No.

Date

プログラミング言語学をめざして

―― プログラミング言語の言語学的な見方

指導教官　和田英一 教授

1981年 2月

金田　泰

プログラミング言語を、機械のための言語というより、人間がかき、人間がよむための言語と認識すれば、その研究はむしろ人文科学に属するものであることがわかる。そして、そこからプログラミング言語を言語学的に研究する可能性がひらけてくる。そして、おなじ認識からプログラミング言語と自然言語を比較研究することの意義がみいだされる。

これまでプログラミング言語の言語学的研究は、興味はもたれていても実際におこなわれたことはないようである。したがって、まずその研究の基礎をきずくことが必要だとおもわれた。そこで、プログラミング言語の言語学的な見方をしめし、プログラミング言語のどの部分にどのような研究方法が適用可能であるかをしらべ、そして研究を方向づけることをこころみた。

本論文でのべる「言語学的な見方」のなかでもとくに重要なのは、まずプログラミング言語を慣習(=「非成文化規則」)をもふくめた体系としてとらえること、つぎにプログラム単位名の意味とその「抽象」との関係としてみることである。またプログラミング言語に言語学的方法をあてはめるための検討のなかでは、自然言語との構造上の類似点などを指摘した。そして、「形態論」から「意味論」におよぶ研究分野をいちおう方向づけした。そのなかで重要な(意味論の)部門のひとつは、プログラミング言語に存在するあいまい性の研究である。

言語学的研究はおもにすでに存在するプログラムを対象とするが、本論文ではそのような系統的研究をおこなうところまでは達していない。しかし、ここから研究のてがかりをえることはできるのではないかとおもう。

目次

No

Date

第1章　はじめに

プログラミング言語（以下「算語」と略する）にかかわる これまでの議論は、機械や処理系が中心であるか、まったくは抽象的な理論でおわってしまうか、さもなければ非科学的な議論におちくれていて、ほんとうに それを かく人間のことを かんがえた 科学的な議論が すくなかったようにおもわれる。たとえば算語の設計をするようなときにも、設計上の選択は 趣味的 あるいは 無意識的におこなわれるか、ある選択枝が たしかな理由もなしに はじめから きめこまれているかの いずれかであることが おおかったように おもわれる。また、プログラミング方法論についても、そのおおくは経験的にうちたてられた科学的根拠の とぼしいものであるか、あるいは理論的な必要性、たとえば検証可能性にもとづいたものであって、プログラマのことをかんがえていないかの いずれかである。

プログラミングを人間のいとなみとしてとらえることの重要性を指摘したひとは 何人かいる。そのうちの ひとり Weinberg は、"The psychology of computer programming"[1971] なる大著を かいている。この本は当時 かなりの反響をまきおこした。しかし、それにもかかわらず そのような分野の研究は それほど さかんでもないし、あまり成果をあげても

いないようである(この点についてはS.1でもうすこしくわしくみる)

従来、このように算語やプログラミングにたいするhumanisticな議論があまりおこなわれなかったおおきな理由は、計算機科学者が人間がかかわる問題の科学的なあつかいかたをよくこころえていなかったためではないだろうか。そのような問題は人文科学や社会科学があつかってきたものであり、したがってわれわれはそれらからまなぶべきである。

これから問題にしようとしているのは、そのようなこころみのひとつで、算語のhumanisticな理論、それも言語学的な理論をうちたてることである。この分野はおおくのひとがばくぜんとめをつけているにもかかわらず、実際にはほとんど研究がはじめられていなかった。はじめられなかったおおきな理由のひとつが、その具体的な着眼点をみいだすこともふくめての研究のむずかしさであることはうたがいがない。したがって、いまきゅうにそれを確立することはできないが、この論文がそのような研究のひとつの方向をしめすことはできるのではないかとおもう。

本章では、まず算語の研究にはどのような分野があるか、とくに

humanistic な研究として どのようなことができるかをみて、それから そのなかのひとつとしての言語学的研究、それもとくに自然言語（以下「人語」と略す）と算語との対照研究にどのような意義があるかについてのべることにする。第2章では言語学的あるいは記号学的な研究方法をあてはめるために必要な条件を算語がそなえているかどうか、また算語のどの部分がそのような条件をもっているかをしらべる。そして、その結果にもとづき、第3章では言語学的研究の分野をいくつかさだめ、そのうちの2分野について第4章および第5章でのべることにする。

No

Date

1.1. プログラミング言語にかかわる研究の分野

人語（自然言語）にかかわる科学的研究には、哲学的研究、心理学的研究，言語学的研究がある（このほかに工学的研究として人工知能をあげることもできよう）が、算語（プログラミング言語）においてもこの3つがあげられるだろう.

A. 哲学的研究

人語における哲学的研究としては、言語哲学，論理学などがある. これまでのプログラム[言語]理論のおおく、たとえば入計算，形式言語学，Turing Machineの理論などは論理学とおなじような演繹的体系であり、ここに分類することができよう.

論理学は哲学の一分野として人文科学にふくまれるものとされているものの、あまり人間くささのない学問であるが、これらのプログラム[言語]理論も、どうすれば"プログラムが人間にとってよみやすく、かきやすくなるかというような問にはこたえてくれない.

B. 心理学的研究

すでにふれたWeinberg[1971]は「心理学」を書名にふくんでいたが、ほかにこの分野をめざしたものとしてShneiderman[1980]がある。Shneiderman[1980]の巻末には、おおくのプログラミングにかんする心理学的研究の論文があげられている。

Weinberg[1971]はプログラミングを人間のいとなみとしてとらえることによって、おおくの、ときには予期できない成果がえられるとのべている。そして、そのような観点からプログラミングにかんするおおくの問題を分析しているが、それは実証的な研究ではなく、むしろプログラマやマネジャを読者として想定しており、研究者にたいしては研究のテーマをあたえたものといえる。算語にかんしても、人間中心のとらえかたをして、人語と対照したり[p.206-210]、言語設計についてかんがえたり[p.210-245]している。「心理学」と題してはいるが、言語学的研究のテーマとなりうることもふくんでいる。

Shneiderman [1980] はより実証的な研究をめざしている. 算語については、注釈のいれかた、変数名のつけかた、段づけ(indentation)や制御構造がプログラムの理解にあたえる影響などについての心理学的実験の結果をしめしているが、あまり意味のある結果はえられていない.

C. 言語学的研究

算語が言語であるならば 言語学的な方法を適用できるのではないか、というかんがえが おこってくる。ただし、算語が言語とよべるものであるかどうか。すなわち言語学的な研究の対象がもつべき性質をもっているかどうかはしらべてみなければならない. そして、言語学の方法を適用する場合に 算語独自の事情に配慮すべきであることはいうまでもない.

言語学的な研究の中心的な方法は帰納的なものであって、すでに存在するプログラムの分析ということになるだろう. とはいっても 算語の理論は「まだ存在しない言語」をあつかいうるようになっていることが のぞましい. すなわち、

人語の場合とちがって、算語の言語学的研究は工学との関係が密接であり、その成果を算語の設計やプログラミング方法論に反映させることがひとつの目標となる。

だが、算語は規定書（文法書）で定義されている。言語学的な研究によってはじめてあきらかにされるようなことがあるのだろうか。すなわち、その存在意義はあるのだろうか。この問に真にこたえるためには実際に言語学的研究をこころみてみなければならないが、現時点でいえることは、つぎの2点である。

第1に、従来の議論（算語一般にかんする哲学的研究や個々の算語についての文法書上の問題にかんする議論）からはみだしているが、言語学的方法ではあつかいうる部分があるとかんがえられること。そのなかには、これまで「スタイル（style）」とよばれていたものにかかわる問題、人語からの語などの借用の問題、算語は人間の言語能力［たとえばChomsky［1975］］を反映しているかという問題など、さまざまの問題がある。これらの問題じたいと、

それが言語学的にあつかいうること（すなわち、それをあつかいうる言語学的視点があること）については第2章でのべる。

第2に、人語と算語の対照は意義があり、言語学的方法（あるいは記号学的方法）はそのための有力な方法とおもわれること。この点については §1.2 でくわしくのべる。

1.2. プログラミング言語と自然言語の対照研究の意義

本節では算語（プログラミング言語）の言語学的研究の一分野となるべき、「算語と人語（自然言語）の対照研究（contrastive study）」にどのような意義があるかについてかんがえてみよう。

算語と人語のちがいは、その目的のちがいによって正当化されるだろうか。それとも算語がもともと機械中心の言語だったという歴史的事情をおおきく反映しているのだろうか。Chomsky［たとえば［1975］］はほとんどすべての人間がそれほど訓練もせずに言語をはなせるようになるのは、人間がうまれつきの言語能力をもっているからだといっているが、人語がそのような人間のうまれつきの能力をよくいかしているのにたいして、算語がそれに反していることがプログラミングをむずかしくしているのではないだろうか。

算語と人語の対照研究の意義をのべている人は何人かいるが、たとえば高橋秀俊［1978］はつぎのようにかいている。

『いろいろの言語をしらべ、比較することが言語の研究に重要であるのと同様に、プログラム言語と自然言語とを比較してその対応関係をしらべ、一方にあって他方に欠けているものがあれば、なぜ欠けているのか、それを補うことによってその言語を改良する

ことはできないか、などのことを考えてみるのは有益なことだろう。』

ほかには藤村 靖[1963, p.15]などが意義をのべているし、公式にのべてはいなくても、対照研究に興味をもっているひとはおおい。

人語は長い歴史をもっており、算語からまなびうる点があまりあるとはおもわれないし、それはいまかんがえようとしている問題ではない。算語の側からすると、対照研究の意義は、対照によって相違点と相違している理由をあきらかにし、また類似点と類似している理由をあきらかにすることによって、算語の発展に貢献することができるとかんがえられるところにある。これをひとことでいえば、「算語はどこまで人語をまねるべきか」をあきらかにすることである。

この問題について極端な意見としてつぎの3つがかんがえられる。

A. プログラムは人語でかくべきだ(算語＝人語とするべきだ、あるいはプログラミングのための人語の部分集合をつくるべきだ)。

B. プログラムは述語論理（あるいはそれを拡張したもの）でかくべきだ。（これは「哲学的研究」の研究者がおちいりやすいかんがえだとおもわれる）。

C. プログラムは機械語でかくべきだ。（今日でも効率のために、高級言語でかけるところをわざわざ低級言語でかくことがある。高級言語といわれる算語のおおくも、効率をあげるために人間にかなりの犠牲をしいている。したがって、C.の意見にもっとも極端にあらわされている視点を、われわれは無視するわけにはいかないのである）。

「算語はどこまで人語をまねるべきか」にかんしてよりこまかくかんがえてみると、つぎのような問題がうかびあがってくる。

A. 人語の柔軟さを算語にとりいれるべきか。とりいれるべきだとすると、どうすればよいか。

B. 算語はどのくらい複雑であってよいか、またあるべきか。この問は「複雑さ」が表現力というような言語ののぞましい性質のうら

No

Date

がえしになっているとかんがえられるゆえに生じてくる。(それらの性質と複雑さとの関係は言語学や算語の言語学的研究によってあきらかにされるべきものである)。複雑さには翻訳系(compilers)にとっての処理の複雑さと、人間にとっての習得困難度とがあって、両者はおなじものさしでははかれないであろう。

C. 人語のあいまい性をどこまで算語にとりいれるべきか。

これまで算語にはあいまい性がないといわれてきた。これがあやまりであることは§2.6.1でしめすが、とくに、最近設計された算語Ada[DoD[1980]]においては識別子(identifier)の多義性が積極的にとりいれられていることをかんがえると、そしてより一般に柔軟性があいまい性と密接な関係にあるとかんがえられることから、この問題が重要なものであることがわかる。

以上のような問にこたえるために、算語と人語との対照が意義ぶかいとおもわれるのだが、それがなりたつためには、算語と人語に対照を可能にするような共通点がなければならない。それが存在することについては §2.3 でのべる。

第2章 プログラミング言語とはどういうものか

本章では算語が言語学的方法をあてはめることができる性質をもっているかどうか、これまでの方法ではあつかえなかったが言語学的方法ならばあつかえるのはどういう問題で、それはどのようにすればあつかえるのか、また算語と人語とを比較する場合にどこを比較することができるか、といった問にこたえるために、様々の角度から算語とはどういうものかをしらべることにする。

§2.1. では算語には3つの表現 (representation) があることについてのべ、そのうちどの表現を研究の対象とするべきかをかんがえよう。

§2.2では、これまでふつうにかんがえられてきた意味での「算語」とは（おそらく）すこしちがう、言語学的研究に役だつとおもわれる「算語」のとらえかた、とくに算語の規則のとらえかたをしめし、そうやってとらえた規則についてかんがえる。この、規則のとらえかたは、本章のなかでも、したがって言語学的研究のために、もっとも重要なことのひとつだとおもわれる。

§2.3では算語が人語から自立した体系であるかどうかを検討するが、それにさきだって、自立的であるかどうかが研究のうえでどういう意味をもっているかについてかんがえる。

§2.4.では算語の「意味」の言語学的な(というよりは記号学的な)とらえかたについてのべる。このとらえかたは、§2.1.における「規則」のとらえかたとならんで言語学的研究にとって重要なものだとおもわれる。

§2.5.では人語の特性といわれていることを列挙し、そのそれぞれが算語にもあるかどうかをしらべる。この節の目的は、算語がこれまでふつうにかんがえられてきたより人語に似ていると

いうことをしめすとともに、言語学的方法の適用可能性をしめし、また算語のどの部分をどのような方法で研究できるかという問題にこたえるための手がかりをえることにある。

§2.6.では§2.5.でとりあげなかった算語の性質について論じる。そのなかには人語と算語の重要なちがい、これまで人語と算語の示差的（distinctive）なちがいといわれてきたが実際はそうではない点などがふくまれる。

§2.7.では§2.5.で指摘する算語の非言語的部分についてかんがえる。

また、§2.8.では§2.6.でふれる算語の変化（variety and change）についてのべる。

本章でのべることは、言語学的方法の適用可能性や算語と人語の対照の可能性をしめすとともに、その限界をもしめすものである。われわれはその限界をこえないように注意しなければならない。

2.1. プログラミング言語の3つの表現

算語の分析をはじめるまえに、この論文で問題にする「算語」というのがどの表現(representation)かということについてのべる必要があるだろう。Nauer [1963] によると算語には3つの表現がある。[以下の引用は 米田,野下 [1976] の訳による。Nauer [1963] では「表現」のかわりに「レベル」ということばが使われている。]

A. 規準言語 (reference language)

規準言語は 規定書 (通常「文法書」とよばれる) の『定義の対象となる言語』であり、ひとつの予約語や特殊記号は全体でひとつの分節されない記号とされている。

B. 発表言語 (publication language)

『発表言語は、印刷や手書きの慣習 (例えば、添字, 空白, 指数, ギリシャ文字) に応じて規準言語の変更を許したものである』.

C. 金物表現 (hardware representation)

『金物表現はそれぞれ, 標準的な入力装置に備わっている文字数に関するやむをえない制限によって、規準言語を圧縮したものである』.

したがって、発表言語や金物表現では ひとつの予約語や特殊記号が2文字以上であらわされることがあり、また英字が大文字にかぎられることもあるわけである.

FortranやCobolでは金物表現だけがさだめられているが、Pascal, Adaなど今日のおおくの算語では規準言語または発表言語と金物表現とが区別されている.(規準言語と発表言語との区別はかならずしも明確でない). そして、それらのちがいは、いずれを対象とするかによって若干研究の結果に影響をおよぼしうる.(それゆえ、どの表現を問題にするかを検討するのである).

さて、この論文で問題にするのがどの表現かということが問題だったが、それは画一的にはきめられない. たとえば"Algol 60 [Nauer [1963]]は人間へのアルゴリズムの伝達だけをめざして規準言語または発表言語の表現でかかれ、よまれることがおおい. 一方Pascalは規準言語または発表言語の表現でかかれることがおおいが、金物表現でかかれることもおおく、よむ場合には金物表現でよまれることがおおい.

われわれが目標としているのは算語のhumanisticな理論なのであるから、その対象は当然人間があつかう表現であり、それもパンチャなどでなくプログラマがあつかう表現ということになる。したがって、それぞれの状況に応じて3つの表現のうちのいずれかを対象としなければならないということになる。

2.2. プログラミング言語の規則

2.2.1. 成文化規則と非成文化規則 1

IFIP-ICC [1966] では、算語をふくむ「言語」について つぎのようにいっている. [訳は Sammet [1969] の訳による].

「言語」は『記号ならびに記号を連結して、意味のあるコミュニケーションにするときの様式と順序に適用される規則」または慣習の定義された集合に対する一般的な用語』である.

これを算語にそのままあてはめるならば、算語は「構造」、すなわち単位(引用文中では『記号』)とそれをくみあわせる規則(文中では『規則または慣習』)とからなるものとしてとらえられるとともに、規則とならんで慣習が算語の一部としてみとめられることになる.

すなわち、プログラムを支配しているのは規定書にかかれた規則だけではなく、慣習もまたそれを支配しているのである. 後者は紙の上に記述されてはいないにしても、やはり規則とよびうるものだとすれば、「非成文化規則(uncodified rules)」(あるいは「慣習規則(customary rules)」)とよぶことができるだろう. これにたいし、前者は「成文化規則(codified rules)」(あるいは「成文規則(written rules)」とよぶことができよう.

規定書の規則をあつかう演繹的理論はすでにかなり発展しているが、非成文化規則をあつかう帰納的理論はまだないといえよう。言語学的方法が、それをあつかうのに適切な方法だとおもわれる。

ひとつの算語が 記号と成文化規則と非成文化規則とからなるものだとすれば、ふつうにはひとつの言語とかんがえられているFortran IVとかStandard Pascal [Jensen, Wirth [1975]] とかいうものは、非成文化規則がことなるいくつかの方言からなる言語族ととらえることができるであろう。(たんにFortranとかPascalとかいった場合には、前者はFortran II, Fortran IV, Fortran77などをふくむし、後者はOriginal Pascal [Wirth [1971]], Standard Pascal, ISO Pascal [Addyman [1980]] などをふくむことになるから。それらが言語族の名称であることはあきらかである。)しかも、人語にくらべるとよりこまかい多数の方言からなっているとおもわれる。

2.2.2. 「非成文化規則」は存在するか

実際に「非成文化規則」とよびうるようなものが存在するかどうかは、具体的にしらべてみてあきらかにされることである。ここでは仮説としてそのようなものの存在をみとめることにするが、存在するとかんがえる根拠はあげることができる。

もし算語であるアルゴリズムを表現するのに、規定書にしたがったかきかたがひとおりしかないとすれば、非成文化規則などというものは存在しえない。しかし、現実にはひとつのアルゴリズムを表現するのに、様々のレベルでかなりはばひろい選択の余地がある。それらの選択枝のうちいずれをとるかはけっしてでたらめにきめられるわけではない。とすればそこには選択をおこなっている規則があるはずである。それが非成文化規則である。以下ではいまのべたことの具体例をあげることにしよう。

従来「スタイル」とよばれてきたもののおおくが非成文化規則のあらわれだとかんがえられる。[ここではいわゆる「スタイル」訳せば「文体」ということばを言語学でいうつう「文体」とよんでいるものとはややちがったものとしてとらえようとしているのである]。たとえば、われわれがひとのかいたプログラムをその仕様書なしにある程度でも

No

Date

よみうるのは、プログラムにあらわれる識別子として しかるべき意味を
もったものがつけられているためだとおもわれる. たとえば

```
procedure INSERT (ITEM, FREE, POSITION);
begin
  NAME [FREE] := ITEM;
  NEXT[FREE] := NEXT[POSITION];
  NEXT[POSITION] := FREE;
end;
```

のような手続き [Aho, Hopcroft, Ullman [1974]] をみて、(その本文中の
説明はみないでも) なれていれば これが linked list への項目の挿入

をおこなう手続きだとわかるのは、そのなかにあらわれる識別子に「意味」のあるつづりがもちいられているからにほかならない。INSERT, ITEM, FREE などのかわりに A, B, C などがつかわれていたら、この手続きの機能は けっして わからなかったであろう。これは、上の手続きに 規定書にはかかれていない なんらかの規則が はたらいていることをしめすものである。

べつの、規則が より明確にあらわれている例として、段づけ (indentation) をあげることができる。これも、これまで「スタイル」という名でよばれていたものの ひとつである。

```
begin
  min := a[1];
  max := min;
  i := 2;
  while i < n do begin
    u := a[i];
    v := a[i+1];
    if u > v then begin
      if u > max then max := u;
      if v < min then min := v end
    else begin
      if v > max then max := v;
      if u < min then min := u end;
    i := i + 2 end;
  if i = n then begin
    if a[n] > max then begin
      max := a[n] end
    else if a[n] < min then begin
      min := a[n] end end;
  writeln(min, max);
  writeln;
end
```

[TSINKY[1980]]

このプログラムにみられるような規則的な段づけは自由欄形式の算語では一般的にみられるものであるが、その規則はまさに非成文化規則とよぶほかはないようなものである。[TSINKY[1980]はこのような段づけ規則にどういう種類があるか、すなわちいろいろな段づけのつけかたについてのべている。ただし、規則をはっきりとりだしてみせているわけではなく、例によってしめしているだけである]。

これまで、このような非成文化規則はあまり重視されなかったようにおもわれるが、上記の例からもわかるように、これらは算語が人間への伝達手段としてうまくはたらくためには成文化規則とならんで重要である。[算語が人間への伝達手段としてつかわれることについては§2.5.2.A.を参照]。INSERTの例で、でたらめな識別子をつけることで伝達が非常に困難になることを示唆したが、段づけもまた、したがわないことによってプログラムの読解はきわめて困難になる。われわれが段づけのつけられたプログラムをよむ場合に、いかに段づけにたよっているかは、つぎのような例をみればあきらかになるだろう。

(1) if Expression1 then

if Expression2 then Statement1

else Statement2.

