ARTEFACT

LES SYSTEMES EXPERTS (II):
UN MOTEUR D'INFERENCE EN PASCAL

Les programmes d'Intelligence Artificielle sont souvent très complexes. Il est cependant relativement facile d'écrire de petits logiciels « intelligents ». Et pour comprendre comment fonctionne un système expert, le mieux n'est-il pas d'en programmer un ? C'est à cette activité que nous vous convions dans ce numéro, en vous proposant un petit système expert de diagnostic de pannes.

Un système expert comprend, outre son moteur d'inférence, une base de connaissance qui lui permet d'accomplir sa tâche dans un domaine particulier. En effet, sans contenu, un moteur d'inférence ne sert à rien. Une tête bien faite, mais vide.

Le diagnostic est l'une des applications les plus fréquentes des systèmes experts, et certainement l'un des champs d'activités dans lesquels ils sont le plus à l'aise. C'est pourquoi nous avons choisi de tester ce petit système expert dans un domaine bien simple : le diagnostic de petites pannes sur un micro-ordinateur. La plupart des manuels d'utilisation présentent quelques-unes des pannes les plus classiques et les moyens d'y remédier. Evidemment, les pannes considérées sont toujours simples, dans tous les cas il ne s'agit que de vérifier si tous les périphériques ont été connectés, et si toutes les procédures de branchement ont bien été suivies.

La base de connaissance qui

```
la prise de courant fonctionne
les plombs sont en état de marche
le disjoncteur est en marche
le contraste est réglé
l'intensite est réglée
fin


si
la prise de courant fonctionne
les plombs sont en état de marche
le disjoncteur est en marche
alors
le courant arrive
fin

si
le courant arrive
l'interrupteur est sur marche
alors
l'ordinateur est sous tension
fin

si
l'ordinateur est sous tension
le moniteur est branché
le moniteur est bien réglé
il y a le curseur á l'écran
la disquette boote
alors
l'ordinateur est prêt á fonctionner
fin

si
l'ordinateur est sous tension
le lecteur de disque fonctionne
la disquette est la disquette master
alors
la disquette boote
fin

si
le moniteur est branché
le contraste est réglé
l'intensite est réglée
alors
le moniteur est bien réglé
fin
```

Fig. 1. - La connaissance du système : sa base de faits et sa base de règles.

a été introduite est assez rudimentaire, mais il est très facile de l'augmenter : la disposition des informations dans le désordre est justement l'une des principales qualités des systèmes à règles de production.

Ces règles servent à définir un état de bon fonctionnement à partir d'un ensemble de critères pour caractériser cet état de bon fonctionnement. Par exemple, pour que l'ordinateur soit sous tension, il est nécessaire que le courant arrive, et que l'interrupteur soit sur marche.

La **figure 1** montre la base des faits initiaux, et la base de connaissance qui a été introduite à titre de démonstration. Un exemple de fonctionnement en chaînage arrière est donnée **figure 2.** A partir d'une hypothèse initiale que l'on désire confirmer, le système cherche toutes les règles qui peuvent lui être utiles, et tous les faits qui permettent de prouver ou de réfuter une hypothèse. Lorsqu'il lui manque des informations, il questionne l'utilisateur sur la validité d'un fait. Par exemple, il demandera si l'interrupteur est sur marche. L'utilisateur devra alors vérifier si celui-ci est effectivement sur la bonne position, et répondre « oui » une fois l'action effectuée.

```
D)éduit C)onfirmer A)fficher S)auvegarder Q)uit  c

Quelle est l'hypothèse que vous voulez vérifier
? l'ordinateur est prêt à fonctionner

le fait: la disquette est la disquette master est-il vrai (o/n)? o
on place le fait: la disquette est la disquette master

le fait: le lecteur de disque fonctionne est-il vrai (o/n)? o
on place le fait: le lecteur de disque fonctionne

le fait: l'interrupteur est sur marche est-il vrai (o/n)? o
on place le fait: l'interrupteur est sur marche

le fait: il y a le curseur a l'ecran est-il vrai (o/n)? o
on place le fait: il y a le curseur à l'écran

le fait: le moniteur est branché est-il vrai (o/n)? o
on place le fait: le moniteur est branché

l'hypothèse: l'ordinateur est prêt à fonctionner a été confirmée
```

Fig. 2. - Un exemple de fonctionnement en mode diagnostic du système. L'hypothèse que l'ordinateur est prêt à fonctionner est vérifiée. Lors de cette confirmation, le programme pose à l'utilisateur des questions sur la validité de certains faits.

