言語研究のための補助手段・思考実験手段としてのドイツ語処理ミニパーザの試み

竹内, 義晴
九州大学言語文化部言語科学部門（ドイツ語）

https://doi.org/10.15017/6796265

出版情報：言語科学. 24, pp.71-96, 1989-03-01. 九州大学言語文化部言語研究会
バージョン：
権利関係：

言語研究のための補助手段・思考実験手段としてのドイツ語処理ミニパーザの試み

竹内 義晴

0. 言語研究者が計算機を相手にする動機

自然言語を計算機で処理することを考える場合、いろいろな動機というものが考えられると思う。たとえば、計算機上にパーザ（構文解析装置）を構築するとしよう。この場合、動機について次のように大きく三つの動機に分類をするのも一つの考え方だと思う。一つには、実用を目的とした動機、二つ目には、自然言語のシステムを忠実に移し取ろうという、写実主義的な動機である。この二つの動機が、およそ、即座に頭に浮ぶものだろう。三つ目に私が考えるのは、自然言語のことを研究する場合の補助手段、思考実験のための道具、あらっぽいスケッチや、石膏細工、化学における分子模型のようなものとして、計算機に言語をシミュレイションするシステムを構築してみる、というような動機である。後でもう少し詳しく議論するが、私は、この三番目の動機を「言語研究のための補助手段・思考実験手段をもとめる動機」と呼びたい。

0.1. 実用を目的とした動機

言語に関する科学的な知識と計算機上のシステムとあいだに、直接的な対応関係を考えることは、実用を目的とした場合、かえって妨げになることが多いだろう。処理速度、能率、を考慮するならば、複雑な部分は適当に処理するか、または人間にまかせるメカニズムにしておいたほうが便利に違いない。日本語ワードプロセッサの大抵のものは、何らかのパーザを備えていて、文節、または文節をこえた大まかな分析を実行しながら、「かな」で表記された表現を、「かな漢字まじり」の表現に変換している。しかし、変換が適切かどうかのチェックは人間がおこなっている。複雑で、細かな処理をするように仕組まれたシステムにかぎって、おかしな変換をおこなって、後で人間が苦労するはめになったりする。

また、処理対象となる自然言語の本質としての豊かさ、変異の可能性と表現の拡がりを考慮するならば、入力されるテキストに偏りがある場合、語彙においても、処理システムにおいても、それなりに偏りを持たせておいたほうが、分析の効率がよいだろう。実用をめざしている自動翻訳システムの多くは、科学論文や、その抄録、機械のマニュアルの翻訳などに焦点を合わせているが、その一つの理由は、それらのテキストが「構文が単純である」という特性を備えているからだ。また、分野にもよるだろうが、この種のテキストでは、語彙の範囲が限られている。同音の語彙が複数ある場合、そのテキストの専門性に応じて、語彙間の選択の順位を決めておくことも、処理の効率を高めるだろう。

醸造学のテキストを処理するシステムでの語彙挿入では、「発行」よりも「発酵」が選ばれるほうが効率がよいだろう。また、多くの日本語ワープロの語彙学習機能というのは、何らかの形で、現行テキストにおける同音語彙の出現頻度をあてにしているが、あれはあれで、困った処理をしてしまうこともあるが、大ざっぱに見て、上に述べたような処理効率を高めるために役だっているといえる。

しかし、このような、語彙の選択を頻度として決定することは、構文の分析の本質とはまったく関係がない。とは言っても、人間が自然言語を処理する場合、何らかの形でそのようなことをしているには違いない。私たち人間は、醸造学の話を聞いている時は、

「ハッコウ」といわれると、何となく「発酵」のことを考えてしまうし、文脈のなかで一度「発酵」という理解をすると、次の「ハッコウ」という音を、おなじように「発酵」と理解しようとする傾向があるだろう。これは、人間の情報処理機構がそのようなメカニズムをもっているのだろう。しかし、「学会誌をハッコウする」という場合に、いくら醸造学という場面設定でも、「発酵」では困る。もちろん意味処理の問題としても、学会誌という対象は「発酵する」ということと結びつけにくいしろものだ。構文解析の問題としても、「発酵す」という述語表現は「～が」とか、「～は」と結び付く表れ方をするのであって、「～を」と結び付く表れ方はしにくい。

実際には、語彙の選択は、「場面設定」レベルにおける処理、「意味理解」レベルにおける処理、「構文解析」レベルにおける処理など、様々なプロセスの相互的・総合的出力として決定されると考えられるべきだ。私たちは「流し読み」、「流し聞き」ということをするし、しようと思わなくてもしてしまう。このような場合、言語理解の情報処理は、「場面設定」レベルにおける処理に依存する度合いがそうとう高いだろう。しかし、そんな、「場面設定」レベルにおける処理への依存度が高い場合でも、そのレベルで変な処理が実行されそうな場合には、「意味理解」レベルにおける処理機構、「構文解析」レベルにおける処理機構など、他のレベルの処理機構が働いて、聞きまちがいや、誤解をふせぐだろう。全体のメカニズムが、柔軟かつ重層的で、ミスを少なくする、巧妙な仕掛けになっているようだ。

このような、巧妙で、重層的なメカニズムの問題を、単純に、頻度・確率の問題にしてしまうのは、くそまじめに、システムを複雑に組んで、計算機の処理速度を落としたり、または、計算機のメモリーに負担をかけたりすることを避けるためだ。計算機の処理速度は毎年驚異的に早くなっているということだけれど、それにもかかわらず人間の科学がそのような研究対象の複雑さに現在のところ追いついてゆけないのだから、そのような処理も知恵の働かせ方なのだろう。人間の科学は、人間が言語をあやつり、文化の枠組のなかで生活いているという、身分自身の現実を、依然としてそんなにきっちりとは把握できていない。人間の科学的な知恵が追いかけつくせないものはいいかげんに処理しておいて、後は人間の能力にまかせられればてっとり早い。

話をもとに戻すが、科学技術的・実用的なテキストでは、いわゆる口語的、または文学的な、文体上の多様性をそなえた複雑な構文が現われる頻度は少ない。また、気紛れ、発話途中の言い換え・言いなおし、発話の作戦変更などの、言語外的な要因がかんでくることを考慮する必要もない。これらの、文体的、また、言語外的な要因による、構文の複雑さは、自然言語のシステムのもっとも自然言語らしい装置がはたらいている結果であって、決して、ことばづかいの「デタラメ」や「正しいことば」からの逸脱などではない。母語としての言語を修得する時期を逸した成人が未知の言語を学ぶ場合、たんなる未熟さからの「文法的間違い」ではなく、ネイティヴスピーカーたちがするような、つっかかりもっかかり、言いよどみ、ある言い方を途中でよして別の言い方に乗り換えること、などの「不作法」をするまでに「進歩」するのは大変なことだ。しかし、実用を考えるならば、このような繁雑なことは無視して、より単純なテキストの処理をめざしても、それなりの便利なものを作り上げることはできる。

0.2. 写実主義・科学指向の動機

他方、自然言語のシステムを計算機上に移し取ろうとするような意味での科学性をもとめるならば、パーザの構築にあたって、上に触れてきたような「いいかげんさ」を無原則に採用するわけにはいかない。このような場合、パーザは、言語研究の歴史がつみあげてきた成果の有効性が試される実験の場としての意味を持つ。このようなことを指向するパーザは、実行速度がいかに遅く、たとえ、一単位の発話の分析に一昼夜を費やそうとも、そのメカニズムは人間の言語能力をなぞるものでなくてはならないし、また言語能力についての科学的洞察を反映するものでなくてはならない。

このように言うのは簡単だが、実際の問題として、言語の科学は、人間の言語について、

今のところ大したことを解明してきているわけではない。このような事情のもとでは、いくら計算機学者が優秀で、ハードウェアやプログラム言語が高級なものであっても、いいかげんでないパーザなどできるはずがない。様々な主張が対立して、結論がでていない問題については、立場を選ばざるをえない場合には、暫定的な立場を選び、それが暫定的な立場の選択であるということをはっきりとさせておくのが科学の態度だろう。この場合、システムが問題なく稼働することが、ある立場の選択の正当性を積極的に保証するということではない。厳密に考えれば、問題をブラックボックスの中に投げ込んだのだから、そのブラックボックスの有効性が確認されるということが結果として得られるにすぎない。

