知識工学と人工知能 57 － 7
（１９８８．３．１０）

自然言語の基礎的な意味処理と知識コンパイル

赤間 清

（北海道大学 文学部 行動科学科）

本論文では，「名詞句＋の＋名詞句」や「名詞句＋は＋名詞句＋です」の意味処理や，ある対象が与えられたクラスの対象を表しうる場合に，その新しい対象を計算する述語の処理を取上げる．そして，それらの処理は２つのクラスから意味処理手続を起動するという類似の構造を含むこと，それらの処理には大量の知識が関係していてコストがかかることを示す．したがって充分な意味処理を目指すには，それらの処理を高速化する方法を考えることが有効である．我々は知識コンパイルによってこの問題を解決する．知識コンパイルの結果は，Prolog-PAL のＦ型の集合束縛変数を用いて表現され，充分小さな領域に記憶され，しかも非常に効率的に高速に実行される．

Semantic processing of natural language and knowledge compilation

Kiyoshi AKAMA

(Faculty of Letters, Hokkaido University, Sappro-shi, 060, Japan)

In this paper we discuss the semantic processing of the noun phase of the form : NP + "no" (possisive perticle) + NP and NP + "ha" (substantive perticle) + NP + "desu" (is), and the semantic processing of the "point" predicate which computes the semantic representation of a new object from an input object and a class. Depending on the classes of the input objects, many different procedures may be used to generate a semantic representation.

These three kinds of semantic processing involve similar subroutines for sellecting the procedure. The selection subroutines have two class names as inputs which correspond to the two NPs, or the object and the class (for the "point" predicate). The sellection of the procedure to use requires a search for the path connecting the classes through their ancestor classes.

We give a method to reduce the time of this search by preparing a set of all procedures exiting and a set of all procedures entering a given class. Then when a procedure joining two classes is sought, the intersection of the two sets gives the procedure name to produce the semantic representation. As these sets are represented by finite set bound variables in our extended Prolg : PAL, the execution is much more efficient because the intersection is computed by the unification of finite set bound variables and is implemented using a bit-wise and opperation.

1．　まえがき

継承階層 Prolog ：PALは，概念などのなす階層構造を陽に扱うことのできる拡張 Prolog である[1]．それが目指すところは，①　現在の Prolog が持つ，汎用の言語処理系としての能力（表現力と推論速度）を失うことなしにPrologを拡張し，②　知識処理などのプログラムを，宣言的で明快かつ容易に記述でき，③　より高速に実行することができ，④　処理する知識量の増加に強い言語処理系である．それらの特徴は，継承階層の宣言によるクラスとインスタンスの定義[1]，クラス束縛変数に関するユニフィケーション[1]，継承階層コンパイラ[2]，継承階層クローズ・インデキシング[2]，集合束縛変数[4]，マルチクラス[4]などで実現されてきた．

現在，継承階層 Prolog：PAL はC言語で書かれたインタープリタ版が利用されている．われわれはPALを用いて，理解から生成までの基礎的な部分をひととおり扱うことのできる自然言語処理実験システムTALK[3]を試作した．またそれをデータ・ベースの問い合せや図形描画システムのための自然言語インターフェイスに適用して PALの評価を試みた．PAL は自然言語処理システムの作成に，とくに意味処理の部分で，非常に貢献することが確認された．

たとえば，集合束縛変数は自然言語処理と関係が深い．集合束縛変数による意味表現は，「住所が東京または大阪，年齢が３０代，仕事はサラリーマンで，その上司が万年という名前の課長である人間」のような複雑な対象を明快に表現できる．また，そのような複数の対象のセマンティク・マッチングのプログラムもユーザーが容易に書くことができる．TALK の意味処理はそのような意味表現を動的に変化させて，文の意味を作り上げることができる．また，集合束縛変数を基礎としたマルチクラスは，自然言語処理における多義語の扱いなどにも貢献できる．それは backtrack を極力さけて，意味処理を自然に効率よく達成する．

