le paramètre formel est noté NOM*, et la répétition d'un schéma de décompilation est symbolisée par des accolades.

```
pgme (PGME-DECLA,PGME-INSTR)
  = PGME-DECLA
    PGME-INSTR
pgme-decla (DECLA*) = VAR { DECLA }
pgme-instr (INSTR*) = { INSTR }

decla (NOM,TYP)  = NOM : TYP
affect (VAR,EXP) = VAR := EXP
affiche (EXP)    = write( EXP )
cond (EXP-COND,THEN-CLAUSE,ELSE-CLAUSE)
  = if EXP-COND
    THEN-CLAUSE
    ELSE-CLAUSE
    endif

exp-egal (EXP1,EXP2) = EXP1 = EXP2
exp-diff (EXP1,EXP2) = EXP1 <> EXP2
then-clause (INSTR*) = then { INSTR }
else-clause (INSTR*) = else { INSTR }
```

figure 1.2.c: arbre syntaxique

PGME = pgme
      PGME-DECLA = *<pgme-decla>*
      PGME-INSTR = *<pgme-instr>*

signifie qu'on définit son programme PGME sur l'opérateur pgme, en instanciant les paramètres PGME-DECLA par *<pgme-decla>* et PGME-INSTR par *<pgme-instr>*.

```
PGME = pgme
        PGME-DECLA = pgme-decla
              DECLA* ->
                 DECLA = decla
                          NOM = "x"
                          TYP = "integer"
                 DECLA = decla
                          NOM = "y"
                          TYP = "real"
        PGME-INSTR = pgme-instr
              INSTR* ->
                 INSTR = affect
                          VAR = "x"
                          EXP = "0"
                 INSTR = affect
                          VAR = "y"
                          EXP = "1.0"
                 INSTR = cond
                          EXP-COND = exp-egal
                                      EXP1 = "x"
                                      EXP2 = "y"
                          THEN-CLAUSE = then-clause
                                INSTR* ->
                                   INSTR = affiche
                                            EXP = "'Ok'"
                          ELSE-CLAUSE = else-clause
                                INSTR* ->
                                   INSTR = affect
                                            VAR = "x"
                                            EXP = "y"
                                   INSTR = affiche
                                            EXP = "'Erreur'"
                 INSTR = affiche
                          EXP = "'fin'"
```


figure 1.3: la représentation textuelle

figure 1.3.a: grammaire

```
(def pgme [illegible]
   [illegible]
   [illegible]

(def pgme-decla ()
   [illegible] DECLA [illegible]
     "VAR " [illegible] DECLA [illegible] DECLA [illegible]))

(def pgme-instr ()
   [illegible] INSTR [illegible]
     INSTR [illegible]

(def decla ()
   ([illegible] NOM) ":" (TYP) [illegible]

(def affect ()
   ([illegible] VAR) [illegible] (EXP) [illegible]

(def affiche ()
   ("write " (EXP) [illegible]

(def cond ()
   ("if " (EXP-COND) [illegible]
   (THEN-CLAUSE)
   (ELSE-CLAUSE)
   "endif" [illegible]

(def exp-egal ()
   ((EXP1) "=" (EXP2)))

(def exp-diff ()
   ((EXP1) [illegible] (EXP2)))

(def then-clause ()
   ("then" [illegible]
   [illegible] INSTR [illegible]
     [illegible] (rep (INSTR) (lst (INSTR)))))

(def else-clause ()
   [illegible] INSTR [illegible]
     "else" [illegible]
     [illegible] (rep (INSTR) (lst (INSTR)))))
```

figure 1.3.b: arbre syntaxique

```
(def PGME
 ((def PGME-DECLA
   ((def DECLA*
     (lst-env
      ((def DECLA
        ((def NOM () ("x"))
         (def TYP () ("integer")))
        ((decla)))
      ((def DECLA
        ((def NOM () ("y"))
         (def TYP () ("real")))
        ((decla))))))
   ((pgme-decla)))
  (def PGME-INSTR
   ((def INSTR*
     (lst-env
      ((def INSTR
        ((def VAR () ("x"))
         (def EXP () ("0")))
        ((affect)))
      ((def INSTR
        ((def VAR () ("y"))
         (def EXP () ("1.0")))
        ((affect)))
      ((def INSTR
        ((def EXP-COND
          ((def EXP1 () ("x"))
           (def EXP2 () ("y")))
          ((exp-egal)))
         (def THEN-CLAUSE
          ((def INSTR*
            (lst-env
             ((def INSTR
               ((def EXP () ("'Ok'")))
               ((affiche))))))
          ((then-clause)))
         (def ELSE-CLAUSE
          ((def INSTR*
            (lst-env
             ((def INSTR
               ((def VAR () ("x"))
                (def EXP () ("y")))
               ((affect)))
             ((def INSTR
               ((def EXP () ("'Erreur'")))
               ((affiche))))))
          ((else-clause))))
        ((cond)))
      ((def INSTR
        ((def EXP () ("'fin'")))
        ((affiche))))))
   ((pgme-instr)))
 ((pgme)))
```

figure 1.4: la concentration des variables

figure 1.5: la concentration des propriétés

figure 1.6: la paramétrisation des propriétés

grammaire

```
def pgme
  PGME_DECLA
  PGME_INSTR

def pgme_decla
  (lst-init (_DECLA+))
    "VAR " (rep (_DECLA) (lst (DECLA)))))

def pgme_instr
 (lst-init (_INSTR+))
    (INSTR)))

def decla ()
 ((NOM ((VAR))) ":" (TYP ((VAR))) "^M"))

def affect ()
 ((VAR) ":=" (EXP) "^M"))

(def affiche ()
 ("write " (EXP) ")" "^M"))

(def cond ()
 ("if " (EXP_COND) "^M"
  (THEN_CLAUSE)
  (ELSE_CLAUSE)
  "endif" "^M"))

(def exp_egal ()
 ((EXP1) "=" (EXP2)))

(def exp_diff ()
 ((EXP1) "<>" (EXP2)))

(def then_clause ()
 ("then" "^M"
  (lst-init ((_INSTR+))
    (* " " (rep (INSTR) (lst (INSTR))))))

(def else_clause ()
 ((lst-init ((_INSTR+))
    ("else" "^M"
     (* " " (rep (INSTR) (lst (INSTR)))))))

(def var
 ((def lect () ((NOM)))
  (def ecr () ((NOM)))
  (def init () ((_INIT)))))
```

arbre syntaxique

```
(def VAR-x
 ((def NOM () ("x"))
  (def TYP () ("integer"))
  (def INIT () ("0"))
  (var)))

(def VAR-y
 ((def NOM () ("y"))
  (def TYP () ("real"))
  (def INIT () ("1.0"))
  (var)))

(def PGME
 ((def PGME-DECLA
   ((def DECLA*
     (lst-env
      ((def DECLA
        ((def VAR ((VAR-x))))
        ((decla))))
      ((def DECLA
        ((def VAR ((VAR-y))))
        ((decla))))))))
  ((pgme-decla)))
 (def PGME-INSTR
  ((def INSTR*
    (lst-env
     ((def INSTR
       ((def VAR ((VAR-x)) ((ecr)))
        (def EXP ((VAR-x)) ((init))))
       ((affect))))
     ((def INSTR
       ((def VAR ((VAR-y)) ((ecr)))
        (def EXP ((VAR-y)) ((init))))
       ((affect))))
     ((def INSTR
       ((def EXP-COND
         ((def EXP1 ((VAR-x)) ((lect)))
          (def EXP2 ((VAR-y)) ((lect))))
         ((exp-egal)))
        (def THEN-CLAUSE
         ((def INSTR*
           (lst-env
            ((def INSTR
              ((def EXP () ("'Ok'"))
               ((affiche))))))))
         ((then-clause)))
        (def ELSE-CLAUSE
         ((def INSTR*
           (lst-env
            ((def INSTR
              ((def VAR ((VAR-x)) ((ecr)))
               (def EXP ((VAR-y)) ((lect))))
              ((affect))))
            ((def INSTR
              ((def EXP () ("'Erreur'"))
               ((affiche))))))))
          ((else-clause))))
        ("cond")))
     ((def INSTR
       ((def EXP () ("'fin'")))
       (affiche))))))
  (pgme-instr))))
(pgme)
```


figure 1.7: le poids sémantique des propriétés

```
(def pgme ()
 ((PGME-DECLA)
  (PGME-INSTR)))

(def pgme-decla ()
 ((lst-init (_DECLA+))
    ("VAR " (rep (DECLA) (lst (DECLA))))))

(def pgme-instr ()
 ((lst-init ((_INSTR+))
    (INSTR)))))

(def decla ()
  ((NOM ((VAR))) ":" (TYP ((VAR))) "^M"))

(def affect ()
  ((VAR) ":="
       (fct unless (egal (use TYP ((VAR))) (use TYP ((EXP1))))
                   "(erreur de type)")
       (EXP) "^M"))

(def affiche ()
  ("write" (EXP) ")" "^M"))

(def cond ()
 ("if " (EXP-COND) "^M"
  (THEN_CLAUSE)
  (ELSE_CLAUSE)
  "endif" "^M"))

(def exp_egal ()
 ((EXP1) "="
       (fct unless (egal (use TYP ((EXP1))) (use TYP ((EXP2))))
                   "(erreur de type)")
       (EXP2)))

(def exp_diff ()
 ((EXP1) "<>"
       (fct unless (egal (use TYP ((EXP1))) (use TYP ((EXP2))))
                   "(erreur de type)")
       (EXP2)))

(def then_clause ()
 ("then" "^M"
  (lst-init ((_INSTR+))
    (* " " (rep (INSTR) (lst (INSTR))))))

(def else_clause ()
 ((lst-init ((_INSTR+))
    ("else" "^M"
     (* " " (rep (INSTR) (lst (INSTR)))))))

(def var
 ((def lect () ((NOM)))
  (def ecr () ((NOM)))
  (def init () ((_INIT)))))
```

figure 2.1: grammaire des fonctions

```
(def affect
  ((VAR) " := " (EXP) ";"))

(def cond ()
 ("IF " (TEST) " THEN" "^M"
  "   " (ALORS) "^M"
  "ELSE" "^M"
  "   " (SINON) "^M"
  "END IF;"))

(def none ()
 ("NONE;"))


(def fonction
  ((def return
     ((def VAR ()
        ("NOM ((bloc-fct))")))))
   ((affect)))
   (def call-rec
     ((def BLOC-FCT
        ((bloc-fct)))
     ((call-fct))))
  ("FUNCTION " (NOM ((bloc-fct))) " (" (formel ((PARAM ((bloc-fct))))) ")"
    " RETURNS " (TYP ((bloc-fct))) ";" "^M"
   "BEGIN" "^M"
   "   " (INSTR ((bloc-fct))) "^M"
   "END;" "^M"))

(def call-fct ()
  ((NOM ((BLOC-FCT))) " (" (effect) ")"))


(def formel ()
  ((NOM) " : " (SNS) " " (TYP)))

(def effect ()
  ((EXP)))
```


figure 2.2: la fonction compte

```
(def fct-compte
 ((fonction)
  (def bloc-fct
    (def NOM () ("compte"))
    (def TYP () ("integer"))
    (def PARAM
      ((param-arbre)))
    (def param-arbre
      (def NOM () ("a"))
      (def TYP () ("arbre"))
      (def SNS () ("IN"))))
   (def INSTR
    ((def TEST
      ((def BLOC-FCT
         ((bloc-fct ((fct-feuille)))))
       (def EXP () ((NOM ((param-arbre ((bloc-fct))))))))
      ((call-fct))))
     (def ALORS
      ((def EXP () ("1"))))
     (return))
     (def SINON
      ((def plus-gche
        ((def EXP
          ((def BLOC-FCT
             ((bloc-fct ((fct-fg))))))
           (def EXP () ((NOM ((param-arbre ((bloc-fct))))))))
          ((call-fct))))
         (call rec)))
        (def plus-dte
         ((def EXP
           ((def BLOC-FCT
              ((bloc-fct ((fct-fd))))))
            (def EXP () ((NOM ((param-arbre ((bloc-fct))))))))
           ((call-fct))))))
         ((call-rec)))
       (def EXP ()
        ((plus-gche) " + " (plus-dte))))
     ((return))))
   ((cond))))))
((fonction)))
```

figure 2.3: les fonctions sur les arbres

```
(def fct-feuille
 ((fonction)
  (def bloc-fct
   ((def NOM () ("feuille"))
    (def TYP () ("boolean"))
    (def PARAM
      ((param-arbre)))
    (def param-arbre
     ((def NOM () ("a"))
      (def TYP () ("arbre"))
      (def SNS () ("IN"))))
    (def INSTR ()
     ((none))))))
 ((fonction)))

(def fct-fg
 ((fonction)
  (def bloc-fct
   ((def NOM () ("fg"))
    (def TYP () ("arbre"))
    (def PARAM
      ((param-arbre)))
    (def param-arbre
     ((def NOM () ("a"))
      (def TYP () ("arbre"))
      (def SNS () ("IN"))))
    (def INSTR ()
     ((none))))))
 ((fonction)))

(def fct-fd
 ((fonction)
  (def bloc-fct
   ((def NOM () ("fd"))
    (def TYP () ("arbre"))
    (def PARAM
      ((param-arbre)))
    (def param-arbre
     ((def NOM () ("a"))
      (def TYP () ("arbre"))
      (def SNS () ("IN"))))
    (def INSTR ()
     ((none))))))
 ((fonction)))
```

figure 3.1: précision de l'exemple

figure 3.1.a: grammaire

figure 3.1.b: arbre syntaxique

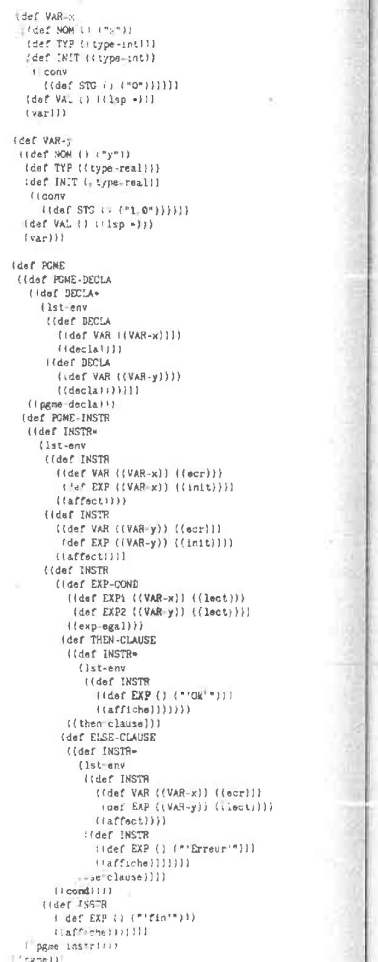


figure 3.2: la grammaire d'exécution

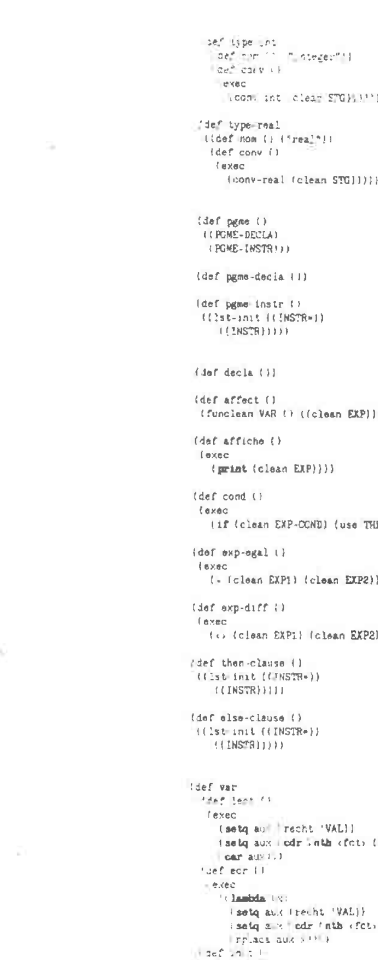


figure 3.3: précision de l'exemple

figure 3.3.a: grammaire

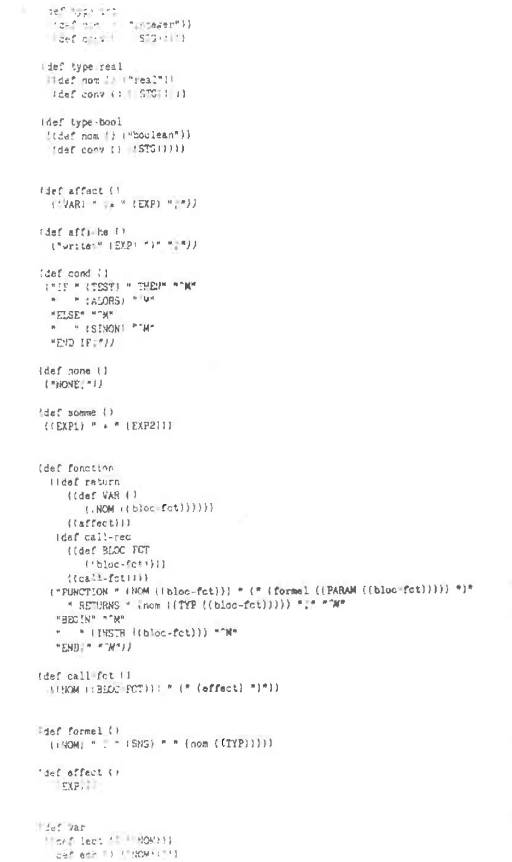


figure 3.3.b: la fonction compte

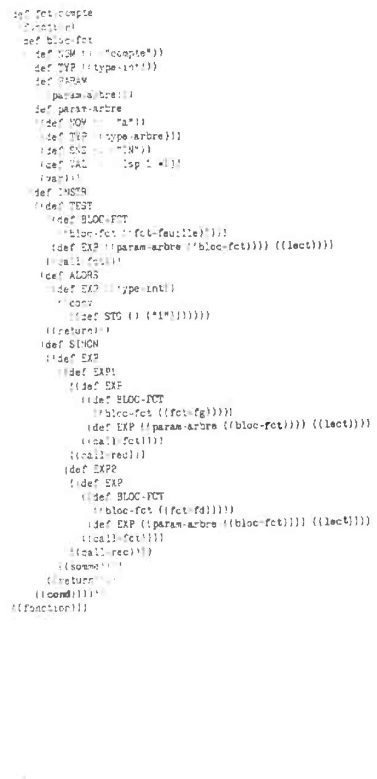
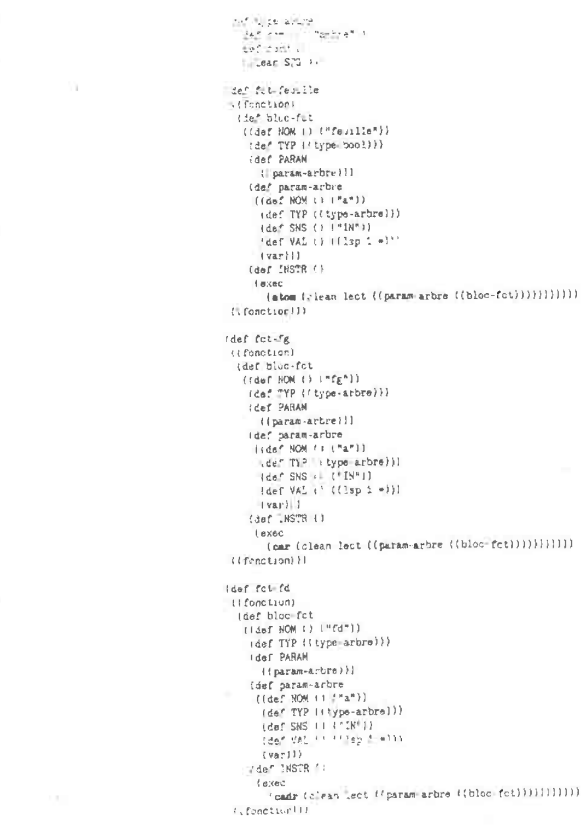


figure 3.3.c: les fonctions sur les arbres

figure 3.4: la grammaire d'exécution

figure 3.5: le type arbre



Chapitre 3.4:

étude de cas:
le langage LTR3 et l'atelier ENTREPRISE

On se propose ici de présenter l'usage pratique qu'on pourrait faire d'un outil d'édition structurée, en s'appuyant sur un langage *de haut niveau* (LTR3) et un atelier de développement dédié à ce langage (il s'agit ici de la première version de l'atelier ENTREPRISE, V5.2, spécifiquement tournée vers le langage LTR3).

1. Le langage LTR3

LTR3 est dans sa forme très largement inspiré du **langage Pascal**. On y trouve les déclarations de variables, de types, de constantes, les procédures et les fonctions, les instructions à structure de bloc. LTR3 étant bien sûr *postérieur* à Pascal (années 80), divers inconvénients propres à ce dernier ont été évités:

- l'ordre des déclarations des types, variables et constantes est libre, ce qui facilite déjà beaucoup la programmation, mais aussi à plus long terme la "génération automatique" de programmes;
- les blocs BEGIN ... END; ont été remplacés par des blocs DO ... END DO; IF ... END IF; etc... – ce qui supprime en particulier la délicate gestion du «else pendant» de Pascal;
- la distinction entre *mot-clé* et *terme prédéfini* est clairement identifiée, ce qui aide à la compréhension du langage et à son emploi;
- les paramètres des sous-programmes sont *explicitement* en entrée ou en sortie ou les deux – de ce fait par exemple les paramètres en retour ne sont pas des paramètres passés par adresse pour les besoins du moment, comme il faut le faire en Pascal.

LTR3 introduit la notion d'**exception**: déclaration, signalisation et traitement d'exception permettent de gérer dans le texte du programme les situations exceptionnelles, soit prévisibles (les exceptions prédéfinies) soit redoutées (les exceptions utilisateur). On notera que la notion, d'introduction récente, a presque aussitôt suggéré à certains une programmation systématique par exceptions [BBG 85]; ce point de vue me paraît un peu dangereux, puisqu'alors l'occurrence vraie d'une exception est indécelable – s'agit-il d'un cas "normal" d'exception, ou d'un cas réellement exceptionnel? on appauvrit considérablement la notion, qui n'est plus le traitement, dans le texte du programme, des cas d'erreur, mais un traitement "un peu particulier" présenté dans une syntaxe "un peu particulière".

Un des traits essentiels du langage est la **modularité**. Elle se présente sous deux aspects:

- Un programme LTR3 est composé d'un assemblage de *modules*, unités syntaxiques dans lesquelles on regroupe des termes du langage «sémantiquement liés»; un module utilise les déclarations présentes dans d'autres modules par une *mention explicite* d'utilisation, qui nomme le module employé, et qui n'est pas transitive – cette dernière propriété permet d'éviter des collisions entre identificateurs semblables qui ne seraient pas directement visibles.
- On a une séparation textuelle de l'*interface* et du *corps* du module:
  - l'interface est la zone des déclarations exportées par le module – l'ensemble des déclarations que verront les modules utilisateur;
  - le corps est la zone de réalisation des déclarations d'interface, où figurent les corps des sous-programmes exportés, et des déclarations locales au module – ces déclarations locales ne sont pas visibles des modules utilisateur.

On trouve aussi la notion de *type opaque*, qui permet de construire, dans un module, un type abstrait de donnée; la simplicité de définition du type opaque rend cependant son usage assez limité.

Mais LTR3 est aussi un **langage Temps Réel** (version 3). Les aspects du Temps Réel sont définis *dans le langage*, et non introduits en supplément du langage. LTR3 connaît les notions de tâches immédiates – les interruptions – et de tâches différées – avec une priorité à la création des tâches. La synchronisation est réalisée par divers

outils: des types prédéfinis ressource, événement, événement impulsionnel, des sections critiques, ... On observe un certain panachage entre les concepts de base du langage, introduits par des mots-clés, et les concepts prédéfinis, ce qui laisse à penser qu'il serait difficile de compléter ou d'améliorer les notions de base de la synchronisation entre tâches. Ces notions sont cependant suffisamment communes et générales pour que l'inconvénient ne paraisse pas trop majeur.

Le dernier point concerne les **entrées/sorties**. Elles sont standards ou formattées. Les entrées/sorties standards traitent des fichiers, dont on peut regretter qu'il s'agisse uniquement de fichiers séquentiels. Les entrées/sorties formattées permettent un dialogue à l'écran – READ et WRITE. On ne peut pas directement accéder aux adresses physiques de la machine, ce qui est malheureusement un peu limitatif.

2. L'atelier ENTREPRISE

Il s'agit de la version d'ENTREPRISE totalement dédiée au langage LTR3.

L'atelier ENTREPRISE est un **gestionnaire d'objets**. Un objet est une unité syntaxique de LTR3 – une interface, un corps ou un module-programme – ou un *contexte*, c'est-à-dire un regroupement d'objets. On trouve dans cet atelier divers outils de développement de programmes: éditeur, analyseur syntaxique, constructeur de liens (les dépendances dans un contexte), indenteur (*pretty-printing*), interpréteur. On peut regretter que la notion de contexte ne soit qu'à un seul niveau: il n'y a pas de contexte contenu dans un contexte, à l'image des répertoires (*directories*) UNIX, mais seulement des objets mis en commun entre plusieurs contextes. Ceci nécessite de la part de l'utilisateur une certaine discipline pour la gestion d'un grand nombre d'objets.

L'atelier ENTREPRISE est construit autour du **système UNIX**. De ce fait, on bénéficie de la richesse d'expression du système UNIX, et d'une certaine facilité de portage entre machines supportant ce système. L'atelier assurant la cohérence de la base d'objets, tous les fichiers utilisés appartiennent à un super-utilisateur, lequel interdit de contourner la logique de l'atelier.

ENTREPRISE est assez **indépendant du langage LTR3**. Il ne connaît de LTR3 que la syntaxe des en-têtes de modules et des utilisations de modules. Ceci permet d'espérer un atelier ENTREPRISE multi-langage, pour la version ultérieure. Il ne s'agit pourtant pas forcément d'un avantage. Par exemple l'atelier "supporte" des modules identifiés par un mot-clé du langage, ce qui conduit à construire des objets totalement inutilisables au regard des outils (l'analyse syntaxique échouera nécessairement). Une limitation plus embarrassante sans doute provient de la taille de l'"objet indivisible" que gère l'atelier: celui-ci ne descend pas, dans son contrôle de cohérence de la base d'objets, à un niveau de détail plus fin que le module LTR3. La correction d'une faute d'orthographe dans un commentaire d'une interface est analysée par l'atelier comme une modification du texte source de cette interface, et peut provoquer bien malencontreusement toute une série de "décompilations" et recompilations en cascade qu'on pourra juger bien peu utiles.

3. Apport d'un éditeur structuré

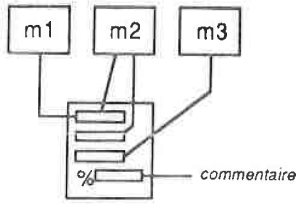
Le langage LTR3 est dans son principe très *traditionnel*:

- Identifier dans le langage une structure de données, c'est définir un type. Il en résulte alors nombre de complications qui sont liées au typage fort: pas de conversion implicite de types, pas d'«objets mutants» (des objets qui changent de type parce que l'utilisateur a changé dans sa manière de les regarder), peu d'outils pour structurer les types (héritage, dérivation, ...).

- Identifier un traitement, c'est définir un sous-programme. Il faut alors lui trouver un nom, lui trouver un endroit pour le définir (ce qui n'est pas nécessairement simple si le traitement se retrouve en de nombreux points du programme). Il résulte de la définition syntaxique du sous-programme qu'à l'exécution on retrouvera ce sous-programme: l'exécution du traitement demande un changement de contexte, parfois inutile, souvent coûteux; on localise le traitement à un point donné du code objet, ce qui complique la gestion des pages mémoires (l'"overlay") et ne conviendrait pas dans le cas d'une exécution distribuée.

Comme il a été dit, l'atelier ENTREPRISE ne descend pas à un niveau plus précis de détail que l'interface, le corps ou le module-programme. On pourrait en fait attendre de lui qu'il réalise une analyse plus fine des objets sources qu'il manipule.

Le module est une unité de programme qui sert à définir conjointement plusieurs éléments du langage (des constantes, des types, des sous-programmes) qui travaillent tous dans le "même esprit" – c'est-à-dire généralement sur une même structure de données. Cela ne signifie nullement qu'un utilisateur d'un tel module souhaite avoir une vue complète des services présentés par le module – typiquement un premier utilisateur ne s'occupe que d'empiler des valeurs et un deuxième de les dépiler: les deux utilisateurs ont la même visibilité du module mais n'en n'ont pas la même vision. L'atelier, en s'arrêtant au module dans son contrôle sur les objets, s'arrête aussi à la visibilité des identificateurs LTR3; un éditeur structuré permettrait de plus finement exprimer le lien de dépendance de l'utilisateur vis-à-vis de la collection des services qui lui sont proposés, dont il a *potentiellement* l'usage mais qu'en pratique il n'utilisera pas en totalité.



(le commentaire appartient en propre au module: sa modification n'entraîne aucun contrôle sur les dépendances).

4. La généricité

L'exemple classique de la pile comme illustration de la généricité a peut- être le tort de présenter cet aspect de la programmation comme un problème d'école, dont les applications "en vraie grandeur" sont très limitées. Pourtant la généricité des programmes est une propriété très fréquemment rencontrée. Un algorithme se présente souvent sous la forme:

```
sortie := accumulation(filtre(énumération(entrée)))
```

Par exemple:

```
S:=0;
for i:integer:=0 to 100 by 2 do
  S:=S+x[i];
end do;
```

```
max:=0;
repeat
  read reponse;
  if reponse>max then
    max:=reponse;
  end if;
until reponse=0;
```

La difficulté est que ce schéma d'algorithme peut rarement s'exprimer sous une forme fonctionnelle: l'entrée est de taille non bornée, certains traitements effectuent des effets de bord, les types retournés sont trop compliqués par rapport à ce que supporte le langage, ... C'est alors qu'on disperse la notion dans le texte du programme, parce que les outils d'expression limitent l'usage des concepts de trop haut niveau. L'éditeur structuré que l'on propose vise à pallier les carences du langage utilisé: on utilisera les outils offerts par le langage tant que ceux-ci répondront *raisonnablement* aux besoins, on utilisera des "méta-outils" d'édition quand le langage ne pourra plus les satisfaire.

La généricité dans les langages souffre certainement de deux maux:

- Le premier est la lourdeur syntaxique d'expression, et la difficulté de construction des outils de mise en œuvre – analyse syntaxique, compilateur, interpréteur, ... Un concept reconnu générique est rarement complet: il nécessite la définition de paramètres de généricité, et pour chaque instanciation la définition des paramètres effectifs, ce qui multiplie le nombre de petites fonctions à contenu sémantique pauvre. De très faibles différences entre deux instances d'un même concept générique forcent le programmeur à définir un nouveau paramètre, qui rend ensuite plus délicates la compréhension des objets construits et l'évolution simultanée du modèle générique et des instances de ce modèle.

- La seconde difficulté propre à l'utilisation de la généricité provient de la manière dont est abordé le problème: la spécification syntaxique d'un module générique conduit le programmeur à appliquer une *démarche descendante* lors de sa conception. En effet, c'est quand le module général, attiché de tous ses paramètres, est défini complètement qu'on réalisera des instances de ce module. Or des comportements voisins ne sont pas toujours détectables *ex nihilo*, et sans doute faut-il aussi s'appuyer sur des formes concrètes du module générique – les instances – pour identifier les bons paramètres. L'approche présentée ici propose de voir, durant la phase d'édition, les formes instanciées des modules génériques, c'est-à-dire des formes spécifiques du fait du choix des paramètres de généricité, mais aussi des formes complètes, sur lesquelles l'utilisateur peut donc porter par simple lecture une première appréciation.

Concernant le langage LTR3, un tel éditeur introduit, en-dehors du langage et donc en-dehors des outils de compilation des programmes sources, le concept de généricité. De ceci découlent un certain nombre de conséquences: on introduit la notion de *dérivation de type*, absente du langage, et qui est une notion difficile à introduire dans un langage fortement typé; on permet de définir des *types opaques ou semi-opaques* ou autres sans complications syntaxiques; on autorise la paramétrisation par un traitement, qui est une procédure ou simplement une suite d'instructions "en ligne"; on facilite la définition de «paramètres par défaut», qui est une notion également absente dans le langage.

Concernant l'atelier ENTREPRISE, c'est-à-dire la gestion des objets de LTR3: on se place à un niveau plus fin de modularité; les textes étant toujours générés en "LTR3 pur", on conserve tous les outils d'analyse syntaxique, compilation, interprétation, génération d'application ... ; on peut construire une réelle hiérarchie de "modules" – ou plus précisément de portions de programmes écrites en LTR3; les liens d'utilisation sont plus précis, et donc plus souples d'emploi dans les phases de construction, mise au point et maintenance des programmes.

En conclusion, on peut remarquer que l'éditeur structuré proposé fournit, par un procédé de «*macro-génération en temps réel*», des textes LTR3 syntaxiquement corrects, et ne nécessite donc nullement le développement, en parallèle, d'outils qui existeraient déjà dans l'atelier ENTREPRISE.

Chapitre 4:

L'Enrichissement du Langage par de Nouveaux Concepts

La *Syntaxe Complétée* permet d'enrichir les fonctionnalités de l'éditeur; mais elle complique la définition du langage de représentation des «textes», sans garantir l'exhaustivité des types de dépendances rencontrés dans un texte source.
Pour s'assurer une plus grande flexibilité dans l'enrichissement du langage, on définit donc ce dernier sous la forme d'une *Syntaxe Abstraite*, dans un formalisme simple et extensible. Partant de cette propriété, on introduit de nouvelles notions, soit pour résoudre certaines difficultés dans l'évaluation des «textes», soit pour présenter de nouveaux services que l'outil pourrait offrir.

Chapitre 4.1:

Présentation de la Syntaxe Abstraite

On reprend le langage de définition des «textes» dans les termes d'une *Syntaxe Abstraite*: la *Syntaxe Croncrète* devient alors la *Syntaxe Initiale*, et la *Syntaxe Complétée* devient une forme complétée de la *Syntaxe Initiale*.

1. Solution Lisp

Position du problème

Bien que dans tout ce qui précède on ait insisté sur le fait qu'il fallait rester *simple*, il ne faut pas non plus être trop simple à un tel point qu'on ne pourrait plus rien utiliser. L'idée est d'offrir un «noyau dur» intouchable, mais de permettre aussi de compléter ce noyau par des concepts plus riches.

Dans la partie précédente, on a présenté la *Syntaxe Complétée*, parce qu'elle introduit de nouvelles fonctionnalités – et en particulier la gestion des listes; on n'a cependant pas abordé la question de savoir *comment* introduire ces nouveaux opérateurs du langage. Du Langage Primitif, qui compose la *Syntaxe Concrète*, et qui par essence est très simple, on est passé à un langage beaucoup plus riche, et donc peut-être aussi plus difficile à représenter.

La question est: faut-il compléter le noyau dur en y introduisant le concept de liste, par exemple, ou peut-on, en l'état, exprimer ce concept? La réponse n'est ni oui ni non: on ne touchera pas au noyau dur, mais on ne peut pas non plus tout exprimer à partir de ce qu'il contient.

La solution est de profiter de la possibilité qui est offerte d'insérer du code Lisp pour dérouter l'évaluation d'un texte vers une évaluation spécifique de ce texte qu'on aura soi-même définie. C'est ce que j'appelle une *macro-définition* de texte, qui ne définit pas un nouveau texte mais un mécanisme d'évaluation des textes.

Remarque:
Comme il est dit parfois, Lisp est un langage qui par certains aspects possède toute la puissance d'un langage machine. La démarche s'apparente ici assez bien à l'appel de routines écrites en langage machine – Lisp – et donc n'est pas déconcertante sur le principe. Mais bien sûr l'utilisation du langage machine dans un langage "évolué" est toujours source des pires tracas, aussi, si la démarche n'est pas déconcertante sur le principe, elle peut l'être dans les faits.

Solution Lisp

La solution tient en deux points:
- utiliser des F-expressions (F-expr);
- garantir, à l'évaluation, une certaine information de contexte.

Note: évaluation des fonctions Lisp
L'appel de la fonction: (f x1 x2 x3) s'évalue comme suit:
1- on évalue, en parallèle, les arguments de la fonction f: x1, x2, x3, qui fournit les valeurs respectives: v1, v2, v3;
2- on applique la fonction f sur les arguments évalués: v1, v2, v3.
Pour une F-expr, on n'évalue pas les arguments: f est alors appliquée sur les arguments non évalués: x1, x2, x3.

Une *macro-définition* est alors:
- placée à l'intérieur d'un texte Lisp lsp,
- préfixée par le nom d'une F-expr.
L'évaluation du texte Lisp lsp provoque l'évaluation, par l'évaluateur Lisp, de l'expression préfixée, laquelle permet le déroutement de l'évaluation vers le texte de définition de la F-expr nommée.

On n'a donc qu'une seule chose à faire pour la définition d'une macro-définition: écrire le texte de la F-expr associée – problème non simple mais clairement identifié. Pour ce faire, on garantit, à l'évaluation de la macro-définition – à l'appel de la F-expr – une unique information de contexte:

> on a une variable Lisp **ctx** dont la valeur est précisément le «contexte d'évaluation» tel qu'il est défini précédemment.

Ceci signifie que dans le texte de la F-expr on pourra invoquer l'évaluation des textes, puisqu'on a la connaissance, "lispienne", du contexte d'évaluation des textes.

Par exemple (ce sont des opérateurs de la *Syntaxe Complétée*):

macro-def: lst-init
  syntaxe : (lst-init env rep)
  sémantique : env est un environnement composé d'environnements: pour chaque environnement, on évalue la représentation rep dans le contexte formé par cet environnement et le contexte reçu (variable: **ctx**).

macro-def: lst
  syntaxe : (lst rep)
  sémantique : lst doit s'évaluer "à l'intérieur" d'une évaluation de lst-init: pour chaque environnement restant, on évalue la représentation rep dans le contexte construit comme il l'est avec lst-init.

macro-def: tst
  syntaxe : (tst (cond1 rep) ... (condN rep))
  sémantique : cond1 ... condN sont des nombres.
  tst évalue le représentation rep associée à condK s'il reste exactement condK environnements dans la liste des environnements.

exemples d'emploi

schéma page suivante

figure a

```
SYNTAXE :
   lst-param ::= ε | param ";" lst-param
EXEMPLE :
   a b c -> a; b; c;
   u     -> u;
   rien  -> rien
TEXTE :
   (lsp
      (lst-init
         (env ...)
         (rep (param) ";")))
```

figure b

```
SYNTAXE :
   lst-param  ::= ε | '(' param lst-param1 ')'
   lst-param1 ::= ε | ',' param lst-param1
EXEMPLE :
   a b c -> (a,b,c)
   u     -> (u)
   rien  -> rien
TEXTE :
b_1 (lsp
      (lst-init
         (env ...)
         (rep "(" (param)
              (lsp
                 (lst
                    (rep "," (param)))) ")" )))
b_2 (lsp
      (lst-init
         (env ...)
         (rep "("
              (lsp
                 (lst
                    (rep (param)
                         (lsp
                            (tst
                               (1 (rep ")"))
                               (T (rep ","))))))))))
```

figure c

```
SYNTAXE :
   lst-param  ::= '(' lst-param1 ')'
   lst-param1 ::= e | param ',' lst-param2
   lst-param2 ::= e | "," param lst-param2
EXEMPLE :
   a b c -> (a,b,c)
   u     -> (u)
   rien  -> ()
TEXTE :
   "("
    (lsp
       (lst-init
          (env ...)
          (rep (param) (lsp
                          (lst
                             (rep "," (param)))))))
   ")"
```


2. simplification de la syntaxe : la syntaxe abstraite

Les exemples précédents ne sont pas d'une lecture très agréable. L'utilisation exclusive des concepts de base contraint en effet à "empiler" les préfixes – par exemple: (lsp (lst (rep ...))) : on obtient une syntaxe simple mais peu lisible.

On pourrait préférer une syntaxe plus concise. Par exemple, la syntaxe d'emploi de la macro-définition lst est:

(lsp (lst (rep { atm })))

il serait plus agréable d'écrire:

(lst { atm })

qui contient la même information mais se lit plus aisément.
Les exemples de la figure précédente s'écriraient alors comme il est indiqué sur la figure suivante.

```
a (lst-init
     (env ...)
     (rep (param) ";"))

b_1 (lst-init
       (env ...)
       (rep "(" (param) (lst "," (param)) ")"))

b_2 (lst-init
       (env ...)
       (rep "(" (lst (param) (tst
                                (1 (rep ")"))
                                (T (rep ","))))))

c "("
    (lst-init
       (env ...)
       (rep (param) (lst "," (param))))
  ")"
```

On voit sur ces exemples qu'une petite simplification améliore grandement la lisibilité: l'ancienne et la nouvelle définitions de lst sont de même nature – typiquement une liste d'atomes atm –, la seule différence étant l'affichage des préfixes – un seul préfixe au lieu de trois.
Mais la différence est en fait plus importante qu'il n'y paraît. Toujours sur ce même exemple, on voit qu'on utilise le texte param sous une forme abrégée, sans le préfixe 'use'; au départ, il n'y a pas d'ambiguïté; sous sa forme simplifiée, l'emploi de lst introduit lui une ambiguïté: il faut en effet savoir que, sous le préfixe 'lst', on a des objets de même nature que ceux qu'on peut trouver sous le préfixe 'rep'. En clair, on doit aussi réécrire les fonctions de saisie et d'affichage des textes pour le cas de lst: on y indiquera que, par défaut, une liste correspond à la forme abrégée d'une utilisation.

De ceci on tire deux conséquences:

- La première est que la macro-définition lst ressemble à s'y méprendre à un opérateur d'une syntaxe abstraite à définir. On y trouve la syntaxe abstraite sur laquelle se dérive l'opérateur; la fonction de saisie qui va compléter l'analyseur syntaxique d'une syntaxe concrète associée; la fonction d'affichage qui va compléter le décompilateur d'arbres de la représentation interne.

- La seconde est qu'on a identifié les attributs de définition d'un opérateur: ce sont les fonctions d'évaluation, de saisie et d'impression; on y ajoutera la fonction de recherche d'une définition dans un contexte d'évaluation, parce qu'elle s'inscrit assez agréablement dans le cadre des définitions d'attributs, même s'il ne s'agit peut-être pas d'un vrai attribut – il n'y a peut-être pas d'exemple d'opérateur où on souhaiterait définir une fonction de recherche bien spécifique.

De là on conclut:

- Les concepts élémentaires du «noyau dur» s'expriment agréablement eux aussi par l'emploi d'une syntaxe abstraite. On réécrit donc la «syntaxe Lisp» dans les termes d'une syntaxe abstraite: l'introduction des macro-définitions revient alors simplement à compléter cette syntaxe.

Cette dernière étape n'est pas indispensable: elle permet juste de mieux harmoniser les concepts, en exprimant *dans un même formalisme* les notions qui sont prédéfinies et celles que l'utilisateur sera amené à introduire.

syntaxe abstraite

On reprend la syntaxe dans les termes d'une syntaxe abstraite (pour une définition plus générale des concepts qu'introduit une syntaxe abstraite, cf. Chapitre 5.5, *«Construction de la Syntaxe Abstraite»*):

```
PHYLA
    (P1) SEX ::= env def ref rep stg use lsp liste atome
    (P2) TRM ::= env def ref lsp
    (P3) ATM ::= rep stg use lsp string
    (P4) ENV ::= env
    (P5) REP ::= rep
    (P6) LSP ::= lsp

OPERATEURS
    (O1) env  -> TRM*...
    (O2) rep  -> ATM*...
    (O3) lsp  -> SEX*...
    (O4) def  -> NOM ENV REP
    (O5) ref  -> NOM ENV
    (O6) stg  -> NOM
    (O7) use  -> NOM ENV

ATOMES
    (A1) liste   -> implemented as LISTE
    (A2) atome   -> implemented as ATOME
    (A3) string  -> implemented as STRING

SYNTAXE CONCRETE
    (C1) env  ::= '(' 'env' { TRM } ')'
    (C2) rep  ::= '(' 'rep' { ATM } ')'
    (C3) lsp  ::= '(' 'lsp' { SEX } ')'
    (C4) def  ::= '(' 'def' NOM ENV REP ')'
    (C5) ref  ::= '(' 'ref' NOM ENV ')' | '(' NOM ENV ')'
    (C6) stg  ::= '(' 'stg' NOM ')'
    (C7) use  ::= '(' 'use' NOM ENV ')' | '(' NOM ENV ')'
```


phyla

(P1) le phylum SEX regroupe tous les opérateurs de Lisp
(P2) le phylum TRM regroupe:
- les opérateurs des termes def et ref,
- l'opérateur des environnements env

(P3) le phylum ATM regroupe:
- les opérateurs des atomes stg et use,
- l'opérateur des représentations rep

(P4) le phylum des environnements est constitué uniquement de env
(P5) le phylum des représentations est constitué uniquement de rep
(P6) le phylum des expressions Lisp est constitué uniquement de lsp

opérateurs

(O1) l'opérateur env est une liste de termes
(O2) l'opérateur rep est une liste d'atomes
(O3) l'opérateur lsp est une liste de S-ex
(O4) l'opérateur def est défini par les champs: NOM, ENV, REP
(O5) l'opérateur ref est défini par les champs: NOM, ENV
(O6) l'opérateur stg est défini par le champ: NOM
(O7) l'opérateur use est défini par les champs: NOM, ENV

atomes

(A1) l'opérateur liste est un atome, du phylum prédéfini LISTE
(A2) l'opérateur atome est un atome, du phylum prédéfini ATOME
(A3) l'opérateur string est un atome, du phylum prédéfini STRING

syntaxe concrète

Elle reprend la syntaxe abstraite, sous une forme Lisp. On a conservé pour def et ref la double dérivation qui dispense de préfixer l'emploi quand il n'y pas pas d'ambiguïté. On notera que tous les opérateurs, sous leur forme concrète, sont préfixés par le nom de l'opérateur: du point de vue de Lisp, les listes ainsi construites sont des appels aux fonctions dont le nom est justement le préfixe qu'on a placé. Sur un plan purement technique, on voit donc que la syntaxe concrète, visible de l'utilisateur, est aussi la syntaxe de la représentation interne des données, laquelle est inconnue de l'utilisateur.

3. évolution de la syntaxe : la syntaxe initiale

On trouve dans la syntaxe abstraite présentée divers schémas répétitifs: env def ref, rep stg use, ... Ceci n'est pas pure coïncidence.

Par exemple, si l'on veut sur cette syntaxe insérer l'opérateur lst, la logique demande qu'on le place partout où on attend un atome atm -- des éléments d'une représentation: sur le phylum SEX et le phylum REP.

Un autre exemple d'opérateur est l'opérateur fct de la *Syntaxe Complétée*:

opérateur: fct
  syntaxe : (fct SEX)
  sémantique : identique à lsp. On attend ici une seule S-ex, la syntaxe sera donc plus simple. Par exemple:

```
(lsp                                    (fct  if (= (use x) 0)
  (if (= (use x) 0)                             "zero"
      "zero"                                    "non nul")
      "non nul"))
```

Ici, on doit logiquement placer fct partout où on attend lsp: soit sur les phyla SEX, TRM, ATM et LSP.

Un dernier exemple est:

opérateur: lst-env
  syntaxe : (lst-env { ENV })
  sémantique : définit une liste d'environnements, sans jouer sur le fait qu'un environnement peut lui-même contenir des environnements: ce sera ce type d'environnement qui sera attendu par les opérateurs de liste lst-init, lst, tst.

On placera lst-env partout où on attend env: soit sur les phyla SEX, TRM et ENV.

On constate que ce que l'on fait sur ces trois exemples, c'est réaliser une sorte de fermeture transitive de la relation:
«l'opérateur appartient au phylum»
supervisée par la relation:
«le phylum 1 contient tous les opérateurs du phylum 2».
Au lieu de demander à l'utilisateur de réaliser cette fermeture transitive, on introduit cette notion dans la définition de la *syntaxe initiale*:

PHYL ::= oper

signifie que l'opérateur oper appartient au phylum PHYL;

PHYL1 ::= PHYL2

signifie que le phylum PHYL2 est contenu dans le phylum PHYL1.

syntaxe initiale

PHYLA
(P1) SEX ::= TRM ATM LISTE ATOME
(P2) TRM ::= ENV LSP def ref
(P3) ATM ::= REP ATM stg use STRING
(P4) ENV ::= env
(P5) REP ::= rep
(P6) LSP ::= lsp

OPERATEURS
(O1) env -> TRM*...
(O2) rep -> ATM*...
(O3) lsp -> SEX*...
(O4) def -> NOM ENV REP
(O5) ref -> NOM ENV
(O6) stg -> NOM
(O7) use -> NOM ENV

phyla
(P1) le phylum SEX se compose des phyla TRM, ATM, LISTE (prédéfini) et ATOME (prédéfini)
(P2) le phylum TRM se compose:
- des phyla ENV et LSP,
- des opérateurs def et ref
(P3) le phylum ATM se compose:
- des phyla REP, LSP et STRING (prédéfini)
- des opérateurs stg et use
(P4) le phylum ENV se compose de l'opérateur env
(P5) le phylum REP se compose de l'opérateur rep
(P6) le phylum LSP se compose de l'opérateur lsp

opérateurs
Ces sont les mêmes que ceux de la syntaxe précédente.

syntaxe concrète
C'est la même que celle de la syntaxe précédente.

Les exemples précédents où l'on complète la *syntaxe initiale* se réécrivent comme suit:

ATM ::= ATM lst-init lst tst
LSP ::= LSP fct
ENV ::= ENV lst-env
ou
ENV ::= ENV LST-ENV
LST-ENV ::= lst-env

On *complète* les phyla concernés. La modification est alors automatiquement répercutée sur les phyla englobants.

On donne ici un exemple, peut-être davantage choisi pour la circonstance que les précédents, de définition d'un nouveau phylum:

```
TRM-FIX ::= ENV-FIX def
ENV     ::= ENV ENV-FIX
ENV-FIX ::= env-fix
    env-fix -> TRM-FIX*...
```

Le phylum ENV-FIX représente les environnements fixes – des environnements qui ne contiennent aucune référence à un texte. On a une "compatibilité ascendante", c'est-à-dire qu'un environnement fixe est aussi un environnement, mais la réciproque est fausse – «un environnement fixe est un environnement» doit s'entendre: si on attend un environnement et qu'on obtient un environnement fixe, alors il n'y a pas d'erreur. Une évolution ultérieure des phyla ENV ou TRM n'aura ici aucune incidence sur les phyla ENV-FIX et TRM-FIX.

syntaxe complétée

S-ex simple

```
LSP ::= LSP fct
    fct -> SEX
```

liste

```
ATM ::= ATM lst-init lst tst
    lst-init -> ENV REP
    lst -> ATM*...
    tst -> SEX*...
ENV ::= ENV lst-env
    lst-env -> ENV*...
```

liste double

```
ATM ::= ATM lst2-init lst2 tst2 atm1 atm2
    lst2-init -> ENV ENV ENV REP
    lst2 -> ATM*...
    tst2 -> SEX*...
    atm1 -> ATM*...
    atm2 -> ATM*...
```

typage

```
TRM ::= TRM deft reft
    deft -> NOM NOM ENV REP
    reft -> NOM NOM ENV
ATM ::= ATM uset
    uset -> NOM NOM ENV
```


Chapitre 4.2:

Les difficultés

Dans cette partie on n'aborde pas les difficultés *méthodologiques* liées à l'emploi des «textes»: quels "bouts de programme" isoler, quel degré de détail choisir, quelle stratégie adopter pour retrouver ou définir des schémas paramétrés, que faire des programmes existants, ... Il s'agit seulement des difficultés techniques qui résultent des choix de représentation et d'évaluation des «textes». On cherche alors à exploiter la nature *extensible* de la *Syntaxe Abstraite* pour pallier certaines limitations.

1. La gestion des noms

La première et principale difficulté qu'on rencontre concerne la gestion des noms de «textes». Le problème se présente sous deux aspects:

- d'une part il faut trouver des noms sémantiquement significatifs, ce qui peut s'avérer complexe si l'on use pleinement de la technique de découpage d'un source en «textes»;
- d'autre part le système ne peut guère supporter une utilisation anarchique des symboles: une certaine *discipline* s'impose.

Le premier point est un problème général, à résoudre au cas par cas selon la "sensibilité" de chacun. Le second est en revanche plus embarrassant, parce que le respect d'une discipline est toujours astreignant et qu'il sous-entend que le programme construit est assez fragile – cette fragilité se révélerait par un non respect de la discipline.

1.1. fragilité des programmes

La relative fragilité des programmes est essentiellement liée au type d'évaluation choisi:

> On n'évalue un *emploi* de texte (utilisation ou référence) que par "extrême nécessité", et toujours dans un contexte dynamiquement construit.

L'"extrême nécessité" signifie qu'on ne peut pas imposer qu'un certain emploi de texte soit statiquement attaché à un autre texte donné. Le contexte dynamique entraîne qu'on ne peut jamais être tout à fait certain du résultat obtenu à l'évaluation des «textes» tant qu'on ne l'a pas invoquée.
Cette fragilité se traduit de diverses manières.

1.2. perte de branches de l'«arbre des textes»

Par surcharge d'un nom essentiel – un nom proche de la racine de l'«arbre des textes», on peut obtenir un effondrement de l'architecture logicielle. Par exemple, on écrit un programme:

```
def PGME
    ....
        .....
            .....
                def A = 0
    = (use B)
def A = ... le schéma de programme ...
def B = ... (use A)
```

On définit un texte PGME, construit sur le schéma B, lui-même construit sur le schéma A; "profondément" dans PGME on surcharge A. Avec de telles définitions, l'évaluation peut, dans les mauvaises situations, retourner pour la totalité du programme PGME la forme évaluée:

```
"0"
```

Le programme a "complètement" disparu.

On peut observer deux choses:

- Le programme n'a pas "complètement" disparu: c'est la forme évaluée qui a disparu, mais le programme saisi par l'utilisateur (la forme non évaluée) existe toujours. Ceci est un trait de l'évaluateur des textes, de toujours travailler *par valeur*: une évaluation ne modifie donc jamais les données d'entrées.
- Le changement d'apparence du programme est suffisamment important pour qu'il soit immédiatement reconnu par l'utilisateur. Cela ne signifie pas que l'erreur pourra rapidement être corrigée, mais du moins elle aura été détectée.

1.3. paramètres effectifs

Le passage des paramètres effectifs n'est pas explicite: c'est un «effet de bord» du mécanisme d'empilement des environnements d'appel. L'instanciation du paramètre formel n'étant effective qu'à l'"extrême limite", il peut advenir qu'une autre définition du même symbole masque celle du paramètre effectif.

Par exemple, le schéma des fonctions est:

```
(def fonction
    ... NOM TYP ... INSTR ...
    (def nom () ((use NOM)))
    ...)
```

(on a les paramètres formels NOM, TYP, ..., INSTR; on a une propriété: nom, qui utilise simplement le paramètre NOM). Une déclaration de fonction:

```
(def fct-1
   ((ref fonction)
    (def NOM () ("fct-1"))
    (def TYP () ("integer"))
    ...
    (def INSTR ...)))
```

fct-1 fait *référence* au modèle des fonctions, en instanciant les paramètres formels NOM, TYP, ..., INSTR. Dans ce cas, l'instruction INSTR pourra difficilement utiliser la propriété nom du modèle fonction: en effet le paramètre utilisé s'appelle NOM, un symbole bien trop "commun" pour ne pas être redéfini par INSTR.

Plusieurs solutions sont possibles:

- On peut utiliser les «opérateurs typés»: deft, uset, reft, en donnant au symbole du «type» une valeur distinctive. Par exemple:

```
(def fct-1
   ((ref fonction)
    (deft bloc-fct NOM () ("fct-1"))
    (deft bloc-fct TYP () ("integer"))
    ...
    (deft bloc-fct INSTR ...)))
```

La propriété nom de fonction utilise alors le paramètre «typé» NOM:

```
(def nom () ((uset bloc-fct NOM)))
```

- Une autre solution voisine est de "typer" les textes en les plaçant sous un texte de nom distinctif. Par exemple:

```
(def fct-1
  ((ref fonction)
   (def bloc-fct
     ((def NOM () ("fct-1"))
      (def TYP () ("integer"))
      ...
      (def INSTR ...)))))
```

La propriété nom de fonction utilise alors le paramètre "typé" NOM:

```
(def nom () ((use NOM ((bloc-fct)))))
```

On rattache alors l'utilisation de nom au texte bloc-fct, ce qui fournit pour nom la bonne définition.
On peut noter que les deux approches repoussent d'un cran le problème: le nom bloc-fct pourrait lui-même être surchargé, auquel cas on ne trouverait peut-être pas la bonne définition du paramètre effectif.

- Une autre solution encore est d'introduire un nouvel opérateur: defevl, qui *force* l'évaluation de ses arguments (environnement et représentation). On définirait alors la propriété:

```
(defevl nom () ((use NOM)))
```

qui, dès l'évaluation de la référence à fonction, fournirait la forme évaluée:

```
(def nom () ("fct-1"))
```

(dans le cas courant de def, les arguments ne sont pas évalués). On supprime alors la difficulté de retrouver, à l'évaluation d'un paramètre formel, le paramètre effectif qu'on souhaitait lui voir attaché. Cette dernière solution a cependant les inconvénients:
- d'introduire une autre sorte de définition (la définition évaluée defevl), ce qui fait perdre en homogénéité des concepts;
- d'interdire, dans la définition évaluée, l'emploi de variables libres: la définition est évaluée dans son contexte de définition – un peu enrichi – et non plus dans le contexte d'utilisation.

1.4. «effets de bord»

La dernière difficulté concerne les «effets de bord»: c'est le problème le plus délicat, puisque le plus imprévisible et le moins décelable. Il peut se traduire soit par un bouclage à l'évaluation des textes – le bouclage peut être contrôlé mais n'en reste pas moins là – soit des effets "parasites" à des points du programme dont l'utilisateur ne se soucie pas.
Se prémunir contre les «effets de bord» dans une évaluation à liaison dynamique c'est s'astreindre à une certaine *discipline* de programmation. La réponse est certainement insatisfaisante, mais cette insatisfaction doit être mise en balance avec la puissance d'expression qu'autorise une telle évaluation.

2. Les modifications non locales sur la forme évaluée

Comme il a été dit, un «texte» lors de son utilisation est *évalué*: cela signifie qu'on applique un traitement sur la définition qu'on connaît du «texte» et qu'on obtient une valeur dont on ne sait rien si ce n'est qu'elle peut être affichée à l'écran. La difficulté qui se pose alors est que certains «textes», qu'on a identifiés assez naturellement, nécessitent la génération "en ligne" de texte source à deux points distincts du programme.

Par exemple, dans le cas de la pile (cf. Chapitre 2.4, *«Exemple de structuration des traitements»*), on a isolé le paramètre:

```
incrémenter-pointeur
```

qu'on instancie par deux valeurs, selon le choix de la structure de données:

```
p.b := p.b + 1
```

ou:

```
(VAR q:pile;)
NEW q;
q.b := p;
p := q;
```

Dans le second cas, il faut, c'est le langage qui le demande, utiliser une variable auxiliaire q pour créer une nouvelle valeur du pointeur: il faut donc aussi déclarer cette variable; cette déclaration devra presque toujours apparaître "autre part", c'est-à-dire dans la zone des déclarations de variables, qui est une zone «textuellement» distincte de la zone des instructions.

Une "mauvaise" réponse pourrait être de placer *deux* utilisations du texte en paramètre:
- une utilisation dans la zone des déclarations, qui permettra la déclaration éventuelle de variables auxiliaires;
- une utilisation en zone d'instructions, qui sera le traitement proprement dit.

D'une part alors on duplique l'information: «on utilise le texte "xxx"»; d'autre part on place des utilisations factices en zone de déclarations, sachant que dans la majorité des cas on n'aura pas besoin de variables auxiliaires pour réaliser le traitement.

Une "bonne" réponse est de placer, à partir du point d'utilisation du traitement, la déclaration dans un «point d'accumulation» prévu à cet effet dans la forme évaluée: la chose n'est pas simple, sachant qu'on n'accède normalement pas à la forme évaluée des «textes». C'est la "bonne" approche qui a été suivie et qu'on présente dans la suite. Elle est réalisée par la définition d'un certain nombre d'opérateurs, qui vont compliquer un peu plus la syntaxe des «textes» mais sont là pour répondre à un problème compliqué.

Les opérateurs de définition

- l'opérateur pos-accu

```
ATM ::= ATM pos-accu
pos-accu -> NOM REP
```

L'opérateur pos-accu pose un «point d'accumulation» sur la forme évaluée d'une représentation – il enrichit le phylum ATM.

• l'opérateur def-accu

TRM ::= TRM def-accu
def-accu -> NOM ENV

L'opérateur def-accu sert à désigner le point de stockage de l'information relative au «point d'accumulation» de nom NOM: on y range l'"adresse physique" du point d'insertion dans la forme évaluée et le schéma formel de décompilation défini par pos-accu.

• l'opérateur accu

TRM ::= TRM accu
accu ->

accu est là pour stocker l'information.

Les opérateurs d'utilisation

• l'opérateur ref-accu

TRM ::= TRM ref-accu
ref-accu -> NOM ENV

ref-accu est à def-accu ce que ref est à def: il permet d'établir une distinction dans les références, vers des définitions "classiques" (def) ou d'accumulation (def-accu).

• l'opérateur use-accu

ATM ::= ATM use-accu
use-accu -> NOM ENV

use-accu est l'utilisation, dans un environnement local ENV, du «point d'accumulation» de nom NOM.

• l'opérateur lst-accu

ATM ::= ATM lst-accu
lst-accu -> ATM*

lst-accu est à pos-accu ce que lst est à lst-init:
- pour une évaluation en cours, il interrompt provisoirement cette évaluation;
- pour une évaluation à venir, il remplace le schéma de décompilation courant par la liste des atomes qui le définit.

Le couple pos-accu/lst-accu présente le même type d'évaluation que le couple lst-init/lst, mais l'évaluation est ici faite de façon incrémentale, au fur et mesure des utilisations use-accu qu'on évalue.

Exemple

On définit le schéma :

```
(def procedure
   ((def decla ()
      ((NOM) ":" (TYP) ";" "^M")))
   ("PROCEDURE " (EN-TETE) ";" "^M"
    (pos-accu DECL
      ("VAR " (decla) (lst-accu "     " (decla))))
    "BEGIN" "^M"
    "  " (CORPS)
    "END;" "^M"))
```


Il est paramétré par:
– EN-TETE (l'en-tête), CORPS (le corps)
– DECL: le nom d'un «point d'accumulation».
Il fournit le schéma de décompilation du «point d'accumulation»; sous la forme de listes, on écrirait:

```
(lst-init  ((DECL))
   ("VAR " (decla) (lst "     " (decla))))
```

qui se lit:
- la première déclaration est introduite par le mot-clé "VAR",
- les déclarations qui suivent sont placées simplement (avec une tabulation de 4 blancs).

• Si la procédure ne "reçoit" aucune déclaration (par use-accu), il ne s'affiche rien.
• Si elle en "reçoit" une, elle l'affiche, précédée de "VAR".
• Si elle en "reçoit" plusieurs, elle les affiche, la première sur le premier schéma et les autres sur le second.

Une utilisation de procedure:

```
(def traitement
   ((def EN-TETE ...)
    (def-accu DECL
       ((ref-accu  bloc-decla)))
    (def-accu bloc-decla
       ((accu)))
    (def CORPS ()
       (...
        (use-accu bloc-decla
           ((def NOM () ("x"))
            (def TYP () ("integer"))))
        ...
        (use-accu bloc-decla
           ((def NOM () ("y"))
            (def TYP () ("real"))))
        ...)))
   ((procedure)))
```

Le traitement définit:
– l'en-tête,
– le paramètre du «point d'accumulation» DECL, par référence (ref-accu) au «point d'accumulation» local au texte bloc-decla,
– le «point d'accumulation» local bloc-decla,
– le corps: les textes du corps utilisent le «point d'accumulation» local bloc-decla pour placer de nouvelles déclarations de variables.

Note:
Ces opérateurs présentent certaines limitations:
• il faut avoir déjà évalué la position du «point d'accumulation» pos-accu avant de l'utiliser par use-accu: on pourrait prévoir un système retardé qui range les utilisations use-accu tant que pos-accu n'a pas été évalué;
• il n'est fait aucun contrôle sur la liste des éléments fournie au «point d'accumulation»: dans l'exemple précédent, si l'on déclare deux fois une variable "N" de type "integer" (parce qu'elle apparaît dans deux «textes» distincts en tant que variable auxiliaire), on retrouvera deux fois la déclaration «textuelle» de "N" dans la forme évaluée.

3. Les schémas optionnels

Cette dernière difficulté se présente selon deux aspects:
- donner une valeur à une définition inexistante,
- "retrouver" un schéma inexistant sur une forme évaluée.

3.1. définition inexistante

On peut se placer dans une situation où beaucoup de paramètres sont définis et peu d'entre eux sont instanciés. Ceci ne révèlera pas toujours une situation d'erreur: on peut définir un cas très général dont chaque instance ne s'intéresse qu'à une faible partie. Ce serait le cas par exemple d'un outil de gestion d'une base de données: chaque élément de la base de données ne définit qu'un nombre restreint de champs.

La solution *en l'état* se présente sous la forme:

```
(lsp
   (if  (egal (use txt) "{txt}")
        (rep ...indéfini...)
        (rep ...défini...)))
```

qui répond à la question, parce qu'on a la connaissance de la forme évaluée d'une utilisation indéfinie (ici: "{txt}"). Pour une expression plus concise, on peut alors introduire un nouvel opérateur.

- l'opérateur opt

  ATM ::= ATM opt

  opt -> NOM ENV REP REP

  opt est défini sur le phylum des atomes ATM; il recherche le nom NOM dans l'environnement local ENV:
  - s'il le trouve il évalue la première représentation REP,
  - sinon il évalue la seconde.

En annexe on donne, avec l'utilisation de l'opérateur optionnel, le schéma de décompilation d'un champ complexe d'une base de donnée.

3.2. schémas inexistants

Le problème ici est un problème à plus long terme: il s'agirait, dans une «interface conviviale», de savoir retrouver un atome ATM dont les schéma de décompilation est vide. Par exemple:

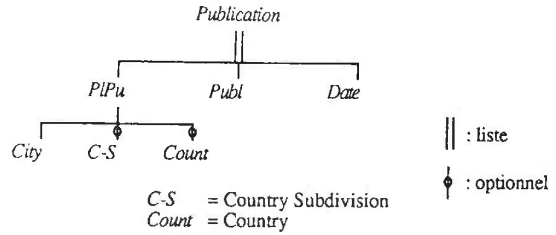
```
(lsp
    (when cond "X"))
```

n'affiche "X"que si la condition cond est vraie; autrement il n'affiche rien. Dans une «interface conviviale», où l'on n'accède qu'aux formes évaluées des «textes», un affichage vide est insaisissable:
- on ne sait pas qu'il existe,
- on ne peut pas le sélectionner, puisque sa représentation «visuelle» est inexistante.

