

QUELQUES PROBLEMES CONCERNANT L'ADEQUATION DES LANGAGES APPLICATIFS
POUR LA REPRESENTATION DE PROCESSUS COGNITIFS

Harald WERTZ

LITP / CNRS
2 Place Jussieu
75005 Paris

&

Universite Paris 8
Dept. Informatique
2 rue de la Liberte
93526 St.-Denis Cedex 02

France

Resume :

Recentment les langages applicatifs ont recu un interet particulier dans le contexte de l'enseignement primaire et secondaire. Apres maintes essaies et recherches sur l'apprentissage et l'utilisation de langages de programmation algorithmiques classiques, un interet accru pour l'utilisation de l'informatique a des fins de formation a des techniques de 'resolution de problemes' s'est manifeste. Cet interet, fruit des recherches en intelligence artificielle, est souvent base sur une comprehension trop superficielle des mecanismes informatiques des langages applicatifs en tant que modele exacte des processus cognitifs.

LOGO et d'autres langages applicatifs semblent etre particulierement adaptes a des activites d'apprentissage autonome : les processus de resolution de problemes peuvent etre explicites grace a la construction de programmes dans de tels langages. Mais ceci n'est vrai que dans une mesure assez restreinte en relation directe avec les limitations des langages utilises. Je suggere quelques ameliorations des langages, susceptibles d'elargir considerablement les possibilites de modelisation.

INTRODUCTION

1.0 INTRODUCTION

Depuis maintenant plus de 10 années, des groupes de recherche éducatifs utilisent des ordinateurs comme 'objets transitoires' de la prise de conscience de processus de résolution de problèmes. La littérature abonde de rapports extrêmement positifs [Papert & al 1979, Bossuet 1982, Wertz & al 1979] sur des expériences du type LOGO. Avec le Groupe d'Aide à l'Éducation de Vincennes, je travaille, avec un succès considérable, depuis plus de 5 ans avec des élèves d'une SES(1) [Mathieu 1981]. Toutefois, si on regarde les réalisations informatiques des élèves, la complexité de leurs programmes, on est frappé par l'uniformité des résultats et par la pauvreté des programmes (en LOGO il est rare de voir des programmes qui échappent au 'carre' et à la 'maison'). Plutôt que les capacités intrinsèques des élèves, je pense que ce sont les mécanismes informatiques mis à disposition de ces personnes qui sont trop limitatifs et qui imposent trop de raisonnement 'informatique' (opposé à 'raisonnement' tout court) aux apprentis programmeurs, et que l'uniformité des résultats n'est que le reflet direct de la pauvreté des moyens mis à disposition et de l'inadéquation des structures informatiques (de ces langages) pour exprimer des mécanismes de résolution de problèmes.

Dans la suite de ce papier, après une courte description des mécanismes d'interprétation inhérents aux langages applicatifs, je vais proposer un mécanisme de 'filtrage' et de 'contexte' à ajouter à ces langages, et, par l'exemple, montrer comment de tels mécanismes peuvent considérablement augmenter le pouvoir expressif de tels langages. Ces mécanismes sont aisés à implémenter, même sur de petites configurations, et je crois qu'une mise à l'épreuve expérimentale générale serait la bienvenue.

2.0 LES LIMITATIONS DES MECANISMES D'INTERPRETATION DE BASE

Tout d'abord, afin de comprendre comment les interprètes des langages applicatifs fonctionnent et afin de voir leurs limitations, j'exposerai brièvement leur fonctionnement.

(1) Une SES est une Section d'Enseignement Spécialisée, c.a.d. les classes particulières installées pour des élèves ayant des problèmes de suivre l'enseignement normal des Collèges.

