知識工学と人工知能 41－11
（１９８５．７．４）

Conceptual Graphによる談話理解に向けて

日本アイ・ビー・エム株式会社　サイエンス・インスティチュート
丸山 宏

１．はじめに

役に立つ自然言語理解システムにおいては、構文解析、意味解析の他に、ユーザーの発話の意図を知るための談話解析が必要である。例えば、秘書システムに向かって『明日Mr.Jonesに会いたい』と言った場合には、『meetingの場所と時間を決めて、appointmentをとってくれ』という意図であると解釈しなければならない。

このように、ユーザーの意図を把握するために、入力文にははっきりと述べられていないことを推察するなんらかの推論機構が必要であることがわかる。この種の推論（『だいたいそうだろう』という推論、または情報補完のための推論）は論理的な演繹とはちがって必ずしも正しいことは保証されない。しかし、人間との知的なインターフェースを作るにあたっては必要不可欠といえるものであろう。

Conceptual Graph（以下ＣＧと略す）[1]による知識表現には(1)joinとよばれるパターン・マッチング、(2)論理的な演繹、(3)actorによる計算、という３つの推論機構が定義されていて、述語論理や、Conceptual Dependency [2]、フレーム[3]などの特徴をうまく取り入れたものだといえる。本稿ではＣＧを談話理解に応用する際における利点と問題点を明らかにする。特に、筆者がインプリメントも考慮に入れた上で、ＣＧに加えた制限について詳しく触れる。現在、残された最も大きな問題は、スキーマを適用する順序と、失敗がおきた場合にどうするかというコントロールに関する問題である。２章でＣＧについて、３章ではＣＧの問題点について述べ、４章ではＣＧのsubsetを提案する。残された問題点については、５章で議論する。

２．Conceptual Graph

ＣＧはセマンティック・ネットワークの一種である。セマンティック・ネットワークは、解釈が個別のLispプログラムに依存するため、意味があいまいになりやすいといわれるが、ＣＧでは各ノードにタイプとidを与えていて、それらを使って述語論理風の意味を与えることができる。例えば、

(a)　"A boy is eating fast."
(b)　"A boy, Koichi, is eating an Otabe."

という文の意味はＣＧを用いてそれぞれ、

(a)　[BOY] ←(AGNT) ← [EAT] →(MANNER) → [FAST]
(b)　[BOY:浩一] ←(AGNT) ← [EAT] →(OBJ)→ [OTABE]

というように表わすことができる。図の中で[...]は個々のコンセプトをあらわし（...）はコンセプト間の関係をあらわす（関係はコンセプトの一種ではないかという議論もあるだろうがここでは一応別のものとする）。[...]の中に書かれているのはこのコンセプトのタイプであり、したがって[...]全体でそのタイプの一つのインスタンスをあらわす。例えば、[BOY] はあるひとりの少年をあらわす。論理式で書けば $\exists x$: boy(x)である。先ほどの(a)のグラフ全体を論理式であらわそうとすると

$$\exists xyz\text{: } boy(x) \ \& \ eat(x) \ \& \ fast(z) \ \& \ agent(x,y) \ \& \ manner(y,z)$$

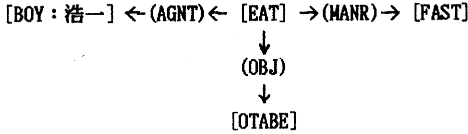
のようになる。

[BOY] はあるひとりの少年であったが、特定の少年を指すには：（コロン）の後ろに対象となるもののidをつけて、例えば [Boy：浩一] のようにあらわす。このidの部分にはその他にも変数や集合や数値などが書けるようになっている。