Il faudrait donc prévoir de brefs retours sur la forme non évaluée des textes, pour permettre la manipulation de ces «textes fantômes».

Annexe : exemple d'emploi partiel

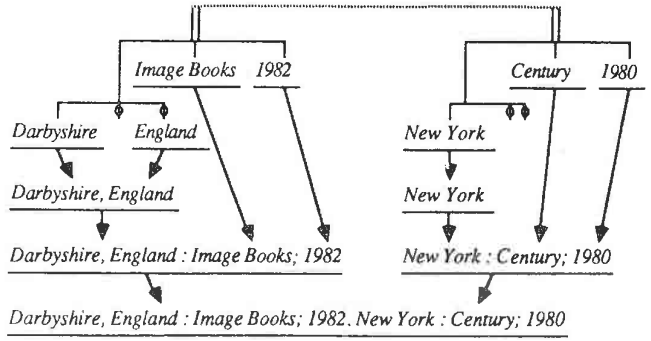
On s'intéresse au champ complexe suivant, qui représente la liste des publications recensées d'un certain ouvrage donné:



On a, par exemple, deux publications à rappeler:

Darbyshire, England: Image Books; 1982. New York: Century; 1980

On construira "logiquement" le texte à imprimer par le schéma:



(la ponctuation est automatiquement fournie).

La représentation «textuelle»

schéma de décompilation

```
(def lst-publ
  ((def publ ()
     ((City((PlPu)))
      (opt C-S ((PlPu))
        (","  (C-S)))
      (opt Count ((PlPu))
        (","  (Count)))
      ":"  (Publ)
      ";"  (Date))))
  ((lst-init  ((publ*))
     ((publ)  (lst  "."  (publ))))))
```

les données de l'exemple

```
(def publ*
  (lst-env
    ((def PlPu
       ((def City () ("Darbyshire"))
        (def Count () ("England"))))
     (def Publ () ("Image Books"))
     (def Date () ("1982")))
    ((def PlPu
       ((def City () ("New York"))))
     (def Publ () ("Century"))
     (def Date () ("1980"))))
  ((lst-publ)))
```

la forme «visuelle»

```
Darbyshire,England:Image Books;1982.New York:Century;1980
```


Chapitre 4.3:

Compléter la Syntaxe

La gestion automatique des relations de dépendances des opérateurs et des phyla dans la Syntaxe Abstraite, initiale puis complétée, facilite l'introduction de nouveaux termes. On présente ici deux nouvelles classes d'opérateurs, pour de nouveaux services de l'éditeur.
Il ne faut cependant pas perdre de vue que l'introduction de nouveaux opérateurs dans une «interface conviviale» demanderait aussi la définition de nouvelles «apparences conviviales» propres à ces opérateurs: le problème ne se pose pas ici, puisqu'on travaille sur des listes Lisp préfixées par le nom de l'opérateur.

1. L'édition des références croisées

L'exemple traite des problèmes relatifs à l'édition d'un document "en langue naturelle", et plus particulièrement à la gestion des références multiples et croisées au sein d'un même texte. L'idée est de placer des étiquettes (labels) dans le texte, et de nommer ensuite symboliquement ces étiquettes pour se référer au numéro de page ou de ligne correspondant.

1.1. présentation

On a ici deux notions différentes:
- le texte, qui est une suite de lignes, et qui contient un certain nombre d'étiquettes de référence;
- les références à des numéros de page ou de ligne, qui ne s'intéressent donc pas à des «portions de texte» mais à certaines propriétés attachées à la mise en forme du texte – les numéros de page ou de ligne.

L'intérêt qu'on attache non plus au texte édité mais à certaines propriétés de la forme décompilée entraîne qu'en plus d'un nombre restreint d'opérateurs qu'on ajoute à la syntaxe abstraite il faut légèrement modifier les fonctions d'évaluation des textes.

1.2. l'exemple

Sur l'exemple on présente deux étapes de la rédaction du texte:
- quatre paragraphes, plus un paragraphe de références au texte;
- un cinquième paragraphe inséré en début de document.

On indique à gauche le texte obtenu, et à droite les références qui sont placées dans le texte. Il s'agit de:
- une référence à la page du paragraphe «Freud» (troisième paragraphe),
- une référence à la page et à la première ligne du paragraphe «intro» (deuxième paragraphe),
- une référence à la référence bibliographique [Brook75] (page et ligne),
- les références au mot «programmation» (page).

Après insertion du paragraphe supplémentaire, on constate que les numéros de page et de ligne ont été correctement modifiés. On peut faire certaines remarques:
- la référence au paragraphe «Freud» est une *référence avant*: il faut donc effectuer un prétraitement sur la totalité du texte utile avant le calcul des numéros de page ou de ligne utiles;
- la gestion automatique des références en autorise un emploi systématique – voire excessif – sachant que les décalages à venir seront correctement répercutés;
- les références à tous les mots «programmation» sont réalisés "à la main": c'est l'utilisateur qui trouve une à une les occurrences du mot.

1.3. réalisation

Spécification

chaînes de caractères
Les retours à la ligne doivent être *explicitement* évalués, pour permettre la calcul des numéros de ligne et de page. De ce fait, on abandonne l'ancienne représentation des retours à la ligne: "^M".

(stg m) = indique un retour à la ligne
(stg p) = indique un saut de page

opérateurs
def-pge <num>
définit la pagination. <num> = numéro de la première page
def-lbl <nom>
définit une étiquette dans le texte de nom symbolique <nom>
use-pge <nom>
désigne l'utilisation du numéro de la page où figure l'étiquette de nom <nom>
use-lgn <nom>
identique à use-pge, mais pour le numéro de ligne

Implantation

chaînes de caractères
Concernant le retour à la ligne, il apparaît des défauts dans la spécification initiale:
- mauvaise spécification: le choix de la chaîne "^M" interdit de rendre *explicite* l'évaluation d'un retour à la ligne;
- défaut de spécification: on introduit la notion de «saut de page» qui était jusqu'à présent ignorée.

De là il résulte les modifications qui suivent.

fonction d'évaluation de l'opérateur stg: **evalt-stg**
On y fait figurer l'interprétation des deux formes spécifiques (stg m) et (stg p).
fonction d'impression des atomes atm: **imptt-ATM-atome**
Le retour à la ligne n'est plus reconnu par "^M".
fonctions d'impression de stg: **imptt-stg et imptc-stg**
On prend en compte ici l'impression des retours à la ligne ou des sauts de page.
fonctions d'impression évaluée externes: **Imp-eval, imp-lst-eval, imp-num-eval**
On y initialise les compteurs de lignes et de pages.
fonction d'impression évaluée interne: **imptt-atm-eval**
On prend ici en compte l'impression des retours à la ligne ou des sauts de page pour la forme évaluée des textes.

opérateurs
Il s'agit essentiellement des quatre fonctions d'évaluation: **evalt-def-pge, evalt-def-lbl, evalt-use-pge, evalt-use-lgn**, lesquelles représentent en moyenne une ligne de code LISP. Les attributs sont directement calqués sur ceux de l'opérateur stg («opérateur nommé» à un champ unique).

1.4. conclusion

Les modifications apportées au programme original sont faibles, et très facilement maîtrisables. Il s'avère qu'à un faible coût on obtient le prototype d'un éditeur de documents où les références croisées jouent un rôle prépondérant – documents techniques par exemple.
L'exemple ne vise nullement à proposer une direction de recherche, puisqu'on trouvera dans la référence [Wal 81] la présentation d'un outil totalement conçu pour la rédaction de documents techniques structurés, et proposant des facilités bien plus riches que celles présentées ici. Il cherche plutôt à montrer que, par des modifications mineures du source, on peut conserver le programme existant et lui apporter de nouvelles fonctionnalités.

2. L'élision

La forme structurée de l'information autorise deux choses:
- manipuler des entités structurellement cohérentes, en s'intéressant non pas à une suite de caractères mais à un nœud de l'arbre des textes, nœud qui est visuellement représenté par une suite de caractères;
- avoir une vue structurée du texte construit: soit l'affichage dispensé permet de reconnaître un nœud (généralement on utilise un affichage en inverse vidéo de la suite des caractères qui dépendent directement du nœud); soit la forme visuelle du texte est rendue plus concise, par le remplacement des caractères attachés au nœud par une dénomination symbolique.

Dans le présent paragraphe, on s'intéresse à ce dernier point.

2.1. présentation

La possibilité d'une vue concise de branches du programme demande l'introduction de deux notions:
- l'**holophraste**: c'est un attribut entier hérité de l'arbre (syntaxique);
- le procédé d'**élision**: c'est la spécification de schémas de décompilation différents du nœud (syntaxique) selon la valeur de l'holophraste.

Cette technique de gestion des formes élidées du programme est très classiquement utilisée dans les éditeurs structurés. Elle n'est cependant pas universelle: dans CPS (*«Cornell Program Synthesizer»* [TRH 81] [ReT 85]) la décision d'élider une branche de l'arbre syntaxique incombe à l'utilisateur; l'argument avancé est que d'une manière automatique il est toujours difficile de deviner les *intentions* du programmeur; aussi le laisse-t-on disposer librement de la possibilité d'élision, sans devancer ses désirs.
Dans l'approche classique, on observe également quelques différences:
- à la manière de Mentor [AnB 87] [MMV 85], l'élision est une propriété uniforme, qui remplace un schéma de décompilation complexe par une autre plus simple dès que l'holophraste est nul;
- à la manière de Cépage [MeN 87], elle cherche à toujours donner à l'utilisateur une vision cohérente du programme: par exemple si l'on construit une fonction, le bloc de déclaration de la fonction sera toujours visible; il servira de «cadre d'édition» au code de la fonction, lequel sera en partie élidé pour des raisons techniques de taille d'écran.

2.2. l'élision dans la *syntaxe initiale*

Dans les «textes», on retrouve la notion d'holophraste:

- Sur les formes non évaluées, il autorise une vue concise des «arbres de textes» manipulés. Par exemple:

```
(def  txt)            ; holophraste = 0

(def txt              ; holophraste = 1
   ((def a)
    (def  b)))

(def txt              ; holophraste = 2
   ((def a ())
    (def b ()))
   ("x:="  (a)  ";"))
```

- Sur les formes évaluées:
  - les environnements étant évalués sur des opérateurs d'environnement, on est replacé dans le cas précédent;
  - les représentations étant évaluées en des chaînes de caractères, la notion disparaît.

Cette dernière propriété est peut-être un peu limitative. Introduire une élision dans l'évaluation de texte demanderait d'*interrompre* l'évaluation d'une utilisation de texte quand l'holophraste atteindrait une certaine valeur; on retournerait alors une forme conventionnelle de l'utilisation élidée.
Par exemple: `(use txt)` donne `"x:=" (use a) ";"`
`(use a)` donne conventionnellement `"...a..."`
On obtient alors:
`"x:=" "...a..." ";"`
ou encore:
`"x:=...a...;"`

Une autre solution serait de retenir, sur la forme évaluée d'une utilisation, le symbole d'utilisation (dans le choix retenu une utilisation retourne une représentation *"anonyme"*). On pourrait alors décider, selon la valeur de l'holophraste, d'afficher la représentation associée à l'utilisation ou une forme conventionnelle construite à partir du symbole.

2.3. l'opérateur d'élision

Pour correctement représenter l'élision on introduit un opérateur qui permet de donner une double représentation à un texte.

L'opérateur reph

reph -> REP REP

reph est placé dans le phylum des représentations REP; il est composé de deux valeurs de représentations:
- la première est utilisée tant que l'holophraste est non nul,
- la seconde est prise quand l'holophraste est nul.

(l'holophraste n'étant pas calculé à l'évaluation des textes, il est globalement défini pour une évaluation donnée).

Par exemple:

```
(def type-pile ()
  (reph
    ("TYPE " (NOM) ":"
             (rep
               "RECORD" "^M"
               "    a:ARRAY[1.." (TAILLE) "] OF " (ELEMENT) ";" "^M"
               "    b:integer;" "^M"
               "END RECORD;" "^M"))
    ("TYPE " (NOM) ":"
             "pile(" (ELEMENT) "," (TAILLE) ");" "^M")))
```

Le texte type-pile est paramétré par:
- NOM: le nom du type de pile,
- ELEMENT: la taille de la pile.
- TAILLE: le type des éléments de la pile,

Une utilisation du type pourra être:

```
(def type-1
  ((def NOM () ("tampon"))
   (def ELEMENT () ("real"))
   (def TAILLE () ("100")))
  ((type-pile)))
```

Une utilisation de type-1 fournit:

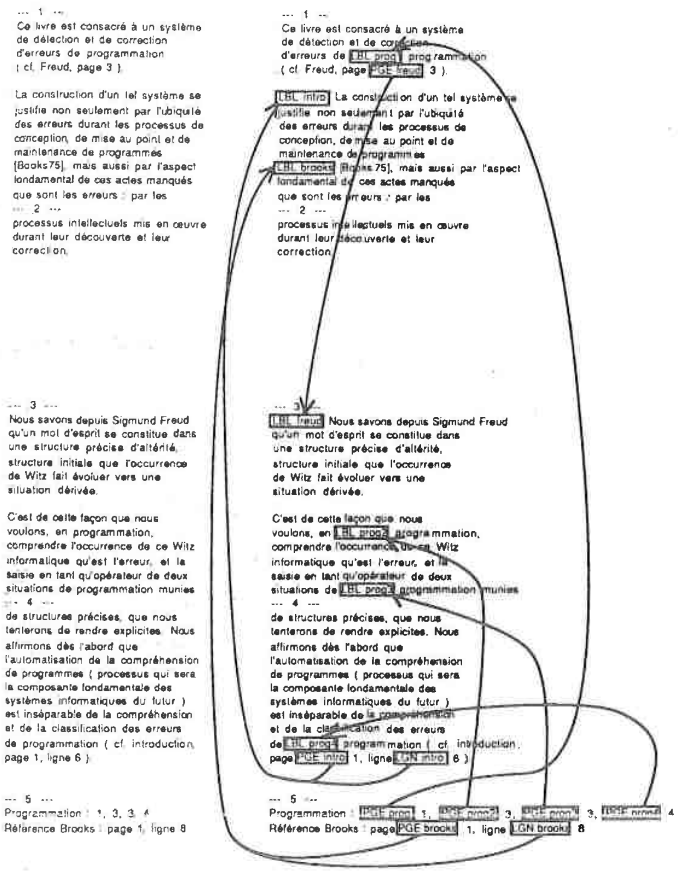
• *vue détaillée* – l'holophraste est non nul:

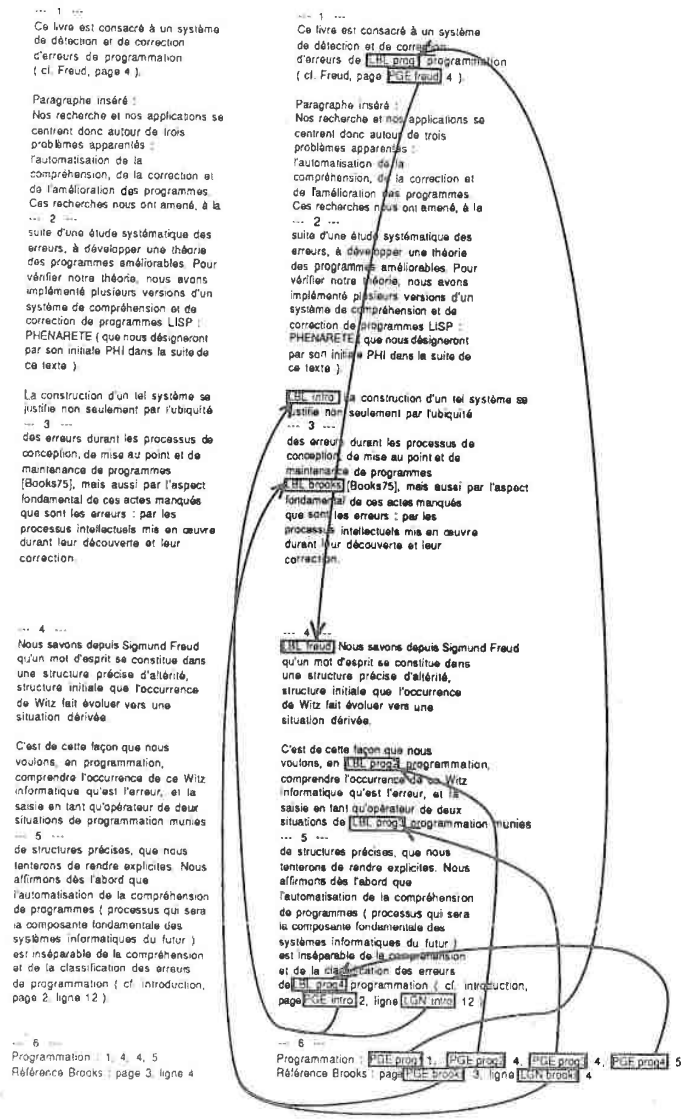
```
TYPE tampon: RECORD
                a:ARRAY[1..100] OF real;
                b:integer;
             END RECORD;
```

• *vue concise* – l'holophraste est nul:

```
TYPE tampon: pile(real,100);
```


Annexe





Chapitre 5:

La Formalisation des Solutions Techniques

On aborde dans ce Chapitre les aspects plus techniques des travaux.
Dans une première partie on discute des choix techniques retenus; ce sont:
- l'évaluation *fonctionnelle* et le calcul *symbolique*,
- la définition des termes du langage par les *objets*.

Ils sont la cause et la conséquence de l'utilisation du langage Lisp (LeLisp V15.2) pour l'écriture du programme d'évaluation.
Dans une deuxième partie on donne la représentation de la Syntaxe Abstraite qu'on a introduite précédemment.

Chapitre 5.1:

L'évaluation fonctionnelle

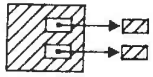
On explique ici pourquoi on a choisi une *évaluation fonctionnelle* des textes, et quel type d'évaluation on a adopté.

1. l'évaluation symbolique

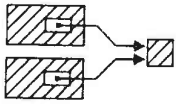
1.1. le problème initial

Le problème initial s'est présenté selon trois aspects:

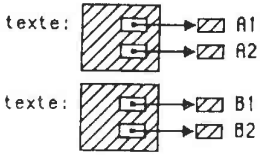
(P1) on veut définir des textes «avec trous», c'est-à-dire des textes incomplets dont on remplit les «trous» par d'autres textes.



(P2) on veut pouvoir mettre en commun un texte entre plusieurs «utilisateurs», c'est-à-dire remplir plusieurs «trous» par le même texte.



(P3) on veut pouvoir manipuler un «texte troué» de la même façon qu'un «texte complet».
Ceci signifie que le lien entre un «trou» et le texte qui le remplit n'est pas un lien statique connu du texte «troué» mais s'apparente davantage à l'instanciation d'un paramètre formel d'un type générique.



Pour satisfaire (P1), (P2) et (P3), on a choisit:

(R1) de NOMMER les textes,

(R2) d'EVALUER un texte.

NOMMER un texte signifie qu'on n'établit pas un *lien statique* entre un «trou» et sa valeur, mais qu'on construit un *lien symbolique* entre l'utilisation d'un symbole et sa définition.
EVALUER un texte signifie qu'on ne travaille pas sur l'objet qui définit un texte mais sur une *valeur* qui correspond à la forme évaluée de cet objet.

Analogie sous UNIX
A titre d'analogie, on peut comparer sous UNIX les deux types de liens suivants:

(a) Par la commande ln, on établit un *lien statique* entre deux fichiers:

```
ln fic1 fic2
```

donne au fichier fic1 les propriétés du fichiers fic2.
Par exemple, la commande:

```
cat fic1
```

est *sémantiquement* équivalente à:

```
cat fic2
```

(b) En revanche, si l'on écrit dans le fichier fic1 le *shell* suivant:

```
fichier fic1:
    echo fic2
```

on établit alors un *lien symbolique* entre fic1 et fic2.
Par exemple, la commande:

```
cat `fic1`
```

est *syntaxiquement* équivalente à:

```
cat fic2
```

La différence qu'on observe entre *(a)* et *(b)* est la suivante: dans *(a)*, on garantit de toujours "s'adresser" au même fichier, qui s'est appelé un jour fic2; dans *(b)*, on "s'adresse" au fichier qui, à ce jour, s'appelle fic2.

1.2. la structuration des textes

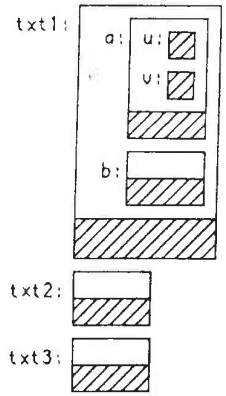
La notion de *lien symbolique* introduit de façon naturelle celle de *recherche de symbole*, qui elle-même induit celle de *contexte d'évaluation*. Ce dernier correspond à l'état dans lequel on est placé au moment où l'on veut *évaluer* un lien symbolique, *sur la forme évaluée* du «texte troué».

Pour définir ce contexte d'évaluation, on commence par donner une *structure* aux textes manipulés: on introduit alors la notion *d'environnement des définitions*. Un texte se définit alors par deux valeurs:
- une valeur d'environnement: on y trouve des définitions de textes nommés;
- une valeur de représentation: c'est la valeur *textuelle* du texte, autrement dit le «texte troué».

Evaluation de l'environnement des définitions

Parce que la choix a paru naturel, on s'inspire des langages à structure de blocs pour construire le contexte d'évaluation: à l'évaluation d'un texte, on place, en priorité dans le contexte d'évaluation, l'environnement des définitions de ce texte.

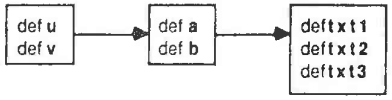
Exemple



dont on donne un représentation textuelle plus concise:

```
def txt1
   env: def a
            env: def u
                 def v
            rep: ...
        def b
            ...
   rep: ...
def txt2
   ...
def txt3
   ...
```

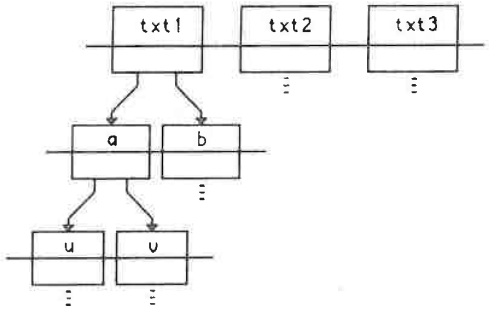
On évalue par exemple le texte u dans le contexte:



1.3. les valeurs d'environnement

A ce stade, on a défini des "objets", qui peuvent se décomposer en couches hiérarchiques, mais qui ne présentent qu'une seule propriété: *l'évaluation du texte*.

L'exemple précédent se présente comme suit:



L'«utilisateur» ne peut accéder qu'aux symboles txt1, txt2 ou txt3; dans txt1, on n'accède qu'à a ou b; dans a on n'accède qu'à u ou v; ... Il manque donc, pour réellement *structurer* les textes, la notion de module:

(P4) on veut pouvoir définir des *définitions connexes* de textes, qui pourront partager ou mettre en commun des services ou propriétés.

Au lieu d'introduire un nouveau type de lien, on introduit un nouveau type d'évaluation:

(R3) se REFERER à un texte, c'est EVALUER l'environnement de définition du texte NOMME, et "recueillir" cet environnement dans un contexte d'évaluation.

L'évaluation ne porte pas ici sur la *valeur de représentation* du texte mais sur sa *valeur d'environnement*. Le résultat ne peut donc pas être utilisé en tant que forme évaluée d'un lien symbolique mais plutôt comme complément au contexte d'évaluation de ce lien symbolique.

Par exemple:

```
use a ref txt1
```

correspond à EVALUER le lien symbolique «a» en se REFERANT (symboliquement) à «txt1», ce qui revient ici à évaluer le texte de nom a défini dans txt1.

2. Le calcul symbolique

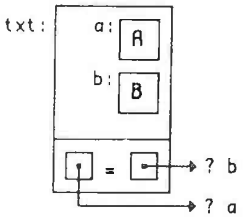
2.1. les liens statiques

La construction du contexte d'évaluation garantit une chose:

(G1) un lien symbolique d'un texte qui définit, dans son environnement de définition, ce symbole, est un *lien statique*.

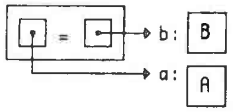
Il faut entendre par là que toute évaluation du *texte*, dans quelque contexte qu'on se trouve, fournira toujours une forme évaluée dans laquelle le lien symbolique sera évalué en utilisant la définition donnée au symbole par ce texte.

Par exemple:




```
def txt
  def a = "A"
  def b = "B"
  = (use a) "=" (use b)
```

fournira:




```
"A" "=" "B"
```

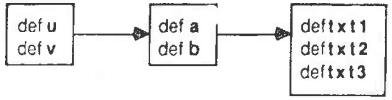
2.2. vue descendante: la fermeture lexicale

Comme généralisation du cas précédent, on peut énoncer:

(G2) l'évaluation d'un texte est identique:
- à l'évaluation du texte dans son contexte de définition,
- dans un contexte englobant, à l'évaluation du texte «le long du chemin» d'accès à la définition de ce texte.

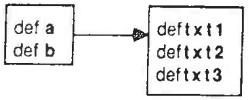
Par exemple, en reprenant les définitions du § 1.1.2.:

• Le contexte de définition de u est **ctx**:



qui est le contexte depuis le*quel* on "voit" le texte u (c'est le contexte d'évaluation du texte a).

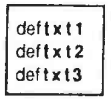
• Le premier contexte englobant est:



qui est le contexte d'évaluation du texte txt1.

Depuis txt1, on accède à u par:
use u ref a
qui place l'environnement de définition de **a** dans le contexte, du fait de la référence, et reconstruit le contexte **ctx**.

• Le deuxième contexte englobant est:



qui serait le contexte d'évaluation d'un texte glo, lequel serait le texte qui définit txt1, txt2 et txt3.

Depuis glo, on accède à u par:
use u ref a ref txt1
qui place, dans l'ordre, les environnements de définition de **a** puis de txt1, et reconstruit donc aussi le contexte **ctx**.

2.3. vue ascendante: la fermeture contextuelle

A l'inverse, on peut vouloir accéder à une définition par un chemin "oblique". Un tel accès n'est alors intéressant que si l'on introduit dans les textes des *variables libres*, c'est-à-dire des liens symboliques vers des définitions de texte qui ne sont pas directement visibles. On a alors:

(G3) dans un contexte voisin d'un texte, toutes les variables restées libres à ce niveau sont libres vis-à-vis de l'évaluation; inversement toutes les variables déjà liées restent liées à la même valeur.

Par exemple, on évalue le texte a, dont la valeur de représentation est:
(use u) ":" (use b) ":" (use txt1)

Relativement à a, u est *statiquement lié*: l'évaluation fournira toujours la valeur de u *définie dans a*.
En revanche, b est libre dans a:
- dans un texte c, voisin de a et b (c'est-à-dire défini dans le même environnement des définitions), le texte b peut être *surchargé* par une définition qui fournira la valeur du lien symbolique (use b);
- depuis txt1, ou tout autre texte de son voisinage, b est lié dans a, et la valeur ne sera pas modifiée.

Il en serait de même pour txt1, libre dans a, libre dans txt1, mais lié dans un texte englobant glo.

2.4. vue mixte

On peut mêler un peu l'un et l'autre des aspects précédents.
En gardant le même exemple, et en se plaçant «dans le voisinage» de txt1 (dans txt2 par exemple), l'utilisation:

```
use a
    def b = "X"
    ref  txt1
```

fournit l'environnement lexical de définition de a par la référence, mais surcharge le texte b par une nouvelle définition.

On peut ainsi traduire des notions de «haut niveau», comme par exemple:

```
use schema
    ref  ctx1
    ref  Rechercher
```

qui utilise le texte schema:
- en fournissant le «contexte de travail» ctx1,
- et en fournissant le type de traitement appelé Rechercher.

2.5. les difficultés

On a vu précédemment:
- le cas du contexte englobant,
- le cas du contexte voisin.

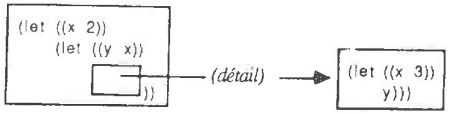
Il reste le cas du contexte englobé.

Le principe d'empilement des environnements, «à la manière des langages à structure de blocs», conduit à des comportements un peu imprévus.
Par exemple:

```
(let ((x 2))
     (let ((y x))
          (let ((x 3))
               y)))
```

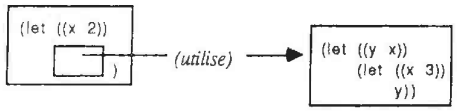
fournit intuitivement la valeur 2. Elle fournira ici la valeur 3.

En effet, une vue classique d'un tel programme est une *vue descendante*:
on voit:



qui fait dire: «x vaut 2, y vaut x: donc y vaut 2», *et puis* on a un traitement spécifique, où x vaut maintenant 3 mais y vaut 2.

On adopte ici une *vue ascendante* du programme:
on voit:



et on dit: «on *utilise* un traitement où y vaut x et x vaut 3, donc y vaut 3», *et puis* on s'intéresse au contexte d'utilisation, où x vaut 2.

Ce résultat est un peu déconcertant, mais, ce qui est plus grave, il signifie qu'on peut introduire, à l'utilisation de traitements «de bas niveau», des modifications importantes sur les concepts «de haut niveau» qu'on avait préalablement dégagés.
Pour se prémunir contre ce phénomène de *capture*, je vois deux possibilités:
- soit *typer* les textes, ce qui consiste à ne pas nommer le symbole "à risque" tel quel mais à l'attacher à un environnement des définitions, lequel tient lieu de type;
- soit définir un *nouvel opérateur* de définition des textes, qui, contrairement à une définition habituelle, évalue ses arguments.

Dans l'un ou l'autre de ces cas, il reste que le problème n'est que partiellement résolu, puiqu'il faut malgré tout reconnaître qu'un symbole est "à risque" et le définir d'une façon un peu particulière – soit en le typant, soit en forçant son évaluation.

3. Résumé

Les Problèmes posés:

(P1) on veut définir des textes «avec trous», c'est-à-dire des textes incomplets dont on remplit les «trous» par d'autres textes.

(P2) on veut pouvoir mettre en commun un texte entre plusieurs «utilisateurs», c'est-à-dire remplir plusieurs «trous» par le même texte.

(P3) on veut pouvoir manipuler un «texte troué» de la même façon qu'un «texte complet».

(P4) on veut pouvoir définir des *définitions connexes* de textes, qui pourront partager ou mettre en commun des services ou propriétés.

Les Réponses apportées:

(R1) NOMMER les textes.

(R2) EVALUER un texte.

(R3) se REFERER à un texte, c'est-à-dire EVALUER l'environnement de définition du texte NOMME, et "recueillir" cet environnement dans un contexte d'évaluation.

Les Garanties à l'évaluation des textes:

(G1) un lien symbolique d'un texte qui définit, dans son environnement de définition, ce symbole, est un *lien statique*.

(G2) l'évaluation d'un texte est identique:
- à l'évaluation du texte, dans son contexte de définition,
- dans un contexte englobant, à l'évaluation du texte «le long du chemin» d'accès à la définition de ce texte.

(G3) dans un contexte voisin d'un texte, toutes les variables restées libres à ce niveau sont libres vis-à-vis de l'évaluation; inversement toutes les variables déjà liées restent liées à la même valeur.

Chapitre 5.2:

La structuration par les objets

On explique ici comment est *naturellement* apparu l'intérêt d'une représentation des éléments de la syntaxe abstraite par des objets.

1. Représentation de la syntaxe abstraite par les objets

1.1. la syntaxe concrète

La première syntaxe contient les éléments essentiels pour la définition des textes:

```
[1] (1) env ::= ε | trm env
    (2) trm ::= def | ref
    (3)     def ::= <nom> env rep
    (4)     ref ::= <nom> env
    (5) rep ::= ε | atm rep
    (6) atm ::= <string> | use
    (7)     use ::= <nom> env
```

(1) un environnement est une liste de termes
(2) un terme est une définition def ou une référence ref
(3) une définition est: un nom <nom>, un environnement, une représentation
(4) une référence est: un nom <nom>, un environnement
(5) une représentation est une liste d'atomes
(6) un atome est une chaîne de caractères <string> ou une utilisation use
(7) une utilisation est: un nom <nom>, un environnement

1.2. la syntaxe abstraite

Parce qu'elle s'introduit assez naturellement, on reprend la première syntaxe dans les termes d'une syntaxe abstraite:

```
[2] (1)  ENV ::= env
    (2)      env -> TRM*...
    (3)  TRM ::= ENV LSP def ref
    (4)      def -> NOM ENV REP
    (5)      ref -> NOM ENV
    (6)  REP ::= rep
    (7)      rep -> ATM*...
    (8)  ATM ::= REP LSP stg use STRING
    (9)      stg -> NOM
    (10)     use -> NOM ENV
    (11) LSP ::= lsp
    (12)     lsp -> SEX*...
    (13) SEX ::= TRM ATM ATOME LISTE
```

(1) le phylum des environnements ENV contient l'opérateur d'environnement env
(2) l'opérateur d'environnement env est une liste d'opérateurs du phylum des termes TRM
(3) le phylum des termes TRM contient:
- les opérateurs des phyla des environnements ENV et des expressions Lisp LSP
- les opérateurs de définition def et de référence ref
(4) une définition est: un nom NOM, un opérateur du phylum des environnements ENV, un opérateur du phylum des représentations REP
(5) une référence est: un nom NOM, un opérateur du phylum des environnements ENV

(6) le phylum des représentations REP contient l'opérateur de représentation rep
(7) l'opérateur de représentation rep est une liste d'opérateurs du phylum des atomes ATM
(8) le phylum des atomes ATM contient:
- les opérateurs des phyla des représentations REP et des expressions Lisp LSP
- les opérateurs de *string* stg et d'utilisation use
- l'opérateur du phylum prédéfini des chaînes de caractères STRING
(9) une *string* est: un nom NOM
(10) une utilisation est: un nom NOM, un opérateur du phylum des environnements ENV
(11) le phylum des expressions Lisp LSP contient l'opérateur d'expression Lisp lsp
(12) l'opérateur d'expression Lisp lsp est une liste d'opérateurs du phylum des S-expressions SEX
(13) le phylum des S-expressions SEX contient:
- les opérateurs des phyla des termes TRM et des atomes ATM
- les opérateurs des phyla prédéfinis des atomes Lisp ATOME et des listes Lisp LISTE

Entre cette syntaxe et la précédente, beaucoup de chemin a été parcouru:
- on distingue les phyla (majuscules), qui sont des "types" d'opérateurs admissibles, et les opérateurs (minuscules), qui sont les "objets" qu'il est loisible de rencontrer;
- on dérive des phyla sur d'autres phyla, ce qui veut dire que les propriétés du phylum dérivé seront *au moins* celles de ceux sur lesquels il se dérive;
- un opérateur est déclaré appartenir à un phylum, mais il peut appartenir à plusieurs phyla, et chacun d'eux aura entre autres propriétés celle de reconnaître cet opérateur.

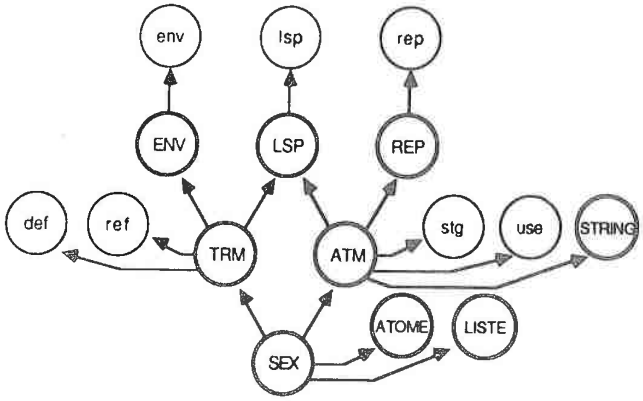
En définitive, on est passé:
- du cas **[1]**:
  - un environnement est une liste de termes;
  - pour un terme, on regarde s'il s'agit d'une définition def ou d'une référence ref
  - etc...
- au cas **[2]**:
  - quand on est placé dans le phylum des environnements ENV, on reconnaît l'opérateur d'environnement env;
  - l'opérateur d'environnement env est une liste d'opérateurs reconnus par le phylum des termes TRM;
  - quand on est placé dans le phylum des termes TRM, on reconnaît les opérateurs de définition def et de référence ref, ainsi que les opérateurs des phyla des environnements ENV et des expressions Lisp LSP;
  - etc...

Dans une telle situation, il n'est alors raisonnablement plus possible de tester, dans chaque cas d'évaluation (plus précisément pour chaque phylum dans lequel on se place à l'évaluation) de quel opérateur il s'agit, puis de traiter cet opérateur quand celui-ci a été reconnu. On devrait en effet répéter dans chaque phylum qui a la visibilité d'un opérateur donné le traitement – ou au moins l'appel à une fonction qui effectue ce traitement. Or il y a beaucoup d'opérateurs partagés entre plusieurs phyla, il y aurait donc beaucoup de répétitions.
Qui plus est, il y a une certaine logique entre les visibilités des phyla et opérateurs: définir un nouvel opérateur, qu'on place dans le phylum des environnements ENV, nécessite raisonnablement de reconnaître aussi cet opérateur dans les phyla des termes TRM et des expressions Lisp LSP.

2. L'expression par les objets

Parce que le programme qui définit la syntaxe devient plus *malléable*, on inverse donc le problème:
– chaque opérateur définit les propres propriétés qu'on peut lui appliquer,
– chaque phylum hérite des propriétés de certains opérateurs et de certains phyla.

Le graphe d'héritage de la *syntaxe initiale* est donné sur la figure:



Etablir un *lien de filiation* entre phyla, c'est définir une nouvelle relation d'héritage entre objets "du type" phylum.
Déclarer qu'un opérateur *appartient en propre* à un phylum, c'est encore définir une nouvelle relation d'héritage, entre un opérateur et un phylum.
Evaluer un attribut dans un phylum, c'est envoyer le message <attribut> à l'objet <phylum>, qui est recueilli par l'opérateur visible concerné.

Cette différence de vue favorise tout particulièrement la définition de nouveaux liens ou de nouveaux concepts – opérateur ou phylum – à partir de la *syntaxe initiale*. On a en effet concentré, à la définition d'un nouveau concept, toutes les spécificités de celui-ci *dans la définition même* de ce concept: l'objet se définit par lui-même, indépendamment du «reste du monde». Pour l'intégrer ensuite à la syntaxe existante, on placera des liens entre les opérateurs et phyla déjà définis et ce nouveau concept.

3. Méta-circularité de l'évaluateur

Il ne s'agit pas ici d'écrire la totalité de l'évaluateur en ses propres termes. Ceci pour au moins trois raisons:
– les formes évaluées des textes ne représentent qu'une partie stricte des formes non évaluées admissibles: on ne peut donc pas reconstruire tous les termes de la syntaxe par des formes évaluées;
– on n'a pas défini de fonctions usuelles de «bas niveau» comme if, or, let, ..., ce qui rend plus fastidieuse l'écriture de l'évaluateur (il faut souvent passer par du code Lisp, même pour des notions élémentaires comme le test);
– plus fondammentalement il est difficile de représenter le passage de paramètres à l'appel d'une fonction à la manière de Lisp: l'argument n'étant en principe pas évalué à l'appel, il faut forcer cette évaluation.

On ne donne donc dans ce qui suit que l'attribut qui *s'adapte bien* à la situation: c'est l'attribut d'impression, qui affiche normalement une suite de caractères, et qui retourne dans le cas présent une suite de chaînes de caractères.
La totalité de la syntaxe se trouve en annexe; on ne donne ici qu'une *sélection choisie*.

Le langage

Les opérateurs sont définis dans des «modules» isolés, pour être éventuellement partagés entre plusieurs phyla – la situation ne se présentera pas ici. Il sont formellement paramétrés par les noms des champs qu'ils définissent.
Un phylum est défini par des références à des déclarations d'opérateurs ou de phyla. Il définit de plus le traitement d'erreur, qui est pris «par défaut» pour un opérateur dont le phylum n'aurait pas la visibilité.

• opérateur d'environnement: meta-env

```
(def decl-env                          ; définit l'opérateur meta-env
  ((def meta-env                       ;    le paramètre formel est LST-TRM
     ((def imptt ()                    ; définit sur meta-env la propriété imptt
        ("(env"                        ;
         (lst-init  ((LST-TRM))        ; appelle pour chaque élément de LST-TRM
              (imptt  ((meta-TRM))))   ;    sa propriété imptt dans le phylum des
         ")"))))))                     ;    des termes meta-TRM
```

• opérateur de définition: meta-def

```
(def decl-def                          ; définit l'opérateur meta-def
  ((def meta-def                       ;    les paramètres sont NOM, ENV et REP
     ((def imptt ()                    ; définit sur meta-def la propriété imptt
        ("(def" (NOM)                  ; utilise le paramètre formel NOM
         (imptt                        ; utilise la propriété imptt du paramètre
           ((ENV)                      ;    formel ENV dans le phylum des
            (meta-ENV)))               ;    des environnement meta-ENV
         (imptt                        ; utilise la propriété imptt du paramètre
           ((REP)                      ;    formel REP dans le phylum des
            (meta-REP)))               ;    des représentations meta-REP
        ")"))))))                      ;
```

• phylum des environnements: meta-ENV

```
(def meta-ENV                          ; définit le phylum meta-ENV
  ((ref  decl-env)                     ; déclare que l'opérateur meta-env
                                       ;    appartient au phylum meta-ENV
   (def  imptt ()                      ; définit la propriété imptt pour les cas
     ("()"))))                         ;    d'erreur
```

• phylum des termes: meta-TRM

```
(def meta-TRM                          ; définit le phylum meta-TRM
  ((ref  decl-def)                     ; déclare les opérateurs meta-def et meta-ref
   (ref  decl-ref)                     ;    appartenir au phylum meta-TRM
   (ref  meta-ENV)                     ; déclare les opérateurs des phyla meta-ENV
   (ref  meta-LSP)                     ;    et meta-LSP appartenir à meta-TRM
   (def  imptt ()                      ; définit la propriété imptt par défaut:
     ((imptt  ((meta-ref)))))))        ;    ici on prend la propriété de l'opérateur
                                       ;    meta-ref
```


Annexe : la syntaxe initiale

• phylum des environnements: meta-ENV

```
(def meta-ENV
  ((ref  decl-env)
   (def  imptt ()
     ("()"))))
```

• opérateur d'environnement: meta-env

```
(def decl-env
  ((def meta-env
     ((def  imptt ()
        ("(env"
         (lst-init  ((LST-TRM))
              (imptt  ((meta-TRM))))
         ")"))))))
```

• phylum des termes: meta-TRM

```
(def meta-TRM
  ((ref  meta-ENV)
   (ref  meta-LSP)
   (ref  decl-def)
   (ref  decl-ref)
   (def  imptt ()
     ((imptt  ((meta-ref)))))))
```

• opérateur de définition: meta-def

```
(def decl-def
  ((def meta-def
     ((def  imptt ()
        ("(def"  (NOM)
         (imptt
           ((ENV)
            (meta-ENV)))
         (imptt
           ((REP)
            (meta-REP)))
         ")"))))))
```

• opérateur de référence: meta-ref

```
(def  decl-ref
  ((def  meta-ref
     ((def  imptt ()
        ("(ref"  (NOM)
         (imptt
           ((ENV)
            (meta-ENV)))
         ")"))))))
```

• phylum des représentations: meta-REP

```
(def meta-REP
  ((ref  decl-rep)
   (def  imptt ()
     ("()"))))
```

• opérateur de représentation: meta-rep

```
(def  decl-rep
  ((def  meta-rep
     ((def  imptt ()
        ("(rep"
         (lst-init   ((LST-ATM))
              (imptt  ((meta-ATM))))
         ")"))))))
```

• phylum des atomes: meta-ATM

```
(def meta-ATM
  ((ref  meta-REP)
   (ref  meta-LSP)
   (ref  decl-stg)
   (ref  decl-use)
   (ref  meta-STRING)
   (def  imptt ()
     ((imptt   ((meta-use)))))))
```

• opérateur de *string*: meta-stg

```
(def  decl-stg
  ((def  meta-stg
     ((def  imptt ()
        ("(stg"  (NOM)
         ")"))))))
```

• opérateur d'utilisation: meta-use

```
(def  decl-use
  ((def  meta-use
     ((def  imptt ()
        ("(use"  (NOM)
         (imptt
           ((ENV)
            (meta-ENV)))
         ")"))))))
```

• phylum des expressions Lisp: meta-LSP

```
(def meta-LSP
  ((ref  decl-lsp)
   (def  imptt ()
     ("()"))))
```


• opérateur d'expression Lisp: meta-lsp

```
(def  decl-lsp
  ((def  meta-lsp
     ((def  imptt ()
        ("(lsp"
         (lst-init   ((LST-SEX))
              (imptt  ((meta-SEX))))
         ")"))))))
```

• phylum des S-expressions: meta-SEX

```
(def meta-SEX
  ((ref  meta-TRM)
   (ref  meta-ATM)
   (ref  meta-ATOME)
   (ref  meta-LISTE)
   (def  imptt ()
     ("()"))))
```

• phyla prédéfinis: meta-STRING meta-ATOME meta-LISTE

```
(def meta-STRING
  ((def  imptt ()
       ("""  (STRING)  """))))

(def meta-ATOME
  ((def  imptt ()
       ((ATOME)))))

(def meta-LISTE
  ((def  imptt ()
       ((LISTE)))))
```

Exemple de texte

On définit le texte:

```
(def  txt
  ((def  a  ()  ("A")))
   ("X:="  (use  a)  ";"))
```

On le représente par:

```
(def  meta-txt
  ((ref  meta-def)
   (def  NOM  ()  ("txt"))
   (def ENV
     ((ref  meta-env)
      (def LST-TRM
         (lst-env
           ((ref  meta-def)
            (def  NOM  ()  ("a"))
            ;; suite
```

```
                ;; suite
                (def ENV
                  ((ref  meta-env)
                   (def LST-TRM
                        (lst-env))))
                (def REP
                  ((ref  meta-rep)
                   (def LST-ATM
                        (lst-env
                          ((ref  meta-stg)
                           (def  NOM  ()  ("A")))))))))))))
    (def REP
      ((ref  meta-rep)
       (def LST-ATM
            (lst-env
              ((ref  meta-stg)
               (def  NOM  ()  ("X:=")))
              ((ref  meta-use)
               (def  NOM  ()  ("a"))
               (def ENV
                 ((ref  meta-env)
                  (def LST-TRM
                       (lst-env)))))
              ((ref  meta-stg)
               (def  NOM  ()  (";")))))))))
```

On utilise sa propriété imptt en se plaçant dans le phylum meta-TRM:

```
    (imptt
      ((meta-txt)
       (meta-TRM)))
```

qui fournit:

```
  ("(def"  "txt"
      "(env"
         "(def"  "a"   "(env" ")"   "(rep"  """A""  ")"  ")"
      ")"
      "(rep"
         """X:="""   "(use"  "a"  "(env"  ")"  ")"    """;"""
      ")"
   ")"  )
```

qui s'affiche sous la forme simplifiée:

```
(def txt
   (env
      (def a  (env)  (rep "A"))
   )
   (rep
      "X:="  (use a (env))  ";"
   )
)
```


Chapitre 5.3:

Modèle sémantique comparé de l'évaluateur

On présente ici un modèle formel de l'évaluateur de textes, en comparant son comportement avec trois grandes catégories d'évaluation des programmes Lisp: la fermeture lexicale, la liaison dynamique, le parallélisme d'évaluation.

1. Notations

1.1. Généralités

Les expressions

On se place sur un ensemble S* construit comme suit:

- on a un ensemble S de *symboles*, S={a,b,c,···,x,y,z,···}
- on a deux symboles spéciaux: λ et apply [1]
- un *terme* de S* est de la forme:
  - (i) u : c'est un symbole (u∈S);
  - (ii) λx.E, où E∈S* : c'est une λ-expression;
  - (iii) (f x1···xn), où f∈S : c'est l'appel d'une *fonction primitive* (c'est-à-dire un symbole qui admet une interprétation fonctionnelle implicite – ou au moins intuitive);
  - (iv) (apply f x1···xn), où f,x1,···,xn sont dans S* : c'est l'*application* du terme f sur les termes x1···xn;
  - (v) on utilise aussi des *symboles primitifs*, c'est-à-dire des symboles qui ont encore une interprétation intuitive.

Par exemple:

(i) a,b,c,···,x,y,z,···

(ii) λx.(+ x 3) : c'est la fonction x ↦ (+ x 3)

(iii) (+ x y) : intuitivement c'est l'addition de x et de y

(iv) (apply λx.x a) : c'est l'application de la fonction x ↦ x sur le symbole a

(v) 1,2,3,··· : intuitivement ce sont les nombres

L'interprétation

On définit une *interprétation* sur S* par les données suivantes:

- on a un ensemble **S** de *valeurs*, **S**={**a**,**b**,**c**,···,**x**,**y**,**z**,···}
- on a une application:

$$\mathbf{I} : [S \to \mathbf{S}] \times S^* \to \mathbf{S}$$
$$(env, E) \quad \mapsto \mathbf{I}\,[\![env\ E]\!]$$

[1] λ est le symbole d'*abstraction* du λ-calcul: il permet de construire des λ-expressions. apply est le symbole d'*application* d'une λ-expression sur un terme. L'usage n'ayant pas précisément fixé de notation, une application se représente par (λx.E a) ou ((λx.E a)) ou (λx.E) a . Ici, on *explicite* l'application en la nommant (apply λx.E a).

où [S → **S**] est l'ensemble des *fonctions*:

- partiellement définies sur un sous-sensemble *fini* de S à valeur dans **S**,
- qui ont une valeur conventionnelle sur les symboles primitifs.

Dans la suite, on appelle ces fonctions des *contextes d'évaluation* ou encore des *environnements*.

Remarques

Intuitivement, on n'a qu'un nombre restreint de *variables* qui ont une valeur dans un environnement donné: c'est pourquoi on considère des fonctions *partielles* sur S. Inversement si on *utilise* dans un terme t une variable indéfinie dans un environnement env (un symbole qui n'a pas d'image par env), on interprète le terme t par une valeur constante **o** de S, qui représente l'indéfini. C'est-à-dire qu'on pose:

$$\mathbf{I}\,[\![env\ t]\!] = \mathbf{o}$$

Savoir dans quels cas on considère qu'on *utilise* un symbole est l'objet de ce qui suit.

Les symboles primitifs s'interprètent de façon "naturelle":

Si l'on a $\mathbf{I}\,[\![env\ x]\!] = \mathbf{x}$ et $\mathbf{I}\,[\![env\ y]\!] = \mathbf{y}$, alors par exemple:

$$\mathbf{I}\,[\![env\ (+\ x\ y)]\!] = (\mathbf{I}\,[\![env\ +]\!]\ \mathbf{I}\,[\![env\ x]\!]\ \mathbf{I}\,[\![env\ y]\!]) = (+\ \mathbf{x}\ \mathbf{y})$$
$$\mathbf{I}\,[\![env\ 1]\!] = 1$$
$$\mathbf{I}\,[\![env\ 2]\!] = 2$$

et d'une façon générale, on note de la même manière un symbole primitif et sa valeur dans un environnement.

1.2. Notations sur les environnements

On introduit des notations particulières pour définir le prolongement d'une fonction d'environnement. Etant donné un environnement env-0 :

(x↦**x**):env-0

est une nouvelle fonction env telle que:

- env(x) = **x**
- env(u) = env-0(u), pour u∈S, u≠x, si env-0 est définie sur u
- env n'est pas définie sur u, pour u∈S, u≠x, si env-0 n'est pas définie sur u

Il s'agit donc *presque* d'un prolongement, l'exception étant le cas où env-0 est déjà définie en x, puisqu'alors on *redéfinit* la valeur de l'environnement sur le symbole.

`env-1:env-0`

est la généralisation de la notation précédente. On l'interprètera en disant que les déclarations de `env` *masquent* les déclarations de `env-0` pour des symboles identiques.

Entre autres propriétés, : est associatif:

`env-2:(env-1:env-0)`
`= (env-2:env-1):env-0`
`= env-2:env-1:env-0` *(notation)*

1.3. Le cas de Lisp

Lisp n'implante pas à proprement parler le λ-calcul, parce que dans le langage on interprète les termes Lisp *dans* le même ensemble, à savoir **S**=S* (S* est alors dit *réflexif*). Ceci signifie que l'interprétation d'une formule est susceptible de *modifier* l'environnement d'interprétation par "effet de bord" – c'est encore plus vrai avec l'emploi des fonctions Lisp `eval` et `apply`[1]. Pour cette raison on ne peut pas donner de *sémantique dénotationnelle* "pure" aux expressions Lisp, puisqu'on doit faire intervenir le *contexte d'évaluation*.

Définition

On a par conséquent un autre symbole "spécial" de S: `def`, qui définit un nouveau symbole qui va compléter le contexte d'évaluation.

Le schéma général d'une définition est:

`(def x val-x)`

où `x` est le symbole défini, et `val-x` la définition proprement dite (elle sera généralement décomposable en deux champs: un champ `env-x` qui est l'*environnement des définitions locales* et un champ `rep-x` qui est la *représentation* – à nouveau décomposable).

L'interprétation d'une telle définition fournit l'environnement:

$(x \mapsto \aleph)$

qui est l'*interprétation d'environnement* du terme (`def x val-x`), ℵ étant l'*interprétation de valeur* de ce même terme.

1. On prendra soin de distinguer la fonction Lisp `apply` du symbole du λ-calcul `apply`.

On distingue ainsi *deux types* d'interprétation des termes:

$\mathbf{I}_{env}$:qui retourne un environnement qui va *compléter* l'environnement courant d'évaluation (c'est donc un élément de $[S \to \mathbf{S}]$);

$\mathbf{I}_{rep}$:qui retourne une *valeur* (un élément de **S**).

λ-expressions

L'une des difficultés essentielles dans l'interprétation des expressions Lisp est l'interprétation qu'on doit donner à l'évaluation d'une λ-expression Lisp, de la forme:

`(lambda(x) E)`

Tant qu'on n'*applique* pas cette forme évaluée, on ne peut pas réellement la dénoter dans le modèle précédent. On introduit donc une troisième forme d'évaluation:

$\mathbf{I}_{rep}$ ⟦`env (lambda(x) E)`⟧
$= \mathbf{I}_{\lambda}$ ⟦`env λx.E`⟧

la forme $\mathbb{I}_{\lambda}$ étant conservée "telle quelle" dans l'attente d'une application. [1]

Environnement

Un «environnement» (au sens des termes Lisp) est un ensemble de définitions connexes, qu'on note ici:

`env = ((def x A) (def y B)···)`

et dont l'*interprétation d'environnement* a principalement deux formes possibles:

- L'*environnement simultané*: c'est la syntaxe `let`:

```
(let ((x A)
      (y B))
  ···)
```

  qui s'évalue:
  $env^{\circ} = \mathbf{I}_{env}$ ⟦`env-0 env`⟧

- L'*environnement parallèle*: c'est la syntaxe `define`:

```
(define x A)
(define y B)
···
```

  qui s'évalue quant à lui:
  $env^{*} = \mathbf{I}_{env}$ ⟦`env*:env-0 env`⟧

1. De ce fait le domaine des termes Lisp peut *ne pas être réflexif*: la forme $\mathbf{I}_{\lambda}$, avec la sémantique qu'on lui attache, n'étant jamais tout à fait un terme du langage.

C'est-à-dire que l'environnement est évalué sur *lui-même.* [1]

2. La fermeture lexicale: le langage Scheme

Références: sur le langage Scheme [AbS 83] [Sch 87].
On présente en premier le langage Scheme parce qu'il répond de la manière la plus satisfaisante à l'interprétation "intuitive" qu'on peut donner d'une λ-expression. La recherche des identificateurs est réalisée *statiquement* («lexical scoping»): ceci facilite *beaucoup* le travail du compilateur.

2.1. termes

Symbole
Un symbole retourne sa valeur:

$\mathbf{I}_{rep}$ ⟦env-0 x⟧
= env-0(x) ; si x est défini dans env-0, sinon <erreur>

λ-expression
Une définition de λ-expression retourne la forme $\mathbf{I}_{\lambda}$ inchangée:

$\mathbf{I}_{rep}$ ⟦env-0 (lambda(u) env rep)⟧
= $\mathbf{I}_{\lambda}$ ⟦env-0 λu.(env rep)⟧

Dans cette "forme figée" on conserve l'environnement d'évaluation de la définition (ici: env-0): celui-ci correspond *de fait* à l'environnement lexical de la définition.

1. Plus formellement, on définit une suite d'environnements évalués:
env^0 = env
env^1 = $\mathbf{I}_{env}$ ⟦env^0:env-0 env⟧
env^2 = $\mathbf{I}_{env}$ ⟦env^1:env-0 env⟧
...
env^{n+1} = $\mathbf{I}_{env}$ ⟦env^n:env-0 env⟧
...
env* = $\mathbf{I}_{env}$ ⟦env*:env-0 env⟧
c'est-à-dire que env* est la *limite* de la suite des env^n (sur la topologie des environnements, qu'on définirait en définissant une mesure sur les termes de S*, ...)
Par exemple:
env^0 = ⟦(def x (cons 1 y)) (def y (+ 2 z))(def z 3)⟧
env^1 = ⟦(def x (cons 1 (+ 2 z))) (def y 5) (def z 3)⟧
env^2 = ⟦(def x (cons 1 5)) (def y 5) (def z 3)⟧
...
env* = ...

Définition
Une définition évalue son argument [1]:

$\mathbf{I}_{env}$ ⟦env-0 (def x env rep)⟧
env^0 = $\mathbf{I}_{env}$ ⟦env-0 env⟧
rep^0 = $\mathbf{I}_{rep}$ ⟦env^0:env-0 rep⟧
= (x ↦ rep^0)

La valeur est donc calculée à la définition du symbole.

Par exemple:

```
(define x                    retourne: ( x ↦ 5 )
    (let ((a 3))
      (+ a 2)))

(define x                    retourne: ( x ↦ Iλ ⟦(a↦3):env-0 λu.(+ a u)⟧ )
    (let ((a 3))
      (lambda(u) (+ a u))))
```

2.2. application

Application
C'est l'application d'une fonction f sur un argument a:

$\mathbf{I}_{rep}$ ⟦env-0 (apply f a)⟧
$\mathbf{I}_{rep}$ ⟦env-0 f⟧ = $\mathbf{I}_{\lambda}$ ⟦env-lex λu.(env-f rep-f)⟧
$\mathbf{I}_{rep}$ ⟦env-0 a⟧ = ●
= $\mathbf{I}_{rep}$ ⟦env-f*:(u↦●):env-lex rep-f⟧
où env-f* = $\mathbf{I}_{env}$ ⟦env-f*:(u↦●):env-lex env-f⟧

env-f* est la *forme évaluée* de l'environnement local des définitions de f [2].

1. L'interprétation donnée ici à un environnement parallèle (de type define) n'est pas garantie par le «manager».
2. L'*algorithme* étant peu simple, le cas des λ-expressions compliquant encore davantage les choses, on n'a en Scheme qu'un pseudo-parallélisme d'évaluation:
- s'il s'agit de *variables*, l'évaluation échoue. Par exemple:
  (define x y) ; [1]
  (define y 3) ; [2]
  L'évaluation échoue en [1], alors que si [1] était placé après [2] elle pourrait être valide: dans Scheme, pour des raisons d'homogénéité, elle échouerait aussi.
- s'il s'agit de *fonctions*, l'évaluation réussit. Ceci permet de définir des fonctions mutuellement récursives, ou même plus simplement récursives (ce qu'on était en droit d'attendre du langage).

Evaluation

S'agissant du langage Lisp, c'est-à-dire d'un domaine *réflexif* d'interprétation, on a deux autres fonctions "spéciales": quote et eval.

I_{rep} ⟦env-0 (quote x)⟧ = x

I_{rep} ⟦env-0 (eval x env)⟧

I_{rep} ⟦env-0 x⟧ = ※

I_{env} ⟦env-0 env⟧ = **env**

= I_{rep} ⟦**env** ※⟧

Scheme demande un environnement d'évaluation pour l'application de eval, qui lui servira de fermeture lexicale. [1]

3. Les Lisps standards: LeLisp

Références: sur le langage LeLisp [Manuel LeLisp V15.2], sur le langage Interlisp [TeM 81], sur le langage Common Lisp, qui mêle la fermeture lexicale *et* la liaison dynamique [Tou 88].

Dans un Lisp standard, l'évaluateur *empile* les environnements d'évaluation: la recherche d'un identificateur – un symbole – est alors *dynamique* («dynamic scoping»). Ceci a deux conséquences importantes:

- la première, néfaste: on a des «effets de bord imprévisibles» par le masquage "sauvage" d'une variable lors d'une redéfinition du symbole;
- la seconde, essentielle: on a la notion de *variable libre*, qui nécessairement a disparu en Scheme. On peut donc réaliser des modèles génériques de fonctions, instanciés dynamiquement par des paramètres effectifs transparents.

3.1. termes

Symbole

I_{rep} ⟦env-0 x⟧

= env-0(x)

1. En particulier, concernant les λ-expressions:
(1) si ※ = (lambda(u) E), **env** est la fermeture lexicale de la λ-expression:
I_{rep} ⟦env-0 (eval x env)⟧ = I_{rep} ⟦**env** (lambda(u) E)⟧ = I_λ ⟦**env** λu.E⟧
(2)si ※ = I_λ ⟦env-0 λu.E⟧, ※ *n'est pas* dans S^*, donc ※ n'est pas un terme Lisp, et l'évaluation *échoue*.

λ-expression

I_{env} ⟦env-0 (lambda(u) env rep)⟧

= I_λ ⟦env-0 λu.(env rep)⟧

Définition

I_{env} ⟦env-0 (def x env rep)⟧

env° = I_{env} ⟦env-0 env⟧

rep° = I_{rep} ⟦env°:env-0 rep⟧

= (x ↦ rep°)

Ici aussi la valeur est calculée à la définition.

3.2. application

Application

L'application de f sur a:

I_{rep} ⟦env-0 (apply f a)⟧

I_{rep} ⟦env-0 f⟧ = I_λ ⟦env-lex λu.(env-f rep-f)⟧

I_{rep} ⟦env-0 a⟧ = ■

= I_{rep} ⟦env-f°:(u↦■):env-0 rep-f⟧

où env-f° = I_{env} ⟦(u↦■):env-0 env-f⟧

(l'évaluation d'un environnement est ***simultanée***, avec la variante ***séquentielle***)

En définitive, une λ-expression est *quotée*, en ce sens que l'environnement lexical de définition est ignoré à l'application – ou plus justement il est inconnu. On a alors la petite variante dans les Lisps standards:

- ceux qui refusent les λ-expressions non quotées,
- et ceux qui les acceptent, et les évaluent identiquement à elles-mêmes.

Evaluation

I_{rep} ⟦env-0 (quote x)⟧ = x

I_{rep} ⟦env-0 (eval x)⟧

I_{rep} ⟦env-0 x⟧ = ※

= I_{rep} ⟦env-0 ※⟧

On a donc les quote et eval "purs", pour lesquels on n'indique pas l'environnement dans lequel on va évaluer – qui sera par défaut pris égal à celui dans lequel on est placé.

4. L'évaluation retardée: «Moi aussi»

Je présente ici le type d'évaluation que j'ai moi-même adopté.

4.1. termes

Définition

On commence par le plus simple, à savoir les définitions.

$\mathbf{I}_{env}$ ⟦env-0 (def x env rep)⟧
= (x ↦ $\mathbf{I}_{\lambda}$ ⟦env-0 λ.rep⟧)

c'est-à-dire qu'une définition est "vue" comme une λ-expression quotée sans argument (en particulier l'environnement de définition env est ignoré).

Symbole

L'utilisation d'un symbole *force* son évaluation:

$\mathbf{I}_{rep}$ ⟦env-0 x⟧
env-0: x ↦ $\mathbf{I}_{\lambda}$ ⟦env-lex λ.(env-x rep-x)⟧
env-x* = $\mathbf{I}_{env}$ ⟦env-x*:env-0 env-x⟧
rep-x* = $\mathbf{I}_{rep}$ ⟦env-x*:env-0 rep-x⟧
= rep-x*

(ici aussi l'évaluation parallèle est imparfaitement réalisée; la difficulté est cependant d'un autre ordre, puisque les définitions ne sont pas évaluées – cf. ref).

On remarquera en particulier que l'environnement d'appel est utilisé à l'évaluation du symbole et que l'environnement de définition est ignoré (inconnu): la recherche des identificateurs est *dynamique*.

λ-expression

De même qu'un symbole est vu comme une λ-expression quotée, une λ-expression est vue comme un symbole; c'est-à-dire qu'il n'y a pas à proprement parler de λ-expression: la notion est implicite.

4.2. application

Puisqu'il n'y a pas de λ-expression, la fonction apply sur un symbole s'identifie à la simple utilisation de ce symbole.

$\mathbf{I}_{rep}$ ⟦env-0 (apply x)⟧ = $\mathbf{I}_{rep}$ ⟦env-0 x⟧

Pour représenter le passage de paramètres, on introduit alors la notion d'*environnement local d'utilisation*:

(apply x env-loc)

qui s'interprète:

$\mathbf{I}_{rep}$ ⟦env-0 (apply x env-loc)⟧
env-loc* = $\mathbf{I}_{env}$ ⟦env-loc*:env-0 env-loc⟧
= $\mathbf{I}_{rep}$ ⟦env-loc*:env-0 x⟧

c'est-à-dire qu'on place, en *priorité* dans l'environnement d'évaluation, l'environnement local évalué en parallèle.

Ceci traduit bien une forme de passage de paramètres. Par exemple:

env-loc = ((def a 1))
$\mathbf{I}_{rep}$ ⟦env-0 (apply x env-loc)⟧
= $\mathbf{I}_{rep}$ ⟦(a↦1):env-0 x⟧

donc ici x est "vu" comme paramètre par le symbole a. La différence importante avec les λ-expressions est que le symbole qui représente le paramètre formel d'une λ-expression n'est plus anonyme dans le cas présent, puisqu'il est défini *par l'utilisateur*.

4.3. référence

Pour structurer les utilisations, on introduit un autre symbole "spécial": ref.

$\mathbf{I}_{env}$ ⟦env-0 (ref x)⟧
env-0: x ↦ $\mathbf{I}_{\lambda}$ ⟦env-lex λ.(env-x rep-x)⟧
env-x* = $\mathbf{I}_{env}$ ⟦env-x*:env-0 env-x⟧
= env-x*

qui signifie qu'on ne garde du symbole x que sa valeur d'environnement. [1]

1. C'est ici qu'intervient la *difficulté algorithmique* liée à l'évaluation parallèle des environnements: en effet si une définition n'est pas évaluée en revanche on évalue une référence.