LES LIMITATIONS DES MECANISMES D'INTERPRETATION DE BASE

Tout interprete de langage applicatif (ici, nous serons concernes essentiellement par les langages LISP et LOGO) se limite principalement a un 'evaluateur' d'appels de fonctions, puisque tout programme dans ces langages n'est rien d'autre qu'une suite d'appels de fonctions de la forme :

un-appel : une-fonction arg-e1 arg-e2 arg-en

```

|
+----- une fonction standard (exemple : *)
|
+----- une fonction definie par l'utilisateur

```

ou les 'arg-ei' correspondent aux differents arguments de la fonction. Des exemples de cette forme sont :

+ 1 2 3 4

ou

FACT 4

si FACT est prealablement defini en LOGO comme :

```

POUR FACT :N
  SI :N = 0 [OUTPUT 1]
  SINON [OUTPUT :N * FACT :N - 1]
FIN

```

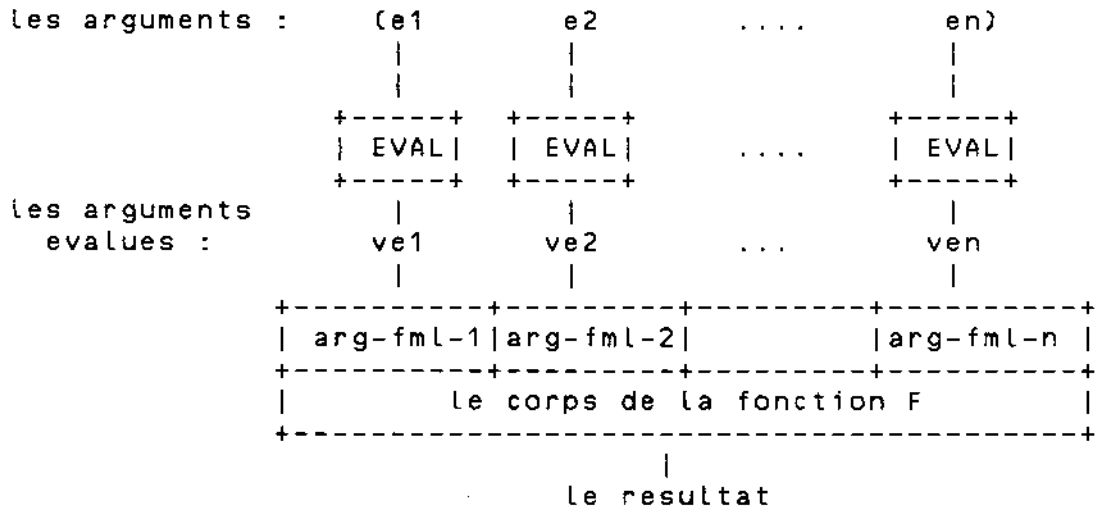
La maniere la plus simple de se représenter l'évaluation d'une fonction, est de s'imaginer la combinaison d'une multitude de 'machines evaluateur' (nomme 'EVAL') et d'une machine 'la-fonction', de la maniere suivante :

soit l'appel :

F e1 e2 ... en

d'abord les arguments e1, e2, ..., en seront evalues un a un, ensuite les valeurs resultant de l'évaluation de ces arguments seront envoyees a la machine 'la-fonction-F' (dans la figure ci-dessous, 'e1', 'e2' etc sont les arguments actuels, 've1', 've2' etc sont les valeurs des arguments actuels, et les 'arg-fml-1', 'arg-fml-2' etc sont les arguments formels).

LES LIMITATIONS DES MECANISMES D'INTERPRETATION DE BASE



Consequence : TOUTE interpretation d'un programme pouvant se resumer a ce mecanisme simple, l'image que le programmeur perçoit des mecanismes de resolution de problemes, s'il est construit a partir de la construction et l'execution de programmes, sera plus ou moins equivalent a ce mecanisme simple.

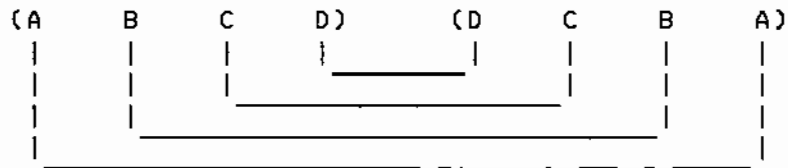
Or, il me semble fort douteux que la simplicite d'un modele soit TOUJOURS en correspondance directe avec la simplicite de l'activite modelisee. Et pourtant, c'est ce qui est pretendu, et pas uniquement de maniere implicite, dans les activites d'apprentissage autonome! Soyons quand meme plus juste : naturellement ce modele-ci est effectivement trop simple. C'est pourquoi, normalement, l'explication de l'execution d'un programme prend en compte un argument implicite supplementaire : la 'continuation', un argument precisant l'endroit de la suite du calcul (indiquant ce qu'il faut faire avec le resultat) ainsi que les mecanismes de liaison et de de-liaison des arguments-formels et des valeurs des arguments actuels.