タイプはラティスになっていて、例えば少年は人間のサブ・タイプだとか、そういうことが（ネットワークとは別のところで）定義されていることを仮定している。ラティスになっているという意味は、任意の二つのタイプに対して共通のスーパー・タイプと共通のサブ・タイプが必ず存在するという意味であり、従ってすべてのタイプのスーパー・タイプである $\top$ (top) とすべてのタイプのサブ・タイプである $\bot$ (bottom) の存在を仮定している。

join

ＣＧはネットワークであるから（つまり図式的な記法で書かれるから）本質的にパターン・マッチングになじみやすい。ＣＧにおけるパターン・マッチングの操作をjoinと呼ぶ。joinは二つのグラフの重ね合せのことである。例えば先ほどの二つのグラフを[EAT]というコンセプトの上でjoinすると次のようなグラフができる。



Joinは特定のコンセプトの重ね合せから始まって、関係のアークをたどって重ね合せることのできるコンセプトを順に重ね合せていき、これ以上できなくなるまでつづけるものである（Sowaによればこれは"maximalな"joinであり、単にjoinといった場合にはどの時点でとめてもよい。しかし、maximalでないjoinは通常考える必要がないと思われるので、本稿ではmaximalなものを単にjoinと呼ぶことにする）。二つのコンセプトを重ねあわすことができるのは次の二つの条件を同時にみたしている時である。すなわち、

（1）双方のタイプが同じか、片方が他のサブ・タイプになっている
（2）どちらかのidが変数であるか、または双方のidが等しい

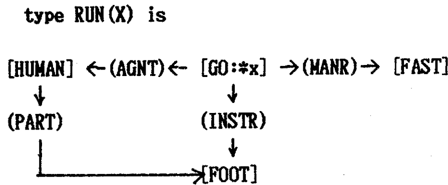
重ね合せた結果のタイプは、どちらか小さい方のタイプになり、idは双方が変数ならば変数のまま、どちらかが特定のidならばそのものになる。上記の例で［BOY］と［HUMAN：浩一］の重ね合せは［BOY：浩一］をつくる。ただし、BOYはHUMANのサブ・タイプであると仮定する。

（Maximalな）joinの求め方は一般には一意でない。どのコンセプトを先に重ね合せるかについては非決定性が残る。この点に関しては次章以下で再び触れる。

Joinをうまく使うと、１章で述べた『常識による情報の補完』を行なうことができる。Sowaによると情報補完のための『常識』には『いつもそうである』というものと『たぶんそうだろう』という２種類があり、それぞれタイプ定義とスキーマ定義と呼ばれる。

例えばＲＵＮというタイプの定義は次のようになるだろう。

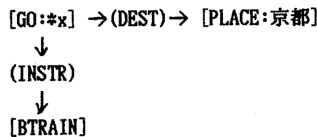
type RUN(X) is



この例は『走る』というのは『行く』の一種であり、『人間』が自分の『足』で『速く』『行く』ことである、というふうに読むことができる。

これに対してスキーマは『そうかもしれない』という知識をあらわす。例えば、『京都に行く』といった場合の『行く』はもしかしたら新幹線で行くことかもしれない、ということをあらわすには

schema go(X) is



というスキーマを書く。もし、『浩一は京都に行く』という事実があった場合には、このスキーマをjoinして、『新幹線で』という情報を付加することができるわけである。

スキーマ定義やタイプ定義は、SchankのいうところのSCRIPTにあたることがわかるであろう。あらかじめ背景となる知識を記憶しておき、それとのパターン・マッチングをとることによって発話の中にあらわれない情報を推定することができるわけである。

<u>deduction</u>

先ほども　述べたようにＣＧには論理式としての意味をもたせることができる。したがって、形式的な推論が可能である。述語論理の意味論としては、公理的意味論とモデルをもとにした意味論とがあるが、Sowaはそのどちらに対してもＣＧの図形的性質をうまく利用した推論規則を用意している。

公理的意味論は論理式の意味（真偽）を、あらかじめ与えられた公理から導くことができるかできないかで与える。ＣＧであらわされた（一階の）論理式に対してはSowaは７つの推論規則を与えている。公理にこれらの推論規則を適用して問題となる論理式を得られるかどうかでその論理式の真偽（厳密にいうと恒真であるかどうか）を問うわけである。この推論規則は健全かつ完全であり、また図形的な直観に訴える規則なので、人手でこれらの演繹を行なうにはむいていると思われるが、他の多くの演繹システムと同様、計算量の点で実用的とはいいがたい。