Un système expert d'ordre zéro

Nous avons vu lors de notre dernier article l'organisation générale d'un système expert. Sa structure est découpée en trois parties distinctes : la base des faits, ou base de travail, la base des règles, qui contient la « connaissance » dont un programme dispose pour mener à bien sa tâche dans un domaine d'expertise particulier, et le moteur d'inférence, véritable noyau du logiciel, qui s'emploie à mener à bien l'application de ces règles.

Rappelons que les règles d'un système expert sont généralement de la forme :

si condition **alors** action

où les parties condition et action sont constituées d'un ensemble de faits.

Deux stratégies de raisonnement sont possibles : dans la première, que l'on nomme **chaînage avant,** le système tente de déterminer tous les faits possibles que l'on peut déduire à partir d'un ensemble de données initiales ; dans la seconde, intitulée **chaînage arrière,** le programme cherche à prouver si des faits (considérés comme hypothétiques) peuvent être déduits de la base de faits. Dans ce cas, le fonctionnement s'effectue à rebours, de l'hypothèse vers les faits.

Le programme qui est proposé ici correspond à un moteur d'inférence d'ordre 0. L'ordre d'un système expert détermine la nature des informations que ces règles peuvent traiter. « Ordre 0 » signifie que le système ne prend pas en compte les variables et que, de ce fait, une règle n'est constituée que d'énoncés. En revanche, lorsqu'un système est d'ordre supérieur à 1, il autorise l'utilisation de variables à l'intérieur des règles. La notion d'ordre découle directement de la logique mathématique : le calcul propositionnel est dit d'ordre 0, alors que le calcul des prédicats relève d'un ordre supérieur (on parle généralement de la logique du premier ordre).

Par exemple, une règle du type :

si humain(x) alors mortel(x)

ne peut se représenter directement dans un système d'ordre 0. Il ne faut cependant pas en conclure qu'ils sont dénués d'intérêt. En effet, lorsque les variables ne correspondent qu'à une seule entité, il est possible d'écrire un système sans faire intervenir de variables.

Par exemple, dans un système expert qui a pour fonction de classer un individu à partir de ses attributs, la règle

si animal(x) et
bipède(x) et
sans(plume, x)
alors humain(x)

peut se réécrire de la manière suivante :

si individu est un animal et
individu est un bipède et
individu est sans plume
alors individu est humain

La structure du programme

Le moteur d'inférence, dont le listing est présenté **figure 3**, fonctionne aussi bien en chaînage avant qu'en chaînage arrière.

La structure du programme a été simplifiée au maximum afin de faire ressortir les éléments importants : les procédures d'application des règles. Le programme général passe la main à un menu, après avoir chargé en mémoire la base des faits et l'ensemble des règles. Ces procédures de chargement supposent que les faits et les règles sont écrites sous un format très strict : ces différentes informations sont placées sur un même fichier. Chaque fait doit se trouver sur une ligne différente (l'écriture d'un fait est donc limitée à 80 caractères), et l'ensemble des faits doit être terminé par le symbole 'fin'. Attention, les faits étant placés dans des chaînes de caractères (type STRING) sans manipulation préalable, les blancs, aussi bien aux extrémités qu'à l'intérieur des phrases, sont pris en considération lors de la mise en correspondance des faits et des règles.

Les règles doivent être formulées comme le montre la **figure 1**. Les symboles 'si', 'alors' et 'fin' doivent se trouver en début de ligne et encadrer les parties condition et action des règles. Il n'est fait aucune analyse syntaxique ni aucune récupération des erreurs. Cependant, le nombre de lignes intermédiaires ne joue aucun rôle : les règles peuvent être séparées par autant de sauts de lignes que désiré.