言うまでもなく，自然言語の意味処理では，大量の知識を背景として，いろいろな処理を進めて行く必要がある．自然言語処理の全体を概観して見るとき，構文解析に比して意味解析が圧倒的に多くの情報処理を必要としていることは明らかである．ところが，構文解析についてはこれまでより効率的なパーザーを求めて非常に多くの努力が投入されてきたが，意味解析についてはまったくその努力が足りなかった．これはそもそも意味処理の研究があまり進んでいないことや，高速化を議論するための土俵ができていないことに関連している．しかし，意味処理が十分になされるシステムが作られるにつれて，意味解析における高速化の努力の効果は，構文解析レベルの努力の効果を遥かに上回るようになることは確実である．

したがって我々は，　①　自然言語の意味処理において重要な情報処理とはなにか，を考え，　②　そのうちで最も時間的コストのかかる種類の情報処理とはなにか，を見出し，　③　それを高速化する手法あるいは枠組みとはなにか，を明らかにして行く必要がある．

本論文の目的は，継承階層 Prolog ： PAL 上に作られた自然言語処理の実験システムである TALKにおける，そのような方向への努力の１つについて述べることである．ここでは，TALKの行なう意味処理の全体を議論するのではなく，そのなかの基礎的な要素となる一群の情報処理を取上げる．それらはいくつかの共通性を持ち，知識コンパイルの方法にも共通性がある．またその知識コンパイルの方法は，PAL のＦ型の集合束縛変数を利用するとき，非常に効果が高くなる．

2．　自然言語の基礎的な意味処理

2．1　「名詞句＋の＋名詞句」の処理

名詞句と名詞句が「の」で結び付けられて，新たな名詞句を構成する場合の意味処理を考える．たとえば，「医者の太郎」の場合を考える．TALKでは「医者」は jobであり，太郎は man_name である．それらはそれぞれ，

```
医者 ----- (*j^doctor)
太郎 ----- (taro)
```

という意味表現を持つ．この場合，doctor はクラス名，taro はインスタンス名として宣言されている．このとき，「医者の太郎」の意味表現として，

```
(*h^man (===>job *j^doctor)
        (===>name taro))
```

が得られる．これは直訳すれば，

「医者という職業の，名前が太郎という男」

となる．また，「犬の目」の場合には，

```
犬 ----- (*a^dog)
目 ----- (*b^eye)
```

に対して，

```
犬の目 ----- (*b^eye (<==part *a^dog))
```

が得られる．

このような処理はTALKでは，m_no_m という述語によって行われる．これは何回も繰り返され，より大きな意味表現を作ることも多い．たとえば，「太郎の母の花子」の場合は，「の」が２つあるので，m_no_m の処理は２回起こる．

太郎 ----- (taro)
母 ----- (*h1^woman
(<==mother *h2^human))

から，まず「太郎の母」にあたる意味表現：

(*h1^woman
(<==mother *h2^human
(==>name taro)))

が得られ（実は母が関数名詞なので，本論文で主に論ずる他の場合とは少し異なる処理が行われる），次にそれと，

花子 ----- (hanako)

から，

(*h1^woman
(<==mother *h2^human
(==>name taro))
(==>name hanako))

が得られる．

このような「の」による結合では，新しく意味を作る処理は，もとの２つの名詞句の意味の属するクラスのペアによって決ることが多い．ここで名詞句の意味の属するクラスとは，たとえば，「太郎の母」における woman や「花子」における woman_name である．

```
(as (xnoy job job=man=name man_name))
(as (xnoy animal animal=body animal_body))
(as (xnoy human woman=name woman_name))

(as (make_m_no_m job=man=name
    *xs *ys (*h (==>job . *xs) (==>name . *ys)))
   (cb *h man))
(as (make_m_no_m animal=body
    *xs (*y . *yr) (*y (<==part . *xs) . *yr)))
(as (make_m_no_m woman=name
    (*x . *xr) *ys (*x (==>name . *ys) . *xr))
   (cb *x woman))
```

