Definite Clause Dictionary
- - -Prologによる辞書項目記述と意味処理

田中穂積
池田光生
奥村　学
（東京工業大学工学部）

１．はじめに

これまでの研究から、記号処理用言語 Prolog は、自然言語処理、とりわけ構文処理に極めて整合性の良い言語であることが明らかにされている[2]。一方、Prolog による意味処理については、研究が緒についた段階にあり、Prolog との整合性の良さが、必ずしも明確とはいえなかった。しかし、意味処理用の辞書の形式を、我々の提案する形式（この形式をＤＣＤ（Definite Clause Dictionary）とよんでいる）で記述することにより、意味処理プログラムの中核を、Prolog に組み込みの機構で代用することができる。このことは、意味処理にも Prolog が整合性の良い言語であることを意味している。２章では、従来の意味処理技術を簡単に説明する。３章では Prolog と意味処理の関係について説明する。４章ではＤＣＤの基本的な考え方を説明し、それによりどのようにして意味処理用プログラムの中核を、Prolog の基本計算機構でおきかえ可能かを説明する。第５章では、英語と日本語の文を意味処理した結果について述べる。第６章ではまとめと今後の課題について述べる。

２．意味処理の基本的な考え方

意味処理の基本となるのは、辞書項目の記述である。辞書項目の形式として最もよく用いられるものは、フレームによる表現である。フレームは、スロットの集合からできている。我々がこれまで用いてきたＳＲＬによる辞書項目記述例（open）を図１に示す[1]。

```
[open,frame,
     [subj,[[human,=agent],
            [[$OR,event,thingOpen],=object],
            [instrument,=instrument],
            [wind,=reason]]],
     [obj,[$OR,event,thingOpen],=object],
     [with,instrument,=instrument],
     [self,act]]
```

図１　ＳＲＬによる辞書項目openの記述例

図１中の [obj,[$OR,event,thingOpen],=object]はスロットの一例である。一般に、スロットは、スロット名(obj)、制約条件([$OR,event,thingOpen])、アクション(=object)の３つ組から構成される。ただし、subjスロットは、制約条件部が制約条件とアクションの対を複数個並べたものになっている。ここで制約条件は、このスロットを満たすことのできるフレーム（これをフィラーとよぶ）の意味的性質を記述する。以下これを意味的制約条件と呼ぶ。

ＳＲＬでは、フィラーが文中で果たす統語的制約条件をスロット名として記述している。したがって上記の objスロットを満たしうるフィラーは、文中で目的格(obj)の位置を占め、”事象”(event)もしくは($OR)”開くもの(thingOpen)”でなければならない、ということが記述されている。フィラーがこのスロットを満たすと、=object という名称のアクションを起動して、フィラーの深層格が対象格(object)である、という意味を抽出する。

図１の辞書記述を用いて行なう意味処理の手順は、およそ次のようにして進行する。ただし、スロットは先頭から順に選択されるものとする。

(i) 選択されたスロットのスロット名が self なら、そのスロットに書き込まれている上位概念のフレームを取り出し(i)へ。さもなければ(ii)へ。

(ii)選択されたスロットの（統語的ならびに意味的）制約条件をフィラーが満たせばアクションを起動して終了、さもなければ(iii)へ。

(iii) 次のスロットがあればそれを選択して(i)へ。さもなければ意味処理失敗ルーチンへ。

(i) では、self スロットを介して、上位概念のもつスロットをアクセスする。それにより、上位概念のもつ性質を下位に伝播させることができる。これは、人工知能の研究分野で知識の遺伝 (inheritance of knowledge) とよばれる技術を利用していることになる。(ii)と(iii) は、バックトラックによるスロット選択であると見なせる。

意味処理プログラムの中核は、フレームに対して(i) から(ii)に記述した操作を行なうフレームインタープリタである。意味的制約条件の検査を行なうプログラムもそこに含まれる。たとえば [$OR,event,thingOpen] という意味的制約条件を door というフィラーが満たすことが可能かどうかは、$OR の解釈を行ない、door が event か、もしくは thingOpen を含意するかどうかを検査するプログラムを用意しなければならない。そのための意味処理プログラムは比較的複雑なものとなる。

