FILTRES CONTEXTE ET DÉMONS

Outils d'aide à la modélisation et à la programmation de jeux

Historiquement plusieurs voies ont été poursuivies à partir du langage LISP : d'un côté s'est développé très tôt le langage LOGO, de l'autre côté se sont développés des langages tels que PLANNER (Hewitt 1969, Durieux 1981) et CONNIVER (Mc Dermott & Sussman 1974, Greussay 1976), ainsi que toute la famille de langages traitant des objets et des messages. Ce que je propose ici est d'inclure dans les langages applicatifs utilisés dans l'apprentissage autonome les constructions développées dans ces langages, et tout d'abord un mécanisme de "filtrage", mécanisme simple et puissant, dont certains neurophysiologues prétendent même avoir trouvé l'équivalent dans une partie du fonctionnement du cortex visuel (Broadbent 1961, 1970). Ensuite nous allons voir, par un exemple, ce que peut donner la combinaison du mécanisme de filtrage avec un mécanisme de base de données, ce qui conduit à la notion de "démon".

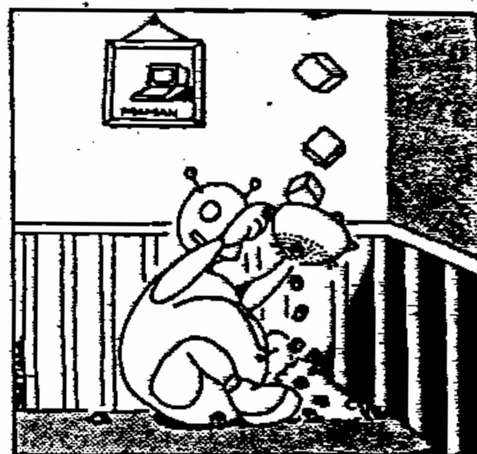
Programmation et psychologie cognitive

J'ai implémenté ces mécanismes de *filtrage* et de *démons* dans la version de VLISP-80 sur TRS-80 et ils sont utilisés quotidiennement par les élèves de la SES avec laquelle nous collaborons. Ils les ont utilisés pour la construction de programmes de 'dialogues' (des programmes permettant de simuler un dialogue sur un sujet restreint), pour un petit programme de robotique similaire à celui donné ici en exemple, ainsi que pour un programme de consultation de base de données.

Naturellement, la raison de tout cet exercice n'est nullement de démontrer quelques algorithmes, mais, plutôt, de montrer la différence fondamentale entre une programmation que j'ai envie de nommer 'conceptuelle' et la programmation algorithmique habituelle. Si nous prenons la métaphore de la programmation au sérieux, si nous voulons utiliser la programmation pour modéliser le fonctionnement cognitif de la résolution de problèmes, nous devons absolument nous séparer des concepts classiques de programmation et ne plus nous laisser trop influencer par nos connaissances informatiques, mais utiliser nos connaissances de la psychologie cognitive pour intégrer cette dernière dans les langages de programmation que nous voulons utiliser. LOGO et l'apprentissage autonome ne doivent pas devenir dépendant des langages et de la technologie informatique, mais doivent — au contraire — influencer et stimuler le dévelopement des langages mis à la disposition des jeunes.

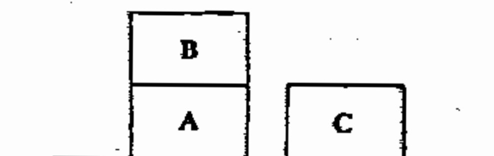
Le filtrage

Le filtrage, ou 'pattern-matching', est un mécanisme permettant d'accéder à des objets non plus par un calcul positionnel, mais par une description de la structure des objets.



Cette description est ensuite comparée avec les objets afin de connaître l'existence ou l'inexistence d'objets lui correspondant. Prenons comme exemple un mini-monde de cubes. Ce mini-monde est constitué d'une table et d'un ensemble de cubes sur la table.