De la même façon que pour une utilisation, on peut définir un environnement local de référence:

$\mathbf{I}_{env}$ ⟦env-0 (ref x env-loc)⟧
= $\mathbf{I}_{env}$ ⟦env-loc*:env-0 (ref x)⟧
où env-loc* = $\mathbf{I}_{env}$ ⟦env-loc*:env-0 env-loc⟧

4.4. symétrie d'évaluation

Etant donné un environnement: env-loc et les définitions:

```
(def a                          (def b
   env-a = env-loc                 env-b = ()
   rep-a = (apply x ()))           rep-b = (apply x env-loc))
```

a et b s'interprètent de manière semblable dans un même environnement, soit:

$\mathbf{I}_{rep}$ ⟦env-0 a⟧ = $\mathbf{I}_{rep}$ ⟦env-0 b⟧

On calcule:

(1) $\mathbf{I}_{rep}$ ⟦env-0 a⟧
env-0: a ↦ $\mathbf{I}_{\lambda}$ ⟦env-lex λ.(env-loc (apply x ()))⟧
env-a* = $\mathbf{I}_{env}$ ⟦env-a*:env-0 env-loc⟧
= $\mathbf{I}_{rep}$ ⟦env-a*:env-0 (apply x ())⟧
dans env-a*:env-0 :
l'environnement local d'utilisation est vide: env-loc-a*=()
on est donc placé dans l'environnement:
env* = ():(env-a*:env-0) = env-a*:env-0
= $\mathbf{I}_{rep}$ ⟦env* x⟧

(2) $\mathbf{I}_{rep}$ ⟦env-0 b⟧
env-0: b ↦ $\mathbf{I}_{\lambda}$ ⟦env-lex λ.(() (apply x env-loc))⟧
l'environnement local de définition est vide: env-b*=()
= $\mathbf{I}_{rep}$ ⟦env-b*:env-0 (apply x env-loc)⟧
dans env-b*:env-0 = ():env-0 = env-0 :
on évalue l'environnement local d'utilisation: env-loc
env-loc-b* = $\mathbf{I}_{env}$ ⟦env-loc-b*:env-0 env-loc⟧
on est donc placé dans l'environnement:
env'* = env-loc-b*:(():env-0) = env-loc-b*:env-0

On a env-a* = env-loc-b* donc env'* = env*
on trouve donc le *même* symbole x. [1]
= $\mathbf{I}_{rep}$ ⟦env* x⟧

5. Lisp parallèle: Symmetric Lisp

Références: sur le langage Symmetric Lisp [GJL 87a] [GJL 87c].
De même que dans Scheme les *fonctions* sont des objets de première classe [2], en Symmetric Lisp les *environnements* sont des objets de première classe. [3]
On n'a donc plus ici les deux types d'interprétation: $\mathbf{I}_{env}$ (environnement) et $\mathbf{I}_{rep}$ (représentation), puisqu'une *valeur* peut servir à modifier l'*environnement* d'évaluation.
On ne distingue donc plus pour une valeur les champs env et rep, et on appelle dans la suite une valeur indifféremment env ou rep selon le sens intuitif qu'on lui attache.

5.1. termes

Définitions
On a deux sortes de définitions:
• définition de *symbole*: def
$\mathbf{I}$ ⟦env-0 (def x rep)⟧
= (x ↦ $\mathbf{I}$ ⟦env-0 rep⟧)
• définition d'*environnement*: alpha
$\mathbf{I}$ ⟦env-0 (alpha env)⟧
env* = $\mathbf{I}$ ⟦env*:env-0 env⟧
= env*

1. On décrète que env-loc-b* = env-a* parce que les deux environnements sont limites d'une même suite: ceci est vrai si la limite existe, et autrement la question ne se pose pas.
2. La notion d'*objet de première classe* est une notion très informatique: elle caractérise les objets manipulés par un langage de programmation qui peuvent être: affectés, passés en paramètre d'une fonction, retournés par une fonction, ... Par exemple en Pascal, les entiers sont de première classe, mais non les pointeurs (ils ne peuvent pas être retournés par une fonction).
3. Au sens strict du terme, dans tous les langages Lisp une fonction est un objet de première classe – grâce entre autre à la fonction eval. Cependant Scheme offre en plus un modèle d'interprétation *cohérent*: un appel de fonction réalise toujours le même calcul quel que soit le point du programme d'où l'on appelle cette fonction. Pour ce faire, Scheme définit pour chaque fonction un unique contexte d'évaluation: ce contexte correspond très précisément au contexte lexical de définition de la fonction.

Ici on a *effectivement* une évaluation parallèle de l'environnement (aux difficultés algorithmiques près).

Symbole

Un symbole retourne sa valeur:

```
I [[env-0 x]]
= env-0(x)
```

Λ-expression

L'évaluation d'une λ-expression est retardée:

```
I [[env-0 (lambda(u) env)]]
= Iλ [[env-0 λu.env]]
```

qui est représentée formellement en Lisp identiquement à elle-même, mais a de fait une sémantique plus riche après son évaluation.

Utilisation contextuelle

La manipulation explicite des environnements permet de définir une *utilisation contextuelle*:

```
I [[env-0 (with e x)]]
      I [[env-0 e]] = e
= if alpha(e)
    then I [[e:env-0 x]]
    else I [[env-0 x]]
```

où `alpha(e)` teste si la valeur **e** est un environnement (`alpha`).

Pour réaliser cette fonction, il faut donc légèrement modifier les évaluations des définitions, de manière à *typer* les valeurs. Le seul véritable changement se situe au niveau de l'évaluation d'un symbole:

```
I [[env x]]
```

qu'on ne peut plus considérer comme l'appel fonctionnel (au sens *mathématique* du terme):

```
env(x)
```


et qu'on remplace donc par l'appel d'une fonction (Lisp par exemple):

```
(recherche x env)
```
[1]

5.2. manipulation d'environnement

Les environnements étant (presque) des objets Lisp, on peut leur appliquer des opérateurs classiquement définis sur des expressions Lisp.

Pour un terme e dont la forme évaluée est de type `alpha`:

e = I [[env-0 e]] = (alpha **x1**···**xn**)

sont définis les opérateurs suivants:

```
I [[env-0 (acar e)]] = x1
I [[env-0 (acdr e)]] = (alpha x2···xn)
I [[env-0 (alast e)]] = xn
```

(si **e** n'est pas de type `alpha`, la sémantique est non précisée).

5.3. application

```
I [[env-0 (apply f a)]]
      I [[env-0 f]] = Iλ [[env-lex λu.env-f]]
      I [[env-0 a]] = a
```

1. Par exemple:
- une définition: (x ↦ **x**) est remplacée par: (list 'def x **x**)
- un environnement: **env** = (x1 ↦ **x1**)···(xn ↦ **xn**) est remplacé par: (list 'alpha **env**)

(où dans **env** on remplace maintenant les définitions par leur équivalent en Lisp).

La fonction de recherche pourra être:

```
recherche (x env)
  (cond
    ((null env) ())
    ((eq (caar env) 'def)
       (if (eq (cadar env) x)
         (caddar env)
         (recherche x (cdr env))))
    ((eq (caar env) alpha)
       (or (eny (lambda (term)
                  (recherche x term))
                (cdar env))
           recherche x (cdr env)))))
```

```
= (car (last env-f*:(u↦a)))
      où env-f* = I [[env-f*:(u↦a):env-0 env-f]]
= I [[env-0 (alast
              (alpha
                (argdef u a)
                env-f))  ]]
```

argdef rappelle ici l'évaluation habituelle des arguments d'une fonction: l'argument est évalué *à l'extérieur* de l'environnement de la fonction appelée.
On notera que l'environnement lexical de la fonction est oublié, et qu'on conserve en revanche l'environnement d'appel à l'évaluation de la fonction: on se place donc dans le cas de la *liaison dynamique* des identificateurs (en particulier quote et eval sont définies à la manière d'un Lisp standard).

5.4. Les plambda-expressions

Il s'agit de λ-expressions qui retournent en valeur l'environnement de la fonction:

```
I [[env-0 (plambda(u) env)]] = I_π [[env-0 πu.env]]
```

et

```
I [[env-0 (apply f a)]]
      I [[env-0 f]] = I_π [[env-lex πu.env-f]]
      I [[env-0 a]] = a
= env-f*:(u↦a)
      où env-f* = I [[env-f*:(u↦a):env-0 env-f]]
= I [[env-0 (alpha
              (argdef u a)
              env-f)  ]]
```

5.5. Application partielle

Il s'agit du cas où on ne fournit qu'une partie des paramètres à l'application de la fonction. La sémantique est ici un peu plus difficile à donner.

On se donne une fonction f:

```
I [[env-0 f]]
= I_λ [[env-0 λu.λv.env]]
= (lambda(u v) env)
```


qu'on applique partiellement:

```
I [[env-0 (apply f a)]]
      I [[env-0 a]] = a
= I [[env-0 (alpha
              (argdef u a)
              (lambda(v) env))  ]]
= (alpha
    (def u a)
    (lambda(v) env))   ; qu'on note g
```

L'application de g sur b répond moins bien au schéma classique:

```
I [[env-1 (apply g b)]]
      I [[env-1 b]] = b
= I [[env-1 (alast
              (alpha
                (argdef v b)
                (def u a)
                env)  ]]
```

- on a l'apparition "automatique" de alast;
- on a toujours le cas particulier de argdef: la définition est évaluée en dehors de l'environnement;
- la définition de u n'est *sans doute* pas réévaluée;
- env est évalué en parallèle après évaluation des arguments (v mais aussi u)

5.6. environnements ouverts

On a enfin une autre notion, qui permet de construire *par étapes* un environnement.

Initialisation

Le symbole d'initialisation d'un environnement ouvert est: open-alpha

```
I [[env-0 (open-alpha)]]
= I_ω [[]]
```

qui se représente par:

```
(alpha *)
```

(et en particulier une forme I_ω est de "type" alpha).

Modification

Le symbole de modification est: `attach!`

$$\mathbb{I}[\![\texttt{env-0 (attach! e a)}]\!]$$
$$\mathbb{I}[\![\texttt{env-0 e}]\!] = \mathbb{I}_\omega[\![\texttt{env}]\!] = \texttt{e}$$
$$\mathbb{I}[\![\texttt{e:env-0 a}]\!] = \texttt{a}$$
$$= \mathbb{I}_\omega[\![\texttt{a:env}]\!]$$

Une utilisation intéressante de `attach!` consiste à *fournir* de nouvelles définitions.

Par exemple:

```
(def e (open-alpha))
(attach! e (def x 1))
(attach! e (def x 2))
(attach! e (def x 3))
...
```

construit un environnement qui énumère (tous) les entiers sur le symbole x.

Ceci introduit un problème qui n'a pas encore été abordé: puisqu'on effectue une évaluation parallèle des environnements, l'utilisation d'un symbole non défini ne doit pas provoquer d'erreur – au moins dans un premier temps. Avec la notion d'environnement ouvert, qu'on évalue aussi en parallèle, mais qu'on peut dynamiquement compléter, un tel cas d'utilisation de symbole non défini *ne doit pas* provoquer d'erreur; on a donc dans les environnements des définitions dont l'évaluation a été bloquée et qui doivent être évaluées au "bon moment".

Dans ce qui est réalisé, une évaluation bloquée retourne simplement le symbole non évalué. Le "bon moment" est alors le moment où on *réévalue* la définition *une deuxième fois*, ce qui ne va pas sans poser quelques problèmes.

Glossaire

application

C'est l'application d'une fonction sur ses arguments:
- notation traditionnelle: `M (e)`
- notation Lisp : `(M e)`
- notation du λ-calcul : `(M) e`
- notation adoptée : `(apply M e)`

On explicite l'application par le symbole `apply`.
Par exemple: `(apply λx.(+ x 3) 2)` retourne 5

contexte d'évaluation

voir environnement d'évaluation.

définition

C'est la définition, en Lisp, d'un nouveau symbole. Son évaluation retourne une valeur d'environnement.
Syntaxe: `(def x val-x)`
Par exemple: `(def x (let ((a 3)) (+ a 2)))` définit x comme un symbole de valeur 5

environnement

(d'évaluation) : c'est une fonction partielle d'interprétation des symboles, prolongée sur les termes par des règles qui varient selon les langages.
Par exemple: $(x \mapsto 1):(y \mapsto 2):(x \mapsto 3)$
est l'environnement qui à x associe la valeur 1 et à y la valeur 2.

des définitions (locales) : ce sont des définitions locales à la définition d'un symbole, en Lisp.
Par exemple:
```
(def x (let ((a 3))
         (lambda(u) (+ a u))))
```
définit x comme un symbole de valeur la λ-expression: `λu.(+ a u)`, où a vaut 3.

indéfini

C'est une valeur conventionnelle qui symbolise les cas d'erreur à l'évaluation des termes du λ-calcul.

interprétation

C'est la donnée d'un ensemble de *valeurs* **S** et d'un ensemble d'*environnements d'évaluation*. L'application d'une interprétation à un terme du λ-calcul retourne une *valeur*.
d'environnement : l'application retourne une *valeur d'environnement*.
de représentation : l'application retourne une *valeur de représentation*.

lambda-expression

C'est une abstraction sur un terme: on l'interprète intuitivement comme une fonction à appliquer sur ses arguments.
Par exemple: `λx.(+ x 1)` s'interprète intuitivement comme la fonction successeur.

parallèle (évaluation)

C'est l'évaluation d'une *liste de définitions* dans un *contexte d'évaluation* formé d'un contexte initial prolongé par cette *liste de définitions*; formellement, c'est la limite d'une suite d'évaluation.
Par exemple:
```
(define x 1)      est évalué en parallèle  (define x 1)
(define y x)                               (define y 1)
(define z (+ x y))                         (define z 2)
```

primitif (symbole, fonction)

Ce sont des symboles du λ-calcul qui ont une interprétation intuitive (+,−,···,1,2,3,···) et pour lesquels, dans une *interprétation*, on choisit l'interprétation intuitive – on identifie alors dans les notations le symbole et sa valeur d'interprétation.

prolongement (d'environnement)

C'est le prolongement d'un *environnement d'évaluation* sur l'ensemble des symboles du λ-calcul.
Par exemple: (x↦1):env prolonge la fonction env en x
(x↦2):(x↦1):env "prolonge" la fonction précédente, en *redéfinissant* la valeur de la fonction en x.

réflexif (domaine)

C'est un domaine interprété sur lui-même; Lisp est *presque* un domaine réflexif, exception faite des λ-expressions Lisp qui ne sont jamais tout à fait des éléments du langage.

représentation

C'est l'expression qui définit un nouveau symbole, abstraction faite des *environnements des définitions* qu'on trouve dans la définition du symbole.
Par exemple: (define x (let ((a 1))
(let ((b 2))
(+ a b))))
la définition de x est:
- l'environnement (a↦1)
- l'expression (let ((b 2)) (+ a b)) qu'on décompose à nouveau:
  - l'environnement (b↦2)
  - l'expression (+ a b)

(+ a b) est la *représentation* du symbole x.

simultanée (évaluation)

C'est l'évaluation d'une *liste de définitions* dans un *contexte d'évaluation* initial commun à toutes les évaluations.
Par exemple: (let ((x 1) (y x)) ···) est évalué en simultané (si x est défini auparavant par la valeur 2) (let ((x 1) (y 2)) ···)

symbole

Un symbole est un objet du λ-calcul:
- une lettre: a,b,c,···,x,y,z,···; en Lisp, il s'agit d'une *variable*;
- le symbole d'abstraction: λ, qui construit les λ-expressions.
  La syntaxe des λ-expressions est: λx.E, où x est une lettre et E un terme du λ-calcul; en Lisp, on écrirait plutôt: (lambda (x) E).
- le symbole d'application: apply.
  La syntaxe d'application est: (apply f x1···xn) où f,x1,···,xn sont des termes du λ-calcul
- les symboles primitifs: 1,2,3,···
- les fonctions primitives: +,−,···
  La syntaxe est par exemple: (+ x y), où x et y sont des termes du λ-calcul.
- en Lisp, le symbole de définition: def.
  La syntaxe de définition est: (def x val-x), où x est une lettre et val-x *un certain nombre* de termes du λ-calcul.

terme (du λ-calcul)

Un terme est un élément de S*, de la forme:
- t (atomique)
  t est soit une lettre, soit un symbole primitif (1,2,3,···)

- (f x1···xn)
  x1···xn sont des termes; f est très précisément:
  - lambda: x1 est alors une lettre, et on note:
    (lambda x1 x2···xn) = (lambda (x1) x2···xn) = λx1.x2···xn
  - apply: x1 est un terme qui représente une λ-expression;
  - +,−,···: une fonction primitive;
  - (en Lisp) def: x1 est alors une lettre.
  En particulier, le "terme" f ne peut être ni un symbole ni un terme du λ-calcul autres que ceux mentionnés ci-dessus. Un paramètre fonctionnel demande l'utilisation *explicite* du symbole apply.

valeur

C'est un élément du domaine d'interprétation **S**.
On note: **I** ⟦env t⟧
où env est un environnement d'évaluation et t un terme du λ-calcul.
d'environnement : c'est un élément de l'ensemble [S → **S**] des fonctions partielles qui à un symbole associent une valeur.
On note: $\mathbf{I}_{env}$ ⟦env t⟧
de représentation : c'est un élément de l'ensemble **S**; on introduit cette notion pour distinguer cette valeur de la valeur d'environnement d'un terme.
On note: $\mathbf{I}_{rep}$ ⟦env t⟧
de lambda-expression : c'est, formellement, un élément de **S**.
On note: $\mathbf{I}_{\lambda}$ ⟦env λx.E⟧

valeur (en Lisp)

Le domaine d'interprétation est l'ensemble:
S ∪ [S → **S**] ∪ [S → **S**]x**S***
- un terme s'interprète dans **S**: **I** ⟦env t⟧ = $\mathbf{I}_{rep}$ ⟦env t⟧
- une définition s'interprète dans [S → **S**]: **I** ⟦env t⟧ = $\mathbf{I}_{env}$ ⟦env t⟧
- une λ-expression s'interprète dans [S → **S**]x**S***: **I** ⟦env t⟧ = $\mathbf{I}_{\lambda}$ ⟦env λx.E⟧

Dire que Lisp est *réflexif*, c'est:
- ignorer les *valeurs d'environnement*; cela signifie:
  - de ne pas décomposer le calcul d'interprétation d'un terme en une interprétation d'environnement suivie d'une interprétation de représentation;
  - de se restreindre à des λ-expression *bien formées*, c'est-à-dire des termes qui ne contiennent pas d'occurrence de définition de symbole là où on n'en attend pas (cf. tous les langages présentés);
- ignorer les *valeurs de λ-expressions*, ce qui revient:
  - soit à poser: $\mathbf{I}_{\lambda}$ ⟦env λx.E⟧ = λx.E
    la λ-expression est *quotée* (cf. LeLisp par exemple);
  - soit à poser: **S** = [S → **S**]x**S***
    ce qui est mathématiquement possible, sachant qu'on ne considère que des fonctions partielles définies sur un nombre *fini* de symboles, mais ce qui ne se retrouve pas dans les langages Lisp présentés (en général, on ne peut pas évaluer une λ-expression évaluée).

Chapitre 5.4:

Comparaison critique

Après une comparaison des quatre langages «de la famille Lisp» présentés formellement dans la partie précédente, on donne les raisons qui ont guidé les choix d'évaluation des «textes».

1. Qualités d'un langage

On compare ici les quatre langages «de la famille Lisp» présentés précédemment. On va voir comment chacun satisfait aux *«critères de qualités d'un langage informatique»* tels qu'ils sont énumérés dans [Mey 80]. On ne s'attachera pas ici à la *syntaxe* des langages, qui est dans tous les cas une syntaxe Lisp, c'est-à-dire une syntaxe dont la simplicité a été poussée à l'extrême, sinon à l'excès. On regardera donc plutôt la *sémantique* donnée à l'évaluation des expressions Lisp de ces langages.

critère	*Scheme*	*LeLisp*	*Symmetric Lisp*	*«Moi aussi»*
homogénéité régularité	Une lambda-expression évaluée n'est pas un objet du langage (un appel à la fonction eval sur un tel objet échoue).	Les lambda-expressions ne sont pas des objets de première classe, et ne peuvent donc être manipulées de la même manière que les variables	Pour les lambda-expression, cf LeLisp.	Oui, du fait de la simplicité des concepts
orthogonalité	L'évaluation parallèle des «environnements»: l'évaluation est *parallèle* pour des définitions de lambda-expressions, mais non pour des variables.	Le terme de tête des expressions *n'est pas évalué*, ce qui nécessite l'emploi explicite de la fonction apply quand on utilise un paramètre fonctionnel.	L'évaluation parallèle des «environnements»: - pour les symboles, un symbole indéfini est *virtuellement* quoté, - pour les lambda-expressions évaluées avec un nombre partiel d'arguments, l'évaluation répond mal au schéma général.	L'évaluation parallèle des «environnements»: *l'ordre des déclarations* importe. Deux environnements qui ne diffèrent que par l'ordre des déclarations qu'ils comportent pourront être évalués différemment.
simplicité	Oui, si l'on ne fait pas de *calcul symbolique* (la fonction eval, qui évalue une expression symbolique, nécessite en paramètre un environnement, ce qui rend son emploi malaisé).	Oui, "par omission" des situations complexes (dont: une lambda-expression est évaluée identiquement à elle-même).	Beaucoup de concepts, beaucoup de cas dont la sémantique n'est pas précisée (par ex: acar sur un argument qui n'est pas de type alpha).	Oui. Ce qui pourrait se révéler être une complication: on peut quasiment tout faire, donc le respect d'une *discipline* s'impose.

critère	*Scheme*	*LeLisp*	*Symmetric Lisp*	*«Moi aussi»*
généralité	Oui, dans le "créneau" de la programmation structurée: - portée locale des déclarations, - possibilité de définir des «modules» (les environnements) = des fonctions + des données.	Oui, dans le "créneau Lisp": - calcul symbolique, - manipulation des listes.	Oui, cf LeLisp.	Non, on s'intéresse spécifiquement à la manipulation de caractères.
extensibilité	Oui, les lambda-expressions sont des objets de première classe, on les utilise donc *de la même façon* que les fonctions prédéfinies. Non, pas de *variable libre*, la fonction eval demande en argument un environnement.	Oui, avec: - les macros Lisp - la possibilité de définir des *variables libres* - la fonction eval	Oui, cf LeLisp. Egalement, la manipulation des environnements.	Oui, avec: - la liaison dynamique. - la possibilité de *référence* à un environnement.
compilabilité "exécutabilité"	Exécutable, et surtout *compilable*. C'est un des arguments avancés pour le choix de la «liaison statique».	Exécutable, "par omission". Difficilement compilable (en particulier, le compilateur *refuse* la définition dynamique de fonctions «flet», qu'il ne peut reconnaître comme les lambda-expressions).	Exécutable, mais certains comportements restent indéfinis (en particulier: cas de cycles dans l'évaluation parallèle d'un environnement). Difficilement compilable Problèmes des environnements ouverts («open-alpha»): comment gérer l'espace mémoire.	Exécutable, et sûrement pas compilable (mais certainement "optimisable", en particulier lors de la reconstruction dynamique de l'environnement lexical de définition).

critère	*Scheme*	*LeLisp*	*Symmetric Lisp*	*«Moi aussi»*
clarté	Oui, du fait de la fermeture lexicale («liaison statique»): dans une *vision descendante* du programme, on a la garantie du lien lexical entre définition et utilisation d'un symbole.	Oui, si on excepte les «effets de bord imprévisibles», qui sont le fait de la «liaison dynamique» des symboles.	Oui, mais cf LeLisp. De plus, la gestion *dynamique* des environnements accroît le nombre des cas «imprévisibles».	Oui, mais cf Symmetric Lisp, *à l'intérieur* d'une définition; *à l'extérieur*, la gestion de l'«environnement local» garantit la construction dynamique de l'environnement lexical de définition.
sécurité fiabilité	Oui, dans une *vision descendante* du programme: les liens sont *statiquement* identifiés.	Oui, exceptés les difficultés inhérentes à la «liaison dynamique» (les «effets de bord»). Le *bon emploi* des variables libres est laissé à la discrétion de l'utilisateur.	Oui, si l'on suppose une méthodologie d'emploi associée. L'utilisateur gérant dynamiquement les environnements, peu de contrôles statiques sont possibles	Oui, mais cf Symmetric Lisp; une première "méthodologie" est l'emploi des «environnements locaux».
souplesse commodité d'emploi	Oui, pour la définition de «modules» (des fonctions + des données). Non, pour le calcul symbolique, du fait de la contrainte de la «liaison statique».	Il est difficile d'attacher à un symbole à valeur fonctionnelle un environnement de définition. Les données à valeur persistante sont *explicitement* déclarées («closure»)	Oui, si l'on *gère bien* les environnements.	Oui, mais cf Symmetric Lisp. La nécessité d'indiquer les «environnements locaux» peut être contraignante (beaucoup d'empilement d'environnements pour nommer un concept).

critère	*Scheme*	*LeLisp*	*Symmetric Lisp*	*«Moi aussi»*
puissance expressive	Oui, une lambda-expression représente un «module». Non, pour le calcul symbolique (et en particulier, *pas de variable libre*).	Oui, excepté le problème de la fermeture lexicale, difficile à mettre en œuvre. Inversement la «liaison dynamique» favorise l'utilisation de *variables libres*, qui autorisent la définition d'«abstractions de haut niveau».	Oui, mais cf LeLisp: demande une *bonne gestion* des environnements par l'utilisateur. Oui, avec les environnements ouverts («open-alpha») pour la manipulation de flots de données infinis.	Oui, mais cf Symmetric Lisp, *à l'intérieur* d'une définition. *A l'extérieur*, on reconstruit dynamiquement l'environnement lexical, et l'utilisateur peut (volontairement) fausser cette reconstruction – surcharge de définition.
complétude de la définition	Oui, les cas d'erreur sont identifiés. Ce sont: l'évaluation parallèle de définitions de variables mutuellement définies; l'évaluation d'une lambda-expression évaluée.	Oui, "par omission".	Oui, si l'on suppose que les formules non valides retournent une valeur conventionnelle.	Oui, du fait du peu de concepts introduits.

2. La traduction pour «Moi Aussi»

On montre ici comment se traduiraient les particularités des évaluateurs Lisp présentés dans l'évaluateur qu'on a défini.

2.1. la fermeture lexicale

Pour réaliser la fermeture lexicale, il faut modifier la *recherche* d'une définition. Telle qu'elle est définie, la recherche retourne la définition isolée. On réalise alors une "sorte" de fermeture lexicale en plaçant dans le contexte d'évaluation l'environnement local qui a permis de trouver cette définition. Pour la *vraie* fermeture lexicale, on a besoin de connaître le contexte de définition du texte trouvé: la recherche retourne donc ce contexte, et on évalue ensuite dans le contexte construit en plaçant en priorité le contexte de définition.

On a alors l'alternative suivante:
– ou bien, à la façon de Scheme, on ne garde que le contexte de définition;
– ou bien on place le contexte de définition *puis* le contexte courant d'évaluation: tout ce qui est défini "du point de vue" de la définition du textes reste défini "du point de vue" de son utilisation; une variable libre à la définition est "vue" libre à l'utilisation.

La première solution fait perdre la notion de *variable libre*, ce qui me paraît regrettable. La deuxième gène une *vue locale* des définitions de texte. Par exemple, le texte:

```
def txt
    = (use a) "," (use b)
```

semble être défini avec les deux variables libres a et b. Mais, si l'on regarde «plus haut», on peut découvrir:

```
def X
    def a = "A"
    def txt = ...
```

qui fait apparaître que a est en fait liée à la valeur "A", et que b semble toujours libre.

On privilégie donc *l'environnement global* dans lequel on évalue, qui est le «dernier environnement» où l'on puisse «remonter», quand on regarde les définitions de textes: c'est le seul environnement où l'on a l'assurance qu'une variable est bien libre. Cette approche se traduit en Scheme par la nécessité, à l'appel de la fonction eval, de fournir un environnement d'évaluation, qui tient lieu d'environnement global. Elle nuit à l'homogénéité des concepts, et n'a pas été retenue pour cela.

2.2. la liaison dynamique

Dans un Lisp standard, les lambda-expressions sont *quotées*, nécessairement ou virtuellement. De ce fait on "oublie" le chemin d'accès à la définition d'une lambda-expression: on réalise ceci en "oubliant" de placer, dans le contexte d'évaluation, l'environnement local d'utilisation ou de référence, qui est utilisé pour *reconstruire* le contexte de définition d'un texte. Cet "oubli" rend plus difficile la construction d'une hiérarchie de définitions, puisque une couche se définit uniquement par des «propriétés» – des fonctions – et non plus un «état» – des valeurs données de certains symboles. On ne peut bénéficier d'aucune information contextuelle dans une définition.

On peut noter que la difficulté se retrouve dans l'évaluateur des «textes», dans le cas de l'utilisation d'un *texte englobant*: on n'a en effet pas besoin d'indiquer de «chemin d'accès» à la définition du texte, ce qui signifie qu'on peut "oublier" des propriétés locales à la définition – ceci peut se révéler en partie intéressant, mais n'aura pas toujours un résultat heureux.

2.3. les environnements comme objets de première classe

La distinction qu'on introduit, dès le départ, entre l'environnement et la représentation d'un texte, a pour conséquence qu'on ne peut considérer les environnements comme des objets de première classe.

Pour se faire, il faudrait *appauvrir* la *syntaxe initiale*, en ne définissant que deux phyla:
– le phylum des déclarations: DCL (qui correspond aux phyla ENV et REP),
– le phylum des éléments d'une déclaration: ELT (c'est TRM et ATM).
La syntaxe initiale s'écrit alors:

```
(1)  DCL ::= dcl
(2)        dcl -> ELT*...
(3)  ELT ::= DCL LSP def uti
(4)        def -> NOM DCL
(5)        uti -> NOM DCL
(11) LSP ::= lsp
(12)       lsp -> SEX*...
(13) SEX ::= ELT ATOME LISTE
```

Le problème alors est de *bien interpréter* un opérateur:
– l'opérateur def a un champ DCL dont les premiers éléments sont des définitions, et dont le dernier élément est une valeur;
– l'opérateur uti a un champ DCL qui est un environnement local d'utilisation (tous ses éléments sont donc des définitions), et l'utilisation selon les cas s'intéressera aux premiers éléments ou au dernier élément du texte nommé.

Par exemple, le texte:

```
(def txt
  (dcl
    (def a (dcl  "A"))
    (def b (dcl  "B"))
    (dcl  (uti a) "," (uti b))))
```

doit être compris comme:
– un texte qui définit localement les textes a et b
– et dont la valeur d'utilisation est: (dcl (uti a) "," (uti b))
En particulier, s'intéresser à la valeur d'un texte, c'est forcer l'évaluation du dernier élément, par une fonction explicitement nommée, par exemple alast:

```
(alast (uti txt))
```

L'inconvénient qu'on peut trouver à une telle simplification est qu'elle paraît *fragiliser* le système: une "mauvaise interprétation" d'un texte conduit à des cas qui n'ont intuitivement pas de sens, et auxquels justement on ne pourra pas donner de sémantique précise. Le choix retenu a été de *typer* les valeurs d'environnement et de représentation en distinguant les phyla ENV et REP, ce qui ramène d'une façon assez simple les cas qui ont une sémantique imprécise à des cas d'erreur de type.

Chapitre 5.5:

Construction de la Syntaxe Abstraite

On donne ici la façon dont on peut construire la *syntaxe initiale* telle qu'elle a été présentée précédemment. On y trouve les fonctions permettant de créer, compléter ou supprimer les opérateurs et les phyla d'une syntaxe abstraite.

1. la syntaxe de la syntaxe abstraite

1.1. définitions

Un *arbre syntaxique* abstrait est un arbre:
- dont les nœuds sont étiquetés par des symboles: ces symboles sont les noms d'opérateurs de la syntaxe abstraite,
- dont les feuilles sont soit des opérateurs d'arité nulle, soit des terminaux instanciables.

Un *opérateur* est un modèle de définition de nœud; il est donné par:
- un nom: c'est le nom de l'opérateur;
- une liste ordonnée de phyla: pour un arbre A dont la racine est étiquetée par un opérateur O, le i-ième fils de la racine de A est l'instance d'un opérateur qui doit appartenir au i-ième phylum de définition de l'opérateur O.

Un *nœud* de l'arbre syntaxique est alors une instance de l'opérateur qui étiquette ce nœud.

Un *phylum* est un sous-ensemble de l'ensemble des opérateurs définis. La notion permet d'imposer, sur un arbre donné et pour un fils donné de la racine, que ce fils soit une instance d'un opérateur appartenant à un sous-ensemble restreint de l'ensemble de tous les opérateurs définis.

1.2. opérateurs

On distingue fondamentalement trois sortes d'opérateurs:

• *opérateur d'arité fixe*
Un opérateur d'arité fixe est défini par un nombre N fixé de phyla. Toute instance de cet opérateur est alors un nœud qui possède exactement N fils. On note symboliquement:

```
oper -> PHYL1 ... PHYLN ;
```

• *opérateur de liste*
Un tel opérateur est défini par un unique phylum, sachant qu'une instance de cet opérateur est un nœud qui possède un nombre quelconque de fils, chaque fils étant l'instance d'un opérateur appartenant à cet unique phylum. On note symboliquement:

```
oper -> PHYL *... ;
```

• *terminal instanciable*
C'est un opérateur d'arité nulle, et qui possède une valeur. Le type de la valeur est fonction du terminal, et n'est pas exprimé au niveau de la syntaxe abstraite. On note symboliquement:

```
oper -> implemented as PHYL ;
```

où PHYL désigne un type de valeur prédéfini.

1.3. phyla

Un phylum est un sous-ensemble de l'ensemble des opérateurs. Il se définit donc par la donnée d'une liste, non ordonnée, d'opérateurs ou de phyla. On note symboliquement:

```
PHYL ::= PHYL1 ... PHYLN oper1 ... operM ;
```

Les opérateurs oper1 ... operM sont déclarés appartenir au phylum PHYL; les opérateurs appartenant aux phyla PHYL1 ... PHYLN appartiennent aussi au phylum PHYL.

1.4. remarque

Avec les notations ensemblistes, les définitions s'expriment comme suit:

• *opérateur d'arité fixe*
```
oper -> PHYL1...PHYLN ;
```
donne:
```
oper ∈ PHYL1 x...x PHYLN
```
(produit cartésien des ensembles PHYL1...PHYLN).

• *opérateur de liste*
```
oper -> PHYL *... ;
```
donne:

$$\text{oper} \in \overset{\infty}{\bigcup_{n=0}} \text{PHYL}^n$$

(ensemble des suites finies à valeur dans PHYL).

• *terminal instanciable*
```
oper -> implemented as PHYL ;
```
donne:
```
oper ∈ PHYL
```
(PHYL est l'ensemble des valeurs que peut prendre le terminal instanciable).

• *phylum*
```
PHYL ::= PHYL1 ... PHYLN oper1 ... operM ;
```
donne:
```
PHYL = PHYL1 ∪...∪ PHYLN ∪ { oper1, ..., operM }
```
(PHYL est la réunion des ensembles PHYL1...PHYLN et de l'ensemble donné en extension qui contient les opérateurs oper1 ... operM).

2. Le graphe des phyla

2.1. position du problème

Dans les parties précédentes, on a dégagé les attributs de définition des opérateurs:
- évaluation de texte,
- recherche d'une définition de texte dans un contexte,
- complétion d'une liste,
- impression complète ou concise,
- entrée dans les champs d'un opérateur,
- sortie des champs d'un opérateur,
- impression évaluée.

Comme il a été dit, ces attributs sont des fonctions Lisp, «fournies en standard» pour la *syntaxe initiale*, et qu'il faut soi-même écrire quand on complète la *syntaxe initiale*. Ceci signifie que la dérivation d'un opérateur:

```
opérateur -> phylum ...
```

n'est pas exprimée de quelque manière, et que c'est au concepteur d'un nouvel opérateur de traduire, dans le texte des fonctions d'attributs, cette propriété. En particulier, on n'a pas à indiquer la syntaxe concrète attachée à la syntaxe abstraite définie, puisque c'est le concepteur qui complète, à chaque définition d'un opérateur, l'analyseur syntaxique et le décompilateur d'arbre, ces deux outils réalisant la passerelle entre la syntaxe concrète – le texte qui s'affiche – et la syntaxe abstraite – la représentation interne sous forme d'arbre.

En revanche on permet d'exprimer, sous une forme synthétique, les dérivations de phyla. Les attributs d'un opérateur sont en effet attachés à cet opérateur, indépendamment des phyla. Donc une fois définis ces attributs, on peut juger raisonnable d'attendre qu'un "traitement automatique" – c'est-à-dire l'exécution d'un programme – sache retrouver, pour chaque phylum, un attribut donné d'un opérateur donné.

Pour ce faire, on construit le graphe des phyla:
Pour chaque phylum, on connaît alors:
- la liste des opérateurs appartenant en propre au phylum,
- la liste des opérateurs appartenant indirectement au phylum – ils lui appartiennent parce qu'ils appartiennent à des phyla contenus dans le phylum –,
- la liste des opérateurs qui n'appartiennent pas au phylum.

2.2. attributs d'opérateurs

C'est une liste Lisp de format nécessaire:

```
(evalt recht cpltt (imptt imptc) entre sorte evale)
```

c'est donc une liste à 7 éléments, dont le quatrième est une paire.

2.3. attributs de phyla

C'est une liste Lisp de format nécessaire:

```
(L-recht  L-cpltt  L-imptt  L-entre  L-sorte  L-evale)
```

chaque élément de la liste ayant la forme:

```
(attribut-nulle  attribut-atome  attribut-liste)
```

c'est donc une liste à 6 éléments, chaque élément étant un triplet.

2.4. fonctions utilisateur

defphyl <nomphyl> <attphyl>
définit un nouveau phylum de nom <nomphyl> d'attributs <attphyl>
Les arguments ne sont pas évalués.
putphyl <nomphyl> <attphyl>
identique à defphyl.
Les arguments sont ici évalués.
getphyl <nomphyl>
retourne les attributs du phylum de nom <nomphyl>, s'il existe.
remphyl <nomphyl>
supprime le phylum de nom <nomphyl>, s'il existe.

defoper <nomoper> <attoper>
définit un nouvel opérateur de nom <nomoper> d'attributs <attoper>
Les arguments ne sont pas évalués.
putoper <nomoper> <attoper>
identique à defoper.
Les arguments sont évalués.
getoper <nomoper>
retourne les attributs de l'opérateur de nom <nomoper>, s'il existe.
remoper <nomoper>
supprime l'opérateur de nom <nomoper>, s'il existe.

defoperphyl <nomphyl> <nomoper1> ... <nomoperN>
déclare les opérateurs de noms <nomoper1> ... <nomoperN> comme appartenant *en propre* au phylum de nom <nomphyl>.
Les arguments ne sont pas évalués.
putoperphyl <nomphyl> <nomoper1> ... <nomoperN>
identique à defoperphyl.
Les arguments sont évalués.
getoperphyl <nomphyl>
retourne la liste des opérateurs qui appartiennent *en propre* au phylum de nom <nomphyl>.
remoperphyl <nomphyl> <nomoper1> ... <nomoperN>
supprime du phylum de nom <nomphyl> les opérateurs de noms <nomoper1> ... <nomoperN>.

defperephyl <nomphyl> <nompere1> ... <nompereN>
déclare les phyla de noms <nompere1> ... <nompereN> comme pères du phylum de nom <nomphyl>. Le phylum <nomphyl> est alors *contenu* dans les phyla <nompere1> ... <nompereN>.
Les arguments ne sont pas évalués.
putperephyl <nomphyl> <nompere1> ... <nompereN>
identique à defperephyl.
Les arguments sont évalués.
getperephyl <nomphyl>
retourne la liste des phyla pères du phylum de nom <nomphyl>.
remperephyl <nomphyl> <nompere1> ... <nompereN>
supprime du phylum de nom <nomphyl> les phyla pères de noms <nompere1> ... <nompereN>.

deffilsphyl <nomphyl> <nomfils1> ... <nomfilsN>
déclare les phyla de noms <nomfils1> ... <nomfilsN> comme fils du phylum de nom <nomphyl>. Le phylum <nomphyl> *contient* alors les phyla <nomfils1> ... <nomfilsN>.
Les arguments ne sont pas évalués.
putfilsphyl <nomphyl> <nomfils1> ... <nomfilsN>
identique à deffilsphyl.
Les arguments sont évalués.
getfilsphyl <nomphyl>
retourne la liste des phyla fils du phylum de nom <nomphyl>.
remfilsphyl <nomphyl> <nomfils1> ... <nomfilsN>
supprime du phylum de nom <nomphyl> les phyla fils de noms <nomfils1> ... <nomfilsN>.

2.5. remarques

Pour un simple problème algorithmique, le graphe ne peut être cyclique.
Ceci interdit la situation suivante:

```
PHYL1 ::= oper1 PHYL2 ;
PHYL2 ::= oper2 PHYL1 ;
```

qu'on pourrait raisonnablement comprendre:

```
PHYL1 ::= oper1 oper2 ;
PHYL2 ::= oper1 oper2 ;
```

Il y aura donc l'envoi d'un message d'erreur, lors d'une tentative de formation d'un cycle, à l'appel d'une des fonctions defperephyl, putperephyl, deffilsphyl, putfilsphyl.

Préalablement à l'emploi des fonctions données ci-dessus, il faut définir deux variables globales: **phylt** et **opert**, qui représentent les têtes des listes respectivement des phyla et des opérateurs:

```
(setq phylt '(phylt) opert '(opert))
```


3. construction du graphe

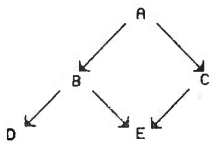
3.1. présentation

Le graphe est construit en synthétisant sur chaque nœud, c'est-à-dire pour chaque phylum, le *degré de connexité* du nœud. Le degré de connexité compte le nombre de fois qu'un nœud est visible depuis un autre nœud.
Par exemple, la figure 1.

On conserve par ailleurs l'information de *lien de filiation* directe, qui rappelle qu'un phylum est directement le fils d'un autre.
Par exemple, la figure 2.

On peut remarquer que tout phylum est en relation directe avec lui-même, ce qui correspond au choix d'initialisation retenu. On notera de plus que la valeur peut être supérieure à un, rien n'interdisant qu'un phylum soit "plusieurs fois" fils ou père d'un autre.

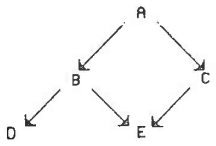
figure 1



degré de connexité

	A	B	C	D	E
A	1	0	0	0	0
B	1	1	0	0	0
C	1	0	1	0	0
D	1	1	0	1	0
E	2	1	1	0	1

figure 2



lien de filiation

	A	B	C	D	E
A	1	0	0	0	0
B	1	1	0	0	0
C	1	0	1	0	0
D	0	1	0	1	0
E	0	1	1	0	1

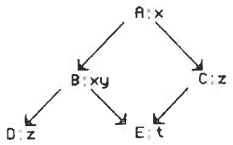
On construit parallèlement le *"degré d'appartenance"* d'un opérateur à un phylum, qui mesure le nombre de fois qu'un opérateur est visible dans un phylum.
Par exemple, la figure 3.

Le degré d'appartenance tient compte de la présence ou non d'un opérateur dans un phylum, mais également de sa présence dans un phylum contenu, pondérée alors par le degré de connexité correspondant.

On conserve enfin l'information d'*appartenance en propre* d'un opérateur à un phylum, en particulier pour le choix de l'attribut d'impression, qui sera concise ou complète selon que l'opérateur appartiendra ou non en propre au phylum dans lequel on imprime.
Par exemple, la figure 4.

Cette valeur aussi peut être supérieure à un.

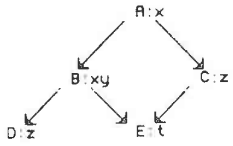
figure 3



degré d'appartenance

	A	B	C	D	E
x	2	1	0	0	0
y	1	1	0	0	0
z	2	1	1	1	0
t	2	1	1	0	1

figure 4



appartenance en propre

	A	B	C	D	E
x	1	1	0	0	0
y	0	1	0	0	0
z	0	0	1	1	0
t	0	0	0	0	1

3.2. matrice des phyla

Notations

On note dans la suite :

$A = [a_{i,j}]$ la matrice des degrés de connexité

$B = [b_{i,j}]$ la matrice des liens de filiation

$\delta_{i,j} = 1$ si $i=j$

$\delta_{i,j} = 0$ si $i \neq j$ *(symboles de Kronecker)*

$a_{i,j}$ se lit: «i est fils de j avec un poids $a_{i,j}$»

$b_{i,j}$ se lit: «i est directement fils de j avec un poids $b_{i,j}$»

3.2.1. matrices compatibles

(A,B) est un couple de matrices *compatible* si:

$$\sum_k b_{i,k}\, a_{k,j} = \sum_l a_{i,l}\, b_{l,j} = a_{i,j} - \delta_{i,j}$$

qui signifie que, au $\delta_{i,j}$ près:

- le nombre de fois que i est fils de j est égal à la somme sur k du nombre de fois que k est fils de j, pondéré par le nombre de fois que i est directement fils de k;
- le nombre de fois que j est père de i est égal à la somme sur l du nombre de fois que l est père de i, pondéré par le nombre de fois que j est directement père de l.

Remarque:

Avant toute chose, il faut savoir si la notion est ***pertinente***, c'est-à-dire s'il existe au moins un couple de matrices (A,B) qui vérifie la condition de compatibilité.
Il en existe. On peut prendre par exemple pour A la matrice identité et pour B la matrice nulle.

3.2.2. transformation directe

Soit T_{pq} la transformation:

$$T_{pq}:\ \alpha_{i,j} = a_{i,j} + a_{i,p}a_{q,j}$$
$$\beta_{i,j} = b_{i,j} + \delta_{i,p}\delta_{q,j}$$

T_{pq} intuitivement rajoute l'arc «p est fils de q».
Alors $T_{pq}(A,B)$ est compatible si $a_{qp} = 0$.

On remarquera que la condition signifie très précisément :

«on reste compatible tant qu'on n'introduit pas de cycle.»

Calcul:

$$\sum \beta_{ik}\,\alpha_{kj}$$
$$= \sum (b_{ik} + \delta_{ip}\,\delta_{qk})\,(a_{kj} + a_{kp}\,a_{qj})$$
$$= \sum b_{ik}\,a_{kj} + \delta_{ip}\,(a_{qj} + a_{qp}\,a_{qj}) + \sum b_{ik}\,a_{kp}\,a_{qj}$$
$$= (a_{ij} - \delta_{ij}) + \delta_{ip}\,a_{qj} + (a_{ip} - \delta_{ip})\,a_{qj}$$
$$= (a_{ij} + a_{ip}a_{qj}) - \delta_{ij} = \alpha_{ij} - \delta_{ij}$$

$$\sum \alpha_{ik}\,\beta_{kj}$$
$$= \sum (a_{ik} + a_{ip}\,a_{qk})\,(b_{kj} + \delta_{kp}\,\delta_{qj})$$
$$= \sum a_{ik}\,b_{kj} + \delta_{qj}\,(a_{ip} + a_{ip}\,a_{qp}) + \sum a_{ip}\,a_{qk}\,b_{kj}$$
$$= (a_{ij} - \delta_{ij}) + \delta_{qj}\,a_{ip} + a_{ip}\,(a_{qj} - \delta_{qj})$$
$$= (a_{ij} + a_{ip}a_{qj}) - \delta_{ij} = \alpha_{ij} - \delta_{ij}$$

3.2.3. transformation inverse

Soit S_{pq} la transformation:

S_{pq}: $\tilde{a}_{ij} = a_{ij} - a_{ip}a_{qj}$
$\tilde{b}_{ij} = b_{ij} - \delta_{ip}\delta_{qj}$

S_{pq} intuitivement supprime l'arc «p est fils de q».
Alors $S_{pq}(A,B)$ est compatible si $a_{qp} = 0$.
De plus, on a : $S_{pq}\,T_{pq}\,(A,B) = (A,B)$

Calcul:
La compatibilité de $S_{pq}(A,B)$ se démontre de la même façon que pour T_{pq}.

On calcule: $S_{pq}\,T_{pq}\,(A,B)$
$\tilde{\alpha}_{ij} = \alpha_{ij} - \alpha_{ip}\,\alpha_{qj}$
$= (a_{ij} + a_{ip}\,a_{qj}) - (a_{ip} + a_{ip}\,a_{qp})\,(a_{qj} + a_{qp}\,a_{qj})$
$= a_{ij} + a_{ip}\,a_{qj} - a_{ip}\,a_{qj} = a_{ij}$
(sachant que (A,B) est compatible)

3.2.4. commutativité

Soit (A,B) un couple de matrices compatible.

U_{pq}: $\tilde{a}_{ij} = a_{ij} + x\,a_{ip}a_{qj}$
$\tilde{b}_{ij} = b_{ij} + x\,\delta_{ip}\delta_{qj}$
V_{pq}: $\tilde{a}_{ij} = a_{ij} + y\,a_{im}a_{nj}$
$\tilde{b}_{ij} = b_{ij} + y\,\delta_{im}\delta_{nj}$
avec x et y = 1 ou -1.

Alors :

$$U_{pq}\,V_{mn} = V_{mn}\,U_{pq}$$

si les transformations préservent la compatibilité.

Calcul
On note:

$U_{pq}\,(A) = [\,Ua_{ij}\,]$ $V_{mn}\,U_{pq}\,(A) = [\,VUa_{ij}\,]$
$V_{mn}\,(A) = [\,Va_{ij}\,]$ $U_{pq}\,V_{mn}\,(A) = [\,UVa_{ij}\,]$

$VUa_{ij} = (Ua_{ij}) + y\,(Ua_{im})\,(Ua_{nj})$
$= (a_{ij} + x\,a_{ip}\,a_{qj})$
$+ y\,(a_{im} + x\,a_{ip}\,a_{qm})\,(a_{nj} + x\,a_{np}\,a_{qj})$
$= (a_{ij} + x\,a_{ip}\,a_{qj} + y\,a_{im}\,a_{nj})$
$+ xy\,(a_{ip}\,a_{qm}\,a_{nj} + a_{im}\,a_{np}\,a_{qj})$
$+ xy^2\,a_{np}\,a_{qm}\,a_{ip}\,a_{qj}$

$UVa_{ij} = (Va_{ij}) + x\,(Va_{ip})\,(Va_{qj})$
$= (a_{ij} + y\,a_{im}\,a_{nj})$
$+ x\,(a_{ip} + y\,a_{im}\,a_{np})\,(a_{qj} + y\,a_{qm}\,a_{nj})$
$= (a_{ij} + x\,a_{ip}\,a_{qj} + y\,a_{im}\,a_{nj})$
$+ xy\,(a_{im}\,a_{np}\,a_{qj} + a_{ip}\,a_{qm}\,a_{nj})$
$+ x^2y\,a_{np}\,a_{qm}\,a_{im}\,a_{nj}$

Les termes croisés sont respectivement :
$xy^2\,a_{np}\,a_{qm}\,a_{ip}\,a_{qj}$ et $x^2y\,a_{np}\,a_{qm}\,a_{im}\,a_{nj}$

Ils sont nuls, puisque par compatibilité on a:

U_{pq} est compatible sur (A,B), donc: $a_{qp} = 0$

V_{nm} est compatible sur U_{pq} (A,B), donc: $Ua_{nm} = 0 = a_{nm} + x\ a_{np}\ a_{qm}$

V_{nm} est compatible sur (A,B), donc: $a_{nm} = 0$

U_{pq} est compatible sur V_{nm} (A,B), donc: $Va_{qp} = 0 = a_{qp} + y\ a_{qm}\ a_{np}$

ce qui montre deux fois que: $a_{np}\ a_{qm} = 0$

On remarquera qu'ici la condition de compatibilité se traduit par le fait que dans le graphe:

$p \rightarrow q$ (concerne l'application de U_{pq})

$m \rightarrow n$ (concerne l'application de V_{mn})

on ne doit pas introduire de cycle: la condition $a_{np}\ a_{qm} = 0$ signifie bien qu'on ne doit pas construire de cycle *en deux étapes*.

Note :

Avec trois transformations (pq,x) (mn,y) (rs,z), la commutativité imposerait:

$$a_{sr} + x\ a_{sp}\ a_{qr} + y\ a_{sm}\ a_{nr} + xy\ (a_{sm}\ a_{np}\ a_{qm} + a_{sp}\ a_{qm}\ a_{nr}) + xy^2\ a_{sm}\ a_{np}\ a_{qm}\ a_{nr} = 0$$

Les paramètres étant quelconques, on en déduit :

(a) $a_{sr} = 0$

(b) $a_{sp}\ a_{qr} = 0$

(c) $a_{sm}\ a_{nr} = 0$

(d) $a_{sm}\ a_{np}\ a_{qm} = 0$

(e) $a_{sp}\ a_{qm}\ a_{nr} = 0$

(f) $a_{sm}\ a_{np}\ a_{qm}\ a_{nr} = 0$

On retrouve donc les conditions de formation d'un cycle en trois étapes – plus particulièrement dans les cas *(d)* et *(e)*.

3.2.5. conclusion

Partant pour A de la matrice identité et pour B de la matrice nulle, qui forment un couple de matrices compatible, on conserve une couple de matrices compatible par application des transformations du type T_{pq} ou S_{mn}. De plus, la transformation S_{pq} est bien l'inverse de T_{pq}, et ceci même quand on ne l'applique pas immédiatement après la transformation T_{pq}, du fait de la commutativité des transformations sur des couples de matrices compatibles.

On peut ainsi dégager les deux propriétés:

- l'ensemble des couples de matrices compatibles est non vide et stable par application des transformations directe ou inverse;
- l'effet de toute transformation directe appliquée sur un couple de matrices compatible peut être "annulé" par application de la transformation inverse, soit immédiatement après, soit après d'autres transformations.

3.3. matrice des opérateurs

Notations

On note:

$C = [c_i^\lambda]$ la matrice des degrés d'appartenance

$D = [d_i^\lambda]$ la matrice d'appartenance en propre

l'indice i est le numéro d'un phylum,

l'indice λ est le numéro d'un opérateur.

c_i^λ se lit: «l'opérateur λ appartient au phylum i avec un poids c_i^λ »

d_i^λ se lit: «l'opérateur λ appartient en propre au phylum i avec un poids d_i^λ »

On notera qu'il ne s'agit plus ici de matrices carrées.

3.3.1. matrices compatibles

C est compatible si:

$$\sum_\kappa c_k^\lambda\ b_{k,i} = c_i^\lambda - d_i^\lambda$$

qui signifie:
le nombre de fois que λ appartient à i est égal à la somme sur k du nombre de fois que λ appartient à k, pondéré par le nombre de fois que k est directement fils de i, somme à laquelle on ajoute le nombre de fois que λ appartient en propre à i.

3.3.2. transformations

On définit:
- la transformation directe

$$M_p^* : \gamma_i^\lambda = c_i^\lambda + \delta_{\varphi\lambda}\, a_{pi}$$
$$\zeta_i^\lambda = d_i^\lambda + \delta_{\varphi\lambda}\, \delta_{pi}$$

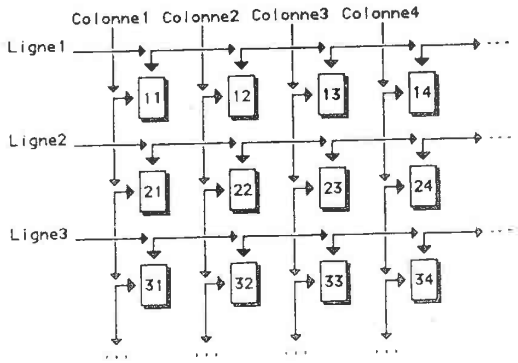
- la transformation inverse

$$N_p^* : \gamma_i^\lambda = c_i^\lambda - \delta_{\varphi\lambda}\, a_{pi}$$
$$\zeta_i^\lambda = d_i^\lambda - \delta_{\varphi\lambda}\, \delta_{pi}$$

Comme il s'agit de simples translations, on vérifie immédiatement que:
- les transformations préservent la compatibilité;
- elles commutent entre elles.

3.4. implantation des algorithmes

Les matrices précédentes sont implantées en Lisp.
Pour permettre le parcours d'une matrice selon les lignes ou les colonnes, chaque élément de la matrice est doublement chaîné, une fois avec l'élément voisin selon les lignes et une fois avec l'élément voisin selon les colonnes:



Ajouter un nouveau phylum ou un nouvel opérateur consiste à créer de nouveaux éléments de matrice, chaînés sur les listes existantes.
Supprimer un phylum ou un opérateur consiste à dérouter les pointeurs des listes sur les éléments des matrices, des éléments précédant ceux à supprimer vers les éléments suivant ceux-ci.
Modifier un phylum ou un opérateur consiste à parcourir, partiellement ou en totalité, les matrices de phyla et d'opérateurs.

Calcul du coût

On considère **N** phyla et **P** opérateurs définis.

putphyl = création d'un phylum: on crée 2N+1 éléments dans la matrice des phyla, et P éléments dans la matrice des opérateurs, soit en tout **2N+P+1** créations.
putoper = création d'un opérateur: on crée **N** éléments dans la matrice des opérateurs.
putoperphyl = déclaration d'appartenance en propre d'un opérateur à un phylum: on modifie le degré d'appartenance des pères du phylum, on réalise donc un parcours des **N** éléments d'une ligne de la matrice des phyla.
putperephyl = déclaration d'un lien de filiation entre deux phyla: on modifie le degré d'appartenance de tous les opérateurs des pères du phylum père, soit PxN éléments visités; pour chaque fils du phylum fils, et chaque père du phylum père, on modifie le degré de connexité de la matrice des phyla, soit P^2 éléments visités; en tout on a donc **PxN+P²** éléments visités.
putfilsphyl = on réalise l'appel symétrique de **putperephyl**.

Les fonctions de suppressions, **rem...**, sont de façon naturelle d'un coût similaire.

3.5. analyse des attributs

L'information construite à l'aide des matrices de phyla et d'opérateurs est suffisante pour décider si un opérateur donné appartient en propre, appartient indirectement ou n'appartient pas à un phylum donné. Cependant toute évaluation d'attribut nécessitant une recherche dans les matrices, il serait très coûteux, à l'exécution, de ne conserver que cette information.

Aussi, après la construction des matrices réalise-t-on une analyse des attributs, qui construit pour chaque attribut et chaque phylum la fermeture transitive de la relation «l'opérateur appartient au phylum». Ce calcul est conservé dans la P-list de variables auxiliaires.

Par exemple, pour le phylum TRM, on construit la P-list des symboles:
- evalt-TRM : évaluation
- recht-TRM : recherche
- cpltt-TRM : complétion
- imptt-TRM : impression
- entre-TRM : entrée
- sorte-TRM : sortie
- evale-TRM : impression évaluée

La P-list de imptt-TRM est, dans la *syntaxe initiale*:

```
( nulle   imptt-err atome   imptt-atome liste   imptt-liste
  env     imptt-env rep     imptt-err   lsp     imptt-lsp
  def     imptc-def ref     imptc-ref
  stg     imptt-err use     imptt-err   )
```

- nulle : c'est l'attribut des opérateurs qui n'appartiennent pas au phylum TRM – ici rep, stg et use.
- atome, liste : c'est l'attribut des objets Lisp qui ne représentent aucun opérateur.
- env, lsp : l'attribut est imptt-, soit l'impression complète, puisque ces opérateurs n'appartiennent pas directement au phylum TRM.
- def, ref : l'attribut est imptc-, soit l'impression concise, puisque ces opérateurs appartiennent en propre au phylum TRM.

La fonction Lisp **get** permet alors d'accéder immédiatement au traitement à effectuer, à l'occurrence d'un opérateur dans un phylum donné.

```
(get 'imptt-TRM 'env)
     traitement = fonction imptt-env
(get 'imptt-TRM 'rep)
     traitement = fonction imptt-err
(get 'imptt-TRM 'def)
     traitement = fonction imptc-def
```

Remarques

1 Les attributs de phyla permettent de construire les trois premières «propriétés» des P-listes: nulle, atome, liste.
Les attributs d'opérateurs permettent de construire, dans chaque phylum, l'attribut attaché à chaque opérateur défini. L'analyse étant réalisée *après* la modification des matrices de phyla et d'opérateurs, l'information: «l'opérateur appartient nouvellement au phylum» ou «l'opérateur appartenait anciennement au phylum» est perdue. Or c'est cette information qui détermine s'il faut ou non modifier la P-list. Aussi la conserve-t-on dans un "champ auxiliaire" de la matrice des opérateurs.

2 Les attributs d'opérateurs étant de deux sortes, on a deux traitements:
- ou bien il s'agit d'un atome Lisp, et l'attribut est cet atome,
- ou bien il s'agit d'une liste, qui doit alors être une paire: si l'opérateur appartient en propre au phylum, on prend pour attribut le second élément de la liste, et sinon le premier.

4. conclusion : une expérience vécue

En conclusion de cette partie, il me paraît important de soulever le point suivant: est-il bien utile de pouvoir définir une syntaxe abstraite par étapes, de pouvoir insérer de nouvelles définitions d'opérateurs ou de phyla, et à l'opposé de pouvoir en supprimer, et ceci d'une façon incrémentale?
Il est bien certain qu'une syntaxe abstraite n'est que le *support* de la représentation interne des objets qu'on manipule, et que la finalité n'est pas de travailler sur cette syntaxe abstraite mais plutôt de l'utiliser pour travailler sur des objets concrets. A ce stade du travail où l'on s'intéresse aux objets concrets, la syntaxe abstraite a logiquement atteint un état stationnaire, et ne devrait vraisemblablement plus être jamais modifiée.

Le seul intérêt qu'on peut trouver à permettre une construction incrémentale de la syntaxe abstraite intervient dans la phase de mise au point de celle-ci.

L'«expérience vécue» est la suivante: le programme Lisp décrit ici a été développé sur Vax, en LeLisp V15.2. On peut appeler Lisp par deux commandes: lelisp ou lelisp++. Les fonctionnalités sont semblables, mais lelisp++ réserve davantage de place mémoire.
Le problème auquel j'ai été confronté était celui-ci: en phase finale de construction du programme, c'est-à-dire à l'étape de mise au point, l'espace mémoire était saturé avec la commande lelisp. En revanche avec lelisp++ je n'ai pas eu de problèmes jusqu'à présent. Malheureusement, en certains cas, il y a trop d'utilisateurs sur le Vax, et lelisp++ ne peut être appelée.
La solution a consisté alors:
- à charger les fonctions de construction de la syntaxe abstraite,
- à charger la *syntaxe abstraite initiale*,
- à supprimer les fonctions de construction de la syntaxe abstraite,
- à charger enfin les autres fonctions utiles.

La suppression des fonctions de construction de la syntaxe abstraite fait que la syntaxe abstraite n'est plus construite d'une façon incrémentale, puisque le chargement est réalisé une fois pour toute, sans retour possible.

La conclusion de l'affaire est que la moindre modification de la *syntaxe initiale* m'obligeait alors à de subtiles manipulations de fichiers, d'où il ressort:
- que la gestion des fichiers sources devenait délicate,
- que le temps passé à contourner ces difficultés était du temps inutilement perdu,
- qu'en moyenne, sur une heure de travail, il y avait bien un quart d'heure exclusivement réservé à ces manipulations.

Ceci ne m'a bien sûr pas empêché de terminer le programme, mais je pense que les inconvénients mentionnés soulignent assez bien l'avantage qu'il y a à permettre une définition progressive de la syntaxe abstraite, et même si cela ne concerne que la phase de mise au point.

Chapitre 6:

Les Comparaisons avec d'autres Approches

On s'attache ici à présenter un certain nombre d'outils de programmation. On délaisse les aspects *techniques* de construction d'un programme exécutable (compilateur, éditeur de liens, chargeur, ...) pour s'intéresser plus spécifiquement aux outils qui se placent *en amont* de la génération de code (langage de commande, éditeur, metteur au point, ...). En fait, l'apparition des environnement intégrés de développement de projet tend à effacer progressivement la frontière entre ces deux traits de la programmation, au profit d'une gestion uniforme des documents qui prennent part à la réalisation d'un projet informatique.

1. Les critères

On a retenu deux catégories de critères pour le classement des outils présentés, qui portent sur les concepts sur lesquels travaillent ces outils: la *nature* et la *structure*.

- **La nature**
  - Objet: chaque concept qui intervient est entièrement défini par lui-même;
  - Type: on définit des modèles, et on travaille sur des instances de ces modèles;
  - Classe: on a une perception uniforme des Objets et des Types, qu'on utilise indifféremment – modulo les restrictions d'usage: le Type est un concept abstrait, l'Objet concret, on peut donc "demander" la valeur d'un Objet, mais pas celle d'un Type.

- **La structure**
  - Plat: on n'a pas de structure des concepts manipulés;
  - Arbre: la structure est celle d'un arbre, éventuellement d'un graphe orienté (acyclique); la vue structurée est donc hiérarchique;
  - Graphe: les concepts sont placés dans un graphe (cyclique).

Remarques:

1 Les outils présentés satisfont rarement d'une manière absolue à un critère donné; on les a en fait placés en regard de celui qu'ils satisfont le mieux.

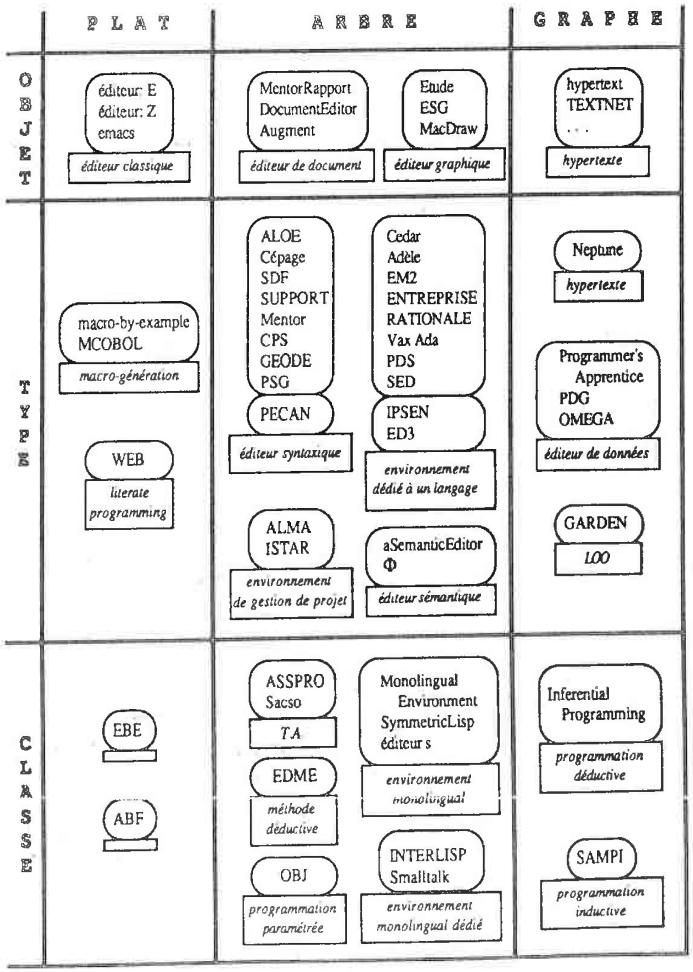
2 L'ordre dans lequel sont définis les critères est celui de l'accroissement de la *complexité* des outils. Je ne pense pas qu'il faille nécessairement le voir comme l'ordre de l'accroissement de la *qualité* des outils; par exemple, l'éditeur *emacs* est "tout en haut à gauche" – il travaille sur des Objets-Plats – mais c'est un éditeur de texte extrêmement puissant et très largement diffusé.