これをPascalあるいはPL/Iのプログラムとおもっていただきたい.

そうすると、このプログラムは つぎのプログラムと等価である.

(2) if Expression1 then

begin

if Expression2 then Statement1

else Statement2

end.

ところが、つぎのプログラムと等価であるようにみえる.

(3) if Expression1 then

begin if Expression2 then Statement1 end

else Statement2.

実際に、横文,段づけともに(1)のかたちをしたプログラムが

バグとして存在した例を、筆者は いくつも しっている.

No.

Date

このように「スタイル」(の一部 —— 伝達にかかわる重要な部分)を非成文化規則」ととらえなおすことによって、§1.2で"ふれた「スタイル」の問題は非成文化規則の問題で"おきかえられることになる。ただし、「個人的なスタイル」の問題はとりのこされてしまうが。

2.2.3. 成文化規則」と非成文化規則」2

§2.2.1で"成文化規則」と非成文化規則」の定義をいちおうのべておいたが、この定義は言語学的研究にとってかならずしも都合のよいものではない。非成文化規則」はプログラムの分析から帰納的にみちびかれるものであるのにたいして、成文化規則」は演繹的なものとしてあたえられており、両者がおなじ観点に立った対立した概念になっていないからである。そこで、つぎのような定義をこころみてみよう。(以下ではとくにことわらないかぎり、この定義を採用する)。

A. 成文化規則」とは、プログラムにみいだされる規則のうち言語規定書や処理系のマニュアルにかかれたもののことである。

B. 非成文化規則」とは、プログラムにみいだされる規則のうち言語規定書や処理系のマニュアルにかかれていないもののことである。

この操作的な定義を採用すれば、規定書にかかれている規則がすべて成文化規則とはかぎらないことになる。たとえばFortranの添字式として、規定書には【たとえばJIS[1976]には】

(i) $c*v+k$

(ii) $c*v-k$

(iii) $c*v$

(iv) $v+k$

(v) $v-k$

(vi) v

(vii) k

(cとkは整定数, vは整数形の変数の引用) だけの かたちが ゆるされているとしても、もし実際には (i),(ii),(iii) のかたちは みいだされないとすれば、これらは成文化規則にはふくまれないことになる. [なお、この問題は Weinberg [1971, p. 213, pp. 218-219] が「心理学的」に考察している].

規定書規則と成文化規則とのくいちがいの例として、ほかにつぎのようなことをあげることができる. 識別子の規定書上の構文はふつう

<letter>{<letter or digit>}

である.
({…} は0回以上のくりかえしをあらわす. <letter> は A

からZまでの英字のうちのいずれかであり、<letter or digit>はAからZまでの英字または0から9までの数字のうちのいずれかである)。したがって abc12d3e のように数字が途中にあらわれるかたちも許されているわけだが、実際には

<letter>{<letter>}{<digit>}

以外のかたちはほとんどあらわれない。また、整数のリテラルの場合、規定書上の構文は

<digit>{<digit>}

となっているのがふつうだが、実際には1のあとに数字がつづくかたちはほとんどあらわれない。

成文化規則と非成文化規則の定義はあまり厳密でないいいかたでいいかえるならば、仮想的な平均的プログラマがある算語の規則ととらえている(規則だと明確に認識している必要はない)もののうち、規定書や処理系のマニュアルにみいだされるのが成文化規則であり、みいだされないものが非成文化規則であるということになる。最初の例の場合、彼のあたまのなか

には Fortran の添字式として (i),(ii),(iii) のかたちは存在しない、それゆえに これらは成文化規則ではないのである.

しかし、ある せまい範囲では実際のプログラムに規定書のすべての規則がみいだされなかったとしても、より ひろい範囲をかんがえれば みいだされるであろう。たとえば、うえの添字式の例でも、(i),(ii),(iii) のかたちも まれには みいだせるであろう。したがって、§2.1.1. の定義と本節の定義とは それほど ちがったものではないともいえる。だが、本節の定義にしたがえば、出現頻度の ひくい構文は、おおいものに くらべると 規則とみなしにくいということが いえる。いわば、本節の定義における成文化規則は、規定書の規則に、出現頻度による おもみづけをしたものといえるであろう.

いま のべたような 成文化規則と規定書規則とのくいちがいは、言語学的研究の重要な対象となるであろう。この問題は、非成文化規則にはなにが あるかという問題とともに、規則を言語学的にみることによって みいだされた重要な問題の ひとつであり、言語学的な見方の有用性を しめすものだとおもわれる。

2.2.4. 成文化規則の細分

成文化規則は、そのいずれの定義を採用するにせよ、さらにつぎの3つに分類できる。[この節でのべることは、以下の節の論旨にそれほどかかわりがあるわけではない]。

A. 実行前に処理系により検証される(あるいは、されうる)規則。

例としては型(データ型)のある言語における型の検証があげられる。たとえば"

C: CHARACTER;

と宣言しておきながら C+1 などとかくと、これは翻訳時(compile time)にあやまりとされる。A.の規則のなかには、翻訳時には検証されないが、リンク時に検証されるものもある。たとえばFortranのプログラムで、あるサブルーティンをつかっておきながら、その宣言をあたえていなければ、それは(通常)翻訳時には検出されず、リンク時にあやまりとされる。

B. 実行時に処理系により検証される(あるいは、されうる)規則。

例としては、数の演算における けたあふれ をあげることができる。式の計算の途中で けたあふれをおこすかどうかは

(一般には) 実行時まで"わからない.

C. 処理系が検証しない規則」

翻訳時には検証できず、実行時に検証するときわめて高くつく規則は、処理系がいっさい検証をおこなわないことがあるが、これがC.の場合である. 検証できないことはさだめない方針の規定書もあるが、たとえばAdaの規定書[DoD[1980]]では処理系が検証しなくてもよい規則が多数とりさめられている.

実行前に検証されうる規則」がすべて実行前に検証されるとはかぎらないし、実行中に検証されうる規則」がすべて実行中に検証されるともかぎらないから、A., B.として「検証される」規則」をとるか、「検証されうる」規則」をとるかによって、ことなった分類ができる.

2.2.5 伝達機能による規則の分類

成文化規則，非成文化規則というわけかたとはべつに、規則をつぎのようにわけることができるだろう。

A. 人間どうしの伝達のためだけにつかわれる規則。

B. 人間どうし、および人間から機械への伝達につかわれる規則

C. 人間から機械への伝達のためだけにつかわれる規則。

[たとえば 実行効率だけに影響をあたえるようなもの]

成文化規則のほとんどはA.にふくまれるであろう。非成文化規則にはA.にふくまれるものとB.にふくまれるものとがある。(ここではC.のことはかんがえない)。これまでにとりあげた非成文化規則の例はみなA.にふくまれるものだったが、B.にふくまれるものとしては、たとえばつぎのような例があげられる(仮想的な例だが)。

Standard Pascal [Jensen, Wirth [1975]] の規定書はきわめてあいまいであるがゆえに、おおくの規則が慣習的である。たとえば、

```
type Person =
    record
      Birth : Data;
      case Sexuality : Sex of
        Male : (Bearded : Boolean);
        Female : (Children : 0..23);
    end;
```

のような例で Sexuality が Male のとき、(存在しないはずの field である) Children を参照してはならないという規則は規定書にはないし処理系もそのことじたいは検証しないが、これはまもられるべき規則である。それは、この規則をやぶると人間にとっても機械にとってもプログラムが意味をうしなうからである。(このような基本的な点を規定書でさだめていないのは、規定書の不備としかいいようがないが)。ただし、この規則は bit pattern をかえない型変換をおこなうというような特殊な目的があるときにはやぶられることがある。

いまの例は非成文化規則が成文化規則をおぎなっている例だったが、両者は矛盾することもありうる。すなわち、規定書にかかれていても処理系が検証しない規則については、それと矛盾した規則が慣習化しうる。

しかしながら、規定書にかかれている「ある規則」が非成文化規則と矛盾しているときには、その規則」が（§2.2.4の定義の）成文化規則にふくまれていることはないだろう。なぜなら、たがいに矛盾する規則がひとつの算語のなかにみいだされることはないであろうから.

2.2.6. プログラミング言語の規則」と自然言語の規則」

人語における規則」は、人語の歴史のなかで"しだいに つくられてきたものであり、あらかじめ定義されたものではない。したがって、これは算語の非成文化規則」に対応するものだといえる。人語の文法書も、こうした規則を記述したもので"あって、算語の規定書のような絶対的な強制」力をもたない。(Kernighan, Plaugen[1974]のようなスタイル・ブックは算語における、人語の文法書にたものといえる)。人語には成文化規則」に相当する規則はないのである。

この点が算語と人語との重要な相違点である。ただし、人工自然言語(?)であるエスペラントやイドなどには成文化規則」がある(非成文化規則」も当然あるだろう)。これらの人工言語は算語にた性質をもっている。

ほかに成文化規則と非成文化規則のような対立が存在する例として、法の体系があげられるだろう。(成文化規則と(成文)法そのもの。非成文化規則と判例などが対応するとおもわれる)。しかし、この論文では論じない。

2.2.7. 規則の柔軟性

成文化規則、それもとくに処理系が検証する規則に違反したプログラムは(ただしく)実行されない。これは成文化規則が"強制的"であり、柔軟性がないということを意味する。これにたいし、非成文化規則(のおおく、とくに人間どうしの伝達にだけかかわる規則)には違反しているとしてもプログラムを実行することはできる。ただし、やぶれはやぶっただけ人間への伝達効率は低下するであろう。すなわち、非成文化規則はそれに対応する人語の規則にについて、柔軟性があるといえる。

ただし、人語では規則をやぶれは(すくなくとも逸脱したもの)とみなされるが、算語では非成文化規則をやぶっても、ふつうはそのようならくいんをおされないことがおおい。§2.1.2.であげたような規則にたいする違反はすべてそうである。したがって、柔軟というよりは放縦にちかい。

放縦はこまるが、このような両規則の特徴から、算語における発展性のある部分あるいは変化のはげしい(または多様性のおおきな)部分は非成文化規則できめるのが適当だろう。

なお、成文化規則のなかでも、強制力に差があるということを

つけくわえておこう。すなわち、§2.2.4.の分類における A.[実行前に検証]はきわめて強制力がつよい。B.[実行時に検証]は強制力がある。しかし、C.[非検証]は強制的でない。ただし、強制的でないからといってやぶると処理系からしっぺがえしをくらうことになる。[C.の規則は検証が困難であるがゆえにC.にふくまれているのであるから、それをやぶることによってうまれたバグは容易にみつからないことがおおいとかんがえられる]。

2.2.8. 規則の複雑さについて

算語の規則が人語の規則にくらべて単純であることはまちがいないだろうが、それでも非成文化規則までふくめれば、ふつうにかんがえられているよりはるかに複雑な(≒規則のかずがおおい)のではないだろうか。算語は1年間もまなべばかなり自由にプログラムをかけるようになるとはいうものの、「スタイル」を確立するまでにはもっと時間がかかるであろう。

また、プログラムのよみやすさ、保守のしやすさなどのためには、すなわち人間にうまく伝達をおこなうためには、規則のおおさという意味での複雑さはむしろ必要なのではないだろうか。

しかし一方、規定書はきわめて特殊な場合まできちんとあつかわなければならないので、こまかい規則のために長大なものとなりうるが、それにもかかわらず成文化規則の体系は、それらの特殊な「規則」が(重要な)規則とみなされないことによって、単純だといいうることがあるだろう。すなわち、「単純さ」とか「複雑さ」とかいうことばを、規則の数ではなく、「重要度の高い規則ほどすくない」かどうかの度合をあらわすものとかんがえるべきなのではないだろうか。

No

Date

「複雑さ」ということばを このようにとらえなおしたとすれば、特殊な規則がおおいことはもはや それほど「複雑さ」を ますことではなくなる。そして、重要度の高い規則がおおいかどうかが「複雑さ」におおきく かかわってくることになる。実際、重要度のたかい規則がすくないことはのぞましい。人語はわずかの規則しかしらなくてもいちおうつかうことができるが、算語でも初心者が規則をたくさん しらなければ かけない言語は よい算語とはいえないだろう。

このように定義された「複雑さ」についてしらべること、すなわち規則の重要度(=使用頻度?)とその かずとの関係をしらべることは重要だとおもわれる。

2.2.9. 規則にかかわる言語学的研究の課題

言語学的研究の基礎をきずくまえにその課題について かたるのは はやすぎるかもしれないが、規則についての記述をおわるまえに それをまとめておくことにしよう.

規定書の規則と成文化規則とのずれが研究の対象となるということは §2.2.3. でのべたが、その研究をとおして、われわれは規定書のとりきめは妥当か、これから規定書をさだめるときにはどのような とりきめをすればよいか —— すなわち どのように算語を設計するか、などの問にこたえるようにするべきである. (「どのように算語を設計するか」という問は、たんに どのような規定書をさだめるかだけでなく、その算語の (§2.2.4. の意味での) 規則としてどのようなものの存在を予想し、どれを成文化規則とし、どれを非成文化規則とするか、という問もふくまれる).

また、成文化規則については、それが、人間がよんだりかいたりするものとしての算語において どういう役割をはたしているかをあきらかにするべきである. 非成文化規則については、まずどのような非成文化規則があるかをあきらかにし、つぎにそれらが どのような役割をはたしているかをあきらかにすべきで

No

Date

ある. なお、前節でのべた問題については ここではくりかえさ

ない.

2.3. プログラミング言語の自立性

人語の かきことば は はなしことば の代替的な記号体系であり、後者とならぶ独立な体系ではない。人間のつくりだした ほかの おおくの記号体系（たとえば モールス符号, 点字, 手話など）もまた 人語の代替的な体系である。では算語はどうだろうか。その検討に はいるまえに、それを検討することが われわれの目標と どうかかわっているかを かんがえることにしよう。

2.3.1. 自立性を検討する理由

例としてモールス符号体系をかんがえよう.

モールス符号は人語(のかきことば)と1対1に対応する代替的な記号体系である. したがってモールス符号は人語のもつ特徴をほとんどそなえているともいえる. ただし、モールス符号でおとられる文章の形式や内容が、モールス符号がかきられた用途にだけつかわれているゆえに、かきことばの一般からみるときわめてかぎられている、あるいは特殊であるという点をのぞけば、という条件つきではあるが.

モールス符号が代替的な体系であるということをわすれて この事実を指摘したとすれば、おおきく判断をあやまらせることになる. 自立的体系でありながら同様の性質を獲得した場合と、代替的体系であるがゆえにそれをうけついだ場合とでは、その性質がもつ意味におおきなちがいがあるからである.

また、つぎのような理由もある. 代替的な体系の場合には、その体系とそれが代替しているもとの体系とのあいだの関係、すなわちコードをしらべることが重要である. しかし、自立的な体系の場合にはそれができず、人語の意味論のように、記号があらわしている内容の体系やそれと記号(表現)との関係は、記号(表現)

の体系をとおしてしかしらべることができない。(もちろん、言語学以外の方法たとえば心理学的方法をつかえば、べつの接近のしかたもできるが)。

これらの理由から、§2、5で"人語の特性"といわれていることが算語にみられるかどうかを検討するまえに、算語の(人語からの)自立性について検討するのである。

2.3.2. 非自立性を支持する根拠

算語は人語（とくに英語）のかきことばから おおくの ものを借用している. 英字, 予約語のつづり, 識別子にあらわれるつづりなどをである. また算語の構文は、英語のただしい構文にはかならずしも あっていないとはいうものの、英語を意識してつくったものだとおもわれる. これらの借用は算語が人語に依存していることを意味するものである.

また、算語の成文化規則は 計算の実現方法（implementations）をあらわすことはできても、人間への伝達のために必要とされている「抽象（abstractions）」すら それじたいではまったく あらわすことができない.［ここで ある プログラム単位（program units すなわち手続き、関数、モジュールなど）の「抽象」というのは、そのプログラム単位の仕様（specifications）といいかえてもよい. 手続きの場合なら その機能のことであり、Clu［Liskov［1977］］の cluster や Alphard［Wulf［1976］］の form の場合なら そのデータ抽象（data abstractions）のことである.］ 抽象は §2.2.2. の最初の例のように識別子への人語からの借用によって象徴的にあらわされるだけである.