Dans ce mini-monde, la scène suivante :



sera décrite par :
(SUR A TABLE)
(SUR B A)
(SUR C TABLE)

Classiquement, si je veux savoir quel objet se trouve sur le cube A, je devrais chercher à l'intérieur de ma description un triplet commençant par le mot SUR et se terminant par le mot A, ensuite prendre le mot au milieu et j'aurais trouvé la réponse. Ce qui donne en LOGO :

```
POUR SUR ? : X : DESCRIPTION
  SI VIDE : DESCRIPTION [OUTPUT "RIEN"]
  SINON [SUR1 ? : X PREMIER : DESCRIPTION
         SUR ? : X SAUFPREMIER : DESCRIPTION
FIN
```

avec

```
POUR SUR1 ? : X : TRIPLET
  SI PREMIER : TRIPLET = "SUR [SI LAST : TRIPLET = : X
                               [OUTPUT
                             PREMIER SAUFPREMIER : TRIPL
                               STOP]]
FIN
```

avec le mécanisme de filtrage, la procédure SUR1 ? deviendra :

```
POUR SUR1 ? : X : TRIPLET
  SI FILTRE : X : TRIPLET [OUTPUT : OBJET]
FIN
```

si (!), préalablement j'ai changé l'appel de SUR1 ?, dans la fonction SUR ? en :

```
POUR SUR ? : X : DESCRIPTION
  SI VIDE : DESCRIPTION [OUTPUT "RIEN"]
  SINON [SUR1 ? : '(SUR ! OBJET : X) PREMIER :
          DESCRIPTION
          SUR ? : X SAUFPREMIER : DESCRIPTION]
FIN
```

Détaillons : pour filtrer il faut un 'filtre' et une 'donnée'. La donnée peut être n'importe quelle liste. Le filtre est soit une liste normale, dans ce cas on cherche une occurence de ce filtre (qui n'est rien d'autre qu'une donnée), soit une liste contenant des variables précédées par le signe '!'. Ces variables sont considérées comme des 'fentes' à travers lesquelles ont peut attraper un élément se trouvant dans la donnée à la même position. Le filtrage est donc un mécanisme de recherche, d'éléments. La puissance de ce mécanisme vient de sa 'lisibilité', la possibilité de compréhension 'visuelle' de ce qui est cherché. Voici quelques exemples de données et de filtres ainsi que les résultats du filtrage :

filtre	donnée	résultat
(SUR B A)	(SUR B A)	oui
(SUR A B)	(SUR B A)	non
(SUR !X A)	(SUR B A)	oui, et X = B
(SUR !X !Y)	(SUR B A)	oui, et X = B, Y = A
si X = B et Y = A :		
(SUR !X A)	(SUR B A)	oui, et X = B
(SUR !Y !X)	(SUR B A)	non

Pour l'instant, ce qui est vraiment intéressant est que ce filtrage livre un mécanisme, visuellement compréhensible, de recherche d'objets. Il suffit de connaître le modèle structural des objets pour pouvoir s'en servir. Le filtrage permet de remplacer des parties algorithmiques de la programmation par des parties descriptives actives.

Avant de voir dans la suite comment utiliser ce mécanisme sur une base de données ET pour le lancement de procédures, reprenons notre exemple de l'inversion d'une liste d'objets, et rappelons-nous les deux algorithmes intuitifs que j'ai énoncés.

Sans autre commentaire, je vais donner — en LISP enrichi du mécanisme de filtrage — les programmes correspondant à l'algorithme échangeant le premier et le dernier objet de la liste. Évidemment, le lecteur, après une petite réflexion, peut VOIR le fonctionnement de ce programme et l'algorithme sousjacent.

```
(DE INVERSER (L)
    (IF (NULL L) ()
        (FILTRE (!X ?Y !Z) L) [!Z @(INVERSER Y) !X]))
```

Il est laissé au soin des lecteurs d'écrire, dans ce style de programmation (rappelant des 'règles de réécriture') le programme implémentant l'autre des deux algorithmes.