L'utilisateur devra donc faire bien attention à la manière dont sont introduites les règles. Ces restrictions n'ont qu'un seul but : conserver sa légèreté au programme, sans l'alourdir de nombreux tests syntaxiques. Il est toutefois possible (et même souhaitable) d'écrire ses propres procédures de chargement : le système n'en sera que plus « convivial ».

```
program syteme_expert;

(*******************************************)
(*                                         *)
(* un systeme expert d'ordre 0             *)
(* qui fonctionne aussi bien en chainage avant *)
(* qu'en chainage arriere                  *)
(*                                         *)
(*            (c) J.Ferber                 *)
(*                                         *)
(*******************************************)
type
  pcell=^cell;
  pregle=^regle_type;
  pfait=^string;

  regle_type = record
    condit: pcell;
    action: pcell;
   end;

  cell = record
    next:pcell;
    case integer of
       1 : (rval : pregle);
       2 : (fval : pfait);
       3 : (cval : pcell);
   end;

var
  base :pcell;        (*   la base de faits  *)
  regles : pcell;     (* l'ensemble des regles *)
  questions : pcell; (* les faits negatifs poses
                        a l'utilisateur *)
  faits : pcell;      (* la liste des enonces *)
  c:char;
  fich,fichout:text;

(* les fonctions de manipulation des faits et regles *)

function membre(f:pfait;b:pcell):boolean;
  begin
    membre:=true;
    while b <> NIL do
      if f = b^.fval then
        exit(membre)
      else
        b:=b^.next;
    membre:=false;
  end;

function consfait(x:pfait; y:pcell):pcell;
  var p: pcell;
  begin
    new(p);
    p^.fval:=x;
    p^.next:=y;
    consfait:=p;
  end;

function consregle(x:pregle; y:pcell):pcell;
  var p:pcell;
  begin
    new(p);
    p^.rval:=x;
    p^.next:=y;
    consregle:=p;
  end;

function existe (fait:pfait):boolean;
  var lstfaits:pcell;
  begin
    lstfaits:=base;
    existe:=true;
    while lstfaits <> NIL do
      if fait = lstfaits^.fval then
        exit(existe)
      else
        lstfaits:=lstfaits^.next;
    existe:=false;
  end;

function placer (fait:pfait):pfait;
  begin
    if existe (fait) then
      placer:=NIL
    else begin
      base:=consfait(fait,base);
      placer:=fait;writeln;
      writeln('on place le fait :',fait^);
    end;
  end;

function test(regle:pregle):boolean;
  var premisse:pcell;
  begin
    premisse:=regle^.condit;
    while premisse <> NIL do
      if not existe(premisse^.fval) then
        begin
          test:=false;
          exit(test);
        end
      else
        premisse:=premisse^.next;
    test:=true;
  end;

function utilise (regle:pregle):boolean;
  var conclusion:pcell;
  begin
    conclusion:=regle^.action;
    utilise:=false;
    while conclusion <> NIL do
      begin
        if placer(conclusion^.fval) <> NIL then
          utilise:=true;
        conclusion:=conclusion^.next;
      end;
  end;

function applic_regle (regle:pregle):boolean;
  begin
    applic_regle:=false;
    if test(regle) then
      applic_regle:=utilise(regle);
  end;

function chain_avant:boolean;
  var lstregles:pcell;
  begin
    lstregles:=regles;
    while lstregles <> NIL do
      if applic_regle(lstregles^.rval) then
        begin
          chain_avant:=true;
          exit(chain_avant)
        end
      else
        lstregles:=lstregles^.next;
    chain_avant:=false;
  end;

procedure deduit;
  begin
    while chain_avant do;
  end;
```