他方、今日私たちにわかっている様々な言語現象についての知識をそれなりに忠実にパーザに組み込む、ということはどのような意味を持つだろうか。有効で実用にたえる構文解析を実現するためには、意味情報、場面情報などが投入されなくてはならない。そういう構文論的ではない要素と、構文分析装置としてのパーザとの役割分担をどこで線引きするのか。その点にかぎっても、問題はなかなか難しい。しかも、そういう問題を思い切って、バッサリと切り捨ててみたとしても、マジメにパーザを組むことは、素人の片手間仕事でできることではない。プログラミングのエキスパートと、言語学者との密度の濃い共同作業と、それに付随する、大量の補助作業が必要とされるだろう。プログラムを適切に構造化し、それぞれのモジュール毎に作業がうまく分配されたとしても、例えば辞書項目の入力だけでも、大変な単純作業の積み重ねになることは容易に予測がつく。

上記のようなパーザを組む試みが、どのような意味合いで生産的であり得るのだろうか。すでに述べたように、システムができ上がり、稼働したとしても、それは、パーザ作成の土台となった、言語学的な知識の正しさを積極的に保証するということではない。うまくいった場合、なるほど、パーザはうまく働くだろう。しかし、そのメカニズムは、すでに言語学者にとっては、予測ずみのものなのだけれど、正当であるという保証は依然として与えられないままなのだ。

実際には、そのようなパーザが首尾よく完成して、問題なく働くという結果にたどりつくことが、大変なことなのだから、完成したパーザと言語学者の仕事との関係について心配することなど、よけいなことなのだろう。むしろ心配なのは、システムを組み上げてゆく過程で様々な問題が出てきた場合、どのように実際的に処理してゆけるか、という問題だ。

もうわかっている、解明済みだ、と思っていた言語にかかわる理屈が、意外や意外、システム全体の中でうまくかみ合わなかったりすることがあるかもしれない。そういうことがあるとすれば、その場合、きっと計算機技術の成果の反映による言語研究の進歩を期待する者は、落胆するのではなく、むしろ、待っていたことがおきたのだと喜ぶべきなのだろう。もちろん、そういう壁には、計算機をつかって全体的なシステムを組んでみなくても、研究者が、頭脳明晰に突き当らなくてはならないのだろうが、人間のやることだから、そのへんが徹底しないのは、それは仕方がないことだ。計算機の助けを借りて、間違いや、思い違いがわかるのだったら、それはそれで結構なことだ。

ある、巨大システムを組むというプロジェクトにおいて、その基礎になるアイディアの変更が必要だということがわかったとする。プロジェクト末端で、膨大な手作業が積み上げられていて、それらの作業を統括して、全体のシステムをまとめあげる仕事が進んでいる段階であっても、もし、その基礎になるアイディアがよくないとわかれば、研究を最初からやり直す、というのがまっとうな理屈だ。もちろん、巨大プロジェクトというのは、そもそもがそういう浪費的な性格のものなのだろう。しかし、文科の研究者の貧乏性の発想なのかもしれないが、そんな浪費をしてしまっていいのだろうかと思ってしまう。また、私を含む文科の研究者一般にとって、そういう浪費的な研究環境は実現可能なものでもない。対象そのものが大きければ大きいほど、小回りのきく研究の環境を実現すべきであるという考えは、理想論にすぎないのだろうか。

言語学的にみて妥当なパーザを作成するということが目標として設定される場合、その作業が、言語研究者にとってどのような意味を持ち得るのかということについては、実は、私にはまだよくわからない。しかし、そのことは別として、そのような妥当なパーザが実現することが望ましいことには違いない。現在、計算機上で扱える、テキストデータベースは、様々な形で私たちにアクセス可能なものとなってきている。しかし、テキストデータベースを処理するための道具として使える検索・計数装置は、今のところ、一次元の線的構造における、単純な表現の組み合わせを手掛かりとするもののみだ。自然言語という、重層的な構造をもつ対象に有効な形で踏みこんでいくためには、どうしても、構文解析という手続きが必要になる。適切なパーザが存在するならば、テキストデータベースの利用の可能性は飛躍的に拡大するだろう。

そのような、構文解析をともなうテキストデータベースの利用が、科学的に妥当であるためには、パーザがいいかげんなものであってはならない。逆にいうと、信頼できる、妥当なパーザが実現されないかぎり、私たちは、テキストデータベースを使って検索・採取してきたデータを、自分自身で解析してみなくてはならない。データの検索における構文解析をパーザに任せることができる、そういう事態が実現するとする（そういうことが可能になるかどうかは、私は知らない）。その場合、そういう事態が、人間である言語研究者にとって諸手を挙げて好ましいことだというふうには、私には決して思えない。言語研究の面白さは自分の言語能力を使って資料を解析することのスリルと意外性にある、と私は感じているからだ。私はその点での留保はここで明らかにしておく。しかし、それにもかかわらず、テキストデータベースの利用にとって妥当なパーザの実現が「夢」であることには間違いない。

自然言語というのは、その本質については豊かで奥行の計り知れない、また、現象面については移ろいゆくことを本性とするような対象である。そのような対象について、ま正面から解析してみせる装置が、どの程度充分なかたちで実現し得るのだろうか。言語科学の現状を見る限りは、自然言語の基底についての議論すら、依然として定まっていないように思われる。そういう点を踏まえると、私はこの疑問についてはかなり懐疑的な見方をせざるをえない。

0.3. 言語研究のための補助手段・思考実験手段をもとめる動機

言語そのものを研究の対象とするする者にとって、写実主義的な科学性よりも、実用性をねらうパーザを作成するという課題は、興味をそそるものではない。科学的な意味で妥当なパーザを作成することについて言えば、実現性という点、そして、自分の言語研究に対する意味付けという点で、その仕事のもつ可能性を否定するというわけではないのだが、現在という時点では、必ずしも積極的な態度をとれるまでのものではない。それでは、私はここでどうしてパーザについての議論をしているのか。私は、言語研究者にとってパーザについての議論、または、自然言語を計算機処理することについての議論がまるっきり意味だ、と考えているわけではない。

現代の自然言語についての科学的探求という試みは、人間の言語活動という対象、または、文法＝人間の言語能力という対象についてのモデル化、形式化という要請の上に成り立っている。私は少なくともそのような立場にたって自分の仕事を進めてきている。しかし、すでに述べたように、そのような、形式化・モデル化されて提示される科学の成果が、全体として計算機の上に構築されるということは、現代の技術や、その技術を担ってゆく体制が充分でないという点からも、そして、自然言語という、奥行き深く、豊かで、多様・多彩な対象の性格という点からも、現実性に欠けている。

言語学における、モデル化、形式化という要請は、もっぱら理論的な関心からのものであって、実際に細部にわたる、全体的なシステムを完成させる、とかいうところまで目を向ける性質のものではない。計算機上に実際に何らかのシステムを築いてみようとする試みが言語研究者の気をひくものであるとしたら、その関心は、システムの本格的な実現に向いているというよりも、理論的な性格のものであり、限定的な性格のものだろ

う。ここでは、おもにパーザについての議論をするわけだが、そのようなシステムは、言語研究のための補助手段・思考実験手段をもとめて構築されるものである、ということを、しばらく議論したい。

言語研究のための補助手段・思考実験手段をもとめて構築されるパーザ（以下、「研究のための補助手段・思考実験手段としてのパーザ」とする）は、結果として実用的なものになれば儲け物だが、だからといって、実用をめざしはしない。また、結果として、自然言語のシステムを全体的に忠実に写像することになったり、また全体システムの一部分を構成することになったりすれば、それも願ったりかなったりだが、そういう結果をめざすものでもない。

すでに述べたが、私は、研究のための補助手段・思考実験手段としてのパーザを、思考の補助手段として位置づけようと思う。例えば、研究のための補助手段・思考実験手段としてのパーザは、研究者が、何らかの自然言語の現象について追求する場合に、研究者の思考を支えてくれるかもしれない。研究者は自分が自然言語の構文について考えたことを簡単なモデルとしてプログラムし実際に作動させてみることができる。この場合、すでに頭の中で考えたことを試してみるというのが、実験、つまり、仮説を検証するという作業という名に値しないことはハッキリとしている。つまり、試されるのは、モデルがその記述の対象となる自然言語の現象を妥当に表現しているか、ということではない。モデルが、モデルそのものとして矛盾がないか、うまくできているか、という意味での実験がなされるにすぎない。