Notons que cette notion de 'filtrage' se trouve dans tous les langages issus de LISP (**Planner** [Hewitt 1969], **Plasma** [Durieux 1981], **Conniver** [Greussay 1976], **Smalltalk** [Goldberg 1981, Cointe 1982], etc) sous des formes les plus variées, aussi bien comme opération de base pour la recherche d'éléments dans les bases de données que - nous le verrons dans la suite - comme mécanismes de lancement des procédures. Je considère le mécanisme de filtrage comme plus fondamental que le mécanisme d'affectation des langages de programmation plus usuels ; il le subsume et le généralise.

Mécanisme de contexte

Je vais, suivant la terminologie du langage CONNIVER appeler combinaison de mécanismes de bases de données et de mécanismes de filtrage un "contexte". Pour mon exposé, nous pouvons considérer une base de données comme une collection d'objets(1). Nous nommons les objets à l'intérieur de la base de données des 'items'. Toute base de données doit posséder au moins deux opérateurs, un pour ajouter un item, que nous nommerons AJOUTE, et un pour enlever un item, que nous nommerons ENLEVE.

Ainsi, pour ajouter le fait que le cube D se trouve sur le cube C, il est suffisant de lancer l'opérateur AJOUTE comme suit :

(AJOUTE '(SUR D C))

et pour enlever ce fait, il suffit d'évaluer l'appel

(ENLEVE '(SUR D C))

Naturellement, la description de notre micro-monde de cubes est représentée comme une collection d'items à l'intérieur de notre base de données.

Toute base de données permet également une question d'existence : est-ce que tel ou tel item se trouve dans la base de données ? Cette interrogation sera implémentée avec la fonction PRESENT. Ainsi, l'évaluation de

(PRESENT '(SUR A B))

nous livre NIL si le fait (SUR A B)) n'est pas affirmé dans le contexte, sinon elle livre le fait (SUR A B) lui-même.

Afin de voir ce qu'on peut faire avec ce mécanisme, reprenons notre petite scène initiale :

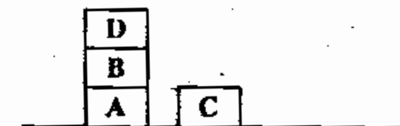
qui est représentée dans le contexte par les 3 triplets suivants :

(SUR A TABLE)
(SUR B A)
(SUR C TABLE)

Si nous voulons poser un cube sur le cube B, disons le cube D, il suffit d'évaluer :

(AJOUTE '(SUR D B))

ce qui modifiera notre contexte de manière à avoir la scène suivante :



Voici à présent quelques exemples d'appels de la fonction PRESENT avec leurs résultats dans ce contexte :

(PRESENT '(SUR B A)) ——→ (SUR B A)
(PRESENT '(SUR A B)) ——→ NIL

avec des filtres :

(PRESENT '(SUR C !X)) ——→ (SUR C TABLE)
(PRESENT '(!REL D B)) ——→ (SUR D B)
(PRESENT '(SUR !X D)) ——→ NIL

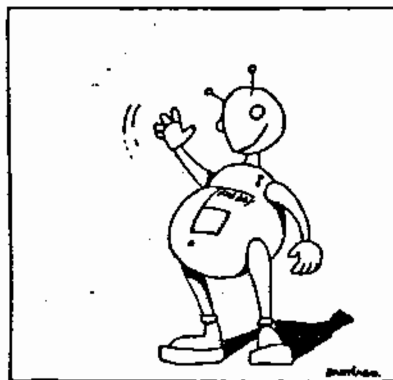
(1) En réalité, notre base de données aura naturellement une organisation plus sophistiquée que celle d'une collection.