3 Il y a deux autres critères dont il faut tenir compte:
- le *degré sémantique* de l'outil, qui mesure la part d'automatisme et de contrôle assurée par l'outil;
- le *niveau de développement* de l'outil, sachant que certains sont déjà du domaine industriel alors que d'autres ne sont que des maquettes démonstratives, voire des idées conceptuelles.

On indique dans les commentaires comment les outils satisfont ces deux derniers critères.

4 On trouvera dans [MvD 82a et b] une présentation générale du domaine de l'édition, un historique détaillé ainsi qu'un aperçu des orientations actuelles dans les travaux de recherche. Bon nombre des références citées ont été le point de départ de mes propres recherches bibliographiques sur le sujet.

2. Le classement



3. La structure Plate

3.1. La nature d'Objet : les éditeurs classiques

Les éditeurs classiques travaillent sur des Objets-Plats:

- **Plat**: on travaille sur un flot de caractères. L'éditeur généralement reconnaît les mots, phrases, lignes, paragraphes, ..., mais il ne s'agit que d'une reconnaissance lexicale: un mot est une suite de caractères comprise entre deux «caractères de rupture» (blanc ou retour à la ligne), une phrase entre deux points, etc.... La frappe d'un caractère peut totalement bouleverser la "structure", sans autre forme de procès.
- **Objet**: une session sous l'éditeur s'intéresse à un certain *fichier de texte*, indépendamment de son contenu ou de ses relations.

On en cite trois, parmi beaucoup d'autres: l'éditeur *E* [GaF 84], l'éditeur *Z* [Woo 81], *emacs* [Sta 81] (vanté pour son caractère *extensible*). Les défenseurs de l'édition classique adoptent de façon naturelle une position critique vis-à-vis des éditeurs structurés: dans [Woo 81], il est estimé que 95% du travail réalisé en phase d'édition est supporté par un éditeur classique; pour les 5% restants, l'emploi d'un éditeur structuré entraîne une complication notable de l'implantation, non compté le temps CPU consommé. Dans [GaF 84], il est jugé qu'«éditer un programme est le même acte physique qu'éditer du texte, et [que] l'on devra développer le moins d'effort mental quand les outils seront semblables, à la fois pour l'édition des programmes et des textes.»

3.2. La nature de Type : la macro-génération

On a déjà abordé le thème de la macro-génération précédemment (cf. Chapitre 2.5, *«Exemple de structurations connexes»*). On la considère comme permettant de travailler sur des Types-Plats:

- **Type**: une macro-définition, éventuellement paramétrée, permet de définir un modèle de suite de caractères.
- **Plat**: une macro-expansion remplace l'invocation d'une macro-définition par du «texte en ligne», avec perte de la connaissance de la relation de dépendance entre la définition et l'utilisation de la macro.

Les exemples ont déjà été commentés: macro-by-example (pour le langage Lisp, [KoW 87]), MCOBOL (pour le langage COBOL, [TrY 80]) (cf. ibid.).

D'un tout autre point de vue, on peut citer [Knu 84], et la «Literate Programming» – la *programmation littéraire*. L'idée centrale, qui n'est pas nouvelle mais qui est abordée avec un regard nouveau, est de dire qu'un programme doit être lu par deux "individus": (1) la Machine, avec son goût prononcé pour la précision et le souci du détail, et (2) l'Informaticien, qui préfère sans doute avoir une vision plus synthétique et explicative du problème exprimé. D. Knuth propose donc un langage, WEB, pour satisfaire les deux parties: partant d'un unique fichier, une première compilation fournit un texte compréhensible par la Machine, une seconde par l'Informaticien.

3.3. La nature de Classe

On propose deux exemples qui satisfont aux critères de Classes-Plates:

- **Classe**: on a un usage dynamique des modèles.
- **Plat**: on reste dans un domaine non stucturé.

Le système EBE («Editing by Example», [Nix 85]) permet de définir dynamiquement, par la donnée de quelques exemples, des schémas de transformation de texte; par exemple, ayant la connaissance des transformations:

```
@i[O.K.] → {\s1 O.K.}
@i[Boston Morning Post] → {\s1 Boston Morning Post}
```

le système déduit, par unification, la règle de transformation:

```
@i[-1-] → {\s1 -1-}
```

(-1- est le paramètre de la macro-définition). On définit donc un dictionnaire de symboles, plus riche que ceux qu'on trouve ailleurs qui sont généralement des macro-définitions sans paramètres. On peut de plus traiter un texte existant simplement en indiquant quelques exemples de transformation pris au sein du texte.

Un autre exemple est le langage ABF [HES 87]: le système aide à la construction de documents juridiques, où l'aspect répétitif et systématique est particulièrement important. L'originalité de l'approche réside dans la simplicité de la représentation des "modules" – en l'occurrence du texte en anglais. L'utilisateur, qu'on suppose ne pas être informaticien, accède directement à la Représentation Interne, ce qui offre une grande souplesse dans la définition des paramètres des "modules". De plus, la forme interprétée des "modules" est rendue dans le même format que le format d'entrée, ce qui permet d'élaborer progressivement le texte final par des évaluations successives des "modules".
ABF a davantage une nature de Classe que de Type parce que l'utilisateur accède directement à la Représentation Interne – l'accès est filtré par le système qui transforme l'appel d'une macro-définition en une question intelligible par un non-informaticien – et que la forme évaluée (macro-expansée) peut à nouveau être traitée par ABF.

4. La nature d'Objet structuré

4.1. La structure d'Arbre : les éditeurs de documents

Les éditeurs (ou plus largement les environnements) de documents travaillent sur des Arbres d'Objets:

- **Arbre**: un document en général, plus particulièrement un document scientifique ou technique, est structuré pour en faciliter la lecture: ce sont les chapitres, sections, sous-sections, ...
- **Objet**: le système peut proposer certains modèles, comme par exemple «un chapitre est composé de sections»; mais l'objet d'intérêt principal est le texte attaché aux nœuds de l'arbre, qui lui a une nature d'Objet: il est unique dans sa définition.

Les outils cités ont une composante de Graphe, par la possibilité de définition d'un dictionnaire, la gestion des références croisées, la génération automatique d'une table des matières. Mais ce caractère de Graphe est très local; il ne concerne que certains aspects précis des documents. On a de fait classé ces outils dans la rubrique des Arbres.

MentorRapport [Mel 83] ou TheDocumentEditor [Wal 81] sont des outils qui aident à la rédaction de documents techniques structurés. Augment [EnE 68] [Eng 78] plus qu'un éditeur est un environnement qui propose de «développer l'intelligence humaine» – en augmenter la faculté de compréhension mais également le goût. L'environnement cherche à intégrer des supports de communication les plus variés: écran *bit-map*, souris (qui tire ses origines de ce projet), conférence électronique, ...

4.2. La structure d'Arbre : les éditeurs graphiques

Les éditeurs graphiques entrent dans la même catégorie d'outils travaillant sur des Arbres d'Objets. On cite Etude [HIA 81] qui présente un traitement de texte gérant plusieurs fontes; un outil qui "n'étonne plus personne", mais l'article n'est pas si ancien (Macintosh date de 1984). MacDraw, qui est support du schéma de classement, se présente par lui-même – mais pas MacPaint, qu'il faudrait ranger dans la classe des Objets-Plats. ESG («Editeur Syntaxique Structuré», [CCC 87]) est un prototype d'éditeur graphique structuré. Les deux derniers exemples pourraient presque se ranger dans la catégorie des Arbres de Types; on ne les y a pas mis parce que, s'ils offrent effectivement des modèles instanciables en revanche l'agencement des instances incombe à l'utilisateur: il n'y a pas de *règles de syntaxe* clairement définies.

4.3. La structure de Graphe : les hypertextes

La notion d'hypertexte remonte à 1967, avec T.H. Nelson [Nel 67] – qui lui-même se réclame de V. Bush, et le "memex", qui date de 1945. Il s'agit d'un texte structuré qui, par définition, ne peut être imprimé sous une forme satisfaisante; les relations qui lient les divers éléments du document sont en effet trop étroites pour qu'une forme imprimée puisse réellement les refléter. Dans [YMD 85] on étend en fait l'hypertexte à l'utilisation de supports d'information plus riches que la seule représentation textuelle: on y inclut aussi le graphisme, qui doit être intimement lié au texte, comme il apparaît par exemple dans la présentation de l'hypertexte Intermedia donnée dans l'article.

Le but d'un tel outil est double:

- Favoriser la lecture d'un document *par analogie*: un point du document qui nécessite quelque approfondissement est "physiquement" attaché à une autre partie du document où ce point est développé; l'utilisateur peut alors lui-même modeler l'organisation du document qu'il consulte selon la précision de l'information qu'il souhaite en retirer. Une telle approche répond donc très simplement à la délicate question des notes, placées en bas de page ou en fin de document: les notes sont ici des références à d'autres textes auxquels le lecteur peut immédiatement accéder à son initiative.
  Un exemple d'une riche application de la notion d'hypertexte peut être pris dans [WeB 85]: il y est présenté une «Encyclopédie électronique» qui mêle le texte au graphisme (et à une note de fantaisie) et qui relie finement certaines notions connexes – soit des références à d'autres articles, soit des notions plus vagues, comme la conversion d'une distance exprimée en miles par celle exprimée en kilomètres.

- *Impliquer* le lecteur, lors de sa lecture, en le faisant participer à la construction du document. Par exemple dans [Cat 79], on utilise un hypertexte pour enseigner la critique littéraire: l'auteur constate une meilleure participation des étudiants au "cours", du fait des dialogues possibles entre les élèves et le professeur ou même entre les élèves. Dans [Wey 82] il s'agit d'une expérience menée avec des élèves du secondaire: il leur est posé des questions assez précises, et l'on compare la facilité avec laquelle ils en trouve la réponse, soit par la consultation d'un "livre-papier", soit par l'utilisation d'un hypertexte; l'expérience montre que la seconde situation donne de meilleurs résultats, quoiqu'il faille se garder d'une généralisation hâtive du fait de la taille réduite de l'expérience menée.

On cite l'hypertexte TEXTNET [TrW 86], parmi beaucoup d'autres. Dans l'article mis en référence, les auteurs relèvent en particulier les caractéristiques des hypertextes:
- hiérarchie des textes – structure de Graphe;
- cheminement; c'est un des problèmes de l'hypertexte: l'utilisateur ne sait pas toujours très bien "où il se trouve", du fait même de la structure de graphe du document;
- support distribué;
- gestion des versions et des accès concurrents; c'est un autre problème de l'hypertexte: l'utilisateur étant *impliqué* dans la rédaction du document, quel contrôle assurer pour éviter les incohérences issues d'une utilisation partagée?

5. La nature de Type structuré

5.1. La structure d'Arbre : les éditeurs syntaxiques

On entend ici par éditeurs syntaxiques des éditeurs syntaxiques paramétrés par la grammaire du langage. On les regarde comme des outils travaillant sur des Arbres de Types:

- **Arbre**: clairement il s'agit des arbres syntaxiques (abstraits).
- **Type**: les modèles d'arbres sont les opérateurs du langage.

Un éditeur paramétré par la grammaire permet à l'utilisateur de définir ses propres types. Mais comme il a été dit auparavant (cf. Chapitre 2.1, *«Présentation de la Syntaxe Concrète»*), on utilise, en phase d'édition, une instance de l'éditeur syntaxique paramétré: l'utilisateur ne peut pas, dynamiquement, modifier les types, c'est-à-dire les opérateurs du langage, ni en créer ou en supprimer.

On en cite un certain nombre, et il y en a beaucoup d'autres:

◊ ALOE [MeN 81] [FeM 80], un élément du projet GANDALF – qui s'intéresse à l'*environnement*, dont ALOE est l'éditeur.
◊ Cépage [MeN 87], déjà cité à propos du choix d'élision adopté (cf. Chapitre 4.3, *«Compléter la syntaxe»*).
◊ SDF [HeK 86] [Hen 87], du projet GIPE.
◊ SUPPORT [Zel 84].
◊ Mentor [KLM 86] [DHK 80] [MMV 85], un des pionners en la matière, avec deux extensions:
  – le projet Concerto: par exemple le VTP («Virtual Tree Processor», [Lan 86], le *noyau* de Mentor),
  – le projet Centaur: par exemple la sémantique naturelle – opérationnelle – avec le langage Typol [CDD 86] [Kah 87].
◊ CPS («Cornell Program Synthesizer», [ReT 85] [TRH 81]), qui développe un *environnement* par l'utilisation de grammaires attribuées ([HoT 86] pour une vue plus formelle sur le sujet); un projet voisin est actuellement en cours de développement au CRIN, à Nancy, GEODE [CrD 87].
◊ PSG [BaS 86], un des derniers venus, il bénéficie des efforts antérieurs:
  – l'éditeur est *hybride*, c'est-à-dire qu'il fonctionne structurellement (par l'affinage des structures syntaxiques, les "templates") ou textuellement (comme un éditeur classique);
  – les contrôles sémantiques sont finement réalisés (par exemple, l'outil peut réaliser un contrôle de type structurel sur des types non déclarés);
  – une édition sensible au contexte («context-sensitive»): les options qu'on peut sélectionner tiennent compte d'une information de contexte, par exemple le type des expressions.

On place un peu à part l'éditeur PECAN [Rei 84]: c'est aussi un éditeur syntaxique, mais qui permet par rapport aux autres un affichage simultané selon plusieurs schémas de décompilation de l'arbre syntaxique – dont un schéma qui affiche les déclarations de variables et un autre qui affiche le graphe de contrôle du programme.

5.2. La structure d'Arbre : les environnements dédiés à un langage

Parce qu'il est plus simple et parce qu'il permet de définir des outils plus puissants, le choix d'un langage donné a conduit au développement d'un grand nombre d'environnements dédiés.
On en cite quelques-uns:

◊ Cedar: Cedar [SZB 86], un environnement entièrement intégré; au dire de S.P. Reiss [Rei 84] il a fortement influé sur la définition de PECAN.
◊ Adèle: Pascal [MRR 83].
◊ EM2: Modula-2 [FGN 83].
◊ ENTREPRISE: LTR3 (cf. Chapitre 3.4, *«étude de cas: le langage LTR3 et l'atelier ENTREPRISE»*).
◊ RATIONALE: Ada, qui travaille exclusivement sur les arbres syntaxiques Ada.
◊ VAX Ada: Ada [Mit 87], où la possibilité de mêler plusieurs langages dans une même application a été particulièrement soignée.
◊ PDS: EL1 [Che 84], un VHLL.
◊ SED: Seti [DDF 87], un autre VHLL.

Un peu à part on en présente deux autres:

◊ IPSEN: Modula-2 [ENS 87], où le choix de la Représentation Interne est *le graphe*: on réalise alors facilement le contrôle de type sur les variables (qui sont liées dans le graphe à leur déclaration), la décompilation selon divers schémas; l'avantage retiré est surtout la plus grande simplicité de l'implantation qu'avec l'approche courante – qui calcule des attributs sur les arbres abstraits, ce qui pose de gros problèmes algorithmiques de modification incrémentale des attributs (cf. CPS par exemple).

◊ ED3: Pascal [Str 86], qui est un éditeur structuré appliqué au langage Pascal: l'utilisateur manipule du texte, dont il peut faire vérifier la correction syntaxique, qu'il peut faire indenter (ou non), et qu'il peut structurer sous forme d'arbre, avec une entière liberté dans sa structuration; l'auteur propose une structuration par les procédures, qui autorise une vue synthétique ou affinée du programme, pourvu qu'on s'applique à rédiger son programme en le structurant par les procédures.

5.3. La structure d'Arbre : les environnements de gestion de projet

On présente deux tels environnements, qui cherchent à offrir un cadre de développement où la cohérence soit automatiquement assurée entre les nombreux documents qui participent à la conduite d'un projet.

L'environnement ALMA [Lam 87]: l'outil propose un environnement générique, paramétré par la méthode de conduite de projet; l'utilisateur rédige le "programme" qui construit les relations de dépendances de la Base de Données du Projet; il travaille ensuite dans l'environnement instancié, où les documents du projet sont traités comme des *annotations* sur les arbres de la méthode – des nœuds auxquels on attache d'autres arbres, définis dans d'autres formalismes: par exemple une spécification formelle, un programme source, ...

L'environnement ISTAR [Dow 87]: le même objectif est visé; comme dans ALMA il propose une interface unique à tous les stades de la conduite de projet, qui est un éditeur "convivial". ISTAR adopte un «approche contractuelle»: une tâche du

développement est un «contrat» passé entre le "client" et le développeur, ce qui permet de décomposer le projet en autant de tâches relativement indépendantes – la seule contrainte porte sur les données qui transitent entre les contractants, au démarrage ou à l'issue d'un «contrat» donné. Le système offre "en standard" de nombreux outils et propose un kit pour l'intégration de nouveaux autres.

5.4. La structure d'Arbre : les éditeurs sémantiques

On place ici deux éditeurs sémantiques qui entrent encore dans la catégorie des outils travaillant sur les Arbres de Types.

Le premier, *a Semantic Editor* [DyS 85], a déjà été présenté (cf. Chapitre 2.4, *«Exemple de structuration des traitements»*); il travaille sur les arbres syntaxiques du langage FP, en maintenant la relation d'équivalence entre les programmes sémantiquement équivalents – on travaille en définitive sur des graphes orientés.

Le second, Φ (pour PHENARETE, [Wer 85]), est un outil de correction des programmes Lisp. Il détecte des erreurs d'ordre lexical ("CRA" au lieu de "'CAR"), syntaxique (le parenthésage Lisp), sémantique (l'absence d'un test de terminaison d'une fonction récursive), "esthétique". Le programme fonctionne par l'activation de *démons*, des traitements qui "démarrent" tout seuls quand l'état du système répond à certains critères donnés:
- les démons dirigés par les données: ce sont les «spécialistes» de Φ, qui représentent des schémas-types d'erreurs communément faites;
- les démons dirigés par les concepts: ce sont les «règles pragmatiques», qui travaillent avec la prise en compte d'une certaine information de contexte.

On l'a rangé dans la classe des Arbres de Types parce qu'il présente des Types définis indépendamment de la session sous l'éditeur (les démons) et qu'il travaille sur les Arbres syntaxiques de Lisp – quoique pour ce dernier point, l'outil s'intéressant plus à la sémantique qu'à la syntaxe des programmes Lisp, il ne soit pas très éloigné d'un outil travaillant sur une structure de Graphe…

5.5. La structure de Graphe : les hypertextes

L'outil présenté, Neptune [DeS 86], est un hypertexte: son principal domaine d'application est d'abord la documentation. Dans l'article mis en référence les auteurs s'intéressent à son emploi dans le domaine du Génie Logiciel: si l'outil, de par sa structure, permet de facilement lier plusieurs documents – textes source, commentaires, spécification, … – en revanche la structuration des textes sources est plus difficilement supportée:
- le nœud est l'unité atomique de Neptune: représenter l'arbre syntaxique d'un programme nécessite alors la définition de nombreux nœuds de l'hypertexte, ce qui rend leur gestion assez malaisée;
- les liens entre nœuds sont statiques: l'outil sera d'un emploi délicat pour la gestion de versions multiples.

En fait, les auteurs s'interrogent sur le *degré de précision* à adopter: faut-il descendre jusqu'aux composants syntaxiques élémentaires ou rester à un niveau plus général? Ils pensent qu'une bonne utilisation de l'hypertexte serait dans la gestion des relations entre des «pièces maîtresses» du projet, mais qu'à un niveau plus fin il faudrait le connecter à une Base de Données relationnelle.

5.6. La structure de Graphe : les éditeurs de données

On présente ici d'autres éditeurs, qui travaillent plutôt sur des Graphes de Types:
- **Type**: les modèles sont statiquement définis.
- **Graphes**: les instances sont gérées dans une structure de graphe.

The Programmer's Apprentice [Wat 82] [Wat 86] a déjà été présenté (cf. Chapitre 2.4, *«Exemple de structuration des traitements»*); il utilise une bibliothèque de programmes prédéfinis et un graphe de flot de données sur les instances des éléments de la bibliothèque. PDG [OtO 84] propose aussi une Représentation Interne par les flots de données. L'approche semble en effet prometteuse, parce qu'elle fait totalement abstraction du *flot de contrôle* qu'on retrouve classiquement dans un langage de programmation «à la Von Neumann» pour s'attacher aux relations de dépendances de la pertinence des variables du programme – cette aide pourrait être particulièrement utile dans la phase de mise au point.

Le système OMEGA [Lin 84] propose un point de vue nouveau pour la gestion des liens sémantiques qu'on trouve à l'intérieur d'un programme: il s'agit de représenter le programme dans une Base de Données (B.D.) – Ingres; "voir" une variable c'est alors demander la valeur de l'attribut «nom» de l'objet «variable» de la B.D.; contrôler son type c'est s'intéresser à l'attribut «type» de ce même objet.

5.7. La structure de Graphe : une approche LOO (Langage Orienté Objet)

L'environnement GARDEN [Rei 86] [Rei 87] est un peu à la croisée des chemins suivis par un système à flot de données et un système à base de données: le concept manipulé ici est l'*objet*, instance d'un type qu'on a défini dans la syntaxe du langage. L'outil permet de facilement définir de nouveaux formalismes, soit textuels, soit graphiques, avec un minimum d'effort de conceptualisation:
- dans un premier temps on définit les *types* qui décriront les objets du nouveau langage; on peut mettre à profit à ce niveau la richesse d'expression d'un LOO, en particulier par l'emploi de l'héritage de types déjà définis;
- ensuite on définit la *syntaxe* du langage, qui peut être saisie "textuellement" ou avec un "éditeur de ressources" pour les langages graphiques;
- enfin on définit la *sémantique* du langage par la donnée d'une fonction d'évaluation des objets du langage.

On est très près dans GARDEN de l'outil qui travaille sur des Graphes de Classes, d'autant que dans l'implantation présentée dans l'article le langage de description des objets est le langage Lisp, interprété, ce qui rend aisé la définition dynamique d'un nouveau type du langage. Cependant l'approche semble toujours privilégier la phase d'*utilisation* du langage, et non celle de sa *définition*: on définit facilement, par des outils interactifs, un nouveau langage, mais l'objet d'intérêt est le «monde des Objets» et non celui des Types, «monde» dans lequel on a défini le langage. En pratique l'utilisateur est placé dans deux environnements de travail bien différenciés, selon qu'il est en train de définir un langage ou de l'utiliser.

6. La nature de Classe structurée

6.1. La structure d'Arbre : T.A. (Type Abstrait)

Une méthodologie fondée sur les T.A. s'intéresse à des Arbres de Classes:

- **Arbre**: c'est la hiérarchie des Types, grâce à laquelle on définit un T.A. à partir d'autres.
- **Classe**: puisqu'on travaille sur des Types – et non des Objets: on définit des *modèles* de traitements et non un certain traitement – et qu'on construit dynamiquement ces Types, on se ramène en définitive à travailler sur des Classes; une Classe terminale correspond à un T.A. prédéfini; une Classe quelconque est construite à partir d'autres T.A.

L'environnement ASSPRO [Bid 87] travaille sur des spécifications dans le formalisme des T.A. Algébriques. Il offre de plus des outils pour un traduction assistée en langage Ada:
– génération automatique d'un squelette de programme Ada;
– assistance dans l'implantation des fonctions Ada (Caty, [BGG 83]): l'utilisateur applique des *stratégies* d'affinage que l'environnement traduit par du texte source.

L'environnement Sacso [LPS 87] travaille aussi sur les T.A. L'objectif visé se situe ici en amont de l'environnement ASSPRO: il s'agit d'un outil d'aide à la construction des spécifications, où l'utilisateur devrait pouvoir appliquer ses propres méthodes d'analyse. Le but est de présenter un outil qui assure la validité des spécifications construites mais qui soit suffisamment souple pour laisser à l'utilisateur le choix de ses méthodes d'appréhension d'un problème.

6.2. La structure d'Arbre : la méthode déductive

On place aussi dans cette catégorie l'éditeur EDME [GuJ 87], initialement présenté dans [Jac 83], qui travaille selon la *méthode déductive* développée à Nancy; on procède à une décomposition progressive du résultat – ce que l'Informaticien a le plus de facilité à saisir au départ – jusqu'à ce que chaque calcul intermédiaire soit une donnée ou une constante:

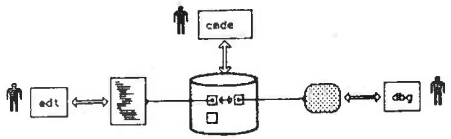
- **Arbre**: on applique une démarche descendante.
- **Classe**: chaque nœud est dynamiquement défini: il s'agit d'un certain calcul, symboliquement nommé dans le langage Médée, et qui sera affiné par la suite.

Pour les auteurs, langage, système et méthode doivent être définis simultanément. C'est le but d'EDME, qui cherche à proposer une simplicité:
- de communication: l'interface doit être conviviale;
- des contrôles: les contrôles sont réalisés *en phase* d'édition;
- du respect d'un discipline de programmation: ici par l'application de la méthode déductive;
- des concepts: on n'a toujours qu'un unique concept – alors que dans un éditeur syntaxique par exemple l'utilisateur doit s'intéresser à tout moment au texte, à l'arbre syntaxique et la structure algorithmique de son programme.

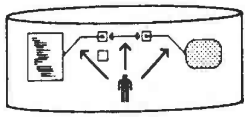
6.3. La structure d'Arbre : l'environnement monolingual

L'environnement monolingual cherche à offrir un cadre unique pour tous les aspects *techniques* de rédaction des programmes: saisie des textes sources, mise au point des programmes, gestion des "fichiers", ... De ce point de vue l'environnement monolingual veut aller plus loin que l'environnement intégré: dans ce dernier on s'attache surtout à garantir une cohérence entre des objets "assez gros", mais pour les détails *techniques* la tâche est confiée à d'autres outils – éditeur syntaxique, metteur au point symbolique, langage de commande, ...

Pour rompre avec l'habituelle interaction "à distance" de l'utilisateur sur la base d'objets:



il faut alors lui proposer un moyen d'expression *uniforme*, indépendant du type d'interaction qu'il souhaite avoir avec les objets de l'environnement: c'est l'idée maîtresse des environnements monolinguaux (ce petit néologisme doit se comprendre: «qui ne reconnaît qu'un seul langage»). Au lieu de conserver cette vue "superficielle" de l'utilisateur sur les objets qu'il manipule, on le *plonge* dans la base d'objets, dont il a alors une perception unique.



J. Heering et P. Klint présentent les principales propriétés d'un environnement monolingual [HeK 85]:

> langage de commande = langage de programmation = langage du metteur au point
> variable = fichier et répertoire = structure (ou "directory" = "record")
> procédure = programme = commande

Pour les deux auteurs, seuls satisfont aux conditions les environnements INTERLISP et Smalltalk; ils donnent une vision presque uniforme des trois principaux aspects de la programmation:
– la programmation proprement dite, en langage informatique,
– la gestion des unités discernables du langage,
– et la mise au point interactive des programmes.

En conclusion, les auteurs relèvent que la réalisation d'un tel environnement monolingual est un problème complexe, et non résolu.

Une possible solution est présentée dans [GJL 87b]: avec le langage *Symmetric Lisp*, on peut espérer construire un environnement de programmation qui garantirait:
- la gestion des flots infinis de données («stream»): en particulier le flot infini des commandes entrées par l'utilisateur;
- le parallélisme d'évaluation des commandes-expressions: c'est l'une des particularités du langage d'élaborer une évaluation parallèle de certains termes du langage (les environnements alpha) – d'où une possible implantation sur machine parallèle;
- la persistance des données: ce qu'on peut attendre d'un environnement de programmation, réalisée ici par les concepts que présente le langage.

En particulier dans un tel environnement on aurait la propriété de *«metacleanliness»*: on ne fait plus de distinction entre variable et fichier.

Un autre exemple, de petite dimension au dire même de l'auteur, est l'éditeur de texte *s* [Fra 80]. Cet éditeur permet, dans le seul langage de commande de l'éditeur:
- l'édition de texte classique,
- la gestion des fichiers – l'envoi des commandes au système UNIX étant transparent à l'utilisateur –,
- la simulation d'une exécution sur une machine à registres.

La «maquette» présentée ne prétend pas proposer un nouveau produit, mais cherche plutôt à être le support d'une reflexion sur le plus grand confort qu'offre une interface quand elle se présente de manière uniforme dans des situations diverses.

6.4. La structure d'Arbre : les environnements monolinguaux dédiés

Pour J. Heering et P. Klint [HeK 85], seuls satisfont aux critères de l'environnement monolingual les environnement INTERLISP (Lisp, [TeM 81]) et Smalltalk (Smalltalk-80, [Gol 83]). Dans les deux cas il s'agit d'un langage *interprété* – éventuellement semi-compilé – qui réalise la *liaison dynamique* des identificateurs. On y trouve donc assez naturellement la notion d'Arbres de Classes:
- **Classe**: les Objets-Types sont ici les fonctions (Lisp) ou les classes (Smalltalk): le caractère interprété des langages permet de facilement construire de nouveaux Objets-Types et de les inclure dans la bibliothèque – plus spécialement avec Smalltalk où la classe du langage s'identifie à la Classe du critère.
- **Arbre**: les "objets" définis (fonctions, classes) sont placés dans une hiérarchie; avec la récursivité on a une composante de Graphe, mais qui n'est apparente en fait que sur la forme évaluée des "objets", et non sur leur structure déclarative. De ce fait on reste placé au niveau des Arbres.

6.5. La structure d'Arbre : la programmation paramétrée

La «programmation paramétrée» est présentée par J.A. Goguen dans [Gog 84], illustrée par le langage OBJ [Gog 84], devenu OBJ2 [FGM 87]. Le langage travaille dans le formalisme des T.A.; la richesse des concepts introduits le fait placer plutôt sous la rubrique des «environnements monolinguaux» parce qu'il se suffit à lui-même pour un développement intégré de projet – quoiqu'on ne retrouve pas dans OBJ le mécanisme d'évaluation du langage exprimé dans les termes du langage, comme on l'a avec Lisp ou Smalltalk.

L'objectif est de favoriser au mieux la clarté, la validité et la réutilisabilité du logiciel. Pour ce faire, on doit disposer d'une grande facilité de paramétrisation des programmes, qui soit à la fois riche quant aux possibilités offertes et sévère quant aux contrôles de cohérence effectués.

Les traits de la programmation paramétrée:

- la modularité: le langage/outil sera modulaire, avec en corollaire la possibilité de masquage de l'information, une structuration hiérarchique de l'application, le développement de librairies de programme;
- le typage fort: ceci aide à la détection des incohérences; de plus il permet d'introduire la notion de *surcharge* des identificateurs;
- la paramétrisation: les modules doivent être faciles à paramétrer; trois autres points sont requis, qui sont généralement mal supportés par les langages:
  - des contraintes sur les paramètres d'instanciation,
  - la possibilité de modifier un module lors de son emploi,
  - la possibilité de déclarer – sinon de déduire – des propriétés qu'on vérifie dans le module (par exemple, la fonction "+" est associative);
- la simplicité, une sémantique formelle, un développement interactif du programme.

• • •

Au regard des caractéristiques de la «programmation paramétrée», je me reconnais d'une inspiration voisine dans mes propres travaux. Je situerai la différence à plusieurs niveaux:

- Dès qu'il est question de *contrôles* sur la cohérence, le respect de contraintes, le type des objets, l'outil proposé ne répond plus; en fait le seul contrôle assuré est celui de la visibilité des identificateurs.

- Je ne propose pas un développement selon une hiérarchie stricte – un graphe a-cyclique. Au contraire, l'outil, par la possibilité d'emploi par référence, offre le moyen de *contourner* la hiérarchie et la masquage des déclarations. Ceci s'oppose peut-être à une bonne méthodologie de programmation, mais doit s'entendre dans l'esprit des exemples qui ont été présentés auparavant.

- Je ne propose pas non plus un langage mais un outil qui travaille sur les langages, ou plutôt sur les textes sources des langages. Le champ d'application serait donc plus large – *un peu* plus large si l'on se réfère au langage LIL [Gog 86]: le langage sert à la gestion cohérente des paquetages Ada, dans l'esprit de la «programmation paramétrée»; ce qu'on propose c'est une gestion *plus* fine sur des langages *moins* complexes que le langage Ada.

- La *réutilisation* n'est pas un processus simple à élaborer: «du logiciel réutilisé c'est plus que du logiciel réutilisable» (cf. [Tra 88] sur les *neuf mythes* de la réutilisation); c'est toute une stratégie de développement tournée vers la réutilisation du logiciel. L'outil proposé veut répondre au problème en offrant à l'utilisateur le moyen de réutiliser du logiciel qui n'était pas réutilisable, c'est-à-dire qui n'a pas été conçu dans cette optique.

6.6. La structure de Graphe : la programmation "déductive"

La programmation déductive, présentée informellement et longuement dans [ScS 83], touche toujours au domaine de la transformation de programme; mais l'objet d'intérêt n'est pas ici *le produit* obtenu par transformation mais *le processus* par lequel il a été obtenu. La différence de l'approche est plus qu'une simple "vue de l'esprit"; en effet on conserve l'historique du programme et donc l'enchaînement logique des décisions de conception qui sont intervenues lors du développement: c'est donc à la fois à la relecture et la modification que l'utilisateur sera guidé dans sa compréhension du

programme. Pour les auteurs on peut même aller plus loin: la preuve du programme est *contenue* dans son schéma de dérivation; dans une perspective ambitieuse on part d'une spécification formelle validée, et on démontre la correction de chaque étape de dérivation; dans une perspective plus modeste, on part d'un "programme" simple, jugé correct, et on s'assure que chaque étape de dérivation préserve la sémantique du programme.

On classe la programmation déductive comme nécessitant un outil qui travaille sur des Graphes de Classes:

- **Graphe**: lier effectivement des concepts liés sémantiquement, cela nécessite de travailler sur des Graphes: les liens sémantiques dans un programme sont trop étroits pour espérer réaliser cette liaison sur une simple structure d'Arbre. La structure d'Arbre est le choix retenu dans une approche classique de décomposition par affinage; elle est insuffisante dans le cadre de la programmation déductive.
- **Classe**: on ne peut pas prétendre réaliser tous les modèles de programme avant de programmer, et réduire l'activité de la programmation à une réutilisation systématique de modèles prédéfinis; on a donc besoin, durant l'étape de programmation, de pouvoir définir de nouveaux modèles, puis de les instancier dans le contexte spécifique du développement en cours.

On notera que l'"outil" est ici une "expression de besoins": les auteurs ne présentent pas un outil opérationnel mais une recommandation qu'ils reconnaissent difficile à satisfaire.

6.7. La structure de Graphe : la programmation "inductive"

Dans la «programmation inductive», on cherche à induire, d'un premier programme déjà écrit, un schéma de programme plus général, et donc plus facilement réutilisable. La démarche s'identifie assez clairement à la réalisation d'un *prototype évolutif*: il s'agit de partir d'un programme traitant une petite partie du problème et de le faire évoluer par ajout de fonctionnalités, amélioration des performances, accroissement de la complexité des cas traités. Elle est à rapprocher de celle proposée par la «programmation paramétrée», parce que:

- on ne peut guère envisager une évolution du logiciel qui n'implique pas une modification du programme déjà écrit;
- la démarche qui consiste à résoudre progressivement le problème en l'étendant peut faire espérer, au moins dans les premiers états de l'avancement, la réutilisation de programmes développés auparavant.

Ces deux points entraînent qu'on doit disposer d'un outil qui permette une forte paramétrisation des programmes, parce qu'à l'instant où l'on écrit son programme on ne peut pas *deviner* quels seront les choix qu'on remettra en cause par la suite.

On range aussi la programmation inductive comme nécessitant un outil qui travaille sur des Graphes de Classes, pour les mêmes raisons qui on fait placer plus tôt la programmation déductive sous cette rubrique.

SAMPI (pour «Système d'Aide à la Maintenance des Programmes Intéractif») est l'outil que je propose; il a déjà été présenté (cf. Chapitres 1 à 4). Ici aussi le terme "outil" est un peu prématuré, puisque l'état actuel de mes travaux n'en permet guère une utilisation «en vraie grandeur».

Chapitre 7:

Les Perspectives

On présente l'état actuel des travaux et les perspectives de leur poursuite.

1. Où se situe-t-on?

La comparaison de l'outil proposé avec d'autres approches touchant à la construction des programmes fait placer celui-ci à l'intersection de deux tendances convergentes:

- l'**intégration**, où l'on aborde avec une vue uniforme tous les problèmes liés à la réalisation d'un projet informatique;
- la **cohésion**, où l'on lie intimement les notions qui sont apparues sémantiquement attachées, même quand les outils supports de ces notions contraignent à un éloignement "physique" de leur déclaration.

La premier point sous-tend le concept d'**environnement intégré**, prolongé par celui d'environnement monolingual: la gestion des objets de l'environnement est assurée en cohérence avec la démarche de développement de projet adoptée. La nature des outils s'intéressant à la question varie avec la précision de détail choisie:

- à un niveau assez élevé, on manipule les «pièces maîtresses» du projet: on trouve alors les environnements de gestion de projet ou des environnements dédiés à un langage;
- à un niveau plus fin, on s'attache à aider l'utilisateur dans la rédaction de programmes syntaxiquement corrects: ce sont les éditeurs syntaxiques;
- à un niveau plus fin encore, on s'intéresse à la méthodologie de résolution d'un problème: ce sont les éditeurs sémantiques, qui contrôlent la pertinence sémantique du programme écrit, ou des environnements attachés à une méthode de conception des programmes.

Le second point est généralement traité par des outils d'édition de documents structurés; c'est en effet dans ce cas-ci que la contrainte de **cohésion** est la plus forte. Au-delà de l'édition d'un document, on trouve *l'hypertexte*, l'outil d'expression du document: le moyen devient alors la fin, parce que l'hypertexte met à profit le support électronique de l'information pour présenter des fonctionnalités qu'on ne retrouve pas dans un "document-papier" – animation, participation active du lecteur, recherches par mises en correspondance («pattern-matching»), ... C'est aussi la direction suivie par les éditeurs à flot de données. Un exemple intéressant est l'environnement GARDEN, qui traite les modèles syntaxiques comme les types et les instances du programme comme les objets d'un Langage Orienté Objet défini par l'utilisateur. La cohésion du programme se traduit alors par le dialogue entre les objets.

Ma position est d'offrir un outil d'**intégration** qui garantisse la **cohésion** du programme, par une expression simple et uniforme et par la grande facilité de liaisons entre les objets de l'environnement.

2. Vers quoi tend-on?

Les exemples précédents d'utilisation de l'outil ont révélé deux choses:

- La première est que l'outil, en l'état, n'assure que faiblement des contrôles de cohérence dans l'emploi des «textes» définis; ceci est certainement une limitation sévère, à laquelle il faudrait chercher à remédier.

- La seconde est qu'on retrouve, comme dans un éditeur syntaxique classique, une dualité à l'utilisation de l'outil:
  - au niveau concepteur, on construit de nouvelles formes d'expression – les opérateurs et les phyla;

  - au niveau de l'utilisateur, on bénéficie des efforts fournis par le concepteur.
  Cela signifie que le concepteur devrait être aidé dans la définition de nouveaux termes du langage, par des outils d'assistance qui prennent en compte les aspects syntaxiques et, pour partie, sémantiques de ces termes.

Enfin, une utilisation «en vraie grandeur» de l'outil, où l'on pourrait confronter l'approche proposée à une réalité informatique de taille conséquente, serait sûrement souhaitable.

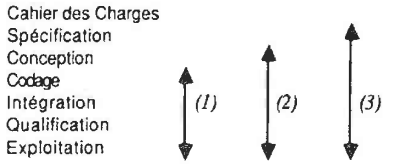
Les perspectives de poursuite des travaux

Une possible poursuite des travaux pourrait être alors de définir, après *l'éditeur à références concentrées*, deux autres éditeurs qui incluent davantage de sémantique dans leur définition:

l'éditeur à références déductives,
l'éditeur à références constructives.

Si la première étape, qui a cherché à définir *l'éditeur à références concentrées*, semblait impérative pour les deux suivantes, celles-ci en revanche paraissent assez indépendantes l'une de l'autre, voire même complémentaires.

Le schéma présente, sur les différentes étapes du cycle de vie, celles pour lesquelles chacun des trois éditeurs annoncés offrirait un cadre de développement facilitant leur conduite.



(1) L'éditeur à références concentrées
Dans un premier temps, on s'intéresse essentiellement à l'étape de Codage, et aux étapes qui la suivent: on propose un outil qui facilite la rédaction du programme, tant sur le plan de l'écriture que sur le plan de la relecture du programme.

(2) L'éditeur à références déductives
On souhaite ensuite attacher une sémantique aux regroupements: l'outil aide alors l'utilisateur dans ses choix de Conception du programme en assurant des contrôles de validité ou suggérant des transformations.

(3) L'éditeur à références constructives
On veut enfin permettre à l'utilisateur de voir son programme selon l'image que lui-même s'en fait: on l'aide alors dans la phase de Spécification, puisqu'il peut ne pas s'éloigner à la rédaction du programme de l'expression du problème dans les termes de sa spécification.

3. L'éditeur à références déductives

Complément de *l'éditeur à références concentrées*, il permet de déduire, à partir de prémisses, des conditions à remplir, soit impérativement, soit sous forme de recommandations.

Par exemple, l'utilisation d'une variable x suggère la déclaration de cette variable dans une zone de visibilité. Plus simplement peut-être, partant des déclarations:

```
TYPE pile :  RECORD
                contenu : ARRAY[1..100] OF real;
                index   : integer;
             END RECORD;
```

on déduit que l'utilisation d'un objet:

P...

sera:

P	: de type pile,
P.contenu	: du type anonyme «tableau de 100 réels»,
P.contenu[ ]	: de type réel, la case à remplir étant de type entier.

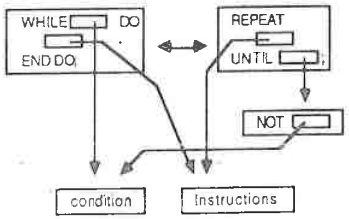
Ainsi la frappe de "P" fait suggérer par l'éditeur la suite du texte:

<rien>
.contenu
.contenu[ ]

dont il peut vérifier ensuite si elle correspond bien au type attendu.

On définit alors un éditeur sémantique, qui vérifie la cohérence entre les objets utilisés et ceux qui sont déclarés dans la zone de visibilité du moment.

A un autre niveau l'éditeur peut suggérer des schémas de transformations à moindre coût. Par exemple:



(les références à «condition» et «Instructions» sont conservées lors de la transformation). On notera que dans ce schéma on ne garantit pas la préservation de la sémantique du programme: l'éditeur *suggère* une transformation dans un «voisinage sémantique» mais il laisse à l'utilisateur le soin d'en apprécier les conséquences.

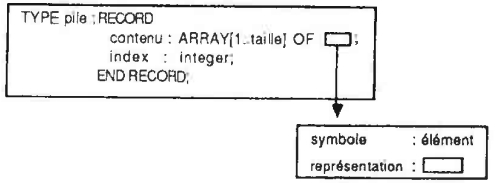
On donne en annexe un exemple de déductions que supporterait l'éditeur.

4. L'éditeur à références constructives

C'est la démarche inverse de celle présentée auparavant d'expansion des termes par leur définition complète:

Un concept du langage auquel on souhaite donner un nom symbolique est représenté textuellement (dans le texte lu sous l'éditeur) par ce nom symbolique. La technique s'apparente très nettement au procédé de macro-expansion qu'on trouve à côté d'un certain nombre de langages. Sa définition plus générale en permettrait aussi un emploi plus riche.

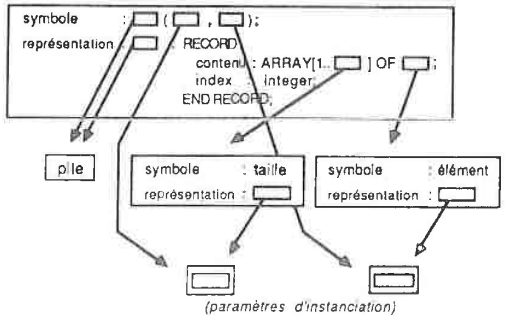
Par exemple, la pile générique est définie par:



Dans le texte de définition de la pile, il apparaît le nom symbolique "élément". A l'utilisation de la pile, le mécanisme d'instanciation demandera de compléter la partie «représentation» du symbole "élément"; ce sera dans l'exemple "real", mais ceci n'apparaîtra pas dans le texte source qui définit la pile de réels tel qu'il est vu sous l'éditeur.

Par ce mécanisme on a tout loisir de paramétrer les objets que l'on construit par des constantes, des types, des traitements, des "bouts de phrases" du langage, ... indépendamment des possibilités offertes par le langage.

Les symboles à leur tour pourraient se construire selon les règles précédemment décrites. Par exemple, le type pile exporté est:



(paramètres d'instanciation)

L'utilisateur décrit alors son type pile par:

("100" et "real" sont ici les paramètres d'instanciation). La distinction *textuelle* entre le symbole et sa représentation permet alors à l'utilisateur de progressivement dériver l'expression symbolique du problème en un programme exécutable en langage informatique, sans jamais perdre de vue les symboles qu'il dérive.

5. Les Problèmes – et les Réponses

L'éditeur à références concentrées

Problème 1.1: améliorer l'existant

Définir un nouvel outil, c'est d'abord améliorer les outils existants. Concernant l'édition de texte, une qualité implicitement admise est celle d'être "WYSIWYG (What You See Is What You Get)": on veut au minimum que le texte saisi – dans lequel on attache des relations sémantiques – et le texte lu – produit final de la phase d'édition – soient semblables.

Réponse 1.1: interface utilisateur

La solution consiste à définir une *bonne interface utilisateur*. Ceci semble peut-être une première évidence, mais fallait-il encore le dire.

Problème 1.2: environnement de programmation

Comment faire le lien entre un tel texte traité et les outils présents *dans l'environnement* – analyseur syntaxique, compilateur, interpréteur, ...?

Le problème soulevé est double:

- temps supplémentaire d'*analyse syntaxique* sur un texte qui pourrait déjà être syntaxiquement traité; distance entre les outils d'édition et d'exécution – en particulier un interpréteur symbolique travaille avantageusement sur l'arbre syntaxique du programme;
- temps requis pour la *modification du texte*: si l'édition est totalement indépendante des autres outils, on ne sait alors établir de dialogue entre l'éditeur et les outils que par un très pauvre moyen: un flot de caractères, produit par la construction du programme à partir de la R.I. de l'éditeur. Un dialogue s'appuyant sur la notion d'arbre syntaxique serait beaucoup plus efficace, la modularité des outils de modification étant directement induite de la structure arborescente de la représentation.

Réponse 1.2: représentation par des arbres syntaxiques

On constate que, s'il est peut-être mauvais d'accorder trop d'importance à la syntaxe des langages de programmation, c'est tout de même celle-ci qui guide le choix des relations sémantiques qu'on détecte dans le programme. On peut en effet s'interroger sur la sémantique commune à deux morceaux d'instructions, tous deux incomplets, et qui ne représentent aucune structure syntaxique.

On s'imposera donc de ne lier sémantiquement deux "morceaux de texte" que si ces deux "morceaux de texte" sont en fait des *structures syntaxiquement complètes*, quoique éventuellement paramétrées. Par exemple, si l'on a:

"X:=val;" et "Y:=val;"

on ne lie pas:

":=val;"

mais:

"<param>:=val;"

sachant que <param> vaut X ou Y selon les cas.

Problème 1.2.1: création des identificateurs

La seule restriction qu'apporterait vraiment cette contrainte concerne la manipulation lexicale des identificateurs. On ne peut en effet plus définir un identificateur comme le produit de la concaténation de deux autres identificateurs, moyen qui permet de construire sans trop d'efforts des identificateurs *"sémantiquement parlants"*.

Réponse 1.2.1: intérêt de la création
La question est de savoir si la création dynamique de symboles est un aspect bien important, ou bien plutôt une facilité dont on peut aisément se dispenser.

Problème 1.3: traitement des programmes existants
Que faire des programmes existants?

Réponse 1.3.a: concentration lexicale des identificateurs
Une première réponse, et non des moindres, concerne les identificateurs: il faut définir un *outil d'analyse lexicale*, qui regroupe les identificateurs. De là on peut déduire:
- une trace des appels de procédures,
- une trace des utilisations des variables globales,
- une cohérence dans l'évolution possible des identificateurs.

Problème 1.3.a.1: langages à structure de blocs statique
Que dire des langages pour lesquels la déclaration des identificateurs a une portée limitée? Par exemple en Pascal:

```
function A(x:integer);
    function B(x:integer);
    begin
      B:=x+1;
    end;
begin
  A:=B(x)+1;
end;
```

Réponse 1.3.a.1: concentration syntaxique des identificateurs
L'outil de concentration des identificateurs *doit* connaître la portée des déclarations. Il faut alors définir un *outil d'analyse syntaxique* qui attache au niveau de l'en-tête de chaque bloc les déclarations et utilisations des identificateurs apparaissant dans le bloc.

Problème 1.3.a.1.1: surcharge des identificateurs
Il s'agit de la notion de surcharge: dans un contexte donné, on a la visibilité de *deux* objets:
- qui sont de même nature syntaxique (procédure ou fonction);
- qui sont reconnus par le même identificateur;
- qui peuvent avoir le même nombre de paramètres.

Comment alors *distinguer* ces deux objets?

Réponse 1.3.a.1.1: concentration syntaxique des identificateurs avec contrôle de type
Il faut donc définir un *outil d'analyse syntaxique avec contrôle de type*. La surcharge est un concept simple dans son principe: si l'on sait déterminer le type des paramètres effectifs, alors on sait aussi sélectionner le bon sous-programme.

Problème 1.3.a.2: langages à structure de blocs dynamique
Que dire des langages pour lesquels la portée de identificateurs est en partie déterminée dynamiquement? Par exemple en Lisp:

```
(de incr ()
    (1+ a))
(de fct1 (x)
    (let((a x))
        (incr)))
```



```
(de fct2 (x)
    (let ((a x)))
    (incr))
```

- dans incr: a est une variable libre;
- dans fct1: a est une variable locale; la variable libre de incr est liée à cette variable;
- dans fct2: a est une variable locale; mais la variable libre de incr est liée à celle de l'environnement global.

La grande différence avec un langage comme Pascal est que, dans le cas de Lisp, les identificateurs ne servent pas à nommer d'une façon concise des objets définis "plus haut" mais sont bien des symboles à part entière, pour lesquels on déterminera dynamiquement quels objets ils représentent. Dès qu'on parle de ***généricité***, ***héritage***, ***polymorphisme***, on ne peut plus statiquement reconnaître les objets symboliquement nommés.

Réponse 1.3.a.2: exécution symbolique
Il ne peut y avoir de solution automatique simple. La tâche incombe au programmeur de concentrer, *a fortiori*, des identificateurs identiques. On se heurte ici à deux problèmes:
- le passage implicite de paramètres,
- le rendu implicite de valeurs, par «effet de bord».

Déterminer les règles de dépendances sémantiques nécessiterait donc l'exécution symbolique des programmes – par exemple: avec les graphes de flot de données.

Problème 1.3.a.3: langages à structure faible
Que dire enfin des langages à portée trop large? Dans un tel langage, on peut utiliser plusieurs fois une même variable alors que les divers emplois de cette variable sont totalement indépendants. Ceci sera dû: aux limitations du langage – pas de mécanisme de masquage des déclarations; à un souci, bien avouable, de limitation de l'espace mémoire utilisé; à un souci, non moins avouable, de moindre effort – trouver un nouvel identificateur alors qu'on veut dire exactement la même chose, revenir à la zone des déclarations dans le programme alors qu'on ne s'y trouve pas, ...

Réponse 1.3.a.3: analyse globale du programme
Là encore, il n'y a pas de solution simple. Il faudrait à nouveau envisager une analyse globale du programme, pour déterminer à quel endroit une variable non seulement est déclarée mais aussi possède une valeur pertinente.

Réponse 1.3.b: concentration des traitements
Il y a d'autres notions qui impliquent des liens sémantiques et ne sont pas de simples identificateurs. La solution ne saurait être simple. Il faut en effet comprendre ce que le texte veut dire pour y reconnaître un schéma plus général. L'existence d'un tel outil d'analyse des programmes est bien sûr vivement souhaitable, mais est bien sûr aussi un très vaste sujet d'étude.

L'éditeur à références déductives

Problème 2.1: gestion des incohérences

L'éditeur, s'il peut effectuer des contrôles de cohérence, ***ne doit pas***, à l'inverse, interdire des incohérences. Ceci se justifie par le fait qu'en phase d'édition certains états intermédiaires sont fortement incohérents et également fugaces. On ne doit pas présenter un gendarme qui police l'activité de programmation mais plutôt un assistant qui conseille sans vraiment imposer son jugement.

Réponse 2.1: supporter des incohérences passagères

Il s'agit de construire un outil qui contrôle la cohérence des choses mais supporte localement des incohérences. La solution est certes complexe.

L'éditeur à références constructives

Problème 3.1: personnalisation des programmes

La possibilité de symboliquement représenter des zones de programme fait largement appel à la "sensibilité" du programmeur: ceci signifie qu'on risque d'obtenir des textes symboliquement représentés par des conventions très personnelles et difficiles à partager entre plusieurs utilisateurs.

Réponse 3.1.a: rendre les programmes anonymes

La solution pauvre, et simple à réaliser, consiste à toujours autoriser la lecture du *texte réel* – celui qui sera construit en dernier ressort – ignorant ainsi l'expression symbolique définie antérieurement par le programmeur.

Réponse 3.1.b: langage universel

La solution riche serait de définir un jeu de règles et de contraintes pour un standard de présentation des programmes qui soit commun:
- à tous les langages,
- à tous les domaines.

L'objectif serait certainement très ambitieux, mais peut-être aussi trop ambitieux.

6. L'état des travaux

L'éditeur à références concentrées

C'est le premier éditeur auquel on s'est attaché. Du fait des difficultés rencontrées, on a distingué deux niveaux:
- *L'évaluation des textes*: c'est un «éditeur ligne», qui retourne en écho à la commande d'évaluation la forme évaluée d'un schéma de textes. Il a servi de support à tous les exemples présentés dans la suite.
- *L'«interface conviviale»*: c'est un «éditeur page», qui permet à l'utilisateur de ne travailler qu'avec la forme évaluée des textes. Le programme est très partiellement défini: il permet la manipulation de textes structurés mais ne gère pas de «zones concentrées» de texte.

Les deux programmes sont écrits en LeLisp V15.2; ils sont relativement rapides à l'exécution mais assez gros consommateurs d'espace mémoire (2 Moctets pour un bon confort d'utilisation).

L'éditeur à références déductives

Le point n'a pas été abordé dans sa généralité. Il s'agirait, d'après les choix effectués pour le premier éditeur, de définir un type d'expression pour la manipulation des «arbres de textes». Dans l'état actuel des travaux l'éditeur ignore *a priori* les intentions du programmeur; il peut en revanche *a posteriori* vérifier les non contradictions – contrôle de type, décompilation contextuelle, visibilité des identificateurs, ...

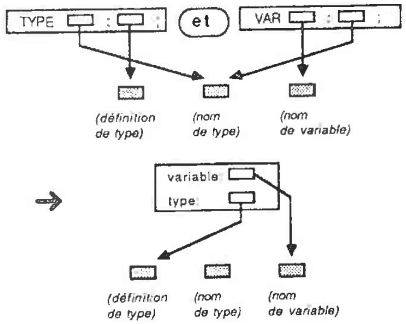
L'éditeur à références constructives

Il n'est pas non plus vraiment réalisé. On peut peut-être l'expliquer par la faible «convivialité» de l'évaluateur des textes: ce dernier travaille sur des listes Lisp et fournit un texte structurellement plat – une suite de caractères; son utilisation n'est donc pas d'un très grand confort. L'expression symbolique de certains traits du programme ne saurait être réellement exploitée avec bénéfice que dans une «interface conviviale» qui sache masquer à l'utilisateur des détails techniques de mise en œuvre pour lui permettre d'avoir toujours une vue de son programme sous un jour ***agréable***.

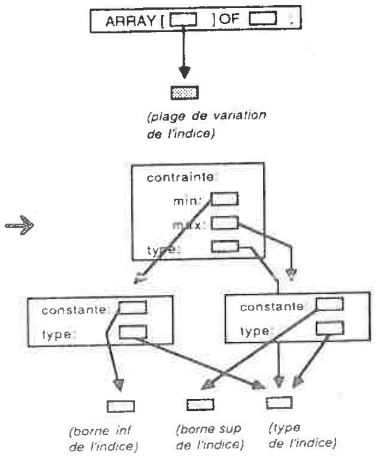
Annexe : exemple de déductions

Dans les schémas, on reconnaît:
– en haut: les modèles génériques paramétrés;
– en bas: les paramètres effectifs, qui réalisent les instances des types paramétrés.

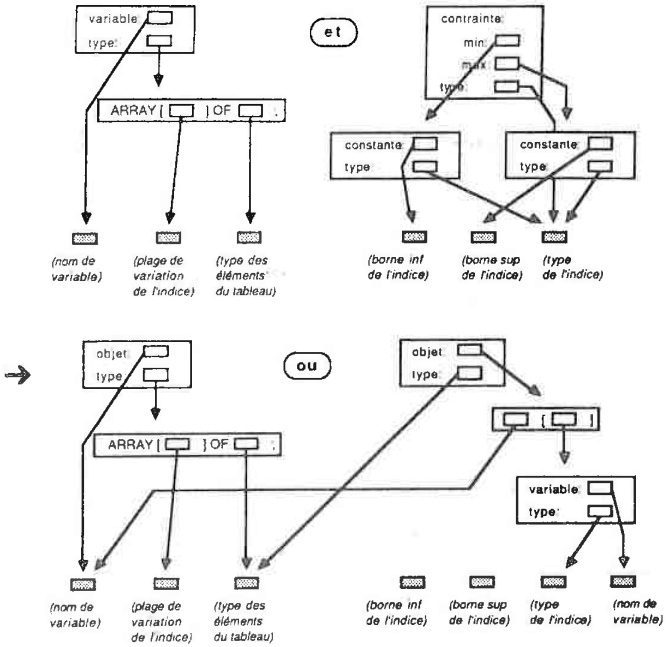
(1) De la déclaration du type et de la déclaration de la variable, on déduit le schéma des propriétés sémantiques de la variable:



(2) On détaille ici la définition d'un type tableau, qui définit le schéma des contraintes à imposer aux indices du tableau – on notera que les deux bornes font naturellement référence au même type d'indice:



(3) Des propriétés de la variable et des contraintes sur la définition du type tableau, on déduit les deux sortes d'objets dont la désignation est préfixée par le nom de la variable:
- le nom simple de la variable: il est du type tableau;
- le nom de la variable indicé: il est du type des éléments du tableau.



Annexes

Chapitre 8:

Les Editeurs

On présente les deux éditeurs qu'on a implantés, tous deux en LeLisp V15.2:
- l'éditeur ligne: on travaille sur la forme non évaluée des «textes»; on peut en *voir* la forme évaluée par une commande explicite;
- l'éditeur page: on travaille sur la forme évaluée des «textes»; de plus, comme son nom l'indique, c'est un éditeur plein écran.

L'éditeur ligne est entièrement implanté; mais s'agissant d'un éditeur ligne il est d'un emploi peu confortable. L'éditeur page n'est que partiellement réalisé: il permet la manipulation d'un texte structuré, mais ne supporte pas la notion de ***concentration*** des textes.

brisé sur la barrière de la complexité (une fois de plus)

Le fil conducteur des travaux n'a pas été l'idée de définir un nouvel outil, un éditeur convivial qui offre de nouvelles fonctionnalités, mais de proposer une *autre* manière de mener l'étape de la programmation, ce qu'on a appelé l'«approche inductive»: répondre d'emblée au problème, précisément et incomplètement, et accroître les fonctionnalités du programme par un procédé de *grossissement* qui préserve l'effort antérieurement fourni.

L'objectif n'était donc pas de construire un produit «industriel» ni même de réaliser une maquette démonstrative: c'était, et cela reste, de **valider** la démarche; pour ce faire, il fallait développer un exemple de taille suffisamment importante, progressivement et entièrement; pour développer l'exemple, il fallait un outil, support de l'expression. C'est pour cet outil que l'on s'est «brisé sur la barrière de la complexité (une fois de plus)», en paraphrasant T. Winograd. L'outil est imparfaitement construit, l'exemple est incomplètement traité, la démarche n'est pas pleinement validée.

Une première maquette a été faite; elle travaillait sur la *syntaxe concrète*, partiellement définie – la référence ref était un peu moins générale qu'à présent.
- Le programme: 80 lignes.
- La saisie: on construit des listes Lisp sous un éditeur classique.
- L'affichage: la forme évaluée est calculée par l'appel d'une fonction Lisp.

Le programme est très simple, mais le développement d'un exemple de grande taille est particulièrement ardu.

Une deuxième maquette a été faite: c'est l'*éditeur ligne*; il travaille sur la *syntaxe abstraite*.
- Le programme: plus de 3000 lignes.
- La saisie: on se déplace sur l'«arbre des textes»; on peut modifier les nœuds de l'arbre directement sous l'éditeur.
- L'affichage: la forme évaluée peut être obtenue par une commande *explicite* sous l'éditeur, mais on n'y accède pas directement.

L'éditeur, dans son principe, offre les fonctionnalités requises pour le développement complet d'un exemple; en pratique, l'utilisation d'un éditeur ligne fait de tout développement d'une certaine envergure une tâche très pénible.

Une troisième maquette a été faite: c'est l'*éditeur page*; il travaille sur la *syntaxe abstraite*, avec une interface plein écran, multi-fenêtre, multi-tampon.
- Le programme: près de 5000 lignes.
- La saisie: on travaille directement sur le texte, par déplacement du curseur dans une fenêtre d'édition.
- L'affichage: on travaille sur la forme évaluée; ce qu'on saisit et ce qu'on voit sont deux choses confondues.

Compte tenu des orientations actuelles dans le programme, l'éditeur dans un premier temps travaillerait *plutôt* sur la *syntaxe concrète*; mais en l'état il ne travaille sur aucune syntaxe: on réalise l'édition conviviale d'un document structuré mais pas la «concentration des textes».

Chapître 8.1:

L'éditeur ligne : Manuel de l'utilisateur

On présente *l'éditeur ligne*, du point de vue de l'utilisateur. Ses caractéristiques sont les suivantes:

- on travaille sur un arbre de «textes» non évalué;
- on peut *voir* la forme évaluée des «textes» par une commande *explicite*;
- l'éditeur retourne en écho le nœud sur lequel on est positionné;
- on modifie l'arbre par des commandes "à distance".

1. généralités

Le programme est écrit en LeLisp. Après chargement des fichiers *ad hoc*, l'appel de fonction: (anal) permet de se placer sous l'éditeur.

On adopte deux conventions:
- l'éditeur renvoie toujours un écho du "terme" sur lequel on est placé (ce "terme" serait, en d'autres lieux, la position du curseur);
- l'attente d'une commande s'affiche sous la forme:

    [nom_de _phylum]> ?

où [nom_de_phylum] est le nom du phylum courant.

Notes:
1 Dans la suite, on désigne par "terme" le lieu sur lequel on est placé, ou susceptible d'être placé: c'est un nœud de l'arbre des textes, ou dit autrement un opérateur instancié de la syntaxe des textes (c'est un terme trm, un atome atm, un environnement env, ...).
2 Quand on est placé sur une *liste* de "termes", et que la liste est vide, l'écho renvoyé est: ...
3 Au départ, on est placé dans l'environnement global. Il s'affiche donc:

    ...
    trm> ?

qui signifie:
- que l'environnement global est vide,
- et qu'on attend un terme trm.

2. syntaxe

Sauf cas particuliers, les commandes sont frappées sur une ligne, en quantité quelconque, elles sont interprétées, puis le contrôle revient à l'utilisateur.
La syntaxe générale est:

    <nom_de_cmde> <num>

où <nom_de_cmde> est un caractère, et <num> est un nombre (en base 10 ...) ou le signe étoile '*', qui représente «un nombre très grand» (100). Par défaut, <num> est égal à 1. Les blancs sont non significatifs.

3. commandes

3.1. holophraste: h

L'holophraste est un entier, initialisé à une valeur donnée à la racine d'un sous-arbre des textes – le "terme" sur lequel on est –, puis décrémenté selon des règles préétablies quand on affiche les champs d'un opérateur de ce sous-arbre.
Par exemple:
- sachant que l'opérateur def est affiché avec un holophraste N,
- l'environnement de définition est affiché avec un holophraste N-1,
- la représentation est affichée avec un holophraste N-2.

Une bonne manière d'exploiter cette notion est de ne pas réaliser l'affichage d'un champ quand l'holophraste est nul.

COMMANDE

h <num>
    fixe l'holophraste à <num>.
    * = holophraste maximal.

3.2. impression: p

COMMANDE

p <num>
    affiche <num> "termes", à partir de la position courante.
    * = affiche la liste des "termes", à partir de la position courante.

3.3. déplacement horizontal: l r

Ces commandes concernent les opérateurs définis par une *liste* de champs.

COMMANDES

l <num>
    déplacement de <num> positions vers la gauche.
    * = positionnement sur le premier "terme".
r <num>
    déplacement de <num> positions vers la droite.
    * = positionnement après le dernier "terme".

3.4. déplacement vertical: e q

Ces commandes permettent d'entrer dans les champs d'un opérateur et d'en sortir.

COMMANDES

e <num>
    porte d'entrée: entre dans le <num>-ième champ de l'opérateur de la position courante, s'il existe.
q
    porte de sortie: quitte le champ d'un opérateur: la position courante est maintenant cet opérateur.
    Si l'on est placé sur un des termes trm de l'environnement global, on ne peut rien quitter: la commande 'q' déclenche l'affichage du message d'erreur:
        q: sommet

3.5. modification: i j k c

Ces commandes sont utiles pour les opérateurs définis par une *liste* de champs.

COMMANDES

i <num>
    insertion de <num> "termes" *derrière* la position courante.
    * = insertion d'une liste de "termes".
j <num>
    insertion de <num> "termes" *devant* la position courante.
    * = insertion d'une liste de "termes".
k <num>
    suppression de <num> "termes", à partir de la position courante.
    * = suppression des "termes" de la position courante jusqu'à la fin.

c
    changement de l'opérateur de tête d'un opérateur de liste: le nouvel opérateur doit être visible dans le phylum de l'opérateur de liste.

3.6. saisie: ( )

C'est un appel récursif de l'éditeur. Entre une commande '(' et une commande ')', toutes les variables d'état sont sauvegardées, elles sont restaurées à la fin.

COMMANDE

(
    début de saisie: on appelle récursivement l'éditeur à partir de cette position.
)
    fin de saisie: on revient à la position du début de saisie.
    fin de session: on quitte l'éditeur, quand il n'y a eu aucune commande de début de saisie antérieure: '('.