Fig. 3. - Le listing du petit système expert écrit en Pascal.

```
(* le chainage arriere *)

function verifier(fait:pfait):boolean; forward;

function prouver(regle:pregle):boolean;
  var premisses: pcell;
  begin
    prouver:=false;
    premisses := regle^.condit;
    while premisses <> NIL do
      begin
        if not verifier(premisses^.fval) then
          exit(prouver);
        premisses:=premisses^.next;
      end;
    prouver:=true;
  end;

function verifier;
  var lstregles:pcell;
      f:pfait; c:char;
  begin
    verifier:=true;
    lstregles:=regles;
    if membre(fait,base) then exit(verifier);
    while lstregles <> NIL do
      begin
        if membre(fait,lstregles^.rval^.action) then
          if prouver(lstregles^.rval) then
            exit(verifier);
        lstregles:=lstregles^.next;
      end;
    if membre(fait,questions) then
      begin
        verifier:=false;
        exit(verifier);
      end
    else
      begin
        writeln;
        write('le fait :',fait^,' est il vrai (o/n)?');
        read(c);
        if(c = 'o') or (c = 'O') then
           begin
             f:=placer(fait);
             verifier:=true;
           end
         else
           begin
             questions:=consfait(fait,questions);
             verifier:=false;
           end;
      end;
  end;

(* les fonctions d'entrees sorties *)

procedure lire(var s:string);
  var c:char;
  begin
    readln(fich,s);
    while (not eof (fich)) and (s='') do
      begin
        readln(fich,s);
      end;
  end;

function lire_fait(var s:string):pfait;
  var lstfaits:pcell;
      f:pfait;
  begin
    writeln ('-->',s);
    lstfaits:=faits;
    while lstfaits <> NIL do
      if lstfaits^.fval^=s then
        begin
          lire_fait:=lstfaits^.fval;
          exit(lire_fait);
        end
      else
        lstfaits:=lstfaits^.next;
    new(f);f^:=s;
    faits:=consfait(f,faits);
    lire_fait:=f;
  end;

procedure lire_regle;
  var lstfact:pcell;
      f:pfait;
      r:pregle;
      ln:string;
  begin
    new(r);
    lstfact:=NIL;
    repeat
      lire(ln);
    until eof(fich) or (ln = 'si') or (ln = 'SI');
    if eof(fich) then exit(lire_regle);
    lire(ln);
    while (ln <> 'alors') and (ln <> 'ALORS') do
      begin
        f:=lire_fait(ln);
        lstfact:=consfait(f,lstfact);
        lire(ln);
      end;
    r^.condit:=lstfact;
    lstfact:=NIL;
    lire(ln);
    while (ln <> 'fin') and (ln <> 'FIN') do
      begin
        f:=lire_fait(ln);
        lstfact:=consfait(f,lstfact);
        lire(ln);
      end;
    r^.action:=lstfact;
    regles:=consregle(r,regles);
  end;

procedure lire_base;
  var ln:string;
      f:pfait;
  begin
    lire(ln);
    while (ln <> 'fin') and (ln <> 'FIN') do
      begin
        f:=lire_fait(ln);
        base:=consfait(f,base);
        lire(ln);
      end;
  end;

procedure afficher;
  var c:char;

  procedure affich(l:pcell);
    var i:integer;
    begin
      i:=1;
      while l <> NIL do
        begin
          writeln(i,' ',l^.fval^);
          l:=l^.next;
          i:=i+1;
        end;
    end;
```

Fig. 3 (suite).

```
  begin
    repeat
      writeln;
      write(' B)ase R)egles E)nonces Q)uit ');
      read(c);writeln;
      case c of
      'b','B': affich(base);
      'r','R': writeln ('bof');
      'e','E': affich(faits);
      end;
    until (c='Q') or (c = 'q');
  end;


procedure sauverbase;
  var lst: pcell;
  begin
    rewrite(fichout,'baseout');
    lst:=base;
    while lst <> NIL do
     begin
      writeln (fichout,lst^.fval^);
      lst:=lst^.next;
     end;
    close (fichout);
  end;


procedure confirmer;
  var f:pfait;
      ln:string;
  begin
    writeln;
    writeln('Quelle est l''hypothese que vous voulez
    verifier');
    write('? ');
    readln(ln);
    if ln='' then exit(confirmer);
    f:=lire fait(ln);
    f:=lire_fait(ln);
    if verifier(f) then
      begin
        write ('l''hypothese ');
        write (f^); writeln(' a ete confirmee');
      end
    else
      writeln('hypothese non confirmee ');
    writeln;
  end;


begin
  base:=NIL;
  regles:=NIL;
  faits:=NIL;
  questions:=NIL;
  reset(fich,'base.text');
  lirebase;
  while (not eof(fich)) do
    lire_regle;
  repeat
    writeln;
    write ('D)deduit C)onfirmer A)fficher S)auvegarder
    Q)uit');
    read(c); writeln;
    case c of
      'c','C': confirmer;
      'd','D':deduit;
      'a','A': afficher;
      's','S':sauverbase;
    end;
  until (c ='q') or (c ='Q');
end.
```

Fig. 3 (suite et fin).