モデル論では、論理式の意味はあるモデルを与えて、そのモデルに照らして考える。すなわち、その論理式であらわされる関係がモデルの中に存在するかどうかを調べ、そうであれば真とするわけである。その際、もしモデルの中に存在しなければ、その論理式を偽とみなす説（閉世界仮説）と、存在しなくても、もしかしたら真かもしれないからそれは明示的に偽だという事実がない限りまだ未知であるとする説（開世界仮説）とがある。ＣＧでは双方の仮説についてあるモデルが与えられた時に、問題となる論理式が真であるかどうかを調べるアルゴリズムを定義している。しかし、これについても筆者の知る限り効率のよい処理系は実現されていない。

actor

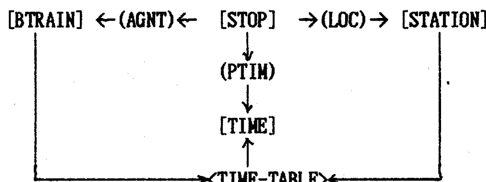
このようにコンセプトと関係のアークで世の中の静的な（あるいは宣言的な）意味表現がひととおり可能なわけであるが、この枠組の中で手続きを表現するのは困難である。したがって、ここでＣＧにもう一つのノード、アクターを導入する。アクターはある手続きを代表していて、入力アークにデータがそろった時に起動されて出力アークに結果を流す。例えば

[NUM:３] ⟶　　<PLUS> ⟶ [NUM:Ｙ]
[NUM:Ｘ] ⟶

というグラフのＸのところに５が入るとアクターPLUSが動いてＹに８を束縛する。つまりＣＧにアクターを導入することによってＣＧをデータ・フロー・グラフとみなすこともできるわけである。

アクターをスキーマのグラフの中に含めれば、そのスキーマをJoinすることによって手続きを起動することができる。例えば、『（新幹線が）停まる』というスキーマをつくった際に、同時に時刻表のデータ・ベースを参照する手続きも書いておくには、

schema stop(X) is



とすればよい。アクターTIME-TABLEは必要な情報がそろった際に実際のデータ・ベースを参照する手続きを起動して、残りの情報を得ることができる。

ネットワークによる意味表現をデータ・フロー・グラフと見做すのは、非常にうまい宣言的／手続き的知識表現の組み合わせであり、このことがＣＧを知識表現のための現実的な道具としての可能性をきわだたせている。

３．ＣＧの問題点

述べたようにＣＧでは３つの推論機構が提案されているわけだが、談話理解という目的のためには、これらをうまく組み合わせなければならない。ＣＧをデータ・ベース照会に用いる枠組はSowaによって示されている。それによると、入力文をまずＣＧになおし、それに対して

（１）　タイプ定義を展開する（joinする）
（２）　スキーマをjoinする
（３）　deductionを行なってそのグラフが今までのモデルに照らして矛盾しないかを調べる。
（４）　（２），（３）を可能な限り繰り返す。
（５）　できあがったグラフのアクターを動かす
（６）　入力グラフにはあらかじめユーザーが求めているノードに印がつけられており、そのノードの値が求まったらそれを印刷する。

というプロセスを行なう。

しかし、この手続きは単文によるデータ・ベース照会ならばよいが、人間と計算機との対話を通して人間の意図を明らかにしていくという談話理解本来の目的とは大きなへだたりがある。例えば、ユーザーは質問だけではなく

て、データの転送とか、計算機に対して実際のアクションを起こして欲しいのかもしれなし、また、計算機の方はユーザーの意図があいまいだった場合にはユーザーに問いあわせを発しなければならないだろう。

また、スキーマを適用する度に重いdeductionを行なうのは極めて効率が悪いし、心理学的に考えても、現実の人間が過去の記憶を使う度に毎回論理的な推論を行なっているとは考えにくい。