しかし、この場合、モデルを実際作動させてみるということは、ダイナミックなスケッチを作るということと似ている。自然言語の本質的な特徴の一つは、言語のシステムそのものが静的なものではない、ということだ。だから、言語のダイナミックなシステムを、それぞれの局面を切り取って提示するというのではなく、因果関係を明示しつつ、連続的に提示できる、というのは、言語研究者にとって、耳に聞えたり、目に見えたりする、線上の現象とは別の、生成過程という次元をとりいれた多次元スケッチブックを手にしたようなものだろう。もちろん、頭のなかでわかっていれば、そして、そのことが規則の体系として提示できれば、それを作動させてみなくても、中身としては同様のことだ。しかし、たとえて言えば、コミュニケイションにおいて、文字ばかりの描写に、図面が加わり、さらに、映画やビデオなどのサポートが加わると、伝わることが結果として同様でも、理解・伝達の効率や、イメージの拡がりが段違いに高まるということと似ているだろう。このような、高次元の思考の補助手段を利用することによって、研究の世界がより柔軟な、また拡がりのあるものになる可能性は否定できないだろう。

以上のような意味合いでは、研究を支える道具・手段としてのパーザは、例えば、言語学の初歩教育のような場で、より有効に役立つかもしれない。言語の構文論的構造は、終端記号としての語彙、語彙、および複合表現についての範疇、それぞれの範疇に属する表現を操作対象とする規則によって記述される。このことについては、現代の言語理論は基本的に一致している。しかし、このような操作的・形式的な議論は、特に、初学者にとっては、少し取りつきにくいところがある。その原因のひとつは、理屈に「イメージ」がついていかないということにあるのだろう。

例えば、ベクトル座標の次元の数が多くなり、三次限を超えるようなものになれば、例えば私のような普通の人間の自然なイメージはついていきにくくなる。それと同様に、自然言語の構文論的構造について、その生成プロセスをダイナミックにイメージするのはむずかしい。自然言語が人間に対して与えるのイメージというのは、時間系列にそった線状の特徴だ。しかし、自然言語の構文論的構造は、この時間系列にそった線状の記号連鎖の全体に対して、同時的であり、しかし、別の次元へのプロセスの掘り下げをもとめる特徴を本質的にもっている。そのようなプロセッシングを私たちの頭脳はこなしていると思われるのだが、それをイメージするとなると、結構むずかしいことである。そのような、直接近づくことのむずかしい構造をもつイメージへの橋渡し、または複雑なことがらをイメージするための補助手段として、研究のための補助・思考実験手段としてのパーザは有効だろう。

また、このようなイメージの補助手段をつかえば、人間のイメージが不慣れな領域での、様々なでき事を実際に疑似体験するということで学習・処理してゆくことが、ある程度容易なのではないだろうか。例えば、ある自然言語の断片についての簡単なプログラムを自分で組んでみることによって、通常自分が使いこなしている自然言語の記号連鎖を、実際に人工的に作ってみるためには、ずいぶん煩雑な手続きが必要なのだということを、しみじみと体験することができる。または、逆に、機械というものは実に融通がきかなくて、その点、人間のことばを使いこなす能力がいかに巧妙で柔軟なメカニズムであるのか、ということがよくわかったりする。

0.4. 本論での試みの性格

ここで実際に私が提示してみることは、計算機の上で記号を操作してみるという議論としても、また、自然言語、そしてドイツ語の構文論的構造についての議論としても、何らの新しい主張ではない。むしろ、(0.3) 項に述べてきたような意味での研究のための補助手段・思考実験手段をもとめて、言語学者が計算機を相手にするということを最も簡単な形で実行してみながら、具体的にいくつかの問題にぶちあたってみる、ということにすぎない。以下に提示する作業を通じて、それが整理されれば、本論の目的は達成される。つまり、「研究のための補助手段・思考実験手段をもとめる」動機という観点から、計算機をめぐる私なりの問題を、実践的に整理し、一定の展望を切り開いてみようというわけである。

言語研究者にとって、計算機というのは、絶ちがたい誘惑力をもった、ブラックボックスで、言語研究者は計算機について、さまざまなこだわりをもっている。しかし、それなりに基礎から勉強をつまずに計算機そのものを専門に相手にするなどということは、できることではない。だから、ここで提示する議論は、計算機による自然言語の解析に専門にたずさわってきた研究者からみれば、お粗末なものにちがいないだろうけれど、私は。ここで計算機専門の研究者の仕事に方を並べようとするわけでは決してない。

1. 道具立てについて

計算機にかかわる道具立てはとしては、私はごく簡単なものを使用している。日本電気という会社が作っている PC-9801vm という 16 ビットのパーソナルコンピュータが手もとにあって、その上で Prolog-KABA というプログラミング言語を走らせている。このプログラミング言語は、岩崎技研という会社から販売されているものだ。 Prolog は、人工知能研究のためのプログラミング言語として一般に知られている。その標準的なもののひとつが、Prolog-KABA というのだということだ。「ということだ」というようなことばの使い方をする程度に、私は計算機や、プログラミング言語のことはよくわからない。しかし、この程度のことを明示しておけば、ここで議論したことを読者が実際に試してみることができるはずで、私が計算機技術のことをほとんど知らなくても問題はないだろう。また、ここで議論される程度のことであれば、 Prolog-KABA 以外の Prolog でも、おそらく、問題なく試してみることができるはずで、パーソナルコンピュータ用の Prolog もたくさん売り出されているようだし、大学の大型計算機センターでも何種類かの Prolog が使えるようになっている。プログラムの内容については、私よりもっと計算機のことが身近ではない、文系の同僚のためにできるだけわかりやすく記述するつもりであるが、それも限度というものがある。 Prolog について知らない読者は、いろいろ出版されている入門書などをひもとく必要もあると思う。

(ただし、私の使っている Prolog-KABA は、 CPM/86 というオペレーション・システムの上で作動するものだ。私が購入して間もなく、 MS-DOS というオペレーション・システムの上で作動するものが売り出され、今日ではそちらのほうが一般に使われているらしい。私のプログラムを、直接コピーして試してみたいという場合、MS-DOS 版 を持っている読者は、なんらかの手を加えなくてはならないだろう。私としても、ソフトウェアの会社に、 MS-DOS 版と交換してくれと文句をいってみたのだけれど、なにぶんに

も向こうは商売なのだからどうにもならない。こんなところで文句たらたらいっている。もっとも、ここでの議論程度のものならば、プログラムをキーボードから打ちこんでも、たいした手間ではない。)

2. 「名詞句＋動詞」構文のプログラムを組む

2.1. 最も単純なプログラム

(A) は、 Prolog で組む、最も簡単なドイツ語の断片についての構文生成・解析プログラムの一つといえる。 Prolog というシステムでは、プログラムが走ることによって、プログラムが取り組むように指向している問題を計算機が処理できる体制に入る。 (A) のプログラムが走ると、計算機はプログラムに組み込まれた構文生成・解析の問題解決を目指す体制に入る。

```
(A)     s(S) :- S=NP+V, np(NP), v(V).          ------ (a)
        np(NP) :- NP=PROPN, propn(PROPN).      ------ (b)
        propn(PROPN) :- PROPN=inge.            ------ (c)
        propn(PROPN) :- PROPN=peter.           ------ (d)
        v(V) :- V=singt.                       ------ (e)
        v(V) :- V=laeuft.                      ------ (f)
```

(A) を走らせたあとで、 (1) のように打ち込み、リターンすれば、 (2) のような答を返してくる。 (2') のようにして (2) 答のすぐ後に下線部のように「;」を打ち込み、リターンすれば、さらに (3) の答を返してくる。この繰り返しで、 (4) 、 (5) の答を返してくるが、さらに続けて「;」を打ち込み、リターンすると、 (7) の答を返して、入力待ちのプロンプトを表示する。 (2) から (5) までの答のすぐ後に何も打ち込まずに、リターンすると、 (7) の答を返して、計算機は入力待ちのプロンプトを表示する。

```
1)      ?- s(S).
2)      S       = inge+singt

2')     S       = inge+singt;
3)      S       = inge+laeuft;
4)      S       = peter+singt;
5)      S       = peter+laeuft;
6)      no

7)      yes
```