Une fois ce chargement effectué, l'utilisateur dispose d'un certain nombre d'options de traitement : déduire tous les faits possibles à partir de la base initiale, sauver la nouvelle base dans un fichier intitulé BASEOUT, confirmer une hypothèse, ou bien afficher la base des faits à l'écran.

Examinons maintenant la manière dont sont représentés les faits et les règles de manière interne.

La liste linéaire est la structure de donnée la plus employée dans ce programme. Elle consiste en une suite de doublets (appelés CELL), composés chacun de deux pointeurs. Le premier pointe vers un autre doublet, et le second vers une règle ou un fait (**fig. 4**).
Une liste linéaire se manipule facilement, mais seul le premier doublet d'une liste est accessible directement. Par exemple, soit P une liste de doublets, on obtient la partie valeur du premier doublet de la liste à l'aide des instructions :
P↑.fval ou P↑.rval
selon qu'il s'agit d'un fait ou d'une règle. Pour accéder à l'élément suivant, il suffit de faire :
P := P↑.next

Ajouter un élément dans une liste s'effectue en attachant un nouveau doublet en tête de la liste. Les deux fonctions CONSFAIT et CONSREGLE servent à cet usage (l'en-tête CONS provient de leur analogie avec la fonction LISP du même nom). Ces deux fonctions prennent deux arguments, l'objet à insérer et la liste sur laquelle est appliquée l'insertion, et retournent un résultat, un pointeur sur cette liste.

Les règles se présentent sous la forme d'une structure composée de deux listes de faits : la première caractérise la condition d'application de la règle et la seconde définit son résultat. L'utilisation du pointeur permet de partager les faits identiques, afin d'optimiser la place, et d'accélérer le mécanisme d'unification des règles sur la base, en ne comparant que des pointeurs et non des chaînes de caractères.

Par exemple, imaginons que le système ne comporte que deux règles et trois faits, comme le montre la **figure 5-a.** Leur structure interne est donnée **figure 5-b.**

Le moteur d'inférence

En chaînage avant, l'algorithme du moteur d'inférence est extrêmement simple : le système applique les règles jusqu'à saturation, c'est à dire tant qu'il est possible de déduire de nouveaux faits. Il se présente dans le programme sous la forme de trois modules : DEDUIT, CHAIN AVANT et APPLIC REGLE.

Les procédures DEDUIT et CHAIN AVANT sont responsables du mécanisme général de déduction. La première se contente d'appeler CHAIN AVANT en permanence, tandis que la seconde parcourt la liste des règles en les essayant toutes les unes après les autres.

La fonction APPLIC REGLE se charge de tester une règle sur la base. Si le test est positif, c'est à dire si tous les faits qui se trouvent dans la partie condition de la règle sont aussi présents dans la base, alors les faits de la partie action sont ajoutés.

L'opération de test est réalisée par la fonction TEST. Si

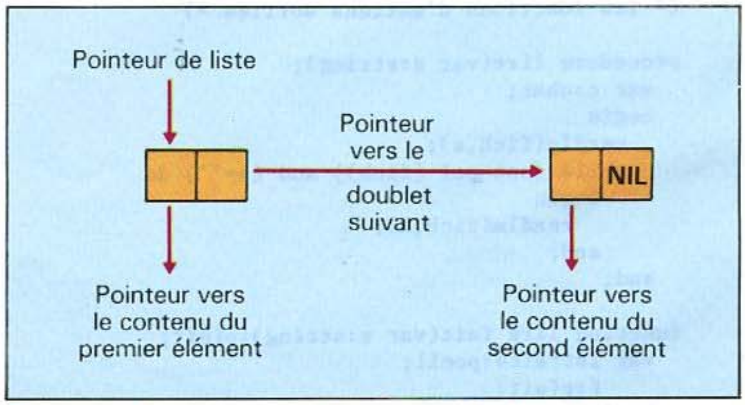


Fig. 4. - Une liste est composée d'une suite de doublets reliés entre eux par l'intermédiaire de pointeurs. La fin d'une liste est indiquée par le symbole NIL.

toutes les prémisses de la règle se trouvent dans la base, alors elle retourne la valeur vraie (TRUE), sinon c'est un échec ; elle retourne alors FALSE. Pour mener à bien sa tâche, elle utilise la fonction EXISTE, qui vérifie la présence d'un fait.