3.7. macro: d y m -

On offre la possibilité de définir des "macros", suites de commandes encadrées par une commande de début et une commande de fin de définition de macro. Ces suites sont ou non définies sur une ligne unique – l'utilisateur reprend ou non le contrôle au cours de la définition de la macro.

COMMANDES

d <nom>
    début de définition de la macro de nom <nom>;
    <nom> est un caractère.
y
    fin de définition d'une macro.
m <nom>
    appel de la macro de nom <nom>.
    Si le caractère <nom> n'est pas surchargé – n'est pas une des commandes définies ici – on peut omettre 'm'. L'appel de la macro est alors: <nom>.
-
    paramètre de la macro.
    A la définition de la macro, la commande est sans effet. A l'appel d'une macro paramétrée, '-' est remplacé par le nombre de l'appel.

Par exemple, on définit la macro 'r':
    d/(h-p*)y
- appeler récursivement l'éditeur,
- fixer l'holophraste à une valeur en paramètre,
- afficher tous les "termes" à partir de la position courante.
On l'utilise:
    / : correspond à: (hp*)
    /3 : correspond à: (h3p*)
    /* : correspond à: (h*p*)

MACRO ECHO: w

L'écho est défini comme l'appel de la macro 'w'. Initialement, celle-ci est définie par:
  dw(h0p)y
qui signifie:
- on définit la macro 'w',
- on sauvegarde la valeur de l'holophraste (appel récursif), on fixe celui-ci à zéro, et on imprime un unique "terme".
L'utilisateur peut donc *modifier* l'écho qu'il reçoit.

3.8. fichier/buffer: f b

Ce sont les commandes d'entrées/sorties de l'éditeur, soit sur fichier, soit dans un buffer. Le buffer est un tampon provisoire, qui disparaît à la sortie de l'éditeur. Les entrées/sorties sur buffer sont toujours réalisées *par copie* – on n'autorise donc pas le partage d'un sous-arbre par deux nœuds distincts de l'arbre des textes.

COMMANDES

f [i j p] <num>
    fichier: le nom du fichier est demandé à part.
fi <num>
    insère <num> "termes" du fichier "File Input" *derrière* la position courante.
    * = insère la liste des "termes" du fichier.
fj <num>
    insère <num> "termes" du fichier "File Input" *devant* la position courante.
    * = insère la liste des "termes" du fichier.
fp <num>
    écrit <num> "termes" dans le fichier "File Output" à partir de la position courante – l'ancien fichier est écrasé.
    * = écrit la liste des "termes" dans le fichier.

b [i j p] <num>
    buffer: le nom du buffer est demandé à part.
bi <num>
    insère <num> "termes" du buffer "Buffer Input" *derrière* la position courante.
    * = insère la liste des "termes" du buffer.
bj <num>
    insère <num> "termes" du buffer "Buffer Input" *devant* la position courante.
    * = insère la liste des "termes" du buffer.
bp <num>
    écrit <num> "termes" dans le buffer "Buffer Output" à partir de la position courante – l'ancien buffer est écrasé.
    * = écrit la liste des "termes" dans le buffer.

3.9. commande Lisp: !

La commande définit un "point d'entrée" de l'interpréteur Lisp.

COMMANDE

!

commande Lisp.
La commande est saisie, puis évaluée, et son résultat est affiché.
Par exemple:

```
!           ; commande
? (+ 3 2)   ; saisie de la commande Lisp
5           ; affichage
```

3.10. affichage évalué: a

COMMANDE

a <num>

affichage évalué: évalue les "termes" sur lesquels on est placé, puis affiche <num> "termes évalués".
* = affiche tous les "termes évalués".

Chapitre 8.2:

L'éditeur page: Guide de l'utilisateur

On présente l'*éditeur page* du point de vue de l'utilisateur. Ses caractéristiques sont les suivantes:

- l'«interface conviviale» est calquée sur celle que propose l'éditeur *emacs* – plein écran, multi-fenêtre, multi-tampon;
- on travaille sur la forme *évaluée* des textes; les liens d'utilisation de «textes» sont gérés par l'éditeur;
- on modifie le texte par des insertions ou des suppressions directes.

En l'état, l'éditeur ne gère pas les liens d'utilisation: le texte structuré qu'on édite est un arbre, à la manière des éditeurs structurés traditionnels.

0. L'état des travaux

Comme il a été dit, l'*éditeur page* n'est que partiellement réalisé. Il présente une interface très proche de l'éditeur *emacs*.

0.1. la syntaxe de l'éditeur

La *syntaxe conviviale* des «textes» est un sous-ensemble strict de la *syntaxe commune* présentée auparavant (cf. Chapitre 3.1, *«Présentation de la Syntaxe Complétée»*), on rappelle cette dernière ici:

syntaxe commune

```
(1) env ::= ε | trm env
(2) trm ::= def | ref | env
(3)       def  ::= <nom> env rep
(4)       ref  ::= <nom> env
(5) rep ::= ε | atm rep
(6) atm ::= stg | use | <string> | rep
(7)       stg  ::= <nom>
(8)       use  ::= <nom> env
```

(1) un environnement est une liste de termes.
(2) un terme est une définition def ou une référence ref ou un environnement env.
(3) une définition est: un nom <nom>, un environnement, une représentation.
(4) une référence est: un nom <nom>, un environnement.
(5) une représentation est une liste d'atomes.
(6) un atome est une *string* stg ou une utilisation use ou une chaîne de caractères <string> ou une représentation rep.
(7) une *string* stg est: un nom <nom>.
(8) une utilisation est: un nom <nom>, un environnement.

syntaxe conviviale

```
(a) env ::= ε | trm env
(b) trm ::= def
          def  ::= <nom> env rep
(c) rep ::= ε | atm rep
(d) atm ::= <string> | rep
```

(a) un environnement env est un tampon pour l'édition de tampons; on l'appelle dans la suite un *«Buffer-Edit»*.
(b) un terme trm est une définition def; c'est une ligne d'une fenêtre d'édition d'un *Buffer-Edit*.
(c) une représentation rep est un tampon d'édition d'un «texte»; c'est le tampon d'édition classique d'un éditeur, qu'on appelle ici *«Buffer»*.
(d) un atome atm est la position du curseur dans une fenêtre d'édition d'un *Buffer*, c'est le «point d'insertion» sous l'éditeur qui se rattache à la notion classique de curseur.

Remarques:

1 Le champ <nom> d'une définition def est décomposé en deux champs:
- le nom du texte,
- le nom du fichier auquel est rattaché le texte ("[None]" par défaut).

2 Une chaîne de caractères <string> devient ici un caractère:
- un caractère «graphique»: " ", "!", "'", "#", ...
- le caractère de retour à la ligne: NL.

La distinction est transparente à l'utilisateur – si ce n'est la forme graphique du caractère NL – mais fondamentale dans l'implantation.

0.2. l'utilisation d'emacs

L'*éditeur page* est très fortement inspiré, dans sa présentation, de l'éditeur de texte *emacs*.

- Les menus
  On ne présente pas, à la façon du Macintosh par exemple, des menus déroulants, des fenêtres de dialogue, ...; il s'agirait de placer, au-dessus de l'éditeur actuel, une couche à "fort degré de convivialité" qu'on n'a pas jugée indispensable dans un premier temps.

- Le multi-fenêtrage
  On *ne réutilise pas* un gestionnaire de fenêtres déjà existant. La raison est qu'on travaille sur des structures de données complexes et qu'il est plus simple de tout redéfinir – les structures des données et la gestion des fenêtres.
  Le gestion des fenêtres n'est qu'une petite partie, non négligeable mais non considérable du programme; la difficulté n'est pas tant de gérer les fenêtres que de savoir ce qu'il faut mettre dedans.

- Le réaffichage
  On réalise un *réaffichage synchrone* du contenu des fenêtres, ce qui ne va pas sans poser de nombreux problèmes; *emacs* réalise un réaffichage asynchrone, c'est-à-dire qu'une tâche de fond est spécialement chargée du réaffichage et que la tâche qui modifie les tampons ne s'en soucie pas.

- Les commandes «textuelles»
  En plus des commandes d'*emacs*, on définit de nouvelles commandes, propres aux «textes». Le grand nombre de commandes résultant rend l'emploi de l'éditeur un peu délicat – quoiqu'on introduise des *commandes assistées* pour aider l'utilisateur dans le choix de la commande. Ce point mériterait sûrement d'être approfondi.

On indique par une astérisque les endroits où l'on s'éloigne d'*emacs*.

1. généralités

1.1. l'écran

L'écran est composé:
- de fenêtres,
- d'une ligne de contrôle.

La ligne de contrôle n'est pas une fenêtre*; c'est la dernière ligne d'écran.
Une fenêtre est composée:
- de lignes de textes: au minimum deux lignes;
- d'une barre d'information, qui donne:
  - le nom du Buffer ou Buffer-Edit,
  - le nom du fichier auquel il se rattache,
  - l'holophraste (pour un Buffer).

1.2. les commandes

On distingue en général plusieurs types de caractères:
- les caractères «graphiques»: ils sont insérés directement dans le texte;
- le caractère NL (retour chariot): on insère un retour à la ligne;
- les autres caractères de contrôle: ce sont les commandes.

On note, selon l'usage, un caractère de contrôle par le signe "^" – ^X = contrôle-X, ESC = ^[= escape, DEL = ^? = delete.

commandes générales

^G bell – annuler la commande en cours de frappe.
^L rafraîchir l'écran.
^U préfixe numérique, suivi d'un nombre en base 10: la commande frappée ensuite est répétée le nombre de fois indiqué.
- par défaut (pas de nombre en base 10 frappé) ce nombre vaut 4;
- ^U répété plusieurs fois est un facteur multiplicatif (^U^U^U = nombre 64);
- "-" annule ^U: le nombre est remis à 1.

On ne donne pas pour les commandes présentées dans la suite la sémantique de la *répétition*: on pourra intuitivement la deviner.

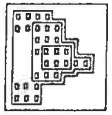
ESC préfixe de commandes.
^X préfixe de commandes.

1.3. le texte structuré

Un texte structuré est un arbre dont les nœuds sont:
– soit une suite de caractères (caractères «graphiques» ou retour à la ligne);
– soit un texte structuré.

Puisqu'on travaille en dehors de toute grammaire, il n'y pas de *schéma de décompilation* des arbres. En revanche on établit une *indentation*, en alignant les débuts de lignes d'un texte structuré.

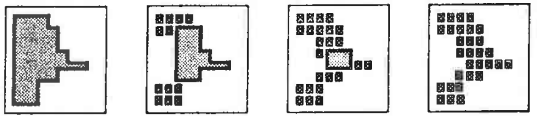
Par exemple:



(chaque zone encadrée correspond à la zone d'affichage d'un texte).

1.4. l'holophraste

L'holophraste* est un entier qui mesure à quelle profondeur de l'arbre on se situe.
Par exemple:



holo = 1 holo = 2 holo = 3 holo = 4

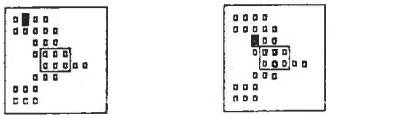
(chaque zone coloriée est le point d'insertion éventuel sous l'holophraste donné).

L'holophraste est *non contraignant*, c'est-à-dire qu'il peut être supérieur en valeur à la profondeur où l'on se situe dans l'arbre. On distingue donc deux holophrastes:

- *l'holophraste logique*
  C'est une valeur entière, qui est l'holophraste sous lequel l'utilisateur veut travailler.

- *l'holophraste physique*
  C'est une valeur entière, maximum de l'holophraste logique et de la profondeur du point d'insertion où l'on est situé.

Par exemple:



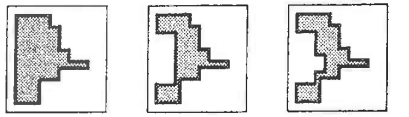
holo logique = 3 holo logique = 3
holo physique = 2 holo physique = 3

commandes d'holophraste

* ^X-^U incrémente l'holophraste de 1.
* ^X-^D décrémente l'holophraste de 1.
* ^X-^H fixer l'holophraste logique à la valeur de l'holophraste physique courant.

1.5. la zone d'accès

La zone d'accès est la zone d'écran où l'on peut être effectivement placé. Elle dépend à la fois de la longueur de la ligne *et* de la hauteur du premier caractère de la ligne.
Par exemple:



1.6. commandes diverses

ESC-^X aide en ligne.
^X-s interruption; on revient sous l'interpréteur Lisp.
^X-^C quitter.
* ^X-! envoi d'une "commande", c'est-à-dire évaluation d'une S-expression par l'évaluateur Lisp. On ouvre une fenêtre de dialogue avec l'interpréteur, qu'on quitte en lui faisant évaluer la S-expression nil: ().
^X-(début de macro.
^X-) fin de macro.
^X-^E exécution de la macro.
* ^X-^L réaffichage complet de l'écran (aspect technique: l'éditeur reconstruit les fenêtres de Buffer-Edits).

2. le curseur

2.1. les curseurs

Le fait qu'on se déplace sur un texte structuré amène à distinguer plusieurs sortes de curseurs.

• *le curseur logique*
C'est un point isolé de l'écran, défini par une abscisse et une ordonnée – un numéro de colonne et un numéro de ligne d'écran.

◦ *le curseur physique*
C'est la position "physique" du curseur logique: elle tient compte de la zone d'accès à l'écran qui:
– ramène le curseur physique en fin de ligne quand le curseur logique a une abscisse trop élevée,
– amène le curseur physique en début de ligne du texte quand le curseur logique a une abscisse trop basse.

La gestion des curseurs logique et physique permet de "traverser" l'écran avec une abscisse logique constante, sans être "chahuté" du fait de lignes finissant trop tôt ou commençant trop tard.

• *le curseur*
Le curseur proprement dit est la zone d'écran correspondant au point de l'arbre sur lequel on est placé:
- ou bien il s'agit d'un caractère isolé, et le curseur s'identifie alors au curseur physique,
- ou bien il s'agit d'un texte structuré, et le curseur est une zone étendue de l'écran.

◦ *le Buffer local*
Le Buffer local est le premier curseur au-dessus du curseur courant.
On gagne à identifier la notion puisque, dans un texte donné sous un holophraste donné, les curseurs sur lesquels on se place appartiennent très exactement au Buffer local en tant que caractères ou arbres de texte. Le Buffer local est donc précisément le Buffer qu'on lit ou qu'on modifie à un instant donné.
Note: quand l'holophraste vaut 1, le Buffer local s'identifie au Buffer d'édition.

forme visuelle des curseurs

• Le curseur logique n'est pas visible.
◦ Le curseur physique est un curseur classique clignotant.
• Le curseur est une zone d'écran en inverse vidéo.
• Le Buffer local n'est pas visible.

modes <curseur>

On a deux modes de déplacement dans un Buffer:
– le mode logique: le déplacement est relatif au *curseur*;
– le mode physique: le déplacement est relatif au *curseur physique*.
Ce second mode est signalé par l'affichage "<cursor>" sur la ligne de contrôle. Il correspond au déplacement classique.

commandes

* ^X-^T bascule en mode logique / physique.
* ^T bascule locale, qui touche uniquement la commande de déplacement qui suit.

2.2. déplacement

On donnera aux préfixes ^U et ^T la sémantique intuitive qu'on devine.

^A début de ligne.
^E fin de ligne.
^F curseur suivant.
^B curseur précédent.
^N ligne suivante.
^P ligne précédente.
ESC-f mot suivant.
ESC-b mot précédent.

^Z déroulement d'une ligne d'écran en bas (scroll).
ESC-z déroulement d'une ligne d'écran en haut (scroll).
^V page suivante.
ESC-v page précédente.

* ESC-ESC ramener le curseur logique au curseur physique.
* ESC-^N contraction de ESC-ESC et ^N
* ESC-^P contraction de ESC-ESC et ^P

ESC-< début de Buffer.
ESC-> fin de Buffer.
ESC-, début de fenêtre.
ESC-. fin de fenêtre.
ESC-! ligne courante en début de fenêtre.
* ESC-a curseur logique en début de curseur.
* ESC-e curseur logique en fin de curseur.
* ESC-[curseur en début de Buffer local.
* ESC-] curseur en fin de Buffer local.

* ESC-n *vraie* ligne suivante (v. note).
* ESC-p *vraie* ligne précédente (v. note).

ESC-= numéro de ligne courante.
^X-= aller à la ligne de numéro donné.

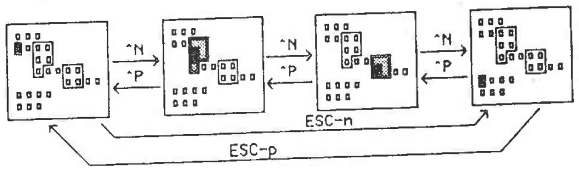
^@ poser une marque.
^X-^X échanger la marque et le curseur.

ESC-^Z page suivante de la fenêtre suivante – page précédente si le préfixe numérique (défini par ^U) est différent de 1.

Note

- Un effort particulier a été fait pour que les commandes de déplacement soient toutes *inversibles*: par exemple, ^F^N^N est l'inverse de ^P^P^B (généralement).

- La *vraie* ligne précédente ou suivante tient compte de l'indentation.
  Par exemple:



2.3. recherche

La recherche s'intéresse aux chaînes de caractères indépendamment de la structure du texte. Toute commande frappée autre que celles présentées ci-après fait quitter le mode de recherche; la commande est de plus interprétée.

^S recherche en avant du curseur.
^R recherche en arrière du curseur.

commandes de recherche

<caractère> caractère suivant recherché; la recherche est *incrémentale*.

^S rechercher le motif suivant dans le texte.
^R rechercher le motif précédent dans le texte.
^T bascule en mode normal / expression régulière
^G abandon: on est replacé au point de début de la recherche.
ESC quitter: on reste au point atteint.
* ^@ le caractère pointé par le curseur physique est pris comme caractère suivant recherché.
DEL effacer le dernier caractère recherché – note: DEL mais pas ^H.

*syntaxe des expressions régulières**

"^" début de ligne de texte.
"$" fin de ligne de texte: le caractère NL de retour à la ligne – accessible par ^@.
"." un caractère quelconque, sauf NL.
"\b" un caractère séparateur de mots.
"\w" un caractère composant de mots.
"\" le caractère suivant n'est pas interprété.
"[x-z]" caractère compris entre "x" et "z" – dans l'ordre des codes ASCII.
"*" répétition un nombre quelconque de fois du motif précédent.

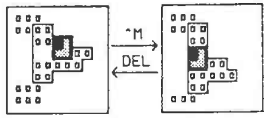
2.4. modification

La modification textuelle – au point d'insertion – ne concerne que les Buffers.

insertion de caractères

" "..."~" le caractère est inséré dans "l'angle supérieur gauche" du curseur, lequel est décallé d'une colonne vers la droite.
^M un retour à la ligne est traité comme un caractère «graphique», en respectant les règles d'indentation.
^O identique à ^M, mais le curseur reste à sa position d'écran.

Par exemple:



destruction

DEL ^H destruction du curseur précédant le curseur courant: ce peut être un caractère ou une zone étendue.
Si le curseur est au début du Buffer local, la destruction est refusée – elle conduirait à supprimer un curseur qui n'appartient pas au Buffer dans lequel on travaille, ce qui serait incohérent.
^D destruction du curseur sur lequel on est placé.
ESC-h destruction du mot précédant le curseur.
Si la zone comprise entre le debut du mot et le début du curseur courant chevauche le Buffer local et un autre Buffer, la destruction est limitée aux éléments du Buffer local.
ESC-d destruction du mot suivant le curseur.
Même remarque que pour ESC-h.

fichiers

^X-^I	insérer le fichier de nom donné devant le curseur.
^X-i	insérer le Buffer de nom donné devant le curseur. Note: l'insertion est faite «à plat», c'est-à-dire qu'on perd l'information: «la zone de texte provient du Buffer de nom "XXX"».
^X-^O	lire dans le fichier attaché au Buffer l'ancienne version du texte.
^X-r	lire le fichier de nom donné, et le placer dans le Buffer. Note: pour les commandes de lecture de fichier, les fenêtres qui *dépendent* du Buffer courant sont fermées – parce qu'elles n'existent plus après la lecture.

Kill Buffer

Le Kill Buffer est un Buffer qu'il est spécialement aisé de remplir ou d'insérer dans le Buffer courant. Il peut aussi être édité dans une fenêtre.
Une suite de commandes plaçant des "bouts de texte" dans le Kill Buffer correspond à des ajouts; mais la première commande écrase le contenu antérieur du Kill Buffer.

Note: l'ajout tient compte de la structure du texte, c'est-à-dire que la zone de texte placée dans le Kill Buffer ne chevauche pas plusieurs Buffers locaux.

^K	détruire la ligne et la placer dans le Kill Buffer: - si le curseur est en fin de ligne, on le place isolément dans le Kill Buffer; - sinon, on place l'ensemble des curseurs présents du curseur courant au premier curseur de fin de ligne, non compris ce dernier.
* ESC-k	identique à ^K, mais ne détruit pas la ligne.
^W	détruire la région, c'est-à-dire le texte compris entre la marque (posée par ^@) et le curseur courant, et la placer dans le Kill Buffer.
* ...	identique à ^W, mais ne détruit pas la région.
ESC-^A	ajouter en fin de Buffer de nom donné la région. Note sur la marque: la marque est maintenue à une place cohérente, même si des modifications à d'autres points du texte la déplacent virtuellement.
* ^X-^K	détruire le Buffer local courant et le placer dans le Kill Buffer.
* ^X-k	identique à ^X-^K, mais ne détruit pas le Buffer local.
^Y	insérer le contenu du Kill Buffer devant le curseur courant.

Du fait du grand nombre de commandes possibles, on inclut une commande assistée:

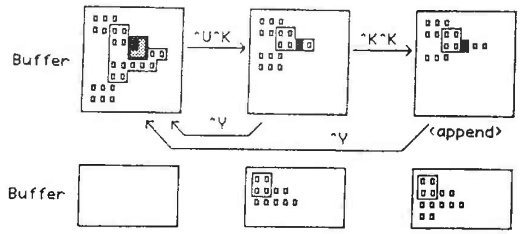
* ESC-^K c'est la commande:
"Copy/Move to Kill Buffer: Buffer/Region"
Copier / Détruire vers le Kill Buffer: le Buffer local / la région.

ajout en fin de Buffer local

On introduit un *mode* d'ajout en fin de Buffer local, parce que:
- l'insertion par défaut est toujours faite *devant* le curseur courant;
- les destructions sont nécessairement limitées au Buffer local: sans une option d'ajout en fin on ne pourrait satisfaire la contrainte d'*inversibilité* des commandes.

L'idée de *mode* est très opposée à la philosophie d'*emacs*: on l'introduit parce qu'elle se révèle nécessaire, mais on le dit clairement – il s'affiche sur la ligne de contrôle le message: "Append to local Buffer".

Par exemple:



On est placé automatiquement en mode «ajout» quand on "dépasse", en fin, le Buffer local. On peut aussi s'y placer explicitement:

* ^^	se placer en mode «ajout».

3. les fenêtres

On distingue trois sortes de fenêtres:
– les fenêtres de Buffer: on y gère du texte;
– les fenêtres de Buffer-Edit: on y gère des Buffers;
– les fenêtres d'alarme: on y place, provisoirement, des messages.
Les fenêtres de Buffer-Edit sont structurellement identiques aux fenêtres de Buffer: c'est l'interprétation des commandes frappées qui change. Les fenêtres d'alarme sont en revanche différentes*: cela tient à ce qu'une fenêtre de Buffer est très grande consommatrice en espace mémoire (en cellules de liste Lisp) et qu'un message est en général un fichier de caractères de taille importante.

commandes

^X-1	la fenêtre courante devient l'unique fenêtre d'écran.
^X-d	détruire la fenêtre courante.
* ^X-2	couper la fenêtre courante en deux; l'éditeur *devrait* gérer de manière cohérente les deux fenêtres ouvertes sur le même tampon d'édition: en fait les deux fenêtres existent mais une seule des deux est maintenue cohérente.
^X-n	déplacement vers la fenêtre suivante.
^X-p	déplacement vers la fenêtre précédente.
^X-^Z	réduire la fenêtre d'une ligne.
^X-z	agrandir la fenêtre d'une ligne.

4. les tampons

Un tampon est une définition de «texte» def; il se définit par:
– un nom de «texte» et un nom de fichier ("[None]" par défaut);
– un environnement des définitions: c'est une liste ordonnée de tampons;
– une représentation: c'est un texte structuré.

forme visuelle des champs du tampon

- Le nom du «texte» et du fichier sont inscrits dans la barre d'information.
- L'environnement est un Buffer-Edit.
- La représentation est un Buffer.

4.1. le Buffer

Le Buffer est un texte structuré:
– on peut fixer l'holophraste;
– on peut se déplacer, en mode logique / physique;
– on peut modifier le texte, par insertions, destructions, ...

4.2. le Buffer-Edit

Le Buffer-Edit est un texte structuré pour les *déplacements* (pour les commandes de modification, voir §2.8):
– on ne peut pas fixer l'holophraste, qui n'est pas défini;
– on peut se déplacer, en mode logique / physique;
– on ne peut pas modifier le "texte" par insertion ou destruction directe de caractères; la modification du texte est un «effet de bord» d'autres commandes.

Par exemple, dans la vue d'écran suivante, on voit le Buffer et le Buffer-Edit relatifs à un tampon:

```
(de cpltt-env arg
  (cons 'env
        (eval-map 'cpltt-TRM arg)))

(de cpltt-rep arg
  (cons 'rep
Buffer  /texte  File  test/aux.rep  Hol  #3  #3
Size  Type    Buffer          File
----  ----    ------          ----
      Def     texte-1         [None]
      Def     texte-2         test/prov.rep

Buffer Edit - /texte/  File  test/aux.rep
```

Il correspond au «texte»:

```
(def texte
   ((def texte-1 ...)
    (def texte-2 ...))
   ("(de cpltt-env arg" "^M"
    ...))
```

sauvegardé dans le fichier "test/aux.rep".

Note sur les champs d'un Buffer-Edit

- Le champ Size est un "faux" champ, qui n'est pas rempli*.
- Le champ Type est toujours rempli par "Def": les seuls termes trm définis sont les définitions de texte def.
- Le champ Buffer est le nom du «texte» tel qu'il a été défini auparavant dans l'*éditeur ligne*.
- Le champ File est le nom du fichier auquel est attaché la définition du «texte» – "[None]" si aucun fichier n'est associé à ce «texte».

4.3. les noms

L'*éditeur page* reprend dans la gestion des «textes» les idées de l'*éditeur ligne*.

Au "sommet" des tampons, on trouve l'*environnement global*, qui est la liste des tampons qu'on a ouvert lors de la session sous l'éditeur au plus haut niveau dans l'imbrication des tampons. Il est nommé par:

/

(qui correspondrait à la racine sous UNIX). Par nature, il s'agit d'un Buffer-Edit.

Un *Buffer* est nommé par le nom du tampon qu'il représente:

texte-1 (notation relative)
/texte/texte-1 (notation absolue)

en notation absolue, on reprend, à partir de la racine, les noms des tampons englobants, séparés par "/".

Un *Buffer-Edit* est nommé par le nom du tampon dont il est l'environnement des définitions, complété par "/":

texte-1/ (notation relative)
/texte/texte-1/ (notation absolue)

la surcharge

L'éditeur peut supporter plusieurs tampons placés dans le même environnement qui sont désignés par le même nom. Ils sont, localement à la session sous l'éditeur, nommés de manière non ambiguë:

texte ou texte<1>
texte<2>
texte<3>
etc...

4.4. recherche sur les noms

L'éditeur propose une aide pour la recherche d'un nom de tampon; cela concerne les commandes qui demandent en paramètre dans la ligne de contrôle un nom de tampon.

commandes de recherche

^M (retour à la ligne) le nom frappé est considéré comme nommé de manière absolu: s'il existe sous ce nom un tampon, il est choisi; sinon, selon la commande, le tampon est créé ou la commande est refusée.

" " (blanc) recherche du plus grand motif sur les noms de tampons qui satisfasse les premiers caractères frappés.

- s'il n'y en a pas, les caractères sont effacés jusqu'au premier motif satisfait;
- s'il y en a un seul, le tampon est désigné;
- s'il y en a plusieurs:
  - soit un motif plus grand est satisfait, et il est affiché,
  - soit c'est exactement le motif qui est satisfait, et une fenêtre d'alarme indique alors l'ambiguité devant être levée.

* ^I (tabulation) identique à " ", mais en cas d'unicité de la solution le nom est simplement complété jusqu'à et y compris le caractère "/" des Buffers-Edit.

Note:
L'ordre des tampons dans un environnement est local à une session sous l'éditeur. De ce fait, la recherche ignore des numéros des tampons. Par exemple, "/texte/" désigne à la fois les Buffers-Edit "/texte<1>/" et "/texte<2>/". En revanche, si aucun texte n'est défini dans texte<2>, "/texte/tex" est non ambigu: la racine "/texte/" fait référence à "/texte<1>/" nécessairement.

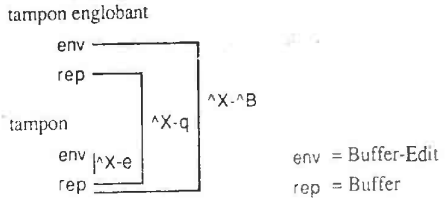
4.5. déplacement entre tampons

Un déplacement entre tampons revient:
- à nommer un nouveau Buffer ou Buffer-Edit;
- à se déplacer vers la fenêtre de ce Buffer ou Buffer-Edit, éventuellement en l'ouvrant.

déplacement absolu

^X-v visiter un tampon: selon que le nom est terminé ou non par "/" il s'agira d'un Buffer ou d'un Buffer-Edit.
^X-^Y visiter le Kill Buffer; pour des raisons de cohérence, la fenêtre ouverte sur le Kill Buffer est fermée dès qu'on la quitte.
^X-^V visiter un fichier: le fichier et le tampon associé sont placés dans l'*environnement des définitions* courant englobant – en particulier, à partir d'un des tampons de la racine, le fichier visité est placé dans l'environnement global.
* ^X-f visiter un «fichier plat»: un fichier plat est un fichier de caractères qui est mis en forme pour être accueilli par l'éditeur.
Note: le tampon d'un fichier plat n'est initialement associé à aucun fichier, et en particulier il n'est pas associé au nom du fichier plat d'origine, pour des raisons de cohérence des fichiers; le nom du tampon, à la manière d'*emacs*, est construit à partir du nom du fichier.

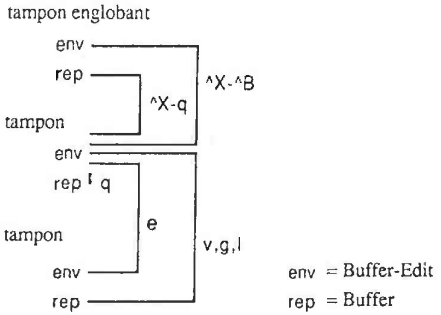
déplacement depuis un Buffer



* ^X-e visiter le Buffer-Edit associé au même tampon.
^X-^B visiter le Buffer-Edit du tampon englobant: on va y retrouver entre autres le tampon dont on est parti.

* ^X-q visiter le Buffer du tampon englobant; si le tampon de départ est l'environnement global, la commande est refusée, puisque l'environnement global n'a pas de représentation.

déplacement depuis un Buffer-Edit



(le tampon désigné est le tampon devant lequel est placé le curseur dans le Buffer-Edit)

^X-^B identique au cas du Buffer.
* ^X-q identique au cas du Buffer.
* q visiter le Buffer associé au même tampon.
* e visiter le Buffer-Edit du tampon désigné.
v visiter le Buffer du tampon désigné.
g aller dans le Buffer du tampon désigné: les autres fenêtres sont détruites.
* l voir le Buffer du tampon désigné: on reste placé dans le Buffer-Edit de départ.

appel récursif

ESC-r appeler récursivement l'éditeur, dans la fenêtre d'édition: le Buffer local devient le Buffer; l'holophraste sous lequel on voit le Buffer local devient l'holophraste minimal accessible.
* ESC-q quitter l'appel récursif.
Note: ces deux commandes permettraient, à plus long terme, de modifier la représentation du «texte» utilisé *depuis* le «texte» utilisateur: par exemple, on définirait les paramètres effectifs d'utilisation d'un autre «texte» dans le *contexte* textuel et visuel de leur apparition.

Note: ESC-q et ESC-r définissent dans *emacs* les commandes de remplacement interactif ou automatique d'un motif textuel par un autre; elles ne sont pas définies ici.

hiérarchie des tampons

* ^X-h afficher, dans une fenêtre d'alarme, la hiérarchie des tampons depuis un tampon donné.
Par exemple, dans les notations précédentes, il s'affiche:

```
-> "texte"
  -> "texte-1"
  -> "texte-2"
```

5. les fichiers

Comme il a été dit, un tampon est désigné par:
- un nom: c'est le nom du «texte»;
- un nom de fichier: c'est le fichier de sauvegarde du «texte».

Du fait de l'imbrication des tampons, on a plusieurs commandes de sauvegarde sur fichier des tampons.

attribut de modification

Les tampons ont un attribut booléen, vrai quand le tampon a été modifié. Quand il est vrai, une commande d'écrasement du tampon est précédée d'une demande de confirmation – commandes de lecture, commande "quitter", ...

La forme visuelle de l'attribut vrai est:
– dans un Buffer, une étoile "*" après le nom du Buffer sur la barre d'information;
– dans un Buffer-Edit, la lettre "M" devant les tampons modifiés.

Cet attribut est *synthétisé* sur l'«arbre des textes»: une sauvegarde d'un tampon ne met pas nécessairement l'attribut à faux, s'il existe un tampon englobé modifié, non sauvegardé et attaché à un fichier.

commandes de sauvegarde

^X-^S sauvegarder le tampon courant dans le fichier auquel il est associé – la commande est refusée si aucun fichier n'est associé à ce tampon. La commande est équivalente qu'on soit dans le Buffer ou dans le Buffer-Edit d'un tampon donné.

^X-^W sauvegarder le tampon dans le fichier de nom donné: ce dernier devient le nom du fichier auquel est associé le tampon.

* ^X-^R sauvegarder, récursivement, le tampon courant et les tampons englobés quand ils sont attachés à un fichier; excepté le cas où le tampon courant n'est attaché à aucun fichier, cette commande met l'attribut de modification à faux.

^X-^M sauvegarder les tampons modifiés du Buffer-Edit englobant: on sauvegarde entre autres le tampon courant à la manière de la commande ^X-^S.

* ^X-^F écrire dans un «fichier plat» du Buffer courant – c'est-à-dire un fichier de caractères. La sauvegarde ne concernant pas la forme structurée du tampon, elle ne met pas à jour l'attribut de modification.

commande assistée

* ESC-^W commande assistée:
"Write: s w r m f h Help: ?"
s = ^X-^S; w = ^X-^W; r = ^X-^R; m = ^X-^M; f = ^X-^F;
h = identique à ^X-h, mais la sortie est réalisée dans un fichier donné – au lieu d'un affichage à l'écran;
? = aide en ligne.

6. le mode "Défaire"

Il n'y a pas de mode "Défaire" ("Undo"), ce qui est certainement *très* limitatif. Réaliser un mode "Défaire" serait sûrement assez complexe, puisqu'il faudrait conserver une grande quantité d'«information de contexte» qui va bien au-delà des seules coordonnées cartésiennes du point d'insertion courant dans le tampon – cette information suffit pour un éditeur travaillant sur des caractères.

7. commandes du Buffer

On a encore trois commandes relatives spécifiquement aux Buffers, qui sont:
– l'aplatissement,
– la sélection,
– la justification.

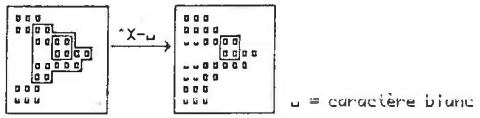
commandes d'aplatissement

* ^X- (contrôle-X et blanc) le curseur courant est *aplati*: la zone étendue qu'il représente devient une suite de curseurs appartenant en propre au Buffer local. La commande est refusée si les caractères qui composent le curseur appartiennent déjà en propre au Buffer local: la situation se présente quand on est placé sur *un* caractère sous un holophraste au moins égal à la profondeur dans le texte structuré de ce caractère – le curseur est vu comme un caractère, et ne peut être aplati.

A plus long terme, la commande permettrait de procéder comme suit:
- on inclut par une utilisation use le texte structuré attaché à un certain tampon;
- on aplatit ce texte, c'est-à-dire qu'on oublie que ce texte appartient à un autre tampon: le lien d'utilisation est brisé.

C'est le classique schéma Copier/Coller qu'on retrouve presque toujours dans un éditeur.

Note: l'aplatissement préserve l'indentation, c'est-à-dire qu'on inclut des caractères blancs si nécessaire.
Par exemple:



commande de sélection

* ^X-^@ début de sélection. On sélectionne une zone du tampon qui s'affiche alors en inverse vidéo. Pour des raisons de cohérence, cette zone reste confinée à un Buffer local donné.

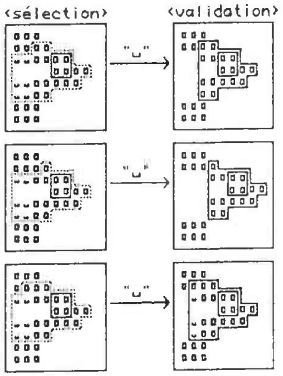
La sélection est un *mode*, avec ses commandes propres. Du fait de la réponse «graphique» très *visible*, l'inconvénient lié à la définition d'un *mode* est mineur – en plus de l'affichage en inverse vidéo on a un message de "Sélection" dans la ligne de contrôle.

déplacement toutes les commandes de déplacement présentées précédemment sont utilisables; le déplacement est seulement limité au Buffer local courant.
^G abandon; on est replacé au point de début de sélection.
q quitter; on reste au point atteint.
? aide en ligne.
" " (blanc) valider.
j valider et entrer en mode de justification.

Valider une sélection, c'est définir la zone sélectionnée comme la zone d'affichage d'un nouveau texte structuré; c'est donc définir un nouveau nœud dans le texte structuré.

A plus long terme, la validation de la sélection demanderait aussi de donner un nom symbolique de «texte» à cette zone et de placer ce «texte» dans l'arbre des tampons: la zone sélectionnée deviendrait la zone d'affichage d'une utilisation use d'un «texte».

Note: la sélection cherche à satisfaire au mieux l'indentation, en supprimant les blancs de début de ligne tout en préservant les alignements.
Par exemple:



commande de justification

* ^X-^J début de justification.

La justification est aussi un *mode*. Elle permet la justification d'un *curseur* par des ajouts ou suppressions de blancs en début de ligne.

^G abandon; on est replacé au point de début de justification.
ESC quitter; on reste au point atteint.
? aide en ligne.
a empiler un niveau de justification, de type *absolu*: on supprime ou on insère le nombre de blancs nécessaire pour obtenir un nombre fixe de blancs (un nombre négatif équivaut à zéro).
r empiler un niveau de justification, de type *relatif*: on supprime un nombre constant de blancs (on insère des blancs si ce nombre est négatif).
q dépiler un niveau de justification; au premier niveau, quitter.

n ligne suivante – pas de modification.
p ligne précédente – pas de modification.

" " (blanc) justifier d'après le type retenu, et passer à la ligne suivante.
c supprimer tous les blancs.
i insérer un blanc.
d détruire un blanc.
, (pause) justifier et rester sur la même ligne.
! justifier automatiquement jusqu'à la fin du curseur -- on quitte le mode interactif.

Les empilements de types de justification permettent de définir une justification globale mais de justifier localement selon d'autres critères.

Par exemple:

a0 = absolu 0 : détruire tous les blancs.
r3 = relatif 3 : détruire 3 blancs.

8. commandes du Buffer-Edit

On regroupe ici les commandes propres au Buffer-Edit – en dehors des commandes de déplacement qui sont identiques à celles d'un Buffer.
Dans un Buffer-Edit, le curseur est placé:
- soit en face du nom d'un tampon: c'est le *tampon désigné*;
- soit ailleurs: il n'y a alors pas de *tampon désigné*.

commandes générales

? aide en ligne
n ligne / tampon désigné suivant.
p ligne / tampon désigné précédent.

commandes de visite

v visiter le Buffer du tampon désigné.
g aller dans le Buffer du tampon désigné.
l voir le Buffer du tampon désigné.
e visiter le Buffer-Edit du tampon désigné.
q quitter pour le Buffer du tampon courant.

commandes de sauvegarde

s sauvegarder le tampon désigné (^X-^S).
w sauvegarder le tampon désigné sous un nom donné (^X-^W).
r sauvegarder récursivement le tampon désigné (^X-^R).
f écrire le «texte plat» du tampon désigné (^X-^F).
h écrire la hiérarchie du tampon désigné dans un fichier de nom donné.

commandes de modification

i insérer un nouveau tampon au point désigné; le nom du tampon doit être donné.
d détruire le tampon désigné.
m déplacer le tampon désigné; la commande peut être utilisée:
- pour renommer le tampon,
- pour le déplacer dans l'arbre des tampons.
t transposer le tampon désigné devant la marque préalablement posée.
Cette commande pourrait être utile à plus long terme, puisque l'évaluation des environnements tient compte de l'*ordre* des termes.

2.9. résumé

(B. = Buffer, B.-E. = Buffer-Edit, K.B. = Kill Buffer)

^@	poser la marque
^A	début de ligne
^B	curseur précédent
^D	détruire le curseur courant
^E	fin de ligne
^F	curseur suivant
^G	abandon (bell)
^H	détruire le curseur précédent
^K	détruire la ligne vers le K.B.
^L	rafraîchir l'écran
^M	insérer un retour à la ligne
^N	ligne suivante
^O	ouvrir la ligne courante
^P	ligne précédente
^R	recherche en arrière
^S	recherche en avant
^T	bascule locale en mode logique / physique
^U	répétition
^V	page suivante
^W	détruire la région vers le K.B.
^X	préfixe
^Y	insérer le K.B.
^Z	déroulement d'une ligne en bas
ESC	préfixe
^^	entrer en mode «ajout»
DEL	détruire le curseur précédent

^X-^@	début de mode sélection
^X-^C	quitter
^X-^D	décrémenter l'holophraste
^X-^E	exécuter la macro
^X-^F	écrire dans un «fichier plat»
^X-^H	fixer l'holophraste logique = physique
^X-^I	insérer le fichier de nom donné
^X-^J	début de mode justification
^X-^K	détruire le B. local vers le K.B.
^X-^L	rafraîchir l'écran
^X-^M	svgder les tampons du B.-E. englobant
^X-^O	lire l'ancien «texte» dans le fichier
^X-^R	sauvegarder récursivement le tampon
^X-^S	sauvegarder le tampon
^X-^T	bascule en mode logique / physique
^X-^U	incrémenter l'holophraste
^X-^W	svgder le tampon dans un fichier nommé
^X-^X	échanger la marque et le curseur
^X-^Y	éditer le K.B.
^X-^Z	réduire la fenêtre
^X-	aplatir le curseur
^X-!	appel de l'interpréteur Lisp
^X-(	début de macro
^X-)	fin de macro
^X-1	la fenêtre courante devient l'unique fenêtre
^X-2	couper la fenêtre courante en deux
^X-=	aller à la ligne de numéro donné
^X-d	détruire la fenêtre courante
^X-e	visiter le B.-E. du tampon courant
^X-f	visiter un «fichier plat»
^X-h	afficher la hiérarchie du tampon
^X-i	insérer le B. de nom donné
^X-k	copier le B. local vers le K.B.
^X-n	aller dans la fenêtre suivante
^X-p	aller dans la fenêtre précédente
^X-q	quitter pour le B. du tampon englobant
^X-r	lire un fichier dans le tampon courant
^X-s	interrompre la session
^X-v	visiter un B. ou un B.-E.
^X-z	agrandir la fenêtre

ESC-^A	insérer le K.B. en fin de B. donné
ESC-^K	cmde assistée copier vers le K.B.
ESC-^N	contraction de ESC-ESC et ^N
ESC-^P	contraction de ESC-ESC et ^P
ESC-^V	cmde assistée visiter
ESC-^W	cmde assistée écrire
ESC-^X	aide en ligne
ESC-^Z	page suivante dans la fenêtre suivante
ESC-ESC	fixer le curseur logique = physique
ESC-!	ligne courante en haut de la fenêtre
ESC-,	curseur en haut de fenêtre
ESC-.	curseur en bas de fenêtre
ESC-<	curseur en début de B.
ESC-=	numéro de ligne
ESC->	curseur en fin de B.
ESC-[	curseur en début de B. local
ESC-]	curseur en fin de B. local
ESC-a	curseur logique en début de curseur
ESC-b	mot précédent
ESC-d	détruire le mot suivant
ESC-e	curseur logique en fin de curseur
ESC-f	mot suivant
ESC-h	détruire le mot précédent
ESC-k	copier la ligne vers le K.B.
ESC-n	*vraie* ligne suivante
ESC-p	*vraie* ligne précédente
ESC-q	quitter l'appel récursif sur le B. local
ESC-r	appel récursif sur le B. local
ESC-v	page précédente
ESC-z	déroulement d'une ligne en haut

Chapitre 9:

Les Aspects d'Implantation

On traite dans ce dernier Chapitre des aspects d'implantation. Cela concerne:
- le *contexte d'évaluation* de la syntaxe concrète;
- l'information qu'il suffit de connaître pour la définition de nouveaux éléments du langage – opérateurs ou phyla;
- un guide de l'implantation de l'*éditeur page*.

Chapitre 9.1:

Contexte d'évaluation

On explique comment est construit le contexte d'évaluation d'un «texte», lors un *emploi* (utilisation ou référence). On expose aussi les propriétés qu'on peut retirer des choix d'évaluation retenus.

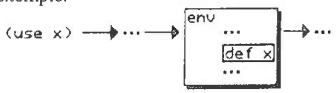
Introduction

On rappelle ici les principales règles de l'évaluation des «textes».

l'utilisation : use

Une utilisation de texte est évaluée en trois temps:

1: Recherche du texte utilisé dans le contexte d'évaluation.
Par exemple:

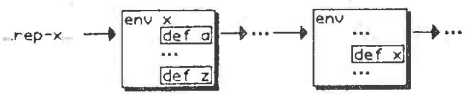


Le texte x est trouvé dans l'environnement, c'est par exemple le texte:

```
def x
   def a
   ...
   def z
= ..rep-x
```

2: Evaluation de l'environnement des définitions locales du texte trouvé – c'est ici l'environnement composé des définitions de textes a, ..., z.

3: Evaluation de la représentation du texte trouvé, dans le contexte formé de l'environnement évalué, placé *en priorité* dans le contexte courant.
Par exemple ici, il reste à évaluer:



la référence : ref

Une référence est évaluée de la même façon qu'une utilisation, mais dans ce cas-là on s'arrête au point **2**: la référence retourne simplement l'environnement évalué du texte trouvé.

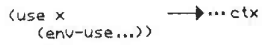
l'emploi : use ou ref

On rappelle qu'un *emploi* de texte est une *utilisation use* ou une *référence ref.*

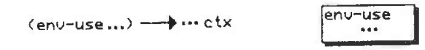
l'environnement local d'emploi

L'environnement local d'emploi est évalué en premier, à l'évaluation d'un emploi de texte. Un emploi de texte est donc évalué en deux étapes:

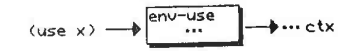
On prend pour exemple l'utilisation:



1: Evaluation de l'environnement local.
Dans l'exemple, on évalue l'environnement local (env-use ...), qui fournit un certain environnement évalué:



2: Evaluation de l'utilisation, sans environnement local d'utilisation, dans le contexte formé de l'environnement évalué, placé *en priorité* dans le contexte courant.
Par exemple ici, on évalue:



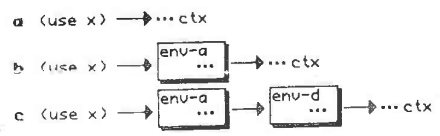
cas particulier

Un cas particulier est celui où l'emploi de texte correspond à ce qu'on appelle une *expression de chemin*: il s'agit d'un emploi de texte pour lequel les environnements locaux d'emploi sont tous réduit à des références simples, s'ils existent.
Par exemple:

```
a (use x)
b (use x (ref a))
c (use x (ref a (ref d))
```

Un certain environnement local d'emploi évalué s'identifie alors à la forme évaluée de l'environnement local des définitions du texte employé.

Dans les trois exemples, on évalue les utilisations du texte x dans les contextes:



1. modularité

Une définition de texte (def) est donnée par :
- un nom, une représentation qui caractérisent le texte,
- un environnement, qui est dans le cas simple une liste d'autres définitions de texte.

Dès le départ on introduit une définition récursive de la définition des textes. Ceci permet assez simplement de construire des textes par emboîtements successifs de modules (les définitions).

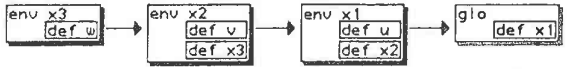
Par exemple on déclare des textes avec trois niveaux d'emboîtement:

```
def x1                                        "U" "V" "W"
   def u = "U"
   def x2
      def v = "V"
      def x3
         def w = "W"
      = (use u) (use v) (use w)
```

On évalue la représentation de x3; on souhaite obtenir le résultat indiqué à droite.
u est alors un concept qui est attaché à x1 "dans sa généralité".
v traduit la même idée, mais pour x2 – x2 étant déjà lui-même un concept attaché à x1 "dans sa généralité".
w se rattache quant à lui à x3, lui-même attaché à x2 attaché à x1.

Cela signifie, par exemple pour u attaché à x1, que dès qu'on "parle" de x1 – c'est-à-dire dès qu'on *utilise* le texte x1, on a la visibilité d'un texte u, défini localement à x1. La représentation de x1, qui utilise un texte nommé u, utilise alors nécessairement le texte u défini localement à x1.

Le contexte d'évaluation de x3 est le suivant:



Il signifie que «plus l'environnement est proche du texte évalué, plus il sera placé prioritairement dans le contexte d'évaluation».

Note: sur le schéma, glo est l'environnement global, c'est-à-dire la liste de toutes les définitions "en tête" qui existent à cet instant – c'est l'«image-mémoire» du système.

2. encapsulation

Nommer les textes à leur définition autorise à s'y référer – plus justement à les employer – uniquement en les nommant, ce qui est donc simple. Le revers de la médaille est la gestion de ces noms: il faut en effet éviter, autant que possible, les problèmes de collision d'identificateurs ou de masquage impromptu de déclaration. La notion de **contexte** permet, dans une bonne mesure, d'éviter ces inconvénients.

Par exemple:

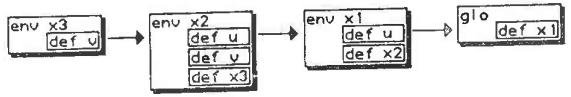
```
def a
= ...
def b
   def a = ...
= ...
```

A l'évaluation de a, on ignore ce qui est défini dans l'environnement de b, il n'y a donc aucun problème de conflit de noms.
A l'évaluation de b, on aura, nécessairement, deux fois la visibilité d'un texte nommé a: on garantit alors que l'environnement propre de b sera placé prioritairement dans le contexte d'évaluation.

Ceci signifie qu'en cas de double définition, on prend la définition qui est "la plus proche" du texte qu'on évalue. Le choix est *a priori* arbitraire, mais reflétera sans doute assez clairement les intentions du programmeur.
Par exemple:

```
def x1                                        "U2" "U3"
   def u = "U1"
   def x2
      def u = "U2"
      def v = "V2"
      def x3
         def v = "V3"
      = (use u) (use v)
```

On peut légitimement penser qu'à l'évaluation de x3 on attend le résultat indiqué à droite.
Le contexte d'évaluation de x3 est:



la recherche s'effectuera alors "de gauche à droite" :
- la déclaration de v dans x3 masque celle qui est présente dans x2,
- la déclaration de u dans x2 celle de x1.

3. paramètre

De ce qui précède, on peut retirer que, partant d'une certaine vision des textes définis, on pourra s'assurer que tel texte est bien celui qui sera *effectivement* utilisé si une définition de ce texte est visible.

Par exemple:

```
def x3
   def v = "V3"
= (use u) (use v)
```

on sait que: (use v) sera évalué en: "V3".
Les autres utilisations, c'est-à-dire les utilisations de texte dont aucune définition n'est visible, seront les paramètres du texte. Ils seront soit définis "plus haut", soit définis "à côté" (ici : le texte u est un paramètre).

Par exemple:

```
def x2
   def u = "U2"
   def v = "V2"
   def x3
      def v = "V3"
   = (use u) (use v)
```

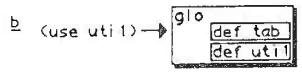
on *sait* que le texte x3 n'est pas paramétré "du point de vue de x2", puisque le texte u, qui est l'unique paramètre du texte x3, est défini localement au texte x2.

exemples de texte paramétré

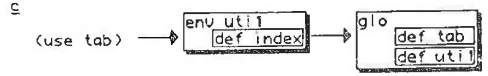
figure A

a
```
def tab = "RGI" (use index) "]"
def uti1
   def index = "["
= (use tab)
```

b



c



d

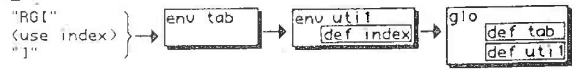


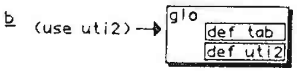
figure A.a: on définit le texte paramétré tab – le paramètre est index.
uti1 définit le texte paramètre index.
figure A.b: évaluation de uti1 dans l'environnement global.

figure A.c: évaluation de la représentation de uti1, l'environnement de uti1 ayant été placé dans le contexte d'évaluation.
figure A.d: évaluation de la représentation de tab: la définition de index de uti1 est bien visible.
Ce comportement est bien celui d'un paramètre formel (l'utilisation dans la représentation de tab d'un texte index) avec passage d'un paramètre effectif (le texte index défini localement au texte uti1).

figure B

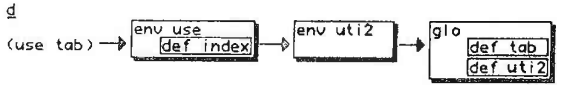
a
```
def tab = "RGI" (use index) "]"
def uti2
= (use tab
     (def index = "]"))
```

b



c

d



e

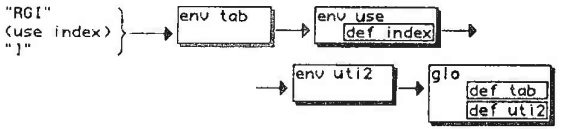


figure B.a: on définit le même texte paramétré tab.
util2 utilise tab en définissant localement le paramètre index.
figure B.b: évaluation de util2.
figure B.c: évaluation de la représentation de util2.
figure B.d: on place l'environnement local de l'utilisation (ici : la définition de index) et on évalue l'utilisation de tab.
figure B.e: évaluation de la représentation de tab: ici encore index joue bien le rôle d'un paramètre.

4. emploi par référence

4.1. exemples

premier exemple
(figure C)

On définit deux textes a et b (figure C.a) :
- Le texte m du texte a utilise un texte x, mais l'"idée" qu'on a en définissant a, x, m est que ce texte x est celui qui est défini dans a.
- Le texte b définit lui aussi un texte x.

Dans sa représentation :
"1:" (use x)
"fait penser" qu'on s'intéresse au texte x de b ("XB").
"2:" (use m (ref a))
"fait penser" que, dans l'utilisation de m, le texte x est celui de la définition de m ("XA").

évaluation de b

figure C.b: c'est le texte évalué qu'on souhaite obtenir.
figure C.c: on évalue l'utilisation de b dans le contexte global d'évaluation glo.
figure C.d: on trouve b dans l'environnement glo; on évalue alors sa représentation, en plaçant *en priorité* l'environnement de définition de b:
"1:" s'évalue en "1:"
(use x) s'évalue en "XB" (qui correspond à la définition de x dans l'environnement b)
"2:" s'évalue en "2:"
Il reste à évaluer: (use m (ref a))
figure C.e: on évalue l'environnement local de l'utilisation: (ref a)
figure C.f: il reste à évaluer l'utilisation du texte m, l'environnement local évalué étant placé *en priorité*.
figure C.g: on évalue enfin la représentation de m dans le contexte où l'environnement de m a été ajouté.
"M:" s'évalue en "M:"
(use x) s'évalue en "XA" (qui est la définition la plus proche de x, à savoir celle de l'environnement a).

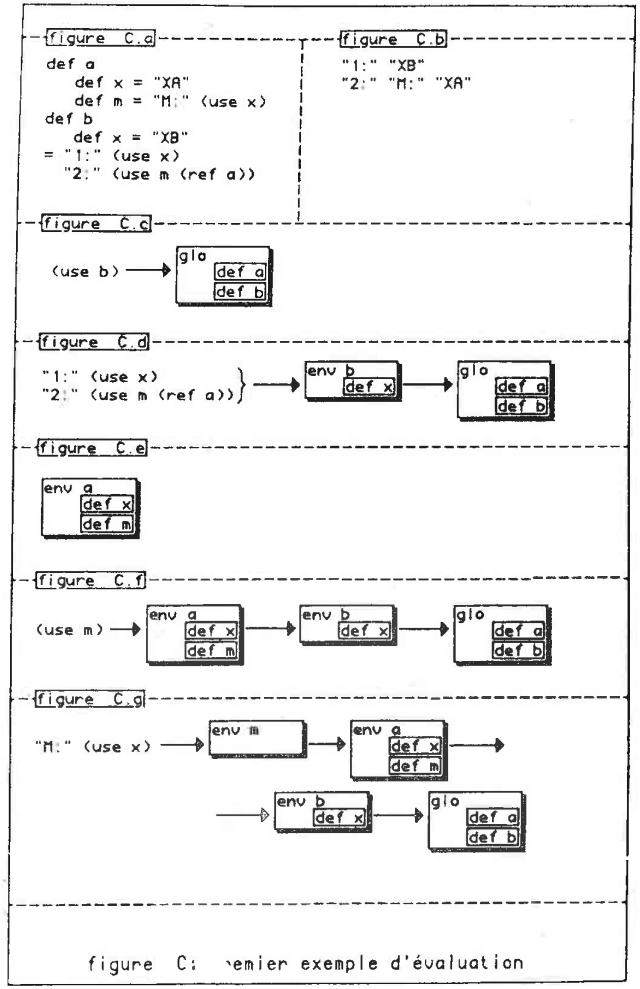


figure C: premier exemple d'évaluation

deuxième exemple
(figure D)

On définit à nouveau deux textes a et b (figure D.a).

- Le texte m du texte a est toujours défini à l'aide d'une définition locale à a d'un texte x. Il est en plus *paramétré* par un texte y.
- Le texte b, à l'utilisation du texte a, définit ce paramètre y.

évaluation de b

figure D.b: c'est le texte évalué qu'on souhaite obtenir.
figure D.c: on évalue b dans l'environnement global glo.
figure D.d: le début étant similaire au cas précédent, on s'intéresse tout de suite à l'utilisation du texte m:
figure D.e: on évalue l'environnement local de l'utilisation:
(ref a) : fournit l'environnement de a;
(def y = "B") : s'évalue tel quel.
On obtient ainsi l'environnement local évalué de l'utilisation.
figure D.f: on évalue ensuite la représentation de m.
(use x) s'évalue en: "XA"
(use y) s'évalue en: "YB"

troisième exemple
(figure E)

On définit encore deux textes a et b (figure E.a).
Le cas est cependant plus complexe que précédemment. Ici, on introduit en effet *deux niveaux* d'emboîtement des environnements. Si on conserve, comme on a fait jusqu'à présent, uniquement le dernier environnement qui a permis de trouver le texte, alors on risque de perdre certaines informations qu'on croyait en principe attachées au texte utilisé.

évaluation de b

figure E.b: c'est le texte qu'on souhaite obtenir.
figure E.c: on évalue b dans l'environnement global.
figure E.d: on en arrive tout de suite à l'évaluation de l'utilisation.
figure E.e.1: on évalue l'environnement local : (ref c (ref a)) :
on évalue d'abord l'environnement local de celui-ci :
(ref a) qui fournit l'environnement de a;
figure E.e.2: on évalue ensuite la référence au texte c, quand on a placé l'environnement local évalué;
figure E.e.3: on retourne le **contexte** qui est attaché à la référence:
(ref c (ref a)) ;
figure E.e.4: et finalement l'environnement local évalué.
figure E.f: on évalue enfin la représentation de m.
(use x) s'évalue : "XA" puisque
- on recherche x dans <env m>: il n'y est pas;
- on le recherche alors dans <env use>:
  - on le recherche dans <env c>: il n'y est pas;
  - on le recherche dans <env a>: on l'y trouve.

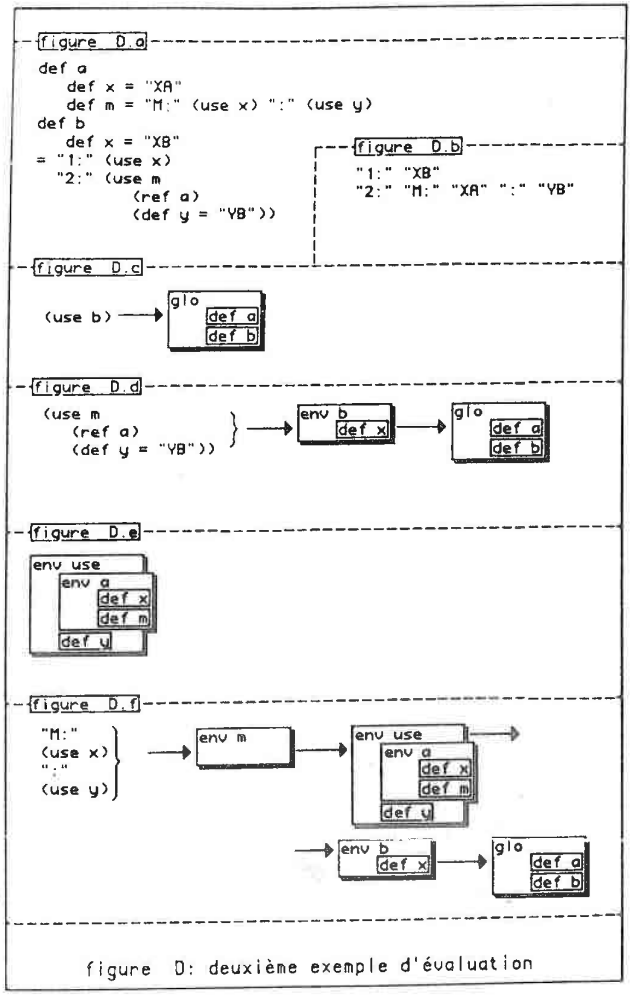


figure D: deuxième exemple d'évaluation

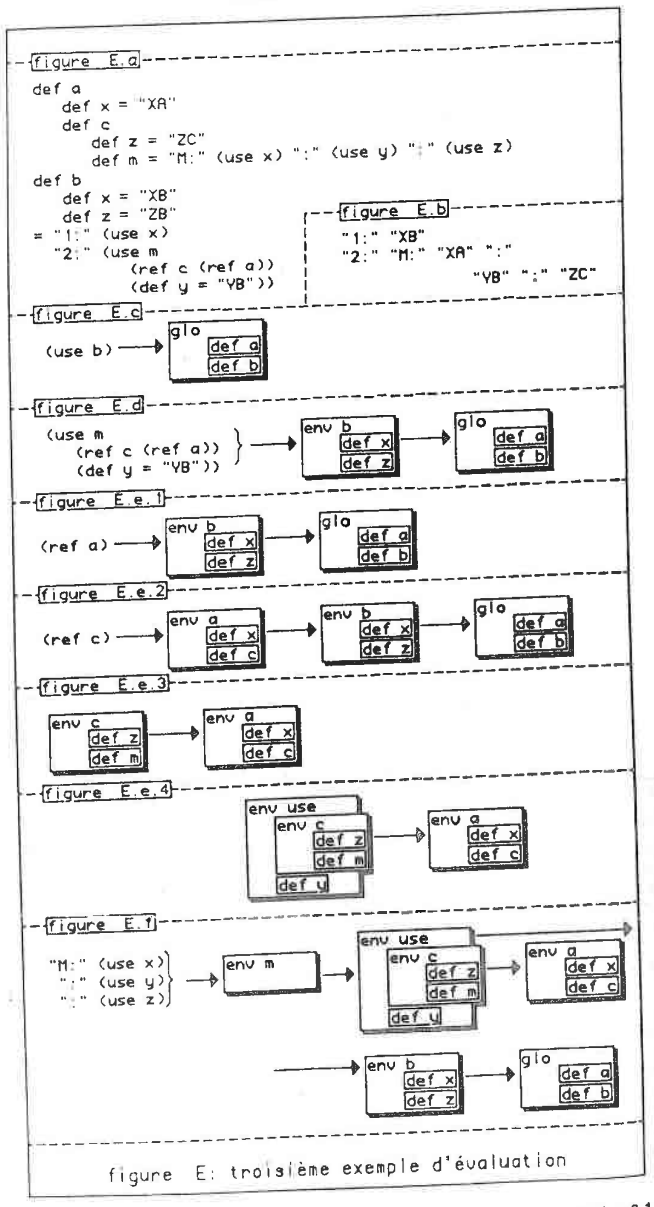


figure E: troisième exemple d'évaluation

Remarque:
Ce mécanisme garantit qu'on ne perdra pas des informations qui seraient liées au *contexte de définition* du texte qu'on utilise.

En effet, dans l'exemple :
- m est défini dans c qui est défini dans a:
  Il est donc légitime d'utiliser dans les définitions de m des textes définis dans a ou c.
- A l'inverse, tout utilisateur "extérieur" du texte m doit suivre le "chemin":
  (use m (ref c (ref a)))
  pour un simple problème de visibilité des identificateurs.

Par là même on garantit de conserver, dans l'utilisation de m, les définitions de a et c.

Quant à l'environnement propre à chaque texte, il est toujours ***le plus prioritaire*** puisqu'il est placé en premier à l'évaluation du texte : les définitions propres au texte seront toujours celles prises en priorité à l'évaluation de ce texte.

4.2. contre-exemple

Puisqu'on ne conserve pas de l'évaluation de chaque texte le ***contexte complet*** dans lequel on l'a trouvé, on a bien sûr de nombreuses situations où apparaissent des «effets de bord» imprévisibles.

Par exemple, on définit les deux textes a et b:

```
def a
   def x
   def c
      def p = (use x)
def b
   def d
      def x
      def m = (use x) (use p)
```

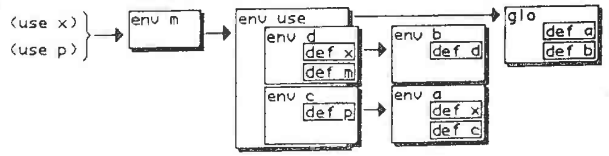
- dans le texte a: le texte p utilise x qui est en principe celui défini dans a;
- dans le texte b: le texte m utilise x qui est celui de d et utilise un texte p qui est ici un paramètre.