2.2. プログラムの説明

(A) のプログラムについて、簡単に説明を加える。 Prolog のプログラムはいくつかの文で構成されているが、文は、頭から始まって、ピリオド、 「.」で区切られる。アルファベットの小文字で始まる表現は、定項または、n-項述語である (n は自然数を表わす) 。述語の後には括弧の中にコンマ、「,」で句切られた項が表記され、この、述語と項が結びついたものを複合項と呼ぶ。大文字で始まる表現は変項である。 「=」と 「+」は、中置き述語であるという点で、他の述語と異なる。 「=」は、その前後の表現が等しいことを表わし、使い方が Prolog のシステムで決まっている述語である。「+」 は、その前後の表現をつなぐ。

メダカのような記号の「:-」についてはいくつかの説明の仕方があるが、ここでは以下のように考えることにする。「:-」の左には、定項、または複合項が現われる。「:-」

の右には、定項、複合項、または、変項が、一つ、または、コンマで句切られて複数現われる。「:-」の右の表現が成功すれば、「:-」の左の表現は成功する。定項、あるいは複合項が成功するかどうかは、そのプログラムの内部で決定する。 (1) のように「?-」で始まる文は、その文が未成功の文であることを表わしているのだが、そのような文が入力されると、プログラムは、その文の成功をめざして動き出す。 (1) の文がプログラムの内部で成功する過程を以下で説明するが、その追跡をすれば、以上の説明のおよそは理解できるだろう。

「?- s(S)」という未成功の文が成功するかどうかを確かめてゆくためには、プログラムの内部で、この文の成功のための情報をもった文を捜さなくてはならない。(a) の文の、「:-」の左側の複合項は「s(S)」であり、「:-」の右に表現が連なっている。この (a) の文は、「:-」の右に連なっている「S=NP+V」、「np(NP)」、「v(V)」の表現が成功しなければ、「s(S)」は成功しないことを示している。このように、未成功の文を解決させるための情報をもった文をみつけることを、Prolog では「マッチ」という。 (a) によると、「s(S)」が成功するためには、まず、「S=NP+V」が成功しなければならない。これは、「S」が変項だから、「S」をそのまま「NP+V」と置きかえれば、成功する。その次に、「np(NP)」が成功しなければならないが、そのために、(b) とマッチする。 (b) では「NP=PROPN」が成功し、「propn(PROPN)」が成功しなければならない。「propn(PROPN)」を成功させるためには、(c) か (d) にマッチすればよい。Prolog のプログラムはは、上の方から順番に処理されてゆくから、 (c) にマッチすると、「PROPN=inge」は、変項「PROPN」に、定項「inge」を代入すれば成功する。すると、 (a) の文の「np(NP)」はこれで成功する。つぎに、(a) の文の「v(V)」は、同様に、 (e) か (f) にマッチすれば成功する。これで、(a) の文の「:-」の左側の複合項「s(S)」が成功するための条件はすべてみたされたことになる。すると、未成功の文、「?- s(S)」は成功ということになって、処理の仮定で変項に代入を繰り返してきた結果、変項「S」は「inge+singt」に置きかえられている。それで、Prolog は (2) の答を返してくる。

(2) のように、変項「S」を「inge+singt」に置きかえる解決の仕方が示された後に、「;」を入力すると、この解決の仕方は強制的に失敗と見なされ、システムは他の解決の仕方をさがそうとする。この場合では、強制的に失敗と見なされた解決が、最後の段階で「v(V)」の解決について、 (e) にマッチして成功したことで得られている。それで、その、 (e) にマッチして得られた成功を取り消して、「v(V)」の解決についての次のマッチの可能性をさぐる。そうすると、このプログラムでは (e) の下に (f) という文があって、「v(V)」はこの行とマッチする。この場合も、変項「V」に定項「laeuft」を代入して成功するので、全体が成功する。こんどは、変項「S」は「inge+laeuft」に置きかえられる。

さらに「;」を入力すると、今度は、「v(V)」の解決についてマッチする選択枝がプログラムの中にないので、さらに前回の成功への過程を後戻りして、「propn(PROPN)」の解決について、 (c) にマッチして成功したことを取り消し、マッチの可能性をさぐる。このようなくりかえしで、変項「S」が置きかえられる最後の可能性が試され、 (6) の答を出したあと、さらに「;」を入力すると、もう、これ以上成功することはないので、システムは (6) のように、「no」の答を返すことになる。上にのべたような、あるマッチした可能性が失敗するとそのマッチをやりなおし、他の可能性を探り、さらには、もっと前の段階まで後戻りしてゆくことを「バックトラック」という。全体が成功して、答を返してきた後に、「;」を入力せず、そのままリターンすると、もちろん、バックトラックを起こさずに、 (7) のように「yes」の答を返してくる。

ここまでの説明を辛抱して読んでもらえば、 (A) のプログラム、そして、Prolog というプログラム言語の最も基本的な動作は感じはつかめただろう。

3. プログラムからことばの問題へ

こんな簡単なプログラムでも気がつくことがいくつかある。ことばに問題を引きつけて頭をひねってみる。

3.1. 活用・変化形のあつかい

(A) のプログラムでは、三人称単数現在の動詞の活用形がそのまま用いられている。その都合で、名詞句にも、三人称単数の固有名詞だけを使った。問題を単純化するために、動詞の問題では、複数の問題、そして時制の問題は考えないことにする。 (A) のプログラムに、一人称、二人称の代名詞を組み込む。さらに、 (A) のプログラムでは、自動詞だけがつかわれていたが、他動詞も組み込むことにする。そうすると、プログラムの全体は (B) のようになる。

```
(B)     s(S) :- S=NP+V, np(NP), v1(V).
        s(S) :- S=NP1+V+NP2, np(NP1), v2(V) , np(NP2).
        np(NP) :- NP=PRON, pron(PRON)
        np(NP) :- NP=PROPN, propn(PROPN).
        pron(PRON) :- PRON=ich.                    ------- (g)
        pron(PRON) :- PRON=du.                     ------- (h)
        propn(PROPN) :- PROPN=inge.
        v1(V) :- V=singt.                          ------- (i)
        v2(V) :- V=kuesst.                         ------- (j)
```

下線を引いた部分が、変更されたところだ。システムによって生成される文の数が多くなるとわずらわしいので、名詞、自動詞、他動詞の選択の可能性をそれぞれ一つだけにしている。このプログラムは (8) のように 12 通りの答を返してくる。

```
8)      ?-s(S).
        S       = ich+singt;
        S       = du+singt;
        S       = inge+singt;
        S       = ich+kuesst+ich;
        S       = ich+kuesst+du;
        S       = ich+kuesst+inge;
        S       = du+kuesst+ich;
        S       = du+kuesst+du;
        S       = du+kuesst+inge;
        S       = inge+kuesst+ich;
        S       = inge+kuesst+du;
        S       = inge+kuesst+inge;
        no
```

(B) の (i) 、 (j) の行にみられるように、三人称単数の動詞の形しか、プログラムに組みこんでいないのだから、当然なのだけれど、 (8) で返されてきた答のうち、一人称、二人称のものは、動詞の人称変化の活用が間違っていて、困った答だ。また、プロプラムには (g) 、 (h) の行にみられるように、代名詞に、主格の形しか組みこんでないのだから、他動詞を用いた文の目的語は、固有名詞である「Inge」をのぞいては、みんなおかしな形になる。

(i) 、 (j) の行を (9) 、 (10) で置きかえ、 (g) 、 (h) の行を (11) 、 (12) で置きかえるのも、こういう問題を解決する一つの考えかただ。引用符の「'」で囲まれた表現は、文字列その物としてあつかわれる。だから、「'SING'」は変項ではなく、「SING」という文字列そのものを表わす。

```
 9)      v1(V) :- V='SING'.
10)      v2(V) :- V='KUESS'.
11)      pron(PRON) :- PRON='ICH'.
12)      pron(PRON) :- PRON='DU'.
```

動詞の活用や、代名詞の格変化形は、基本的に構文的な問題というよりも、形態的な問題だ。だからら、構文論的な抽象構造にたいして、何らかの、形態処理の解釈装置を構文解析装置と別に想定しておくと考えているのが、この解決法といえる。この考え方では、「singen」という動詞の様々な活用の形が、「SING」という、一つの文字列に抽象されている。システムのはきだす解答は (13) のようになる。

```
13)      ?-s(S).
         S       = ICH+SING;
         S       = DU+SING;
         S       = inge+SING;
         S       = ICH+KUESS+ICH;
         S       = ICH+KUESS+DU;                    ------ (k)
         S       = ICH+KUESS+inge;
         S       = DU+KUESS+ICH;
         S       = DU+KUESS+DU;
         S       = DU+KUESS+inge;
         S       = inge+KUESS+ICH;
         S       = inge+KUESS+DU;
         S       = inge+KUESS+inge;
         no
```