3. Prolog による意味処理

2. で述べた意味処理の手順を観察すると

(i) バックトラックによるスロットの選択を行なっている
(ii)意味的制約条件は、論理式で表現されている
(iii) 知識の遺伝はユニフィケーションによるサブゴールの展開として解釈可能

であることがわかる。

Prolog による意味処理を考える時、

[a] (i) は、Prolog の基本計算機構で代用可能
[b] (ii)は、Prolog プログラムとして直接表現可能

なことが分かる。

最後の(iii) を例により説明する。

```
a :- b ; c.
d :- a.
```

に対して

?- d.

なるゴールを実行することを考える。ホーン節のヘッドとのユニフィケーションにより、このゴールは最終的に

?- b ; c.

というサブゴールに展開される。このことは、a に対して d が下位概念であるとすれば、d から d の上位にある a のボディの知識 "b;c" が参照（利用）可能なことを意味している。これは人工知能の研究で知識の遺伝といわれていたものに他ならない。したがって

[c]（iii）は Prolog の基本計算機構で代用可能

であると結論することができる。

以上のことから意味処理の中核となるプログラムを Prolog の基本計算機構で代用可能なことが読み取れる。そのためには、意味処理に用いる辞書項目を、リスト構造としてではなく、ホーン節として記述しておく必要がある。そこで４章では、どのようなホーン節の形式にすべきかを検討し、ＤＣＤ(Definite Clause Dictionary)とよばれる辞書記述形式を提案する。

４．辞書記述形式ＤＣＤを用いた意味処理

第三章では、ホーン節形式による辞書項目記述を行なうことにより、意味処理の中核となるプログラムを、Prolog の基本計算機構で代用することが可能かどうかを検討した。本章で提案するＤＣＤ形式の辞書項目記述によれば、実際それが可能なことを実例により示す。はじめにＤＣＤによる辞書項目記述例を説明し、それにより意味処理の中核となるプログラムをすべて Prolog の基本計算機構に任せることが可能なことを示す。

図１のリスト形式による辞書項目の記述は、図２に示すホーン節で置き換えることができる。図２に示す辞書の記述形式を我々はＤＣＤと呼んでいる。

そして図２の記述で特徴的なことは、述語　sem をヘッドに持つホーン節（以後これをＤＣＤと呼ぶ）の並びによって、スロットの並びが表現されていることである（図２中の変数Ｎは、当該スロットのフィラーであることに注意）。それによりスロットの選択が、Prolog　のバックトラックの機構を利用して自動的になされる。図２の最後のＤＣＤ節は、フィラーＮがそれより前のスロットを満たすことができなかった時に起動され、それにより open の上位概念 act　にあるスロットを選択することができる。

図２の各ＤＣＤ節のボディ には、意味的制約条件が記述されているが、意味的制約条件は述語 sem　の任意の論理結合式として表現することができる。

図２のボディ に書かれている addProp　は、意味構造を抽出するための述語である。図２の例では、単に深層格を抽出しているに過ぎないが、この部分を書きかえることにより、任意の意味構造形式（たとえば Montague 文法で使う内包論理式など）を抽出することができる。

以上から、図２に示すＤＣＤ形式の辞書項目記述により、意味処理手順の大半を、Prolog　に内蔵された基本計算機構で代用可能なことが理解できよう。

```
: -op(100,yfx,'^'),op(100,yfx,':').
sem(open,subj:N^In^Out):-nonvar(N),
        (sem(N,human),
        addProp(agent:N^In^Out)
        ;
        (sem(N,event);
         sem(N,thingOpen)),
        addProp(object:N^In^Out)
        ;
        sem(N,instrument),
        addProp(instrument:N^In^Out)
        ;
        sem (N,wind),
        addProp(reason:N^In^Out),!.
sem(open,obj:N^In^Out) :-nonvar(N),
        (sem(N,event);
         sem(N,thingOpen)),
        addProp(object:N^In^Out),!
sem(open,with:N^In^Out) :-nonvar(N),
        sem(N,instrument),
        addProp(instrument:N^In^Out),!.
sem(open,Prop) :-
        sem (act,Prop).
sem(X,X).
```