On peut alors vouloir écrire l'utilisation:

```
(use m
   (ref d (ref b))
   (ref c (ref a)))
```

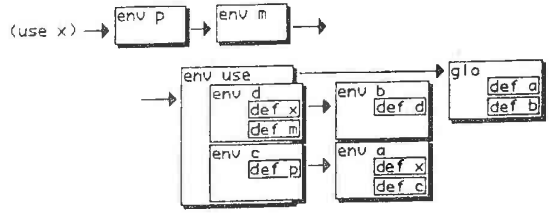
- le premier chemin : (ref d (ref b)) permet d'accéder à la définition de m;
- le deuxième chemin : (ref c (ref a)) permet de fournir un définition de p (puisqu'on sait que le texte m est paramétré par un texte p).

On évalue donc (...) la représentation de m:



- (use x) fait référence à la définition de x dans d, ce qui est le résultat souhaité;

- (use p) est trouvé dans c: on évalue alors sa représentation:



(use x) fera alors référence à la définition de x dans d *et non* dans a.

On peut remarquer que l'exemple est déjà un cas assez particulier :
- on a choisi le même identificateur x pour deux textes qui ne semblent avoir aucuns liens sémantiques;
- on utilise m le long de son chemin de définition : (use m (ref d (ref b))), ce qu'on appelle dans la suite une *expression de chemin*, tout en *complétant* ce chemin par d'autres définitions : (ref c (ref a))
- le texte p possède justement x comme pseudo-paramètre: de là vient le conflit.

4.3. contre-contre-exemples

premier contre-contre-exemple

On pourrait décider, quand un texte est trouvé, de conserver tout le contexte dans lequel il a été trouvé:
Dans l'exemple précédent, la recherche du texte p aurait retourné le *contexte complet*:

Mais on introduirait alors d'autres effets de bord.

Par exemple (figure F):
figure F.a: on définit les textes k et X:
- le texte a est paramétré par k,
- le texte b fournit explicitement le paramètre k ("K1").
C'est pourtant la définition de k "la plus externe" ("K0") qui sera prise.
figure F.b: en effet, on évalue b dans l'environnement global.
figure F.c: on évalue la représentation de b.
figure F.d: on évalue l'environnement local de l'utilisation.
figure F.e: on indique ici le contexte dans lequel on trouve le texte a.
figure F.f: on évalue alors la représentation de a dans ce contexte.
(use k) s'évalue alors en: "K0" et non en: "K1".

L'inconvénient majeur que l'on peut voir ici est qu'on ne peut plus avoir de vision locale des textes. Une vision locale de l'exemple est la suivante:

```
def a = (use k)
def b = (use a
           (def k = "K1"))
```

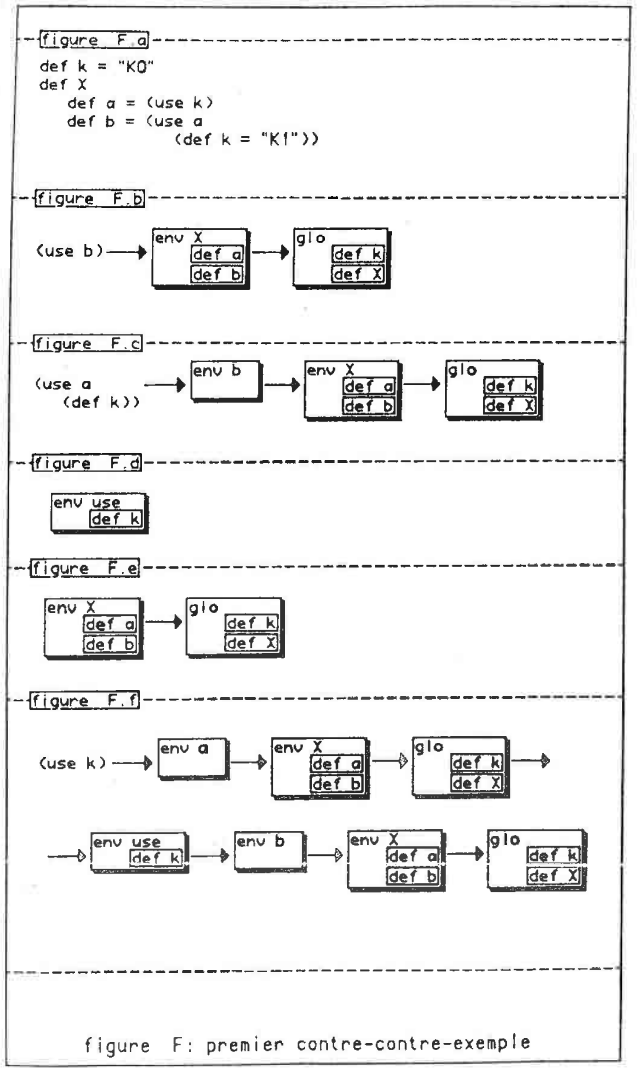



figure F: premier contre-contre-exemple

Les déclarations regroupées des textes a et b admettent une interprétation simple: k est un paramètre de a instancié par b, ce qui se révèle être faux quand on sait que dans les environnements supérieurs on a déjà défini un texte k.

deuxième contre-contre-exemple

On peut souhaiter effectivement réaliser un effet de bord. Par exemple:

```
def a
   def x = "X0"
   def m = (use x)
def c = (use m
           (def x = "X1")
           (ref a))
```

- x est un pseudo-paramètre de m (c'est bien un paramètre de m, mais il est défini dans le même environnement que m: "du point de vue de a", m n'est pas paramétré);
- dans c, on utilise le texte m, le long du chemin : (ref a) , en modifiant explicitement le pseudo-paramètre x, qui a donc alors le comportement d'un paramètre.

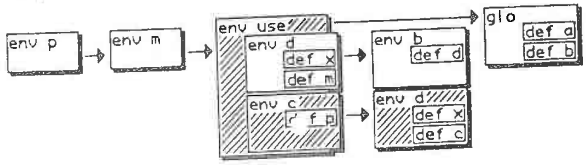
On peut remarquer ici encore que le cas présenté est assez particulier :
- on utilise le texte m, pour lequel le paramètre x est normalement instancié, mais qu'on redéfinit spécialement pour l'utilisation;
- on n'utilise pas simplement le texte, par le chemin : (use m (ref a))
  mais on ajoute une autre définition locale à cette utilisation: l'effet de bord observé est donc ici encore détectable.

Si on garantit l'ordre de recherche – indispensable dans l'exemple – on peut alors masquer des déclarations par des redéfinitions. Dans tous les cas, puisque l'on définit un environnement local qui n'est pas une simple expression de chemin, on *sait* qu'on se place dans une situation où pourront se produire des effets de bord non voulus.

remarque

Il serait toujours loisible d'assurer un contrôle de visibilité des identificateurs: il suffit en effet de vérifier qu'à un *même niveau* on n'a pas deux définitions du même identificateur. On pourrait donc prévoir l'envoi d'un message de **mise en garde**, qui permettrait à l'utilisateur d'avoir connaissance des zones sensibles.

Par exemple, dans le (contre-) exemple précédent, sachant qu'on a trouvé la définition de x dans d, on n'a à rechercher une éventuelle autre définition de x que dans la zone hachurée:



- dans la partie en-deçà: on a déjà cherché et on n'a pas trouvé de définition pour x;
- dans la partie au-delà: on est dans le cas d'application de la règle classique de masquage des identificateurs et une éventuelle autre définition de x ne saurait être vue comme une possibilité d'effet de bord imprévisible.

5. exemples d'application

5.1. exemples d'école

On présente divers exemples d'utilisation du texte c à l'intérieur du texte t.

```
def t                          1
   def c = (use m)             (use c) ≡ (use m)
   def x                       2
      def m = "M"              (use c
                                  (ref x)) ≡ "M"
```

(1) utilisation de c: m est un paramètre.
(2) utilisation de c, "du point de vue" de x: le paramètre m est instancié.

```
def t                          1
   def m = "X"                 (use c) ≡ "X"
   def c = (use m)             2
   def x                       (use c
      def m = "M"                 (ref x)) ≡ "M"
```

(1) utilisation de c: m est instancié par la définition apparaissant dans t.
(2) cf premier: la définition de m dans le texte x masque celle qui apparaît dans le texte t.

```
def t                          1
   def c = (use m)             (use c) ≡ (use m)
   def x                       2
      def m = "M"              (use c
      def c = "C"                 (ref x)) ≡ "C"
```

(1) cf premier.
(2) utilisation de c, "du point de vue" de x: la définition de c dans le texte x masque celle qui apparaît dans t.

5.2. "démasquage" des textes

La notion d'environnement de définition des textes permet de déclarer de nouveaux textes, sans pour autant perturber l'évaluation d'autres textes utilisant des textes de même nom. A la déclaration d'un texte, les textes présents dans l'environnement des définitions sont masqués :
- ils sont donc inutilisables pour les textes extérieurs au texte de définition,
- on évite ainsi tout risque de collision entre identificateurs.

Le concept de référence à un texte permet justement de lever cette barrière de visibilité, mais la mention est alors *explicite*.

On peut tirer de ce fait divers avantages :

- Elargir l'emploi de concepts identifiés dans un texte pour un usage plus général – les définitions internes à un texte peuvent être exportées vers un autre texte.

- Elargir la notion de définition de texte. Par exemple:

```
def varglobal
   def varA = "A"
   def varB = "B"
   def varC = "C"
```

varglobal est un texte pour lequel on n'a pas particulièrement défini de représentation: ce qui nous intéresse ici n'est pas d'*utiliser* le texte mais de s'y *référer* (pas use mais ref).
On peut ainsi réaliser des *définitions connexes* de textes, même si aucun texte particulier n'a l'usage de la totalité des définitions qu'on a regroupées.

- Structurer des déclarations de textes «en profondeur», c'est-à- dire par emboîtement de textes, tout en permettant l'accès aux définitions internes.
Par exemple:

```
(use varA (ref varglobal (ref modulecommun)))
```

on nomme le texte varA du texte varglobal du texte modulecommun, ce dernier regroupant par exemple toutes les déclarations globales, dont le texte varglobal.
On peut ainsi assez précisément nommer un texte donné, et s'affranchir au mieux des problèmes d'évaluation erronée de texte due à la présence de plusieurs textes de même nom.

5.3. paramètre

La recherche d'un texte référencé étant une fois encore réalisée dans un contexte construit dynamiquement, on peut laisser certaines références indéterminées, qui jouent alors le rôle de paramètres formels de l'environnement de définition du texte.

Par exemple:

```
def declVAR
   (ref defVAR)
   def decla = "VAR " (use nom) ":" (use type) ";"
   def util  = (use nom)
def txt
   def defVAR
      def nom = "A"
      def type = "integer"
   def varA
      (ref declVAR)
= ...
```

declVAR est paramétré par le texte defVAR, duquel on attend qu'il définisse les textes nom et type. Il offre deux "propriétés" (il définit deux textes) :
- decla : déclaration de la variable,
- util : utilisation de la variable.

txt définit :
- le texte defVAR pour instancier le paramètre formel de declVAR,
- le texte varA, qui par référence au texte declVAR définit une variable de nom: "A" de type: "integer" .

Le texte txt dans sa représentation utilisera alors le texte varA au travers des propriétés définies dans declVAR:

```
(use decla (ref varA))
(use util  (ref varA))
```


6. héritage de propriétés

6.1. héritage commun

L'exemple précédent laisse déjà entendre que les concepts présentés s'apparentent assez bien à la notion d'**héritage** des Langages Orientés Objet. On peut, du fait que la notion n'est pas explicite mais plutôt induite, tirer parti de la situation pour réaliser diverses sortes d'héritages.

Par exemple:

```
def com
   def a = "A[" (index) "]"
   def b = "B[" (index) "]"
def util
   def util1
      def index = "i"
   = (a(com)) ":=" (b(com)) ";" "^M"
   def util2
      def index = "i+1"
   = (a(com)) ":=" (b(com)) ";" "^M"
= (util1)
  "TRAIT;" "^M"
  (util2)
```

```
A[i]:=B[i];
TRAIT;
A[i+1]:=B[i+1];
```

la définition du paramètre effectif index est mise en commun entre les textes a et b: à une seule définition du paramètre correspond deux instanciations.
On peut noter que l'exemple profite directement du fait que les textes a et b sont paramétrés par un texte de *même nom* index. Cependant les paramètres traduisant dans les deux cas la même notion, on peut juger raisonnable de leur donner le même nom.

6.2. héritage multiple

Réaliser un texte par héritage multiple est simple, si l'on s'en tient à la terminologie employée précédemment : il suffit qu'un texte fasse référence à *deux* autres textes pour qu'on soit dans ce cas-là.
Ici encore, le problème induit par la présence de deux textes de même nom importés par deux références à des textes n'est pas traité: il serait nécessaire de définir, avec une référence à un texte, une syntaxe «**rename**» qui permette de gérer ce type de situation – en renommant "à la main" les textes qui auraient le même nom. Le choix retenu ici est de limiter au maximum le nombre de concepts élémentaires introduits: on ne s'encombre donc pas de complications syntaxiques visant à contourner le problème des conflits d'identificateurs. Ce choix est peut-être sujet à critique, en tous les cas il permet de conserver une simplicité et une homogénéité des concepts élémentaires (def, ref, use, stg).

6.3. structuration de l'application

Avec ce qui précède, on a maintenant suffisamment d'éléments pour structurer son application – ou plus exactement le texte source de son application – selon une hiérarchie de textes emboîtés.

On peut immédiatement se demander :
- comment construire cette hiérarchie ?
- pourquoi ne pas utiliser un Langage Orienté Objet, ou tout du moins un langage de «haut niveau»?

Je n'ai pas de réponse précise à la première question, si ce n'est: employer une Méthode Orientée Objet (MOO) ce qui revient à reporter la question vers la suivante: comment se définit une MOO?
A ma connaissance, il n'y a pas de réponse formelle, mais plutôt diverses philosophies de programmation dont le point de convergence est justement la construction de programmes orientée par les objets. C'est de l'une de ces philosophies qu'il faudrait s'inspirer pour construire sa hiérarchie.

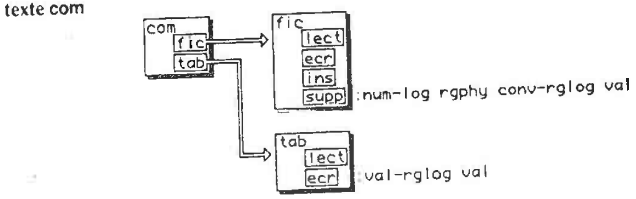
Quant à la seconde question, je dirai:
- qu'il n'est pas toujours possible d'employer un langage de haut niveau – pour des raisons de personnel, de matériel, de contexte historique informatique, de crédits, de politique, ...
- qu'un langage de haut niveau n'est pas la panacée, mais une proposition d'harmonisation de concepts essaimés dans d'autres langages, et qu'il est donc lui-même sujet à limitations.

On présente un exemple, un peu simple, de situation où même l'emploi d'un langage de «haut niveau» ne permet pas d'éviter la dispersion de notions logiquement liées.

schéma page suivante

6.4. exemple de structure hiérarchique

On présente ici, très partiellement, un exemple un peu plus réaliste, qui a été traité plus en détail auparavant (cf. Chapitre 2.3, *«Exemple de structuration des données»*).



Texte commun, qui définit la représentation physique des données:
fic = fichier, tab = tableau.

exemple :

On s'intéresse à une boucle :
- initialement, on a : X = A(I)
- dans la boucle : on lit une nouvelle valeur de X
- le test de fin de boucle est : X = A(I+1)

1: le langage impose l'emploi d'une variable comme indice de tableau :
```
X:=A(I);
I:=I+1;
WHILE X<>A(I) DO Lire(X);
```

2: le langage, plus évolué, reconnaît un objet du type A(I+1) :
```
X:=A(I);
WHILE X<>A(I+1) DO Lire(X);
```

3: A n'est pas un tableau mais une fonction : on utilise une variable auxiliaire Y pour optimiser la recherche
```
X:=A(I);
Y:=A(I+1);
WHILE X<>Y DO Lire(X);
```

4: le compilateur associé au langage possède un bon optimiseur : le bon cas
```
X:=A(I);
WHILE X<>A(I+1) DO Lire(X);
```

5: dans le bon cas : le paramètre de la fonction A est passé par adresse :
```
X:=A(I);
I:=I+1;
WHILE X<>A(I) DO Lire(X);
```

6: dans le bon cas : en plus, la fonction A effectue un effet de bord (par exemple : un affichage à l'écran)
```
X:=A(I);
I:=I+1;
Y:=A(I);
WHILE X<>Y DO Lire(X);
```

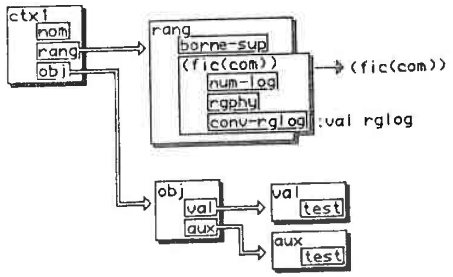
Le cas simple est bien le cas 2. Cependant, diverses contraintes de contexte peuvent conduire à se placer dans le cas 6, et ceci même dans le meilleur des cas — langage évolué, optimiseur puissant.

figure 27 : exemple de limitations des langages

cas fic
- *paramètres*
  - num-log = numéro logique
  - rgphy = rang physique
  - conv-rglog = conversion rang logique / rang physique
  - val = valeur lue / écrite sur le fichier
- *propriétés*
  - (lect (fic)) = lecture sur fichier
  - (ecr (fic)) = écriture sur fichier
  - (ins (fic)) = insertion dans le fichier
  - (supp (fic)) = suppression dans le fichier

cas tab
- *paramètres*
  - val-rglog = représentation du tableau
  - val = valeur d'un élément du tableau
- *propriétés*
  - (lect (tab)) = lecture dans le tableau
  - (ecr (tab)) = écriture dans le tableau

texte ctx1



Texte de définition d'un contexte de travail ctx1:
rang = rang d'accès aux données, obj = les données.

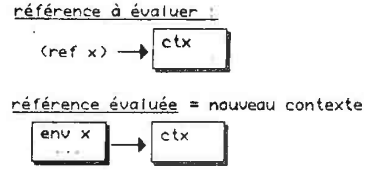
paramètres
- val = valeur lue / écrite
- rglog = rang logique

propriétés de ctx1
- (nom) = nom du contexte
- (borne-sup (rang)) = valeur maximale des rangs logiques
- (lect (rang)) = (lect (fic (com)))
- (ecr (rang)) = (ecr (fic (com)))
- (ins (rang)) = (ins (fic (com)))
- (supp (rang)) = (supp (fic (com)))
- (val (obj)) = donnée
- (test (val (obj))) = champ test de la donnée
- (aux (obj)) = donnée auxiliaire
- (test (aux (obj))) = champ test de la donnée auxiliaire

On a d'autres propriétés: par exemple (num-log (rang)). Il s'agit en fait de "fausses propriétés", puisqu'elles ne sont déclarées que pour instancier les paramètres de (fic (com)). Cependant, aucun contrôle n'est effectué, on peut donc tout aussi bien les utiliser.

7. polymorphisme

Un sujet qui n'a pas encore été abordé concerne le contexte dans lequel on évalue une référence à un texte. La réponse n'est pas forcément simple. En effet, une référence évaluée enrichit le contexte d'évaluation : faut-il alors évaluer cette référence dans le contexte enrichi, ou faut-il évaluer la référence dans le contexte initial puis enrichir le contexte?



7.1. exemples d'évaluation

On présente un exemple d'évaluation non simple:

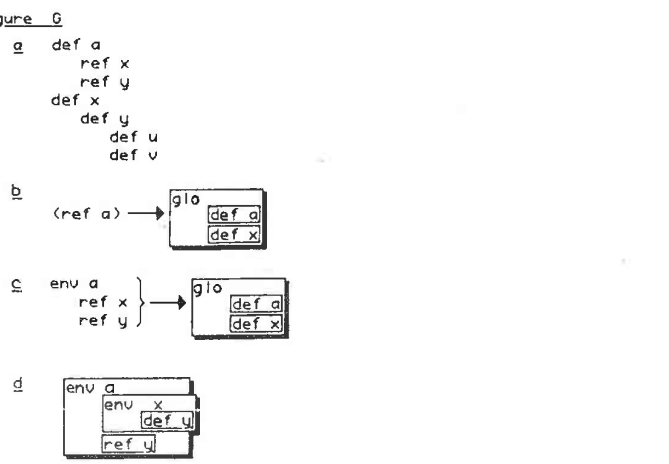
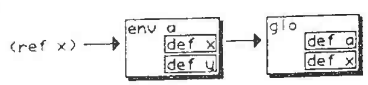


figure G.b: on évalue l'environnement de a.
figure G.c: on trouve le texte a dans glo, on évalue alors son environnement.
figure G.d: c'est l'environnement évalué qui sera retourné.
(ref x) : on a trouvé le texte x dans glo, la référence a donc été remplacée par son environnement.
(ref y) : on ne trouve pas y, et la référence reste indéfinie.

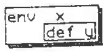
Or l'exemple pourrait laisser entendre que la référence à y est celle qui est importée par la référence évaluée de x. On évalue alors l'environnement de a en considérant qu'on remplace les références trouvées par leur valeur:

figure H

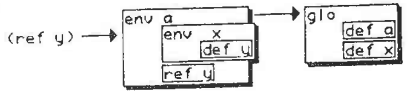
a



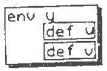
donne



b



donne



c

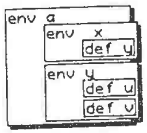


figure H.a: on évalue: (ref x) dans le contexte total – dont l'environnement à évaluer a.

figure H.b: on évalue: (ref y) dans le contexte total partiellement modifié par l'évaluation précédente.

figure H.c: c'est le nouvel environnement évalué de a.

On peut évidemment avoir des situations beaucoup plus complexes.

Par exemple:

```
def a          def x          def y
   ref x          ref t          def t
   ref y          def u             def z
   ref z                         def v
```

• 1: la référence à x donne l'environnement de x:

```
ref t
def u
```


• 2: la référence à t n'est pas trouvée; en revanche, la référence à y donne:

```
def t
   def z
def v
```

• 3: l'évaluation de y permet de trouver t; ceci donne:

```
def z
```

• 4: d'où l'on trouve la référence à z.

L'évaluateur doit donc subtilement interrompre une partie de l'évaluation d'un environnement pour la reprendre ultérieurement.

Divers problèmes sont sous-jacents à cette remarque :
- il devient très difficile de détecter les cas de bouclage : sachant qu'on n'abandonne pas l'évaluation d'une référence si on ne trouve pas de texte de même nom, mais qu'on la reporte simplement à plus tard, on ne peut pas facilement dire à quel moment on s'arrêtera;
- la solution pourra ne pas être unique.
  Par exemple:

```
def a          def x          def y
   ref x          def y          def x
   ref y
```

on a deux valeurs possibles à l'évaluation de l'environnement de a.

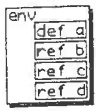
7.2. solution retenue

La solution retenue consiste à limiter le nombre de cas acceptables, *en garantissant l'ordre d'évaluation.*

On évalue donc un environnement :
- en respectant l'ordre des déclarations qui y figurent,
- et pour chaque déclaration: en enrichissant le contexte d'évaluation par un environnement provisoire, composé:
  - de la forme évaluée des déclarations précédant celle qu'on évalue dans l'environnement,
  - de la forme non évaluée de celles qui la suivent.

On a par exemple les étapes suivantes d'évaluation progressive d'un environnement:

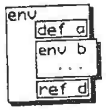
environnement à évaluer :



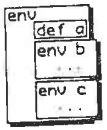
def a
s'évalue dans l'environnement :

ref b
s'évalue dans l'environnement :

ref c
s'évalue dans l'environnement :



ref d
s'évalue dans l'environnement :



7.3. liens avec l'environnement local

Il a été dit au début que l'environnement local permettait de faciliter l'écriture d'emplois de texte, en particulier de références. Le choix retenu pour l'évaluation des références permet de considérer effectivement, *dans une bonne mesure*, la notion d'environnement local comme une facilité purement syntaxique, et non pas une nouvelle notion.

Dans les exemples qui suivent, on donne:
- à gauche: la définition du texte t,
- à droite: le contexte dans lequel est évaluée l'utilisation du texte a, qui apparaît dans la représentation du texte t, quand on a déjà évalué l'environnement local d'utilisation, s'il y a lieu.

Premier exemple:

```
def t                    (use a) ──→ env-use ──→ env t
   def x = "X"                                     def x
= (use a)
```

```
def t                    (use a) ──→ env-use ──→ env t
= (use a                             def x
     (def x = "X"))
```

l'évaluation des représentations des deux textes t retourne la même valeur.

Deuxième exemple:

```
def t                    (use a) ──→ env-use ──→ env t
   ref u                                         env u
= (use a)                                        ...
```

```
def t                    (use a) ──→ env-use ──→ env t
= (use a                             env u
     (ref u))                        ...
```

ici aussi les représentations s'évaluent de la même façon.

Troisième exemple:

```
def t                    (use a) ──→ env-use ──→ env t
   ref u                                         env u
   ref v                                         ...
= (use a)                                        env v
                                                 ...
```

```
def t                    (use a) ──→ env-use ──→ env t
   ref v                             env u       env v
= (use a                             ...         ...
     (ref u))
```

```
def t                    (use a) ──→ env-use ──→ env t
   ref u                             env v       env u
= (use a                             ...         ...
     (ref v))
```

Ici les trois cas sont légèrement différents:
- Dans le premier cas, on évalue, dans l'ordre:
  - l'environnement u,
  - l'environnement v, dans le contexte enrichi de l'environnement u,
  - puis l'utilisation de a dans un contexte qui place dans l'ordre l'environnement u puis l'environnement v.
- Dans le deuxième cas, on évalue, dans l'ordre:
  - l'environnement v,
  - l'environnement u, dans le contexte enrichi de l'environnement v,
  - puis l'utilisation de a dans un contexte qui place dans le même ordre les environnements u et v.
- Dans le troisième cas enfin, on évalue, dans l'ordre:
  - l'environnement u,
  - l'environnement v, dans le contexte enrichi de l'environnement u,
  - puis l'utilisation de a dans un contexte qui place les environnements u et v dans l'ordre inverse.

Les résultats de ces évaluations pourront donc être différents.

7.4. polymorphisme

Le fait que l'environnement s'évalue en partie sur lui-même permet d'observer un comportement qui s'apparente assez bien au polymorphisme – l'usage d'un traitement sur des objets différents conduit à utiliser des définitions de ce traitement différentes. En effet on pourra dans un environnement utiliser une première référence pour caractériser l'objet, et une deuxième pour se référer au traitement, traitement qui sera alors celui défini pour l'objet référencé.

Par exemple:

```
def declVARentiere
   def decla = "VAR" (nom) ":integer;"
   def incr  = (nom) ":=" (nom) "+1;"

def declVARarbre
   def decla = "VAR" (nom) ":arbre;"
   def incr  = (nom) ":=" (nom) "@." (chppteur) ";"
```

declVARentiere: sert à déclarer une variable entière, et la propriété incr.
declVARarbre: sert à déclarer une variable arbre (liste chaînée), et la propriété incr.

```
def varE
   def nom = "E"
   (ref declVARentiere)

def varA
   def nom = "A"
   def chppteur = "suiv"
   (ref declVARarbre)
```

varE: déclare une variable entière de nom "E", de type "integer".
varA: déclare une variable arbre de nom "A", de type "arbre", dont le champ pointeur s'appelle "suiv".

```
def txt
= (use incr (ref var))
```

txt: utilise la propriété incr de l'objet var qui joue ici le rôle d'un paramètre. C'est à ce niveau-ci qu'on observe un comportement polymorphe de l'utilisation du texte incr.

```
def txtE                          donne    E:=E+1;
= (use txt
     (def var
        (ref varE)))

def txtA                          donne    A:=A@.suiv;
= (use txt
     (def var
        (ref varA)))
```

txtE: utilise txt en définissant var comme étant varE.
txtA: utilise txt en définissant var comme étant varA.

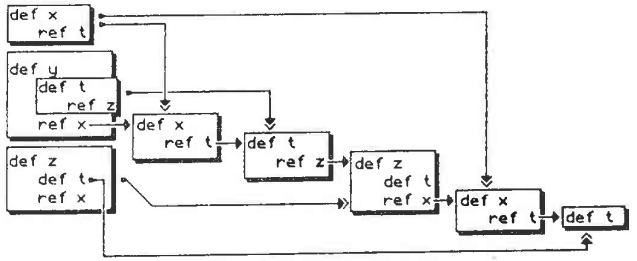
7.5. bouclage de l'évaluation

Puisque l'environnement est en partie évalué sur lui-même, on peut observer, assez facilement, un bouclage infini de l'évaluateur de texte. Par exemple:

```
def a          def b
   def x          def y
   ref b          ref a
```

l'évaluation de a ou b bouclera nécessairement. On peut regretter un peu la chose, puisque l'exemple présenté admet une interprétation "intuitive" simple.

On pourrait alors prévoir, en cours d'évaluation, de vérifier qu'on n'évalue pas deux fois le même texte: dans un tel cas, on détecte alors un bouclage possible et on interrompt l'évaluation. Ceci est en fait trop restrictif, comme le montre l'exemple qui suit:



On évalue l'environnement de y. Le texte x défini en premier est évalué *deux fois*, et pourtant le processus d'évaluation s'arrête.

On pourrait donc ici encore prévoir l'envoi d'un message de mise en garde qui permettrait à l'utilisateur d'interrompre l'évaluation s'il le souhaite.

8. manipulation symbolique

La première étape consiste à faire de la manipulation symbolique de textes. Au lieu d'utiliser des chaînes de caractères, on définit des textes, qu'on utilise ensuite symboliquement en les nommant au lieu de leur représentation.
Tout de suite on introduit la notion de module: on ne peut guère considérer toutes les définitions de textes placées au même niveau, où tout le monde verrait tout le monde; on comprend qu'on se heurterait alors rapidement à de franches difficultés pour retrouver une définition de texte, pour inventer un nouveau nom de texte, ...

La deuxième étape consiste à faire de la manipulation symbolique de symboles. Ceci tient à diverses raisons:
- le problème des zones communes (par exemple une déclaration de variable commune à deux textes);
- le souhait d'une construction progressive, du plus général vers le plus spécifique; par exemple:

```
def x                          def z
def y = def x complété            def y
def z = def y complété               def x
                                  ...
```

la traduction de la notion de «forme complétée de texte» a pour conséquence de *masquer* alors la visibilité des textes x et y – qui sont des cas généraux qu'on pourrait souhaiter utiliser plusieurs fois.

Un symbole a alors deux valeurs:
– une **valeur d'environnement**,
– une **valeur de représentation**.
La manipulation symbolique de textes s'intéresse à la valeur de représentation.
La manipulation symbolique de symboles s'intéresse à la valeur d'environnement.

On peut alors redouter qu'avec une telle approche, on ne doive ensuite définir la manipulation symbolique des symboles de manipulation symbolique de symboles, et finalement ne jamais s'arrêter vraiment. En fait, la manipulation symbolique des symboles peut se traduire par des symboles de manipulation symbolique des textes: il suffit en effet de pouvoir donner à ces symboles une valeur d'environnement et une valeur de représentation. On s'arrêtera donc finalement à cette deuxième étape.

Note:
On peut rapprocher du propos précédent le type d'interprétation qu'on donne aux environnements locaux des utilisations:

```
(use a
     (def x ...)
     (def y ...))
```

on définit un *nouveau symbole* de manipulation des symboles: $, qui est tel que:
- l'environnement de $ est composé des textes x et y,
- la représentation de $ est, par défaut: (use a).
La représentation est alors évaluée dans le contexte de définition du symbole $, ce qui correspond bien à ce qu'on souhaite.

exemple 1

définition des variables

```
def var
   def decla = "VAR " (use nom) ":" (use type) ";"
   def util  = (use nom)
```

déclaration d'une variable

```
def varA
   def nom = "A"
   def type = "integer"
   (ref var)
```

- nom et type sont des symboles de texte.
- varA lui est un symbole de symboles: en particulier on ne lui associe aucune valeur de représentation significative. varA concentre la déclaration *et* l'utilisation de la variable "A" de type "integer".

varA admet les propriétés :
nom, type, decla, util
La déclaration *textuelle* de "A" est:
(use decla (ref varA))
L'utilisation *textuelle* de "A" est:
(use util (ref varA))
Mais aussi, par exemple: le type de "A" – qui est un contrôle *non textuel* – est:
(use type (ref varA))

exemple 2

définition des tableaux

```
def tab
   def utiltab = (use nom) "[" (use index) "]"
   def type = "ARRAY [1.." (use taille) "] OF "
                                        (use typeelt) ";"
   (ref var)
```

définition d'une variable tableau

```
def tabB
   def nom = "B"
   def taille = "100"
   def typeelt = "real"
   (ref tab)
```

Dans tab:
- on *ajoute* la propriété: utiltab,
- on *masque* le paramètre type en l'instanciant
Les paramètres sont alors: nom, taille et typeelt.

L'utilisation *textuelle* de "B" peut s'intéresser au tableau global ou au tableau indicé:

```
le tableau global :  (use util (ref tabB))
le tableau indicé :  (use utiltab
                         (def index = "1")
                         (ref tabB))
```

tabB concentre ces deux sortes d'utilisation du même objet "B" dans deux perspectives différentes – deux types différents.

9. le contexte d'évaluation

9.1. présentation

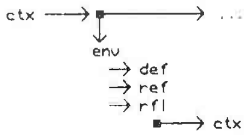
On résume ici la façon dont on construit le contexte d'évaluation d'un texte.

Le contexte est une *liste ordonnée* d'environnements évalués. Chaque environnement évalué se compose:
- de définitions def,
- de références indéfinies ref,
- de références évaluées rfl.

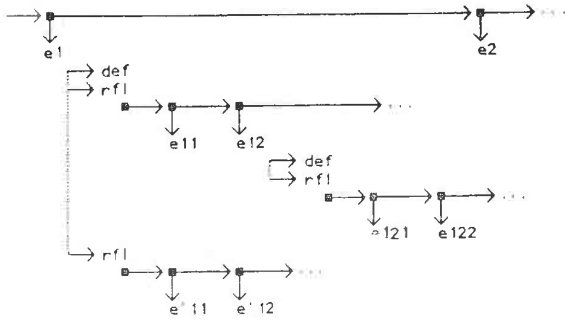
- Une définition def est invariante par évaluation: sa valeur est la définition elle-même.
- Une référence indéfinie est une valeur issue:
  - soit d'une référence évaluée pour laquelle on n'a pas trouvé de définition du texte référencé;
  - soit du fait qu'on est en cours d'évaluation de l'environnement où apparaît cette référence, et que celle-ci n'a pas encore été évaluée.

  On *ne réévalue pas* une référence indéfinie: elle est retournée identiquement à elle-même.
- Une référence évaluée rfl est un contexte d'évaluation: c'est la liste des environnements locaux de référence attachés à la référence évaluée rfl et rangés "dans le bon ordre".

Avec la représentation graphique du contexte:



on peut voir sur un exemple de contexte quel est l'ordre de parcours de l'arbre:



:: suite

ordre d'évaluation

```
:[e1 :def
     :[e11]
      [e12 :def
           :[e121]
            [e122]
            ...
           ...]
      ...
     :[e'11]
      [e'12]
      ...
     ...]
 [e2]
 ...
```

syntaxe évaluée

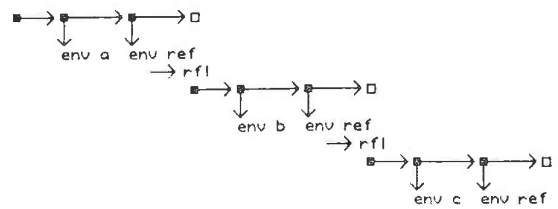
(1) ctx ::= ε | env ctx
(2) env ::= ε | trm env
(3) trm ::= def | ref | rfl
(4) def ::= SINGLETON
(5) def ::= SINGLETON
(6) rfl ::= ctx

(1) un contexte est une liste d'environnements
(2) un environnement est une liste de termes
(3) un terme est une définition def ou une référence ref ou une référence évaluée rfl
(4) une définition est un atome du langage
(5) une référence est un atome du langage
(6) une référence évaluée est un contexte

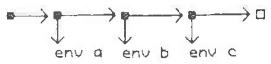
9.2. aplatissement des contextes

Il s'agit des contextes issus d'une expression de chemin.
Par exemple:

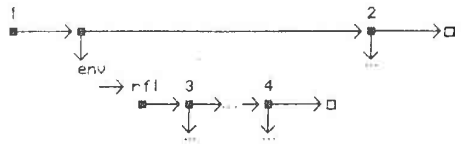
contexte construit



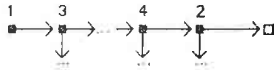
contexte équivalent



La figure suivante présente le principe général de *l'aplatissement des contextes*: si l'environnement est composé uniquement d'une référence évaluée rfl, alors on insère, au lieu de l'environnement, le contexte attaché à cette référence évaluée.



contexte équivalent



syntaxe aplatie

```
(1) env ::= ε | trm env
(2) trm ::= def | ref | env
(3)     def ::= SINGLETON
(4)     def ::= SINGLETON
```

Cette syntaxe est très proche de la syntaxe évaluée précédente: on a simplement unifié ici la notion d'environnement et la notion de contexte, sachant que:
- la conversion d'un contexte en un environnement est l'ancienne notion de référence évaluée rfl;
- la conversion d'un environnement en un contexte correspond à la notion d'aplatissement des contextes.

Un environnement sera alors aussi bien un environnement évalué qu'une référence évaluée.

Note:
L'aplatissement des contextes n'a pas de justification théorique profonde, mais elle ne modifie pas la sémantique du contexte construit. Il s'agit plutôt d'un procédé d'optimisation de l'espace mémoire utilisé et du temps d'accès aux définitions de texte. Il permet néanmoins de bien faire ressortir l'aspect spécifique des expressions de chemins: en effet c'est justement dans un tel cas que le contexte pourra être en partie simplifié.

Chapitre 9.2:

La Syntaxe Abstraite : Manuel du concepteur

On doit dégager deux niveaux d'utilisation de l'outil:
- *le niveau utilisateur*: l'utilisateur utilise les définitions de textes, les références à des textes, ..., et les opérateurs existants; il ne se soucie pas de leur construction;
- *le niveau concepteur*: le concepteur conçoit de nouveaux opérateurs ou de nouveaux phyla, à partir de la *syntaxe initiale* ou d'une *syntaxe complétée*.

On trouve ici ce qui intéresse le concepteur, à savoir les services qui lui sont offerts et les contraintes qu'il doit respecter à la réalisation des fonctions Lisp qui vont définir les attributs de nouveaux éléments du langage.

1. Attributs de la syntaxe abstraite

1.1. introduction

Les attributs sont des fonctions Lisp, qui sont fournies à la définition d'un nouvel opérateur ou d'un nouveau phylum. L'argument de ces fonctions est toujours le pointeur sur la liste préfixée par l'opérateur, non compris le préfixe.
Par exemple:

```
(use txt (env))
```

l'argument est la liste (deux arguments):

```
(txt (env))
```

ou encore:

```
(def point (env) (rep "."))
```

l'argument est la liste (trois arguments):

```
(point (env) (rep "."))
```

1.2. évaluation des attributs

L'évaluation d'un attribut est l'appel de la fonction que désigne cet attribut. Elle est toujours réalisée *au sein d'un phylum*. L'évaluation a lieu en deux temps:
- recherche de la fonction,
- appel de la fonction.

La recherche de la fonction est réalisée comme suit:

- on reçoit, au sein d'un phylum, un objet Lisp – une S-ex;
- si la S-ex est un atome Lisp, on appelle le traitement défini dans le phylum pour les atomes;
- sinon, c'est une liste: on regarde alors le préfixe de la liste, c'est-à- dire le premier élément de la liste:
  - s'il s'agit du nom d'un opérateur appartenant au phylum dans lequel on se situe, on appelle alors la fonction associée à cet opérateur;
  - il en est de même s'il s'agit du nom d'un opérateur appartenant à un phylum contenu dans le phylum dans lequel on se situe;
  - sinon, s'il s'agit quand même d'un nom d'opérateur, on appelle le traitement défini dans le phylum pour les cas d'occurrence d'un opérateur non attendu;
  - sinon enfin, on appelle le traitement défini dans le phylum pour les listes – le traitement est appelé à défaut d'autre chose;

Par exemple, avec la *syntaxe initiale*, on évalue dans le phylum TRM:

```
(def x (env ...) (rep ...))
```

def appartient au phylum TRM, on appelle la fonction associée à l'opérateur def.

```
(env (def a ...) (def b ...))
```

env appartient au phylum ENV, lui-même contenu dans le phylum TRM: on appelle la fonction associée à l'opérateur env.


```
(rep "a" "b" ...)
```

rep n'est pas visible du phylum TRM, mais est bien un opérateur: on appelle le traitement défini dans le phylum TRM pour les cas d'occurrence d'un opérateur non attendu.

```
(if (...) ...)
```

if n'est pas un opérateur: on appelle le traitement des listes défini dans le phylum TRM.

1.3. attributs d'opérateur

1.3.1. évaluation

C'est la fonction d'évaluation des textes. L'évaluation des textes étant activée soit par une demande explicite soit par l'évaluateur Lisp, on définit une "passerelle" entre ces deux évaluateurs: ce sont les *fonctions définies*.

FONCTIONS DEFINIES

- Pour un opérateur de nom oper de fonction d'évaluation evalt-oper:

  - on définit la F-expr:

    ```
    (df oper arg
        (apply 'evalt-oper arg))
    ```

  - la représentation interne de l'opérateur est une liste dont le premier élément est le nom oper:

    ```
    (oper ... )
    ```

    Ainsi, l'évaluateur Lisp activera nécessairement l'appel de la fonction d'évaluation evalt-oper sur la liste des arguments de l'opérateur *non évalués*.

- Parce que le besoin s'en est fait sentir, on définit aussi une fonction qui teste si une liste représente un opérateur donné: pour un opérateur de nom oper:

  - on définit la fonction:

    ```
    (de operp (!oc)
        (eq 'oper (car !oc)))
    ```

    Cette fonction teste si le préfixe de la liste (le "car") est égal au nom de l'opérateur.

VARIABLE HERITEE

A l'activation de la fonction d'évaluation, on a la visibilité de la variable **ctx** qui a pour valeur le contexte courant d'évaluation.

FONCTIONS UTILISATEUR

cons-env <env1> ... <envN>
contruit le contexte d'évaluation formé des environnements <env1> ... <envN> qui sont aplatis, excepté le dernier environnement <envN>.

flat-env <env>
retourne l'environnement <env> aplati.

1.3.2. recherche

C'est la fonction qui recherche une définition dans un contexte d'évaluation. Comme il a été dit, la fonction de recherche n'est peut-être pas un attribut bien spécifique des opérateurs. Pour ce qui est de la *syntaxe initiale*, cet attribut n'est utilisé qu'à l'intérieur d'un environnement, autrement dit depuis le phylum des termes TRM.

VARIABLE HERITEE

A l'appel de la fonction de recherche, on a la visibilité de la variable **nom** qui est le nom du texte qu'on recherche.

FONCTION UTILISATEUR

recht <nom>
recherche le texte de nom <nom> dans le contexte d'évaluation représenté par la variable **ctx** (qui doit être visible).

1.3.3. complétion

C'est la fonction qui complète une liste pour construire une liste conforme à la représentation interne des données. Par exemple:

```
(txt)
```

sera complétée en:

```
(use txt (env))
```

1.3.4. impression

On a deux fonctions d'impression:
- la fonction d'*impression concise*, qui est appelée quand l'opérateur obtenu était attendu – c'est-à-dire quand l'opérateur appartient au phylum dans lequel on imprime;
- la fonction d'*impression complète*, qui est appelée dans les autres cas – ce sont les cas où l'opérateur n'appartient pas au phylum dans lequel on imprime mais à l'un des phyla contenus dans ce phylum, directement ou non.

VARIABLES HERITEES

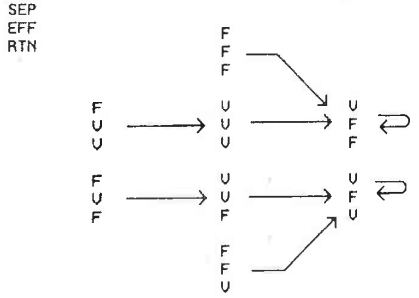
Elle sont au nombre de trois:

variable: **holo**
C'est l'*holophraste*: elle permet de mesurer à quelle profondeur on se situe dans l'arbre imprimé, et de ne réaliser éventuellement qu'une impression partielle.

variable: **tab**
C'est la *tabulation*: 0 = début de ligne.

variable: **etat**
C'est, comme son nom l'indique, une *variable d'état*. Elle est composée de trois champs booléens – c'est une liste à trois éléments:
- SEP = séparateur,
- EFF = effacement,
- RTN = retour à la ligne.

Les changements d'états sont représentés sur la figure:



- SEP = faux: aucun affichage de séparateur; SEP devient vrai;
- EFF = vrai: on affiche le séparateur, puis on inverse le champ RTN; EFF devient faux;
- RTN = faux: le séparateur est un blanc;
- RTN = vrai: le séparateur est un retour à la ligne.

FONCTIONS UTILISATEUR

print-deb <arg1> ... <argN>
imprime les arguments, *après* avoir tenu compte de l'état.

print-RTN <arg1> ... <argN>
imprime les arguments, en forçant localement un retour à la ligne.

print-noRTN <arg1> ... <argN>
imprime les arguments, en forçant localement "pas de retour à la ligne".

print-tab <tab>
imprime un retour à la ligne, puis <tab> blancs (la tabulation).

1.3.5. autres attributs

Il reste trois attributs, qui sont présentés dans le paragraphe traitant de l'*éditeur ligne* (cf. § 1.6). Il s'agit de:
- l'entrée dans les champs d'un opérateur,
- la sortie des champs d'un opérateur,
- l'impression d'un opérateur évalué.

1.4. attributs de phylum

traitement nul
C'est le traitement appelé à l'occurrence d'un opérateur qui n'est pas attendu dans le phylum dans lequel on se place. Ce sera vraissemblablement un cas d'erreur.

traitement des atomes
C'est le traitement réservé aux atomes Lisp dans le phylum.
L'argument de la fonction est alors l'atome lui-même.

traitement des listes
C'est le traitement réservé aux listes dans le phylum, c'est-à-dire aux listes dont le premier élément (le préfixe de la liste) n'est pas le nom d'un opérateur. Ce sera encore souvent un traitement d'erreur.
L'argument de la fonction est ici un pointeur sur la liste – il y a donc autant d'arguments à l'appel de la fonction que d'éléments dans la liste.

On définit ces traitements pour:
- la recherche,
- la complétion,
- l'impression, nécessairement complète,
- l'entrée dans les champs d'un opérateur,
- la sortie des champs d'un opérateur,
- l'impression évaluée.

On constate en particulier qu'on n'a pas d'attribut de phylum pour l'évaluation. Ceci se comprend bien: l'évaluation d'un objet Lisp est en effet appelée soit par l'opérateur lsp soit par l'évaluateur Lisp, dans la phase d'évaluation d'une S-ex définie dans lsp. Il est donc inutile de définir une fonction d'évaluation pour les cas exceptionnels (nul, atome, liste) qui ne serait alors jamais appelée.

Remarque:

De ce qui précède, on peut conclure:
- le traitement appelé sur un opérateur est généralement celui qui est défini pour cet opérateur: il est donc indépendant du phylum dans lequel on évalue;
- les traitements des cas exceptionnels (nul, atome, liste) sont définis pour le phylum dans lequel on évalue, indépendamment des opérateurs.

Pour reprendre l'exemple des environnements fixes env-fix:
- dans le phylum TRM-FIX on attend une définition def mais pas une référence ref;
- mais à l'occurrence d'une définition def on appelle un traitement *identique* à celui qu'on aurait appelé en se plaçant dans le phylum TRM.

On peut se demander s'il s'agit d'un avantage ou d'une limitation.

1.5. syntaxe concrète

La syntaxe concrète est le point sensible du système, puisque c'est elle qui est visible de l'utilisateur, pour la saisie ou l'affichage. La syntaxe concrète est actuellement très fortement inspirée de la syntaxe Lisp, par l'emploi d'un parenthésage systématique. De ce fait, elle est sans doute d'un abord un peu rugueux.

En général, un opérateur est représenté sous la forme d'une liste préfixée par le nom de l'opérateur. On omet le préfixe à l'affichage, et on dispense d'indiquer le préfixe à la saisie, si l'opérateur est attendu "de manière privilégiée" dans le phylum dans lequel on se situe:
- phylum SEX : pas d'opérateur
- phylum TRM : opérateur privilégié = ref
- phylum ATM : opérateur privilégié = use
- phylum ENV : opérateur privilégié = env
- phylum REP : opérateur privilégié = rep
- phylum LSP : opérateur privilégié = lsp

On retrouve ainsi la double écriture des emplois de textes, «quand il n'y a pas d'ambiguïté possible». On aura en fait d'autres cas d'écriture simplifiée – par exemple si un champ est vide, on ne l'affiche pas.Par exemple:

écriture complète

```
(def txt
  (env
    (def a
      (env)
      (rep "A"))
    (def b
      (env)
      (rep "B")))
  (rep
    (use a (env)) ":=" (use b (env))))
```

écriture simplifiée

```
(def txt
  ((def a ()
     ("A"))
   (def b ()
     ("B")))
  ((a) ":=" (b)))
```

1.6. attributs d'opérateur dans l'éditeur ligne

Cette partie s'intéresse plus spécifiquement à l'*éditeur ligne*.

1.6.1. les utilitaires

FONCTIONS EXTERNES

evaie <exp> <ctx>
retourne l'expression <exp> évaluée dans le contexte <ctx>.

sais <nomphyl>
saisie d'un opérateur dans le phylum <nomphyl>.

sais-lst <nomphyl>
saisie d'une liste d'opérateurs dans le phylum <nomphyl> – «liste» s'entend ici au sens de Lisp.

sais-num <nomphyl> <num>
saisie de <num> opérateurs dans le phylum <nomphyl>.

imp <nomphyl> <exp> <tab> <holo>
impression de l'expression <exp> dans le phylum <nomphyl>, avec une tabulation initiale <tab> et un holophraste initial <holo>.
imp-lst <nomphyl> <lst-exp> <tab> <holo>
impression de la liste d'expressions <lst-exp>.
imp-num <nomphyl> <lst-exp> <tab> <holo> <num>
impression des <num> premiers éléments de la liste <lst-exp>, s'il se peut.

imp-eval <nomphyl> <exp-eval> <tab> <holo>
impression de l'expression évaluée <exp-eval> dans le phylum <nomphyl>, avec une tabulation initiale <tab> et un holophraste initial <holo>.
imp-lst-eval <nomphyl> <lst-exp-eval> <tab> <holo>
impression de la liste d'expressions évaluées <lst-exp-eval>.
imp-num-eval <nomphyl> <lst-exp-eval> <tab> <holo> <num>
impression des <num> premiers éléments de la liste d'expressions évaluées <lst-exp-eval>, s'il se peut.

VARIABLES HERITEES

gtab : tabulation – incrémenté par la commande '('
gholo : holophraste – modifié par la commande 'h'
gnum : valeur de <num> à l'interprétation d'une commande.

genv : environnement courant – il permet de construire le contexte d'évaluation des textes.

gref : référence sur la position courante; on utilise une référence sur l'objet, et non l'objet lui-même, pour réaliser les effets de bord (insertion, suppression).
glst : référence de début de liste des "termes".
goper : opérateur de la position courante.
gphyl : phylum de la position courante.

FONCTIONS UTILISATEUR

ERREUR-com <msg>
affiche le message <msg>, et vide le tampon des commandes: les commandes qui suivent sont ignorées, et le contrôle est rendu à l'utilisateur.
num-com
retourne dans la variable **gnum** la valeur de <num> à l'interprétation d'une commande.
eval-macro <macro>
construit une liste de commandes "calquée" sur <macro>, où les commandes de paramètre '-' sont instanciées.
eval-ctx <lst-env>
construit le contexte d'évaluation à partit de la *liste ordonnée* des environnements <lst-env>.

sais-rsp
construit le tampon des commandes, sous forme de liste, à partir de la ligne frappée par l'utilisateur.
sais-exp
saisit une expression Lisp (S-ex).

imp-vde
impression d'une liste vide: ...

pred <loc> <lst>
renvoie la référence qui précède immédiatement <loc> dans la liste <lst>.
succ <loc> <lst>
renvoie la référence qui suit immédiatement <loc> dans la liste <lst>.
sets <var1> <val1> ... <varN> <valN>
sauvegarde, *en parallèle*, les valeurs des variables <var1> ... <varN>, puis affecte à ces variables les valeurs respectives <val1> ... <valN>.
setb <var1> ... <varN>
restaure les valeurs des variables <var1> ... <varN>.

Note: **sets** et **setb** utilisent, pour une variable **var** donnée, une variable tampon Lisp: lst*var. On accède donc à chaque instant au contexte d'évaluation *non évalué* par la variable lst*genv.

1.6.2. entrée

L'*entrée* dans les champs d'un opérateur: on numérote les champs d'un opérateur, de 1 à N, dans l'ordre où ils apparaissent dans l'expression de la *syntaxe initiale*.
Par exemple:
- la commande: 'e2' devant un opérateur use permet d'entrer dans l'environnement local de l'utilisation;
- la commande: 'e' devant un opérateur def permet d'entrer dans le champ <nom> de l'opérateur (le nom de l'opérateur).

1.6.3. sortie

La *sortie* des champs d'un opérateur: elle ne dépend pas du champ dont on vient mais de l'opérateur dans lequel on était entré.

1.6.4. impression évaluée

L'*impression évaluée*: il s'agit généralement de la même impression que l'impression non évaluée; cependant pour les représentations ou les atomes atm, on présente une impression plus "attractive" – du moins plus éloignée de la représentation interne des textes.

2. La Syntaxe Initiale

2.1. schémas

Ce sont des schémas de définition incomplète de phyla ou d'opérateurs. On ne leur attache aucune sémantique précise, ils permettent uniquement d'éviter des répétitions inutiles dans la définition de la *syntaxe initiale*.

Ces schémas sont de deux natures:
- ou bien il s'agit de fonctions Lisp ("de"): tous les phyla ou opérateurs utilisant ces fonctions subiront un traitement identique – aux paramètres de la fonction près;
- ou bien il s'agit de dmacros Lisp ("dmd"): pour chaque phylum ou opérateur défini sur la dmacro, on appliquera alors le traitement contenu dans la forme expansée de la dmacro, qui prendra en compte certaines spécificités du phylum ou de l'opérateur défini — les paramètres de la dmacro.
  On peut noter que, dans ce dernier cas, l'emploi de dmacros n'est pas pénalisant pour l'efficacité du programme: en effet, dans la version Lisp utilisée (LeLisp V15.2), une dmacro est expansée la première fois qu'elle est utilisée; l'interpréteur Lisp remplace alors *physiquement* l'appel à la dmacro par la forme expansée qui vient d'être calculée; on n'a donc qu'une unique expansion de la dmacro.

2.1.1. schéma de phylum: erreur (err)

C'est le phylum des erreurs. Toutes les fonctions retournent la valeur nil (qui est le 'faux' de Lisp).

recht-err <arg> ...
erreur à la recherche.

cpltt-err <arg> ...
erreur à la complétion.

imptt-err <arg> ...
erreur à l'impression.

entre-err <arg> ...
erreur en entrée: l'évaluation de cet attribut est généralement signe de mauvais augure – le cas ne devrait normalement pas se produire.

sorte-err <arg> ...
erreur en sortie: ici aussi l'évaluation de cet attribut est un mauvais signe.

evale-err <arg> ...
erreur à l'impression évaluée: l'opérateur ne peut être évalué individuellement – par exemple: un élément d'une S-ex.

«Erreur» doit s'entendre ici: on obtient un objet Lisp qui n'est pas attendu dans le phylum dans lequel on se situe: par exemple un atome, une liste, un opérateur non visible du phylum.

2.1.2. schéma de phylum: liste (phylLST)

C'est le schéma des phyla pour lesquels:
- il est défini un opérateur de liste,
- cet opérateur est l'«opérateur privilégié» du phylum.

Il s'agit donc des phyla ENV, REP et LSP sur les opérateurs privilégiés respectifs env, rep et lsp.

Les fonctions ont deux paramètres nécessaires:
- <nomphyl>: le nom du phylum
- <nomlst>: le nom de l'opérateur de liste privilégié.

cpltt-phylLST-nil <nomphyl> <nomlst>
complète un objet Lisp par l'opérateur de liste <nomlst>, la liste étant vide.

imptt-phylLST-nil <nomphyl> <nomlst>
imprime un objet Lisp en imprimant en fait l'opérateur de liste <nomlst>, la liste étant vide.

evale-phylLST <nomphyl> <nomlst>
impression évaluée, dans le phylum <nomphyl>, de l'opérateur de liste <nomlst> référencé par la variable: **gref**.

2.1.3. schéma d'opérateur: liste (operLST)

C'est le schéma des opérateurs de liste. Cela concerne:
- l'opérateur env du phylum ENV,
- l'opérateur rep du phylum REP,
- l'opérateur lsp du phylum LSP.

Les fonctions ont au moins les deux paramètres nécessaires:
- <nomlst>: le nom de l'opérateur de liste,
- <nomelt>: le nom du phylum des éléments de la liste que constitue l'opérateur <nomlst>.

cpltt-operLST <nomlst> <nomelt> <arg>
complète la liste d'opérateurs <arg> dans le phylum <nomelt> et retourne l'opérateur <nomlst> ainsi complété.

imptt-operLST <nomlst> <nomelt> <arg>
imprime l'opérateur <nomlst> sous sa forme complète.

imptc-operLST <nomlst> <nomelt> <arg>
imprime l'opérateur <nomlst> sous sa forme concise.

entre-operLST <nomlst> <nomelt>
entrée dans l'opérateur de liste <nomlst>, d'éléments du phylum <nomelt>, référencé par la variable: **gref**.

sorte-operLST <nomlst> <nomelt>
sortie de l'opérateur de liste <nomlst>.

Pour l'impression complète:
- on préfixe la liste par le nom de l'opérateur de liste;
- chaque nouvel élément de la liste provoque un retour à la ligne.

Pour l'impression concise:
- on ne préfixe pas la liste;
- si la liste est vide, on force un "non-retour" à la ligne: la liste vide est imprimée sur la ligne courante.

2.1.4. schéma d'opérateur: utilisation (operUTI)

Ce schéma regroupe les deux opérateurs d'emploi:
- l'opérateur ref du phylum TRM,
- l'opérateur use du phylum ATM.

Les fonctions ont au moins le paramètre nécessaire:
- <nomuti>: le nom de l'opérateur d'emploi.

cpltt-operUTI <nomuti> <arg>
complète l'opérateur <nomuti> par le nom et l'environnement local définis dans <arg>.

imptt-operUTI <nomuti> <nom> <env>
imprime l'opérateur <nomuti> de nom <nom> d'environnement local <env>, sous sa forme complète.

imptc-operUTI <nomuti> <nom> <env>
imprime l'opérateur <nomuti> sous sa forme concise.

entre-operUTI
entrée dans l'opérateur référencé par la variable: **gref**.

sorte-operUTI
sortie de l'opérateur.

Pour l'impression, l'environnement local est généralement imprimé sur une nouvelle ligne. Cependant, si les environnements sont en *cascade*, c'est-à-dire si l'emploi de texte est une «expression de chemin», alors on l'imprime sur une même ligne. On souligne donc ici aussi le rôle spécifique des expressions de chemin.

2.1.5. schéma d'opérateur: opérateur nommé (nom)

Ce schéma regroupe les opérateurs nommés. Ce sont :
-l'opérateur def du phylum TRM,
-l'opérateur ref du phylum TRM,
-l'opérateur stg du phylum ATM,
-l'opérateur use du phylum ATM.

entre-nom
modification du nom.
On n'"entre" pas dans le champ nom, on n'a donc pas d'attribut de sortie: **sorte-nom**.

2.2. syntaxe abstraite

L'"expérience" montre qu'on peut dégager deux sortes d'attributs:
- des attributs liés à un phylum: tous les opérateurs du phylum considéré auront un semblable attribut;
- des attributs liés à un opérateur: l'attribut est spécifique à l'opérateur désigné.

On trouve généralement pour le premier cas:
- les attributs de phylum, qui sont les fonctions appelées pour les cas exceptionnels (nul, atome, liste);
- l'attribut d'impression évaluée: ceci n'est pas vraiment surprenant, on conçoit bien que l'évaluation d'un opérateur sera rarement indépendante du contexte de définition, et dépendra donc davantage du phylum auquel l'opérateur appartient que de son comportement isolé.

On trouve généralement pour le deuxième cas:
- les attributs d'évaluation et de recherche, qui sont spécifiques à chaque opérateur;
- les attributs de complétion, impression, entrée et sortie des champs de l'opérateur, qui sont souvent regroupés dans un schéma d'opérateur.

Note: on présente ici les vrais attributs de phyla et d'opérateurs (et non les *schémas* d'attributs): l'argument des fonctions sera donc toujours la liste préfixée par l'opérateur, privée de ce préfixe.

2.2.1. phylum des environnements: ENV

ATTRIBUTS DE PHYLUM

cpltt-ENV-nil <arg> ...
cas d'erreur: on retourne un environnement vide: (env)
schéma cpltt-phylLST-nil

imptt-ENV-nil <arg> ...
cas d'erreur: on imprime l'environnement vide
schéma imptt-phylLST-nil

evale-ENV <arg> ...
impression évaluée: schéma evale-phylLST

OPERATEUR: env

evalt-env <arg> ...
évaluation progressive d'un environnement.

recht-env <arg> ...
recherche d'un nom de texte dans un environnement.

cpltt-env <arg> ...
complétion d'un environnement: schéma cpltt-operLST

imptt-env <arg> ...
impression complète d'un environnement: schéma imptt-operLST

imptc-env <arg> ...
impression concise d'un environnement: schéma imptc-operLST

entre-env <arg> ...
entrée: schéma entre-operLST

sorte-env <arg> ...
sortie: schéma sorte-operLST

2.2.2. phylum des représentations: REP

ATTRIBUTS DE PHYLUM

cpltt-REP-nil <arg> ...
cas d'erreur: on retourne une représentation vide: (rep)
schéma cpltt-phylLST-nil

imptt-REP-nil <arg> ...
cas d'erreur: on imprime la représentation vide
schéma imptt-phylLST-nil

evale-REP <arg> ...
impression évaluée: schéma evale-phylLST

OPERATEUR: rep

evalt-rep <arg> ...
évaluation d'une représentation.

cpltt-rep <arg> ...
complétion d'une représentation: schéma cpltt-operLST

imptt-rep <arg> ...
impression complète d'une représentation: schéma imptt-operLST

imptc-rep <arg> ...
impression concise d'une représentation: schéma imptc-operLST

entre-rep <arg> ...
entrée: schéma entre-operLST

sorte-rep <arg> ...
sortie: schéma sorte-operLST

Remarques:
- On n'a pas de fonction **recht-rep** de recherche dans une représentation, qui est un cas d'erreur, ce qui est bien naturel.
- Les impressions ne sont pas *exactement* calquées sur le schéma d'impression de operLST; par souci de lisibilité, on imprime les atomes atm de la représentation sur une même ligne, sauf mention expresse – le retour à la ligne "^M".
En conséquence, les attributs d'impression ne sont pas des appels aux dmacros du schéma operLST, mais des fonctions "en ligne". L'introduction des schémas n'est donc pas faite pour mieux lier encore les concepts mais juste pour faciliter l'écriture du programme, et on les utilise tant que c'est possible.

2.2.3. phylum des expressions Lisp: LSP

ATTRIBUTS DE PHYLUM

cpltt-LSP-nil <arg> ...
cas d'erreur : on retourne un texte Lisp vide: (lsp)
schéma cpltt-phylLST-nil

imptt-LSP-nil <arg> ...
cas d'erreur: on imprime le texte Lisp vide
schéma imptt-phylLST-nil

evale-LSP <arg> ...
impression évaluée: schéma evale-phylLST

OPERATEUR: lsp

evalt-lsp <arg> ...
évaluation d'un texte Lisp.

cpltt-lsp <arg> ...
complétion d'un texte Lisp: schéma cpltt-operLST

imptt-lsp <arg> ...
impression complète d'un texte Lisp: schéma imptt-operLST

imptc-lsp <arg> ...
impression concise d'un texte Lisp: schéma imptc-operLST

entre-lsp <arg> ...
entrée: schéma entre-operLST

sorte-lsp <arg> ...
sortie: schéma sorte-operLST

Remarque:
Ici encore on ne définit pas de fonction **recht-lsp** de recherche dans un texte Lisp.

2.2.4. phylum des termes: TRM

ATTRIBUTS DE PHYLUM

Complétion et impression dans les cas exceptionnels (nul, atome, liste) seront des cas d'erreur.

evale-TRM <arg> ...
impression évaluée: on imprime l'environnement en son entier.

OPERATEUR: def

evalt-def <nom> <env> <rep>
évaluation d'une définition: la définition est retournée identiquement à elle-même.

recht-def <nom> <env> <rep>
recherche d'un nom de texte: on teste ici si le nom de la définition est égal au nom recherché, qui est contenu dans la variable **ctx**.

cpltt-def <arg> ...
complétion d'une définition.

imptt-def <nom> <env> <rep>
impression complète d'une définition.

imptc-def <nom> <env> <rep>
impression concise d'une définition: elle est dans ce cas identique à l'impression complète de la définition.

entre-def <arg> ...
entrée: e1 = nom e2 = environnement e3 = représentation

sorte-def <arg> ...
sortie.

OPERATEUR: ref

evalt-ref <nom> <env>
évaluation d'une référence.

recht-ref <nom> <env>
recherche d'un nom de texte: la fonction retourne toujours 'faux' – on *ne réévalue pas* une référence indéfinie.

cpltt-ref <arg> ...
complétion d'une référence: la fonction est appelée:
- quand on a rencontré un opérateur de référence ref,
- quand on a rencontré une liste qui n'était aucun opérateur connu – traitement par défaut.
schéma cpltt-operUTI

imptt-ref <nom> <env>
impression complète d'une référence: schéma imptt-operUTI

imptc-ref <nom> <env>
impression concise d'une référence: schéma imptc-operUTI

entre-ref <arg> ...
entrée: e1 = nom e2 = environnement local
schéma entre-operUTI

sorte-ref <arg> ...
sortie: schéma sorte-operUTI

2.2.5. phylum des atomes: ATM

ATTRIBUTS DE PHYLUM

cpltt-ATM-atome <arg>
complétion d'un atome Lisp: on retourne l'atome <arg> s'il s'agit d'une chaîne de caractères (string), et la valeur nil sinon – cas d'erreur.

imptt-ATM-atome <arg>
imprime l'atome <arg>: s'il s'agit d'une chaîne de caractères, on l'encadre par des guillemets et on tient compte du cas où cette chaîne spécifie un retour à la ligne "^M".

Ces deux fonctions prennent en compte le choix de représentation interne des chaînes de caractères: une chaîne de caractères de la *syntaxe initiale* est une string de Lisp. Or une string est un atome de Lisp, il faut donc lever la petite ambiguïté qui en résulte.

evale-ATM <arg> ...
impression évaluée.