Il est bien évident que le filtrage, combiné avec des mécanismes de bases de données, peut donner un style de programmation beaucoup plus puissant et beaucoup plus 'lisible' que la programmation de style purement applicative. Afin de rendre ceci tout à fait clair, je vais développer un petit programme simulant un robot dans ce micro-monde. Ce robot, s'appelant FREDDY, peut prendre et déplacer des cubes, à raison d'un seul cube à la fois.
Naturellement, nous avons tout d'abord à écrire un programme qui assure que le cube que FREDDY doit déplacer est 'libre', c'est-à-dire, que rien ne se trouve sur lui. Ce programme (en LISP enrichi de ces mécanismes) s'écrit donc :

```
(DE LIBERER (X)
    (IF (PRESENT '(SUR !Z !X))
        (SUR-TABLE Z)))
```

Remarquons que nous suivons ici la méthode de programmation *'top-down'* : nous descendons du général vers le particulier. Ainsi, à l'endroit où nous voulons être assurés que l'objet Z soit placé sur la table, nous appelons une autre 'routine' (SUR-TABLE), pas encore écrite, qui doit régler ce problème. Nous connaissons le 'contrat' pour cette 'routine' : placer, après s'être assuré que Z est libre, cet objet sur la table. Nous pouvons l'écrire simplement :

```
(DE SUR-TABLE (X)
    (LIBERER X)
    (ENLEVE (PRESENT '(SUR !X !Z)))
    (AJOUTE '(SUR !X TABLE)))
```



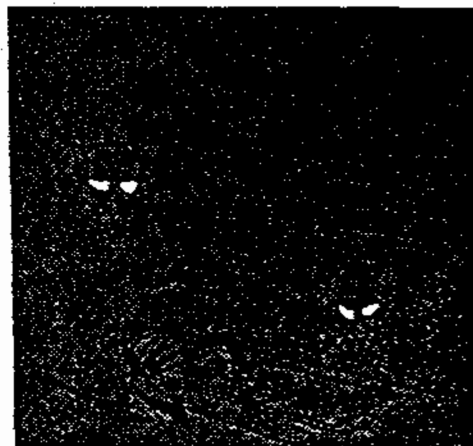
Puisque nous simulons le robot FREDDY, les actions du robot correspondent à des mises à jour de la base de données. Maintenant, nous pouvons écrire le programme posant un cube sur un autre. Tout ce que ce programme doit faire est de s'assurer que les deux cubes sont 'libres' et ensuite il doit adapter la base de données à la nouvelle situation où l'un des cubes se trouve sur l'autre. Voici le programme :

```
(DE POSER-SUR (X Y)
    (LIBERER X)
    (LIBERER Y)
    (ENLEVE (PRESENT '(SUR !X !Z)))
    (AJOUTE '(SUR !X !Y)))
```

Remarquez que le seul problème dans ce programme était de se rappeler qu'il ne suffit pas de placer l'un des cubes sur l'autre, mais qu'il faut également enlever l'information sur la position précédente du cube !
Ce programme fournit une simulation très sommaire du robot. L'unique problème est que nulle part dans le programme n'est dit ce que le 'robot' doit faire : c'est un programme de simulation de robot qui se permet d'ignorer les actions du robot. Afin de montrer une manière très élégante d'introduire le robot, mais aussi afin de montrer une manière de programmation complètement différente de celle utilisée dans des langages applicatifs usuels, nous allons d'abord élargir notre notion des mécanismes de contextes.

Les démons

Jusqu'à maintenant nous connaissons les filtres comme descripteurs de la structure d'un item à chercher ou à consulter à l'intérieur de la base de données. Ce qui nous a permis de programmmer des procédures d'accès ou de consultation ainsi que de test d'existence d'items en termes de description de leur structures. Il est relativement aisé de s'imaginer les filtres comme descripteurs de situations:dans notre mini-exemple de robotique, chaque addition d'un item à la base de données correspond à une situation particulière dans notre micro-monde de cubes. Un test de 'présence' d'un filtre, correspond à un test de situation, à un moment instantané, dans ce micro-monde