せっかく３つの強力な推論機構があるのに、それらを組み合わせて一つの意図をもったプロセスにする枠組がはっきりしていないのがＣＧの第一の問題点である。

ＣＧの第二の問題点は各コンセプトから出るアークの周囲や数がきちんと定義されていないことである。Sowaはある特定のタイプのコンセプトに対する制限をselectional constraintと呼んでいるが、これをどのように定義するかについては述べていない（入力グラフとタイプ定義、スキーマ定義がすべてselectional constraintを満たせば、それらをjoinして得られたグラフがselectional constraintを満たすことは示されている）。selectional constraintがあいまいであると、maximal joinの求め方のあいまいさを増大させる危険がある。すなわち、

```
[C1] →(R)→ [C2]
     →(R)→ [C3]
```

のように同じコンセプトから同じ名前のアークが２本でている場合にはjoinの相手はどちらのアークをたどるかによって２つの意味をつくってしまう。

アクターは強力な概念であるが、ＣＧではアクターに対してはjoinのオペレーションが定義されていない。また、ちょっとまともなことをやろうとすると、手続き的に書かなければならない部分が多くなり、アクターによってグラフが複雑になりがちであることも問題点といえよう。

４．ＣＧ subset

知識表現について一般的にいえることだが、推論機構についてはよく整理されているのにもかかわらず、それらは応用システムの呼び出される側のコンポーネントにすぎず、それらを呼び出す側の、トップ・レベルのループはad hocなものになりやすい。先ほどのSowaのデータ・ベース照会でもそうだし、SchakのＳＡＭ[4]などもそうである。Chat－80[5]のトップ・レベルは求まった論理式をPrologで実行して値のリストを印刷するだけである。つまり『システム全体の意図』をどのように実現するかが、あまり明確でなく、場当たり的であったわけである。この点でよくかんがえられているものにＧＵＳ[6]がある。ＧＵＳはフレームの『開いているスロットを埋めようとする』手続きが、システムのトップ・レベルであり、これがシステムの意図をあらわしている。

ここで提案するＣＧのsubsetでは、ＣＰ（Conceptual Processor）と呼ばれる一つの手続きがシステム全体の意図を表現するように気を付けた。すなわち、ＣＰを呼び出すシステムのトップ・レベルのループというものは存在せず、一度ＣＰを動かしはじめたらその中でユーザーへの入力促進、ユーザーの入力の解析、推論、判断、記憶、ユーザーへの応答、外界への働き掛けをすべて行なえるようにしたつもりである。

また、ＣＰは特定の分野に依存した知識をもたないものとする。したがって、後に述べるスキーマとアクターの定義をいれかえるだけで、あらゆる応用分野に適用できるインタープリターとしてＣＰを設計することにする。

ここで提案するＣＰは推論規則としてＣＧのjoinとアクターの２種類のみをつかう。先にも述べたようにdeductionは効率のよい実現が困難と思えること、５章で述べるようにdeductionはアクターの特別な場合として見做せること、実際に人間の意図の理解のためにdeductionが必要な場面は少ないと思われることが、deductionをやめることにした理由である。さらに、deductionをやめれば、deductionをいつ行なうかというコントロール上の問題がなくなりＣＰがそれだけ簡単になる。

グラフの構成要素としてはアクターとコンセプトの区別をやめる。すべてのコンセプトは裏に手続きを持つことができることにする（もちろん手続きのないコンセプトがあってもかまわない）。例えば、『（新幹線が）停まる』というコンセプトの裏に『時刻表をひく』という手続きをつけておけば、グラフの構成要素として別にアクターを用意する必要がなくなり、先ほどのスキーマはずっと単純化される。アクターはスキーマと同様にタイプ毎に定義しておくことにする。

オリジナルのＣＧではselectional constraintの意味がはっきりしていないと前節で述べたが、ここで提案するsubsetでは、あるタイプのコンセプトから出るアークの数と形はタイプ・ラティスの中に定義しておくことにする。また、１つのコンセプトから２つの同じ名前のアークは出ないことにする。このようにするとＣＧはフレームにより近くなる。つまり、例えばタイプＧＯから出るアークは

```
[GO]  →(AGNT)→ [HUMAN]
      →(DEST)→ [PLACE]
      →(PTIM)→ [TIME]
```