形式的な仕様記述法をとりいれた言語も、もしそれが一般のプログラマへの伝達につかわれるなら、そのためには人語にたよるほかはない。将来のことをかんがえても、形式的な仕様記述だけでプログラム単位の抽象を理解するなどということは天才プログラマでもないかぎりかんがえられない。したがって、形式的仕様記述のための言語がつかわれるようになったとしても、算語が人語の影響からはなれるということはありえない。

No

Date

2.3.3. 自立性を支持する根拠

算語の構文規則は人語とかなりちがっているようである。すくなくともモールス符号と人語、あるいは人語のかきことばとはなしことばのあいだにみられるような1対1にちかい対応は算語と人語とのあいだにはみいだせない。

また、計算機はかなり以前から算語の意味を解読することはできるのに、人語のかんたんな文さえ理解することができていない。これは人語と算語のあいだに距離があることをしめしているのではないだろうか。

さらに、算語がもともと機械の言語(すなわちmachine language)から段階的に発展してきたものだということも、証拠のひとつとしてかぞえることができるだろう。

そして、前節で指摘した、算語がおおくの言語単位を人語から借用しているということは、非自立性のつよい根拠とはならない。なぜなら、言語単位の借用は人語どうしのあいだにもみられる現象であり、そのために借用したほうの言語が借用されたほうの代替的な体系だなどとはいえないからである。[また、算語の表現内容を人語で表現できることをもって自立性への反証とす

ることももちろんできない。なぜなら、それは日本語が英語に訳せるのとおなじだからである。]

2.3.4. 工学的検討

以上 非自立性を支持する根拠と自立性を支持する根拠をならべてみたが、いずれも決定的でない。したがって、結局のところそれぞれの分野において、両方の面を考慮にいれながら研究をすすめるほかはないようにおもわれる。

そこで事実認定の問題はたなあげにして、ソフトウェア工学的な視点から自立性の問題をながめてみることにしよう。

§2.3.1.ですでにふれたように、算語にある人語的なものを排除して形式的な言語でおきかえようというようなこころみはみのりおおいものではない。形式的な体系でなくても、これまで人語から借用してきた部分を自立的なものでおきかえようとすれば、あらたに導入される単位や規則をプログラマがおぼえなければならないので、プログラマの負担がふえるであろう。むしろ積極的に人語からの借用をすることによって、それをゆるさない場合より算語の学習を容易にすることができるであろう。これはプログラマにとってもその雇用者にとってものぞましいことではないか。

さて、人語からの借用を積極的にみとめるとすると、伝達の効率をあげるために借用のしかたを規則化することがのぞましいとおもわ

れる。そこで、人語でかかれた簡略な「抽象」の記述を算語で表現するための非成文化規則を確立するべきではないだろうか。そのような規則は実際にある程度存在するとかんがえられる。(だから§2.2.2.の最初の例のようなプログラムをよむことができるのである)。それについて研究し、将来より伝達効率のたかい規則を確立することに寄与するのが言語学的研究の目的のひとつである。

すでにあるこのような非成文化規則については、第4章でのべることにする。

No 10

Date

2.4.「意味」のとらえかた

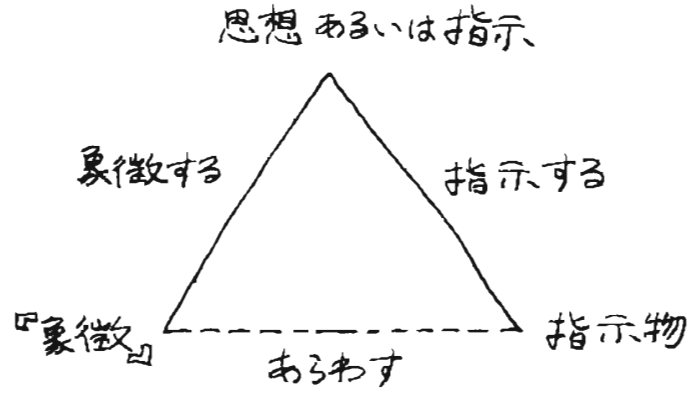
次節以降では「有意性」などのことばで「意味」を参照する必要が生じる。また実際に言語学的研究をおこなう場合には、意味の研究にはもちろん、文法の研究においても「意味」を参照する必要がある。　したがって「意味」の意味をはっきりさせる必要がある。そこで、「意味」をどうとらえるべきか、についてかんがえてみよう。

プログラムの意味がどのようにしてきまるかといえば、そのプログラムを構成している語がもつ「語彙的意味」が、文法に付随した「文法的意味」によって統合されて全体の意味になるのだとかんがえることができる。その「語彙的意味」も「文法的意味」も規定書によってさだめられているともいえるわけだが、構文の場合とおなじように、規定書で定義された規則のすべてが実際のプログラムに（おなじ重要度をもって）みいだされるわけではないだろう。規定書で定義されていない規則も存在するであろう。というよりも、意味というのはプログラムの表面にあらわれないものであるから、それが規定書にとってと、人間にとってとでちがっているとしても不思議はないであろう。また、もちろん機械にとってもちがってい

るかもしれない)。 ここで かんがえようとしているのは、おそに人間にとっての意味である。そして、ここでしめす「意味」のとらえかたは言語学的なものであり、算語の言語学的研究の基礎をなすかんがえかただとおもわれる。

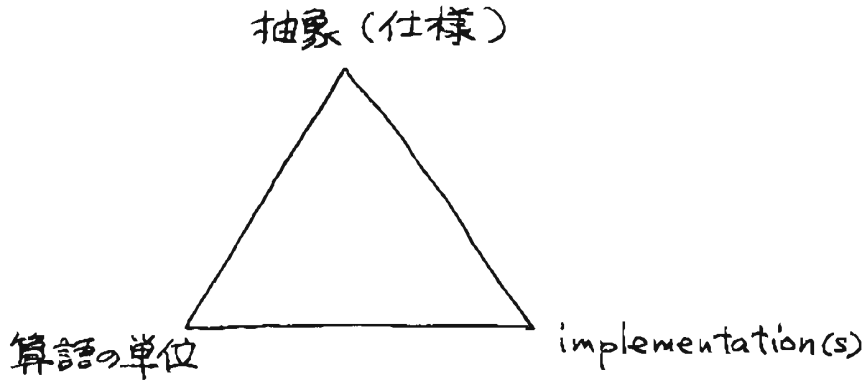
2.4.1. 基本的3角形

「記号」の意味にかんして Ogden=Richards [1923] の「基本的3角形」といわれる有名な図式がある。それは つぎのようなものである。



ここで『象徴』とは「記号」あるいは「名称」のことだとおもってよい。そうすると、うえの図式は つぎの ことをあらわす。「記号は『思想あるいは指示』を『象徴し』、『思想あるいは指示』は 実際の事物を『指示する』。それにより、『象徴』と 指示物とは 関接的にむすびつけられる。人語の意味論では この図式の『思想あるいは指示』を「意義」とよぶことがおおい。

算語においても これに にた、(ただし完全に対応するわけではない) 3角形をかんがえることができる。



ここで「算語の単位」としては、たとえば式のなかにあらわれた変数名や、手続き名あるいは手続きよびだしをかんがえている。しかし、それ以外の単位たとえば形態素をもってくることも可能である。（なお、この図式が人間についてのものであることを強調しておく。処理系についてもこれににた図式をかくことができるかもしれないが、それは本論文のテーマではない）。

算語においてこのような3角形をかんがえる理由は、

算語の単位 ——— implementation

というかんたんな図式では説明のつかにくいことがあることである。それは、ひとことでいえば「抽象」あるいは「仕様」とよばれるものがimplementationsとはことなるものとして、単位とimplementationsとのあいだに存在していることである。

No 54

Date

仕様をimplementationから区別する(言語学的)理由は、

A. 仕様の変更なしにimplementationをかえる(挿入する)ことができること.

B. implementationをあたえないうちから、ある手続きやデータをつかうプログラム(部分)をかいて、そのただしさをしらべることができること.

である. たとえば"配列Aを整列(sort)する手続きをimplementする方法は星の数ほどある. 効率上の要請からそのなかのすべてが特定の情況に適合するわけではないが、それでも選択の余地はかなりある. すなわち挿入可能なimplementationが存在するわけである. そこで実際に、すでにかかれたプログラムの効率を向上させることが必要になった場合には、Aのようなことがしばしばおこなわれる.

また、下降型(top-down)のプログラミングではB.のようなことがおこなわれるし、処理系もそれを支持していることがおおい。(たとえば、Adaの規定書では下降型プログラミングを支持するための規定がかなりこまかくかかれている)。

そして、仕様を"算語の単位"から区別する理由は、抽象の変更なしに単位の字づら(識別子のつづりなど)をかえることができるということである。

つぎに、算語の単位、抽象、implementationの3者の関係について、人語の場合との比較をしながらかんがえてみよう。

人語をOgden-Richardsの3角形にあてはめると、左下にくるのは語彙素であろう。語彙素よりおおきな単位をもってくることもできるだろう。いずれにしても、指示物は存在しないことがある(たとえば"抽象的な現象についてのべるような場合[Ullmann[1962], p.66(日)])。指示と指示物との関係は1対1に対応するというようなものではない。抽象名詞の場合は、指示しうるものはかずおおい(あるいは無限にある)。また、『象徴』と指示との関係も1対1ではなく、多対多であるのがふつうである。すなわち、同義性、多義性、同音性といった現

No

Date

象がみられる。

算語の場合も 抽象とimplementationsとの関係は1対1ではない。ひとつの抽象はおおくの可能なimplementationsを指示しうる。(このことは、算語の識別子が人語の固有名詞よりも普通名詞に にているということをしめしている。もっとも、それは人間にとっての はなしであって、機械にとっては、単位は抽象をとおさずにimplementationにむすびつけられていて、しかもその対応は1対1であり、識別子は固有名詞的であるといえよう)。

また、算語においては ひとつのimplementationは ことなる抽象とむすびつけることができる場合があるだろう。それから、3角形の左下に識別子の構成要素(形態素)をもってくれば" implementationは存在しない。さらに、単位と抽象も、非成文化規則によってむすびつけられているので"明確な関係にはなく、その対応は 多対多でありうる。

いままで おもにOgden=Richardsの3角形と算語の基本3角形との共通点を中心にのべてきたが、ふたつの3角形の重要なちがいを指摘しておこう。そのひとつは、Ogden=Richardsの3角形における指示物は記号体系のそとのものだが、算語におけるimplementation

はふたたび算語によってかかれるものだということである。ある単位は算語で"implement"され、そのimplementationのなかにふたたびいくつかの単位がふくまれるという再帰的な構造をしている。(再帰よびだしがあると主には、その単位じたいがそのimplementationのなかにふくまれることになる)。

ただし、もっとも下のレベルの抽象はその算語にかくりつけられていて、その算語によってimplementされないので、この場合は算語のそとのものを指示することになる。

ふたつの3角形のもうひとつの重要なちがいは、さきほどのべた、機械にとって、あるいは頑定書にとっても、単位とimplementationが抽象化を経由せず直接にむすびついれているという点である。

このように2つの3角形には相違点もあるけれども、その共通点に着目して今後「抽象」あるいは「仕様」のことを人語の場合と同様「意義」とよぶことにする。

No

Date

2.4.2.「意味」の定義

人語の「意味」について Ullmann [1962] は『名称[=『象徴』]と意義[=指示]の間の相互的，かつ可逆的な関係を私は「意味」と呼びたい」とかいている [p.64]．これにならって算語の単位と抽象との関係（算語では Ullmann のいうような可逆的な関係はなりたたないかもしれないが）を「意味」とよぶことにしよう．

これで一応「意味」の意味がさだめられたが，まだはっきりしていない問題がある．それは「抽象」あるいは「仕様」というのはなにかということである．ここでふたたび算語を自立的な体系とかんがえるかどうかが問題になる．

算語を自立的な体系とかんがえれば，「抽象」あるいは「仕様」は人語の「指示」とおなじくかたちのあるものではないので，それを直接言語学的な方法ではあつかいえないことになる．

しかし，一方算語を人語に従属する体系とかんがえれば，算語の単位が意味するものは人語のなんらかの単位であるはずだということになる．すなわち，「抽象」あるいは「仕様」ということばは人語でかかれたものとしてのそれらを意味することになる．ただし，算語の単位に直接対応しているのは人語の単位の深層構造であるかもしれ

ない。

どちらの立場をとるかによって、意味の研究方法もかわってくる

ことになる。この点については第3章でのべる。

2.5. 言語の特性

本節では まず"人語の特性といわれているものを列挙し、そのそれぞれが算語に存在するかどうかを検討しよう.

2.5.1. 「自然言語の特性」

Mounin [[1968], [1970a]] は人語をほかの事象からきわだたせている特性として つぎのものをあげている. ただし、ここで「人語」が はなしことばを意味していることを注意しておく. [以下の文章は『...』で引用した部分をのぞいては直接的な引用ではないし、筆者自身のかんがえもふくまれていることを ことわっておく].

A. 伝達機能

まず 言語が伝達機能をもっていること. しかも われわれがほかの手段によってあらわしうることの おおくが、人語をつかってまがりなりにもあらわすことができるという、伝達可能な意味領域のひろさが人語の特徴といえるだろう.

そして、言語による伝達が意図をもっておこなわれること. ただし、言語がつかわれる すべての場合に伝達の意図がみられるとい

うわけではなく、通常のつかわれかたをしているときにみとめられるということである。

さらに、伝達が可逆的におこなわれること。すなわち、AがBに伝達をするそのおなじ媒体をとおしてBがAに伝達をすることによって会話をなりたたせることができるということも特性のひとつである。ただし、これも言語がつかわれるすべての場合になりたつわけではない。たとえばテレビやラジオははなしことばの重要な不可逆媒体である。しかしながら、はなしことばの歴史上それがこのような不可逆な媒体をとおして伝搬されるようになったのは最近のことであり、それがはなしことばの構造に重要な影響をあたえることがあるとしても、それは将来のことだとかんがえられる。

なお、伝達の意図がみとめられないという点で人語などと区別される事象（それは「徴候（symptoms）」とよばれる）として、天体じたいが発する電波をあげることができる。これにたいして宇宙人が発する電波はおそらく徴候ではないだろう。

No

Date

B. 記号の恣意性

記号の恣意性とは、記号表現（シニフィアン）と記号内容（シニフィエ）が恣意的にむすびついていることである。たとえば"catという語に≪猫≫という記号内容がむすびつく必然性はまったくない。catが≪犬≫という記号内容とむすびついた言語があったとしても、なんのふしぎもないだろう。

しかし、人語でもすべての語（正確には語彙素）が（完全に）恣意的であるわけではなく、たとえば"擬声語はその記号内容すなわちそれがあらわす音と有契性がある。［有契的な単位は「象徴」とよばれる］。

C. 体系性

体系性とは『はっきり規定された規則や、よく知られた規則によって、はっきり規定された単位が組み合わされたり構成されたり』［Mounin［1970a］, p.88（日）］すること、すなわち構造があることである。

無体系的であるという点で人語とことなっている手順の例として、Buyssensは　造形美術、広告のポスター、看板、礼儀、

自然な身振りなどをあげているという［同上］．［たしかに造形美術や自然なみぶりなどが「規則にはずれている」というようなことはいわれない．けれども これらがほんとうに無体系的かどうかは疑問のあるところである．算語の非成文化規則をさぐりだそうというこころみは、造形美術やホロスターなどから規則をみつけだそうとするこころみに にているともいえる．］．

D. 記号とメッセージの線条性（linearity）

線条性とは、『時間の系の上で発信され、時間の系の上で認められる記号の連続によって構成されているということである』．［Mounin［1970a］, p.90（日）］．時間のうえで展開されるということから、それは当然、1次元的なかたちをとる．線条性をもたないものの例として、Mounin は工業図面をあげている．

E. 単位の離散性（discreteness）

離散性とは、単位が『相互に二者択一的な対立を示す』［Mounin［1968］, p.66（日）］ことをいう．たとえば人語の継起的な（すなわち時間のうえで展開される）最小の単位である音素（phonemes）についてならば、ある音素は『/p/であるか/非p/

であるかどちらかで、ある程度に/p/であるということは決してない』[同上]. このような性質は、音素よりおおきな単位である形態素(morphemes)についてもみられるものである.[音素,形態素についてはFを参照].

地図上では、川の幅は実際の川の幅に比例する連続的なものだが(すなわち離散性がないが)、道路の幅はその種類に応じて離散的にかかれることがおおい[Mounin [1970a], p.91(日)].

F. 2重分節 (double articulation)

2重分節あるいは構造の2重性(duality of structure)は、Mouninによって人語とほかの事象とをわける もっとも重要な特徴とされているものである.(2重分節はこれまでのところ人語にしかみられない、とMouninはいっている[[1968], p.78(日)]).

2重分節とは、人語のメッセージが最小の有意単位(すなわち記号内容とむすびつきのある単位)である形態素に分節することが(第1次分節)でき、その形態素はさらに最小の継起的単位(すなわち、これ以上は時間のうえでこまかい単位に分節することができない

(第2次分節)

単位)である音素に分解できることを意味する.[したがって音素は有意的な単位ではない].

形態素は「語」にちかいが、語が言語学的にはっきり定義できない単位であるのにたいして、形態素は換入(commutation)などの操作によってより客観的にみいだされる単位である. たとえばヘブライ語のつぎの3つの項目をみてみよう(これらは音素の連続というかたちでしめされている).

/zakartiihuu/ (私は彼を思い出した),

/zakartiihaa/ (私は彼女を思い出した),

/zakartiikaa/ (私はあなた[単数]を思い出した).

これらから /huu/, /haa/, /kaa/ (という形態素)がとりだされる[Gleason[1961]].

同様に音素も換入などによってとりだされる. たとえば

bit bet bot

などから /i/, /e/, /a/ などがとりだされる([同上]). (ただし、いまのべた換入の操作はかなり単純化されている. 実際にはより精密な方法が必要である).

2重分節はすくない単位(音素, 英語の場合46個[Gleason

[1961]]) から構成されるメッセージに多様な意味をあたえるために必要な性質である。もしひとつの記号内容にひとつの音素が対応しなければならないとしたら、音素の数は何1000個あってもたりないだろう。(不可分な)

ところで、はじめに形態素は最小の有意単位だとのべたが、これはあくまで原則であって、文脈によってはなりたたないこともあるということを注意しておく。たとえば常とう句などでは、いくつかの形態素があつまったものが記号内容と(かなり恣意的に)むすびつけられている。[このように記号内容とむすびつけられている最小の単位が「語彙素」である]。

以上のうち F. をのぞいては、どのひとつの特性も人語をほかのすべての事象からわけるのに充分ではないが、[Mounin によれば] これらすべてをみたす事象は人語以外にはみられない。

2.5.2. プログラミング言語は言語の特性をそなえているか

§2.5.1でのべた6つの特性が算語にみられないかどうかを検討することにしよう。これは算語が比喩的に「言語」であるにすぎないのか、それとも真に言語とよぶにあたいするものなのかを検討することでもある(もっとも形式言語学などでは「言語」ということばをもっとひろい意味でもちいているが)。

A. 伝達機能

算語に伝達機能が存在し、しかも意図的に伝達がおこなわれているという点はまちがいないだろう。ただ、第1に算語では人語よりはるかにかぎられた意味領域内のことしかつたえられないという点、第2に人語の伝達が人間どうしのあいだにかぎられているのにたいして、算語はひとから機械へとひとからひとへの両方の伝達につかわれるという点がことなっている。

まず意味領域のせまさについてのべる。[この点はWeinberg [[1971], p.210]が人語と算語のちがいとして指摘している]。§2.4.1.でのべたように、算語はそれじたいでは「抽象」すらあらわすことはできない。まして「計算」の表現にかかわらないこと、たとえば

感情をあらわすことはできない.

つぎに機械, ひとの両方が伝達の相手になるという点についてのべる.

過去には算語の伝達の相手は機械だけだとかんがえられることがおおかった. 計算機科学者がそのようなあやまったかんがえをもっていたのであるから, 言語学者があやまった判断をくだしたとしてもむりはないだろう. たとえば池上嘉彦[1972]は『「コンピューターの言語」のもっている機能はコンピューターにある一定の操作を行なわしめるということだけであり, それ以外の目的には使われない』といっている. (ただし, この記述は「コンピューターの言語」ということばを算語一般ととらえず, もっと せまい意味だとかんがえれば ただしいといえるだろう). このような認識がこれまで計算学を算語から遠ざけていたのだとおもわれる.

たしかに, 伝達相手が人間だけなら人語で記述すればよいのだから, 算語の存在理由は機械への伝達のためだといえるかもしれない. しかし, Algol 60 [Nauer [1963]] のように, 算語に属しているが はじめから ひとへのアルゴリズムの伝達を第1の目的としてつくられた言語もはやくから存在するし, またソフトウェアの保守性

や再利用性が問題とされるようになって、しだいに人間への伝達が重視されるようになった.

伝達の相手という点についてさらにかんがえてみよう. 人語の場合には、ひとへの伝達を意図して発せられることば と、ひとりごとのように伝達を意図せずに発せられることばとがあるが、算語の場合には

(1) 機械への伝達だけを意図してかいたプログラム,

(2) 人間への伝達だけを意図してかいたプログラム,

(3) 機械と人間の両方への伝達を意図してかいたプログラム,

(4) 伝達を意図せずにかいたプログラム

の4種類があるとかんがえられる.

機械への伝達だけを意図してかいたプログラムの例としては、ただ1回の使用のために「スタイル」上の配慮をまったくはらわずに(たとえば段づけをつけず、識別子はすべて「意味」のない英字1文字とし、注釈もつけずに)つくられたプログラムがかんがえられる.

ひとへの伝達だけを意図してかいたプログラムの例としては、

アルゴリズムの教育のために処理系の存在しない言語でかいたプログラム，たとえば Dijkstra の "A discipline of programming" [1976] のなかのプログラムをあげることができる．

また，「スタイル」上の配慮を充分におこなった ソフトウェア・メーカーのプログラムは 機械，ひとの両方への伝達を意図してかいたプログラムである．この場合の「ひと」はもちろん，そのプログラムの保守あるいはその一部の再利用をしようとするプログラマである．また，アルゴリズムの教育のためにかかれたプログラムのなかでも，ただちに処理系にかけられるかたちをしているものは この部類にいれられる．この場合の伝達相手としての「ひと」は，アルゴリズムの学習者である．

最後に，伝達を意図せずにかいたプログラムの例としては，プログラミングの学習のためにかいた，計算機にもかけず，かいた本人以外のひとにもみせないプログラムがある．

さて，つぎに伝達の可逆性だが，この性質は Weinberg [[1971], p. 208] も指摘しているように，算語にはみとめられない．まず，算語は機械から人間への伝達にはつかわれていない．将来計

算機にある種のアルゴリズムをつくらせて、それを人間がよむということはかんがえられるが、その場合も人語のように、人間と機械との「会話」に算語がつかわれているわけではない。またひとからひとへの伝達の場合にも、会話の手段とならないという点で可逆性はみとめられない。

このように算語による伝達が非可逆的なのは、現在の算語が命令的言語（imperative language）であるためである。現在までにもAI（Artificial Intelligence）のような分野ではかならずしも命令的でない算語がつかわれてきた（たとえば"Micro-Planner [Sussman[1971]], Prolog [Kowalski[1974]]など。Lispもこれにくわえることができるだろう）。このような算語の場合には、それをつかっての計算機と人間との会話をかんがえることはできる（もっとも、これらの言語をつかっての会話は人間にとってあまりよろこばしいものとはおもわれないが）。

いまみてきたように、ふつう、算語とよばれるものは会話につかわれていないのだが、TSS（Time Sharing System）の普及、あるいは今後予想される個人用計算機（personal computers）の普

(すなわち算語よりうえのレベル)

反は、応用(application)のレベルでの人間と計算機との「会話」の必要性をたかめつつある。しかしながら、まともな会話のできるシステムはまだ存在しないといってよい。算語の場合にも、可逆性の不在はかなり科学的・技術的な問題によっているとかんがえられる。

B. 記号の恣意性

記号の恣意性は算語にもみられる。たとえば"あるプログラムのなかで宣言され、つかわれている おなじつづりの識別子をすべて べつの識別子でおきかえても支障はない。これはちょうど人語で《猫》という記号内容が cat とむすびついていたのを、dog とむすびつけかえるようなものである。また、同様にAlgolの begin という予約語のかわりに start とか { とかいう記号をつかうことにしても さしつかえない。これらの例は算語の識別子や予約語に恣意性があることをしめしている。

C. 体系性

算語に構造があるのはあきらかである. しかも構造上類似点がおおい. そのうちの一部はF.であきらかにするが、それ以外にも、Chomskyらが人語について指摘している構文の再帰性、すなわちある構文がそれじたいをなかにふくんでいるという構造が存在すること(ただし、すべての算語におなじようにみいだされるわけではないが)があげられる.