Naturellement, meme le modele plus elabore est trop simple pour capter les fonds des mecanismes cognitifs actifs pendant la resolution de problemes. Je doute que des raffinements supplementaire de CE modele puissent exprimer beaucoup plus que des processus necessaires a la comprehension de resolution de problemes formels et mathematiques d'une part, ou, d'autre part, des processus d'abstraction precedant la resolution de problemes.

Je ne veux nullement nier de maniere definitive le bien fonde de l'hypothese que la programmation d'un probleme peut mettre en evidence des mecanismes generaux de la pensee. Mais encore faut-il que la pensee ne soit pas entravee par des contraintes inutiles! Toute l'evolution des langages de programmation est la pour illustrer ce point.

UN PETIT EXEMPLE DE PROGRAMME

Une deuxième solution intuitive serait que pour inverser une liste d'objets, il suffit de construire une nouvelle liste, où le premier élément est le dernier de la liste d'origine, le deuxième est l'avant-dernier de la liste d'origine, ... etc, jusqu'à ce que j'ai transféré tous les objets.



Ce deuxième algorithme peut se traduire en LISP en :

```
(DE INVERSE (L)(COND
  ((NULL L)())
  ((NULL (CDR L)) L)
  (T (CONS (CAR (INVERSE (CDR L)))
            (INVERSE (CONS (CAR L)
                            (INVERSE (CDR (INVERSE (CDR L))))))))))
```

ce qui représente un programme d'une élégance certaine, mais d'un pouvoir modélisant fort douteux. Le même programme en LOGO ne peut sûrement pas se vanter d'une clarté substantiellement accrue :

```
POUR INVERSE :L
  SI VIDE :L [OUTPUT :L]
  SINON SI VIDE SAUFPREMIER :L [OUTPUT :L]
  SINON [OUTPUT PHRASE LAST :L
        INVERSE SAUFDERNIER :L]
FIN
```

Rappelons encore une fois que rien n'exclut, naturellement, d'écrire une procédure LOGO plus simple (d'un point de vue informatique). L'unique but de ce petit programme est de montrer la difficulté d'adapter les idées intuitives de résolution de problèmes - même d'un problème aussi simple que l'inversion d'une suite d'objets - à un programme informatique. Très souvent, trop souvent encore, la programmation de la solution d'un problème s'écarte (doit s'écarter) considérablement des méthodes éprouvées et vérifiées dans la pratique "manuelle". Je prétends que, contrairement aux affirmations des défenseurs de LOGO et de l'apprentissage autonome, la programmation est encore une activité TRÈS difficile et TROP contraignante sur la démarche cognitive à suivre dans la modélisation de SES propres procédures cognitives.

La suite de cet article se centrera sur quelques propositions d'élargissement des capacités des interprètes, afin de rapprocher les solutions informatiques de la démarche cognitive.

FILTRAGE ET BASE DE DONNEES

4.0 FILTRAGE ET BASE DE DONNEES

Historiquement plusieurs voies ont ete poursuivies a partir du langage LISP : d'un cote s'est developpe tres tot le langage LOGO, de l'autre cote se sont developpes des langages tels que PLANNER [Hewitt 1969, Durieux 1981] et CONNIVER [McDermott & 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 utilises dans l'apprentissage autonome les constructions developpees dans ces langages, et tout d'abord un mecanisme de 'filtrage', mecanisme simple et puissant, dont certains neurophysiologues pretendent meme avoir trouve l'equivalent 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 mecanisme de filtrage avec un mecanisme de base de donnees.