Très souvent, dans la vie courante, les algorithmes ne sont point exprimés comme des descriptions de suite de processus actifs, l'un à la suite de l'autre, suivant un ordre pré-établi, mais comme des actions à faire si telle ou telle situation se présente. Par exemple, afin de déterminer si l'on doit manger, personne ne semble suivre l'algorithme suivant :

pour savoir s'il faut manger
    vérifier
        si l'on a faim, alors il faut manger
        sinon 'savoir s'il faut manger'

Et pourtant, cet algorithme ressemble tout à fait à un algorithme récursif classique. Les deux absurdités dans cet algorithme semblent être :

1) le bouclage jusqu'à ce qu'on ressente la faim, et
2) la répétition continue du test 'est-ce que j'ai faim'.

Il nous semble plutôt que nous avons un ensemble de petits processeurs très spécialisés, travaillant en permanence, et ne se faisant sentir qu'à partir de l'instant ou une certaine limite est atteinte. Ainsi, j'ai l'impression que j'ai un processeur particulier, ne faisant (en parallèle à l'activité de tous mes autres processeurs) rien d'autre que vérifier si j'ai faim. L'activité de ce processeur ne devient consciente qu'à l'instant où j'ai atteint une certaine limite de mon sentiment 'faim', et seulement à partir de cet instant-là, mais là très intensément, ce processeur me rappelle son existence et sa raison d'être.
Ce que je viens d'écrire est une sorte de calcul situationnel : attente, en *'background'*, d'une certaine situation, et à l'arrivée de cette situation, émission d'un signal activant un autre processeur (dans l'exemple : le processeur qui donne envie de manger). Exprimé dans ce calcul-ci, l'algorithme de 'comment savoir quand manger' devient beaucoup plus simple et pourrait s'écrire :

quand j'ai le sentiment de faim, alors il faut manger

Mais, du coup, je n'ai plus de boucle de contrôle. Je n'ai qu'un test de situation et une action attachée à chaque situation.

Pour revenir à notre programme-exemple : rappelons-nous que chaque situation est exprimée par un ensemble d'items bien précis dans la base de données. Chaque modification de situation est due, soit à un ajout d'un item dans le contexte, soit à un enlèvement d'un item du contexte. Tout ce qui est nécessaire pour pouvoir programmer des algorithmes s'exprimant en termes de situations et d'actions connectées à des situations, est un mécanisme permettant d'associer des programmes à des items, ou, plus généralement, à des 'filtres'. De tels mécanismes sont appelés des 'démons' (en référence au 'démon de Maxwell').

Plus formellement : une méthode est une combinaison d'un filtre avec un programme:

où le filtre est la condition d'activation du programme capable de décrire des items. Je peux associer des méthodes à chacun des opérateurs travaillant sur la base de données. Ainsi, je peux associer des démons à l'opérateur AJOUTE, j'appellerai de tels démons des démons SI-AJOUTE. Si le démon est associé à l'opérateur ENLEVE, je parlerai d'un démon SI-ENLEVE, et si le démon est associé à l'opérateur PRESENT, je parlerai d'un démon SI-BESOIN. L'activité d'un démon sera de surveiller les opérations auxquelles il est associé et dès que l'item, qui se trouve soit ajoute, soit enlève, soit recherche dans le contexte (dépendant de l'opérateur), correspond au filtre du démon, le démon s'active automatiquement sans spécification supplémentaire. Ainsi le fonctionnement des démons SI-AJOUTE peut être décrit comme :

```
si AJOUTE doit placer un 'item' dans un 'contexte'
d'abord les démons de type SI-AJOUTE regardent
 si
   leur filtre est capable de décrire l'item
 et si c'est le cas,
   leur programme est activé
puis
  l'item est placé dans le contexte
```

Les méthodes SI-ENLEVE et SI-BESOIN ont naturellement un comportement analogue.