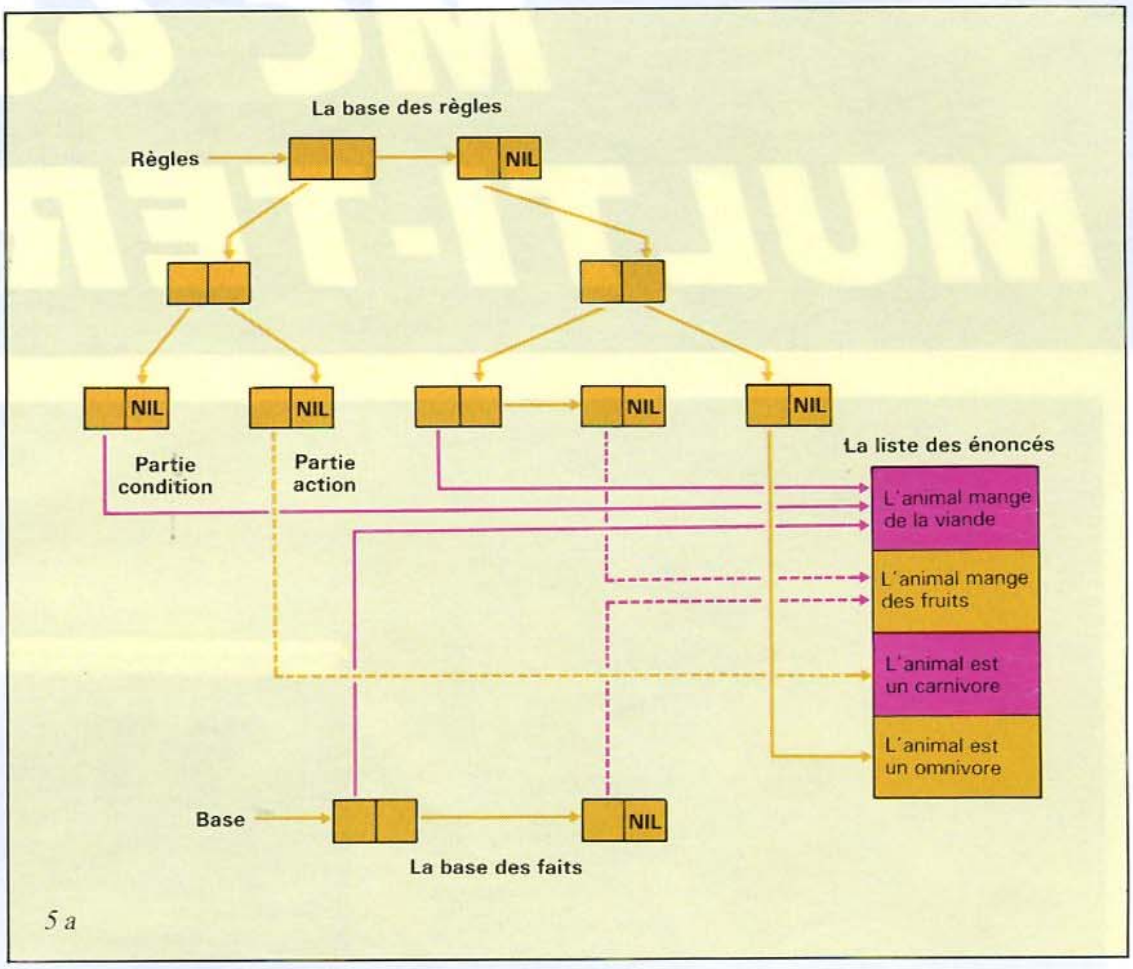
Lorsqu'une règle est activée, sa partie action est ajoutée à l'ensemble des faits de la base, par l'intermédiaire des fonctions UTILISE et PLACER.

Le fonctionnement en chaînage arrière est un peu plus compliqué, car il utilise un algorithme récursif. Les deux fonctions fondamentales sont VERIFIER et PROUVER. La première vérifie la véracité d'un fait qui lui est passé en argument. Pour qu'un fait soit vrai, il suffit qu'il appartienne à la base de connaissance ou qu'il soit membre de la partie action d'une règle. Dans ce dernier cas, il demande à la seconde procédure de prouver que la règle est applicable, c'est à dire s'il est possible de vérifier toutes les prémisses de cette règle. Lorsqu'un fait qui demande à être vérifié ne se trouve dans aucune règle, un message est envoyé à l'utilisateur, lui demandant de spécifier si le fait en question est vrai ou non.