図２．ＤＣＤによる辞書項目の記述

以上をまとめると、次のことがわかる。図２では、

(i) 一つのスロットを、sem 述語をヘッドとする一つのホーン節で表現する。これをＤＣＤ節とよぶ。
(ii)スロットの並びは、ＤＣＤ節の並びとして表現する。それによりスロットの選択が Prolog のバックトラック機構により自動的になされる。
(iii) スロット中の意味的制約条件を、ＤＣＤ節のボディに直接記述する。したがって意味的制約条件の検査は Prolog プログラムを直接実行することにおきかえられる。

図１では意味的制約条件は、スロットに対応するリストの第二要素に記述されていた。そのためリストの第二要素として書かれた意味的制約条件をコンパイルすることができない。これに対して、

(iv)ＤＣＤによる辞書記述は、意味的制約条件も含めて完全にコンパイルすることができる。
(v) フィラーが、ある概念を含意するかどうかを調べるため、上位概念をたどるための述語もまた述語 sem によって記述されている。述語 sem は辞書項目をＤＣＤ節で定義するための述語そのものであるから、上位概念をたどるための特別な述語を用意する必要がないだけでなく、たとえば「血液型 a もしくは b の人」などという比較的複雑な意味的制約条件も次に示すように容易に記述できる。

```
sem(N,human),
(sem(N,blood_type:a);
 sem(N,blood_type:b))
```

なお、図２の

```
sem (X,X).
```

は、意味的制約条件の検査に対する停止節としてはたらく。

(vi)知識の遺伝は、図２の最後にあるＤＣＤ節でおこなう。それにより、それより前のスロット(subj,obj,with) を、フィラーが満たしえなかった時に、Prolog のバックトラックおよびユニフィケーション機構をそのまま利用して、上位のスロットを調べることができる。さらに多重知識遺伝（継承）を実現したければ、

```
sem(open,Prop) :-
        sem (event,Prop).
```

を付加すればよい。

上位概念からの知識の遺伝に関して、ここで次の注意をしておく。一般に

```
sem(X,Prop):-sem(Y,Prop).
```

と書いた時、Y がX の上位概念だから、概念の包含関係に注目すると

```
sem(Y,Prop):-sem(X,Prop).
```

と書いた方が自然のように思われる(Xが human で Yが animal の場合を考えよ）。しかし、

```
sem(X,Prop):-sem(Y,Prop).
```

は、「上位概念Y が性質 Prop を持てば、下位概念X もその性質を持つ」ということ、すなわち知識の遺伝そのものを記述している。

さらにＤＣＤ形式の辞書の検索には、Prolog に組み込みの機構をそのまま用いてることができる。たとえば

```
sem(craw,color:black).
sem(craw,Prop) :-
    sem(bird,Prop).
sem(bird,numberOfLegs:2).
sem(bird,canFly:yes).
```

と定義してあれば、

```
?-sem(craw,X).
```

を実行することにより、craw　の持つ性質を辞書から、

```
X=craw;
X=color:black;
X=bird;
X=numberOfLegs:2;
X=canFly:yes
```

のように次々に得ることができる。すべての性質を一度に得たければ

```
?-setof(X,sem(craw,X),S).
```

を実行すれば、S に得られる。また、

```
?-sem(craw,canFly:X).
```

を実行すれば、

```
X=yes
```

が得られる。以上のことから

(vii) Prolog　の基本計算機構を利用して、ＤＣＤ節で記述した辞書項目を検討し応答することができる。

(vi)と(vii) は、概念のリンクを下位から上位に向けてたどり、下位から上位の性質を、（ユニフィケーションを利用して）アクセスするものであった。それに対して

```
?-sem(X,bird).
```