どの算語にもみいだされる再帰的な構文としては、式の構文がある. 式のなかに(…)でかこんでふたたび式をかくことができる. また、おおくの算語で「構造のある文(structured statements)」はそのなかにふたたび文をふくむ. たとえば、

If Expression1 then Statement1
else if Expression2 then ...

のようなかたちが可能である.

そして、人語と算語との比較が、また算語に人語の研究方法をあてはめることが可能になるおおきな理由が、構造の類似性にあるのである。目的や媒体のことなる人語と算語との比較は、構造上両者の対応がつけられるがゆえに、対応する単位や規則について可能になるのであり、また言語学の方法は、対応する単位や規則のうえに移植することができるのである。

とはいっても、さきほど指摘した構造上の類似点も、よりこまかくみると ちがいがある。たとえば つぎのような点である。

算語にはしばしば かっこ構造がみられる。式のなかにみられる"(" と ")" が かっこ構造をつくっていることはいうまでもないが、このほかにもAlgolやPL/Iなどで"ブロック"をはじめるときのbeginとそれをおえるときのend〔PL/Iでは BEGIN, ENDのようにかくが〕, Algol 68やAdaなどの条件文のはじまりをしめすifとおわりをしめすfiまたはend ifなどがあげられる。〔Algol 60にはifに対応するとじかっこがないので、条件文はかっこ構造をなさない〕.

これにたいして人語にはこのようなかっこ構造が"ほとんど"

みられない（おもてにはあらわれない）．たとえば

What thinking Americans do about language is ···

は「分別のあるアメリカ人が言語について行なうことは···」の意味にも「アメリカ人が言語についてどういう考え方をしているかは···」の意味にもとれるが、これは構文上のあいまい性にもとづいている[Gleason[1961], pp. 188－189]．このような構文上のあいまい性はかっこ構造があればかんたんにふせぎえたものである．

構文上のあいまい性はあまりのぞましいものとはかんがえられないから、なぜそれを解決できるにもかかわらずかっこ構造がとりいれられていないのかということをかんがえてみる必要がある．そうすれば、算語のかっこ構造を、それがいくらふかくなっても計算機にとってなんら理解をさまたげるものにならないからといって乱用することは、いましめなければならないという結論がえられるであろう．そして実際、プログラムをしらべてみればあまりふかいかっこ構造がめったにつかわれていないことをしるであろう．

D. 記号とメッセージの線条性

算語もまた、かいたりよんだりするときのことをかんがえてみれば、時間のうえに展開される性質をもっている。とはいっても、プログラムじたいは空間のうえに展開されるし、人間にとってははなしことばのように受信者が時間のうえに展開されたメッセージと順をおってうけとることを強制されているわけではなく、途中からよんだり、まえへもどってよみつづけたりすることができる。送信側にとっても事情はおなじである。しかし、その場合でも局所的にみれば順をおってよんだりかいたりしているのであり、なんらかの理由でそのようなながれをさまたげられると、われわれはおそらく理解するのに困難をかんじるのである。

より線条性が完全なのは機械の場合である。コンパイラはほとんど例外なくプログラムをはじめからおわりへと逐次的によむ。ただし、全体を、あるいは部分ごとに2回以上よむことはある。

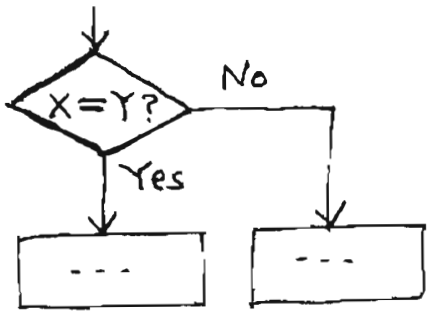
線条性は人間の逐次処理の能力をいかすために存在しているという面があるとおもわれるが、いまみたように技術的な存在理由もおおきい。したがって、計算機技術の進歩によって技術的

な理由がうしなわれれば、線条性が部分的にうしなわれる可能性もある。[まえの理由から、それが完全にうしなわれることはないだろうが]。そして、算語にはすでに非線条的な部分も存在するのである。

たとえば§2.2.2.でのべた段づけはプログラムを2次元的なものとするから非線条性をうみだしているといえるし、Fortranなどにみられる固定欄形式[Fortranでは、注釈は第1欄に"C"をかくことによってあらわし、それ以外のときは第1～5欄に文番号、第6欄は継続行かどうかをあらわし、第7～72欄に文をかく]や1行に1文しかかけないことは、(かかれたものとしての)プログラムに(段づけの場合よりはゆるいが)2次元性をうみだしているといえるだろう。ただし、これらの事実は人間にとっては非線条的なものだとしても、処理系にとっては非線条的に作用してはいない。

そのまま計算機にかけるのが目的でない算語の場合には、より つよい非線条性があらわれることがある。それは、フロー・チャートの場合である。フロー・チャートにあらわれる各種の処理をあらわす箱は2次元的な記号だし、それらは1本の矢印でつながれているとはかぎらず、「判断」の箱の場合には行先が2つに

わかれる.



算語の非線条性については §2.6. でさらに検討する.

E. 記号の離散性

算語の場合にも、文字はあらかじめきめられた数10~百数10字しかつかえず、それらの中間の文字は存在しない. また記号もある程度にbeginであるというようなことはなく, 離散的である. すなわち、この性質では人語と算語は完全に一致する.

F. 2重分節

A.~E. の性質はMounin によっても、人語以外の事象にみいだされていた. しかし、2重分節はほかの事象には観察されず、人語とほかの体系をへだてる最大の特性とされているものであった. だが、われわれは容易に つぎのような対応をみいだすことができる.

No

Date

人語　　算語

第1次分節 形態素 ------ 記号 (予約語, 識別子, リテラル, 特殊記号)

第2次分節 音素 ------- 文字

すなわち、算語において記号は最小の有意単位であり、文字は最小の継起的単位だということである。ただし、この対応がただしいかどうかはよく検討してみなければならない。

すこし検討してみると、上の図式への反例をいくつかあげることができる。まず特殊記号[すなわち"(",")",<,>=など]であるが、これには第2次分節がみられない。リテラル[1.25,"abc"など]も同様である。これらをふくめて一般に式にあらわれる記号には第2次分節がみられないことがおおい。この点については§2.7.1.でのべる。

また、識別子はさらに人間にとって意味のある単位に分節できる場合がある[この場合の「意味」は人語からの借用によって生じているのだが]。たとえば、IN_FILE, OUT_FILE, CONSTRAINT_ERROR, NUMERIC_ERROR [DoD[1980]]

などは _ のところで分節できるであろう．そうすると、識別子は人語の形態素に対応する単位ではないことになる．(したがって「語」にちかい単位であるといえる．ただし、「語」は言語学的には客観性のない単位とされているのにたいし、識別子ははっきり規定された客観的単位である点がことなる)．しかし、識別子が形態素に対応しないとしても、この場合は形態素に対応する単位は、識別子じたいとそれを構成する文字とのあいだのレベルに存在するはずである．(上の例では IN, OUT, FILE などがそのような単位であろう)．したがって、この場合にはそのような単位(以下形態素とよぶ)を前の対応表の「記号」のかわりにおけば、識別子については対応がなりたつことになる．

いまの考察は人間についてのものだった．これはhumanisticな理論をめざすわれわれにとっては当然のことだったが、処理系にとっては識別子が最小の有意単位であるということも、老婆心でつけくわえておくことにしよう．

予約語の場合についても検討をくわえよう．この場合は識別子の場合とほぼおなじであるが、おおくの言語では予約語

は下位の有意単位に分節することはできないであろう。したがって、予約語はそれじたいを形態素とよぶことができる。(ただし、分節される予約語をもった言語もある。Cobolの予約語はハイフンさえふくんでいる)。

修正した表をつくるまえに、もうひとつ検討しておかなければならないことがある。それは文字が継起的な単位であるかどうか、すなわち文字のならびが左から右へよまれ、かかれるかどうかという問題である。これが、文字を第2次分節の単位と認定できるかどうかにかかわっているからである。

なれたプログラマはけっして予約語を構成する文字を逐次的にはよまないだろう。すなわち、ひとめで同定することができるであろう。識別子についても、それがよくあらわれるものならば同様だろう。しかし、いずれについても、それをはじめてみたときには左から右へよむであろうから、本来的には継起的だといえるであろう。かくときには(局所的には)まずまちがいなく左から右へかかれる。したがって、文字は一応最小の継起的単位とみてよいだろう。そこで、つぎのような表ができあがる。

No

Date

人語	算語
第1次分節 形態素	形態素（識別子の構成要素，予約語）
第2次分節 音素	文字

［この表からは予約記号，リテラルといった単位は除外してある］

以上の考察では算語が（完全な自立性をもたないとしても）一応自立性をもつ体系であるという点から、自立的体系である人語のはなしことばとの対応づけをおこなってきたが、対応づけする相手としてかきことばをとれば、うえの表の「音素」のところが「文字素（graphemes）」となり、対応はより完全になるだろう。

人語の2重分節についてのべたときとおなじように、算語における2重分節のもつ意味についてかんがえてみることにしよう。算語を自立的体系としてみる場合には、2重分節は人語の場合と同様にすくない単位（文字）から多様なプログラムを構成するために重要なのだといえる。識別子についてみれば、このことはあきらかである。また予約語についても、通常の算語では予約語が数10個あるのにたいし、文字のかずもおなじくらいしかないから、

2重分節が不可欠であることがわかる。文字のかずがかぎられていなければ2重分節はなくてもよいということになるが、それはハードウェア上の問題としてもむずかしいし、人間にとってものぞましくないであろう。

一方、算語を人語に従属したものととらえると、2重分節は人語からの借用をするためにそれからコピーしてきた性質ということもできるだろう。

以上のように、算語は人語の特性といわれてきたもののおおくを、条件つきではあるが、もっている。すなわちおおとの事象のなかでももっとも人語にちかいものだということができる。そして、以上の考察でえられた様々の知識は、言語学的研究に役だてることができるであろう。それらの知識のなかには、「算語には非言語的部分がある」ということがあったが、この点については§2.7.でさらにかんがえることにする。

2.6. プログラミング言語のほかの特徴

§2.5. でとりあげた点以外に算語にはどのような特徴があるだろうか. 本節では、従来の議論で算語が人語とちがう点として指摘されてきた点をとりあげ、そのうちいくつかを否定し、のこりは肯定する. また. 従来指摘されなかったとおもわれる点もとりあげる.

2.6.1. あいまい性について

これまで算語にはあいまい性がないといわれてきた。本論の目的はこれを否定することにある。

あいまい性にはいろいろなレベルのものがあるが、多義性（polysemy）すなわちひとつの語（正確には語彙素）が複数の意義（記号内容）とむすびつけられていること、同形性（homography）すなわち複数の語がおなじつづりをもっていることなどの存在は語レベルのあいまい性をしめすものである。（このあいまい性が文脈によって解決されるかどうかは、このレベルであいまい性が存在するかどうかとは、またべつの問題である）。

ところが、池上嘉彦［1972］は記号表現と記号内容とのむすびつきについて、「人工的な記号体系（交通信号やコンピューター言語）では、当然のことながら1対1の対応がふつうである」といっている。この指摘は算語でも多義性や同形性があるからまちがっている。［さらに算語には同義性（synonymy）すなわち複数の語がおなじ意義をもつこともあるとかんがえられるので、対応は1対多でさえない。引用した記述はその点でもまちがっている］

たとえば、おおくの算語で"＋という記号は≪実数の恒等演算≫、≪整数の恒等演算≫≪実数の加算≫、≪整数の加算≫の4つの意義をもちうる(前2者は＋が単項演算子としてつかわれる場合であり、のこりは2項演算子としてつかわれる場合である。前2者はたがいに区別する理由はないかもしれないが、すくなくとも3つの場合がはっきりと区別される。[後2者はおなじ≪数の加算≫という一般的な意義にまとめたくなるかもしれないが、整数とか実数とかいうものは、数学でおなじなまえでよばれているものとはちがって、上限,下限があり、しかもそれらは整数と実数とでちがっている。また、実数の場合精度が有限である。したがって「加算」の意味も整数と実数とではちがってくるから、ひとつにまとめるのは不正確なのである]。

このように、ひとつの演算子や識別子(のつづり)が複数の意義とむすびつけられる現象をAda[DoD[1980]]ではoverloadingとよんでいる。Adaがあらわれるまえはoverloadingということばがつかわれることはすくなかったし、Adaほど徹底したoverloadingは存在しなかったようである[うえの＋の場合には、あいまいさが文脈によって

解決されずにのこって、プログラムがあやまったものとみなされることはふつうないが、Adaにおいてはそのようなことがおこることがある。Adaのoverloadingについては[第5章参照]。また現在でも算語にかんして多義性とか同形性とかいうことばがつかわれることはないようだが、うえの例のように、この現象はひろく存在していたのである。

つぎに構文のあいまい性についてのべよう。人語の構文のあいまい性については§2.5.2.C.でふれたが、算語では規定書上で構文にあいまい性があることはまれである。しかし、Algol60の最初の版, Nauer [1960]にはそのようなあいまい性があった。

```
if P then
    for I := 1 do
        if Q then J := 1 else K := 1
```

は

```
if P then
    {for I := 1 do
        if Q then J := 1 else K := 1}
```

のように解釈することもできるし、

if P then

{ for I := 1 do if Q then J := 1 }

else K := 1

のように解釈することもできる[米田, 野下[1976]].

しかし、規定書上であいまい性がなくても、人間にとってはあいまい性がないとはかならずしもいえないだろう. PascalやPL/Iなどでは文脈依存の規則によって条件文の構文にあいまい性はないが、§2.2.2.にしめした(上記のAlgol 60の例ににた)条件文の例は、プログラム上にあらわれた規則に上の例のようなあいまい性が存在することをしめしているようにおもわれる. なぜなら、§2.2.2.の条件文は成文化規則としての構文と、段づけとが矛盾していて、全体として非文法的になっているか、そのことが明確でない(みのがされやすい)からである.

overloadingやAlgol 60の構文のあいまい性の例は人間にとっても規定書にとってもあいまい性が存在する例だったが、Pascalの「構文のあいまい性」は規定書にとってはあいまい性がないが、人間にとってはあるという例だった.

以上のほかにもあいまい性は様々のレベルで存在する。overloading やこれらのあいまい性の問題については第5章で考察する。(構文のあいまい性は統辞論の問題なので第5章ではふれない)。

蛇足になるが、池上が取りあげていた「交通信号」は、交通標識や自動車の停止灯までふくめた交通法規の体系というふうにひろく解釈すると、そのなかにはあいまい性がみられる[Mounin [1970b]]。それに、交通信号機じたいも、歩行者の行動をみればあいまい性がないとはいえないだろう。すなわち、あいまい性はしばしば人語だけにある特質だといわれるが、それどころではなく、かなりひろくみられるものなのである。

2.5.2. 閉鎖性と進化性

モールス符号のように完全に規定された体系については『人間が伝達を目的にコードをつくったのに対し、[自然]言語は伝達自身の中で自然と作られる。だから、コードは、閉鎖的で動かないものであり、使用者たちの明白な賛成なしに変えられることはないのに対し、言語は、開放的で、すべての発話ごとに新たに問い直されるのである』[Mounin [1970c], p.102 (日)]ということができるが、うえの引用文中の「コード」を「算語」でおきかえると、あやまった記述があらわれる。「コード」を「算語の規定書規則」でおきかえれば、『使用者たちの明白な賛成なしに変えられることはない』かどうかはべつとして、ほかの部分はただしいが、とくに非成文化規則は『閉鎖的で動かないもの』ではない。また、はっきり規定された規則(いまの場合、規定書規則)についても、Mouninはかわりうることをただしく指摘しているが、『閉鎖的で動かない』ことを強調するあまり、このことをわすれてはならない。

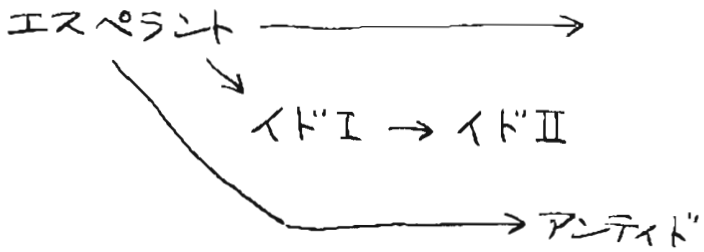
算語が進化することは、たとえば"つぎ"のような進化の系列をみることによってわかる。

No

Date

Fortran (I) → Fortran II → Fortran IV → Fortran 77 → …

82.2.6. で人工自然言語は算語ににた性質をもっているとのべたが、閉鎖性，進化性という点でも両者はにている。すなわち、いずれも『閉鎖的で不動の体系でありながら、しかも進化的である』[Mounin [1970c], p.103(8)].



これらの人工言語 —— 算語と人工自然言語に共通な進化の特徴は、通常人語が連続的に進化するのにたいして、人工言語はその成文化規則が不連続的にかわってあたらしい言語が登場することである。ただし、それはあたらしい言語がうまれたとたんにふるい言語が駆逐されるということではない(事実はむしろ逆に、人工自然言語の場合、イドやアンティドがうまれてもあいかわらず

エスペラントがもっとも幅ひろくつかわれている).

算語の場合

この不連続的な変化の過程をつぎのように弁証法的にとらえることができるのではないかとおもう. プログラミングの技術の発展にともなって「抽象」の構造(下部構造)も変化する. 非成文化規則はそれにつれて徐々に変化することができるが, 規定書にしばられた成文化規則は抽象の構造の変化を充分に反映することができない. したがって抽象と成文化規則とのあいだに矛盾が生じ, 拡大していく. それがたえがたいところまでひろがったときに規定書が改訂される(止揚される). 人工自然言語の場合にもほぼ同様にとらえられるであろう.

No.

