プログラミング言語が含む「自然言語らしさ」について

渡邊弘喜
武村知子

0. はじめに

未来のことはわからないが、少なくとも現時点では、人間と同等の主体が、コンピュータという機械装置において自律的に存在しているわけではない。一般的によく述べられる言い方で言えば、人間と同等の知性をもっているわけではない。それにもかかわらず、人間とコンピュータとの関係をめぐって、しばしばコンピュータの知性だとかそれに関する何かが問題になる。この問題はデジタルコンピュータ史の最初期から見られるものであるが、いわゆる人工知能（AI）関連技術の発展と浸透もあって、「コンピュータが人間の仕事を奪う」といったAIに対する脅威と不安という文脈の中で、近年大きな話題になっているように思われる。

本論では人間の知性が何であるのか、コンピュータは（人間と同等の）知性なるものをもちうるのか、を問うことはしない。それは私(1)にはわからないし、またわかるようになるともとても思えない（わかりたいと思うことがないわけではないが）。

むしろこうした現状の中でより興味深いのは、なぜ単なる機械装置であるコンピュータがあたかも主体的存在であるかのように扱われうるのか、そしてそのような言説やイメージが、一般的なレベルではそれなりに説得力をもちうるのはどうしてか、ということの方である。というのも、主体的存在が主体的存在として見えることは、主体的存在とは何かということを別にすれば不思議なことでもないが、主体的ではまったくないはずの存在が主体的存在として見えることはそれだけで不思議

（1）本文および大半の註は渡邊による。武村の付す註には（T）と記してこれを明示する。（T）

なことだからである。ただ、この問いに直接答えることも私には十分難しすぎることのように思われる。そこで、その足がかりとして、人間がコンピュータを使用するという営みの根底では何か生じているのか、その営みが成立する基盤とはどのようなものなのか、というところから、技術的な面を考慮しつつ始める。そうすることで、人間とコンピュータとの関係を、よりよく、すなわち機械あるいは技術的産物としてのコンピュータというあり方を無視することなく、人間の側から再考することができるのではないかと思われる。そしてそのことはまた、人間というものの思索的な営み、とりわけ言語的な営みというものの根底において、生じていることを考える上でも意味をもつだろうと思われる。

本論における問題の核は、人間の言語的な営みが何かしらの物事――具体的にはコンピュータとその使用――と関係を結ぶことができるということがどういうことかである。したがって、コンピュータの仕組みを説明することやプログラミングのノウハウの有用性を検討することではない。人間がコンピュータを使用するという営みの根底を考える上で、取り上げる題材にプログラミングの技術的な要素が含まれてはいるが、それらの要素はプログラミングという専門技術領域の中では特別なものでも高度なものでもない。本論は、技術的な観点としてはごく当然だと考えられていることに言及するだけである。むしろ本論が問題にしたいのは、まさにそうした、当然と考えられていることを当然なままに問題なく現に人間が扱うことができるときに、人間はコンピュータとどのような関係を構築しているのかということである。コンピュータサイエンスを応用した自然言語の探求であるとか、あるいは専門的な知識を用いてプログラミング言語やその設計思想などを分析するといったことは、これまでにもなされてきたことであるが、本論はそうした方向を目指すものではない。本論は、人間の思索的な営みの基盤が人間の言語（自然言語）であり、どのような領域であれ、差異はあれども言語の存在なしには成り立ちえないという仮定のもと、プログラミングという専門技術的な領域において生じている思索的プロセスを、人間の言語（自然言語）という基盤に引きつけて再構築することを試みるものである。

人間とコンピュータとの関係を考えるとき、しばしばコンピュータで動作している（アプリケーション）プログラムと人間との関係――つまりコンピュータプログラムが人間に対して何ができるのかということ――に焦点が当てられがちである。

その種のことも重要な問題ではあるが、そのような問題に焦点が当たるときには、その前段階であるコンピュータプログラムが作られる局面で取り結ばれることになる人間とコンピュータの関係は、（しばしばプログラムを使う局面とプログラムを作る局面が切り離されているがゆえに）省略されるか簡略化されてしまいやすい。あるいは、プログラム作成段階のことに焦点が当てられるにしても、ある程度専門的な技能を持った人間すなわちプログラマの実用的・実践的問題（であるかそうでなければ美的問題）という文脈に回収されてしまいがちである。確かにコンピュータは高度に専門的な知識と技術の上で機能しているものであるが、現代では必ずしも高度に専門的な知識と技術を習得した人間だけがコンピュータと関わるわけではない。コンピュータに関連する知識や技術を一から十まで完全に理解していなくとも、表面的には問題なくコンピュータを使用できているかのような事態も現にある。そのような事態においても、人間がコンピュータと当座関係を取り結べているのは、人間の言語的な基盤の上で何かしら理解しているからであるはずで、そうしたある種の神秘的な理解の根底にあるものを見つめることに本論の目論見がある。つまり、何が分からないまま（分かっていると言えるような状態でないまま）分かったことになりえているのか、そのようなときに人間の言語的な基盤ではどのようなことが生じているのか、である。この問いは、言語的な基盤において想定される主体のあり方とも、また表象的な存在としてのコンピュータのあり方とも密接に関わるものである。