Fig.1 Examples of knowledge for computing the meaning of the noun phrase of the form : NP + "no" + NP

図１ 「名詞句＋の＋名詞句」の処理のための知識例

TALKでは名詞句の意味はＳ式で表現されており[3,4]，その第１要素に対象が何であるかを示すクラスあるいはインスタンスが書かれたシンタクスを持つ．名詞句の意味の属するクラスは，述語bcを用いて，

```
(bc (*y^animal) *c)    ---> *c = animal
(bc *y^cat *c)         ---> *c = cat
(bc (poti) *c)         ---> *c = dog
(bc poti *c)           ---> *c = dog
```

というように得られる．

与えられるクラスのペアがどんな処理を必要とするかの知識を，たとえば，図１のように表現したとする．ここで，述語 xnoy は，クラスのペアが第１引数と第３引数に与えられたとき，そのペアが要求する処理名を第２引数に与えるために使う．たとえば job と man_nameのペアは job=man=name というアトムで指定される処理を起動する．述語 make_m_no_m は処理の本体部分で，処理名（第１引数）と２つの意味表現（第２引数と第３引数）が与えられると，「の」で結ばれてできる新しい名詞句の意味表現（第４引数）を作る．

```
(as (m_no_m *xs *ys *zs)
    (bc *xs *xcc)
    (bc *ys *ycc)
    (xnoy *xc *rel *yc)
    (subclass_of *xcc *xc)
    (subclass_of *ycc *yc)
    (make_m_no_m *rel *xs *ys *zs))
```

Fig.2 a simple program for the noun phrase : NP + "no"+ NP

図２ 「名詞句＋の＋名詞句」の処理をする簡単なプログラム

これを利用して「の」の処理をやる簡単な方法は，たとえば図２のようなプログラムでかける．ただし subclass_of はＰＡＬの組込述語で，２つのクラスの間の子孫－祖先関係を扱う．たとえば，

(subclass_of woman animal)

である．

２．２ 「名詞句＋は＋名詞句＋です」の処理

「名詞句＋は＋名詞句＋です」という形の文の意味を求める処理について述べる．たとえば，「太郎は医者です」を例にとると，現在のTALKは，

```
太郎 ----- (taro)
医者 ----- (*j^doctor)
```

をもとにして，文の意味表現：

```
(*s^sentence
    (verb be)
    (mod now)
    (actor *h1^man (==>name taro))
    (comp  *h2^human (==>job *j^doctor)))
```

を得る．また「父は課長だった」ならば，

```
(*s^sentence
    (verb be)
    (mod past)
    (actor *h1^man (<==father *h2^human))
    (comp  *h3^human
            (==>rank *j^head
                    (==>org section))))
```

となる．

これらの処理に使われる知識は図3のようになる．述語 xisy は，クラスのペア（第1引数と第3引数）が与えられたとき，そのペアが要求する処理名（第2引数）を割出すために使う．たとえば man_name と job のペアは name=man=job というアトムで指定される処理を起動する．

```
(xisy man_name name=man=job job)
(xisy human human=rank rank)

(as (make_x_is_y name=man=job
    *xs *ys (*h1^man (==>name . *xs)
            (*h2^human (==>job . *ys))))
(as (make_x_is_y human=rank
    *xs *ys *xs (*h^human (==>rank . *ys))))
```

Fig.3 Examples of knowledge for computing the meaning of the sentence of the form : NP + "ha" + NP + "desu".