この場合、このような抽象的な表現を含む記号列に、形態的な解釈をほどこし、動詞の活用形を決定するというのは、構文解析との比較の問題として、決して簡単なものとはいえない。記号列中のある動詞の主語がどれかを決定しなければ、その動詞の人称変化は確定できないわけで、そのためには、構文解析の逆行という形で、出力された抽象的な記号を含む記号列に再び構文解析を加えていかなくてはならない。

ところで、実際にあるドイツ語の文を与えられいて、その解読を行う場合には、ある動詞の主語の確定したり、目的語を確定する場合には、動詞の人称変化形や、名詞句の格変化形をあてにしている。(13) ではきだされた文のうち、他動詞を用いた文では、こんな簡単な文でも、主語や目的語の特定はできない。

動詞の人称や、名詞句の格についての情報が欠けている、(13) の (k) のような記号列に形態的な解釈を加えると、複数の名詞句に同じ格が付与されることを排除するとして、(k1) と (k2) の二つの解釈ができてしまう。 (k) の記号列は、形態的な解釈を加えるためにはあいまいで、役にたたない。 (k) の記号列から、あいまいさを排除するためには、「KUESS」などの動詞を表わす抽象的な記号に、人称についての情報が付加され、また、「ICH」などの名詞句を表わす抽象的な記号には、格についての情報が付加されていなければならない。だから、この解決の試みは、かなりまずいといえる。

```
k1)      ich+kuesse+dich
k2)      du+kuesst+mich
```

3.2. 活用・変化形をあつかうプログラム

文の構文論的な出力に形態論的な解釈がされる場合、使われる語彙についての文中の役割・他の語彙や構成要素と関係が明示されていなければならない。そうでないと、形態論的な解釈では、必要最小限以上のものが出力されてしまう。つまり、文の構文論的な出力は、形態論的な解釈に対してあいまいだ。そういうことがらを適切に明示するため

には、語彙ばかりではなく、文のそれぞれの構成要素について、文中の役割・他の構成要素と関係などの情報が明示されていて、その情報が相互にやり取りされている必要がある。その結果、終端の出力まで、きちんと情報がおろされてくれば、その出力は適切であり、形態論的な解釈に対してあいまいさを含まないだろう。

(B) のプログラムを、(C) のように書きかえることによって、このような、文の構成要素間の関係を、それぞれの構成要素に付け加えることができる。このプログラムでは、それぞれの処理過程で、これらの情報をそれ以前の処理において発生、または引き渡されていた必要な情報があれば、それと引き較べる。そして、さらに、次の処理に引き渡す。(C) のプログラムは、(B) に較べれば、随分 (?) 複雑なプログラムになってしまった。

(C) 構文プログラム

```
formate(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=[CAT,SPECIFICS_1],
        CAT=s,
        NODE=[SPECIFICS_1,NODE_OUT],
        s(NODE,TERMS).

formate(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=[CAT,SPECIFICS_1],
        CAT=np,
        NODE=[SPECIFICS_1,NODE_OUT],
        np(NODE,TERMS).

formate(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=[CAT,SPECIFICS_1],
        CAT=pron,
        NODE=[SPECIFICS_1,NODE_OUT],
        pron(NODE,TERMS).

formate(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=[CAT,SPECIFICS_1],
        CAT=propn,
        NODE=[SPECIFICS_1,NODE_OUT],
        propn(NODE,TERMS).

formate(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=[CAT,SPECIFICS_1],
        CAT=v,
        NODE=[SPECIFICS_1,NODE_OUT],
        v(NODE,TERMS).
```

```
s(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=PLACE,
        PLACE=1,
        NODE_OUT=[NODE_1,NODE_2],
        TERMS=[TERMS_1,TERMS_2],
        NODE_1=[SPECIFICS_1,NODE_OUT_1],
        SPECIFICS_1=[CAT_1,[PERSNUM_1,CASE_1]],
        CAT_1=np,
        CASE_1=nom,
        formate(NODE_1,TERMS_1),
        NODE_2=[SPECIFICS_2,NODE_OUT_2],
        SPECIFICS_2=[CAT_2,[PLACE,PERSNUM_2]],
        CAT_2=v,
        PERSNUM_2=PERSNUM_1,
        formate(NODE_2,TERMS_2).

s(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS_0,NODE_OUT],
        SPECIFICS_0=PLACE,
        PLACE=2,
        NODE_OUT=[NODE_1,NODE_2,NODE_3],      ----- (l)
        TERMS=[TERMS_1,TERMS_2,TERMS_3],
        NODE_1=[SPECIFICS_1,NODE_OUT_1],     ------ (m)
        SPECIFICS_1=[CAT_1,[PERSNUM_1,CASE_1]], -- (n)
        CAT_1=np,
        CASE_1=nom,
        formate(NODE_1,TERMS_1),
        NODE_2=[SPECIFICS_2,NODE_OUT_2],
        SPECIFICS_2=[CAT_2,[PLACE,PERSNUM_2]],
        CAT_2=v,
        PERSNUM_2=PERSNUM_1,
        formate(NODE_2,TERMS_2),
        NODE_3=[SPECIFICS_3,NODE_OUT_3],
        SPECIFICS_3=[CAT_3,[PERSNUM_3,CASE_3]],
        CAT_3=np,
        CASE_3=acc,
        formate(NODE_3,TERMS_3).

np(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS,NODE_OUT],
        NODE_OUT=[SPECIFICS_OUT,NODE_OUT_OUT],
        SPECIFICS_OUT=[CAT_OUT,SPECIFICS],
        CAT_OUT=pron,
        formate(NODE_OUT,TERMS).

np(NODE_IN,TERMS) :-
        NODE_IN=[SPECIFICS,NODE_OUT],
        NODE_OUT=[SPECIFICS_OUT,NODE_OUT_OUT],
        SPECIFICS_OUT=[CAT_OUT,SPECIFICS],
        CAT_OUT=propn,
        formate(NODE_OUT,TERMS).
```

```
pron(NODE_IN,TERM) :-
        NODE_IN=[SPECIFICS,NODE_OUT],
        NODE_OUT=TERM,
        DIC_IN=[CAT,SPECIFICS],
        CAT=pron,
        dictionary(DIC_IN,TERM)

propn(NODE_IN,TERM) :-
        NODE_IN=[SPECIFICS,NODE_OUT],
        NODE_OUT=TERM,
        DIC_IN=[CAT,SPECIFICS],
        CAT=propn,
        dictionary(DIC_IN,TERM)

v(NODE_IN,TERM) :-
        NODE_IN=[SPECIFICS,NODE_OUT],
        NODE_OUT=TERM,
        DIC_IN=[CAT,SPECIFICS],
        CAT=v,
        dictionary(DIC_IN,TERM)
```

辞書プログラム

辞書基本規則

```
dictionary(SPECIFICS,TERM):-
        word(SPECIFICS,M_SPECIFICS),
        morph(M_SPECIFICS,TERM).
```

語彙規則

```
word(SPECIFICS,M_SPECIFICS):-
        SPECIFICS=[CAT,REST_SPECIFICS],
        REST_SPECIFICS=[PERSNUM,CASE],
        PERSNUM=[PERS,NUM],                    ----- (o)
        M_SPECIFICS=[WORD,CASE],
        CAT=pron,
        PERS=1,
        NUM=sg,
        WORD=ich.

word(SPECIFICS,M_SPECIFICS):-
        SPECIFICS=[CAT,REST_SPECIFICS],
        REST_SPECIFICS=[PERSNUM,CASE],
        PERSNUM=[PERS,NUM],
        M_SPECIFICS=[WORD,CASE],
        CAT=pron,
        PERS=2,
        NUM=sg,
        WORD=du.
```

```
word(SPECIFICS,M_SPECIFICS):-
        SPECIFICS=[CAT,REST_SPECIFICS],
        REST_SPECIFICS=[PERSNUM,CASE],
        PERSNUM=[PERS,NUM],
        M_SPECIFICS=[WORD,CASE],
        CAT=propn,
        PERS=1,
        NUM=sg,
        WORD=inge.

word(SPECIFICS,M_SPECIFICS):-
        SPECIFICS=[CAT,REST_SPECIFICS],
        REST_SPECIFICS=[PLACE,PERSNUM],
        M_SPECIFICS=[WORD,PERSNUM],
        CAT=v,
        PLACE=1,
        WORD=sing.

word(SPECIFICS,M_SPECIFICS):-
        SPECIFICS=[CAT,REST_SPECIFICS],
        REST_SPECIFICS=[PLACE,PERSNUM],
        M_SPECIFICS=[WORD,PERSNUM],
        CAT=v,
        PLACE=2,
        WORD=kuess.
```