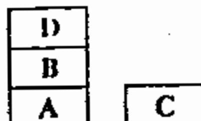
Utilisons alors ces mécanismes pour introduire le robot. Remarquons que, puisque nous n'avons pas de robot réel, il nous suffit de voir s'afficher sur l'écran les activités du robot pendant le déplacement des cubes. Pour cela, nous allons introduire deux démons dans notre contexte du micro-monde de cubes : un pour réagir chaque fois qu'on enlève un fait de la base de données (la première manière de changer de situation), qui sera de la forme :

```
(SI-ENLEVE (SUR !X !Y)
  (PRINT (FREDDY PREND !X)))
```

disant juste que chaque élimination d'un fait de la base de données correspond à l'action de prendre quelque chose. La méthode, que nous associons à l'ajout d'un item dans la base de données, exprime que cette action correspond à la pose d'un objet :

```
(SI-AJOUTE (SUR !X !Y)
  (PRINT (ET LE POSE SUR !Y)))
```

Nous avons maintenant un programme (de seulement 16 lignes !) tout à fait respectable pour la simulation d'un petit robot dans un micro-monde simplifié. Si, dans la situation suivante :



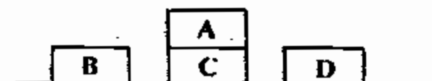
nous évaluons l'appel

```
(POSER-SUR A C)
```

le résultat sera la suite d'impressions suivantes :

```
(FREDDY PREND D)
(ET LE POSE SUR LA TABLE)
(FREDDY PREND B)
(ET LE POSE SUR LA TABLE)
(FREDDY PREND A)
(ET LE POSE SUR C)
```

et la situation résultante, exprimée comme une suite d'items dans le contexte, correspondra à la situation suivante :



J'invite le lecteur à vérifier la correction de cet algorithme, et à le compléter à volonté, pour le rendre plus général, et pour s'habituer à programmer de cette manière-ci.

Le mécanisme de 'démons' est connu également sous le nom de 'lancement de procédure par filtrage' et est le processus de base de mécanismes d'inférence des langages style 'PLANNER'. Il est — comme le filtrage lui-même — intégré de manière standard dans tous les langages utilisés dans les recherches en intelligence artificielle, permettant une programmation aisée d'algorithmes non-hiérarchiques. Il est également à la base de la notion d'acteur des langages du style PLASMA ou SMALLTALK.

[Suivant M.Minsky 1982], il représente une manière élégante et aisée de représenter des entités conceptuelles actives du cerveau.

Je me rappelle encore un vieux lexique de mon enfance où le cerveau était représenté fonctionnant comme un bureau administratif. Entre-temps, j'ai vu des lexiques plus modernes, représentant le fonctionnement du cerveau comme un ordinateur séquentiel, ensuite le représentant comme une sorte de réseau d'ordinateurs. La leçon à tirer de ces changements successifs de représentation populaire du fonctionnement du cerveau est, peut-être, que tant que nous ne le connaissons pas mieux, nous ne devons pas nous fixer sur la manière de le modéliser (et la modélisation des processus cognitifs en fait bien partie), mais laisser la voie ouverte pour explorer des chemins supplémentaires et expérimentaux. Il serait triste que LOGO et l'apprentissage autonome se limitent au stade de développement de l'informatique du début des années 70, pour aider à concrétiser des processus cognitifs.

Le but de ce papier était de suggérer que l'informatique et la psychologie cognitive se développent continuellement, nous livrant continuellement des modèles plus puissants et plus accessibles. Intégrons ces découvertes, ces développements dans nos langages et nos environnements informatiques, alors, peut-être, la programmation pourra vraiment un jour nous aider effectivement à prendre conscience de nos propres procédures cognitives et intellectuelles.

Harald Wertz
Département d'Informatique
Université Paris VIII - Vincennes
2 rue de la Liberté 93256 - St. Denis

Remerciements :

Les idées exprimées dans ce papier ont été largement influencées par M. Patrick Greussay. Ces recherches sont soutenues partiellement par l'Agence de l'informatique, le Centre National de la Recherche Scientifique et le CNET et par la RCP-LOGO.

(cf. Références p 12)