図3 「名詞句＋は＋名詞句＋です」を処理する知識例

述語 make_x_is_y は処理の本体部分で，処理名（第1引数）と「a は b だ」の a と b にあたる2つの意味表現（第2引数と第3引数）が与えられると，a と b から得られる意味表現 A と B（第4引数と第5引数）を返すプログラムである．たとえば，make_x_is_y の最初の節について説明しよう．x_is_yの働きによってname=man=jobという名の処理が起動されるとき，make_x_is_y の第2引数には男性名を表わす意味表現が，第3引数には職業を表わす意味表現が与えられる．そのとき「a（男性名） は b （職業） だ」という文は，a という名前の男は b という職業の人間だというように解釈される．

2．3 point 述語

ある対象（概念）があるクラスに属する対象（概念）を表わし得るか，表わし得るとすればそれはどんな対象になるかを求める情報処理も基本的である．たとえば，「太郎は医者を投げ飛ばした」において，「投げ飛ばす」に関する情報は，

```
投げ飛ばす --- throw
  actor   human  (ga z)
  object  object (wo z)
```

などとなっている．TALKは，

```
太郎 ----- (taro)
```

が human になれるか否か，なれるとすればそれはどんなS式で書けるかを求めなければならない．そのための知識は図4のrepresent述語でかかれている．

```
(represent man ==>name man_name)
```

はクラス man_name に属するもので man を指し示すことができ，それらは ==>name の関係にあることを意味している．それを利用して，対象を計算するのは point 述語である．point 述語は一般に，第1引数に対象を，第2引数にクラスを入力として受取り，第3引数に新しい対象を返す．その素朴な定義が図5にある．たとえば，

```
(point (taro) human *result)
```

と呼出せば，その結果は，

```
(point (taro) human (*^human (==>name taro)))
```

となる．

```
(represent object ==>shape figure)
(represent man    ==>name man_name)
(represent woman  ==>name woman_name)
(represent human  ==>job job)
```

Fig.4 Example of knowledge showing that a concept may mean some other concept

図4 ある概念が他の概念を表わしうることに関する知識の例

```
(as (point *xs *zc (*y (*rel . *xs)))
    (bc *xs *cc)
    (represent *yc *rel *xc)
    (subclass_of *cc *xc)
    (cb *y *yc)
    (cb *y *zc))
```

Fig.5 naive "point" predicate
図5 素朴な point 述語

図5の point述語の中で，cb述語は，bc述語とは逆に，与えられた対象を与えられたクラスまで限定する働きをする．たとえば，

```
(cb (*y^animal) dog)   ---> *y^dog で成功
(cb *y^cat animal)     ---> *y^cat で成功
(cb *y^dog cat)        ---> 失敗
(cb poti dog)          ---> 成功
(cb poti cat)          ---> 失敗
(cb (poti) dog)        ---> 成功
```

となる．

point述語には，前節までで述べた２つの場合と違う点がある．以下ではそれを，

① 「太郎は医者を投げ飛ばした」
② 「医者が赤ちゃんを生んだ」

の２つの文の「医者」の解釈に焦点をあてて説明する．動詞「投げ飛ばす -- throw」の object スロットはobject（物体），「生む -- bear」の actor スロットは woman と書いてあるとする．このとき，

① 「医者」は物体でありうるか？
② 「医者」は女性でありうるか？

という２つの質問に対して，point 述語は yes と答え，結果として，

① 医者という職業の人間
② 医者という職業の女性

を計算する．このときの概念図が図６である．医者という単語から human が求まるが，目標となるクラスは，それぞれ，object と woman である．このように point 述語の場合は，目標となるクラスがもとの対象が表わすクラス(human)の祖先でも子孫でもよい．これに対して前の２つの例では，目標となるクラスはもとの対象から xnoy や xisy の知識で移った先のクラスの子孫クラス(subclass_of)に限られていた．

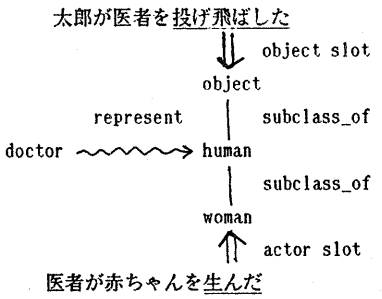

図６ 医者の文のための意味処理

point 述語のこれらの結果を用いれば「太郎は医者を投げ飛ばした」の意味は，

```
(*1^sentence
   (verb throw)
   (mod 24 kako si)
   (theme *2^man)
   (actor *2^man (==>name taro))
   (object *3^human (==>job *4^doctor)))
```