OPERATEUR: stg

evalt-stg <nom>
évaluation de stg: on retourne la conversion en chaîne de caractères du nom (fonction Lisp string).

cpltt-stg <arg> ...
complétion de stg.

imptt-stg <nom>
impression complète de stg.

imptc-stg <nom>
impression concise de stg: on n'imprime que le nom.

entre-stg <arg> ...
entrée: schéma entre-nom
Comme il a été dit, on n'a pas d'attribut de sortie: **sorte-stg**.

OPERATEUR: use

evalt-use <nom> <env>
évaluation d'une utilisation.
On peut noter que la fonction est *presque* identique à celle de l'évaluation d'une référence; la différence réside uniquement dans la valeur retournée par l'évaluation – qui une représentation ou une chaîne de caractères, qui un environnement évalué ou une référence indéfinie.

cpltt-use <arg> ...
complétion d'une utilisation: schéma cpltt-operUTI

imptt-use <nom> <env>
impression complète d'une utilisation: schéma imptt-operUTI

imptc-use <nom> <env>
impression concise d'une utilisation: schéma imptc-operUTI

entre-use <arg> ...
entrée: e1 = nom e2 = environnement local
schéma entre-operUTI

sorte-use <arg> ...
sortie: schéma sorte-operUTI

Remarque:
On n'a encore aucune fonction de recherche, ni pour stg ni pour use.

2.2.6. phylum des S-expressions: SEX

On regroupe ici les attributs attachés au phyla "prédéfinis" ATOME et LISTE.

On ne définit pas de fonction de recherche pour les cas exceptionnels (nul, atome, liste): ce seront toujours des cas d'erreur.

cpltt-atome <arg>
complétion d'un atome: l'atome est retourné identiquement à lui-même.

cpltt-liste <arg> ...
complétion d'une liste: on retourne la liste dans laquelle on a récursivement appelé l'attribut de complétion sur chaque élément de la liste.

imptt-atome <arg>
impression d'un atome: pour une meilleure lecture, les chaînes de caractères Lisp (string) sont encadrées par des guillemets.

imptt-liste <arg> ...
impression d'une liste.

entre-atome <arg>
entrée dans un atome: l'entrée est refusée.

entre-liste <arg> ...
entrée dans une liste: on définit une liste auxiliaire préfixée par lsp, qui construit un objet sur le modèle de l'opérateur lsp. On en déduit qu'on n'a pas d'attribut de sortie, ni sur les atomes, ni sur les listes.

Ces deux fonctions sont toujours les attributs attachés aux cas exceptionnels des autres phyla, respectivement pour le cas atome et le cas liste. On obtient donc éventuellement des impressions "surchargées", où apparaissent des termes où on n'en attend pas, mais aussi on a une impression plus conforme à l'état courant de la représentation interne des objets qu'on manipule.

2.3. résumé

Les tableaux suivants présentent les différents attributs définis pour les phyla et les opérateurs de la *syntaxe initiale*.

phylum ENV

	cas nul	**cas atome**	**cas liste**
recherche	recht-err	recht-err	recht-err
complétion	cpltt-ENV-nil	cpltt-ENV-nil	cpltt-env
impression	imptt-ENV-nil	imptt-atome	imptt-liste
entrée	entre-err	entre-err	entre-err
sortie	sorte-err	sorte-err	sorte-err
impression évaluée	evale-err	evale-err	evale-err

phylum REP

	cas nul	**cas atome**	**cas liste**
recherche	recht-err	recht-err	recht-err
complétion	cpltt-REP-nil	cpltt-REP-nil	cpltt-rep
impression	imptt-REP-nil	imptt-atome	imptt-liste
entrée	entre-err	entre-err	entre-err
sortie	sorte-err	sorte-err	sorte-err
impression évaluée	evale-err	evale-err	evale-err

phylum LSP

	cas nul	**cas atome**	**cas liste**
recherche	recht-err	recht-err	recht-err
complétion	cpltt-LSP-nil	cpltt-LSP-nil	cpltt-lsp
impression	imptt-LSP-nil	imptt-atome	imptt-liste
entrée	entre-err	entre-err	entre-err
sortie	sorte-err	sorte-err	sorte-err
impression évaluée	evale-err	evale-err	evale-err

phylum TRM

	cas nul	**cas atome**	**cas liste**
recherche	recht-err	recht-err	recht-err
complétion	cpltt-err	cpltt-err	cpltt-ref
impression	imptt-err	imptt-atome	imptt-liste
entrée	entre-err	entre-err	entre-err
sortie	sorte-err	sorte-err	sorte-err
impression évaluée	evale-err	evale-err	evale-err

phylum ATM

	cas nul	**cas atome**	**cas liste**
recherche	recht-err	recht-err	recht-err
complétion	cpltt-err	cpltt-ATM-atome	cpltt-use
impression	imptt-err	imptt-ATM-atome	imptt-liste
entrée	entre-err	entre-err	entre-err
sortie	sorte-err	sorte-err	sorte-err
impression évaluée	evale-err	evale-ATM	evale-err

phylum SEX

	cas nul	cas atome	cas liste
recherche	recht-err	recht-err	recht-err
complétion	cpltt-err	cpltt-atome	cpltt-liste
impression	imptt-err	imptt-atome	imptt-liste
entrée	entre-err	entre-atome	entre-liste
sortie	sorte-err	sorte-err	sorte-err
impression évaluée	evale-err	evale-err	evale-err

opérateurs

	env	rep	lsp	
évaluation	evalt-env	evalt-rep	evalt-lsp	
recherche	recht-env	recht-err	recht-err	
complétion	cpltt-env	cpltt-rep	cpltt-lsp	
impression complète	imptt-env	imptt-rep	imptt-lsp	
impression concise	imptc-env	imptc-rep	imptc-lsp	
entrée	entre-env	entre-rep	entre-lsp	
sortie	sorte-env	sorte-rep	sorte-lsp	
impression évaluée	evale-ENV	evale-REP	evale-LSP	

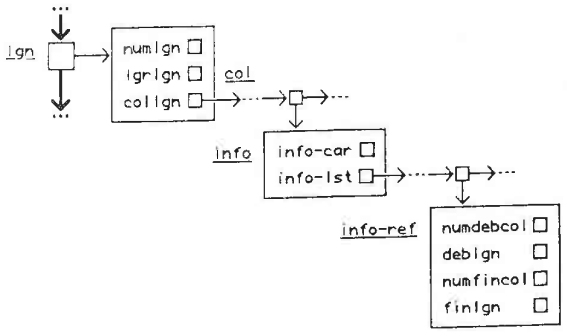
	def	ref	stg	use
évaluation	evalt-def	evalt-ref	evalt-stg	evalt-use
recherche	recht-def	recht-ref	recht-err	recht-err
complétion	cpltt-def	cpltt-ref	cpltt-stg	cpltt-use
impression complète	imptt-def	imptt-ref	imptt-stg	imptt-use
impression concise	imptc-def	imptc-ref	imptc-stg	imptc-use
entrée	entre-def	entre-ref	entre-stg	entre-use
sortie	sorte-def	sorte-ref	sorte-err	sorte-use
impression évaluée	evale-TRM	evale-TRM	evale-ATM	evale-ATM

Chapitre 9.3:

L'éditeur page : Guide de l'implanteur

On donne ici un guide pour l'implantation de l'*éditeur page*. Il s'agit de la description du programme déjà rédigé.

1. la structure de données du Buffer



1.1. les fonctions le lecture

la ligne: lgn

Les lignes de Buffer sont une suite chaînée de colonnes.

succ-lgn <lgn> : ligne suivante de la ligne <lgn>.
pred-lgn <lgn> : ligne précédente de la ligne <lgn>.
numlgn <lgn> : numéro de la ligne <lgn> – première ligne = numéro 1.
lgrlgn <lgn> : longueur de la ligne <lgn>, non compris l'éventuel caractère de retour à la ligne LF – ligne composée d'un unique retour à la ligne LF = longueur 0.
collgn <lgn> : colonne associée à la ligne <lgn>.

incr-lgn <num> <lgn> : retourne la <num>-ième ligne suivant la ligne <lgn>.
decr-lgn <num> <lgn> : retourne la <num>-ième ligne précédant la ligne <lgn>.
num-abs-lgn <y> : retourne la ligne de numéro absolu <y>.

la colonne: col

Les colonnes d'une ligne sont une suite d'informations chaînée.

succ-col <col> : colonne suivante de la colonne <col>.
pred-col <col> : colonne précédente de la colonne <col>.
numcol <x> <lgn> : retourne la colonne de numéro <x> dans la ligne <lgn> – première colonne = numéro 0.

incr-col <num> <col> : retourne la <num>-ième colonne suivant la colonne <col>.
decr-col <num> <col> <lgn> : retourne la <num>-ième colonne précédant la colonne <col> dans la ligne <lgn>.

l'information: info

Les informations d'une colonne sont les éléments *indivisibles* du Buffer – les "caractères" du tampon ou les points de l'écran.

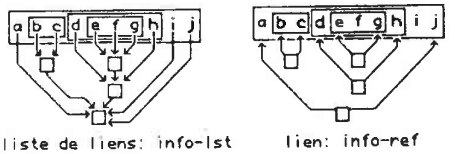
info <col> : information de la colonne <col>.
info-car <col> : information de «caractère» de la colonne <col>: c'est soit le code ASCII du caractère, soit le symbole "LF" qui représente le retour à la ligne LF.
info-lst <col> : information de «liste des liens» de la colonne <col>: c'est une liste de liens.
info-hol <col> : information d'«holophraste» de la colonne <col>: c'est le nombre de liens, autrement dit la longueur de la liste des liens de la colonne.
info-ref <hol> <col> : information de «lien» de la colonne <col> sous l'holophraste <hol> – le premier lien est sous l'holophraste 1: c'est donc le <hol>-ième lien de la liste des liens.
info-lst-hol <hol> <col> : information de «liste des liens» de la colonne <col> sous l'holophraste <hol> – la première liste des liens est sous l'holophraste 1: elle correspond à la liste des liens retournée par **info-lst**.

la liste des liens: info-lst

info-L-cdr <info-lst> : liste des liens suivante de la liste <info-lst>.
info-L-car <info-lst> : première information de «lien» de la liste <info-lst>.
info-L-hol <info-lst> : information d'«holophraste» de la liste <info-lst>.
info-L-ref <hol> <info-lst> : information de «lien» de la liste <info-lst> sous l'holophraste <hol>.
info-L-lst-hol <hol> <col> : information de «liste des liens» de la liste <info-lst> sous l'holophraste <hol>.

le lien: info-ref

Les liens associent pour un holophraste donné à une colonne donnée la première colonne de début du *curseur* et la première colonne qui suit immédiatement la fin du *curseur*. Par exemple:



numdebcol <info-ref> : numéro, dans la ligne, de la première colonne du *curseur* – le numéro est relatif, puisqu'il dépend des colonnes précédant celle qui est liée.
deblgn <info-ref> : ligne de la première colonne du *curseur* – la ligne est connue de manière absolue, c'est-à-dire que le lien est ici un lien "physique", portant sur des pointeurs.
numfincol <info-ref> : numéro, dans la ligne, de la première colonne qui suit la fin du *curseur*.
finlgn <info-ref> : ligne de la première colonne qui suit du *curseur*.

1.2. les fonctions de modification

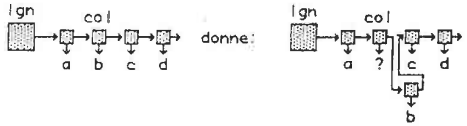
le Buffer

cons-bof-lgn : construit un début de Buffer.
cons-eof-lgn <col> <lgn> <x> : construit une fin de Buffer, sur la colonne <col> de numéro <x> sur la ligne <lgn>.
cons-eol-lgn <info-lst> <flag-nil> <lgn> <col<> <x<> <>col> <>x>
construit une fin de ligne (caractère de retour à la ligne LF) et retourne la ligne suivante:
<info-lst> liste des liens du caractère LF
<flag-nil> drapeau:
vrai: on remplit le champ d'information des colonnes
faux: on ne le remplit pas – cas des colonnes hors de la *zone d'accès*
<lgn> ligne à finir

<col<>	dernière colonne de la ligne – avant LF
<x<>	numéro de <col<> dans <lgn>
<>col>	première colonne de la ligne suivante
<>x>	numéro de <>col> dans la ligne suivante

la colonne

ins-col <col> <lgn> : insérer une colonne sur la colonne <col> de la ligne <lgn>.

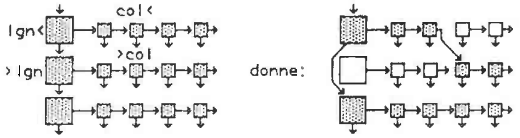


del-col <col> <lgn> : détruire la colonne <col> de la ligne <lgn>.

la ligne

unir-lgn <num> <lgn<> <col<> <>lgn> <>col> <lgrlgn>
unir les lignes:

<num>	nombre de lignes détruites
<lgn<>	ligne de "réception"
<col<>	colonne de "réception"
<>lgn>	ligne d'"envoi"
<>col>	colonne d'"envoi"
<lgrlgn>	longueur de la ligne de réception – après union

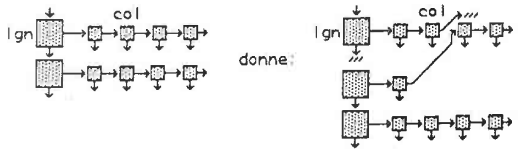


fermer-lgn <hol> <info-lst> <oldfinlgn> <oldfinx> <newfinlgn> <newfinx>
fermer la ligne:

<hol>	holophraste de début de fermeture
<info-lst>	liste des liens de début de fermeture – sous l'holophraste <hol>
<oldfinlgn>	ancienne ligne fermée
<oldfinx>	ancien numéro de la colonne fermée dans la ligne <oldfinlgn>
<newfinlgn>	nouvelle ligne ouverte
<newfinx>	nouveau numéro de la colonne ouverte dans la ligne <newfinlgn>

La fonction retourne l'holophraste de fermetu
Fermer une ligne consiste, partant d'une liste de liens et d'un holophraste donnés, à remplacer dans les liens les champs **finlgn** et **numfincol** par leur nouvelle valeur new; l'holophraste retourné est celui du premier lien pour lequel les champs n'ont pas l'ancienne valeur old.

ouvrir-lgn <lgn> <col>
ouvrir la ligne <lgn> sur la colonne <col>: retourne la nouvelle ligne.



completer-lgn <col> <x>
compléter la ligne sur la colonne <col> de numéro <x>; on ajoute un nombre fixe, *assez* grand, de colonnes dernière <col>.

clean-lgn <hol> <oldlgn> <newlgn> <col> <x>
nettoyer la ligne:

<hol>	holophraste de début de nettoyage
<oldlgn>	ancienne ligne à nettoyer
<newlgn>	nouvelle ligne à substituer à l'ancienne
<col>	colonne de début de nettoyage
<x>	numéro de la colonne <col> dans la ligne

Nettoyer une ligne c'est remplacer les liens "physiques" **deblgn** et **finlgn** sur l'ancienne ligne <oldlgn> par la nouvelle ligne <newlgn> – on nettoie entre la colonne <col> et la fin de ligne.

l'information

set-info <col> <car> <info-lst> : fixer l'information de la colonne <col>: le caractère <car> et la liste de liens <info-lst>.
set-info-nil <col> : fixer l'information de la colonne <col> – cas des colonnes hors de la *zone d'accès*
set-info-car <col> <car> : fixer l'information de «caractère» de la colonne <col> à <car>.
set-info-lst <col> <info-lst> : fixer l'information de «liste de liens» de la colonne <col> à <info-lst>.
set-info-lst-nil <col> : fixer l'information de «liste de liens» de la colonne <col> à la liste vide.
set-info-lst-hol <hol> <col> <info-lst> : remplacer l'information de «liste de liens» de la colonne <col> par la liste de liens <info-lst> sous l'holophraste <hol>.

la liste des liens

cons-info-lst <debx> <deblgn> <finx> <finlgn> <info-lst>
retourne la liste de liens construite par le lien <debx> <deblgn> <finx> <finlgn> suivi de la liste <info-lst>.
cons-info-lst-deb <debx> <deblgn> <info-lst>
retourne la liste de liens construite par le lien <debx> <deblgn> suivi de la liste <info-lst>; on ne remplit que les champs de début du lien.
set-info-lst-fin <info-lst> <finx> <finlgn>
remplace dans la liste de liens <info-lst> le premier lien par <finx> <finlgn>; on ne remplace que les champs de fin du lien – cette fonction complète **cons-info-lst-deb**.

incr-numdebcol <num> <info-ref> : incrémenter de <num> le champ **numdebcol** du lien <info-ref>.
incr-numfincol <num> <info-ref> : idem, incrémenter le champ **numfincol**.
decr-numdebcol <num> <info-ref> : idem, décrémenter le champ **numdebcol**
decr-numfincol <num> <info-ref> : idem, décrémenter le champ **numfincol**.

1.3. les variables du Buffer

Une variable du Buffer est une "structure" à 4 champs:
- x : entier = abscisse,
- y : entier = ordonnée,
- col : colonne,
- lgn : ligne.

On a quatre règles de cohérence, qui sont *normalement* respectées:
- x = numéro de la colonne col dans la ligne lgn;
- y = numéro absolu de la ligne lgn;
- la colonne col appartient à la ligne lgn.
- la ligne lgn appartient au Buffer.

On adopte plusieurs choix de représentation selon la fréquence d'emploi des variables:
- la représentation symbolique
  Par exemple, la variable log est en fait défini par les quatre variables: logx, logy, logcol et loglgn – par concaténation du symbole "log" et des suffixes "x", "y", "col" et "lgn".
- la représentation par des listes
  Par exemple, la variable curdeb est un symbole dont la valeur est la liste à quatre champs: (y x lgn col).
- la représentation immédiate
  Par exemple, la paire: (x y).

Les fonctions suivantes prennent en paramètres des expressions interprétées:
– pour un symbole: comme un symbole de représentation par des listes,
– pour un symbole quoté: comme un symbole de représentation symbolique,
– autrement, comme une représentation immédiate des champs x et y.

affectation du Buffer

set-glo <dest1> <orig1> ... : affecter à la variable <dest1> la variable <orig1> – nombre quelconque (pair) d'arguments.

set-prd <dest> <orig> <tab> : affecter à <dest> le prédécesseur dans le Buffer de <orig>, <tab> étant l'abscisse de début de ligne du Buffer local – en cas de retour en fin de ligne précédente, <dest> est placé à une abscisse *très grande*.
set-mPRD <dest> <orig> <tab> : identique à **set-prd**, mais en cas de retour en fin de ligne précédente <dest> est placé en fin de ligne.
set-olgn <dest> <orig> : affecter à <dest> la position de fin de ligne précédente de <orig>, à une abscisse *très grande*.
set-mOLGN <dest> <orig> <tab> : identique à **set-olgn**, mais <dest> est placé en fin de ligne.
set-pred <dest> <orig> : identique à **set-prd**, mais un éventuel retour à la ligne précédente n'est pas envisagé.

set-nxt <dest> <orig> : affecter à <dest> le successeur dans le Buffer de <orig>.
set-nlgn <dest> <orig> : affecter à <dest> la colonne de début de ligne suivante de <orig>.
set-succ <dest> <orig> : identique à **set-nxt**, mais un éventuel passage à la ligne suivante n'est pas envisagé.

set-col <dest> <orig> <tab> : affecter à <dest> la colonne de numéro <tab> de la ligne de <orig>
set-eol <dest> <orig> <tab> : affecter à <dest> la colonne de fin de ligne de la ligne de <orig>, à une abscisse *très grande*.
set-mEOL <dest> <orig> <tab> : identique à **set-eol**, mais <dest> est placé en fin de ligne.
set-bol <dest> <orig> <tab> : affecter à <dest> la colonne de début de ligne de la ligne de <orig>.

affectation de la page écran

La page écran utilise deux variables globales beg (début) et end (fin).

set-end : affecter à end sa valeur, compte tenu de la valeur de beg, de la taille du Buffer et de la taille de la fenêtre.
set-beg <num> : affecter à beg sa valeur, <num> lignes devant sa valeur actuelle, compte tenu de la taille du Buffer.

affectation du curseur

set-deb <dest> <info-lst> : affecter à <dest> la position de début du premier lien de <info-lst>.
set-fin <dest> <info-lst> : affecter à <dest> la position de fin du premier lien de <info-lst>.

info-debx <info-lst> : retourne le champ x de la position de début du premier lien de <info-lst>.
info-deby <info-lst> : idem, sur le champ y.
info-debcol <info-lst> : idem, sur le champ col.
info-deblgn <info-lst> : idem, sur le champ lgn.
info-finx <info-lst> : identique à **info-debx**, sur la position de fin du premier lien de <info-lst>.
info-finy <info-lst> : idem, sur le champ y.
info-fincol <info-lst> : idem, sur le champ col.
info-finlgn <info-lst> : idem, sur le champ lgn.

info-debp <x> <lgn> <info-lst> : retourne la valeur "vraie" si <x> <lgn> correspond à la position de début du premier lien de <info-lst>.
info-finp <x> <lgn> <info-lst> : identique à **info-debp**, sur la position de fin du premier lien de <info-lst>.
info-deb-eq <info-lst1> <info-lst2> : retourne la valeur "vraie" si <info-lst1> et <info-lst2> une même position de début du premier lien.
info-fin-eq <info-lst1> <info-lst2>
identique à **info-debx**, sur la position de fin du premier lien.

comparaison

glo-< <1> <2> : retourne la valeur "vraie" si <1> devance strictement <2> dans le Buffer – le test porte sur les champs entiers x et y; les paramètres peuvent donc être de représentation immédiate.
glo-<= <1> <2> : idem, égale ou devance strictement.
glo-> <1> <2> : idem, précède strictement.
glo->= <1> <2> : idem, égale ou précède strictement.

2. les utilitaires

fonctions externes

cmde : appel de l'éditeur: la commande demande un nom de fichier, auquel sera associé le premier Buffer à l'entrée sous l'éditeur.
re-cmde ou le macro-caractère **%** : rappel de l'éditeur, après une interruption (commande ^X-s).

cmde-init : initialisation; à appeler une première fois avant tout appel de l'éditeur.

fonctions de saisies

cmde-int : interprétation des commandes.
cmde-rsp : interprétation d'un caractère
cmde-err : commande inexistante (bell).
cmde-tyi : saisie d'un caractère.
cmde-get : recherche de la fonction attachée au caractère.
cmde-call : appel de la fonction attachée au caractère.

initialisation

cmde-init-0 *; variables globales*
initialisation des variables globales:

cmde-beg	ancien début de page
cmde-end	ancienne fin de page
cmde-curdeb	ancien début de curseur
cmde-curfin	ancienne fin de curseur
flag-cmde-key	drapeau, vrai si en cours de définition de macro
flag-cmde-keydef	drapeau, vrai si la macro est définie
flag-cmde-T	drapeau, vrai si mode physique
flag-kil	drapeau, vrai si la dernière commande a touché au Kill Buffer
ynkinfo-lst	liste des liens du Kill Buffer
flag-jst	drapeau, vrai si mode justification
flag-rch	drapeau, vrai si mode recherche
cmde-rechbuf	dernier motif recherché
flag-cmde-S-T	drapeau, vrai si le dernier motif cherché contient des expressions régulières
flag-msg	drapeau: 0 : pas de message sur la ligne de contrôle, 1 : un message à détruire sur la ligne de contrôle, 2 : un message à laisser, le drapeau prend la valeur 1.
flag-aff	drapeau, vrai si localement on affiche le curseur physique
grsp	liste des commandes qui composent la macro
cmde-fin	drapeau, vrai si fin de session

fonctions d'appel des commandes
Elles ont toutes un paramètre fonctionnel; on les utilise pour grouper la mise à jour des variables globales.

cmde-DEPABS <cmde-fun> : déplacement absolu du curseur – la fenêtre est recalée sur le curseur.
cmde-DEPFEN <cmde-fun> : déplacement absolu de la fenêtre – le curseur est recalé sur la fenêtre.
cmde-DEPREL-nxt <cmde-fun> : déplacement relatif du curseur vers l'avant.
cmde-DEPREL-prd <cmde-fun> : déplacement relatif du curseur vers l'arrière.
cmde-CMDE <cmde-fun> : pas d'effet immédiat (^L, ^X, ESC, ...).
cmde-STAT <cmde-fun> : pas de déplacement.
cmde-ZONE <cmde-fun> : pas de déplacement des curseurs – déplacement dans les fenêtres, ...
cmde-MSG <cmde-fun> : envoi d'un message.
cmde-MODIF <cmde-fun> : modification
cmde-LECT <cmde-fun> : lecture.

fonctions de mise à jour des variables globales

cmde-nxt : positionnement, dans un déplacement vers l'avant.
cmde-prd : positionnement, dans un déplacement vers l'arrière.
cmde-ind : positionnement, dans un déplacement dans un sens indéterminé.

cmde-app-ON : entrer en mode «ajout».
cmde-app-OFF : sortir du mode «ajout».
cmde-eob-ON : entrer dans l'état «fin de Buffer».
cmde-eob-OFF : sortir de l'état «fin de Buffer».
cmde-aff-ON : entrer dans l'état «afficher localement le curseur physique».
cmde-aff-OFF : sortir de l'état «afficher localement le curseur physique».
cmde-msg-ON : entrer dans l'état «un message apparaît sur la ligne de contrôle».
cmde-msg-OFF : sortir de l'état «un message apparaît sur la ligne de contrôle».

cmde-phy : afficher ou non le curseur physique.

cmde-mod-ON <flag-fen> : modifier l'attribut de modification du Buffer pour la valeur "vraie" – <flag-fen> indique s'il y a lieu d'afficher la *forme visuelle* de l'attribut.
cmde-kil-ON : entrer dans l'état «la dernière commande touche au Kill Buffer».
cmde-kil-OFF : sortir de l'état « la dernière commande touche au Kill Buffer ».

exit-bell : fin de commande: la commande est refusée.
exit-Abort : fin de commande: la commande est abandonnée (^G).
Note: dans les deux cas on sort de la boucle ERREUR d'interprétation de la commande.
append-err : la commande est refusée parce qu'on est en mode «ajout».
mark-err : la commande est refusée parce que la marque n'a pas été posée dans le Buffer.

fonctions auxiliaires

affiche-cur : afficher le curseur du Buffer courant.
affiche-ecr : afficher l'écran.

affiche-ban : afficher la barre d'information (bannière) de la fenêtre courante.
affiche-ban-part : idem, incomplètement, pour "faire patienter" l'utilisateur.
affiche-ban-init : idem, incomplètement, à l'ouverture d'une fenêtre.
affiche-nom-abs-mem : afficher le nom absolu du Buffer dans la barre d'information.
aff>ban : afficher les holophrastes (logique et physique) – fenêtre de Buffer.

bell : bell.
aff>phy : afficher le curseur physique.
eff>phy : effacer le curseur physique.
aff>scr : afficher la zone de déroulement (scrolling).
aff>msg-cursor : afficher le message "<cursor>" sur la ligne de contrôle.
eff>msg-cursor : identique, effacer.
eff>end-of-line : effacer la fin de la ligne.
clear-scroll <y1> <y2> : effacer la zone de déroulement entre les lignes d'écran <y1> et <y2>.
clear-region <y1> <y2> : effacer la région d'écran entre les lignes d'écran <y1> et <y2>.

cst-1 : numéro de ligne d'écran du début de fenêtre (def=1).
cst-20 : idem, de la fin de fenêtre (def=22).
cst-21 : idem, de la barre d'information de la fenêtre (def=23).
cst-22 : idem, de la ligne de contrôle (def=24).
cst-16 : nombre de lignes d'écran déroulées par un saut de page (def=18).
cst-8 : idem, par un demi-saut de page (def=10).
cst-60 : abscisse de "<cursor>" sur la ligne de contrôle (def=72).

min-80 <x> : taille de la ligne physique d'écran – minimum de <x> et d'une constante (def=79).
test-80 <x> : test, vrai si <x> dépasse la constante de taille de la ligne physique d'écran (def=79).
test-min-80 <x> : idem, si <x> égale la constante (def=79).
max-80 : taille de la ligne logique – constante (def=200).
test-max-80 <x> : test, vrai si <x> dépasse la constante de taille de la ligne logique (def=180).

liste-separ <car> ... : liste des caractères séparateurs de mots, en plus de caractères « », «"», «'», «(», «)», «,».

fonctions de dialogue

print-msg <msg> ... : afficher la liste de messages <msg> sur la ligne de contrôle.
print-bell <msg> ... : idem, et sonner.
print-prompt <msg> ... : idem, et appeler **get-prompt**.
print-ack <msg> ... : idem, et demander un acquittement "y".

print-msg-mem <x> <stg> : afficher la liste de codes ASCII <stg> à partir de la position <x> sur la ligne de contrôle – il s'agit des noms absolus de Buffer: si le nom est trop long, il est élidé.
print-msg-write <msg> <stg> : afficher le message <msg> et le nom de fichier <stg> – le nom de fichier est précédé du chemin absolu d'accès au fichier sous UNIX.

eff>msg : effacement du message de la ligne de contrôle; si l'on est placé dans un *mode*, il se réaffiche l'indicateur de mode correspondant – sélection, «ajout», justification, recherche.

get-prompt : saisir un paramètre sur la ligne de contrôle; caractères spéciaux:
^G abandon
^M acquittement – une entrée vide est un cas d'erreur
^H ou DEL détruire le caractère précédent
^U détruire les caractères précédents, jusqu'au premier caractère "/" rencontré.
get-prompt-mem : idem, pour la saisie d'un nom de Buffer – on a encore les caractères spéciaux
^I compléter jusqu'au premier Buffer-Edit
" " acquittement.

print-sequence <lst-msg> : afficher et saisir une liste de messages et de réponses pour les commandes assistées; syntaxe des messages:
Stg saisir une chaîne de caractères
Buf saisir un nom de Buffer
Ok saisir un acquittement (caractère "y")
<stg> afficher la chaîne de caractères <stg>
(<stg>...) afficher les chaînes de caractères <stg>
(selectq (<stg1>...<stgN>) (<car1>...<carN>) [<stg>])
saisir un choix: <stg1>...<stgN> sont des messages, <car1>...<carN> sont des codes ASCII valides, <stg> est un message optionnel.
Par exemple:
paramètre <lst-msg>:
((selectq ("Copy" "Move") (99 109) "to Kill Buffer")
" " (selectq ("Buffer" "Region") (98 114)) Ok)
suite des affichages:
Copy / Move to Kill Buffer ? *; réponse: c*
Copy to Kill Buffer : Buffer / Region ? *; réponse: r*
Copy to Kill Buffer : Region Ok ? *; réponse: y et "retour chariot"*

MORE <instr> ... : évaluer les instructions <instr> avec des affichages du type "more".
fic-MORE <nom-fic> : afficher le contenu du fichier <nom-fic> par un affichage du type "more".
more-print <stg> : afficher la chaîne de caractères <stg> par un affichage du type "more": si les lignes de la fenêtre sont toutes remplies, on attend un acquittement de l'utilisateur pour poursuivre l'affichage:
^M afficher la ligne suivante,
" " (blanc) afficher la page suivante,
DEL quitter.

3. l'écran

On regroupe ici les fonctions d'écran:
- détermination des zones du curseur, du Buffer local, ...
- affichage du contenu d'une fenêtre de Buffer.

On utilise les variables globales suivantes:

- log : curseur logique (représentation symbolique),
- phy : curseur physique (représentation symbolique),
- curdeb : début du curseur (représentation par une liste),
- curfin : fin du curseur (représentation par une liste);

- loghol : holophraste logique,
- reghol : holophraste physique (holophraste du Buffer local),
- bufhol : holophraste minimal du Buffer (initialement 1);

- reginfo-lst : liste des liens du Buffer local,
- bufinfo-lst : liste des liens du Buffer courant,
- ficinfo-lst : liste des liens du Buffer de la fenêtre.
(note: le Buffer courant peut être plus petit que le Buffer de la fenêtre si l'on a appelé récursivement l'éditeur par la commande ESC-r)

Note: en-dehors des variables log et phy les données sont calculées d'après la liste des liens reginfo-lst, bufinfo-lst et ficinfo-lst. L'appel récursif sur le Buffer local courant est donc réalisé par empilement de la seule variable bufhol.

On utilise dans la suite la terminologie suivante:
– *données du curseur logique*: log et loghol,
– *données du curseur physique*: phy,
– *données du curseur*: curdeb et curfin,
– *données du Buffer local*: reghol et reginfo-lst,
– *données du Buffer*: bufhol et bufinfo-lst;
– *données de la page*: beg et end;

détermination des zones

log>reg : calcule les données du Buffer local, du curseur, du curseur physique d'après celles du curseur logique.
log>cur : calcule les données du curseur, du curseur physique d'après celles du curseur logique.
log>phy : calcule les données du curseur physique d'après celles du curseur logique.

log>fen : calcule les données de la page d'après celles du curseur logique.
fen>log : calcule les données du curseur logique d'après celles de la page.

affichage de la page

aff>ecr : affiche la page.
aff>ecr-part <beg> <end> <pge> : affiche la page entre les lignes <beg> et <end> du Buffer – <pge> est le numéro de ligne du Buffer de la première ligne de la fenêtre.
eff>ecr-page <beg> <end> <num> : efface les <num> dernières ligne de la page <beg> <end>.
re-aff>ecr : réaffiche la page.

aff>cur : affiche le curseur.
eff>cur : efface le curseur.
re-aff>cur : réaffiche le curseur.

aff>zon : affiche la zone sélectionnée en mode sélection.
sel-eff>ecr : efface la zone affichée en sortie de mode sélection.

affichage interne
Les fonctions ont toutes la visibilité des variables
- zondeb, zonfin: la zone à afficher,
- zontab: l'abscisse du début de ligne du Buffer local.

aff>ecr-zone <beg> <end> <tab> <pge> : afficher la zone:
- entre <beg> et <end>,
- avec l'abscisse <tab> de début de ligne du Buffer local,
- la première ligne d'écran ayant le numéro <pge> dans le Buffer.

eff>ecr-zone <beg> <end> <tab> <pge> : idem, effacer la zone.
aff>zon-zone <beg> <end> <deb> <fin> <tab> <pge> : idem, afficher la portion du curseur <deb> <fin> contenue dans la page <beg> <end>.
eff>zon-zone <beg> <end> <deb> <fin> <tab> <pge> : idem, effacer la portion.
aff>tab-zone <beg> <end> <deb> <fin> <tab> <pge> : idem, afficher les points hors de la *zone d'accès*.
eff>tab-zone <beg> <end> <deb> <fin> <tab> <pge> : idem, effacer les points.
aff>ecr-init : initialiser les variables zon...

aff> <beg> <end> <tab> <pge> : afficher de <beg> à <end> en inverse vidéo.
eff> <beg> <end> <tab> <pge> : idem, en standard.
aff>tab <beg> <end> <tab1> <tab2> <pge> : afficher de <beg> à <end> en inverse vidéo entre les abscisses <tab1> et <tab2>.
eff>tab <beg> <end> <tab1> <tab2> <pge> : idem, en standard.

4. les fenêtres

les données globales

fen-Init-0 : initialisation des variables globales des fenêtres:
- fen-log : liste des fenêtres sauvegardées
- var-22 : numéro de ligne d'écran de la ligne de contrôle (def=24)
- var-60 : numéro d'abscisse d'écran du message "<cursor>" (def=72)
- fen-nbr : nombre de fenêtres ouvertes
- fen-nomlog : nom logique de la fenêtre ouverte (entre 0 et fen-nbr -1)

fen-log>old : sauvegarde de la fenêtre active.
fen-old>log : restauration de la fenêtre, de nom fen-nomlog.
fen-new>log : ouverture d'une nouvelle fenêtre, concernant les données du Buffer.

données globales d'une fenêtre
- fen-nommem : nom du tampon ouvert dans la fenêtre.
- fen-typmem : type de la fenêtre:
  - aux : fenêtre du Kill Buffer,
  - alm : fenêtre d'alarme,
  - env : fenêtre de Buffer-Edit,
  - rep : fenêtre de Buffer.

var-1 var-20 var-21 var-16 var-8 var-11 données de cadrage.

reginfo-lst reghol données du Buffer local.
phyx phyy phycol phylgn données du curseur physique.
curdeb curfin données du curseur.
beg end données de la page.

les fonctions externes

ACTIVER <log> : activer la fenêtre <log>.
REACTIVER <log> : idem, et réafficher la fenêtre.
DESACTIVER : désactiver la fenêtre active fen-nomlog.
RELIRE <log> : relire les données de la fenêtre <log> – complet.
RELIRE-log <log> : relire les données de la fenêtre <log> – les données de la fenêtre uniquement.
ECRIRE : écrire les données de la fenêtre active fen-nomlog – complet.
ECRIRE-log : écrire les données de la fenêtre active fen-nomlog – les données de la fenêtre uniquement.

VISITER <mem> <typ> : ouvrir une fenêtre pour visiter le tampon <mem> de type <typ>. Si la fenêtre active est celle du Kill Buffer ou une fenêtre d'alarme, elle est écrasée. Sinon, la fenêtre qui suit immédiatement la fenêtre active est écrasée. Sinon, c'est-à-dire s'il n'y a qu'une seule fenêtre ouverte, celle-ci est coupée en deux.
REENTRER : réentrer dans la fenêtre après un écrasement par la commande ^X-! – celle-ci n'étant pas une vraie fenêtre, elle est détruite dès qu'on sort du dialogue avec l'interpréteur.
AFFICHER <mem> <typ> : afficher le tampon <mem> de type <typ> sur la totalité de l'écran.
REAFFICHER <mem> <typ> : idem, avec le message "Exiting to Buffer ...".
fen-DELETE : détruire la fenêtre active.

deb-ALARME <mem> : la fenêtre courante devient une fenêtre d'alarme sur le tampon <mem>.
fin-ALARME : la fenêtre courante d'alarme est fermée

fen-APPARAIT <mem> <typ> : test, vrai si la fenêtre sur le tampon <mem> de type <typ> est ouverte.
fen-FAMILY <mem> : retourne la liste des fenêtres qui sont de la famille du tampon <mem> – les fenêtres ouvertes sur des tampons englobés au sens large par <mem>.

les fonctions internes

PLACER : placer une nouvelle fenêtre sur l'écran; retourne "faux" si la fenêtre active a été coupée et "vrai" si c'est la fenêtre suivante qui a été activée.
REDUIRE-haut <num> <log> : réduire la taille de la fenêtre <log> de <num> lignes par le haut.
REDUIRE-bas <num> <log> : idem, réduire par le bas.
ETENDRE-haut <num> <log> : idem, étendre par le haut.
ETENDRE-bas <num> <log> : idem, étendre par le bas.

REDUIRE <num> <flag bas> : réduire la fenêtre active, par le bas si <flag-bas> est vrai et réciproq.
AGRANDIR <num> <flag-bas> : idem, agrandir.
ETENDRE <log> <log1> <log2> : étendre la fenêtre <log> de <log1> à <log2>.
COUPER : couper la fenêtre active en deux: retourne "faux" en cas d'échec, c'est-à-dire si l'une des deux fenêtres coupées est d'une taille strictement inférieure à 2 lignes.

les fonctions auxillaires

fen-pred <log> : fenêtre précédente de <log> – la fenêtre précédant la première fenêtre à l'écran est la dernière.
fen-succ <log> : fenêtre suivante.
fen-fstq <log> : première fenêtre d'écran.
fen-lstq <log> : dernière fenêtre d'écran.

fen-fstp <log> : test, vrai si <log> est la première fenêtre d'écran.
fen-lstp <log> : idem, dernière fenêtre d'écran.

fen-put <symb> <val> <prop> : sauvegarder sur le symbole <symb> pour la propiété <prop> la valeur <val> – <symb> = fen-log; <prop> = le nom de la fenêtre; <val> = les données de la fenêtre.
fen-get <symb> <prop> : retourner la valeur sur le symbole <symb> de la propriété <prop>.
fen-rem <symb> <prop> : supprimer sur le symbole <symb> la propriété <prop>.
fen-ins <symb> <prop> : insérer sur le symbole <symb> la propriété <prop>.

5. la modification

les fonctions externes : insertion

caract-ins <flag-fen> <flag-O> <deb> <num> <hol> <info-lst> <car>... : insérer des caractères:
- <flag-fen> drapeau, vrai si une fenêtre est ouverte sur le Buffer modifié
- <flag-O> drapeau, vrai si l'insertion est réalisée *devant* le curseur (commande ^O)
- <deb> variable de début d'insertion
- <num> nombre d'insertions répétées
- <hol> holophraste d'insertion
- <info-lst> liste des liens d'insertion sous l'holophraste considéré
- <car>... liste de caractères <car>... à insérer.

caract-ins-LF <flag-fen> <flag-O> <deb> <hol> <info-lst> : insérer un retour à la ligne: pour les paramètres cf. **caract-ins**.

caract-ins-buf <flag-fen> <flag-O> <deb> <hol> <info-lst> <cpyhol> <cpydebcol> <cpydebign> <cpyregx> <cpyfincol> : insérer une zone de Buffer:
pour les premiers paramètres cf. **caract-ins**
- <cpyhol> holophraste de la zone copiée
- <cpydebcol> colonne de début de la zone copiée
- <cpydebign> ligne de début de la zone copiée
- <cpyregx> abscisse du premier caractère du Buffer local de la zone copiée
- <cpyfincol> colonne de fin de la zone copiée.

caract-ins-ynk <flag-O> <cpyhol> <cpydebcol> <cpydebign> <cpyregx> <cpyfincol> : insérer une zone de Buffer dans le Kill Buffer:
pour les paramètres cf. **caract-ins-buf**.

caract-ins-fic <flag-fen> <flag-O> <deb> <hol> <info-lst> <grep> : insérer un fichier:
pour les premiers paramètres cf. **caract-ins**
- <grep> variable tampon de la représentation du texte lu dans le fichier.

les fonctions externes : suppression

caract-cursup <flag-fen> <deb> <fin> <hol> <info-lst> : détruire une zone de Buffer:
- <flag-fen> drapeau, vrai si une fenêtre est ouverte sur le Buffer modifié
- <deb> variable du début de la zone à détruire
- <fin> variable de la fin de la zone à détruire
- <hol> holophraste de la zone à détruire
- <info-lst> liste des liens de la zone à détruire sous l'holophraste considéré.

les fonctions externes : écriture sur fichier

caract-write <hol> <info-lst> <flag-app> : écrire le Buffer local d'holophraste <hol> de liste des liens <info-lst>:
- <flag-app> drapeau, vrai si l'écriture est fait en *ajout* dans le fichier.

caract-write-flat <hol> <info-lst> : écrire *à plat* le Buffer local d'holophraste <hol> de liste des liens <info-lst>.

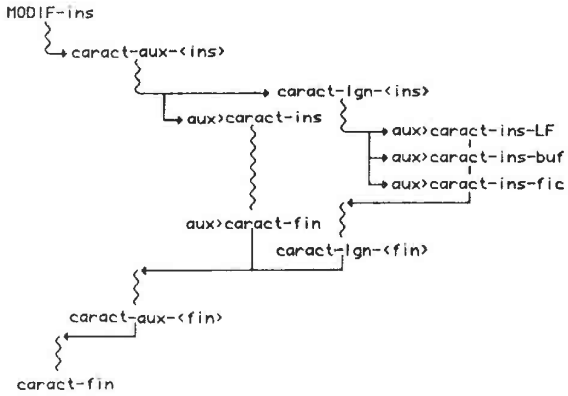
les fonctions externes : détermination des zones

det>caract-sup <fin> <num> <hol> <info-lst> : retourne le début de la zone qui contient <num> curseurs et qui finit sur <fin>, sous l'holophraste <hol> dans la liste des liens <info-lst> (commande ^D par exemple).
det>caract-del <deb> <num> <hol> <info-lst> : identique à **det>caract-sup**, retourne la fin de la zone qui contient <num> curseurs et qui débute sur <deb> (commande ^H par exemple).

det>region <deb> : retourne sur les variables auxiliaires auxdeb et auxfin les début et fin de la *région* – la zone comprise entre le curseur courant et la marque.

les fonctions internes : insertion

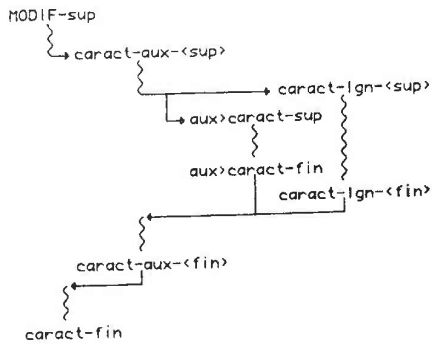
Le graphe d'appel des fonctions est le suivant:



MODIF-ins : commande d'insertion.
caract-fin : mise à jour des variables globales du curseur, de la marque, ..., et affichage.
caract-aux-<ins> et **-<fin>** : modification sur les curseurs qui suivent le point de modification – sur une même *vraie* ligne.
caract-ign-<ins> et **-<fin>** : insertion éventuelle de lignes.
aux>caract-ins et **-fin** : insertion de caractères.
aux>caract-LF et **-fin** : insertion d'un caractère de retour à la ligne LF.
aux>caract-buf et **-fin** : insertion d'un Buffer.
aux>caract-fic et **-fin** : insertion d'un fichier.

les fonctions internes : suppression

Le graphe d'appel des fonctions est le suivant:



MODIF-sup : commande de suppression.
caract-fin : cf. précédemment.
caract-aux-<sup> et **-<fin>** : cf. **caract-aux-<ins>**.
caract-ign-<sup> et **-<fin>** : suppression éventuelle de lignes.
aux>caract-sup et **-fin** : suppression de caractères, sur une même ligne.

les fonctions internes : boucles

Les fonctions de boucles traitent des *vraies* lignes du Buffer – cf. précédemment, §2.2.2. sur les commandes ESC-n et ESC-p.

boucle-ins <num> : paramètres implicites: auxcol auxlgn auxhol auxinfo-lst;
reçoit:
<num> nombre de caractères insérés
auxcol auxlgn colonne et ligne de début d'insertion
auxhol auxinfo-lst holophraste et liste des liens du point d'insertion;
renvoie:
auxcol auxlgn colonne et ligne du dernier point modifié par l'insertion
auxhol auxinfo-lst holophraste et liste des liens du dernier point modifié par l'insertion;
le dernier point modifié par l'insertion tient compte de la *vraie* ligne: c'est le premier caractère de début de ligne qui ne dépend pas du point de début d'insertion.

boucle-del <num> : paramètres implicites: auxcol auxlgn auxhol auxinfo-lst;
identique à **boucle-ins**, <num> étant le nombre de caractères supprimés.

boucle-sel <num> : paramètres implicites: auxcol auxlgn auxhol auxinfo-lst;
identique à **boucle-ins**, concernant une sélection validée (commande ^X-^@); <num> est l'abscisse de début du curseur défini par la sélection – qu'on cherche à rendre maximal.

boucle-flt : paramètres implicites: auxcol auxlgn auxhol auxinfo-lst;
identique à **boucle-ins**, concernant l'aplatissement d'un curseur (commande ^X- (blanc)).

fonctions internes : boucles générales

boucle-COPY <hol>
<origcol> <origlgn> <origregx> <origfincol>
<destcol> <destlgn> <destx> <destregx> <destinfo-lst> :
copie de l'origine:
<hol> holophraste, <origcol> colonne de début de la zone copiée, <origlgn> ligne de début, <origregx> abscisse de début du Buffer local, <origfincol> colonne de fin;
vers le destinataire:
<destcol> colonne du point d'insertion, <destlgn> ligne, <destx> abscisse, <destregx> abscisse de début du Buffer local, <destinfo-lst> liste des liens du point d'insertion.

boucle-NEXT <num> <hol>
<origcol> <origlgn> <origregx> <origfincol> :
retourne le début de ligne de la <num>-ième *vraie* ligne suivant l'origine.

boucle-ERASE <num> <hol>
<origcol> <origlgn> <origregx> <origfincol> :
retourne le <num>-ième curseur suivant l'origine.

fonctions auxilliaires : mise à jour des variables

caract-clean-log <num> : paramètre implicite: auxlgn;
met à jour la variable de curseur logique log:
<num> nombre de lignes insérées
auxlgn première *vraie* ligne non modifiée.
caract-clean <num> <loc> : paramètre implicite: auxlgn;
identique à **caract-clean-log**, sur une variable quelconque <loc>.

caract-aux <num> : détermine la nouvelle page et les nouveaux curseurs.
caract-fin <hol> : détermine si le curseur logique doit être considéré comme étant en fin de ligne.
caract-curfin : retourne le numéro de ligne qui suit le curseur.

aff>caract-fin : réalise l'affichage après modification; variables globales:
MODIF-tot drapeau, vrai si l'on réaffiche entièrement la fenêtre
MODIF-y1 numéro de la première ligne du Buffer à réafficher
MODIF-y2 numéro de la dernière ligne du Buffer à réafficher.
aff>caract <num> <deby> <finy> : met à jour les variables globales de réaffichage:
<num> nombre de lignes modifiées
<deby> numéro de la première ligne modifiée
<finy> numéro de la dernière ligne modifiée.
aff>ligne-mod <y1> <y2> : affichage des lignes, entre les numéros <y1> et <y2>.

6. la structure de données du tampon

6.1. les fonctions des «textes»

Les définitions reprennent les termes des «textes» de l'*éditeur ligne*.

nomq <def> : retourne le nom du «texte» <def>.
envq <def> : retourne l'environnement des définitions locales du «texte» <def>.
repq <def> : retourne la représentation du «texte» <def>.

6.2. les fonctions de représentation

liens entre tampons
rac-mem <mem> [<x>] : si le paramètre <x>:
- est absent: retourne le tampon "père" du tampon <mem>, c'est-à-dire le tampon dont la liste des tampons locaux contient le tampon <mem>;
- est présent: modifie le champ par la nouvelle valeur <x>.

fic-mem <mem> [<x>] : identique, pour le nom du fichier auquel est attaché le tampon <mem>.
len-mem <mem> [<x>] : identique, pour la longueur à l'écran demandée pour l'affichage du nom complet du tampon <mem> – le nom du tampon et celui du fichier.
buf-mem <mem> [<x>] : identique, pour le nom du tampon <mem>.
lst-buf-mem <mem> [<x>] : identique, pour la liste des noms de tampons englobants.
Note: la liste des noms des tampons englobants est un lien *physique*: si le nom de la racine change, la modification est automatiquement répercutée sur les tampons englobés.

attributs de modification
mod-mem <mem> [<x>] : identique, pour l'attribut de modification relatif au seul tampon – sa représentation ou la liste des tampons locaux.
mod-abs-mem <mem> [<x>] : identique, pour l'attribut de modification absolu du tampon – il est mis à vrai dès qu'un attribut d'un tampon englobé est vrai.

champs d'un tampon
lst-mem <mem> [<x>] : identique, pour la liste des tampons locaux du tampon <mem> – la liste des tampons englobés dans le tampon <mem>.
env-mem <mem> [<x>] : identique, pour le champ d'environnement du tampon <mem> – champ non rempli.
rep-mem <mem> [<x>] : identique, pour la représentation «textuelle» du tampon <mem>.
flag-rep-mem <mem> [<x>] : identique, pour la représentation sous "forme éditée" du tampon <mem> – le champ est initialement mis à faux; il devient vrai dès que le Buffer du tampon est édité dans une fenêtre, et contient alors l'information attachée sous l'éditeur au Buffer du tampon.

test sur les tampons
global-mem <mem> : test, vrai si le tampon est l'environnement global.

fonctions du Buffer-Edit
On donne au Buffer-Edit une "forme éditée" d'après la liste des tampons locaux qu'il représente.

cons-rep <lst-mem> : retourne la "forme éditée" de la liste des tampons <lst-mem> pour le Buffer-Edit concerné.
cons-rep-lgn <mem> : retourne la "forme éditée" de la ligne du Buffer-Edit relative au tampon <mem>.

env-def <y> : retourne, s'il se peut, le *tampon désigné* correspondant à la ligne <y> du Buffer-Edit.

6.3. les fonctions de construction

inserer-mem <def> <rac-mem> <fic-mem> : insérer le «texte» <def> dans la liste des tampons locaux de <rac-mem>, attaché au fichier de nom <fic-mem>.
inserer-old-mem <mem> <def> <fic-mem> : remplacer l'ancien tampon <mem> par le «texte» <def> attaché au fichier de nom <fic-mem>.
inserer-nom-mem <mem> <nom-buf> : remplacer dans le tampon <mem> le nom par <nom-buf>.
inserer-mve-mem <mem> <rac-mem> <nom-buf> : déplacer le tampon <mem> dans la liste des tampons locaux de <rac-mem>, sous le nom <nom-buf>.

supprimer-mem <mem> : supprimer le tampon <mem> de la liste des tampons locaux.

6.4. les fonctions sur les noms

old-fic-mem <fic-mem> <lst-mem> : retourne le tampon attaché au fichier de nom <fic-mem>, dans la liste des tampons <lst-mem> ou dans l'un de leurs descendants; retourne conventionnellement *nil* si le tampon n'est pas trouvé

new-nom-buf <fic-mem> : retourne le nom d'un tampon construit à partir du nom de fichier <fic-mem>.
nom-buf <mem> : retourne le nom simple du tampon <mem> sous forme de chaîne de caractères, éventuellement complété par un numéro d'ordre en cas de *surcharge* des noms de tampons ("<2>", "<3>", ...) – le champ **buf-mem** n'est pas une chaîne de caractères.
restore-nom-buf <mem> : retourne le nom simple du tampon <mem> sous forme de chaîne de caractères, sans complément enventuel.
nom-abs-mem <mem> : retourne le nom complet du tampon <mem> sous forme de chaîne de caractères.

cons-buf-mem <stg> : retourne le champ **buf-mem** construit à partir de la chaîne de caractères <stg>.
inserer-buf-mem <buf-mem> <lst-mem> : retourne le champ **buf-mem** inséré dans la liste des tampons locaux <lst-mem> – la fonction tient compte de la possible *surcharge* des noms de tampon.

buf-RECHERCHE <mem> <msg> : recherche assistée d'un nom de Buffer:
<mem> tampon de départ pour la recherche – c'est la racine d'où débute la recherche si le nom est donné sous forme relative, c'est-à-dire que le premier caractère n'est pas "/"; autrement le nom est donné sous forme absolue et la recherche est entreprise à partir de l'environnement global.
<msg> message précédant le nom du tampon recherché.
Sortie de la recherche:
- le tampon existe: la fonction retourne la valeur "vraie", et la variable auxiliaire aux est une liste de deux valeurs:
  - le tampon retenu,
  - le type du tampon retenu (rep = Buffer ou env = Buffer-Edit).
- le tampon est nouveau: la fonction retourne la valeur "faux", et la variable auxiliaire aux est une liste de trois valeurs:
  - le tampon "père", dans lequel insérer le nouveau tampon local,
  - la chaîne de caractère qui nomme le nouveau tampon,
  - le type du tampon retenu.
- abandon (commande ^G): la commande est annulée.

7. les tampons

les données globales

mem-init-0 : initialisation des variables globales des tampons:
lst-fen-nommem liste des tampons de l'environnement global; initialement il est vide
lst-fen-auxmem liste des tampons auxiliaires: ce sont les tampons "Command execution", "Kill Buffer" et "Help".

lst-fen-auxmem <num> : retourne le tampon auxiliaire de numéro:
0 "Command execution" – dialogue avec l'interpréteur Lisp
1 "Kill Buffer"
2 "Help" – ouvert dans une fenêtre d'alarme

mem-log>old : sauvegarde du tampon actif.
mem-old>log : restauration du tampon, de nom fen-nommem de type fen-typmem.
mem-new>log : ouverture d'un nouveau tampon, concernant les données du Buffer.

mem-new>rep <grep> : construit la "forme éditée" de la représentation <grep> – retourne la liste des liens dans la variable globale ficinfo-lst.

données globales d'un tampon

cmde-lst	liste des commandes accessibles dans une fenêtre ouverte sur le tampon
flag-mark	drapeau, vrai si la marque a été posée; s'il est vrai:
	mrk variable de position de la marque dans le tampon
flag-sel	drapeau, vrai si le mode "Sélection" est actif; s'il est vrai:
	select-stat état de la sélection – avant / arrière
	seldeb variable de position du début de sélection
	selfin variable de position de la fin de sélection
	cmde-log curseur logique au début de la sélection
	cmde-hol holophraste logique au début de la sélection
	Note: on peut quitter une fenêtre dans laquelle le mode "Sélection" est actif
flag-phy	drapeau, vrai si le mode de déplacement "physique" est actif–sinon "logique"
flag-eob	drapeau, vrai si le curseur logique est en fin de Buffer
flag-app	drapeau, vrai si le mode "ajout" est actif
ficinfo-lst	liste des liens du Buffer global du tampon
bufinfo-lst bufhol	données du Buffer courant
lst-bufhol	liste des holophrastes bufhol empilés
logx logy logcol	loglgn données du curseur logique
loghol	holophraste logique.

les fonctions externes

RELIRE-mem <mem> <typ> : relire les données du tampon <mem> de type <typ> – les données du tampon uniquement.
ECRIRE-mem <mem> <typ> : écrire les données du tampon <mem> de type <typ> – les données du tampon uniquement.
ENTRER <mem> <typ> : entrer dans le tampon <mem> de type <typ>.
SORTIR : fermer le tampon de la fenêtre active.

les fonctions internes de service

INSERER <def> <rac-mem> <fic-mem> : insérer le «texte» <def> dans la liste des tampons locaux de <rac-mem>, attaché au fichier de nom <fic-mem>.
INSERER-old <mem> <def> <fic-mem> : remplacer l'ancien tampon <mem> par le «texte» <def> attaché au fichier de nom <fic-mem>.

INSERER-nom <mem> <nom-buf> : remplacer dans le tampon <mem> le nom par <nom-buf>.
INSERER-mve <mem> <rac-mem> <nom-buf> : déplacer le tampon <mem> dans la liste des tampons locaux de <rac-mem>, sous le nom <nom-buf>.

SUPPRIMER <mem> : supprimer le tampon <mem> de la liste des tampons locaux.
ANCETRE <mem> <rac-mem> : test, vrai si le tampon <mem> apparaît dans la liste des tampons locaux de <rac-mem>.

les fonctions internes sur les fichiers

fic-READ <nom-fic> <nom-buf> : paramètre implicite rendu: gdef:
lit dans le fichier de nom <nom-fic> le «texte» de nom par défaut <nom-buf>; retourne "vrai" si la lecture est correcte, et "faux" en cas d'erreur; retourne le «texte» lu dans la variable globale gdef.
fic-READ-VIDE <nom-buf> : paramètre implicite rendu: gdef:
retourne le «texte» vide (environnement et représentation vides) de nom <nom-buf> dans la variable globale gdef.
fic-READ-FLAT <nom-fic> <nom-buf> : paramètre implicite rendu: gdef:
lit dans le fichier de nom <nom-fic> le «texte plat» pour la représentation du «texte» de nom <nom-buf> – l'environnement est vide; retourne le «texte» dans la variable globale gdef.

fic-WRITE <nom-fic> <mem> : écrit dans le fichier de nom <nom-fic> le tampon <mem>.
fic-WRITE-FLAT <nom-fic> <mem> : écrit dans le fichier de nom <nom-fic> le «texte plat» relatif au tampon <mem>.
fic-WRITE-ENV <nom-fic> <mem> : écrit dans le fichier de nom <nom-fic> la *hiérarchie des tampons* du tampon <mem>.
fic-WRITE-MORE <nom-fic> <mem> : affiche dans une fenêtre d'alarme la *hiérarchie des tampons* du tampon <mem>.

les fonctions internes sur les fichiers : boucles

boucle-READ <hol>
<destcol> <destlgn> <destx> <destregx> <destinfo-lst> :
lecture vers le destinataire:
<hol> holophraste, <destcol> colonne de début d'insertion du texte lu, <destlgn> ligne, <destx> abscisse, <destregx> abscisse de début du Buffer local, <destinfo-lst> liste des liens

boucle-WRITE <hol>
<origcol> <origlgn> <origregx> <origfincol> :
écriture de l'origine:
<hol> holophraste, <origcol> colonne de début du texte à écrire, <origlgn> ligne, <origregx> abscisse de début du Buffer local, <origfincol> colonne de fin.

boucle-WRITE-FLAT <hol>
<origcol> <origlgn> <origregx> <origfincol> :
écriture *à plat* de l'origine.

8. la configuration

La configuration concerne les codes graphiques des terminaux VT100 ou z29.

clear-screen : effacer l'écran.
cursor-pos <x> <y> : placer le curseur en <x> <y> – 1ère colonne = 1, 1ère ligne = 1.

set-reverse : entrer en mode inverse Vidéo.
reset-reverse : sortir du mode.
set-underline : entrer en mode souligné.
reset-underline : sortir du mode.
set-underline-protect : entrer en mode souligné = protégé.
reset-underline-protect : sortir du mode.
set-reverse-underline : entrer en mode inverse Vidéo et souligné.
reset-all : sortir de tout mode.

cursor-on : afficher le curseur (physique).
cursor-off : masquer le curseur (physique).
set-scroll <y1> <y2> : définir la zone de déroulement (scroll) entre les lignes <y1> et <y2> – <y2> ≥ <y1> +1.
reset-scroll : revenir à la zone de déroulement initiale: (**set-scroll** 1 24).

insert-Dn : insérer une ligne, avec déroulement vers le bas.
delete-Up : supprimer une ligne, avec déroulement vers le haut.
scroll-Up <num> <y> : dérouler l'écran de <num> lignes vers le haut à partir de la ligne <y>.
scroll-Dn <num> <y> : identique, vers le bas.

erase-entire-line [<y>] : effacer entièrement la ligne, de numéro <y>.
erase-beg-of-line [<x> <y>] : effacer le début de ligne, depuis la position <x> <y>.
erase-end-of-line [<x> <y>] : effacer la fin de ligne, depuis la position <x> <y>.

prin-cursor-on <c> [<x> <y>] : afficher le caractère de code ASCII <c> à la position <x> <y> en inverse Vidéo.
prin-cursor-off <c> [<x> <y>] : identique, en affichage normal.

9 les modes

9.1. la recherche

les variables globales

flag-rch	drapeau, vrai si la recherche est active.
cmde-rch	liste des commandes du mode "recherche".
flag-cmde-S-T	drapeau, vrai si la dernière recherche a été faite avec des expressions régulières ER.
cmde-rechbuf	dernier motif recherché.
flag-cmde-init	drapeau, vrai si la recherche débute.
flag-rech-Ok	drapeau, vrai si la recherche a réussi.
flag-rech-Syn	drapeau, vrai si la syntaxe du motif recherché est correcte.
flag-rech-S	drapeau, vrai si la recherche est faite vers l'avant.
flag-rech-S-T	drapeau, vrai si la recherche est faite avec des expressions régulières ER.
rechbuf	motif recherché – liste de codes ASCII
rechcom	identique, sous forme compilée.
rech	variable de position de début de recherche courant.
cmde-rech	identique, ancienne position.

rech-INIT <flag> : initialisation de la recherche, <flag> vrai si la recherche est avant.
rech-EXIT : fin de la recherche.
rech-SUIV : recherche suivante (commande ^S ou ^R).

la recherche

rech-RECH : recherche du motif suivant.
rech-TEST-S : position de début de recherche suivant, en recherche avant.
rech-TEST-R : identique, en recherche arrière.

rech-egal-S <buf> <num> : test, vrai si le motif <buf> de taille <num> est trouvé à la position courante.
rech-egal-S-T <com> <num> : test, vrai si le motif compilé <com> de taille <num> est trouvé à la position courante – cas de la recherche avec les expressions régulières ER.
rech-egal-tst <loc> : test, vrai si le motif simple <loc> est trouvé – cas ER.

la compilation

La compilation retourne:
- cas simple: flag-rech-Syn = vrai,
- cas ER: flag-rech-Syn = vrai et rechcom = forme compilée, si la compilation est correcte.

rech-comp <buf> : compilation du motif <buf>; retourne:
- cas simple: flag-rech-Syn = vrai,
- cas ER: flag-rech-Syn = vrai et rechcom = forme compilée, si la compilation est correcte.

rech-length <com> : longueur du motif, en nombre de codes ASCII, du motif compilé <com>.

Syntaxe – cas ER

<forme compilée> ::= *liste de* <champ>
<champ> ::= <valeur> <drapeau>
<drapeau> ::= *vrai si le caractère suivant est "•"*
<valeur> ::= (94) *pour "^" ; début de ligne*
(36) *pour "$" ; fin de ligne*
(46) *pour "." ; caractère quelconque*
(98) *pour "\b" ; caractère séparateur de mots*
(119) *pour "\w" ; caractère composant de mot*
(92 . <car>) *pour "\<car>" ; caractère non interprété*
(91 <c1> . <c2>) *pour "[<c1>-<c2>]" ; intervalle de codes ASCII*

9.2. la sélection

les variables globales

flag-sel	drapeau, vrai si la sélection est active.
cmde-sel	liste des commandes du mode "sélection".
seldeb	variable de position de début de sélection.
selfin	variable de position de fin de sélection.
select-stat	drapeau, vrai si la sélection est faite en avant.

cmde-sel-INIT : initialisation de la sélection, par défaut en avant.
cmde-sel-EXIT : fin de la sélection – pas de validation.
cmde-sel-VALID : validation de la sélection.

fonction auxilliaire

select-num <x0> <regdebx> : retourne l'abscisse de validation de la sélection:
<x0> est l'abscisse du point de début de sélection,
<regdebx> est l'abscisse de début du Buffer local.
On efface le plus possible de caractères blancs, en respectant les alignements.

9.3. la justification

les variables globales

flag-jst	drapeau, vrai si la justification est active.
cmde-jst	liste des commandes du mode "justification".
flag-cmde-init	drapeau, vrai si la justification débute, sur une ligne donnée.
lst-just	liste des types de justification, relatif ou absolu, empilés.
flag-just-A	drapeau, vrai si la justification est de type absolu.
just-old-tab	tabulation, avant toute modification de la ligne – la *tabulation* est le nombre de caractères blancs en début de ligne.
just-new-tab	nombre de caractères blancs insérés – nouvelle tabulation attendue.
just-dif-tab	différence: nouvelle - ancienne tabulation. just-dif-tab = just-old-tab - just-new-tab.

cmde-just-INIT : initialisation de la justification, par défaut de type absolu.
cmde-just-EXIT : fin de la justification.
cmde-just-RECIN <flag> : empilement d'un niveau de justification, <flag> vrai si la justification est de type absolu.
cmde-just-RECOUT : dépilement d'un niveau de justification.

fonctions auxilliaires

just-tabloc <x> <col> : retourne la tabulation de la colonne <col> d'abscisse <x>.
just-init : initialisation des variables, dans le cas d'une modification autre que la validation (" ").
just-init-apply : initialisation des variables, dans le cas d'une validation (" ").
just-aff : modification et affichage.

9.4. l'appel récursif

les variables globales

buthol	holophraste du Buffer courant.
lst-buthol	liste des holophrastes des Buffers empilés.