形態規則

```
morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,CASE],
        WORD=ich,
        CASE=nom,
        TERM=ich.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,CASE],
        WORD=ich,
        CASE=acc,
        TERM=mich.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,CASE],
        WORD=du,
        CASE=nom,
        TERM=du.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,CASE],
        WORD=du,
        CASE=acc,
        TERM=dich.
```

```
morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,CASE],
        WORD=inge,
        CASE=nom,
        TERM=inge.
morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,CASE],
        WORD=inge,
        CASE=acc,
        TERM=inge.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,PERSNUM],
        PERSNUM=[PERS,NUM],
        WORD=sing,
        PERS=1,
        NUM=sg,
        TERM=singe.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,PERSNUM],
        PERSNUM=[PERS,NUM],
        WORD=sing,
        PERS=2,
        NUM=sg,
        TERM=singst.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,PERSNUM],
        PERSNUM=[PERS,NUM],
        WORD=sing,
        PERS=3,
        NUM=sg,
        TERM=singt.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,PERSNUM],
        PERSNUM=[PERS,NUM],
        WORD=kuess,
        PERS=1,
        NUM=sg,
        TERM=kuesse.

morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,PERSNUM],
        PERSNUM=[PERS,NUM],
        WORD=kuess,
        PERS=2,
        NUM=sg,
        TERM=kuesst.
```

```
morph(M_SPECIFICS,TERM):-
        M_SPECIFICS=[WORD,PERSNUM],
        PERSNUM=[PERS,NUM],
        WORD=kuess,
        PERS=3,
        NUM=sg,
        TERM=kuesst.
```

このプログラムでは、いわゆる構成規則を「formate」という述語で表わし、述語の第一項は規則の入力となる成分をあらわしている。この文構成要素には、それぞれ、カテゴリーが記載されている。それぞれの文構成要素は、そのカテゴリーに従って、「s」、「np」、「pron」、「propn」、「v」などの述語の操作に引き継がれている。「formate」の第二項には終端記号列が示されるようにプログラムが組んである。つまり、この構成規則が適用され、さらに必要ならば複数の規則が連続して適用された結果最後に出力される記号列が、「formate」の第二項に代入される。

「pron」、「propn」、「v」などの述語は、文構成要素を語彙に結びつける。それで、これらの述語の処理の結果は辞書規則に引き渡される。辞書規則には辞書基本規則があり、この規則は辞書が語彙規則と形態規則からなっていることを示している。

Prolog では「[」と「]」で囲まれた、コンマで句切られた記号列をリスト表現という。リスト表現は記号列の表現が得意だ。リスト表現によって、埋め込み構造の階層的記号列が容易に表現できる。また、リスト表現で表わされた記号列の要素に対して、削除、付け加え、並べかえ、置きかえなどの操作が容易にできる。このプログラムでは、文と、文の構成要素の構造を表わすのにリスト表現を用いている。例えば、(l) の行、つまり、(14) に見られるように文を表わすことになる「NODE_OUT」は、主語の名詞句を表わすことになる「NODE_1」、動詞を表わすことになる「NODE_2」、目的語の名詞句を表わすことになる「NODE_3」の三つの要素のリストとして表現されている。さらに、リスト内のそれぞれの要素はリスト表現に置きかえられることができ、例えば、(15) のように、記号列の階層的な表現を実現することができる。

```
14)     NODE_OUT=[NODE_1,NODE_2,NODE_3]
15)     NODE_OUT=[[SUBNODE_1,SUBNODE_2],NODE_2,NODE_3]
```

リスト表現のもう一つの特徴は、項目を列挙することが比較的容易にできるということだ。このプログラムでは、それぞれの構成要素が担っている情報が、リスト表現によって表現されている。(m) の行、つまり、(16) に見られるように、「NODE_1」は名詞句を表わすことになる変項である。この変項は、構成要素の全体の担っている情報、別の言い方をすれば、この構成要素を特徴づけている情報「SPECIFICS」と、この構成要素の本体を表わす変項の「NODE_OUT_1」のリストに置きかえられる。(n) の行、つまり、(17) に見られるように、この「SPECIFICS」という構成要素を特徴づけている情報を表わす変項は、この構成要素のカテゴリーを表わす「CAT_1」と、もう一つのリストからなるリストによって置きかえられる。このもうひとつのリストとは、人称と数を表わす変項「PERSNUM_1」と、格を表わす「CASE_1」とで構成されるリストである。人称と数を表わす変項「PERSNUM_1」は、(o)、つまり、(18) に見られるように、人称を表わす変項「PERS」と数を表わす変項「NUM」に置きかえられる。この名詞句の担う情報のうち、人称と数を表わす「PERSNUM」の部分が、動詞の担う情報と比較され、動詞の正しい変化形が保証される。名詞句を特徴づける情報「SPECIFICS」は、この一連の代入を行った結果としては、(19) に見られるような、重層的リストをなしていると考えられている（ただし、「CAT_1」に「np」を代入した）。このプログラムでは、複数は扱わないのだから、本当はこれらの情報のうち、「NUM_1」の項は、不要なのだが、動詞の変化形を扱うのに、一般に人称と数を組み合わせて考えるので、一応、数の情報も組みこんでおいた。プログラムの拡大を考えるのなら、名詞句の性を表わす項目も最低必要なわけではあり、そうすると、名詞句という構成要素が担う情報のリストは (19') のようなものになるだろう。

```
16)     NODE_1=[SPECIFICS_1,NODE_OUT_1]
17)     SPECIFICS_1=[CAT_1,[PERSNUM_1,CASE_1]]
18)     PERSNUM=[PERS,NUM]
19)     [np,[[PERS,NUM],CASE_1]]
19')    [np,[[PERS,NUM],[GERNDER_1,CASE_1]]]
```

(19) に表わされたようなリスト表現の形で、構成要素によって担われる情報がプログラムの実行の際に引きつがれる結果、このプログラムの出力では、(20) に見られるように、動詞の人称変化、名詞句の格変化の誤りはみられない。ただし、このプログラムで使っている、Prolog-KABA のシステムは、出力をベタ打ちしてくるが、(20) の表記では、読みやすくするために、適当に整理してある。

```
20)     ?-formate(NODE_IN,TERMS).

        NODE_IN = [[s,1],
                    [[[np,[[1,sg],nom]],[[pron,[[1,sg],nom]],
                       ich]],
                     [[v,[1,[1,sg]]],
                       singe]]],
        TERMS   = [ich,singe];

        NODE_IN = [[s,1],
                    [[[np,[[2,sg],nom]],[[pron,[[2,sg],nom]],
                       du]],
                     [[v,[1,[2,sg]]],
                       singst]]],
        TERMS   = [du,singst];

        NODE_IN = [[s,1],
                    [[[np,[[3,sg],nom]],[[propn,[[3,sg],nom]],
                       inge]],
                     [[v,[1,[3,sg]]],
                       singt]]],
        TERMS   = [inge,singt];

        NODE_IN = [[s,2],
                    [[[np,[[1,sg],nom]],[[pron,[[1,sg],nom]],
                       ich]],
                     [[v,[2,[1,sg]]],
                       kuesse],
                     [[np,[[1,sg],akk]],[[pron,[[1,sg],akk]],
                       mich]]]],
        TERMS   = [ich,kuesse,mich];

        NODE_IN = [[s,2],
                    [[[np,[[1,sg],nom]],[[pron,[[1,sg],nom]],
                       ich]],
                     [[v,[2,[1,sg]]],
                       kuesse],
                     [[np,[[2,sg],akk]],[[pron,[[2,sg],akk]],
                       dich]]]],
        TERMS   = [ich,kuesse,dich];
```

```
NODE_IN = [[s,2],
             [[[np,[[1,sg],nom]],[[pron,[[1,sg],nom]],
                ich]],
              [[v,[2,[1,sg]]],
                kuesse],
              [[np,[[3,sg],akk]],[[propn,[[3,sg],akk]],
                inge]]]],
TERMS   = [ich,kuesse,inge];
NODE_IN = [[s,2],
             [[[np,[[2,sg],nom]],[[pron,[[2,sg],nom]],
                du]],
              [[v,[2,[2,sg]]],
                kuesst],
              [[np,[[1,sg],akk]],[[pron,[[1,sg],akk]],
                mich]]]],
TERMS   = [du,kuesst,mich];

NODE_IN = [[s,2],
             [[[np,[[2,sg],nom]],[[pron,[[2,sg],nom]],
                du]],
              [[v,[2,[2,sg]]],
                kuesst],
              [[np,[[2,sg],akk]],[[pron,[[2,sg],akk]],
                dich]]]],
TERMS   = [du,kuesst,dich];

NODE_IN = [[s,2],
             [[[np,[[2,sg],nom]],[[pron,[[2,sg],nom]],
                du]],
              [[v,[2,[2,sg]]],
                kuesst],
              [[np,[[3,sg],akk]],[[propn,[[3,sg],akk]],
                inge]]]],
TERMS   = [du,kuesst,inge];

NODE_IN = [[s,2],
             [[[np,[[3,sg],nom]],[[propn,[[3,sg],nom]],
                inge]],
              [[v,[2,[3,sg]]],
                kuesst],
              [[np,[[1,sg],akk]],[[pron,[[1,sg],akk]],
                mich]]]],
TERMS   = [inge,kuesst,mich];

NODE_IN = [[s,2],
             [[[np,[[3,sg],nom]],[[propn,[[3,sg],nom]],
                inge]],
              [[v,[2,[3,sg]]],
                kuesst],
              [[np,[[2,sg],akk]],[[pron,[[2,sg],akk]],
                dich]]]],
TERMS   = [inge,kuesst,dich];
```

```
NODE_IN = [[v,[2,[1,sg]]],kuesse],
TERMS   = kuesse;

NODE_IN = [[v,[2,[2,sg]]],kuesst],
TERMS   = kuesst;

NODE_IN = [[v,[2,[3,sg]]],kuesst],
TERMS   = kuesst;

no
```