となり，「医者が赤ちゃんを生んだ」の意味は，

```
(*1^sentence
   (verb bear)
   (mod 24 kako n)
   (actor *2^woman (==>job *3^doctor))
   (object *4^human (==>age *5^[0 2])))
```

となる．

３． 知識のコンパイル

３．１ 知識コンパイルの必要性

「名詞句＋の＋名詞句」に関する意味処理では，新しく意味を作る処理は，もとの２つの名詞句の意味の属するクラスのペアによって決ることが多いが，そのようなペアの数は，よく使うものに限っても，非常に多い．それは第１に，結果として必要な処理の数（xnoyのファクトの数に対応する）が多いこと，第２にxnoyのファクトが１つあるごとに，それで関係し得るクラスのペアはたくさんありうるからである．

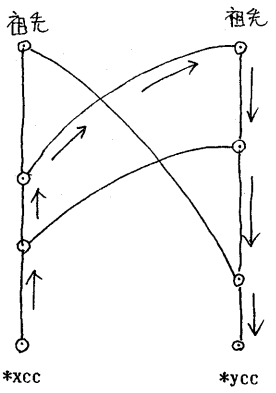

図7　「の」の処理のためのパス探索

図2の m_no_m プログラムが働く場合，核となる処理は，2つのクラス*xccと*yccが与えられたとき，図7で示されるようなパスを求める探索である．これは2つのクラスの祖先の数の積に比例した時間がかかる．継承階層を深くしたり，可能な処理の数（xnoyの数）を十分たくさん許せば処理時間は爆発してしまう．知識をコンパイルしてこの問題を解決しなければ，「名詞句＋の＋名詞句」の意味処理は十分にはできない．「名詞句＋は＋名詞句＋だ」の場合も事情はまったく同様である．point の場合は，与えられるクラスの祖先だけでなく，子孫も探索しなければならないので，負担はもっと重くなるので，知識コンパイルの必要性はさらに大きなものとなる．

3．2　知識コンパイルの方法

2章の意味処理を高速化するための知識コンパイルの方法を示そう．たとえば，つぎのような形の知識がたくさんあるとする．

```
(xnoy class_x procedure class_y)
```

これは，class_x と class_y のペアに対して procedure という名前の処理が働くという意味である．ただし xnoy の各ファクトに出てくる procedure 名はすべて異なるアトムであると仮定する．そのような知識の集合（祖先子孫関係などで展開して得られたものすべて）をSとする．そのとき，Sから2種類の述語を定義する．それらは，述語 xnoy_xと述語 xnoy_y で，

```
(as (xnoy_x class_x . proc_list_x))
(as (xnoy_y class_y . proc_list_y))
```

の形を持つ．proc_list_x は，Sの中の

```
(xnoy class_x procedure ?)
```

という形のすべての知識に対する procedure のリストである．同様にproc_list_y は，Sの中の

```
(xnoy ? procedure class_y)
```

という形のすべての知識に対する procedure のリストである．class_x と class_y が与えられたとき求めるものは，

```
(and (xnoy class_x *procedure ?)
     (xnoy ? *procedure class_y))
```

を満たす *procedure である．それは，それぞれ，

```
(xnoy_x class_x . *proc_x)
(xnoy_y class_y . *proc_y)
```

で求められた *proc_x と *proc_y の共通集合*procsの元である．これらを利用すると m_no_m のプログラムは図8のように書ける．すなわち，xnoy に関して本論文で行なう知識コンパイルとは，述語 xnoy の情報を xnoy_x と xnoy_y の情報に書換えることである．

```
(as (m_no_m *xs *ys *zs)
    (bc *xs *xc)
    (bc *ys *yc)
    (xnoy_x *xc . *lx)
    (xnoy_y *yc . *ly)
    (member *rel *lx)
    (member *rel *ly)
    (make_m_no_m *rel *xs *ys *zs))
```