4.1 Le Filtrage.

Le filtrage, ou 'pattern-matching', est un mecanisme permettant d'accéder a des objets non plus par un calcul positionnel, mais par une description de la structure des objets. Cette description est ensuite comparee 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 constitue d'une table et d'un ensemble de cubes sur la table. La description :

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

correspondra a la scene suivante :

```

+---+
| B |
+---+ +---+
| A | | C |
-----+---+---+-----
                    TABLE
```

Afin de savoir quel objet se trouve sur le cube A, classiquement, je devrais chercher a l'interieur de ma description un triplet commençant par le mot SUR et se terminant par le mot A, ensuite de prendre le mot au milieu et j'aurais trouve la reponse. Ce qui donne en LOGO :

FILTRAGE ET BASE DE DONNEES

```

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 :TRIPLET
                          STOP]]
FIN

```

avec le mecanisme de filtrage, la procedure SUR1? deviendra :

```

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

```

si (!), prealablement j'ai change 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

```

Detaillons : pour filtrer il faut un 'filtre' et une 'donnee'. La donnee peut etre n'importe quelle liste. Le filtre est soit une liste normale, dans ce cas on cherche une occurrence de ce filtre (qui n'est rien d'autre qu'une donnee), soit une liste contenant des variables precedees par le signe '!'. Ces variables sont considerees comme des 'fentes' a travers lesquelles on peut attraper un element se trouvant dans la donnee a la meme position. Le filtrage est donc un mecanisme de recherche d'elements. La puissance de ce mecanisme vient de sa 'lisibilite', la possibilite de comprehension 'visuelle' de ce qui est cherche. Voici quelques exemples de donnees et de filtres ainsi que les resultats du filtrage :

FILTRAGE ET BASE DE DONNEES

| filtre | donnee | resultat |
|-------------|-----------|----------------------|
| (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 interessant est que ce filtrage livre un mecanisme, visuellement comprehensible, de recherche d'objets. Il suffit de connaitre le modele 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 mecanisme sur une base de donnee ET pour le lancement de procedures, reprenons notre exemple de l'inversion d'une liste d'objets, et rappelons nous des deux algorithmes intuitifs que j'ai enonces.

Sans autre commentaire, je vais donner - en LISP enrichi du mecanisme de filtrage - les programmes correspondant a l'algorithme echangeant le premier et le dernier objet de la liste. Evidemment, le lecteur, apres une petite reflexion, peut VOIR le fonctionnement de ce programme et l'algorithme sousjacent.

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

Il est laisse au soin des lecteurs d'ecrire, dans ce style de programmation (rappelant des 'regles de reecriture') le programme implementant 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 variees, aussi bien comme operation de base pour la

FILTRAGE ET BASE DE DONNEES

recherche d'elements dans des bases de donnees que, comme nous verrons dans la suite, comme mecanisme de lancement de procedures. Je considere le mecanisme de filtrage comme plus fondamental que le mecanisme d'affectation des langages de programmation plus usuels : il le subsume et le generalise.

4.2 Mecanismes De Contextes

Je vais, suivant la terminologie du langage CONNIVER, appeler une combinaison de mecanismes de bases de donnees et de mecanismes de filtrage un 'contexte'. Pour mon expose, nous pouvons considerer une base de donnee comme une collection d'objets(1). Nous nommons les objets a l'interieur de la base de donnees des 'items'. Toute base de donnees doit posseder au moins deux operateurs, 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'operateur AJOUTE comme suit :

(AJOUTE '(SUR D C))

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

(ENLEVE '(SUR D C))

Naturellement, la description de notre micro-monde de cubes est representee comme une collection d'items a l'interieur de notre base de donnees.

Toute base de donnees permet egalement une question d'existence : 'est-ce que tel ou tel item se trouve dans la base de donnee?'. Cette interrogation sera implementee avec la fonction PRESENT. Ainsi, l'evaluation de

(PRESENT '(SUR A B))

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

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

(1) En realite, notre base de donnees aura naturellement une organisation plus sophistiquee que celle d'une collection.

FILTRAGE ET BASE DE DONNEES

```

      +---+
      | B |
      +---+ +---+
      | A | | C |
-----+---+---+---+-----
                TABLE

```

qui est representee 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'evaluer :

```
(AJOUTE '(SUR D B))
```

ce qui modifiera notre contexte de maniere a avoir la scene suivante :

```

      +---+
      | D |
      +---+
      | B |
      +---+ +---+
      | A | | C |
-----+---+---+---+-----
                TABLE

```

Voici a present quelques exemples d'appels de la fonction PRESENT avec leurs resultats 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

```

Il est bien evident que le filtrage, combine avec des mecanismes de bases de donnees, 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 a fait clair, je vais developper un petit programme simulant un robot dans ce micro-monde. Ce robot, s'appellant FREDDY, peut prendre et deplacer des cubes, a raison d'un seul cube a la fois.