1. 諸前提

自然言語(2)が何かしら具体的な言葉や文として成立する際には、その言葉や文において言語を発する主体が不可避的に想定される。人間は誰かが発したものではない言葉を想定することができない。それが自然言語として機能しているかぎり、実際に発語した実在の存在を知らなくとも、何者かが発語したという形式の上で自然言語の意味内容を読み取ってしまう。言語を出力する主体は実際に発語した実在

（2）この論は、人間が発語する言語とプログラミング言語との関係について考察をするものであるので、プログラミング言語との明示的な区別のために人間が発語する言語のほうをここでは自然言語と呼ぶことにする。

の存在とは関わりなく想定されるはずなので、そのような主体に名を付けるなら、仮構的な発語主体と呼べるだろう。書記言語で記された自然言語の領域ではしばしば書き手と呼ばれる主体だが、本論ではそのような主体も発語主体に含めることにする。本来、自然言語はそれを発語するのも、発語されたものを理解するのも、人間である。だから、自然言語で想定される発語主体は、つねに人間であるかのような主体として想定されてしまうものであるはずだ。

自然言語のみを考えるのであれば、その発語主体に関してそのことを想定するだけでまずもって問題ないように思われる。しかしながら、現代において「言語」と呼ばれるものは、自然言語だけではない。それらは人間が用いる言語とはまったく異なるといってもよいほどのものであるが、どのようなわけか「言語」という語が当てられているものがある。その中のひとつにコンピュータプログラムを作成する際に用いるプログラミング言語がある。この言語は、人間が発語するためのものではないので、人間的な主体のみを単純に想定してすますことはまずできない。しかし、プログラミング言語は、人間の自然言語とも大いに関係がある。プログラミング言語を用いて記述されたソースコードは、記述された命令を実行するコンピュータという主体とプログラムを作成する人間（プログラマ）という主体が、関わり合う場となっているからだ。

人間がコンピュータを使用することの根底で生じている何事かを記述するという、本論の試み(3)のために、コンピュータの使用全般を成立させるのに必要となるプロセスを「見なし」と呼ぶことにする。この語は、コンピュータの特定の技術的領域や人間がコンピュータと関わる直接的かつ表面的な局面を指すものではなく、人間においてコンピュータの使用をそもそも成立させるうえで根本的に機能している全般的要素を自然言語という基盤の上で包括的に示すものである(4)。この「見な

（3）この論文は、言語一般とその支持体の種々の様相に関心を抱く向きの読者を広く想定して書かれている。そのため、プログラミングに関する詳細な記述を含みつつも、プログラミングやコンピュータの仕組みに全くなじみのない読者にも理解されうるような記述がめざされており、専門的な記述とコンピュータ入門的な記述がないまぜにならざるをえない文章構成になっている。最適化（本文後段参照）およびオブジェクト化の工夫がより多く行われるべきでもあろうが、本稿においては、最も原理的なところで思考するために、基本的に最適化を無効にしている。（T）