Date

2.6.3. プログラミング言語は はなされない

算語は原則的には音声を媒体として伝達されることがなく、人語のかきことばのように、かかれるものである（この点もWeinberg［[1971], p.207］が指摘している）。ただし、プログラムの一部をよみあげることはある。［その場合は文字どおりの同音性（homonymy）が問題になることがある。たとえば筆者はgetcとgetsyがいずれも［get si:］と発音されたために混乱しかけたという経験をしている］。しかし、その場合には人語とは逆に、「音声言語」のほうが代替的につかわれているわけである。このように、人語と算語には主要な媒体が音声であるか文字であるかというちがいがあるわけだが、重要なのはそのことだけではなく、それが構造上のちがいをうみだすこと、また構造上のちがいを正当化することである。

No

Date

2.6.4 命名の頻度と語の有効範囲 (scope)

プログラムをかく場合には、始終「命名」をおこなう必要がある。すなわち、手続き、変数などを宣言するたびに、その名前をかんがえだされなければならない。これにたいして人語でふつうに伝達をおこなう場合には、めったに命名の必要はおこらない。

命名がおこなわれないとき、したがって人語における通常の場合には、語はある談話にあらわれるまえから語彙的意味をもっている。しかし、命名がおこなわれるときには、したがって算語における通常の場合には、プログラムのなかで識別子に語彙的意味があたえられているようにみえる[ただし、この点には疑念がある。それについては本節のおわりにのべることにする。]。しかも定義の有効範囲はひとつのプログラムのなかのさらに せまい部分に限定されることがおおい。このことは、ひとつの「語」が複数のプログラムにまたがってあらわれることがすくないということを意味している。

以上のようなちがいは、それじたい研究の対象として興味あるものであるが、このちがいは意味論を展開するうえでおおきな問題となるであろう(統辞論にはそれほどおおきく影響しないとおもわれる。[統辞論、意味論については第3章を参照されたい。])。しかし、

算語でも予約語や特殊記号はひとつの言語のなかでは共通につかわれるし、標準の手続き、関数もあるし、標準的なパッケージの使用もおこなわれる。また、人語でかかれたもののなかでも、学術書はその本のなかだけで通用する定義（命名）をしばしばおこなうから、示差的（distinctive）なちがいではないだろう。人語における命名を研究し、これを算語におけるそれと比較することは、算語の研究におおきく寄与するとおもわれるが、いまはそれについてのべる用意がない。

さらに、このちがいを工学的な観的からかんがえてみよう。ひとつの語が複数のプログラムにあらわれることがすくないということは、ある抽象（機能、データなど）が必要になったとき、それがほかでつかわれている可能性のあるものであっても、そのimplementationをいちいちかかなければならないということを意味する。はばひろくつかわれうる語をかずおおく定義することができれば、それだけかくべきプログラムの量をへらすことができ、プログラミングコストをさげることができるであろう。はばひろくつかわれうる語を定義し（というよりはみいだし）、それをあつめて辞書をつくること、あるいはこれまでいわれてきたいいかたをするならば、汎用性のあ

No

Date

る規格化されたソフトウエア部品としてのモジュール（手続き，データ型など）をあつめたカタログ（それはIchbiahなどにより、自動車部品のカタログにたとえられている）をつくることが、ソフトウエア工学者のひとつの目標である。

ところで、本節では算語においてはプログラムのなかで識別子に語彙的意味があたえられているとのべ、そのことにもとづいて議論を展開してきたが、そのことには疑念がある。算語における「命名」はふつう まったく恣意的におこなわれるわけではなく、人語から語を借用しておこなわれる。そのために ことなるプログラムにあらわれる類似の（プログラミングによくつかわれる）抽象にたいして、implementationは「命名」のたびにあたえられるとしても、同一のまたは類似のつづりの識別子がつけられることがおおい。それらのあいだの意味のちがいはむしろ おなじ語が ことなる情況に適用されたというだけのちがいとかんがえたほうがよいかもしれない。このような語の『適用のしかたのずれ』[Ullmann[1962]]は人語にふつうにみられるもので、それが「命名」ということばでよばれるものでないことはもちろんである。しかし、それにしても算語のほうが より、名前（識別子）との対

心の確立されていない概念(抽象)がよくあらわれるということは

おそらくいえるだろう[このことはしらべてみる必要がある].

2.6.5. プログラムは「つかわれ」，かきかえられる

人語のことばは「はなしかえる」ことはできないが、プログラムはかきかえられる。この文には2つの意味がある。

まず第1に、話しことばはいったんしゃべったことを伝達相手に達するまえにけしさることはできない。だが、プログラムはかいている途中あるいはかきおわってからも、機械やひとに伝達するまえに何度でも修正することができる。この点ではかきことばは箪語にちかい。

第2に、人語のことばは、それがきいてにつたわれば通常は、目的が達せられてそれでおわりとなり、さらにつかわれることはない。しかし、プログラムは翻訳系によってそれが理解されたときがおわりではなくてはじまりであり、それから何度もつかわれる。【1回しかつかわれないこともあるが】。そして、その過程で改訂が必要なことがわかると、何度も（部分または全体が）かきなおされる。はなしことばでも過去にあやまったことをしゃべったときには、あとで訂正することがあるが、これはプログラムの改訂とは目的もちがう。プログラムの改訂はバグが発見されたときにおこなうだけでなく、その使用目的の変化や機械の交換などがあった場

合にもおこなわれるが、はなしことばのほうはそれに相当するようなことはない。ただ、かきことばである文学のような場合には、何度もよまれるし、かきかえられることもある点がプログラムにている。

——

2.7. プログラミング言語の非言語的部分

§2.5.2.であげた算語と人語の相違点のうちのいくつかは、算語体系全体にかかわるものであるが、ほかの点はおもに算語の一部分とかかわっているようにおもわれる。たとえば§2.5.2.F.でのべたように、算語の2重分節をみたさない要素は文の部分よりは式の部分にあつまっているようにおもわれる。また、段づけは、いわば算語におけるパラ言語的な部分(人語でいえばアクセント、イントネーションなど)であって、ほかの部分と一応きりはなして論じることができる。(この場合の「部分」は「式の部分」などという場合の「部分」とは意味がことなるが)。

段づけについては§2.2.2.でのべたので、本節では前者のような意味での非言語的部分についてだけ論じる。

2.7.1. 式の性質

算語のなかで 式は他の部分とはちがった独特の意義をもった部分だとかんがえられる。2重分節の欠如も、そのために生じてくるのだとかんがえられるし、ある種の算語にみられる式の非線条性もそれと関係しているようにおもわれる。本節ではこのふたつの問題についてかんがえてみよう。

A. 非2重分節性

Mounin [1970d] は数式に人語におけるような2重分節が存在しないことを指摘している。すなわち、数式にあらわれる記号（変数名, 演算子など）は最小の継起的な単位であって、しかも最小の有意単位であるから、数式には人語の音素に対応するような継起的で非有意的な単位は存在しないということである。[算語では変数名として2文字以上からなる識別子もよくつかわれるが、数式ではそれがほとんどつかわれないことに注目されたい]。ただし、例外は存在するであろう。たとえば mod という記号は、継起的で非有意的な単位に分節できる。

算語における式も、数学における式ほどではないが、同様

の傾向がある. たとえば"

B**2 − 40*A*C　[DoD [1980], p4−10]

という式のなかで"第2次分節がみられるのは ** だけである. (リテラル40に第2次分節がみられないという点については後述する). この場合 識別子がすべて1文字で"できていることが"非2重分節性をつよめている. もちろん 式のなかには2文字以上からなる識別子がつかわれることもおおいが、みじかいもの、とくに1文字のものがつかわれることが、文のなかよりもおおいようにおもわれる. 文のなかだけにあらわれる名前として手続き名があるが、これに1文字の識別子がつかわれることは、実用的なプログラムではほとんどない.

MacLennan [1979] は さまざまな記号体系を検討して、算語でも式(Maclennanは"formula"とよんでいる)にはsymbolicな記法をつかい、文(Maclennanは"sentence"とよんでいる)にはnon-symbolicな記法をつかうべきだ"といっている. これは算語のなかで"文がはたす役割と式がはたす役割をはっきりわけ、式にあらわれる記号については第2次分節のないもののほうがよいという趣旨だ"と解釈される.

以上の仮説，MacLennanのかんがえがただしいとすれば，これはLispやAlgol 68のような式言語(すなわち文が存在せず，普通の言語では文である if … や for … do … も式(expression)であるような言語(statement)の否定ともとらえられる．ただし、それは文のある言語を肯定するとはいえない．なぜなら、文のある言語でも、MacLennanの提案しているような算語とはちがって 文のなかに式がいたるところ あらわれて、両者の性格をうすめているからである．

ここで、より具体的に Pascal の式にあらわれる記号について、第2次分節がみられるかどうかを検討してみよう． Pascalの式にあらわれる記号は つぎの3種類にわけることができる．

(1) 1分節されない記号．

＋ － ＊ ／ ＝ ( ) [ ] , ↑ . < >

(2) 有意的で継起的な単位に分節されるもの

<> <= >= 無符号数 文字列定数

(3) 第2次分節のみられるもの

div mod nil in or and not .. 識別子

ここで(1)にふくまれる記号が分節されないというのは、それが1文字からなっているからである。

(2)にふくまれる記号については説明を要するだろう。<>, <=, >= の3つは、処理系にとっては最小の有意単位ともいえるが、人間はそれを2つにわけてとらえることができる。<>は「<または>」、したがって《ひとしくない》という意義をもつのだし、<=も「<または=」と解釈される。>=についても同様である。すなわち この3つの記号は、おなじ統辞法(構文)によって、より ちいさな単位から構成されているとかんがえられる。

あるいは、人間にとって これらの記号は ひとめで同定されるがゆえに 継起的な単位に分節できないとかんがえられるかもしれない(すなわち、<=における<と= などは人間にとって同時的なものとかんがえられるかもしれない)。そうだとすれば、これらは(1)に分類しなおされることになるが、その場合にも第2次分節がみられないことにかわりはない。

(2)にふくまれている ほかの記号についてかんがえよう。無符号数(すなわち 1.2e, 421 など)は数字や小数点に

分節されるが、それらは非有意単位ではない。無符号数はそれを構成しているそれぞれの文字によって意味づけられているのであって、それ全体が記号内容と恣意的にむすびつけられているわけではない。(これがまえにあらわれた(なかの)式の40に第2次分節がみられないことの説明でもある)。この点は文字列定数の場合もおなじである。(ただし、おなじ無符号数でもラベルあるいは文番号としてあらわれる場合[Pascalでは文につくラベルは無符号数がつかわれるし、Fortranの文には文番号として無符号数がつかわれる]は、電話番号とおなじように第2次分節がある)。

最後に(3)にふくまれる記号について説明しよう。".."以外は説明を要さないであろう。".."は≪…から…まで≫という内容をもつ記号だといえる。たとえば"[1..5]は≪1から5までの整数をふくむ集合≫という意義をもつ。さて".."が特殊記号とよばれるもののなかで"特別の位置にあるとする理由は、".."と、それを分節してえられる"."とのあいだに意味的なつながりがあるとはかんがえられないということ

である．すなわち，“..”における“.”は非有意単位であり，“..”とその記号内容とが恣意的にむすびつけられているとかんがえられるということである．しかし，この場合もし“.”への分節性が否定されれば，“..”はいつに属することになる．

No

Date

B. 非線条性

数式は2次元性をもっている。たとえば"

$$\frac{a+\frac{b}{c}}{d} \qquad \sum_{n=1}^{5} n^2$$

のように。これは数式が かかれる記号体系であるということをいかして、より直観にうったえる形式をとっているということだといえる。算語も、かかれる記号体系だということをいかして、2次元性をとりいれるべきだとかんがえられる。そして、それをとりいれるとすれば、まず"式にとりいれるべきだろう。

実際、(限定された範囲ではあるが)2次元的な式をもつかうことのできる算語もある。たとえば" COLASL, MADCAP [Sammet [1969]] がある。おおくの算語で式が2次元的にかけないのは,処理の困難,入出力の困難といった まったく技術的な理由によっているのであろう。たとえば" うえの分数式を線条的にかけば

(a+b/c)/d

のようになるが、このかたちより もとのかたちの方が よみや

すいとおもわれる。また、この場合にもかっこが必要になっているが、さらに複雑な式の場合には、かっこがふえることによってよみにくさがいっそうますことになる。

2.7.2. 非言語的部分の存在意義

自然科学書には人語にまじって数式があらわれる。そして人語部分にその式の一部を引用して説明している。また、必要に応じて図や表がつかわれる。このようにすることによって、人間の理解力をできるだけひきだそうとしているのだとおもわれる。このことは、算語における文と式が MacLennan [1979] のいうように、ことなった機能をもつものとして相補的につかわれる可能性をしめすとともに、算語に図や表を文や式とならぶ相補的な部分としてとりいれる可能性をしめすものである。decision table は表が算語としてひろくつかわれている例といえるだろうが、表にしろ図にしろ、それが算語の一部として文や式とくみあわせられてもちいられたという例はきかない。この問題は工学的には興味があるが、言語学的研究の範囲外の問題なので、これ以上はのべない。[しかしながら、算語における図式的部分の欠如を、注釈として図表をかくことによっておぎなうことはよくおこなわれている。このような注釈は言語学的研究の対象となりうる].

2.8. プログラミング言語の変化 (variety and change)

§2.6.2 で算語の規則は変化するとのべた. 人語の場合、単位と規則はともに、変化するものとしてあげられる. しかし、現在の算語についていえば 単位は変化するものというより むしろプログラムごとにことなっており、あまりそれじたいは研究の対象にならないとおもわれる. そこで、規則についてだけかんがえることにする.

変化にはつぎのふたつがある.

A. 共時的変化

共時的変化とは、ある時代におけるひとつの言語圏のなかにみいだされる多様性のことであり、方言は それがはっきりしたかたちであらわれたものである. 人語では方言として地域方言と社会方言が区別される. これに比較できるのは、ある計算センタ(あるいはネットワーク)のまわりにあつまった人々が おなじ処理系をつかうことや、プログラムの共同利用などをとおして、ほかのセンタやネットワークを利用するひとびととはちがった算語の方言をつくること(地域方言), あるいは おなじ大学にいながら大学生は教育用の処理系をつかい、(教官や)大学院生は実用的な処理系

をつかうために、またプログラミングの経験や目的がちがうために ことなった方言をつくること(社会方言)などが想像される。(実際にそのような現象があるかどうかは、しらべてみなければはっきりしたことはいえないが)。

B. 通時的変化

通時的変化とは、言語の時代による変化のことである。その例は§2.6.2.であげたFortranの進化である。そのFortranの場合には互換性に注意をはらっているので、ふるいFortranのプログラムを あたらしいFortranの処理系にかけるために それほど大幅な変更を必要としないが、この変化がもっとひどくなると まったく あたらしい言語が誕生したものとみなされる。たとえばAlgol 68 [Wijngaarden [1975]] はAlgol 60の後継者といわれたし、おなじAlgolの名前をもっているが、互換性は まったくない べつの言語である。

共時的変化にしろ通時的変化にしろ、どこまでがひとつの言語のなかの変化で、どこからがべつの言語になるかはあいまいであ

る.[Sammet[1969] もこのことを指摘している]. 人語の場合にも おなじことがいえる. たとえば"ロマンス諸語は 政治地図にしたがってわけられているが、言語学的には どこで区切ってよいかが はっきりしないものである. したがって、この節で"のべることは 一応 ひとつの言語内に範囲をかぎっているが、ことなる言語どうしをとっても ある程度おなじ議論をすることが"できる.

成文化規則と非成文化規則のうち、前者は規定書や処理系に依存するので"かわりにくい. 後者のうちでも、処理系に依存する部分はかわりにくいで"あろうが、人間のあいだだけで意味をもつ規則はかわりやすいで"あろう. むしろ プログラムの「スタイル」はおなじものをみつけることの ほうがむずかしいくらいで"あろう.

ふたたび工学的な視点をとると、算語の標準化(規格化)の問題があらわれる. 標準化は方言統一のこころみだ"が、これは規定書規則の統一だけをめざしている. しかし、伝達効率の向上という点からは、規定書にかかれない非成文化規則も ある程度統一する必要があるだろう. だが、これを規定書にくわえるこ

とは非成文化規則を成文化規則にすることであり、ある場合にはのぞましいが、おおくの場合にはよほどその柔軟性に配慮をしないかぎり柔軟性をうしなわせることになるのでのぞましくない。したがって、標準化にあたっては、つねにまもるべき規則を規定した規定書をつくるとともに、ときにはやぶったほうがよいこともある規則を指示する統一されたスタイル・ブックをつくるのがよいとかんがえられる。

算語の標準化についてもうひとつのべておこう。人語の場合、すでにうまれた方言をいそいで統一しようとすることは、その方言のうえにうまれた文化的遺産のことをかんがえるとこのましくない。算語の場合は、統一によってふるいプログラムがあたらしい処理系のうえでうごかなくなるということをのぞけば、あまり問題はないようにみえるかもしれないが、各方言のうえで発達してきた非成文化規則（それは文化的遺産とよびうるであろう）をよく検討せずに統一すると、おおくのものをうしなうことになるだろう。

第3章 プログラミング言語学とその研究分野

第2章における考察によって、算語の言語学的研究のための基盤がえられるとともに、算語はおおくの記号的事象のなかで独特の位置にあることがわかった。これらの事象の全体を研究するのが記号学であるならば、算語の言語学的研究はそのなかで言語学に匹敵する地位をしめうるものだとかんがえられる。そこで、今後この学問をプログラミング言語学とよぶことにしよう。

それでは、プログラミング言語学にはどのような研究分野、どのようなテーマ、どのような方法が存在するだろうか。本章では第2章でみいだされた問題を整理してみよう。ただし、これはまだ実際におこなわれていない研究の分類であるから、研究がすすむにつれて修正する必要が生じるであろう。

プログラミング言語学も（言語学と同様に）ひとつの算語体系にたいする個別的研究と、複数の算語体系あるいは算語体系とほかの体系とを比較対照する研究とがある。いずれも、「ひとつの算語体系としてどのくらいひろがりをもったものをかんがえるか、たとえばひとつの方言をとるか、Cobol, Fortran といった言語族をとるかによって様々の分野にわかれるであろう。そしてまた、それぞ

れが"共時的研究と通時的研究とにわけられるで"あろう。共時的研究とは共時態（synchrony）すなわちある時期における言語の状態を記述しようとするもので"あり、通時的研究とは通時態（diachrony）すなわち言語の時代的変化をあつかうもので"ある。通時的研究としては、算語の一部分が"時代とともにどう"かわったかをしらべるとか、2つの算語の一部分をくらべるとかいったこともありうるが、本来は算語の構造の変化や比較など"をおこなうもので"あり、したがって対象となる算語の共時的研究がその基盤となる。

No 1

Date

3.1. 個別的研究

3.1.1. 個別的研究の分野

個別的な
共時的研究をおおきくつぎの3つにわけることができるであろう。(A.とB.をわける根拠が2重分節であることはいうまでもない。)

A. 文字論 (graphemics)

文字論は言語学の音韻論に対応する分野ということができるが、算語にはどういう文字がつかわれ、それがどのようにして形態素をかたちづくるかを研究する。現在の算語のおおくはASCIIコードやEBCDICコードにふくまれる、あらかじめきめられたかぎられた文字だけをつかうようになっているので、計算機界で主流をしめている算語についてはあまりおもしろい成果は期待できないであろう。有名な算語のなかで研究の対象として興味がもてるのはAPL, Algol60, Algol68くらいであろう。(Algol60は、発表言語のレベルで規定書にもハードウエアにも束縛されずに文字がつかわれうるので興味の対象となる)。

B. 文法論 (grammar)

文法論は形態素がどのような規則でプログラムをかたちづくっ

ているかを研究する。これは言語学の文法論に対応する分野で"あり、したがって算語の場合「文法」のなかに通常ふくめられる「意味」の問題は原則」としてあつかわない。(ただし、従来の意味(プログラムの)での「意味」とプログラミング言語学における「意味」とはちがうので、従来の意味での「意味」の一部は「文法」に属することになるだろう)。

これまでの、算語の文法論は、規定書の記述をあつかう演繹的なものだった(これはBirkhoffの分類では哲学的研究にふくまれるものである)が、ここであつかおうとしている言語学的文法論は、実際のプログラムにあらわれる文法を研究する帰納的なものである。また、両者ともプログラムを支配している規則」を問題にするわけだが、後者は前者があつかわなかった問題を中心にあつかうことになるだろう。(たとえば"、§3、1、2、で"定義する形態論があつかうのは そのような問題である)。

C. 意味論(semantics)

意味論は算語の意味の構造をあきらかにすることをめざす。人語の意味論に対応させれば"、意味論には哲学的意味論

(denotational semantics など、従来の算語の意味論はここに属するであろう), 心理学的意味論, 言語学的意味論が存在しうるであろう. ここであつかうのは もちろん言語学的意味論である.

§2.4. で人間にとっての意味と、規定書にとっての、あるいは機械にとっての意味とはちがっているだろうとのべたが, そのうち規定書にとっての意味を研究するのが 従来の意味論であり、人間にとっての意味を研究するのが心理学的および言語学的意味論であるといえるだろう.

§2.4. でしめした算語の基本的3角形のうえで かんがえれば、算語の規定書における「意味」は3角形のおもに底辺部をあつかう (すなわち抽象ぬきで 単位とその implementations との関係をあつかう). これにたいして人語の言語学的意味は §2.4. でのべた Ullmann の「意味」の定義からもわかるように Ogden=Richards の3角形の左半分をあつかうが、それにならって算語の言語学的意味論も算語の3角形の左半分をかんがえるものだととらえることができる.