(20) では、変項 「NODE_IN」に構成要素全体の持っている情報がリストの形で代入されるのだ、ということが出力されている。例えば、第一番目の解、つまり、 (21) を見ると、リストの第一要素は 「[s,1]」というリストであり、このリストで表される構成要素のカテゴリーが文であることが、リストの第一要素 「s」によって示されている。このリストの第二要素 「1」は、この文が一項述語による文であることを示している。文全体を表わすリストの第二要素は文本体を表わしているが、それを抜き出せば、(21a) だ。このリストでは、この構成要素＝文に支配される二つの構成要素、つまり、名詞句と動詞、がリストでくくられている。 (21a) のリストの第一、第二構成要素は、やはりそれぞれリストである。第一構成要素は、名詞句の本体は代名詞であり、さらにこの代名詞の本体は 「ich」であることを示している。この第一構成要素を特徴づけている成分からは、カテゴリーの他に、一人称、単数、主格であること、などの情報が読み取られる。第二構成要素を表わすリストからは、この構成要素が動詞であり、一項動詞、つまり、自動詞であること、一人称の単数形であること、そして、終端シンボルの語彙として「singe」が代入されたことが読み取られる。

```
21)    NODE_IN = [[s,1],
                  [[[np,[[1,sg],nom]],[[pron,[[1,sg],nom]],
                     ich]],
                   [[v,[1,[1,sg]]],
                     singe]]],

21a)   [[[np,[[1,sg],nom]],[[pron,[[1,sg],nom]],
         ich]],
        [[v,[1,[1,sg]]],
          singe]]]
```

述語「formate」の第二番目の変項「TERMS」についての解は、 (22) にみられるように、構成要素全体の持っている情報が展開された結果が示されている。

```
22)    TERMS   = [ich,singe]
```

このプログラムでは、「?-formate(NODE_IN, NODE_OUT).」と入力すると、文ばかりではなく、それぞれのカテゴリーの分析がなされるようになっている。出力を見ると、例えば、 「ich」という終端記号が二重に出されているが、これはその構造記述をみれば、名詞句としての出力と、人称代名詞としての出力が、それぞれ別に出されているということがわかる。このことは、また、固有名詞の出力についても同様になっている。

「formate」 という述語の第一項には文の構成要素の構造記述、第二項には終端記号が代入されるようになっている。 Prolog のプログラムではよく見られることだが、このプロクラムでは、この第一項に構造記述の、例えば「[[pron,[[1,sg],akk]],mich]」を入れると、 (22) にみられるように、その終端記号を返してくる。また、 (22) にみられるように、この第一項に終端記号の,例えば「[inge,kuesst,dich]」を入れると、(23) にみられるように、その構造記述を返してくる。Prolog が、与えられた結果に行きつくまで計算を繰り返した結果を最初に指示された変項に戻してくるので、このよ

うなことが可能になる。

```
22)     ?-formate(NODE_IN,[inge,kuesst,dich]).

        NODE_IN = [[s,2],
                   [[[np,[[3,sg],nom]],[[propn,[[3,sg],nom]],
                      inge]],
                    [[v,[2,[3,sg]]],
                      kuesst],
                    [[np,[[2,sg],akk]],[[pron,[[2,sg],akk]],
                      dich]]]];

        no

23)     ?-formate([[np,[[1,sg],akk]],
                   [[pron,[[1,sg],akk]],mich]],TERMS).

        TERMS   = mich;

        no
```

3.3. 再びプログラムからことばの問題へ

3.3.1. 動詞句は必要か

ここでの (c) のプログラムを組むまでのプロセスでは、動詞句というカテゴリーを立てなかった。他動詞を二項述語として扱ったので、この程度の言語断片を扱うためには、動詞句というカテゴリーを立てなくてもよかった。しかし、たとえば、モンタギューのPTQ におけるように、他動詞を、目的格名詞句と結合して自動詞となるカテゴリーだと考えれば、動詞句というカテゴリーが必要になる。このことは、単に工学的な意味でプログラムを組むという場合と違って、単なる技術上の問題ではなく、ことばの問題として重要な問題だ。具体的には、ドイツ語の場合、動詞と目的語名詞句との結び付きは、主格名詞句との結び付きよりも強いのか、という考察が結論を左右する。私は、ここでの議論の範囲では、動詞と目的語名詞句との結び付きの強力さをしめす根拠は稀薄であると思う。

ただし、プログラム (c) のあつかう言語断片にさらに、文副詞ではない副詞を持ち込むと、議論は変わってくる。動詞と文副詞ではない副詞との結び付きは非常に強力だから、その結び付きを動詞句というカテゴリーであつかうことは、むしろ自然といえるかもしれない。

結局のところ、プログラムの中にどのようなカテゴリーを組み込むべきなのかという議論は、そのプログラムがあつかう言語断片の質と量の問題にかかわっている。こういう議論は、当然だけれども、プログラムそのものの議論としては解決のしようがない。しかし、それでは、こういう問題をことばの議論としてどのように考えてゆくのか、という問いかけは、実は言語研究の議論の基盤にいつも存在しているといえる。

3.3.2. リスト表現・変項による情報の受け渡しと、文法の一般性の問題

(A) や (B) のようなプログラムでは、かなり単純な、古典的な句構造文法が Prolog のプログラムに組んである。このプログラムでは、文のそれぞれの構成要素にどのような情報が付与されているべきか、さらにその情報が、他の構成要素の担っている情報とどのように比較され、移しかえられるのか、ということは考慮されていない。

それに対して、 (C) のプログラムでは、それぞれの文の構成要素は、必要な情報を担っている。そして、その構成要素情報がもう一つの構成要素に引き渡される場合には、変数の引き渡しによって、情報がそれらの構成要素を結び付けている接点までさかのぼった後に、もう一つの構成要素まで降りてゆくことになっている。文の構成要素間の情報の引き渡しは、このような手続きを踏まなければ、少なくともこの Prolog のシステムでは実現しない。大脳の言語中枢がどのように言語解析をするのかはわからないが、計算可能性を科学の基準の一つとするならば、構成要素間の情報の情報のやり取りがシステム的にはどのように行われるのかということは、言語研究にとっても、おさえておかなくてはならないことだろう。