のように決めておく。これ以外のアークは許さない。AGNT, DEST, PTIMのそれぞれのアークがフレームのスロットに対応する。

オリジナルなＣＧでは、情報補完のための『常識』にはタイプ定義とスキーマ定義の２種類があったわけだが、タイプ定義はそもそもスキーマの中で『100%そうなる』という特殊なものであった筈である。したがって、もしス

キーマに対して何らかの『確からしさ』を与えることができればタイプ定義は『確からしさ』が100%であるスキーマ定義でおきかえることができる。したがってタイプ定義はやめることにする。

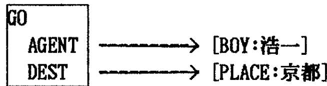
先ほど述べたように、ＣＰの中でユーザーとの入出力、システムの他のコンポーネント（例えばデータ・ベース、OSなど）とのインターフェースを実現してしまうことが大きな目標の一つである。このために、これら外界とのインターフェースはすべてアクターに任せることにする。何かのタイミングでユーザに問い合わせをするアクターが起動されるとユーザへの質問が発せられ、入力待ちになり、ユーザの入力が得られるとそれを解析して自らのグラフにjoinしてまたＣＰの処理が続けられる。『それを印刷して欲しい』というコンセプトがグラフの中にあって、そのコンセプトに関する必要な情報がそろうと、『印刷』のアクターが動きだして実際にプリントアウトを行なう。アクター（すなわち手続き的意味表現）に負うところが大きいようにも思えるが、いずれにせよ役に立つシステムにするには最低限の手続きを用意しなければならないわけで、知識の中で不可欠な部分であることには間違いがない。ただし、あまりにも安易に手続き型の表現に頼ることには危険がある。

<u>表記の問題</u>

今述べたような制限を設ければ、ＣＧはフレームの表記に似た形でかきあらわすことができる。すなわち、

[GO] →(AGNT)→ [BOY:浩一]
→(DEST)→ [PLACE:京都]

を



というように表わすことにする。すなわち、各アークはその出発点であるコンセプトの中のスロット名として書くわけである。スロットをもたない（あるいはどのスロットにも値のない）コンセプトは略記法として［　］という形で書いてもよいことにする。さらに、あいまいさのない場合には、idの定まっているコンセプトのタイプは省略してもよいものとする。

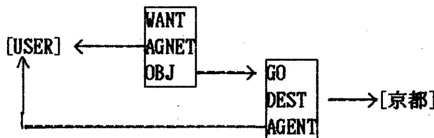
<u>例題</u>

例題として、『京都にいきたい』というユーザーの入力を

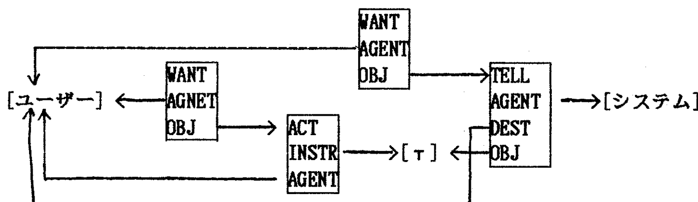
（１）『京都にいきたい』は『京都に行く手段を教えてくれ』だと解釈する。
（２）京都に行く手段として新幹線を考える。
（３）ユーザーの希望の時間を尋ねる。
（４）実際の時刻表のデータ・ベースを照会して特定の列車を求める。
（５）それをユーザーに表示する。

のように処理する例を示す。

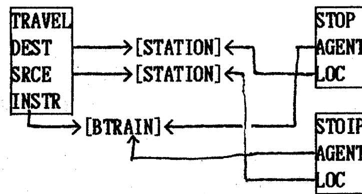
まず、ユーザーの入力は次のようなグラフになっているとする。



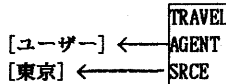
これに対して『ユーザーが何かをしたいと述べているときは、その手段を教えて欲しいという意味である』というスキーマ