No

Date

個別的な通時的研究にたいしては「歴史プログラミング言語学」という名をあたえることができるだろう．しかし，これについてはのべない．

3.1.2. 文法論

文法論はさらにいくつかのレベルにわけることができるであろう。それについてすぐにおもいつくのは、「記号」が規定書で定義された客観的な単位であることから、記号以下のレベルをあつかう部門と、記号以上のレベルをあつかう部門とにわけることである。すなわち、形態素がどのようにして記号をかたちづくるかを研究する部門と、記号がどのようにしてプログラムをかたちづくるかを研究する部門とにわけることである。しかし、手続きや関数のよびだしのような場合には、このレベルでわけてかんがえるのはかならずしも適当でないとおもわれる。この点については第4章でさらにあきらかにするが、ここではとりあえず"つぎのような例をかんがえよう。

IS_IN?(klist, id1)

[これはGuttag[1977]にある例である。?が識別子中にあらわれるのは普通でないが、いまの問題には関係がない]。これはリストklistのなかに識別子id1があれば"true"なければ"false"をかえす関数である。ここで"IS_IN?"は本来それじたいで単位をなすのではなく、むしろISとIN klistがそれぞれ単位をなすものだとかんがえられる。したがってIS_INだけをきりはなして研究するのは適当でないであろう。

したがって文法論のもっとも下のレベルとしては、よびだしの場合にはそれ全体をまとめてかんがえ、変数や定数の場合にはひとつの名前（Adaの用語ではname）を単位としてかんがえる部門をおきたい。これを言語学との類推から形態論（morphology）とよぶことにしよう。

算語の構造からかんがえて、このうえのレベル（統辞論（syntax））もまたいくつかにわける必要があるだろうが、どのようにわければよいかわからないので、それについてはのべない。

A. 形態論

形態論であきらかにすべきことは、算語を自立的体系としてみた場合には、おもに形態素がどのようにくみあわされて変数名や手続きよびだしなどを構成するかということである。算語を人語に従属する体系としてみれば、人語の語や文がどのような規則で算語にとりいれられるか、という§2.3.4.でのべた問題にたいするこたえをみいだすことである。また§2.6.4.でのべた命名の問題（の文法的な面）をあつかうことでもある。形態論であつかう規則はほとんど非成文化規則であり、したがってこの部門は

これまでの文法論にはなかったものである.

さて、変数名やよびだしなどの算語内の構造をあきらかにするための方法としては「換入」がつかえる[「換入」については §2.5.1.Fを参照]. ただし、この方法をつかうためにはまとまった量のおなじ方言でかかれたプログラムをあつめる必要があるだろう. このような方法を中心とする形態論をかりに「換入形態論」とよぶことにしよう.

これにたいして、変数名やよびだしなどの抽象を記述した人語の文や句が算語に変形されるときにつかわれる規則をあきらかにするための方法としては、対象となる変数や手続きなどの抽象をその仕様書などからみいだし、それと算語での表現とをくらべるという方法がつかわれるだろう. このような方法を中心とする形態論を「変形形態論」とよぶことにしよう.

これら2つの形態論は相補的なもので あって、あい矛盾するものではない. プログラミング方法論には「変形形態論」のほうが示唆をあたえるものとおもわれるが、「換入形態論」のほうが方法のうえでよりたかい客観性を獲得できるであろうから、「変形

形態論」でみいだされる構造は、「挿入形態論」でみいだされる構造にあっているかどうかで検証されるであろう。

B. 統辞論

統辞論は、第2章でとりあげた問題のうちでは、つぎのような問題をあつかう。まず、§2.2.3.の意味の「規則」(すなわち操作的にみいだされる規則)にはどのようなものがあるかをあきらかにし、それらと規定書規則がどのようにくいちがうか[という§2.2.9.ですでに問題にした点]をあきらかにすること。それから、§2.2.8.でふれた規則の重要度とその数の関係の問題。§2.5.2.Cでのべたかっこ構造の問題も、人語との比較のまえに統辞論的な研究を必要としている。これらの問題をあつかうには統計的な方法が有効だろう。また、§2.6.1.でのべた構文のあいまい性をあつかうのも統辞論である。

これまでにあげなかった問題だが、句読法の問題も統辞論の一部だろう。たとえばAlgol系言語では";"をその直前の記号とのあいだに空白をおかずに打ち、つぎの記号とのあいだには(「スタイル」によって)ひとつまたはふたつの空白をおく(ま

たは改行する）ことがおおい．また＜，＞，≒などの記号の前後に空白をいれることがおおい．このような規則についてしらべることもテーマのひとつである．

そのほかにも統辞論で研究すべきことはおおいとおもわれるが，研究の視点がまだ確立していない．人語との比較をすすめればあらたな視点をえることができるだろう．

なお，以上は言語的な部分を中心にかんがえてきたが，非言語的部分についてはややべつのとりあつかいが必要であろう．

3.1.3. 意味論

§2.4.で"しめしたように、算語を自立的体系ととらえるかど"うかで"意味のとらえかたもかわってくる。

算語を自立的体系としてとらえれば"、§2.6.4.で"のべたようにプログ"ラムにあらわれる単位のおおくはせまい範囲でしか有効でないことが"おおいという点が"人語の場合とおおきくことなる。そのために意味論は人語の場合とはかなりちがったものになるであろう。

一方、算語を人語に従属した体系ととらえれば"どうだ"ろう。その場合「意味」は「単位」と人語で"かかれた「抽象」との関係と定義されたが、意味論がその「意味」を研究するものだ"とすれば"「変形形態論」は意味論の一分野ということになり、「挿入形態論」と「変形形態論」との対立関係は統辞論と意味論との対立関係だ"ったということになる。このようにかんが"えたときには算語の意味論は人語のそれとは質的にちが"ったものとなる。しかし、この方がいは算語の意味論が人語の意味論より客観的なもので"ありうることを意味しているといえる。

ところで"、プログ"ラムの文面だけから意味の構造をあきらかにしようとする立場(すなわち自立性を支持する立場)には"つぎの

ような（人語の意味論にも共通する）限界がある．それは単位とimplementationと抽象との関係が複雑なときには抽象の構造をとらえることがむずかしい．したがって意味そのものをとらえがたいということである[この困難はassertionsのはいっていないプログラムの検証をしようとする場合の困難とよくにている]．

かかれたものとしての仕様をつかえば，意味をあきらかにするために単位と抽象の両面からせめることができるぶんだけ有利である．実際に，プログラムは（単位にしろimplementationにしろ）抽象をよく反映していない（単純な関係でない）ことがおおいとおもわれる．たとえば変数xとyのあたいを交換(swap)するのに，おおくの言語ではべつの変数tをつかって

t:=x; x:=y; y:=t;

のようにしなければならない．このプログラム部分と

t:=y; x:=y; y:=t;

とは表面的にはにているが，意味はまったくことなるのである．

意味論の研究テーマとしてはつぎのようなものがあげられる．

まず、形態素がもつ「語彙的意味」の分析がある。これをどのように分析したらよいかはまだよくわからないが、人語の言語学的意味論でいうところの「場」の概念が重要な意味をもってくるのではないかとおもわれる。「語彙的意味」の分析のひとつとして、§2.6.1.でのべた語彙レベルのあいまい性の問題がある。これについては第5章でのべる。

つぎに「文法的意味」の分析がある。これは文法論と対応するいくつかのレベルにわけられる。たとえば"形態論に対応するレベルでは、形態素の意義が名前やよびだしの意味とどのような関係にあるかをしらべるという仕事がある。

No

Date

3.2. 比較・対照研究

「比較言語学」ということばは人語については2つ(またはそれ以上)の同族の言語をきめられた方法で"比較する通時的研究"というせまい意味につかわれている。プログラミング言語学においても、言語学に敬意を表して「比較」ということばに同様の意味をあたえることにしよう。すなわち、「比較プログラミング言語学」とは2つ(またはそれ以上)の同族の算語の比較をおこなう分野とする。(比較の手順は人語の場合とはちがってくるだろう)。しかし、そのような分野は算語の場合、人語の場合ほど興味のあるものではないとおもわれる。その理由は、まず(「歴史プログラミング言語学」にも共通するが)算語の歴史はそれほどながくないので通時的研究の重要性は人語ほどたかくないとおもわれることである。また、成文化規則の面では算語どうしの関係は人語にくらべるとはるかにはっきりとわかっているということもある。

「比較プログラミング言語学」とはことなる方法で比較・対照をおこなう分野は、やはり言語学にならって「対照プログラミング言語学」とよぶことにしよう。これにはまず算語どうしの対照をおこなう分野がある。また、より広い意味では算語と人語との対照をおこなう

分野をさすものとする.（高橋秀俊[1978]はそれを『人間と機械の比較言語学』とよんでいる）. この対照の意義は§1.2.でのべたとおりである.

さらに§2.2.6.では算語が法体系とにたところがあるのではないかとのべたが. このような言語以外の体系との対照もえるところがあるだ

ろう．このように「対照プログラミング言語学」をひろくとらえると，それはもはや言語学的というよりは文化人類学的な研究といえる．

ずいぶん研究の範囲をひろげてしまったが，ここでもう一度狭義の対照プログラミング言語学にもどってみると，すでにそれですら困難にみちたものであることがわかる．2つの算語を対照するといっても，それらが非常にことなった構造をもっているとしたら，どうやってその対照をしたらよいのだろうか．たとえばAlgol 60とLispはどうすれば意味のある対照ができるだろうか．人語でも同族でない言語どうしの比較（対照）はあまりうまくいっていないが，算語でもおなじ問題があるとかんがえられるのである．しかし，そのように大上段にかまえずに，できるところから対照をしていくことにすれば，人語との対照でも，他の体系との対照でも，いまからできることはあるだろう．

第4章 形態論ことはじめ

形態論の対象である なまえ や よびだし はいくつかの記号からなっている。そこで §4.1. では まず 算語の記号を分類し、その構造をしらべることからはじめよう。§4.2. では そのうち 識別子について その構造をさらにしらべる。§4.3. では その結果をふまえて なまえ の構造をかんがえる。§4.4. では よびだし の構造をかんがえる。§4.3. および §4.4 にあたる部分では、本来なるば 系統的な方法による研究をおこなうべきだが、まだ その方法が確立されていないので、例をあげながら仮説をのべるにとどめる。§4.5. では識別子の縮約（本来は長いはずのものを みじかくすること）についてかんがえる。ここでも仮説をのべるにとどめる。［本章のとくに §4.3, §4.5. をかくにあたっては、TSINKY［1979］を参考にした］。

4.1. 記号の分類とその構造

算語につかわれる記号は形式上つぎの4種類に分けられる.〔これらのうち「識別子」,「リテラル」,「必要語」はこれまでも一般につかわれてきた語であるが、「必要符」はつかわれたことがない〕.

A. 識別子(identifiers)

B. リテラル (literals)

リテラルとは 10, 1.25, "abc" のようなものである.(負号つきの数を全体でひとつのリテラルとみるべきか、負号だけでひとつの記号とみるべきかは、(現時点では)はっきりしない).

C. 必要語 (key words)

必要語とは、区切り記号(delimiters)のうち第2次分節があるもののことである. おおくの言語で必要語は予約されている(すなわち規定書できめられた意味以外の意味をあたえてつかうことはできない). その場合には予約語とよばれているものが必要語とほぼ一致する.(この論文では そのような場合を想定して、これまで「予約語」ということばを定義せずに「必要語」のかわりにつかってきた).ただし、例外もある. §2.7.1. でのべたように Pascal の ".." は第2

次分節をもつ区切り記号なので"必要語である.

D. 必要符(key marks)

必要符は区切り記号のうち第2次分節がないもののことである.

例としては",","<="をあげることができる[§2.7.1.参照].

ここでは4種類の記号をはっきりとは定義しなかったが、それは、個々の算語についてそれらを厳密に定義することはやさしいが、なるべく広範囲の算語についてなりたつように定義するのはそれほどかんたんではないからである.

これら4種類の記号のうち、識別子は2.5.2.下でのべたように、さらにちいさな有意単位(形態素)で構成されていることがある). そして、その形態素は非有意単位である文字から構成されている. すなわち、2重に分節される. 次節以下ではほとんど"識別子だけをあつかう.

リテラルはそれを構成している文字が最小の有意単位である[§2.6.1.参照]. したがって第2次分節は存在しない. 必要語は通常1つの形態素であり、非有意単位に分節される. 必要符は1文字で構成されているか、あるいは2つ以上の有意単位で構成されている.

4.2. 識別子の分節

§2.5.2.F.で分節される識別子の例としてあげたIN_FILEなどはみな_をふくんでいる。このように識別子の分節をあらわすための文字をbreak characterという。break characterとしてはつぎのような文字がもちいられる。

A. 空白

たとえばFortranでは識別子の途中に任意に空白を挿入することができるので、[逆に識別子の前後に区切り文字をいれないこともできるが] 空白を分節をあらわすためにもちいることができる。しかし、実際にはほとんどつかわれていない。(Fortranの成文化規則では識別子に空白は含まれえないのだろう)。これには、空白を入れることがゆるされていることがしられていないのと、識別子のながさが6文字(空白はのぞく)に制限されていることが影響をあたえているのだろう。

実際に空白がみいだされるのはAlgol系の言語の場合である。

例: parameter and array, towers of hanoi

[和田[1978], いずれもPascalのプログラム識別子]

name of month, current state, completion possible

[Brinch Hansen [1973], p.42, p.54 および p.55(日), Pascalのかなり標準からはずれた方言でかかれたプログラムから. これらは型名, パラメタ名 および 関数名]

imaginary roots, double root, real roots

[島内 [1972], p.10. いずれも Algol 60 の名札名]

はじめの2つはPascalの例だが、Pascalの標準語では break character は存在しないとかんがえられる. Pascalの処理系もそれをゆるしていない. Algol 60の場合もほぼ同様である. しかし、いずれも規準言語や発表言語のレベルでは空白をふくむことを禁止していない. とくにAlgol 60では「空白や新しい行への移行のような印刷における体裁上の事柄は, 規準言語において意味を持たない」[Naur [1963], §2.3. 訳は米田, 野下 [1976] による]とはっきり規定している. [しかしながら、この規定が区切り記号についての説明のなかにあるために、空白を分節文字としてもちいることの根拠とするのには疑問もある].

しかし Pascalでは金物表現で分節文字をつかうことは のぞましくないこととされている. [Jensen, Wirth [1975] のReportの §14. には「blank, end of line および comment は separator と考える. [中略] identifier, number および 綴りの symbol [= 区切り記号] のなかに separator はあってはならない」とある[和田の訳による]].

B. _

たとえば PL/I や Ada では下線が break character としてつかわれる. Adaの例はすでにしめした. PL/I でも例をさがすのはやさしい.

C. -

英語のハイフンににているので - は break character に適しているといえるだろう. しかしおおくの算語では - は単項または2項の演算子としてつかわれ、しかもその前後に空白をおかずにつかえるようにしているので、break character としてもちいることができない. - が break character としてもちいられる算語(族)としては Cobol, Lisp などがある.

例: string-length [MIT Lisp Machine Lisp の関数]

D. 大文字

A, B., C. とおなじ意味での break character ではないが、各節の最初の文字を大文字にすることによって分節をあらわすことがある.

例: FileOfInteger, CopyFiles [Addyman[1980]]

分節される部分に break character がないこともある. break character をもたない算語の場合はもちろん、それをもつ場合でもみられる. たとえば C. の例にあげた string-length に対応する Maclisp の関数は stringlength である.

break character があるときには識別子がどこで分節されるかは容易にわかる(break character が分節をあらわすためだけにつかわれるという仮定のもとで). ただし、break character があるときでも、それがないところで分節される可能性があることをわすれてはならない. たとえば、Ada の INOUT_FILE は、§2.5.2.F. の例としてあげた IN_FILE, OUT_FILE などとくらべると、

IN OUT FILE

のように分節されるだろう.

break characterがないときには、(従来の算語にはないことがおおいが)、人語の場合のように換入によるか他の方法——おそらくは意味につよくたよった方法で"分節をみいだ"さなければならない。第3章では、前者のような方法による形態論を「換入形態論」、後者のような方法による形態論を「変形形態論」とよんだわけである。

換入形態論においては、客観化のため、で"きるかぎり分節をアルゴ"リズ"ム化することがのぞましい。そうすることによって、(おおくのプログ"ラムは計算機にかけられるかたちで存在するがゆえに)研究の省力化にもやくだつ。すなわち、プログ"ラムを計算機にかけて分節をおこない、あとで人間がみて不適切なところをなおすとともに、そのアルゴ"リズ"ムの限界をさぐ"るようにすれば"よい。(筆者はPascalのプログラムについてそのようなアルゴ"リズ"ムをみいだ"そうとしているところで"ある)。

変形形態論においては問題となる単位の意義をどのようにしてみいだ"すかが問題になる。そのための一般的に適用で"きる方法は存在しないだ"ろう。場合によって仕様書や注釈を、あるいは論文にあらわれたプログ"ラムの場合には本文をつかうことが"必要だ"ろう。そして、それらからどのようにして意義を抽出するかがおおきな問題となる。

No

Date

のかきことば

分節によってえられる形態素には人語(おもに英語)の単位をそのまま借用したものと、算語にだけみられるものとがある。ただし後者も人語の単位と関係がある(たとえば人語の単位の頭文字をとったもの)のが普通であり、なかには人語の語のつづりを縮約したことがはっきりわかるものもある。

さらにくわしく構造をしらべるためには識別子の種類に応じた考察が必要であろう。そこで、変数と手続き、関数とについてかんがえることにする。手続き、関数については第3章でのべたようによびだしを単位としてあつかう。あらかじめことわっておくが、これからのべるこれらの単位の構造の分析は、仕様書などの記述や摸入といった客観的な材料や方法をつかったものではなく、かなり主観的な方法によっており、理論ではなくて仮説である。また、ひとつの算語をとりあげたものではなくて、一般論であり、算語によっては以下にのべることが観察されないこともあろう。[なお、ある算語の形態論の完全な記述のためには変数とよびだし以外の種類のものについてもとりあつかわなければならないのは当然である]。

No 34

Date

4.3. 変数名の構造

変数名は いくつかの識別子と ".", 添字などから構成される. (この論文では Adaの用語法にしたがい、「名前(names)」を「識別子(identifiers)」と区別してもちいる). たとえば"Adaではつぎのようなものが変数名としてつかえる.

BOARD (M, J + 1)

STARS (1..15)

APPOINTMENT . DAY

[DoD[1980], P.4-2 ~ 4-5]

しかし、ここでは変数名が ひとつの変数識別子によって構成されている場合についてだけあつかう.

変数識別子には英語の名詞が借用されることがおおい.

A. そのなかでも名詞1語だけからなる変数識別子がおおい.

例: state, schedule, timer [Brinch Hansen [1973]]

(この種の例は容易にいくらでもみいだすことができるので、これ以上例をあげない).

B. 論理変数には形容詞や過去分詞がつかわれることもある.

例: busy [Hoare [1974]]

completed [Brinch Hansen [1973]]

C. 英語から(名詞単独でない)名詞句のかたちで借用されることもある. 英語で(break characterとして)空白がもちいられるところに空白以外のbreak characterがもちいられたり、break characterがもちいられない(すなわち空白がとりのぞかれている)ことがあるが、そのような場合もここにふくめることにしよう.

例: initial process, page table [Brinch Hansen [1973]]

STRING_LENGTH

[(Brinch Hansenらによる) Concurrent Pascal コンパイラ (の原始プログラム)]

D. 人語からもとのままのかたちでの借用がみとめられる場合として、以上のほかに英単語に接頭辞や接尾辞がついたものがある. 接頭辞や接尾辞としては変数の型とかかわりのあるものがおおい. たとえば" Fortranでは整数型の変数識別子にIやNをつけることがおおい.

例: ILEFT, IRIGHT [西村, 荻野 [1976]]

NEND, NUPPER, NLOWER, NSTEP, NLIMIT,

NSTART [西村[1977]]

Iは integer, Nは natural (natural number)の頭文字とかんがえられる. (Nには負にならないという意味がこめられていることがおおいのだろう). このような接頭辞をつけるおもな目的は変数識別子を暗黙の型宣言にしたがわせることにあるとかんがえられる. しかし、Fortranでも整数として宣言されているにもかかわらずIのついた変数もあるし、型をプログラマが定義できる言語でも、たとえばポインタ型の変数に接尾辞 p や ptr をつけるとか、集合型の変数に接尾辞 s や set をつける[Pascalの場合]とかすることがある.

例: INTER_PASS_PTR

[Concurrent Pascal コンパイラ]

LETTERS, DIGITS, CONSTANTS

[同上. ただし これらの -S が≪集合≫という意義をもっているのか≪複数≫という意義をもっているのかは あきらかでない.]