Fig.8　A program to process compiled knowledge
図8　コンパイルされた知識を利用する方法

知識コンパイルのための基本述語について説明する．図9の述語 compile_u は，述語名*pとクラス名*aと手続き名*nが与えられたとき，*aのすべての祖先（*aを含む）*bに対して，*bと*pなる関係を持つものの1つとして*nを追加登録する述語である．述語compile_d は*aのすべての子孫（*aを含む）*bに対して，同様のことを行なう．compile_d を定義するために使われる組込述語 parent と children は，次の形を持ち，

```
(parent class parent_class)
(children class child_classes)
```

任意のクラスに対してその親クラスや子（クラスまたはインスタンス）リストを第2引数に求める．compile_uとcom

pile_dは，副作用としていくつかの述語を assert したあと失敗する述語である．

```
(as (compile_u *p *a *n)
    (if (*p *a . *ras)
        (and (retract (*p *a . *ras))
             (as (*p *a *n . *ras)))
        (as (*p *a *n)))
    (parent *a *b)
    (compile_u *p *b *n))

(as (compile_d *p *a *n)
    (if (*p *a . *ras)
        (and (retract (*p *a . *ras))
             (as (*p *a *n . *ras)))
        (as (*p *a *n)))
    (children *a *bb)
    (member *b *bb)
    (classp *b)
    (compile_d *p *b *n))
```

Fig.9 The core of the knowledge compiler
図9 コンパイラの核部分

```
(set-kl xnoy_x xnoy_y)
(retract xnoy_x)
(retract xnoy_y)
(as (xnoy *x *r *y)
    (true (compile_d xnoy_x *x *r))
    (true (compile_d xnoy_y *y *r)))
(xnoy job job=man=name man_name)
(xnoy animal animal=body animal_body).
(xnoy human woman=name woman_name)
          ...
```

Fig.10 A program to compile the given knowledge of [NP + "no" + NP]
図10 「名詞句＋の＋名詞句」の知識をコンパイルをするプログラム

知識コンパイルを行うには，compile_uやcompile_dを各知識から呼出せばよい．たとえば，図10を実行すれば，後ろに並べられた xnoy に関する知識は compile_d を用いてコンパイルされて，xnoy_x と xnoy_y で書かれた知識が得られる．set-kl は引数で指定される述語名が「知識」であることを宣言する．そのように宣言された述語は，節が存在しないときに呼出されても，エラーを引起こさずに単に失敗する．

xisy の知識コンパイルは，xnoy の場合とほぼ同様である．represent に関する知識コンパイルは，compile_d による子孫の処理だけでなく，compile_u を用いた祖先に対する処理が追加される．

4． 集合束縛変数による高速化

4．1 F型の集合束縛変数を用いた知識コンパイル

コンパイルされた知識を利用する部分は，2つの集合の共通部分をとる演算を含む．上記のプログラムではそれは述語 member を2回用いて行われていた．共通部分を求める演算を高速に行うことができれば，推論はさらに効率的になる．

```
(as (m_no_m *xs *ys *zs)
    (bc *xs *xc)
    (bc *ys *yc)
    (xnoy_x *xc *l)
    (xnoy_y *yc *l)
    (decode *l *list)
    (member *n *list)
    (xnoy_r *n *xs *ys *zs))
```

Fig.11 A program of predicate m_no_m which is using set bound variables of finite set type
図11 F型集合束縛変数を利用したプログラム

そのために，集合束縛変数のF型[4]を使うことができる．F型の集合束縛変数は，たとえば，

```
*var^{ dog cat pig }
```

のようなもので，この場合変数 *var は，集合 { dog cat pig } に束縛されており，その束縛は，

```
(member *var (dog cat pig))
```

と宣言的には同じ意味を持っている．しかし，手続き的には異なり，共通部分を求める処理は非常に高速である．それはF型の集合の内部表現が bit列 になっており，共通部分を求める演算は bit列の論理積(and) を求める演算でできるからである．手続き名の集合をあらかじめF型の集合束縛変数にコード化しておけば，共通部分を求める演算は継承階層 Prolog に組込のユニフィケーションで高速に行われる．あとは decode して用いればよい．