EMPILER-hol : empiler un nouveau Buffer.
DEPILER-hol : dépiler un Buffer.

9.5. la recherche sur les noms

les variables globales

flag-alm	drapeau, vrai si une fenêtre d'alarme a été ouverte – cas ambigus de la recherche.
rac-mem	tampon "racine" des noms relatifs.
flag-abs	drapeau, vrai si le nom est absolu – premier caractère = "/".
stg	variable tampon du dialogue – chaîne de caractères.
lst-buf-mem	liste des noms de tampon relative à la chaîne de caractères traitée stg.
nb-buf-mem	nombre moins un des noms de lst-buf-mem – chaque nom est séparé par un caractère "/".
num-buf-mem	nombre de caractères du dernier nom de lst-buf-mem.

buf-mem	nom d'un tampon.
liste-globale	liste des tampons dont le nom satisfait le motif de lst-buf-mem sur les nb-global premiers noms.
nb-global	nombre maximal de noms de la liste lst-buf-mem satisfait par des tampons.
liste-p-globale	sous-liste de liste-globale des tampons dont le dernier nom satisfait le motif du dernier nom de lst-buf-mem sur les num-global premiers caractères.
num-global	nombre maximal de caractères du dernier nom de lst-buf-mem satisfait par des tampons de liste-globale.
typ-global	type du nom – env si le dernier caractère est "/" et rep sinon.

buf-RECHERCHE <mem> <msg> : recherche sur les noms de tampons, à partir de la "racine" <mem> avec le message <msg>.
buf-rech-OLD <mem> <typ> : le nom retenu existe, du tampon <mem> de type <typ>.
buf-rech-NEW <rac-mem> <buf-mem> <typ> : le nom retenu est nouveau, sur le tampon "père" <rac-mem> de nom <buf-mem> de type <typ>.
buf-rech-ABORT : abandon de la commande.

la recherche

buf-rech-nil : recherche sur un motif vide – vide ou "/" = l'environnement global.
buf-rech-32 : recherche après la frappe du caractère blanc.
buf-rech-13 : recherche après la frappe du caractère "retour chariot".

les erreurs

buf-rech-AMB <liste-mem> : ambiguïté, sur la liste des tampons <liste-mem>.
buf-rech-AMB-/ <liste-mem> : identique, on affiche les noms de Buffers-Edits.
buf-rech-ERR <nb> : erreur, le chemin donné ne peut être suivi après le <nb>-ième nom.
buf-rech-Alarm : ouverture éventuelle de la fenêtre d'alarme.
buf-rech-DEL-num <nb> <num> <buf-mem> : erreur, on conserve les <nb> premiers noms de lst-buf-mem et les <num> premiers caractères de <buf-mem>.
buf-rech-DEL-tab <nb> <buf-mem> : erreur, on conserve les <nb> premiers noms de lst-buf-mem et le nom <buf-mem>.
buf-rech-DEL-cpy : erreur, on conserve le nom de la racine rac-mem.

les boucles

buf-rech-egal <lst-buf-mem> : paramètres implicites rendus: nb-global liste-global;
retourne nb-global maximal et la liste des tampons liste-global le satisfaisant.
buf-rech-partielle <liste-mem> <buf-mem> :
paramètres implicites rendus: num-global liste-p-global;
retourne num-global maximal et la sous-liste des tampons liste-p-global de liste-global le satisfaisant.

buf-rech-part-max <liste-mem> : paramètres implicites rendus: num-global buf-mem;
retourne num-global maximal et buf-mem satisfait – cas nécessairement d'erreur.
buf-rech-part-egal <liste-mem> <buf-mem> : paramètre implicite rendu: liste-p-globale;
retourne la sous-liste des tampons liste-p-global de liste-global satisfaisant <buf-mem>.

les fonctions auxiliaires

eqbuf-mem <buf1> <buf2> : test, vrai si les noms de tampon <buf1> et <buf2> sont égaux;
l'égalité est vraie si:
- les noms sont égaux,
- les numéros de tampons sont égaux, ou alors le numéro de tampon de <buf1> est indéfini – c'est-à-dire qu'il est nul.

eqnumbuf-mem <buf1> <buf2> : retourne le nombre de caractères identiques dans <buf1> et <buf2> -- le numéro compte pour trois caractères.

num-buf-mem <buf-mem> : retourne le nombre de caractères de <buf-mem>.

num-rech-msg : affichage du message et du nom construit, sur la ligne de contrôle.

num-rech-nom-abs <lst-buf-mem> <num> : retourne le nom absolu, sous forme de liste de codes ASCII , construit sur <lst-buf-mem> et le numéro de tampon <num>.

les fonctions d'analyse des noms

analyse <stg> : retourne la forme de "type" lst-buf-mem de la chaîne de caractères <stg>.

Syntaxe des noms

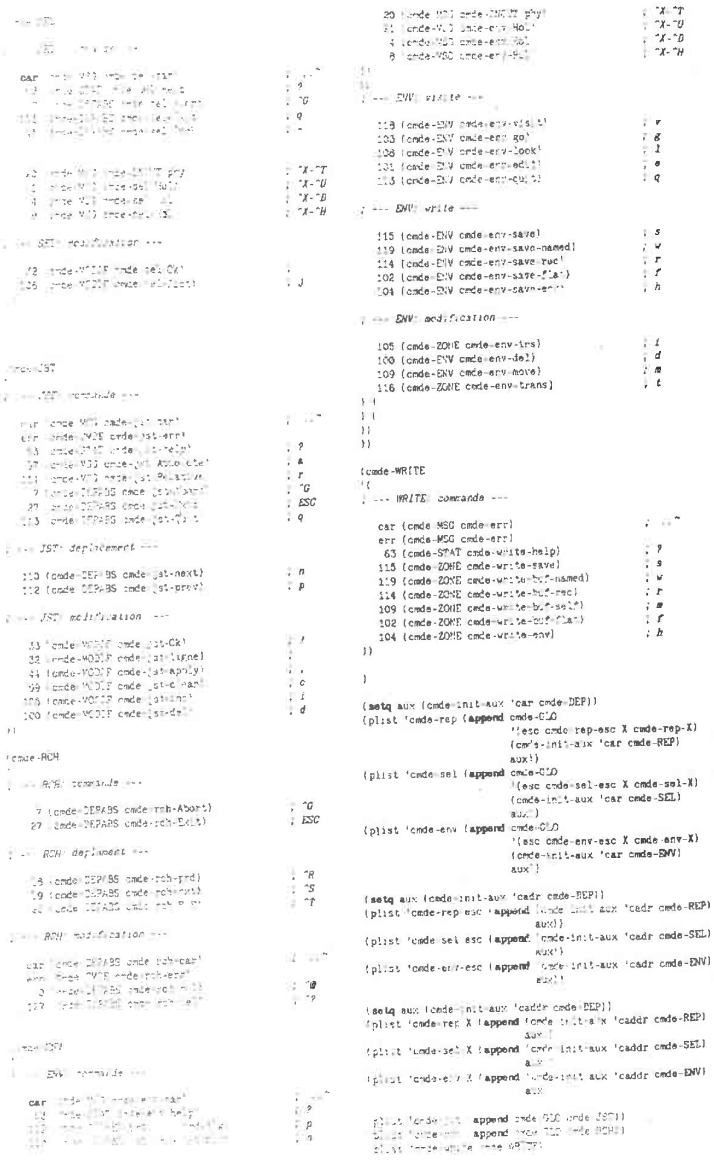
```
<stg>   ::= <exp>                   ; nom relatif
          | <exp> "/" <stg>         ; nom absolu si <exp> est vide
<exp>   ::= <nom>                   ; numéro du tampon indéfini: <num> = 0
          | <nom> "<" <num> ">"     ; numéro du tampon défini
```

10. les commandes

On indique pour chaque commande la *fonction d'appel de la commande* et le paramètre fonctionnel spécifique de la commande.

voir pages suivantes


```
de cmde init

let

 [illegible]

   [illegible]
    12 (cmde-[illegible] cmde-[illegible])            ; ^L
    21 (cmde-[illegible] cmde-[illegible])            ; ^U

 cmde-DEP
  '(
 ; --- DEP: commande ---
  (
   err (cmde-MSG cmde-err)
    20 (cmde-CMDE cmde-dir-T)                         ; ^T
    27 (cmde-CMDE cmde-dir-esc)                       ; ESC
    24 (cmde-CMDE cmde-dir-X)                         ; ^X
 ) (
   car (cmde-MSG cmde-err)                            ; ESC- ..~
   err (cmde-MSG cmde-err)
    24 (cmde-STAT cmde-help)                          ; ESC-^X
 ) (
   car (cmde-MSG cmde-err)                            ; ^X- ..~
   err (cmde-MSG cmde-err)
    33 (cmde-STAT cmde-command)                       ; ^X-!
    12 (cmde-STAT cmde-redraw)                        ; ^X-^L
    40 (cmde-STAT cmde-key-deb)                       ; ^X-(
    41 (cmde-STAT cmde-key-fin)                       ; ^X-)
     5 (cmde-STAT cmde-key-exec)                      ; ^X-^E
 ))
 ; --- DEP: deplacement fenetre ---
 ((
 ) (
    26 (cmde-STAT cmde-fen-page)                      ; ESC-^Z
    61 (cmde-MSG cmde-noline)                         ; ESC-=
 ) (
   122 (cmde-STAT cmde-fen-agrandir)                  ; ^X-z
    26 (cmde-STAT cmde-fen-reduire)                   ; ^X-^Z
    49 (cmde-STAT cmde-fen-1)                         ; ^X-1
 ))
 ; --- DEP: deplacement hors fenetre ---
 ((
 ) (
 ) (
   112 (cmde-STAT cmde-fen-prd)                       ; ^X-p
   110 (cmde-STAT cmde-fen-nxt)                       ; ^X-n
   100 (cmde-STAT cmde-fen-del)                       ; ^X-d
    50 (cmde-STAT cmde-fen-2)                         ; ^X-2
   115 (cmde-ZONE cmde-quit-S)                        ; ^X-s
     3 (cmde-ZONE cmde-quit-C)                        ; ^X-^C
 ))
 ; --- DEP: deplacement dans fenetre ---
 ((
     0 (cmde-DEPABS cmde-mark)                        ; ^@
    18 (cmde-DEPABS cmde-prd-search)                  ; ^R
    19 (cmde-DEPABS cmde-nxt-search)                  ; ^S
     1 (cmde-DEPABS cmde-prd-lign)                    ; ^A
     5 (cmde-DEPABS cmde-nxt-lign)                    ; ^E
    22 (cmde-DEPFEN cmde-nxt-page)                    ; ^V
    26 (cmde-DEPFEN cmde-nxt-scro)                    ; ^Z
     2 (cmde-DEPREL-prd cmde-prd-curs)                ; ^B
     6 (cmde-DEPREL-nxt cmde-nxt-curs)                ; ^F
    16 (cmde-DEPREL-prd cmde-prd-clgn)                ; ^P
    14 (cmde-DEPREL-nxt cmde-nxt-clgn)                ; ^N
 ) (
    27 (cmde-DEPABS cmde-phy-zone)                    ; ESC-ESC
    60 (cmde-DEPABS cmde-prd-buf-zone)                ; ESC-<
    62 (cmde-DEPABS cmde-nxt-buf-zone)                ; ESC->
    44 (cmde-DEPABS cmde-fen-haut)                    ; ESC-,
    46 (cmde-DEPABS cmde-fen-bas)                     ; ESC-.
    33 [illegible]                                    ; ESC-!
    97 (cmde-DEPABS cmde-prd-cur-zone)                ; ESC-a
   101 (cmde-DEPABS cmde-nxt-cur-zone)                ; ESC-e
    91 [illegible]                                    ; ESC-[
    93 [illegible]                                    ; ESC-]
   118 [illegible]                                    ; ESC-v
   122 [illegible]                                    ; ESC-z
   112 [illegible]                                    ; ESC-p
   110 [illegible]                                    ; ESC-n
    98 [illegible]                                    ; ESC-b
   102 [illegible]                                    ; ESC-f
    16 [illegible]                                    ; ESC-^P
    14 [illegible]                                    ; ESC-^N
 ) (
    61 (cmde-DEPABS cmde-goline)                      ; ^X-=
    24 (cmde-DEPABS cmde-mark-Exch)                   ; ^X-^X
 ))
 ; --- DEP: write ---
 ((
 ) (
    23 (cmde-ZONE cmde-write)                         ; ESC-^W
 ) (
    19 (cmde-ZONE cmde-write-save)                    ; ^X-^S
    23 (cmde-ZONE cmde-write-buf-named)               ; ^X-^W
    18 (cmde-ZONE cmde-write-buf-rec)                 ; ^X-^R
    13 (cmde-ZONE cmde-write-buf-self)                ; ^X-^M
     6 (cmde-ZONE cmde-write-buf-flat)                ; ^X-^F
   104 (cmde-ZONE cmde-write-more-env)                ; ^X-h
 ))
 ; --- DEP: visit ---
 ((
 ) (
    22 (cmde-ZONE cmde-visit)                         ; ESC-^V
 ) (
    22 (cmde-ZONE cmde-visit-file)                    ; ^X-^V
   102 (cmde-ZONE cmde-visit-flat)                    ; ^X-f
   118 (cmde-ZONE cmde-visit-buffer)                  ; ^X-v
   101 (cmde-ZONE cmde-visit-self)                    ; ^X-e
   113 (cmde-ZONE cmde-visit-quit)                    ; ^X-q
     2 (cmde-ZONE cmde-visit-pop)                     ; ^X-^B
    25 (cmde-ZONE cmde-visit-ynk)                     ; ^X-^Y
 ))
 ))

 (cmde-REP
  '(
 ; --- REP: read ---
 ((
 ) (
 ) (
     9 (cmde-MODIF cmde-fic-ins)                      ; ^X-^I
   105 (cmde-MODIF cmde-buf-ins)                      ; ^X-i
    15 (cmde-ZONE cmde-read-old)                      ; ^X-^O
   114 (cmde-ZONE cmde-read-file)                     ; ^X-r
 ))
 ; --- REP: commande ---
 ((
    30 (cmde-DEPABS cmde-append)                      ; ^^
 ) (
 ) (
    20 (cmde-MSG cmde-INOUT-phy)                      ; ^X-^T
     0 (cmde-MSG cmde-IN-sel)                         ; ^X-^@
    10 (cmde-DEPABS cmde-IN-jst)                      ; ^X-^J
    21 (cmde-ZONE cmde-HOL-Up)                        ; ^X-^U
     4 (cmde-ZONE cmde-HOL-Dn)                        ; ^X-^D
     8 (cmde-ZONE cmde-HOL-St)                        ; ^X-^H
 ;  114 (cmde-DEPABS cmde-reg-edit)                   ; ^X-r
 ;  113 (cmde-DEPABS cmde-reg-quit)                   ; ^X-q
 ))
 ; --- REP: modification ---
 ((
   car (cmde-MODIF cmde-car)                          ; ..~
    13 (cmde-MODIF cmde-LF)                           ; ^M
    15 (cmde-MODIF cmde-LF-O)                         ; ^O
   127 (cmde-MODIF cmde-prd-del)                      ; ^?
     8 (cmde-MODIF cmde-prd-del)                      ; ^H
     4 (cmde-MODIF cmde-nxt-del)                      ; ^D
 ) (
   104 (cmde-MODIF cmde-prd-del-mot)                  ; ESC-h
   100 (cmde-MODIF cmde-nxt-del-mot)                  ; ESC-d
 ) (
    32 (cmde-MODIF cmde-flatten)                      ; ^X-
 ))
 ; --- REP: kill buffer ---
 ((
    11 (cmde-MODIF cmde-kill)                         ; ^K
    23 (cmde-MODIF cmde-reg-kill)                     ; ^W
    25 (cmde-MODIF cmde-ynk)                          ; ^Y
 ) (
   107 (cmde-LSTT cmde-cpy)                           ; ESC-k
    11 (cmde-STAT cmde-copy-ynk)                      ; ESC-^K
     1 (cmde-STAT cmde-copy-ynk-buf)                  ; ESC-^A
 ) (
   107 (cmde-LSTT cmde-buf-cpy)                       ; ^X-k
    11 (cmde-MODIF cmde-buf-kill)                     ; ^X-^K
```



Références

Bibliographie

[AbG 85] **J.R. Abrial, A. Guillon**, *Un outil de conception de logiciels*, Journées SM90, CNET ADI, 4-6 dec 85, Versailles (pp 871-880)
Présentation d'un outil de spécification par le formalisme mathématique.
Règles de production et inférences, syntaxe très souple.

[Abr 84] **J.R. Abrial**, *Spécifier ou comment matérialiser l'abstrait*, TSI, 1984 (pp 201-219)
Présentation d'un exemple de construction de programme par Spécification/Réalisation

[AbS 83] **Abelson, Sussman**, *Structure and interpretation of computer programs*, MIT Press, 1983 (500 p.)
Présentation progressive et cohérente de Lisp, et plus généralement des modes de programmation dans les langages (Lisp est le support d'expression plus que le sujet du livre).

[AFQ 87] **F. Alexandre, J.-P. Finance, A. Quéré**, *SPES – a system for transforming logic programs*, Rapp. CRIN, Nancy, 87-R-090 (15 p.)
Outil de transformation: spécification par TAA -> programme fonctionnel (contrôlé, ≠ guidé).

[Agh 86] **Gul Agha**, *An Overview of Actors Languages*, ACM SIGPLAN Notices, Vol 21 No 10, oct 86 (pp 58-67)
Revue des langages d'acteurs.

[AnB 87] **P. Andrieu, P. Borras**, *PPML : Manuel d'utilisation (version 2.0)*, Rapp. Int., INRIA, Jan 87 (20 p.)
PPML : langage de spécification pour la décompilation des arbres abstraits sous Mentor.

[ArF 88] **G. Arango, P. Freeman**, *Application of Artificial Intelligence*, ACM SIGSOFT Software Engineering Notes, Vol 13 No 1, Jan 88 (pp 32-38)
Rapport du «4th International Workshop on Software Specification and Design, IEEE, Monterey, Ca, april 3-4 1987» – présentation des articles.

[Ars 79] **J.J. Arsac**, *Syntactic Source to Source Transforms and Program Manipulation*, Comm. of the ACM, Vol 22 No 1, Jan 79 (pp 43-54)
Expression mathématique rigoureuse de programmes pseudo-pascal, pour des transformations «à sémantique constante».

[Bac 78] **John Backus**, *Can Programming be liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, Comm. of the ACM, Vol 21 No 8, Aug 78 (pp 613-641)
Introduction du Langage fonctionnel FP (Functional Programming)
– FFP (Formal System for FP) + AST (Application State Transition Systems).

[Bad 83] **Scott Baden**, *Berckley FP User's Manual, 4.1*, July 27, 1983 (33 p.)
FP : langage fonctionnel initialement défini par Backus [Bac 78].

[Bal 81] **Robert Balzer**, *Transformational Implementation: An Example*, IEEE Trans. on Software Engineering, Vol 7 No 1, Jan 81 (pp 3-14)
Exemple des *8 reines*; transformation par schémas guidée par l'utilisateur + génération automatique du «document de développement» qui retrace les étapes de la transformation.

[Bar 87] **David Barstow**, *Artificial Intelligence and Software Engineering*, Proc. of the IEEE 9th International Conference on Software Engineering, march 30-apr 2 1987, Monterey, Ca (pp 200-211)
Défenseur de l'utilisation de l'IA en Génie Logiciel; gains chiffrés (approximatifs) dans les étapes du développement.

[BaS 86] **R. Bahlke, G. Snelting**, *Context-sensitive editing with PSG environments*, Lecture Notes in Computer Science (Goos et Hortmanis ed.), Advanced Programming Environments, Proceeding of an International Workshop, Trondheim, Norway, June 16-18 1986 (pp 26-38)
Générateur d'éditeur syntaxique + interpréteur + sémantique, par règles d'inférence.

[BBG 85] **M. Bidoit, B. Biebow, M.-C. Gaudel, C. Gresse, G. Guiho**, *Exception Handling: Formal Specification and Systematic Program Construction*, IEEE Trans. on Software Engineering, Vol 11, No 3, march 85 (pp 242-252)
Présentation d'ASSPRO = traduction d'une spécif. en PLUSS en un programme Ada.

[BDE 87] **D. Bert, P. Drabik, R. Echahed**, *Manuel de référence de LPG*, version 1.8, à paraître en tant que rapport LIFIA, 1987 (67 p.)
LPG : langage "nouvelle génération" : programmation fonctionnelle, types abstraits, généricité, + spécification, règles (preuve).

[Ber 87] **Philip A. Bernstein**, *Database System Support for Software Engineering – An Extended Abstract*, Proceeding of the IEEE 9th International Conference on Software Engineering, march 30-apr 2 1987, Monterey, Ca (pp 166-178)
Les documents gérés par un CASE (Computer Aided Software Engineering), les services requis.
+ biblio (109 ref.).

[BGG 83] **M. Bidoit, C. Gresse, G. Guiho**, *Caty*, Journées BIGRE 83, Le Cap d'Agde, 17-19 oct 83 (pp 110-122)
Approche par TAA + *schémas* = transformation de structures.

[Bid 87] **M. Bidoit et al**, *ASSPRO: en environnement de programmation intéractif et intégré*, TSI, Vol 6 No 1, 1987
Spécification par TAA en langage PLUSS. Génération de squelettes Ada.

[Bjø 87] **Dines Bjørner**, *On The Use of Formal Methods in Software Development*, Proceeding of the IEEE 9th International Conference on Software Engineering, march 30-apr 2 1987, Monterey, Ca (pp 17-29)
Défenseur d'une formalisation de l'activité de programmation; la méthode VDM = utilise un «graphe des dépendances» à tous les niveaux du développement.

[Blu 84] **Bruce I. Blum**, *Three Paradigms for Developing Information Systems*, Proc. of the 7th International Conference on Software Engineering, Orlando, Florida, march 26-29 1984 (pp 534-543)
Emploi d'un VHLL pour la génération automatique de programmes COBOL.

[BMP 86] **H.J. Barnard, R.F. Metz, A.L. Price**, *A Recommanded Practice for Describing Software Designs; IEEE Standards Project 1016*, IEEE Trans. on Software Engineering, Vol SE-12 No 2, feb 1986
Les documents à produire et leurs relations vis-à-vis des intervenants.

[Boe 81] **Barry W. Boehm**, *Les facteurs du coût du logiciel*, TSI, Vol 1 No 1, 1981, (pp 5-24)
Etude chiffrée des coûts du logiciel, et leurs causes dans le cycle de vie.

[Boe 86] **Barry W. Boehm**, *A Spiral Model of Software Development and Enhancement*, ACM SIGSOFT Software Engineering Notes, Vol 11 No 4, aug 86 (pp 14-24)
Modèle spiral du cycle de vie. Prototypes -> Réalisation.

[Boo 82] **Grady Booch**, *Naming Subprograms with Clarity*, ACM SIGPLAN Notices, Vol 17 No 1, Jan 1982 (pp 18-22)
Un complément lexical aux langages de programmation pour des appels de procédure lisibles – à suivre dans [Gro 82].

[Boo 86] **Grady Booch**, *Object-Oriented Development*, IEEE Trans. on Software Engineering, Vol 12 No 2, feb 86 (pp 211-221)
L'emploi d'une méthode orientée objets (MOO) appliquée à Ada.
Caractérisations d'un objet.

[Bro 85] **Manfred Broy**, *Structure Algebraic Specification of Backus' Functional Programming Language*, TSI, Vol 4 No 5, 1985 (pp 447-458)
Suite de [Bac 78], définition sous forme de TAA du langage FP.

[Cam 86] **John R. Cameron**, *An Overview of JSD*, IEEE Trans. on Software Engineering, Vol 12 No 2, feb 86 (pp 222-240)
Présentation de «Jackson System Development» (JSD), par des exemples.

[Cat 79] **James V. Catano**, *Poetry and Computers: Experimenting with the Communal Text*, Computers and the Humanities, Vol 13, 1979 (pp 269-275)
Expérience d'utilisation de l'«hypertexte» (Van Dam, cf [Nel 67]) pour l'enseignement de la critique littéraire d'un poème.

[CaW 85] **Cardelli, Wegner**, *On Understanding Types, Data Abstraction and Polymorphism*, ACM Computing Surveys, Vol 17 No 4, dec 85 (pp 471-522)
Le point sur les types, types abstraits de donnée et polymorphisme.
Définition informelle du langage *fun* qui supporte ces concepts.

[CCC 87] **Ceugniet, Chabrier, Chauvin, Derriau, Graf, Lextrait**, *Prototypage d'un Générateur d'Editeurs Syntaxiques Graphiques*, DESS ISI, Gen. Log. Info. Temps Réel, 13 mai 1987 (110 p.)
Langage de spécification SSL + Langage de description GSL.
Définition de la machine virtuelle graphique — gère boîtes et fenêtres.
+ implémentation et exemples à la fin.

[CDD 86] **D. Clement, J. et T. Despeyroux, L. Hascoet, G. Kahn,** *Specification in Natural Semantics*, GIPE : Intermediate Report CS-R8620 May 1986 (64 p.)
Spécification de la syntaxe abstraite (Metal) et de la sémantique dynamique (Typol). exemples : ASPLE, ML, ESTEREL.

[CGV 80] **P.Y. Cunin, M. Griffiths, J. Voiron,** *Comprendre la compilation*, Springer-Verlag, 1980
Aspects d'*implantation* et d'*optimisation* des «contextes d'évaluation» (sic).
Pb. d'optimisation abordé à la fin, dont pb. d'optimisation globale (dt les algo.).

[ChD 83] **J.J. Chabrier, J.C. Derniame,** *TYP: programming with abstract types*, TSI, Vol 1 No 4, 1983 (pp 267-274)
Langage exécutable de T.A., l'exécution peut être lancée dès la *spécification* du type (sans *représentation* associée).

[Che 84] **T. E. Cheatham,** *Reusability Trough Program Transformation*, IEEE Trans. on Software Engineering, Vol 10 No 5, sept 84 (pp 589-594)
ELI : langage de très haut niveau (VHLL)
+ PDS : env. de programmation.

[Cou 85] **G. Cousineau,** *ML : un langage fonctionnel typé*, Journées SM90 4-6 dec 85, CNET ADI, Versailles (pp 103-110)
Présentation du langage ML : mécanisme d'inférence des types, polymorphisme, définition de fonctionnelles.

[CrD 87] **S. Cruzlara Silva, J.C. Derniame,** *Yet another programming environment generator based on attribute grammars*, Rapp. CRIN, Nancy, 87-R-079
Présentation du système GEODE, environnement de programmation intégré défini par les grammaires attribuées.

[DDF 87] **V. Donzeau-Gouge, C. Dubois, P. Facon, F. Jean,** *Development of a programming environnement for SETL*, ESEC'87, Strasbourg, sept 87 (pp 23-36)
lgge SETL (VHLL) et environnement de développement SED.

[DeF 79] **J.-C. Derniame, J.-F. Finance,** *Types Abstraits de Données: spécification, utilisation et réalisation*, Ecole d'été de l'AFCET, Monastir, Rapp. CRIN, Nancy, 79-E-57 (200 p.)
Panorama sur les T.A. : définition, spécification, les T.A. dans les lgges.

[DeS 86] **N. Deslisle, M. Schwartz,** *Neptune: a Hypertext System for CAD Applications*, SIGMOD Record, Vol 15 No 2, June 1986, Proc. of SIGMOD'86 International Conference on Management of Data, Washington, D.C., may 28-30 1986 (pp 132-139)
CAD = Computer Aided Design. Historique des hypertextes, application à la documentation, spécialisation -> CASE.

[DHK 80] **V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang,** *Programming environments based on structured editors : the Mentor experience*, Rapport de recherche INRIA No 26, 1980 (14 p.)
éditeur de texte des programmes Pascal.

[Dia 84] **Jorge L. Diaz-Herrera,** *Pragmatic Problems with Step-wise Refinement Program Development*, ACM SIGSOFT Soft. Eng. Notes, Vol 9 No 2, april 1984 (pp 80-88)
Décomposition des algorithmes par *affinage* et non l'emploi (excessif) des procédures.

[DLP 79] **R.A. De Millo, R.J. Lipton, A.J. Perlis,** *Social processes and proofs of theorems and programs*, Comm. of the ACM, Vol 22 No 5, 1979 (pp 271-280)
cf. [ScS 83] : opposés à une formalisation de l'étape de Spécification.

[Dow 87] **Mark Dowson,** *ISTAR–An Integrated Project Support Environment*, ACM SIGPLAN Notices, Vol 22 No 1, Jan 1987, Proc. of the ACM SIGSOFT/ SIGPLAN Software Engineering Symp. on Practical Software Development Environment, Palo Alto, Ca, dec 9-11 1986 (pp 27-33)
environnement «orienté projet»; interface unique = editeur.
Nombreux outils proposés – gestion de projet, des ressources, B.D., + kit.

[DyS 85] **R. Dybvig, B. Smith,** *A Semantic Editor*, SIGPLAN Notices, Vol 20 No 7, July 85 (pp 74-82)
Editeur qui permet de réaliser des transformations de programmes écrits en ML, en *conservant* la sémantique des programmes, et en *conservant* dans la R.I. la modification.

[EnE 68] **D. C. Engelbart, W. K. English,** *A research center for augmenting human intellect*, AFIPS, Vol 33, 1968 (pp 395-410)
Un environnement "moderne" de programmation — réparti, souris, vidéo.
dt le souci de réaliser de l'édition structurée.

[Eng 78] **D. C. Engelbart,** *Toward integrated, evolutionary office automation systems*, Proc. Jt Engineering Management Conf., Denver, Colo., oct 16-18 1978, IEEE, New York (pp 63-68)
Présentation d'AUGMENT, environnement de «production de document».

[ENS 87] **G. Engels, M. Nagl, W. Schäfer,** *On the structure of structure-oriented editors for different applications*, ACM SIGPLAN Notices, Vol 22 No 1, Jan 1987, Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environment, Palo Alto, Ca, dec 9-11 1986 (pp 190-198)
Edition structurée par *graphes*, pour Modula-2 («programming in the small»), pour les interfaces («programming in the large»), pour du texte + des graphiques.

[Fea 82] **Martin S. Feather,** *A System for Assisting Program Transformation*, ACM Trans. on Programming Languages and Systems, Vol 4 No 1, Jan 1982 (pp 1-20)
Exemples d'emplois du système ZAP = transformations par groupements de transformations.

[Fea 87] **Martin S. Feather,** *Language Support for the Specification and Development of Composite Systems*, ACM Trans. on Programming Languages and Systems, Vol 9 No 2, april 1987 (pp 198-234)
Langage Gist de *spécification opérationnelle clos* (clos = on représente aussi l'environnement logiciel) – représentation par des ensembles ou des arbres (<-> CCS).

[FeM 80] **P.H. Feiler, R. Medina-Mora**, *An Incremental Programming Environment*, Dpt of Computer Science, Carnegie-Mellon Univ., Pa. 15213, April 1980 (23 p.)
Un environnement qui propose une R.I. commune pour Edition/Interprétation (= arbre syntaxique + code exécutable).

[FGM 87] **K. Futatsugi, J. Goguen, J. Meseguer, K. Okada**, *Parameterized Programming in OBJ2*, Proceeding of the IEEE 9th International Conference on Software Engineering, march 30-apr 2 1987, Monterey, Ca (pp 51-60)
langage OBJ2: «langage fonctionnel sur l'algèbre initiale» (égalité -> règles de réécriture).

[FGN 83] **G. Falquet, J. Guyot, L. Nerima**, *EM 2 : un environnement pour Modula 2*, Journées BIGRE 83, le Cap d'Agde, 17-19 oct 83 (pp 334-343)
environnement pour Modula 2.

[Fra 80] **Christopher W. Fraser**, *A Generalized Text Editor*, Comm. of the ACM, Vol 23 No 3, March 80 (pp 154-158)
Editeur pour: des textes, des directories, des exécutions symboliques, ... – toutes les commandes sont semblables, les effets de bord sont transparents.

[Fra 81] **Christopher W.Fraser**, *Syntax-Directed Editing of General Data Structures*, ACM SIGPLAN Notices, Vol 16 No 6, June 1981. Proceeding of the ACM SIGPLAN/SIGOA Symposium on text manipulation, Portland, Oregon, June 8-10 1981 (pp 17-21)
Présentation de l'éditeur *sds*. — cf. [Fra 80]

[Fre 87] **Peter Freeman**, *A Conceptual Analysis of the Draco Approach to Constructing Software Systems*, IEEE Trans. on Software Engineering, Vol 13 No 7, July 87
Transformation par *connaissance du domaine* (experts): on définit le langage de spécification et les transfo. dans le projet.

[GaF 84] **Gabriel, Frost**, *A Programming Environment on a Timeshared System*, SIGPLAN Notices, Vol 19 No 5, May 84 (pp 185-192)
Editeur de texte : E.

[GBB 87] **C. Godart, K. Benali, N. Boudjida, F. Charoy, J.-C. Derniame**, *Les bases de données sur le chemin du génie logiciel*, Journées d'étude AFCET «Des bases de données aux bases de connaissances», Ed. PSI 1987 (pp 37-58)
Projet ALF: SE et SGBD définis *conjointement* (≠ juxtaposés). Fondé sur la notion d'objet (-> propriétés, héritage).

[GJL 87a] **D. Gelernter, S. Jagannathan, T. London**, *Environments as First Class Objects*, 4th Annual ACM Symp on Principles of Programming Languages, Munich, W. Germany, 21-23 Jan 1987 (pp 98-110)
Symmetric Lisp: Lisp où l'on peut manipuler explicitement les environnements d'évaluation (= objets de «première classe»).

[GJL 87b] **D. Gelernter, S. Jagannathan, T. London**, *Parallelism, Persistence and Meta-Cleanliness in the Symmetric Lisp Interpreter*, Proceeding of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, St Paul, Minnesota, June 24-26, 1987 (pp 274-282)
Présentation de Symmetric Lisp, vu comme un *langage de commande* d'un environnement de programmation.

[GJL 87c] **D. Gelernter, S. Jagannathan, T. London, M. Day**, *A Symmetric Language*, Univ. of Yale, Techn. Report TR-568, oct 87 (37 p.)
Présentation de Symmetric Lisp + sémantique formelle du langage.

[GMT 86] **F. Gallo, R. Minot, I. Thomas**, *The Object Management System of PCTE as a Software Engineering Database Management System*, ACM SIGPLAN Notices, Vol 22 No 1, Jan 1987, Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments, Palo Alto, Ca, dec 9-11 1986
Project Common Tool Environment = développer une structure d'accueil pour un environnement plutôt qu'un environnement (-> Emeraude).

[Gog 84] **Joseph A. Goguen**, *Parameterized Programming*, IEEE Trans. on Software Engineering, Vol 10 No 5, sept 84 (pp 528-543)
Traits des langages de programmation supportant la "programmation paramétrée" OBJ : on retrouve l'essentiel de ces caractéristiques.

[Gog 86] **Joseph A. Goguen**, *Reusing and Interconnecting Software Components*, IEEE Computer, feb 86 (pp 16-28)
Présentation de LIL, langage de gestion de composants modulaires.

[Gol 83] **Adele Goldberg**, *The Influence of an Object-Oriented Language on the Programming Environment*, Proc. of the 1983 ACM Computer Science Conf., Orlando, Florida, feb 1983 (pp 35-54)
Présentation de Smalltalk-80.

[Gor 80] **R.D. Gordon**, *The Modular Application Customizing System*, IBM Systems Journal, Vol 19 No 4, 1980 (pp 521-541)
Génération automatique de programmes, à partir d'un questionnaire. -> programmes de gestion sur matériel de petite taille.

[Gra 86] **Anna Gram** *(collectif)*, *Raisonner pour programmer*, Dunod, 1986 (400 p.)
Décomposition fine du processus de conception/programmation. Cependant plus une analyse de comportement qu'une méthode de développement.

[Gro 82] **Lindsay J. Groves**, *Using Simple English Sentences to Call Procedures*, ACM SIGPLAN Notices, Vol 17 No 11, Nov 1982 (pp 31-38)
Suite de [Boo 82], les appels de procédures sont des phrases (en anglais).

[GuH 78] **J.V. Gutag, J.J. Horning**, *The Algebraic Specification of Abstract Data Types*, Acta Informatica, Vol 10, 1978 (pp 27-52)
Spécification Algébriques des TA (les TAA).

[GuJ 87] **J. Guyard, J.P. Jacquot**, *Systematic Structure-Oriented Program Edition*, Rapp. Univ. Nancy 87-R-003, 1987
Présentation de EDME, langage de définition d'algorithmes construit autour du langage MEDEE, selon la *méthode déductive* développée à Nancy.

[HCF 80] **C.F. Herot, R. Carling, M. Friedell, D. Kramlich**, *A Prototype Spatial Data Management System*, Computer Graphics, ACM SIGGRAPH, Vol 14 No 3, July 1980 (pp 63-70)
Interface graphique (icônes) pour la consultation de bases de données (pb: trouver des icônes "sémantiquement significatives").

[HeK 85] **Jan Heering, Paul Klint**, *Towards Monolingual Programming Environments*, ACM Trans. on Programming Languages and Systems, Vol 7 No 2, April 1985 (pp 183-213)
Environnement «monolingual» (≠ polyglote) pour : lgge de commande, lgge de programmation, lgge du debugger symbolique.

[HeK 86] **J. Heering, P. Klint**, *User Definable Syntax for Specification Languages*, GIPE, Esprit Project 348, Int. Report CS-R8620, May 1986 (40 p.)
Langage qui permet de spécifier entièrement (lexical + syntaxe) un langage de manière *simple* et efficace.

[Hen 86] **Peter B. Henderson**, *Data-Oriented Incremental Programming Environments*, Lecture Notes in Computer Science (Goos et Hortmanis ed.), Advanced Programming Environments, Proceeding of an International Workshop, Trondheim, Norway, June 16-18 1986 (pp 39-46)
Environnement incrémental orienté données: les attendus (dont ex: VisiProg).

[Hen 87] **P.R.H. Hendriks**, *Type-checking Mini-ML: an Algebraic Specification with User Defined Syntax*, GIPE, Esprit Project 348, 2nd annual review report, Jan 1987 (31 p.)
Contrôle de type dans le formalisme SDF - cf. [HeK 86].

[HES 87] **H. Harr, M. Evens, J. Spowl**, *Interpreting ABF — A language for Document Construction*, Proceeding of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, St Paul, Minnesota, June 24-26 1987 (pp 205-213)
Construction de documents juridiques par déf. de variables / utilisation.

[HIA 81] **M. Hammer, R. Ilson, T. Anderson, E. Gilbert, M. Good, B. Nianir, L. Rosenstein, S. Schoichet**, *The Implementation of Etude, An Integrated and Interactive Document Production System*, ACM SIGPLAN Notices, Vol 16 No 6, June 1981. Proceeding of the ACM SIGPLAN/SIGOA Symposium on text manipulation, Portland, Oregon, June 8-10 1981 (pp 137-146)
Maquette de formatteur de texte "WYSIWYG" et structuré (concept unique = région).

[HoT 86] **S. Horwitz, Tim Teitelbaum**, *Generating Editing Environments Based on Relations and Attributes*, Cornell Univ., ACM Trans. on Programming Languages & Systems, Vol 8 No 4, oct 86 (pp 577-608)
Editeur par grammaire attribuée : -> recherche rapide, efficace
+ algorithmes de mise en œuvre.

[HSE 84] **M. Hozumi, Y. Sekine, H. Ejima**, *A Method of Large-Scale Software Development*, Proc. of the 7th International Conference on Software Engineering, march 26-29 1984, Orlando, Florida (pp 520-527)
Présentation de l'étape de test avec l'emploi d'un exécutif spécifique qui évite l'intervention des *modules factices* en phase de mise au point.

[Jac 83] **J.-P. Jacquot**, *Etude d'un outil d'assistance à la conception méthodique de programmes*, thèse INPL, Nancy, 15 sept 83 (120 p.)
Maïday, environnement "confortable" de développement de petites applications -> le but est l'enseignement de *méthodes* de programmation.

[JoL 86] **J.-P. Jouannaud, P. Lescanne**, *la Réécriture*, TSI, Vol 5 No 6, 1986 (pp 433-452)
Panorama général sur l'état de l'art en matière de réécriture. + biblio.

[Jou 86] **Jouvelot**, *Designing New Language or New Language Manipulation Systems using ML*, ACM SIGPLAN Notices, Vol 21 No 8, Aug 86 (pp 40-52)
Syntaxe + sémantique décrites par des fonctions ML.

[Kah 87] **G. Kahn**, *Natural Semantics*, GIPE, Esprit Project 348, 2nd annual review report, Jan 1987 (19 p.)
Présentation de la «Sémantique Naturelle» (sémantique opérationnelle typée): le lgge Typol.

[KaL 84] **R.H. Katz, T.J. Lehman**, *Database Support for Versions and Alternatives of Large Design Files*, IEEE Trans. on Software Engineering, Vol 10 No 2, march 84
Mécanisme de gestion des versions.

[KLM 83] **G. Kahn, B. Lang, B. Mélèse, E. Morcos**, *Metal : a formalism to specify formalisms*, Séminaire Aussois, INRIA, 12-22 avril 1983 (pp 169-203)
langage Metal (méta-langage de Mentor).

[Knu 84] **Donald E. Knuth**, *Literate Programming*, Computer Journal, Vol 27 No 2, may 1984 (pp 97-111)
la *programmation littéraire*: une présentation agréable pour un lecteur humain.
cf. Rubrique *«Literate Programming»*, à partir de: CACM, Vol 30 No 7, July 1987.

[KoW 87] **E.E. Kohlbecker, M. Wand**, *Macro-by-examples: Deriving Syntactic Transformations from their Specification*, 4th Annual ACM Symp on Principles of Programming Languages, Munich, W. Germany, 21-23 Jan 87 (pp 77-84)
syntaxe des macros Lisp de "haut niveau".

[Lam 82] **Axel van Lamsweerde**, *Automatisation de la production de logiciels d'application: quelques approches*, TSI, Vol 1 No 6, 1982 (pp 475-494); Vol 2 No 1, 1983 (pp 5-20), Vol 2 No 2, 1983 (pp 81-93)
Panorama sur les transformations de programme: les différentes approches, des exemples, + biblio.

[Lam 87] **A. van Lamsweerde et al.**, *The Kernel of a Generic Software Development Environment*, ACM SIGPLAN Notices, Vol 22 No 1, Jan 1987. Proceeding of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symposium on Pratical Software Development Environments, Palo Alto, Ca, dec 9-11 1986 (pp 208-217)
Environnement *générique*, paramétré par la méthode.

[Lan 86] **B. Lang**, *The Virtual Tree Processor*, GIPE : Esprit Project 348, 3rd Report, Sept 86 (29 p.)
Manuel de référence du processeur d'arbres virtuel, noyau de l'environnement Mentor – ici la version Lisp.

[LaS 79] **H.C. Lauer, E.H. Satterthwaite**, *The Impact of Mesa on System Design*, 4th International Conference on Software Engineering, sept 79
langage Mesa + langage de connexion des composants.

[Leg 87] **Bruno Legeard**, *Prototypage de logiciels avec le langage PROLOG: Méthode et outils*, thèse de l'INSA de Lyon, 27 oct 87 (230 p.)
Présentation d'une méthodologie pour le prototypage en PROLOG.

[Lem 83] **M. Lemoine**, *Apport de la programmation automatique à la conception des environnements de programmation*, Journées BIGRE 83, le Cap d'Agde, 17-19 oct 83 (pp 597-604)
Présentations programmation automatique / environnement "classique" de programmation. Qualités du «bon atelier», cahier des charges d'un «bon atelier».

[LeT 87] **M. M. Lehman, W. M. Turski**, *Essential Properties of IPSEs*, ACM SIGSOFT Software Engineering Notes, Vol 12 No 1, Jan 87 (pp 52-55)
Revue des propriétés attendues d'un IPSE (Integrated Project [Programming] Support Environment).

[Lev 86] **Leon S. Levy**, *A Metaprogramming Method and Its Economic Justification*, IEEE Trans. on Software Engineering, Vol 12 No 2, feb 86 (pp 272-277)
Métaprogrammation = on écrit d'abord un outil de traduction automatique spécification -> programme, puis le projet est écrit et maintenu dans le langage de spécification.

[Lin 84] **Mark A. Linton**, *Implementing Relational Views of Programs*, ACM SIGPLAN Notices, Vol 19 No 5, may 84 (pp 132-140)
Maquette ou la R.I. des pgmes est une base de données: vues ≠ en fonction de la position où l'on est (<-> consulter des attributs ≠ de la B.D.).

[Lis 72] **B. H. Liskov**, *A design methodology for reliable software systems*, AFIPS, Vol 37, 1972 (pp 191-199)
Dans l'optique de tests (preuves) : -> hiérarchie par niveaux d'abstraction, programmation structurée.

[LiZ 75] **B. H. Liskov, S. N. Zilly**, *Specification Techniques for Data Abstraction*, IEEE Trans. on Software Engineering, Vol 1 No 1, march 75 (pp 7-19)
Les critères d'évaluation d'une méthode de spécification; application aux méthodes utilisées pour les T.A.

[Loy 83] **M. Loyer**, *Modularité, Composition des programmes et Gestion des Composants*, Séminaire Aussois, INRIA, 18-22 avril 83 (pp 357-364)
Les «bonnes règles» de la modularité, et ce qu'on est en droit d'attendre d'un environnement modulaire (à défaut d'un langage).

[LPS 87] **N. Levy, A. Piganiol, J. Souquières**, *Specifying with SACSO*, Proc. of the 4th Intrenational Workshop on Software Specification and Design, IEEE, Monterey, Ca, april 3-4 1987; Rapp. CRIN, Nancy, 87-R-002
Présentation du système SACSO: mise au point d'une spécification paramétrée par la méthode.

[MaR 73] **P.A. de Marneffe, D. Ribbens**, *Holon Programming*, (A. Günther et al. ed.), International Computing Symposium, 1973, North-Holland Publ. Co. 1974 (pp 67-71)
Programmation par «holon» (= «la partie du tout»): *«une solution bien structurée pour un programme bien intégré»* — cf. [Knu 84]

[Mar 79] **P. Marchand**, *Théorie des Graphes*, Cours du C1 d'Info., partie Algèbre chapitre 3, CRIN 79-E-20 (90 p.)
Cours sur la théorie des graphes – terminologie, exemples, algorithmes.

[Mat 84] **Yoshihiro Matsumoto**, *Some Experience in Promoting Reusable Software: Presentation in Higher Abstract Levels*, IEEE Trans. on Software Engineering, Vol 10 No 5, sept 84 (pp 502-513)
Comment les logiciels sont construits par niveaux successifs d'abstraction chez un industriel (Toshiba).

[Mel 83] **Bertrand Mélèse**, *Mentor Rapport*, Séminaire Aussois, INRIA, 12-22 avril 1983
Application de Mentor à l'édition de texte structuré.

[MeN 81] **R. Medina-Mora, D.S. Notkin**, *ALOE Users' and Implementors' Guide*, Dpt of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa. 15213, CMU-CS-81-145, nov 1981 (85 p.)
A Language Oriented Editor – un élément du projet GANDALF.

[MeN 87] **B. Meyer, J.M. Nerson**, *Cépage User's Manual (version 1.0)*, Interactive Software Engineering, may 1987 (18 p.)
Cépage : éditeur syntaxique, écrit en Eiffel.
—, *LDL : a Language Description Language User's Manual*, ibid. (45 p.)
LDL : langage d'expression des syntaxes abstraite et concrète pour Cépage.
+ grammaires de Pascal, Ada, LDL (pp 46-110).

[Mey 80] **B. Meyer**, *Langages de programmation – Présentation*, Encyclopédie "Techniques de l'Informatique", Paris, dec 80 (H 2040, 22 p.)
Les divers aspects des langages de programmation — illustrés par divers exemples de langages.

[Mey 85a] **B. Meyer**, *On Formalism in Specification*, IEEE Software Engineering, Vol 3 No 1, Jan 85 (pp 6-25)
De l'utilité de la spécification.

[Mey 85b] **B. Meyer**, *étapes sur le chemin du Génie Logiciel*; dont 1ère partie, chapitre 1 : «*survol partial d'une discipline*» (pp 1-33), thèse sur travaux Nancy (1975-1985), 9 Sept 85 (tome I et II, 700 p.)
Panorama sur la Génie Logiciel : cycle de vie, méthodes, langages, outils.

[MKM 85] **A. Menegaux, A. Krioule, B. Moussaid**, *Interprétation des Types Abstraits Algébriques*, Projet D.E.A., Nancy I, disponible auprès des auteurs
Présentation d'un Interprète de TAA.

[Mit 87] **Charles Z. Mitchell**, *Engineering VAX Ada for a Multi-language Porgramming Environment*, ACM SIGPLAN Notices, Vol 22 No 1, Jan 1987, Proc. of the ACM SIGSOFT/SOGPLAN Software Engineering Symp. on Practical Software Development Environment, Palo Alto, Ca, dec 9-11 86 (p 49-58)
Ada chez DEC (dt. effort particulier pour des dialogues Ada <-> autres lgges).

[MMV 85] **B. Mélèse, V. Migot, D. Verove**, *The Mentor-V5 Documentation*, Rapp. Techn. INRIA No 43, Jan 85 (180 p.)
Documentation technique sur Mentor-V5 (Mentor multi-langage) = doc. en ligne dont les commandes de Mentol, Mentorkit (pour définir un nouveau langage sous Mentor), les environnements Pascal, Ada, Metal, Mentor-Rapport.

[MRR 83] **Mossière, Raymond, Rouzaud**, *Représentation interne et manipulation de programmes dans l'atelier de logiciel Adèle*, Séminaire Aussois, INRIA, 18-22 avril 83 (pp 313-326)
On travaille sur Pascal : pb de la R.I. dans l'environnement = arbres abstraits.
Cheval, Estubier, Ghoul, Krakowiak, *Modularité et composition des programmes dans l'atelier de logiciel Adèle*, ibid. (pp 327-341)
Toujours en Pascal : modularité -> un schéma complet Interface + Corps + Spécification.
Herrman, Raymond, *Le poste de travail du projet Adèle*, ibid. (pp 342-355)
Interface multifenêtre attrayante d'Adèle.

[MvD 82a] **N. Meyrowitz, A. van Dam**, *Interactive Editing Systems : Part I*, ACM Computing Surveys, Vol 14 No 3, sept 82 (pp 321-352)
Présentation dans le détail des caractéristiques générales d'un éditeur — de texte ou de structure.
Historique, présentation des services généralement offerts dans les éditeurs.

[MvD 82b] **N. Meyrowitz, A. van Dam**, *Interactive Editing Systems : Part II*, ACM Computing Surveys, Vol 14 No 3, sept 82 (pp 353-415)
1. revue des éditeurs existants; 2. les éditeurs actuels et à venir.

[Nei 84] **J. H. Neighbors**, *The Draco Approach to Constructing Software from Reusable Components*, IEEE Trans. on Software Engineering, Vol 10 No 5, sept 84 (pp 564-574)
Draco : système de transformation de programmes "source-à-source".

[Nel 67] **Theodor H. Nelson**, *Getting It Out of Our System*, Information Retrieval: A critical review, G. Schecter Ed., Thomson Book Co., Washington D.C., 1967 (pp 191-211)
Présentation de l'«hypertexte» = texte structuré, qui par nature ne peut être imprimé sous une forme satisfaisante (liens trop ténus entre les composants).

[Nix 85] **Robert P. Nix**, *Editing by Example*, ACM Trans. on Programming Languages and Systems, Vol 7 No 4, oct 85 (pp 600-621)
édition de motifs répétitifs par l'exemple (unification -> pb NP-complet).

[OtO 84] **Karl J. et Linda M. Ottenstein**, *The Program Dependence Graph in a Software Development Environment*, ACM SIGPLAN Notices, Vol 19 No 5, May 84 (pp 177-184)
Présentation de PDG, support de la R.I. des sources dans un environnement.

[PaC 86] **D.L. Parnas, P.C. Clements**, *A Rational Design Process: How and Why to Fake It*, IEEE Trans. on Software Engineering, Vol 12 No 2, feb 1986, (pp 251-257)
Comment "truquer" les règles de bonne conception pour le développement effectif d'un logiciel. Points clés: la décomposition modulaire, la documentation.

[Pag 87] **F. Pagan**, *A graphical FP Language*, ACM SIGPLAN Notices, Vol 22 No 3, March 87 (pp 21-39)
Présentation d'une interface graphique pour la construction de fonctions FP récursives.

[Pai 87] **C. Pair**, *Programmation et langages de programmation*, Journées d'étude ESIEE, 17-19 juin 87 (10 p.)
L'état de l'art en matière de programmation et langages. Panorama général.

[Par 72] **D. L. Parnas**, *On the Criteria To Be Used in Decomposing Systems into Modules*, Comm. of the ACM, Vol 15 No 1, dec 72 (pp 1053-1058)
Les avantages de la modularisation et d'une structuration hiérarchique.

[PaS 86] **C. Parent, S. Spaccapietra**, *Une approche sémantique de la définition d'une algèbre entité-relation*, NBD-recherche No 2, Jan 86 (pp 3-14)
Algèbre pour l'interrogation des bases de données.

[Per 85] **G.-R. Perrin**, *La comunication: un outil pour la spécification, la construction et la vérification de systèmes parallèles*, thèse Univ. Nancy, 3 oct 85 (250 p.)
expression de la communication dans les systèmes parallèles sous forme de Types Abstraits Algébriques; classement des «types de communication».

[PPS 79] **N.S. Prywes, A. Pnueli, S. Shastry**, *Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development*, ACM Trans. on Programming Languages and Systems, Vol 1 No 2, oct 1979 (pp 196-217)
Model II: "spécification" et génération automatique de programme en PL/I, + contrôles de cohérence, complétude.

[Ree 87] **Karl Reed**, *Practical Software Engineering Environments*, ACM SIGSOFT Software Engineering Notes, Vol 12 No 1, Jan 87 (pp 56-62)
Résumé de «Software Engineering Symposium on Software Development Environments», SIGPLAN Notices, Vol 22 No 1, 9-11 dec 86.

[Rei 84] **Steven P. Reiss**, *Graphical Program Development with PECAN Program Development Systems*, ACM SIGPLAN Notices, Vol 19 No 5, may 84 (pp 30-41)
Vues multiples des programmes à partir de la S.A.: -> ed. syntaxique + déclarations des variables + graphe de contrôle.

[Rei 86] **Steven P. Reiss**, *An Object-Oriented Framework for Graphical Programming*, ACM SIGPLAN Notices, Vol 21 No 10, Oct 86 (pp 49-57)
Présentation du système GARDEN.

[Rei 87] **Steven P. Reiss**, *A Conceptual Programming Environment*, Proc. of the IEEE 9th International Conf. on Software Engineering, march 30-april 2, 1987, Monterey, Ca (pp 225-235)
technique des LOO pour la définition de l'environnement GARDEN, paramétré par le formalisme (texte + graphique).

[ReT 85] **T. Reps, T. Teitelbaum**, *The Synthesizer Generator Reference Manual*, Dept of Computer Science, Cornell Univ., Ithaca, NY 14853, Aug 85
Générateur d'éditeur syntaxique
définition d'un langage — manuel d'utilisation de l'éditeur.

[Rez **] **I. Reznikoff**, *Logique – Théorie de la démonstration et de l'intuitionnisme*, Encyclopædia Universalis (8 p.)
Présentation des Systèmes Formels et de leurs propriétés.

[Rip 83] **Knut Ripken**, *L'environnement Ada : vers une intégration des outils de gestion et de développmeent*, Journées BIGRE 83, le Cap d'Agde, 17-19 oct 83 (pp 675-694)
Regard "industriel" sur le pb des environnements de programmation.
Triangle de cohérence : développement — gestion de projet *et* gestion de configuration.

[Rob 87] **Arch D. Robison**, *The Illinois Functional Programming Interpreter*, Proceeding of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, St Paul, Minnesota, June 24-26, 1987 (pp 64-73)
IFP = FP de [Bac 78] efficace (comparaison avantageuse avec BASIC).

[Roy 87] **Wiston W. Royce**, *Managing the Development of Large Software Systems*, Proc. of the IEEE 9th International Conference on Software Engineering, march 30-apr 2 1987, Monterey, Ca (pp 328-338)
Le cycle de vie, le modèle de la cascade complété par: des documents à chaque étape; une revue critique du Client; une étude et une réalisation préliminaires, dont le résultat est exploité dans la suite du développement (paru initialement en 1970).

[Sch 86] **Kurt J. Schmucker**, *Object-Oriented Programming for the Macintosh*, Hayden Book Company, 1986
Revue des LOO avec accès à la bibliothèque MacApp du Macintosh : principalement Object Pascal, Smalltalk, Lisa Clascal.

[Sch 87] **Daniel Schneider**, *The Programming Language Scheme*, AICOM, Vol 0 No 1, Aug 87 (pp 17-27)
Description du dialect Lisp *Scheme*.

[ScS 83] **W.L. Scherlis, D.S. Scott**, *First Steps towards Inferential Programming*, Inf. Proc. 83, R.E.A. Mason ed., Elsevior Science Publishers B.V. (North Holland), IFIP 83 (pp 199-212)
Programmation par *dérivation* où l'objet d'intérêt n'est pas le programme dérivé mais le processus de dérivation. L'historique de la dérivation remplace la preuve.

[ShK 87] **Stan Shebs, Robert Kessler**, *Automatic Design and Implementation of Languages Datatypes*, Proceeding of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, St Paul, Minnesota, June 24-26 1987 (pp 26-37)
Expression par des fonctions Lisp des «types de base» (integer, string, real, ...) d'un langage interprété + des spécificités de la machine (registres, adresses) + implantation (utilisation d'heuristiques + intervention humaine).

[Sol 84] **Elliot Soloway**, *A Cognitively-Based Methodology for Designing Languages / Environments / Methodologies*, ACM SIGPLAN Notices, Vol 19 No 5, May 84 (pp 193-196)
Méthode "cognitive" d'évaluation des «bons logiciels»
cf *Elliot Soloway, Kate Ehrlich*, Empirical Studies of Programming Knowledge, IEEE Trans. on Software Engineering, Vol 10 No 5, sept 84 (pp 595-609) : expérimentation (139 étudiants) sur des exemples de rédactions d'un programme correct lisible / illisible.

[Sou 83] **Loufti Soufi**, *Un système de construction de programmes*, Journées BIGRE 83, le Cap d'Agde, 17-19 oct 83 (pp 204-224)
Le Système Intéractif d'Explicitation (l'outil présenté) = un éditeur, un bibliothéquaire de stratégies, un constructeur, fonctionnant selon la méthode déductive.

[Sta 81] **Richard M. Stallman**, *EMACS: The Extensible, Customizable, Self-Documenting Display Editor*, Proc. of the ACM SIGPLAN/SIGOA Conf. on Text Manipulation, Portland, Oregon, June 8-10 1981 (pp 147-156)
Les traits d'EMACS, de extensible, la «famille EMACS».

[Str 86] **Ola Strömfors**, *Editing Large Programs Using a Structure-Oriented Text Editor*, Lecture Notes in Computer Science (Goos et Hortmanis ed.), Advanced Programming Environments, Proceeding of an International Workshop, Trondheim, Norway, June 16-18 1986 (pp 39-46)
Editeur structuré, appliqué à Pascal (+ analyseur syntaxique, *pretty-printing*).

[SZB 86] **D.C. Swinehart, P.T. Zellweger, R.J. Beach, R.B. Hagmann**, *A Structural View of the Cedar Programming Environment*, ACM Trans. on Programming Languages and Systems, Vol 8 No 4, oct 1986 (pp 419-490)
Langage environnement Cedar: un environnement entièrement intégré.

[TeH 77] **D. Teichroew, E.A. Hershey**, *PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems*, IEEE Trans. on Software Engineering, Vol SE-3 No 1, Jan 77
Spécifier le problème en PSL (Problem Statement Language) -> analysé par PSA (Problem Statement Analyser).

[TeM 81] **W. Teiltelman, L. Masinter**, *The Interlisp Programming Environment*, Computer, IEEE, Vol 14 No 4, april 81 (pp 25-34)
Présentation des traits de l'environnement INTERLISP.

[Tou 88] **David S. Touretzky**, *How Lisp Has Changed*, BYTE, Vol 13 No 2, feb 88 (pp 229-236)
le langage Lisp, et plus spécifiquement Common Lisp.

[Tra 88] **Will Tracz**, *Software Reuse Myths*, ACM SIGSOFT Software Engineering Notes, Vol 13 No 1, Jan 1988 (pp 17-21)
Les 9 mythes concernant la réutilisation du logiciel et les 9 réalités.

[TRC 83] **H. Tardieu, A. Rochefeld, R. Colleti**, *La méthode Merise – principes et outils*, les editions d'organisation, Paris, 1983 (300 p.)
La méthode Merise: replacée dans le contexte de l'informatique de gestion; présentée; les outils.

[TRH 81] **T. Teitelbaum, T. Reps, S. Horwitz**, *The Why and Wherefore of the Cornell Program Synthesizer*, ACM SIGPLAN Notices, Vol 16 No 6, June 1981. Proceeding of the ACM SIGPLAN/SIGOA Symposium on text manipulation, Portland, Oregon, June 8-10 1981 (pp 8-16)
Présentation de CPS. Exemple de construction de programme pas-à-pas.
Editeur syntaxique + contrôle des déclarations + exécution (d'un programme éventuellement incomplet).

[TrW 86] **R.H. Trigg, M. Weiser**, *TEXTNET : A Network-Based Approach to Text Handling*, ACM Trans. on Office Information Systems, Vol 4 No 1, June 86 (pp 1-23)
Présentation de l'hypertexte TEXTNET.

[TrY 80] **J.M. Triance, J.F. S. Yow**, *MCOBOL — A Prototype Macro Facility for Cobol*, Comm. of the ACM, Vol 23 No 8, Aug 80 (pp 432-439)
Macros syntaxiques COBOL, dans la syntaxe COBOL.

[TYF 86] **Toby J. Teorey, Dongqing Yang, James P. Fry**, *A Logical Design Methodology for Relational Database Using the Extended Entity-Relationship Model*, ACM Computing Surveys, Vol 18 No 2, June 86 (pp 197-222)
Modèle entité-relation (EER) pour la réalisation de B.D.

[Vit 84] **J.S. Vitter**, *US&R : a new framework for redoing*, ACM SIGPLAN Notices, Vol 19 No 5, May 84 (pp 168-176)
Undo, Skip, Redo.
Gestion de l'historique des commandes par un arbre (≠ classiquement linéaire).

[Wal 81] **Janet H. Walker**, *The Document Editor : A Support Environment for Preparing Technical Documents*, ACM SIGPLAN Notices, Vol 16 No 6, June 1981. Proceeding of the ACM SIGPLAN/SIGOA Symposium on text manipulation, Portland, Oregon, June 8-10 1981 (pp 44-50)
Editeur de documents techniques structurés.

[Wat 82] **Richard C. Waters**, *The Programmer's Apprentice : Knowledge Based Program Editing*, IEEE Trans. on Software Engineering, Vol SE-8 No 1, Jan 82 (pp 1-12)
L'apprenti du programmeur, aide à la construction du programme en se chargeant des "basses besognes". Interface en pseudo-anglais, écho en programme Lisp. R.I. par plans = graphes data-flow.

[Wat 86] **Richard C. Waters**, *Reuse of Clichés in The Knowledge-Based Editor*, Lecture Notes in Computer Science (Goos et Hortmanis ed.), Advanced Programming Environments, Proceeding of an International Workshop, Trondheim, Norway, June 16-18 1986 (pp 537-550)
éditeur->emacs, Lisp->Ada, langage de "commande" plus souple. — cf. [Wat 82]

[WeB 85] **S.A. Weyer, A.H. Borning**, *A Prototype Electronic Encyclopedia*, ACM Trans. on Office Information Systems, Vol 3 No 1, Jan 85 (pp 63-88)
Présentation formelle d'une encyclopédie électronique construite avec un hypertexte – schémas de principe.

[Wer 85] **H. Wertz**, *Intelligence Artificielle – Application à l'analyse des programmes*, Masson, 1985 (220 p.)
Exposé sur Φ, système "intelligent" de correction de programmes pour Lisp
+ biblio (pp 196-214).

[Wey 82] **Stephen A. Weyer**, *The design of a dynamic book for information search*, Int. Journal Man-Machines Studies, Vol 17 No 1, 1982 (pp 87-107)
expérimentation (réussie) de l'utilisation d'un hypertexte pour une recherche de documentation sur des sujets précis.

[Wil 86] **Maurice V. Wilkes**, *Empiric : A sketch of programming language designed to facilitate a fine grain of protection*, ACM SIGPLAN Notices, Vol 21 No 8, Aug 86 (pp 16-27)
Langage avec de forts contrôles de cohérence (au-delà du simple typage).

[Woo 81] **Steven R. Wood**, *Z - The 95 % Program Editor*, ACM SIGPLAN Notices, Vol 16 No 6, June 1981. Proceeding of the ACM SIGPLAN/SIGOA Symposium on text manipulation, Portland, Oregon, June 8-10 1981 (pp 1-7)
Z éditeur de texte classique, qui réalise 95% de ce qu'on souhaite faire en phase d'édition. Résolument opposé aux éditeurs syntaxiques — complication notable de l'implantation.

[YMD 85] **N. Yankelovich, N. Meyrowitz, A. van Dam**, *Reading and Writing the Electronic Book*, Computer, IEEE, Vol 18 No 10, oct 85 (pp 15-30)
Défense de l'hypertexte; mise en parallèle avec le livre habituel. Présentation des hypertextes développés à Brown University – FRESS, The Electronic Document System, BALSA, Intermedia.

[Zav 84] **Pamela Zave**, *The Operational versus the Conventional Approach to Software Developpement*, Comm. of the ACM, Vol 27 No 2, feb 1984 (pp 104-118)
Spécification opérationnelle opposée à la spécification traditionnelle par décomposition descendante. Nécessité d'outils de transformation de programme (sinon: -> prototypes, études préliminaires, études de faisabilité, ...).

[Zav 88] **Pamela Zave**, *Assessment*, ACM SIGSOFT Software Engineering Notes, Vol 13 No 1, Jan 1988 (pp 40-43)
cf. [ArF 88] – définition de la spécification dans l'optique d'une spec opérationnelle.

[Zel 84] **Marvin V. Zelkowitz**, *A small contribution to editing with a syntax directed editor*, ACM SIGPLAN Notices, Vol 19 No 5, may 84 (pp 1-6)
SUPPORT: une petite contribution au domaine de l'édition syntaxique.

Table

Chapitre 1:
Le Problème et la Proposition

Chapitre 2:
Le Langage Primitif de Représentation Textuelle

Chapitre 3:
Le Langage Complété pour la Structuration des Textes

Chapitre 4:
L'Enrichissement du Langage par de Nouveaux Concepts

Chapitre 5:
La Formalisation
des Solutions Techniques

Chapitre 6:
Les Comparaisons
avec d'autres Approches

Chapitre 7:
Les Perspectives

Annexes

Chapitre 8:
Les Editeurs

Chapitre 9:
Les Aspects d'Implantation

Références