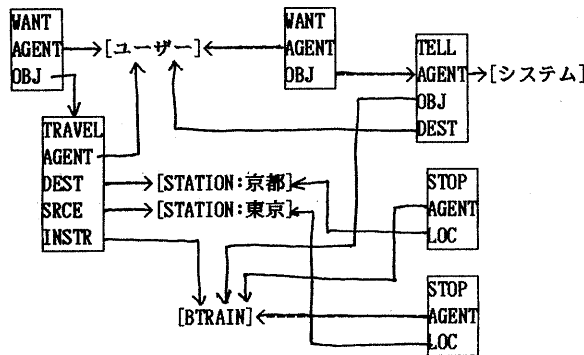
があって、それをjoinすると、『京都に行く手段を教えてくれ』という意味のグラフになる。ここで『（駅に）行く』とは新幹線でいくことかもしれないというスキーマ



旅行の出発点は多分東京だろうというスキーマ



をjoinすると、『東京から京都に行く新幹線を教えてくれ』というグラフ



になる。ここでSTOPのアクターが動きはじめて（まだ時刻表をひきはじめるには情報が足りないので）ユーザーに何時に京都につきたいかをたずねる。その時刻がわかると（データ・フロー・グラフであるから）実際の時刻表をひきにいくアクターが動きだして、ひかり号を一つさだめる。すると、コンセプトTELLにつけられたアクターが動くことができて、そのひかり号をユーザーに知らせる。

このようにして、アクターとスキーマをうまいタイミングで適用していくことによって、ユーザーの意図にそったシステムのふるまいを実現することができる。この際、最も問題となるのが『適当なタイミング』をどのように定めるかである。これについては次章で考察する。

インプリメントについて

ＣＧのjoinのオペレーションはPrologのunificationになじみやすい。すなわち、２つのコンセプトの重ね合せのできる二つめの条件、『少なくとも片方が変数であるか、双方のＩＤが等しい』はunificationが成功する条件と同じだし、そのjoinの結果、『両方が変数ならば変数、片方がＩＤならばそのＩＤ』もunificationの結果と一致する。

タイプの一致については、タイプはラティスではなく木であることにすればunificationのアルゴリズムで実現することができる。例えば、HUMANというタイプとBOYというタイプをPrologのデータ構造としてそれぞれ、

```
object(animate(human(*)))
object(animate(human(boy(*))))
```

とあらわすことにすれば、どちらかが相手のサブ・タイプであるときに限り、unificationが成功し、その結果はどちらか小さい方になる。

スロットの値の重ねあわせについても親のタイプのスロットを全部受け継ぐことにすれば同じ考えでunificationのアルゴリズムに任せることができる。すなわち、ACTとGOのスロットをそれぞれ、

```
agent(X1): ptim(X2): *
agent(Y1): ptim(Y2): dest(Y3): *
```

のようにあらわせば、これらのunificationだけで、それらの先につながっているコンセプトのjoinの一部をやってくれる。joinの一部、とことわったのは、このunificationでできるのは、これらのコンセプトが『指している』コンセプトたちだけで、これらのコンセプトが『指されている』コンセプトのjoinは行なわないからである。したがって最初にキーとなるコンセプトどうしのunificationをおこなったのち、それらの対応するコンセプトを指すものどうしのjoinを繰り返さなければならない。

５．問題点について

コントロール

前章でみたように、『適当な』タイミングでアクター／スキーマを適用すれば、人間の直観によくあうシステムの応答が達成されそうである。しかし、この『適当なタイミング』というのがくせもので、例えば『何時に京都のつきたいか』を先に質問するか、『何時に東京を出たいか』を先に質問するか、というような問題は、この『タイミング』によって決まってくる。

このようなシステムのふるまいのちがいはシステムが何かの選択をせまられた時に、どの選択肢をとるかによって決まってくる。前章で提案したCGのsubsetの場合、このような選択のポイントは以下の４つになる。