を実行することにより、bird　からはじまり bird の下位にある概念を次々に得ることができる。従って、

(viii)概念のリンクを上位から下位に向けてたどることができる。
それにより上位から下位に位置する概念を知ることができる。

(vii) と(viii)の結果は、次のことを意味している。

(ix)意味処理結果がＤＣＤ形式であれば、それに対する質問応答は Prolog に組み込みの機構をそのまま利用して行なうことができる。

(vii) と(viii)については、文献[8] を参照。

５．実験結果

図３と４にＤＣＤを用いて英語と日本語の意味処理を行なった結果を示す。図３(a) の述語 sem1 は、主語を意味処理するためのものである。図３に示すように述語 sem1 は、形式を整えて単にＤＣＤ節を呼ぶものである。{} で囲まれた部分にある他の述語は構文処理を行なうときに用いるものである。この例の場合には意味処理用プログラムをほとんど記述する必要がないことが分かる。図３(b) は、"I open the door with a key" を意味処理した結果である。% 記号の右は、意味処理結果がＤＣＤ節になっていることを示している。% 記号の左は、意味処理結果を見やすくするために、フレーム的な形式で出力している。

図４は、「過酸化水素水に二酸化マンガンを作用させると、酸素が発生し、石灰石に塩酸を作用させると、二酸化炭素が発生する」という文を意味処理した結果を示す。

使用した Prolog は、ＳＵＮ２ワークステーション上の C-Prolog インタープリタで、実行速度は約１ＫＬＩＰＳである。

```
/*semantic interpretation programs */
seml([Uname,Sem | OldSem],
      Sname:[Fname|Rest],NewSem) :-
                  sem(Uname,Sname:Fname
                          ~[Uname,Sem | [[Fname|Rest] | OldSem]]
                          ~NewSem).
-------------------------------------------------

        ..........

sdec([not_adv | VP_A],[SDEC_S | ADV1_S]) -->
   subj(SUBJ_A,SUBJ_S),
  ([],
    {ADV1_S=[] };
   adv(ADV1_A,ADV1_S) ),
   vp(VP_A,VP_S).
      { subj_v(SUBJ_A,VP_A),
        seml(VP_S,subj:SUBJ_S,SDEC_S) }.

sentence(SDEC_A,SDEC_S) -->
   sdec(SDEC_A,SDEC_S).

        ..........

-------------------------------------------------
```

図３　(a) 意味処理プログラムと文法例

```
Input sentences
|: |: i open the door with a key.

i open the door with a key.

Processing Time = 2.26667 sec.
No. 1
 |-sentence
    |-sdec
       |-subj
       |  |-np
       |     |-pron -- i
       |-vp
          |-vp
          |  |-v --open
          |  |-obj
          |     |-np
          |        |-ddet
          |        |  |-det -- the
          |        |-nomhd
          |        |  |-n -- door
          |-pp
             |-p -- with
             |-obj
                |-np
                   |-a -- a
                   |-nomhd
                   |  |-n -- key

open#5::
  prototype:open          % sem(open#5,Prop) :- sem(open,Prop).
  agent:i#4               % sem(open#5,agent:i#4).
  instrument:key#7        % sem(open#5,instrument:key#7).
  object:door#6           % sem(open#5,object:door#6).
```

```
i#4::
  prototype:i              % sem(i#4,Prop) :- sem(i,Prop).

door#6::
  prototype:door           % sem(door#6,Prop) :- sem(door,Prop).
  det:the                  % sem(door#6,det:the).

key#7::
  prototype:key            % sem(key#7,Prop) :- sem(key,Prop).
  det:a                    % sem(key#7,det:a).
```