（4）ここでの定義は、専門の分野においてはより厳密に形式化されていることどもとも

し」は、人間がコンピュータを使用できているときには、人間が自覚しているか否かにかかわらず、常に生じている。本来まったく別のもの同士を、一定の規則を定めて結びつけることで、あるものを別の何かであるかのように存在せしめる、というのが「見なし」という行為である。たとえば、コンピュータを使用して文章を読むとき、通常ディスプレイにテキスト（文字列）が表示されるが、そこで表示される文字は、コンピュータ内部でデータとして保存されているものをコンピュータが読み取った結果現れたものである。それらの文字は、コンピュータ内部のデータとしては、0と1という二値（ビット）を特定の決まりに従って並べたものとして保存されている。さらに言えば、0と1という二値は、電荷のあるなしや磁極の状態などとこの二値を結びつけることでコンピュータが識別できるようにしたものである。電荷のあるなしや磁極の状態を0と1という二値の数と見なし、0と1という二値の数を文字と見なすことで、コンピュータがディスプレイ上にテキスト（文字列）を適切に表示させてくれる。AをBと見なし、BをCと見なす、というときに、例えばAが実際に何であるかはさして問題にならない。一方ではAをBと見なし他方ではA′をBと見なすとして、AとA′がどれほど異なっているとしても、一定の規則に基づいてそれぞれをBと見なすのでさえあれば、この見なしは問題なく可能である。したがって、Aが電気の状態であれ、磁気の状態であれ、そこに記録されるデータの内容に差異は生じえないとされる。場合により磁気ディスクを用いたり半導体メモリを用いたりするのは、現在の記録密度や読み書きの性能、コストなどの要因によるに過ぎず、データそれ自体の本質的なあり方に関わることではない。もし物としてまったく別物の記録装置からそれぞれ読み取られたデータであるとしても、最終的に同じ情報をもつと見なされるのであれば（おおよその場合ビットの並びが同じであると見なされれば）、それらは同一のデータと見なされる。データの入出力はコンピュータの使用の根本的な基盤をなすものであるが、データはすべてこの「見なし」を経て存在できるようになる。「見なし」のプロセスとは、このようなコンピュータの使用が成立するまでの一連の流れを総称するものである。

関わってくることであるが、本論は自然言語という基盤の上でそこから離れずに考えるというのがひとつのスタンスであるので、「見なし」という平易な語をあえて当てる。

見なしのプロセスを成り立たせる一定の規則は、きわめて恣意的かつ便宜的なものである。実際のテキストデータにしろ、画像データや音声データにしろ、同じ種類のデータの中に様々なデータ形式があることが、そのことをよく示している。ただし、見なしのプロセスを成り立たせる一定の規則は、一度定められたあとでは安易には変更することができない類のものでもある。

人間がコンピュータを使用することに不可欠なプロセスとして見なしを考えるとき、単に映っている文字列だけではなく、文字列が映るに至るプロセスないしそのプロセスがもつ意味も含めた、言語を軸とした人間とコンピュータとの関係が問題となる。この見なしのプロセスでは、コンピュータが出力するもの――たとえばディスプレイの映像など――を人間が実物の何かのように見立てることだけが生じているのではない。最終的にはそのような局面になるが、ここで問題にする見なしは人間の認識だけで完結するものではない。たしかに、コンピュータが人間と関わりのないところから生じたり機能したりするものではない以上、人間の主体的な関わりが絡むことではある。しかし、人間が主体として行なう見なしを可能にするために、その前段階に人間という主体からいったん切り離された形で機能しているプロセスがある。すなわちコンピュータ上でプログラムが実行されるプロセスである。電荷のあるなしや磁極の状態を0と1という二値の数と見なすことや、0と1という二値の数を文字と見なすことは、人間が主体として行なっていることではない。それらのプロセスはたしかに人間（設計者・プログラマ）がコンピュータにそうさせるようにしているのではあるが、使用する人間（ユーザ）がそれらのプロセスに関してまったく無知・無自覚であったとしても、コンピュータによってその見なしは着実に行なわれてしまう。したがって、たとえば人間がある種の線状の模様を文字と見なすようには、それらのプロセスに人間が主体として関わっているわけではない。しかし反対に、人間からの操作なしにそれらの見なしが行なわれるものでもないという点で、コンピュータという存在を人間と同レベルの主体として考えることも難しい。見なしのプロセスは、人間が主体として行なうプロセスと人間という主体から切り離されたプロセスという、二つのプロセスを結びつける仕組みである。

この見なしのプロセスを考える上で大きな手がかりとなるのが、プログラミング言語の中にある自然言語（的な何か）である。

見なしのプロセス自体は、人間とコンピュータの関係が生じているところ全般で

常に起こっていることではあるが、ここではその複雑な様相を具体的な事例から検討するため、基本的にプログラミングの中で起こる見なしのプロセスを検討する(5)。プログラミングは、プログラムを作るという点で、単にプログラムを使用することと異なる特殊な意味合いをもつ営為ではあるが、それゆえにプログラムを使用する営為と複雑に絡み合っている。現在ではプログラムを使用することを通じてプログラムは作られる。そのうえ作成したプログラムを使用することもプログラミングという営為全体の中で不可欠な局面である。プログラミングはコンピュータを使用する営みの土台を築くための営みでありながら、それ自体がコンピュータを使用する営みでもある。

「言語」と呼ばれるものは、本来人間が声や文字において発語するもののことであるが、現在ではプログラミング言語という、コンピュータを使用する際に必要となるプログラムを作るための道具(6)のようなものに対しても「言語」という語が当てられている。プログラミング言語は自然言語とは多分に異なっているにもかかわらず、これを「言語」という語を含んだ呼称で呼んでいることは、プログラミン