集合束縛変数にコンパイルされた知識を使うm_no_m のプログラムの例を図１１に示す．ここでのコンパイルされた知識は，図１０のプログラムで得られる知識より少しだけ抽象的になっている．すなわち，以前は手続き名がそのまま用いられていたが，新しい知識コンパイルでは，手続き名が数字（第何番目の bitかを示す）に置き換えられ，手続き名はコンパイル結果には現れないことになる．述語 make_m_no_m もそれに合わせて，符号化された数字を扱う述語 xnoy_r に書換えられ，ユニフィケーションではなく（ベクトルのように）数字でアクセスされる．この符号化も処理の高速化に貢献する．

４．２　F型集合束縛変数と並列処理

集合束縛変数のユニフィケーション[4]の速度は，変数を束縛する集合の積の演算の速度できまる．積の演算を高速に行なう専用の外部装置と結合すれば，集合束縛変数は Prolog-PAL の高速化をさらに強力に推し進めることになる．

たとえば上記のF型の集合束縛変数について考えると，集合の積の演算は，bit 列の and 演算であるから，bit の幅をいくら大きくしても，並列の and 演算を容易に実現できる．そのような大規模，単純，かつ高速な外部装置を前提として，本論文で述べたようなF型集合束縛変数への知識コンパイルを採用すれば，逐次処理言語である Prolog-PAL を大規模並列処理と比較的簡単に結合して大きな効果をあげることができる可能性がある．

集合束縛変数の考え方は，われわれがすでに導入したF型，I型，C型，M型だけでなく，いろいろな集合データに応用できる．現在の Prolog-PAL は，F型，I型，C型，M型が逐次処理の範囲で高速に計算されることを充分に利用して構成されているが，専用の外部装置との関わりで考える時には，もっといろいろな集合を，並列に，そして非常に高速に扱いうる可能性がある．このように集合束縛変数は，Prolog の構造を外部専用装置と柔軟に結びつけ，対象データの特性を処理の高速化に直接的に反映させることを可能にする．この方向は，自然言語の意味処理における知識の量の問題などを解決するための有望な道の１つであると考えられる．

５．　むすび

自然言語の意味処理の研究を健全に進めるためには，知識量の増加が引起こす困難を吸収するための枠組みを作る必要がある．本論文では，２つのkey から決る処理手続きを引出すことがいくつかの意味処理にとって重要であることを示し，それを高速に処理するために知識コンパイルする方法を与えた．この方法は，２つの key だけでなくｎ個の key にも自然に拡張できる．また，F型の集合束縛変数を利用しているので，時間効率と空間効率が，ともによい方法である．本方法は TALKのなかにすでに実現され，意味処理を高速化するのに役立っている．

文　献

[1] 赤間清：ＰＡＬ：継承階層を扱う拡張PROLOG，情報処理学会論文誌　Vol.28 No.4 pp.27-34 (1987)

[2] 赤間清：継承階層 prolog の高速化機構，人工知能学会誌，Vol.2, No.4, pp.492-500 (1987)

[3] 赤間清：継承階層 prolog による自然言語処理，情報処理学会，自然言語処理研究会資料，62-7, pp.45-52 (1987)

[4] 赤間清：集合束縛変数に基づく意味表現とユニフィケーション，情報処理学会，自然言語処理研究会資料，62-8, pp.53-60 (1987)

[5] Sowa,J.F. : Conceptual Structures, Addison-Wesley P.C., p.481 (1984)

[6] Ait-kaci,H. and Nasr,R : LOGIN: A Logic Programming Language with Built-in Inheritance, the journal of logic programming, Vol.3, No.3, pp.185-215 (1986)

[7] Dincbas,M.: Domains in Logic Programming, Proceedings of AAAI '86, pp.759-765 (1986)