図３　(b) 英語の意味処理結果（１）

```
Processing Time - 1.01669 sec.
No. 2
 |-sentence
    |-sdec
       |-subj
       |  |-np
       |     |-pron -- i
       |-vp
          |-v -- open
          |-obj
             |-np
                |-ddet
                   |  |-det -- the
                   |-nomhd
                   |  |-n -- door
                   |-ncomp
                      |-pp
                         |-p -- with
                         |-obj
                            |-np
                               |-a -- a
                               |-nomhd
                                  |-n --key
```

```
open#5::
  prototype:open          % sem(open#5,Prop) :- sem(open,Prop).
  agent:i#4               % sem(open#5,agent:i#4).
  object:door#6           % sem(open#5,object:door#6).

i#4::
  prototype:i             % sem(i#4,Prop) :- sem(i,Prop).

door#6::
  prototype:door          % sem(door#6,Prop) :- sem(door,Prop).
  det:the                 % sem(door#6,det:the).
  with:key#7              % sem(door#6,with:key#7).

key#7::
  prototype:key           % sem(key#7,Prop) :- sem(key,Prop).
  det:a                   % sem(key#7,det:a).
```

図３　(b) 英語の意味処理結果（２）

```
 kasankasuisosuini nisankamanganwo sayousaseruto, sansoga hasseisi,sekkaisekini
ensanwo sayousaseruto, nisankatansoga hasseisuru.

 Processing Time  = 11.4167 sec.
 No. 1
  |-bun
     |-kouchishiku
     |  |-meishiku
     |  |  |-meishi --H202
     |  |-joshi
     |     |-kakujoshi -- ni
     |-bun
        |-kouchishiku
        |  |-meishiku
        |  |  |-meishi -- MnO2
        |  |-joshi
        |     |-kakujoshi -- wo
        |-bun
           |-renyouku
           |  |-jutsugo
           |  |  |-doushi
           |  |  |  |-meishi -- sayou
           |  |  |  |-doushi
           |  |  |     |-gokan -- s
 ～～～～～～～～～～～～～～～～中途省略～～～～～～～～～～～～～～～～
 ～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～
                    |-meishi -- hassei
                                 |-doushi
                                    |-gokan -- s
                                    |-gobi -- uru

sayou@shieki#37::
 prototype:sayou@shieki % sem(sayou@shieki#37,Prop) :- sem(sayou@shieki,Prop).
 proposition:sayou#36   % sem(sayou@shieki#37,proposition:sayou#36).
```

```
sayou#36::
  prototype:sayou              % sem(sayou#36,Prop) :- sem(sayou,Prop).
  goal:H2O2#0                  % sem(sayou#36,goal:H2O2#0).
  agent:MnO2#0                 % sem(sayou#36,agent:MnO2#0).

hassei#40::
  prototype:hassei             % sem(hassei#40,Prop) :- sem(hassei,Prop).
  object:O2#0                  % sem(hassei#40,object:O2#0).

hassei#49::
  prototype:hassei             % sem(hassei#49,Prop) :- sem(hassei,Prop).
  object:CO2#0                 % sem(hassei#49,object:CO2#0).

sayou@shieki#46::
  prototype:sayou@shieki       % sem(sayou@shieki#46,Prop) :- sem(sayou@shieki,Prop).
  proposition:sayou#45         % sem(sayou@shieki#46,proposition:sayou#45).

sayou#45::
  prototype:sayou              % sem(sayou#45,Prop) :- sem(sayou,Prop).
  goal:CaCO3#0                 % sem(sayou#45,goal:CaCO3#0).
  agent:HCl#0                  % sem(sayou#45,agent:HCl#0).

causalChain#50::
  antecedent:sayou@shieki#46   % sem(causalChain#50,antecedent:sayou@shieki#46).
  consequent:hassei#49         % sem(causalChain#50,consequent:hassei#49).

causalChain#51::
  antecedent:sayou@shieki#37   % sem(causalChain#51,antecedent:sayou@shieki#37).
  consequent:hassei#40         % sem(causalChain#51,consequent:hassei#40).
```