FILTRAGE ET BASE DE DONNEES

Naturellement, nous avons tout d'abord a ecrire un programme qui assure que le cube que FREDDY doit deplacer est 'libre', c'est a dire que rien ne se trouve sur lui. Ce programme (en LISP enrichi de ces mecanismes) s'ecrit donc :

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

Remarquons, que nous suivons ici la methode de programmation 'top-down' : nous descendons du general vers le particulier. Ainsi, a l'endroit ou nous voulons assurer que l'objet Z soit place sur la table, nous appelons une autre routine (SUR-TABLE), pas encore ecrire, qui doit regler ce probleme. Nous connaissons le 'contrat' pour cette routine : placer, apres s'etre assure que Z est libre, cet objet sur la table. Nous pouvons l'ecrire 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 a des mises a jours de la base de donnees. Maintenant, nous pouvons ecrire 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 donnee a la nouvelle situation ou 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 probleme dans ce programme etait de se rappeler qu'il ne suffit pas de placer l'un des cubes sur l'autre, mais qu'il faut egalement enlever l'information sur la position precedente du cube!

Ce programme fournit une simulation tres sommaire du robot. L'unique probleme 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 maniere tres elegante d'introduire le robot, mais aussi afin de montrer une maniere de programmation completamente differente de celle utilisee dans des langages applicatifs usuels, nous allons d'abord elargir notre notion des mecanismes de contextes.

LES DEMONS

5.0 LES DEMONS

Jusqu'a maintenant nous connaissons les filtres comme descripteurs de la structure d'un item a chercher ou a consulter a l'interieur de la base de donnees. Ce qui nous a permis de programmer des procedures d'acces ou de consultation ou de test d'existence d'items en termes de description de leur structures. Il est relativement aise de s'imaginer les filtres comme descripteurs de situations : dans notre mini-exemple de robotique, chaque addition d'un item a la base de donnees et chaque enlevement d'un item de la base de donnees correspond a une situation particuliere dans notre micro-monde de cubes. Un test de 'presence' d'un filtre, correspond a un test de situation, a un moment instantane, dans ce micro-monde.

Tres souvent, dans la vie courante, les algorithmes ne sont point exprimes comme des descriptions de suite de processus actifs, l'un a la suite de l'autre, suivant un ordre pre-etabli, mais comme des actions a faire si telle ou telle situation se presente. Par exemple, afin de determiner si l'on doit manger, personne ne semble suivre l'algorithme suivant :

```

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

```

Et pourtant, cet algorithme ressemble tout a fait a un algorithme recursif classique. Les deux absurdites dans cet algorithme semblent etre :

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

Il nous semble plutot que nous avons un ensemble de petits processeurs tres specialises, travaillant en permanence, et ne se faisant sentir qu'a partir de l'instant ou une certaine limite est atteinte. Ainsi, j'ai l'impression que j'ai un processeur particulier, ne faisant (en parallele a l'activite de tous mes autres processeurs) rien d'autre que verifier si j'ai faim. L'activite de ce processeur ne devient consciente qu'a l'instant ou j'ai atteint une certaine limite de mon sentiment 'faim', et seulement a partir de cet instant la, mais la tres intensement, ce processeur me rappelle son existence et sa raison d'etre.

Ce que je viens d'ecrire est une sorte de calcul situationnel : attente, en 'background', d'une certaine situation, et a l'arrivee de cette situation, emission d'un signal activant un autre processeur (dans l'exemple : le processeur qui donne envie de manger). Exprime dans ce calcul-ci, l'algorithme de 'comment savoir quand manger' devient beaucoup

LES DEMONS

plus simple et pourrait s'ecrire :

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

Mais, du coup, je n'ai plus de boucle de controle. Je n'ai qu'un test de situation et une action attachee a chaque situation.