NILPTR, TEXTPTR [Pascal-P4 コンパイラ]

E. また、状態をあらわす論理変数には is をあたまにつけたかたちがもちいられることがある。is のあとには補語がくる。(したがって英語から動詞句のかたちで借用されたものとか人がえられる)。これは B. のより完全なかたちとみることができるだろう。

人語からのもとのままのかたちでの借用がみとめられない場合としては、うえでのべたかたちのいずれかを縮約したものととらえられるものから、識別子のつづりとその意義との関係がまったく恣意的(すなわち人語に依存していないとおもわれるさのまで)ある。

A.～E. のかたちの縮約形とおもわれる例:

DIAM [西村, 田中 [1976a], diameter の縮約形]

KDIAM [同上。K+diameter。Kの意義は不明だが、Fortran の例であって暗黙の宣言にしたがっているから、暗黙の型にあわせるのがひとつの目的であることはまちがいない]。

DIREC, DIST

[西村, 田中 [1976b], direction および distance の縮約形]

恣意的な例：

J, K, L　[Fortranで"どこにでも"みいだされるなまえ．たいがい整数型である．]

X, Y

縮約については§4.5.でのべる．

4.4. よびだしの構造

よびだしの構造のおおわくは 成文化規則によっている。そこで本節ではまず規定書でよびだしの構造がどのようにさだめられているかをみてから、検討にはいることにする.

4.4.1. 規定書上の構造

現在の算語のおおくは手続きおよび関数のよびだしをつぎのようなかたちにかく.

RIGHT_SHIFT

[DoD [1980], p.6-6. 引数のない手続きの例]

SEARCH_STRING (STRING, CURRENT_POSITION,

NEW_POSITION) [DoD [1980]]

[Adaの手続きよびだしには本来セミコロンがつくが、ここではほかの算語とあわせるためにそれをはぶいている].

ただし、算語によっては手続きよびだしが必要語CALLではじまるものもある。また、引数のない手続きや関数のよびだしのうしろに()すなわちからのかっこがつくことがある〔Adaでも関数の場合には()がつく〕。

Adaではさらにつぎのようなやたらも可能である。

TABLE_MANAGER.INSERT(E)

[DoD[1980]. この場合手続き名は2つの識別子をふくんでいるが、INSERTが手続き識別子である.]

PRINT_HEADER (PAGES => 128, HEADER => TITLE,
CENTER => TRUE) [DoD[1980]]

REORDER_KEYS (NUMBER_OF_ITEMS,
KEY_ARRAY => RESULT_TABLE) [同上].

ここで"PAGES, HEADER, CENTER, KEY_ARRAYは仮引数識別子である. このような仮引数識別子をつけたかたちの実引数をAdaでは名前付引数(named parameters)とよぶ. 名前付引数の場合は実引数の順序は仮引数の順序に一致している必要がない. 名前付でないときは順序が一致していなければ"ならないので"位置引数(positional parameters)という. [さらにこまかい規則についてはDoD[1980]を参照されたい].

これまでのおおくの算語には名前付引数はないが、JCL(Job Control Language)など"では名前付引数があるのがふつうなので"、(keyword parametersとよばれることがおおい)、むしろこれまで"の算語にそれがなかったことのほうが異常だとおもわれる.

4.4.2. 関数よびだしの構造

関数よびだしは式のなかにあらわれるため、手続きよびだしより簡潔であることをもとめられているとかんがえられる。したがって言語学的な興味は手続きよびだしにたいするほどつよくないから、さきにかんたんにあつかうことにしよう。

簡潔であることをもとめられているとはいっても、関数よびだしも2つ以上の形態素からなっているとかんがえられる場合がしばしばある。関数識別子としては英語の名詞，形容詞，過去分詞あるいはこれらを中心とする句がもちいられることがおおい。

例：factorial(n−1)　　[和田[1978]など]

empty(queue)　　[Brinch Hansen[1973]. p.326(0)]

all transactions completed(state)　　[同上，p.54(0)]

また、isやhasといった状態をあらわす動詞の3人称単数現在形があたまについたかたちもつかわれる。isがついた場合は、isの主格が引数としてみいだされることがおおい。したがって英語の完全な文と対応がつくことがおおい。

例：　IS_EMPTY?(NEW),　IS_SAME?(id, id1)

[Guttag[1977]. NEWはQueue型のあたいをかえす関数]

t.has(i)

[Hoare[1972]. tは整数の集合 とかんがえてよい.

t.has(i)は整数iがtにふくまれれば"true"そうでなければfalseをかえす. したがって、この場合hasの主格はt.]

形容詞,過去分詞が中心の場合は is … の isがはぶかれた場合とかんがえられる.

以上の3つの場合は、関数よびだしはそれが"最微する"文の真偽値をあたいとするのだとかんがえることができる. 名詞を中心とする場合は引数をふくめてひとつの名詞句をなし、それが"最微する"ものが関数のあたいになるのだとかんがえられる.

関数識別子には、変数とおなじく、型をあらわす接頭辞,接尾辞がつくことがある.

例: IFIX, ISIGN [Fortranの くみこみ関数]

また縮約形がもちいられることもあり、恣意的なつづりがもちいられる[たとえばF,G]こともおおい.

4.4.3. 手続きよびだしの構造

手続き識別子には動作をあらわす英語の動詞がつかわれることがおおい。そのなかでも動詞が単独であらわれることがおおい。

例: copy, send [Brinch Hansen[1973]].

また、(動詞単独ではない)動詞句がもちいられることもある。

例: complete transactions, initiate process[同上]

手続きには型がないので、型をあらわす接頭辞や接尾辞がつくことはない。

変数の場合とおなじく、上記のかたちが縮約されたかたちのものもある。手続き識別子のつづりが人語に依存していないことは、実用的なプログラムではまれである。ただし、算譜やその処理系の説明のためにかかれたプログラムにはP,Q,Rなどの恣意的な手続き識別子がつかわれることがある。

以下ではこれらのかたちのうち、動詞またはそれを縮約したものをふくむかたちだけをかんがえる。そのようなよびだしが引数をもつ場合には、その引数のそれぞれが識別子にふくまれる動詞のとる格のひとつとしてとらえられることがおおい。

例； enter(t, q)　　[Brinch Hansen [1973], p.69(8)]

← Enter t into q.

remove(t, q)　　[同上]

← Remove t from q.

しかも、うえの例のような場合、英語の構文にあうように前置詞をいれて"("や","をとってみると、うえにしめしたように完全な動詞句になる。そして、実際、よびだしのなかに前置詞があらわれることがある。

例； IS_IN(klist, id1)　　[Guttag [1977]]

SWITCH(FROM=>X, TO=>NEXT)

[DoD [1980], p.6-6]

ACTIVATE(X, AFTER=>Y)　　[同上, p.6-7]

TRUNCATE(FILE, TO=>MAX)

[これは よびだしとして実際にあらわれた例ではないが、Adaの標準入出力用packageにはTRUNCATEという手続きがあり、その仮引数にTOという識別子をもつものがある。それをnamed parameterでかくと このようになる。]

そこで、(このような)手続きよびだしは英語の動詞句(の表層構造)あるいは深層構造に変形をくわえてつくられたものではないかという仮説をたてることができる。算語には冠詞などがあらわれないことから、深層構造からの変形によってつくられているとかんがえるのが妥当とおもわれる。すでにのべたように手続き識別子は名詞などをふくむことがあるが、そのような場合はその識別子がふくむ動詞がとる格のうちの一部が識別子内にあらわれ、のこりが引数としてあらわれているとかんがえられる。

いまのべた、かなり明確なかたちの変形規則は、存在するとしても一部の算語方言(一部のプログラム)にしかみられないとおもわれるが、このような規則をより広範に成立させることによって、プログラムのかきやすさ、よみやすさを向上させることができるとおもわれる。(といっても、それはこのような規則をプログラマに意識的に適用させるのではなく、人語の規則とおなじく、プログラマの言語能力を利用して無意識的に、すなわちプログラマが真に意識的な努力をはらうべきことがらをじゃましないように、適用させようというのである。もちろん、そのようなことが可能かどうかはわからないが)。

そこで"さらに このような構造について かんが"えてみよう.

すで"にのべたように位置引数のときには引数の順をかえることはでき ないが、これは人語で"前置詞や後置詞(あるいは助詞)が ない場合 (孤立語の場合)に格順をかえられないのとおなじで"ある. また、名前付引数のときに引数の順をかえることが"で"きるのは、前置詞や後置詞が"ある場合(膠着語の場合)に格順を(場合によっては)かえられるのに にている. この場合 よび"だ"しにあらわれた仮引数識別子の役割は格を指定することにあるととらえられる.

さらに、英語の場合 格がしかるべき順序で"なら人で"いるときには前置詞を必要としない(つけない)が、格順をかえると前置詞を必要とする場合が"ある. §4.4.1.で"例示したAdaのよび"だ"しの構造は これにに<u>た</u>柔軟性をもっているといえよう. (ただ"し、同一の規則に支配されているわけで"はないから、英語の表層構造が自然にプログラムに変形で"きるとはかぎ"らない. うえにしめした SWITCHや TRUNCATEの例は自然に変形で"きた例といえよう).

例: Give him the book.

→ Give (he, the_book)

Give the book to him.

→ Give (of => the_book , to => he)

[of => なしのかたちは Ada では ゆるされない。なお、実は of は Ada では 予約されているので、仮引数識別子としてはつかえない。]

以上のような例をあげると、なるべく英語にちかいかたちによびだしをかくのがよいようにもおもわれてくるが、実際には、引数の順序として英語の格順に一致しないものが規則化していることがおおい。たとえば スタックに対する演算 push は

Push (Stack, Element)

のような語順でかかれるのがふつうだが、英語では

Push Element into Stack.

となる。ほかに ファイルとの入出力をおこなう GET や PUT [Pascal や Ada など] を同種の例としてあげることができる。

4.4.4. 手続きよびだしの構造にかんする工学的考察

従来のおおくの言語は名前付引数をゆるしていない. Algol 60をはじめおおくの言語では よびだしのなかに注釈をかくことができるから、それで名前つき引数をいくぶんまねることはできるが、あとでのべるように それはまったく不満足なものである. しかしながら、算語においては前置詞あるいは後置詞にあたるものが人語以上に必要なのではないだろうか. なぜなら、人語では動詞はあらかじめ存在していて、その言語の使用者ならばそのつかいかたをよくしっていることがおおいが、算語では手続きはかぎられた範囲のなかだけでもちいられることがおおく、それぞれについて格順すなわち引数の順序を記憶するのは余分の努力を要するとかんがえられるからである. [筆者の個人的体験だが、筆者はPascalをすでに4年以上、しかもほかの算語よりはるかにたかい頻度でつかっているにもかかわらず、いまだにpack, unpackというふたつの標準手続きの引数の順序をおぼえることができない. (これらは3つの引数をもつ手続きである). そのために それらをつかうたびに規定書あるいは それらをつかったプログラムを参照しなければならなくなっている].

前節で名前付引数は格をあらわすものだとのべたが、そうだとする

と仮引数識別子としては前置詞をつかうのがもっとも適当だとおもわれる。ふつうの言語では仮引数識別子として変数識別子と同様の構造をもったものがつかわれることがおおいが、これは仮引数識別子がよびだしのなかでつかわれないためだとおもわれる。しかし、変数識別子と同様の構造をした識別子を名前付引数にもつかうと、しばしばそれと実引数としてあらわれる識別子が同一であったり、よくにていたりすることになる。この冗長さは無益だとおもわれる。また、つぎのような場合には、名前付引数は人間にたいしては格を象徴しない。

```
COPY (F => F, G => G)
```

または

```
COPY (FILE1 => F, FILE2 => G).
```

ただし、

```
COPY (SOURCE => F, DESTINATION => G)
```

のようにかけば格を象徴することができる。しかし、これと

```
COPY (FROM => F, TO => G)
```

とでは後者のほうがはるかにかんたん明瞭である。

このように名前付き引数に前置詞をつかう場合に2つの問題が生じる.

ひとつは 英語の前置詞のいくつかが 手続き言語においては予約語になっていて、仮引数識別子としてはつかえないことである.[そのことをわすれて仮引数識別子としてつかおうとした例をあげることができる. ISOのPascal規格案にはっぎのような手続きの例があげられていた (すくなくともAddyman[1980]までは).

procedure CopyFiles(var from,to : FileOfInteger);

Pascalでは to を識別子としてつかうことはできないから、これはあきらかなあやまりである. なお、Pascalでは名前付き引数をつかうことはできないから、仮引数識別子として前置詞をつかうことはAdaにおけるほど意味のあることではない. それにもかかわらず前置詞がつかわれている点にも注目したい.].

もうひとつの問題は、仮引数識別子として前置詞をつかうと、仮引数は手続きのなかでは変数としてつかわれるわけだが、その変数名としては奇妙だということである. Adaならば手続きの宣言部で名前のつけかえをすることができるが、わずらわしい.

No

Date

例: procedure COPY (FROM, TO : in INOUT_FILE) is

I : INOUT_FILE renames FROM;

O : INOUT_FILE renames TO; ...

[これ以降 I, O はそれぞれ FROM, TO の同義語としてつかえる.]

これらふたつの問題を解決するには仮引数と前置詞(keyword)をわければよい. たとえば, うえのような手続きは

procedure COPY from I : in INOUT_FILE

to O : in INOUT_FILE is ...

のように宣言し,

COPY from F to G.

のようにつかうようにすればよい. もちろん

COPY to G from F.

とかくこともできる.

Algol 60 にはこのような目的でつかうことのできる特殊な形式の注釈がある. それをつかうと, うえの例はつぎのようにかくことができる. 宣言は

procedure COPY (I) to: (O);

のようにし、

COPY(F)to:(G)

のようにしてよぶことができる。(いずれもCOPYの直後に注釈をかくことはできない)。しかし、)to:(は注釈であるから、使用時にほかの注釈をかいて よみ手をまどわすこともできる。たとえば

COPY(F)from:(G).

また、引数の順序をいれかえることはできない。ほかの算語(族)でも、もっとエレガントでないかたちでこれにた注釈をかくことができるが、いずれも以上にのべたような理由から まったく不満足なものである。

[Pascalでこのような形式をまねようとすれば、つぎのようになるだろう。

procedure Copy ({from} I : FileOfInteger;
{to} O : FileOfInteger);

Copy({from} F, {to} G).].

以上のように 人語との関係を重視して 手続きよびだし(やその宣言)の構造をかんがえるならば、さらに前置記法 すなわち手続き名をよびだしのはじめにかく記法が よびだしの かたちとして適当かどうか

も検討する必要がある．Adaをふくむほとんどの算語では手続きにたいして前置記法しかゆるしていない．宣言的言語（declarative languages）もおおくは前置記法しかゆるしていないが、つぎのようなかたちをゆるしているものもある．

例：　(Henry is_alive)

(Jill_and_Jack are_parents_of Paul)

[Hewitt [1977] の算語Plasmaによる例．これらはふつうの意味の手続きよびだしではないが、is_alive, are_parents_of が手続き名，ほかの識別子が引数とかんがえることができる]．

このような形式を一般の算語にとりいれるべきかどうかが検討されるべきである．

4.5. 識別子の縮約

4.5.1. 識別子のながさの制約と対策

識別子のながさは いくつかの理由によってながさに制約をうける。もっともおおきな理由は算語上あるいは処理系の制約である。たとえばFortranでは識別子のながさは6字以下とさだめられている。また、このような制約のない算語の場合には、ながすぎる識別子は、かきやすさはもちろん、よみやすさもおこすことがあるということが理由となる。

この第2の理由が理由となるのは、算語においては ながい名前が伝達効率をおとす場合に そのことへの対策がまったく用意されていないか、貧弱だからである(人語においては、代名詞の使用とか、省略形の使用とかが対策となる)。したがって根本的な解決策をみいだすことが必要なのだが、それは いますぐにはできない。とりあえず かんがえられる対策としては つぎのようなものがある。

A. 人語から借用する語 として できるだけ みじかいものをえらぶ。

B. 借用すべき それぞれの語に略語を対応させる。

C. 縮約規則にしたがって縮約する。

B.とC.とのちがいは、B.はそれぞれの場合に応じたやりかたで略語をつくるのにたいして、C.はつねにおなじ規則にしたがって縮約形をつくることである。したがって、C.のほうが縮約されたかたちからもとのかたちを推定するのが容易である。だから、B.は使用頻度のとくにたかいものにたいする方法として適当であり、C.は使用頻度の比較的ひくいものにたいする方法として適当だとかんがえられる。(使用頻度のさらにひくいものについては、できることなら縮約はしないほうがよい)。

次節以降ではC.についてのべる。

4.5.2. 縮約規則

縮約規則はプログラムの詳細な分析によっておきらかにされるべきものであるが、つぎのようなものは比較的容易にみいだすことができる。

A. かさね字の省略

例: maprint [Interlispの関数. mapprintの変形とかんがえられる]

procall [三五ほか[1981], procedure call → proccall → procallのような変形によって生じたとかんがえられる]

B. to → 2

例: c2i, i2c, s2ac, ac2s, …

[Clu [Liskov [1977]] の標準型(char, string)の演算. character to integer → c2i, string to array[char] → s2ac などの変形によって生じたとかんがえられる]

なお人語では これににた for → 4 という変形の例が観察

されている（規則としてみいだされるとはいえないだろう）：

"Me. 4U" [= Me. For you. 米メイン州の広告から.

[田中ほか[1975], p. 159]]

これらの変形はあまり短縮にならないようにみえるが、c2i のc と 2, 2 と i が cti のことも, ti と i より明確に分離して いるという点で cti より利点があるといえるだろう.

C. 前置詞の省略

例: putbuf (buf, nsave)

[Kernighan, Plauger[1976], p. 44. putbuf は put ... into buf[fer] のようなかたちが もとに なっているとおもわれる]

より体系的な変数の縮約規則が Jackson[1967] によって定義されている. それは、英語で"変数の意義を記述したうえで"、

(1) そのなかの重要な語を3語まで短縮する.

(2) [語の] 最初の文字はつねに存在しなければならない.

(3) 子音は母音より重要である.

(4) 語のはじめの部分はおわりの部分より重要である.

(5) 全体で6~15字に短縮せよ.

この「規則」は規則としてはきわめてよわく、あまり具体的にどうすればよいかをおしえてくれない. より強い規則として (3),(4)のかわりに「各語の右から左へ母音をおとし、(2)に反しないかぎりすべての母音をおとしてもなおながすぎれば、右から左へ子音をおとす」というものがある[TSINKY[1979]]. しかし、この規則はきわめて人工的であり、不自然である.

この、Jacksonの規則およびそれを強化した規則にかわるべき規則は言語学的な研究のなかからみいだされるべきであろう.

No

Date

第5章　あいまい性の研究

あいまい性は意味論の重要な研究分野のひとつである．これまで人工言語にはあいまい性がなく、またないのがよいとおもわれてきた．しかしじつはあいまい性があるということは§2.6.1.でしめしたとおりである．また，あいまい性がないのがほんとうによいかどうかは疑問である．なぜなら、人語ではあいまい性が重要な役割をはたしているということが言語学によってしだいにあきらかにされてきたが、算語にもそれとにた状況があるとおもわれるからである．また、算語にはあいまい性がとりいれられる傾向があるようである．(もちろん、すべての種類のあいまい性がとりいれられつつあるわけではないが)．たとえば"Ada [DoD[1980]] はかなり大胆なoverloadingをとりいれている．

本章ではおもに語彙レベルのあいまい性についてのべるが、その内容はつぎのとおりである．§5.1.では「一般的意味」，多義性，同形性の関係についてのべる．§5.2.では多義性，同形性の起因についてかんがえる．§5.3.ではあいまい性はなぜ"解決されなければ"ならないか、どういうときには解決されなくてもよいかということについて、人間の場合と機械の場合とを区

No

Date

別しながらのべる. §5.4. ではそれらの解決のされかた、すなわち情況によって複数の意味のうちどれがえらばれるかという問題についてのべる. 最後に§5.5. では さきほどふれた あいまい性の存在意義についてのべることにしよう.