(C) のプログラムでは、リスト表現を用いてそれぞれの文の構成要素について記述している。それぞれの文の構成要素は、階層的な構造を持ったリスト表現をなしていて、その、リスト表現に示されている情報が、他の構成要素の情報と比較・受け渡しがされるようになっている。このような文法記述は、最近の言語理論である LFG (= Lexical Functional Grammar) 、つまり、「語彙機能文法」や、GPSG (Generalized Phrase Structure Grammar) 、つまり、「一般化句構造文法」の記述と似通ったものだ。このことは、上記の言語理論が LISP や Prolog で記述され得る、計算機にのることを前提として考えられた理論であることを考えてみれば、当たり前と言えば当たり前のことではある。計算機にのる形での文法の整理の仕方というのは、このような形で、文法形式を整理してゆくことになると思われるが、その結果としては、構造的な面での自然言語の基底をハッキリと映し出すことになっている。

古典的な変形文法では、それぞれの構成要素について適用される文法の規則というのは、それぞれの規則ごとに記述されていた。しかし、LFG や、 GPSG などのような、最近の言語理論では、それらの規則はむしろ一般的な性格を持っているのだ、ということが説得力を持つ形で主張されている。このことは、自然言語の記述をできるだけ一般的な形式において行なおうとする現代の言語理論の考え方にとって、大切な問題を構成している。 (c) のプログラムでは、それぞれの規則が、おおきな枠組では共通の形式をもっていて、パラメーターの種類や、変項の引き渡し方という所で結果 (作用) が異なるように工夫されている。このような工夫については、プログラムを組む場合には、むしろ、プログラム上の問題として考えている。しかし、このような簡単なパーザのプログラムを組んでみる例でも、プログラム上の何らかの工夫をほどこせば、その問題は、規則の一般化を始めとして、言語理論がさまざまな形で提示している主張や仮説とからんでくることがよくわかる。

3.3.3. 計算機上の自然言語処理の問題が、人間の自然言語処理の問題と関係を持つことができるか

ここで取り組んだパーザの仕掛けは、処理の方向の縦・横からいえば縦方向処理、上・下の処理の向きという点では、トップダウン型といわれる。縦方向処理のパーザは、トップダウン型でも、ボトムアップ型でも、再帰型の規則を組み込むと、無限ループに入ってしまって、出てこなくなる。この点、横方向処理のパーザを組めば、無限ループの問題は回避できる。横方向処理では、可能な規則の適用をすべて平行処理する。だから、ループに入ってしまっても、そのループを繰り返す処理と、そこから出す処理を平行しておこなう。だから、処理全体としては、平行している処理の、どこか無限ループをまぬがれた処理がおわった段階で、終了することができる。しかし、この処理は、可能な処理をすべて平行して実行し、覚えて置くのだから、縦方向処理とはくらべものにならない記憶容量を必要とする。

計算機上に効率のよいプログラムを組もうとする場合、記憶容量はできるだけ節約したいだろう。縦方向処理のプログラムを組む場合でも、なんとか工夫をこらせば、無限ループの問題をかわすことができないわけではない。私は、たとえば、それぞれの再帰型の規則の適用回数を制限する工夫をしたことがあるが、計算機の技術者やプログラマーは、問題を処理するために、さまざまな工夫をこらすものらしい。

4. まとめ

以上、簡単な構文解析プログラムを組むことについてではなく、そこから「言語研究にとっての様々な問題が映し出される」、その可能性について述べてきた。だから、すでに述べたように、計算機による構文解析や、ドイツ語という言語についての、新しい事実の提示などの貢献はまるでない。実用や、写像主義的な科学をめざさなくても、または、そういうことをめざさないからこそ、計算機という思考装置を用いた考察は、以外と、対象の様々な面を照らしだしてくれることもあるのだと思う。

ダ・ヴィンチやゲーテの時代であっても、やはり天才でなければできなかったことではあっただろうが、現代において、学際的な関心を持つということは、大変なことであるようだ。世界大戦を経験し、核兵器や環境汚染の問題を始め、さまざまなグローバルな問題をかかえた人類にとって、科学は、決して閉じられた営為にしてはならない領域である。このような理念的な要請をかかえながら、しかし、現代における個別の科学領域は限りなく細分化し、専門化している。「学際的」という概念は、現代において、とんでもない矛盾をかかえているようにも思える。

そのような意味では、本当は、私のような、様々な科学の境界領域に興味をもってしまった研究者は、中途半端な困った研究者であって、計算機という対象そのものにのめり込むことは、そのような研究者にとっては、手に負えない、また危険な仕事にちがいない。しかし、逆の視点を取ってみれば、計算機というのは、人間に仕え、人間の仕事を助けるために考案された道具にすぎないし、機構そのものも、並列処理などということをかんがえてみても、最終的には、二進法の最も単純な演算が巨大規模に集積されたということにすぎないのだろう。自分が機械そのものの内部の問題に立ち入ったり、または、機械の下僕にはならないということさえ気をつけていれば、自分のばく然とした考えなどをチェックしたり、または新しい構想を生み出す手掛かり、足掛かり、または鏡やスケッチブックとして利用することに徹すれば、計算機に足をすくわれることは避けられるのではないだろうか。また、全能的な知識の持ち主としてではなく、自分の主領域以外に目配りをすることができる、というような意味合いでの、「学際性」というものの考え方は可能なのではないだろうか。ただし、その場合、それぞれの科学の領域について、他領域からの接近の可能性が保証されているということが、大前提とされるだろう。

例えば、ここでのプログラムでは、変項への値の引き渡しにはすべて、「=」の操作をつかった。これは、Prolog のもっと上手な表現の仕方は、値を引き渡す項と、引き渡される項を同じ表現にしてやれば済むことであって、そうすれば、プログラムの行数は非常に少なくできただろう。例えば、(C) の一番目の規則は全部で六行からなっているが、(24) に比較したようにニ行で書き表わせてしまう。きっと、この方が、実行速度も早いのだろう。しかし、ちょっと頭の固い、私のような人間には、(C) で書き表わしたような書き方がすごくわかりやすい。古典的な変形文法風の表記になれているということにしか過ぎないのだろうが、私にとっては、冗長でも (C) のような表記がわかりやすく、また、私の目的のためには、これでこと足りる。

```
22)     formate(NODE_IN,TERMS) :-
                NODE_IN=[SPECIFICS_0,NODE_OUT],
                SPECIFICS_0=[CAT,SPECIFICS_1],
                CAT=s,
                NODE=[SPECIFICS_1,NODE_OUT],
                s(NODE,TERMS).

24)     formate([[s,SPECIFICS_1],NODE_OUT],TERMS) :-
                s([SPECIFICS_1,NODE_OUT],TERMS).
```

こういう言い方をすると、正面から計算機に取り組んで、実証や実用をめざしている人たちには叱られそうなのだけれど、私は、計算機の利用の仕方として、こういう「研究

文献 （「ABC」と「アイウエオ」順）：

Kaplan, R.M. & Bresnan, J. 1982: Lexical-Functional Grammar: A Formal System for Grammatical Representation. In: The Mental Representation of Grammatical Relations. Ed. by Joan Bresnan. The MIT Press, Cambridge Mass. 1982

Kay, Martin 1985: Parsing in functional unification grammar. In: Natural language parsing, Psychological, computational, and theoretical perspectives., ed. by David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky.

Montague, R. 1970: The proper treatment of quantification in ordinary English In R. Thomason(Ed.), Formal Philosophy, Selected Papers of Richard Montague. New Haven: Yale Univ Press, 1974. Pp. 247-270

石川　彰 1986: 言語理論の新しい動向。古川、溝口編、「自然言語の基礎理論」所収、共立出版、pp. 1-50

郡司隆男 1987: 「自然言語の文法理論」、産業図書

後藤滋樹 1984: 「PROLOG 入門」、サイエンス社

田中穂積、元吉文男、山梨正明 1983: 「LISP で学ぶ認知心理学 3、言語理解」、東京大学出版会

テナント、ハリー 1984: 「自然言語処理入門」、産業図書 (Harry Tennant 1981: Natural Language Processing. Petrocelli Books.)

長尾　真 1986: 「機械翻訳はどこまで可能か」、岩波書店

中島秀之 1983: 「Prolog」、産業図書

萩野達也、桜川貴司、柴山悦哉 1984: 「Prolog-KABA Reference Manual」、岩崎技研工業

萩野達也、桜川貴司、柴山悦哉 1986: 「Prolog-KABA 入門」、岩波書店