Pour revenir a notre programme exemple : rappelons nous que chaque situation est exprimee par un ensemble d'items bien precis dans la base de donnees. Chaque modification de situation est due, soit a un ajout d'un item dans le contexte, soit a un enlevement d'un item du contexte. Tout ce qui est necessaire pour pouvoir programmer des algorithmes s'exprimant en termes de situations et d'actions connectees a des situations est un mecanisme permettant d'associer des programmes a des items, ou, plus generalement, a des 'filtres'. De tels mecanismes sont appeles des 'demon' (en reference au 'demon de Maxwell').

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

```

+-----+
|  filtre  |
+-----+
|         |
| programme |
|         |
+-----+

```

ou le filtre est la condition d'activation du programme capable de decrire des items. Je peux associer des methodes a chacun des operateurs travaillant sur la base de donnees. Ainsi je peux associer des demons a l'operateur AJOUTE, j'appellerai de tels demons des demons SI-AJOUTE. Si le demon est associe a l'operateur ENLEVE, je parlerai d'un demon SI-ENLEVE, et si le demon est associe a l'operateur PRESENT, je parlerai d'un demon SI-BESOIN. L'activite d'un demon sera de surveiller les operations auxquelles il est associe et des que l'item, qui se trouve soit ajoute, soit enleve, soit recherche dans le contexte (dependant de l'operateur), correspond au filtre du demon, le demon s'active automatiquement : sans specification supplementaire. Ainsi le

fonctionnement des demons SI-AJOUTE peut etre decrit comme :

```

si AJOUTE doit placer un 'item' dans un 'contexte'
  d'abord les demons de type SI-AJOUTE regardent
  si
    leur filtre est capable de decrire l'item
    et si c'est le cas,
    leur programme est active
puis
  l'item est place dans le contexte

```

LES DEMONS

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

Utilisons alors ces mecanismes pour introduire le robot. Remarquons que, puisque nous n'avons pas de robot reel, il nous suffit de voir s'afficher sur l'ecran les activites du robot pendant le deplacement des cubes. Pour cela, nous allons introduire deux demons dans notre contexte du micro-monde de cubes : un pour reagir chaque fois qu'on enleve un fait de la base de donnees (la premiere maniere de changer de situation), qui sera de la forme :

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

disant juste que chaque elimination d'un fait de la base de donnees correspond a l'action de prendre quelque chose. La methode, que nous associons a l'ajout d'un item dans la base de donnees, exprime que cette action correspond a 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 a fait respectable pour la simulation d'un petit robot dans un micro-monde simplifie. Si, dans la situation suivante :

```

+---+
| D |
+---+
| B |
+---+ +---+
| A | | C |
-----+---+---+-----
                TABLE
```

nous evaluons l'appel

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

Le resultat 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 resultante, exprimee comme une suite d'items dans le

LES DEMONS

contexte, correspondra a la situation suivante :

| | | | | |
|-------|-------|-------|-------|-------|
| | | +---+ | | |
| | | A | | |
| | +---+ | +---+ | +---+ | |
| | B | C | D | |
| ----- | +---+ | +---+ | +---+ | ----- |
| | | TABLE | | |

J'invite le lecteur a verifier la correction de cet algorithme, et a le completer a volonte, pour le rendre plus general, et pour s'habituer a programmer de cette maniere-ci.

Le mecanisme de 'demon' est connu egalement sous le nom de 'lancement de procedure par filtrage' et est le processus de base de mecanismes d'inference des langages style 'PLANNER'. Il est - comme le filtrage lui-meme - integre de maniere standard dans tous les langages utilises dans les recherches en intelligence artificielle, permettant une programmation aise d'algorithmes non-hierarchiques. Suivant M. Minsky [Minsky 1982], il represente une maniere elegante et aise de represente des entite conceptuelles actives du cerveau. Il est egalement a la base de la notion d'acteur des langages du style PLASMA ou SMALLTALK.

J'ai implemente ces mecanismes de filtrage et de demon dans la version de VLISP-80 sur TRS-80 et ils sont utilises quotidiennement par les eleves de la SES avec laquelle nous collaborons. Ils les ont utilises 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 a celui donne ici en exemple, ainsi que pour un programme de consultation de base de donnees.