（1）入力グラフのどのコンセプトに注目してスキーマ／アクターを適用するか
（2）そのコンセプトに対してどのスキーマ／アクターを適用するか
（3）スキーマをjoinした際に生じるあいまい性
（4）アクターを動かした際に、答えが複数個ある可能性がある。それらのどれを選ぶか

これらの選択点で、できるだけ正しい選択をすることがユーザーの意図をできるだけ正しく把握し、無駄な質問やまちがった推測をさけることにつながるわけである。

さらに問題をややこしくしているのが、これらの選択点のやり直しの問題である。システムがどれかの選択肢をえらぶということは、『たぶんこうだろう』という推測を行なっているということであるが、もし、この推測がまちがっていたらどうなるのだろうか？　そのまちがった原因をうまく捜しだし、それだけをやりなおしてユーザーの意図を正しく推測することが可能だろうか？

現在検討中のコントロール・ストラテジーは以下の４つである。

(1)request mark法

ある入力グラフが与えられた時、その中の１つのコンセプトにあるマークがついていると仮定する。これを、request markと呼ぶ。システムはrequest markのついたコンセプトのidを求めようと努力する。もしこのコンセプトのidが求まらないと、このidを求めるために必要なコンセプトにrequest markを伝ぱする。もしそれがわかれば、適当なアクターによってもとのコンセプの値が求まるわけである。

(2)優先順位法

各ノードに優先順位を与えておいて、優先順位の高いものから順にスキーマ／アクターを適用していくもの。この戦略はインプリメントされていて、前章で述べた例は、このプログラムの実際の実行例である。

(3)スキーマ／アクター定義のLRU法

各ノードではなく、スキーマ／アクターのそれぞれに優先順位を与えて、それらの全体をひっくるめてLRU法で優先順位を管理する方法。

(4)inference step法

ある段階で、適用可能な一連のアクター／スキーマの集合に関して非決定性を考える。

やり直しを考えなければ、今述べたいずれかの方法で、うまくＣＰを作ることはできそうである。しかし、やり直しをきちんとおこなおうとすると様々な問題が出てくることがわかった。知的なやり直しを行なうことも考えたが、ＣＰのコントロールのメカニズムを複雑にすることは避けたい。LRU法ですべてがうまく行く方法を摸索中である。もしかしたら、やり直しなどというものはそもそも必要ないのかもしれない。人間が自分のまちがいをやり直しをすることによって忘れてしまうことなどありそうにないからである。

deductionの導入

例えば『京都につく終電車は？』ときかれた時、終電車の意味は『それより遅い電車がない』という意味である。『それより遅い電車がない電車を求めよ』などという推論は、パターン・マッチングむきでなく、logicにむいている。

オリジナルなＣＧのように、コンセプトの値として、グラフそのものをもつ、すなわち、グラフの入れ子構造を許せば、deductionの手続きはアクターとして定義できる。論理式をあらわすコンセプトのタイプとして、

PROPOSITION, AND, OR, NOT

などを用意しておき、それを用いて論理式をつくる。それらの論理式をスロットの値としてもつタイプとして、

YESNO, ALLOF

というタイプを用意して、それぞれ、その論理式の真偽と、その論理式をみたすすべての指定された変数の値を求めるアクターを定義しておけばよい。

論理式の中身もやはりＣＧで書くわけであり、これに対するjoinやアクターの動きを考えると、ＣＰが複雑にならざるを得ないだろう。このあたりは十分に検討の余地がある。

参考文献

[1]Sowa,J.F.: Conceptual Structure, Addison-Wesley, 1984.
[2]Schank,R.C.: Conceptual Information Processing, North Holland, 1975.
[3]Minsky,M.: A Framework for Representing Knowledge, in:
The Phychology of Computer vision, McGraw-Hill, 1975.
[4]Schank,R.C. and Riesbeck,C.K. (eds.): Inside Computer Understanding, Lawrence Erlbaum, 1981.
[5]Pereira,F.: Logic for Natural Language Alalysis, SRI technical Note 275, 1983.
[6]Bobrow,D.G. et. al. : GUS, A Frame-driven Dialog System, Artificial Intelligence 5, 1977.