図４　日本語の意味処理結果

６．　おわりに

ホーン節の形式で辞書項目の記述を行なうことにより、意味処理の中核をなす手続きをすべて Prolog の基本計算機構で代用可能なことを示した。我々はこれをＤＣＤによる辞書項目記述とよんでいる。４．の(vii) と(viii) で説明したようにＤＣＤを用いれば、辞書項目を検索してその意味的性質を取り出す操作も、Prolog の基本計算機構に委ねることができる。図２の open の記述における addProp は深層格を抽出していたが、ＤＣＤの形式の意味構造を抽出することが望ましい。それにより抽出した意味構造に対する質問応答システムの基本部分を、Prolog の基本計算機構で代用可能になるからである。小規模な実験ながら、５章ではそれが可能なことをすでに確認している。

また図２の記述から明らかなように、ＤＣＤによる辞書項目記述中のカット記号(!) をコミットオペレータとみれば、concurrent Prolog の記述形式に酷似している。これは、Concurrent Prolog によって並列に意味処理を行なう可能性を示唆している[5] 。今後、この方向の研究を進める必要がある。

ＤＣＤの記述形式にシンタックス．シュガーを加えて、辞書項目の記述者が書きやすい形式を開発する必要がある。現在我々は、図６に示す形式を採用している。これは Prolog で書かれたトランスレータにより、図３のＤＣＤ節に変換される。辞書項目記述用の高水準言語としてどのようなものがよいかは、使用経験を含めて検討する必要がある。また、辞書項目の更新、修正、付加に伴い、辞書の保守管理を行なうプログラムを開発する必要がある。これは、知識の獲得と管理の問題でもある。

```
open::
    subj  $  agent       <=  human,
             object      <=  thingOpen!
                             view,
             instrument  <=  Instrument,
             reason      <=  wind          :
    obj   $  object      <=  thingOpen
                             view          :
    with  $  instrument  <=  instrument    :
    is_a  $  act                           .
```

図５　辞書項目記述用高水準言語による記述例（この形式で書かれたものが、トランスレータにより、図２のＤＣＤ節に変換される。）

ＤＣＤ形式の辞書記述を見い出したことにより、意味処理のためのプログラミング労力が軽減され、意味処理について本質的に困難な問題の解決に直接取り組むことが可能な段階にきたと考えられる。今後、ＤＣＤをベースにして意味理解を中核に据えた自然言語理解システムの作成を行ないたいと思っている。

謝辞

本稿をまとめるにあたり、新世代コンピュータ技術開発機構の渕一博所長から、貴重な助言をいただいた。ここに深く感謝する。

参考文献

[1] Tanaka,H. : Semantic Representation Langrage, in Ktagawa,T. (ed.):Japan Anual Reviews in Electronics, Computers and Telecomunications,Computer Science and Technologies, Ohm/North-Holland(1982),71-86.

[2] 田中穂積　松本裕治 : 自然言語処理におけるProlog, 情報処理, 25,12 (1984) 1396-1403.

[3] Nilsson,N.J : Principles of Artificial Intelligence, Palo Alto, Calif, Tioga(1980).

[4] Hayes,P.J. : The Logic of Frame, in Metzing,D. ed: Frame Conceptions and Text Understanding, Walter de Gruyter, Berlin, New York,(1980),46-61.

[5] Shapiro,E. and Takeuchi,A.: Object Oriented Programing In Concurrent Prolog, New Generation Computing, 1,(1983), 25-48

[6] Clocksin,W.F. and Mellish,C.S.: Programming in Prolog,springer-Verlag,(1981).

[7] 渕　一博 : 述語論理的プログラミング－EPILOGの提案, 情報処理学会記号処理研究会 1-2,(1977).

[8] 田中穂積　小山晴夫 : Definite Clause Knowledge Representative - - - Prolog による structured object の表現形式と推論, Proc.of ロジックプログラミングコンファレンス'８５, (1985).