5.1. 一般的意味，多義性，同音性

英語のchildという語（語彙素）についてかんがえてみよう．childは≪子供≫という「一般的意味（generic sense）」をもった語だろうか（すなわち，ひとつの広い意義をもった語だろうか），あるいは≪男の子≫と≪女の子≫という2つの意義をもった多義語だろうか，それとも≪男の子≫という意義をもつ語と，おなじ音をもった≪女の子≫という意義をもつべつの語とがあるのだろうか．この場合は≪子供≫はちょうど≪男の子≫と≪女の子≫からなるので，childは≪子供≫という「一般的意味」をもつものとみなせる．すなわち，ある語の意義をあわせたものがちょうどその上位範ちゅうに一致していれば一般的意味をもつ語だとみなせるが，欠けた部分があれば多義語とみなされる．そして，直接の上位範ちゅうがないときには同音語とみなせるであろう．［「一般的意味」と多義性との関係については池上［1975］による］．

算語について同様のことをかんがえてみよう．たとえば"手続きよびだしPUT(x)"がすべての単純型の式xについてつかうことができ，またそれらが型をのぞいてみなおなじ仕様をみたすなら，PUTは一般的意味をもったものとみなせるだろう．もしINTEGER型とCHARACTER型だけにつかえるなら，（これら2つの型だけから

なる上位範ちゅうがないときには）多義性をもつものといえるだろう。
（§2.6.1. でのべた"+"の例も多義性の例である）。
また、変数Vに式eであらわされる値を代入する手続きよびだし

SET(V, e) と 式eの値を要素としてふくむ集合をつくる関数よ

びだし SET(e) は、直接の上位範ちゅうが存在しないので、つっ

りのおなじ べつの語（すなわち同形語）とかんがえるべきだろう。

5.2. 多義性，同形性の起因

算語で多義性，同形性がおこってくる原因としては、まず人語の影響がある．すなわち、人語において多義的あるいは同形的であった語が算語にとりいれられると、算語においても多義語あるいは同形語となりうる．

しかしながら、従来の算語では 変数はもちろん使用者定義の手続きや関数にもoverloadingがゆるされていない．このような場合には完全に同形になることはさけられる．そのため、類義語や類形語がうまれる．

例： const と konst

［Pascal P4コンパイラにおける類形語の例．

const はPascalでは予約語であるから 識別子としてはつかえず、そのためにkonstがつかわれた．］

overloadingがゆるされている場合には多義語や同形語が生じる．従来のおおくの算語でも、規定書に定義された手続き，関数，演算子には多義であるものがおおいが、Adaではつぎのようなかたちで多義語を宣言，使用することができる．［ここに しめした以外にも いろいろの かたちで宣言，使用することができる］．

例1: 手続きのoverloading [DoD[1980]]

```
procedure PUT(X:INTEGER);   -- 1.
procedure PUT(X:STRING);    -- 2.
PUT(28);  -- 1. の PUT.
PUT("no possible ambiguity here");  -- 2. のPUT.
```

例2: リテラルのoverloading

```
type NEW_INT is new INTEGER;
```

[この宣言がおこなわれると1, 128などの整数リテラルは INTEGER型としても、NEW_INT型としてもつかうことができる].

```
I: INTEGER;
NI: NEW_INT;
  ---
I := 1;   -- INTEGER型の1.
NI := 1;  -- NEW_INT型の1.
```

また、Adaでは同様にして同形語を宣言、使用することができる.

(規定書上多義性と同形性は区別されていない).

例: procedure SET(VAR: out T; VALUE: in T);

function SET(ELEMENT: T) return SET_OF_T;

SET_VAR: SET_OF_T;

SET(V, e);

SET_VAR := SET(e);

人語では品詞の与がいがあるため容易に区別できた同音語(または同形語)が算語に借用されてきとの品詞がわからなくなって区別しにくくなることがある. たとえば"手続き識別子に動詞句を借用する非成文化規則"が確立していれば"この例の手続きSETは容易にもと動詞のほうのSETだとわかるが、規則として確立されていなければ"あいまい性は解決されない.

同形性については人語の影響のほかに縮約が起因となるであろう. ただし、これもoverloadingがゆるされないときは類形語をうみだす.

例: compile > comp

compare > comp

人語では音韻変化によって同音語がうまれるが、縮約による同形語の発生はこれにたとえられよう。ただし、縮約は識別子の宣言のたびにおこなわれるのにたいして、音韻変化はすくなくとも数10年というスケールでおこるのがおおきなちがいである。

また、多義性については意味の特殊化が起因となる。算語における識別子は本来もっと広義につかわれるはずのものを特殊化された抽象とむすびつけてつかわれるのがふつうである。そのために多義性が生じることがおおい。§2.6.1.でしめした"+"の例も、このような原因から多義性が生じたものである。

overloadingが生じない場合でも、抽象がプログラム上に明確にあらわされていないためにあいまい性が生じる。すなわち、どのように意味が特殊化されているのかが容易にわからないことがおおい。たとえば"SORT"という手続きがあるとしよう。これが昇順の整列をするのか、降順の整列をするのかは手続きよびだしをみただけではわからない。この例の場合にはSORTの仕様をみるか、implementationをよくしらべるかすれば"昇順か降順かはわかるだろうが、注釈やプログラムの仕様をみても、そこには本来の抽象が完全にはかか

れていないことがおおいので、その場合にはあいまい性がのこる。(したがって、かかれた仕様を基本的3角形の頂点に位置づけるか人が考えかたには弱点があるといえる)。たとえば、仕様には例外条件(exception conditions)がすべてはかかれていないことがおおい。

いまのべた「意味の特殊化」による多義性の発生は、人語における「意味の特殊化」による多義性の発生(Ullmann(1962), p. 183以)と比較することができるだろう。人語における「意味の特殊化」とは、ある社会的環境のもとではある語に本来修飾語句をつけていうべきときにそれをつけずにいうことがあり、これが固定化すると意味の特殊化がおこるということである。

5.3. あいまい性の解決の必要性

まず、あいまい性の「解決」および「本質的にあいまい」ということばを定義しておこう。多義性の場合には複数の意義のうち問題の単位がおかれている情況に適合するただひとつの意義をみいだすことを解決とよぶことにする。また、同形性の場合には、同形の単位のうちの情況に適合するただひとつの単位をみいだすことを解決とよぶことにする。いずれも情況によって解決がおこなわれえない、すなわち2つ以上の意義が排除できずにのこってしまう場合を、本質的にあいまいであるということにする。

あいまい性は放置しておけば算語による伝達を困難にしてしまう。機械への伝達の場合には、ある単位がどのimplementationをあらわしているかがわからないと翻訳不可能になってしまう。人間への伝達の場合には、ある単位がどのような抽象を象徴しているのかがわからなければならない。(デバックのような場合にはimplementationがわかることも必要である)。[これでわかるように、人間にとってと機械にとってとでは、ことなる「あいまい性」が問題になる]。

そのために、機械の立場からは(あるいはデバックのような場合には)ただちに実行可能なプログラムにおいては単位とその

implementationとの関係は 情況をかんがえあわせたうえで"完全に1対1または多対1にならなければならない。また、人間の立場からは単位と抽象との関係も情況をかんがえあわせたうえでほぼ"1対1または多対1にならなければならない.(ただし、抽象はふつう機械には無縁のものであり、人間は機械ほど厳密さを要求しないから、多少のあいまい性がのこっても問題は生じないだろう)。しかも、いずれについても人間に理解可能とであるためには、解決のために要する操作があまり複雑であってはならないだろう。

第1の要請によって たとえば x, y が実数のときの x+y における"+"と, x, y が整数のときの x+y における"+"とは(処理系によって)明確に区別することができなければならない。(しかし、誤差や例外条件が問題にならないあいだは、人間にとっては2つの"+"を区別する必要はないだろう。すなわち多義性がのこっていてもかまわないだろう)。

第2の要請によって たとえば

t:=x; x:=y; y:=t;

のようなプログラム部分[ふつうの意味での「単位」をなすものとは

いえないだろうが、基本的3角形の左下におかれるべきものである]。と≪変数xとyの値の交換をする≫という抽象との関係が(人間にとって)多対1にならなければならない。[識別子もはべつの識別子と換入可能なので、この場合1対1ではありえない。すなわちうえのプログラム部分と同義な表現が存在する]。しかし、それが達せられなくても処理系にとってはなんら不都合ではない。

5.4. 多義性，同形性の解決

人語の場合、文脈によって区別することがむずかしい同音語はあいまい性の存在じたいをなくしてしまうというかたちで処理されることがおおい。すなわち、それらのうちのいずれか、あるいは両方が駆逐されることによって解決される。算語においても、これににた現象がある。すなわち、本来つかえるはずの語を、それとおなじつづりのべつの語が定義された範囲でしめだしてしまう、あるいはふつうの方法ではつかえなくしてしまう「隠蔽(hiding)」という現象である。

Pascal と Ada における隠蔽の例をあげよう。

例1; Pascalの例

```
procedure P(X:INTEGER);
  ---
  procedure Q;
    procedure P; begin ---1. end {P};
  begin ---2. end {Q};
```

[この例では 1. および 2. の部分では最初の行の P を使用することはいかなる方法によってもできない]。

例2: Adaの例

```
procedure O is
  procedure P(X: INTEGER);
  procedure Q(X: INTEGER);
  procedure R is
    procedure P;
    procedure Q(X: INTEGER);
    --- 1.
  end R;
  ---
end O;
```

[この例では 1. の部分で"外側のPは隠蔽されないので"ふつうにつかうことが"できる(overloadされる). 1.の部分で"外側のQは識別子1個のかたちで"は参照することが"できないが、O.Qというかたちで参照することは"できる].

人語においても算語においても、おおくの場合隠蔽よりはもっとおだやかな方法がつかわれる. すなわち、あいまい性の存在をいち

おうゆるしたうえで、それを情況、とくに文脈によって解決しようとする。

従来の算語(の規定書)においては、関数や演算子の場合 解決がその演算結果の型によらずにおこなわれることがおおかった。たとえば R を実数型, I を整数型の変数とするとき、

R := I/2;

の "/" は右辺の式の型が実数になるべきであるにもかかわらず《整数の除算》を意味することがおおかった。[Pascal では "/" はつねに《実数/実数の除算》を意味するので、ことなっている]。

しかし、Ada(の規定書)においては 関数や演算子の overloading はその演算結果を考慮して解決される。たとえば §5.2. の例2の代入文においては、左辺が INTEGER型か NEW_INT型かによって右辺がどちらの1であるかがきまる。[Ada のコンパイラにおいて overloading が解決される方法(まず bottom up に、つぎに top down に構文木をなめる)は示唆的である]。

多義性や同形性の存在がゆるされた場合に、それを解決するための(うえにのべたような)規定書上の規則と、人間がそれを解決する規則はちがっているであろう。後者がどのようなものか

ということはいまはわからない。しかし、それは人語における あいまい性の解決の規則とにているのではないかとおもわれる。そこで、つぎのようにAdaにおける解決の例と人語における解決の例とを対比してみると、両者のあいだには かなり共通点があるようにおもわれる。

たとえば英語の manという語は《人間》,《男》,《夫》, 《召使》,《矢》などの意味をもつが、men and womenという表現においては《男》の意味に一応きまる[池上[1975], p.133]. man and wife, masters and men, officers and menのような表現ではそれぞれべつの意味にきまる。これにたいし、Adaではたとえば§5.2.の例2のような宣言[くりかえすと、

```
type NEW_INT is new INTEGER;
I : INTEGER;
NI : NEW_INT;
```

]があるとき 1+NIという式においては1はNEW_INT型の1にきまり、1+Iという式においては1はINTEGER型の1にきまる。この2つの例についてはかなりよい対応関係がみられる。

No.

Date

いまの例のような対応が、人間があいまい性を解決する場合と処理系(規定書)があいまい性を解決する場合とのあいだにいつもみられるわけではないだろう。そうかんがえる第1の理由は人間の処理能力をかんがえると、あいまい性の解決のために処理系が使用する情報をすべて人間が利用するとはかんがえられないことである。第2の理由は逆に人間があいまい性を解決するためにつかうとかんがえられる非局所的な情報を処理系がすべて使用するわけではないということである。(§5.3.でのべたように、人間にとって解決することが必要なあいまい性が、処理系にとっては問題にならないこともあるということに注意されたい)。

また、人語と算語とでは人間がおこなうにしても解決のしかたにちがいもあるだろう。それらのことに注意する必要はあるが、解決のしかたをさらにくらべてみることは意義のあることだとおもわれる。

No.

Date

5.5. あいまい性の存在意義

本章の序で"あいまい性は算語においても重要な役割"をはたしているとおもわれるとのべた。本節ではそのことについてのべよう。

まず"同形性の意義についてかんた人にのべる。同形性にはあまり積極的な意義があるとはかんがえられない。しかし、算語が人語からの借用に強く依存していることをかんがえると、それを円滑におこなうためには同形性をみとめることは必要であろう。

つぎに多義性についてだが、この場合にももちろんいまのべた同形性についての意義とおなじ意義がある。また、§5.2.でのべた「意味の特殊化」によるoverloadingは、みとめられないと名前の衝突(name confliction)がしばしば"おこるということがある。そして、そのほかにより積極的な意義がある。

§5.3.で"処理系にとって解決しなければ"ならないoverloadingが"みな人間にとっても解決されなければ"ならないわけで"はないということをのべた。overloadingをゆるさないことにすると、人間にとっては区別する必要のないものを機械のつごうで"人間も区別してあつかわなければ"ならないことになる。また、多義語は、人間が区別する必要のないときには区別しないですみ、区別する必要があるときは文脈をか

No

Date

んがえて区別することができるという柔軟性をもっているといえるだろう.(もちろん、文脈によって区別できるようにかいておけば、という条件つきだが).

これらの意義は、算語に多義性,同形性が存在することを承認し、またAdaのようにそれを積極的にとりいれることを支持するものだとおもわれる.

No.

Date

むすび

この論文では プログラミング言語学という研究の分野をうちたてるために、より具体的にいえば プログラミング言語に言語学的方法をあてはめ、またプログラミング言語と自然言語との比較研究の基盤を確立するために、おもに プログラミング言語の言語学的なみかたを しめし、それと自然言語との共通点をみいだすことにちからをそそいできた。そして、うえの目的のためには両者の相違点をあきらかにすることも重要なのだが、そちらには充分に手がまわらなかった。また、研究の方向づけをある程度おこなったが、具体的な研究にふみこむことはできなかった。したがって、それらが今後の課題である。

言語学的方法以外の方法でも プログラミング言語を humanisticにとらえることはできるだろうし、言語学的なみかたや方法も、この論文で しめしたものは そのほんの一部にしかすぎないだろう。今後 プログラミング言語の humanisticな研究がさかんになることを 期待したい。

文献目録

Addyman, A.M. [1980],

A draft proposal for Pascal, SIGPLAN Notices 15:4, 1–66

[その旧版: Addyman, A.M. [1979],

A draft description of Pascal, Software-practice and

experience 9, 381–424]

Aho, A.V., Hopcroft, J.E., and Ullman, J.D. [1974],

The design and analysis of computer algorithms,

Addison-Wesley

[日本語版: 野崎昭弘, 野下浩平 訳代 [1977],

アルゴリズムの設計と解析 (全2巻), サイエンス社]

Brinch Hansen, Per [1973],

Operating system principles, Prentice Hall

[日本語版: 田中穂積 他訳 [1976],

オペレーティング・システムの原理, 近代科学社]

Chomsky, Norm [1975],

Reflections on Language, Pantheon Books, a division of Random House

[日本語版: 井上和子, 神尾昭雄, 西山佑司訳 [1979],

言語論, 大修館書店

Dijkstra, E.W. [1976],

A discipline of programming, Prentice-Hall

DoD [1980],

Reference manual for the Ada programming language,

Department of Defence, U.S.A.

[reprint: [1981],

プログラム言語Ada基準文法書, 共立出版]

藤村 靖 [1963],

言語研究と計算機, 言語生活 137, 14-22

Gleason, H. A. Jr. [1961],

An introduction to descriptive linguistics,

Holt, Rinehart and Winston, Inc.

[日本語訳: 竹林 滋, 横山一郎 訳 [1970],

記述言語学, 大修館書店]

Guttag, John [1977],

Abstract data types and the development of data structures,

CACM20:6, 396-404

Huwitt, Carl [1977],

Vewing control structures as patterns of passing messages,

Artificial Intelligence 8, 324-364

Hoare, C.A.R. [1972],

Proof of correctness of data representations,

Acta Informatica 1, 271-281

Hoare, C.A.R. [1974],

Monitors: an operating system structuring concept,

CACM 17:10, 549-557

IFIP-ICC [1966],

IFIP-ICC vocablary of information processing (first English language edition), North-Holland Publishing Co.

池上嘉彦 [1972],

言語記号と非言語記号, 言語 1:5, 33-41

池上嘉彦 [1975],

意味論, 大修館書店

Jackson, Michael [1967],

Datamation April 1967

Jensen, K., and Wirth, N. [1975],

Pascal user manual and report, Springer Verlag

[reportの部分の日本語訳: 和田英一訳 [1976],

プログラム言語Pascalの文法, bit 8:4, 341-366]

JIS [1976],

電子計算機プログラム用言語FORTRAN (水準7000),

JIS C 6201

Kernighan, B.W., and Plauger, P.J. [1974],
The elements of programming style, McGraw-Hill
[日本語版: 木村泉 訳 [1976],
プログラム書法, 共立出版]

Kowalski, R. [1974],
Predicate logic as programming language, Proc. IFIP Conference

Liskov, Barbara et. al. [1977],
Abstraction mechanisms in Clu, CACM 20:8, 564-576

Mounin, Georges [1968],
Clefs pour la Linguistique, Éditions Seghers
[日本語版: 福井芳男, 伊藤晃, 丸山圭三郎 訳 [1970],
言語学とは何か, 大修館書店]

Mounin, Georges [1970],
Introduction à la sémiologie, Les Éditions de Minuit
[日本語版: 福井芳男, 伊藤晃, 丸山圭三郎 訳 [1973],
記号学入門, 大修館書店]

Mounin, Georges [1970a],

言語学と記号学, in Mounin [1970], 83-95 (日本語版)

Mounin, Georges [1970b],

交通法規の記号学的研究, in Mounin [1970], 193-210

(日本語版)

Mounin, Georges [1970c],

言語学における コードの概念, in Mounin [1970], 96-108

(日本語版)

Mounin, Georges [1970d],

記号学における分節の観念についての若干の考察,

in Mounin [1970], 167-184

Nauer, P. ed. [1963],

Revised report on the algorithmic language Algol 60,

CACM 6:1, 1-17, and Numerische Mathematik 4,

420-453, and Computer Journal 5, 349-367

西村恕彦 [1977],

新Fortran演習 (11), bit 9:3, 230-237

西村恕彦，荻野綱男［1976］，

新Fortran演習⑥，bit 8:11，1034－1040

西村恕彦，田中裕一［1976a］，

新Fortran演習③，bit 8:7，618－622

西村恕彦，田中裕一［1976b］，

新Fortran演習④，bit 8:9，880－884

Sammet，J.E.［1969］，

Programming languages：history and fundamentals，

Prentice-Hall

［日本語版：竹下亨訳［1971］，

プログラミング言語ハンドブック，日本経営出版会］

島内剛一［1972］，

プログラム言語論 ―Algol 60からAlgol Nへ―，共立出版

Shneiderman，Ben［1980］，

Software psychology ― human factors in computer and

information systems，Winthrop Publishers，Inc.

Sussman, G. J. [1971],

Micro-Planner Reference Manual, MIT AI-Memo No. 203

高橋秀俊 [1978],

人間と機械の比較言語学, 言語生活324

田中春美 他 [1975],

言語学入門, 大修館書店

TSINKY [1979],

プログラミング・セミナー: 名前のつけ方,

bit 11:12, 1206-1210

TSINKY [1980],

プログラミング: セミナー; 凸凹プログラム, bit 12:13,

1838-1845

Ullmann, Stephen [1962],

Semantics : an introduction to the science of meaning,

Basil Blackwell & Mott Ltd.

[日本語版: 池上嘉彦 訳 [1967],

言語と意味, 大修館書店]

和田英一 [1978],

プログラム言語 Pascal ④, bit 10:5, 619-627

Weinberg, Gerald M. [1971],

The psychology of computer programming, Litton Educational Publishing, Inc.

Wijnhaarden, A. van, et al [1975],

Revised report on the algorithmic language Algol 68, Acta Informatica, 1-236

[Springer Verlag から単行本としても出版されている.]

Wirth, N. [1971],

The programming language Pascal, Acta Informatica 1, 35-63

Wulf, W.A., London, R.L., and Shaw, M. [1976],

An introduction to the construction and verification of Alphard programs, IEEE Trans. on Software Engineering, SE2:4, 253-265

米田信夫, 野下浩平 [1976],

Algol 60 講義, 共立出版

No.

Date

謝辞

自由な研究をさせていただいた和田英一先生，研究のためのヒントをおしえてくださった水谷静夫先生，高橋秀俊先生ほかのかたがたに感謝するとともに、時間上の、および私の知識のうえの制約からそれらのかたがたのおかんがえをあまりいかすことができなかったのを遺憾におもいます．また、荻野綱男氏には論文の草稿をよんで貴重なご意見をいただいたことを感謝します．