Ces deux fonctions s'appelant mutuellement laissent au langage Pascal le soin de traiter la récursivité. Cette pratique est envisageable pour des systèmes experts « jouets » comme celui qui est présenté ici. Dans un contexte professionnel, le programme devrait prendre lui-même en compte la récursivité, afin qu'une définition maladroite des règles affiche un message d'erreur sans « planter » le système.

Adaptation et améliorations

Ce programme a été écrit sur Pascal UCSD (version 1.1) et testé sur la version Apple II. Il sera transportable directement sur toutes les machines disposant du Pascal UCSD. Pour les détenteurs d'une ancienne version de l'UCSD, une petite modification mineure doit être effectuée : après chaque instruction READLN(LN), il faut rajouter un READ(C) où C est une variable de type caractère.



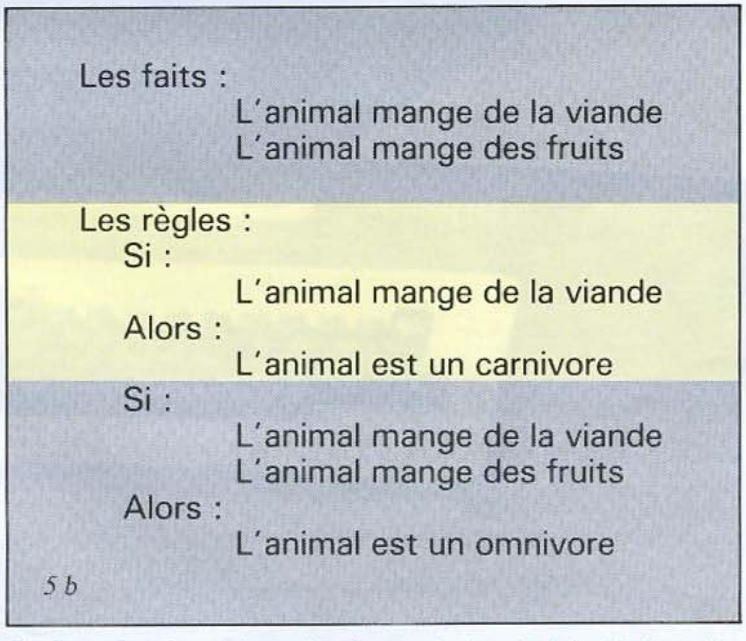


Fig. 5 a et b. – La représentation d'une petite base de faits (a) de manière interne est réalisée à l'aide de doublets et de pointeurs vers des chaînes de caractères (b).

Pour les autres compilateurs Pascal, deux difficultés peuvent se présenter : l'absence du type STRING et de l'instruction EXIT. La première ne pose pas de problème majeur : il suffit de se constituer son propre type STRING, et de l'adapter à la circonstance en écrivant des procédures de lecture, d'écriture et surtout de comparaison de deux chaînes de caractères. La seconde est plus gênante. L'instruction EXIT, telle qu'elle est employée ici, a pour but de faire sortir le programme de la procédure en cours. Vous devrez donc remplacer chaque EXIT par un GOTO vers une étiquette correspondant au dernier END de la procédure en question.

Ce programme peut être amélioré de plusieurs façons sans devoir modifier la structure des procédures de raisonnement. Par exemple, il serait intéressant de disposer d'une petite commande permettant de supprimer ou d'ajouter un fait dans la base de connaissances. Donner un nom aux règles de manière à pouvoir les modifier 'on line' serait aussi utile. Enfin, il serait agréable de pouvoir lire la base de faits et celle des règles à partir de fichiers séparés.

D'autres améliorations sont possibles, mais elles réclament une bonne compréhension du programme : réaliser un mécanisme de justification (pourquoi a-t-on besoin de tel ou tel fait, pourquoi obtient-on tel ou tel résultat) ou transformer ce moteur d'ordre 0 en moteur d'ordre 1 constituent des tâches difficiles mais à bien des égards passionnantes... ■

J. FERBER