6.0 CONCLUSIONS

Naturellement, la raison de tout cet exercice n'etait nullement de demontrer quelques algorithmes, mais plutot, de montrer la difference fondamentale entre une programmation que j'ai envie de nommer 'conceptuelle' et la programmation algorithmique habituelle. Si nous prenons la metaphore de la programmation au serieux, si nous voulons utiliser la programmation pour modeliser le fonctionnement cognitif de la resolution de problemes, nous devons absolument nous separer 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 l'integrer dans les langages de programmation que nous voulons utiliser. LOGO et l'apprentissage autonome ne doivent pas devenir dependant des langages et de la technologie informatique,

CONCLUSIONS

mais doivent - au contraire - influencer et stimuler le développement des langages mis a disposition des jeunes gens.

Je me rappelle encore d'un vieux lexiques de mon enfance ou le cerveau etait represente fonctionnant comme un bureau administratif. Entre temps, j'ai vu des lexique plus modernes, representant le fonctionnement du cerveau comme un ordinateur sequentiel, ensuite le representant comme une sorte de reseau d'ordinateurs. La lecon a tirer de ces changements successifs de representation populaire du fonctionnement du cerveau est, peut-etre, que, tant que nous ne le connaissons pas mieux, nous ne devons pas nous fixer sur la maniere de le modeliser (et la modelisation des processus cognitifs en fait bien partie), mais laisser la voie ouverte pour explorer des chemins supplementaires et experimentaux. Il serait triste, que LOGO et l'apprentissage autonome se limitent au stade de developpement de l'informatique du debut des annees 70, pour aider a concretiser des processus cognitifs.

Le but de ce papier etait de suggerer que l'informatique et la psychologie cognitive se developpent continuellement, nous livrant continuellement des modeles plus puissants et plus accessibles. Integrons ces decouvertes, ces developpements dans nos langages et nos environnements informatiques, alors, peut-etre, la programmation pourra vraiment un jour effectivement nous aider a prendre conscience de nos propres procedures cognitives et intellectuelles.

7.0 REFERENCES

- Bossuet, 1982, 'L'ordinateur a l'ecole', PUF, Paris
- Broadbent, 1961, 'Behaviour', Basic-Books, New-York
- Broadbent, 1970, 'Visual Behaviour', Proc. Royal Society Britain, Londres, no. 175, pp 333-350
- Cointe, 1982, 'Fermetures dans les Lambda-Interpretes. Application aux Langages LISP, PLASMA et SMALLTALK', these de troisieme cycle, Universite Paris 6, rapport de recherche LITP no. 82-11
- Durieux, 1981, 'Semantique des liaisons nom-valeur : application a l'implementation des lambda-langages', these, Universite Paul Sabatier, Toulouse

REFERENCES

- Goldberg & Ross, 1981, 'Is the Smalltalk-80 System for Children?', BYTE, vol. 6, no. 8, Aout 1981, pp.. 348-368
- Greussay, 1976, 'Descriptions compactes d'interpretes implementables : une application au langage Conniver', dans : programmation, ed. B. Robinet, Dunod, Paris, pp. 218-231
- Hewitt, 1969, 'PLANNER : a language for manipulating models and proving theorems in a robot', 1st International Joint Conference on Artificial Intelligence, Washington D.C.
- Mathieu, 1981, 'Resume de l'experience SES', Rapport Simon, Annexe 2, La Documentation Francaise, Paris
- Minsky, 1982, 'Learning Meaning', draft, Artificial Intelligence Laboratory, M.I.T., Juillet 1982
- McDermott, Sussman, 1974, 'The CONNIVER reference manual', MIT Artificial Intelligence Laboratory, AI-memo no 259a
- Papert, Watt, DiSessa, Weir, 1979, 'Final report of the Brookline LOGO project', Memo LOGO, MIT-LOGO laboratory
- Wertz, Perolat, Mathieu, 1979, 'L'experience d'Arc et Senans', Rapport Interne, Dept. Informatique, Universite Paris 8

Remerciements :

Les idees exprimees dans ce papier ont ete largement influencees par M. Patrick Greussay. Ces recherches sont soutenues partiellement par l'Agence pour l'Informatique, no. de contrat ADI 81/331, partiellement par le Centre National de la Recherche Scientifique et le CNET, no. de contrat ATP 4097, et partiellement par la RCP-LOGO.