（5）したがって、ここでの人間は原則としてプログラマということになる。ここでいうプログラマは、ソースコードを読み書きする者を全て含むものとし、プログラミングの能力や理解力の高低は問わないものとする。また、プログラミング言語におけるいくつかの側面を例として取り上げることになるが、プログラミングにおける特定の技術的要素に限った問題として取り上げるのではない。

　なお、プログラミングの技術的な側面に関しては、まったく知識のない読者も対象とするため、限られた領域にとどめている。歴史的には現在のようなプログラミングの状況が形成されるまで様々な変化があるが、そうした歴史的側面についても本論では取り上げない。見なしのプロセスという観点からは、人間とコンピュータ（そして自然言語とプログラミング）の関係に、最初期から現在まで通底するものがあると思われるが、その通底をどこまで明確に言いうるのかは今後の課題の一つである。

（6）これを「プログラムを作るため道具」と言ってしまってよいものなのかどうかは幾分ためらいがあったので「のようなもの」というぼかした言い回しにしたが、当座ほかに簡潔で平易な言い回しが見あたりそうにないので、そのように記述した次第である。このような留保を述べたのは、ここで考察する問題の終着点と関わってくるかもしれない、という不確かな予感があるからだが、それについては今後の課題の一つである。

グ言語で自然言語の文字が用いられるという事態を介して、人間とコンピュータとの関係に、そして自然言語とプログラミング言語との関係に潜在する、特有の事態を示唆しているように思われる。実際しばしばプログラミング言語では、自然言語の単語に見えるものが記述の中に多く含まれており、ソースコードを記述するプログラマ自身があえて意図的にそうすることも多い。

ソースコードの記述に含まれる自然言語の文字ないし単語は、コンピュータがプログラムを実行するという目的だけでなく、人間がそこからプログラムの意味内容を読み取るという目的にも寄与する。ソースコードの読み書きにおいては、プログラムを実行する主体と自然言語の意味内容を伝達する主体という、二つの側面が分ち難くない交ぜになった主体が、人間であるプログラマにおいてひとまず仮構される。その様相を以下に見ていく。

ここで仮構という語には二つの側面がある。ひとつは、コンピュータは少なくとも人間がもっているような主体性はもっていないはずだが、自然言語での語りにおいて人間は、自覚しているか否かにかかわらず、主体的な存在であるかのようにコンピュータを仮定してしまうということである。コンピュータは根本的には単なる機械装置にすぎないので、人間の側からコンピュータの電源スイッチを入れ、プログラムを作成してコンピュータにそれを実行させようとしなければ何もしない。しかしながら、コンピュータ上でプログラムが実行されるとき、本来人間が主体的に行なわなければ生じない出来事が、人間が主体的に行なっているわけでもなく生じうる。このようなとき、コンピュータが勝手にやってくれているように、人間からは見える。そう見えてしまうとき、すなわち人間にはコンピュータが主体性のある存在として見えてしまっているのである。人間がそのような様相を、あまり深く考えることなく自然言語で語るとき、「コンピュータが〇〇してくれる」というような語りになりがちである。そのような語りではないかたちで語ろうとすると、途端に大きな困難に直面する。本来、単なる機械装置（コンピュータ）には（少なくとも人間と同じ）主体性なるものはないはずであるが、語りの形式的な面では人間と同種の主体性を前提とした語りに自然となってしまう。

もうひとつは、プログラマはソースコードを書く（プログラミングする）ときに、つまり実際にはプログラムがまだ実行されていない段階で、人間の代わりに作業してくれる存在としてのコンピュータという主体を仮定するということである。プロ

グラムが作成される前に、コンピュータがその作成前のプログラムを実行することは当然不可能である。だが、プログラマはどのようなプログラムをコンピュータに「やってもらう」べきかソースコードを書く前に想像しなければならず、その際に想像上の存在としてコンピュータを仮定することになる。

自然言語において想定される発語主体の存在が、人間に言語の意味内容を受け取らせ、あるいはまた、その発語が受け手である人間に知覚させたものを何か別のものに見なさせる。自然言語では発語する主体が人間自身のみであることによって、この言語がもつ働きは人間という主体のそれと同一化している。しかしコンピュータを使用する営為、とりわけプログラミングにおいては、その働きは、人間がその一部をコンピュータに明け渡すかたちで実現される。そこでは自然言語の意味内容において現れる主体は、コンピュータというプログラムの実行主体とは切り離された形で存在する。そのように切り離されることで、プログラムの実行主体という、人間という主体とは本質的に相容れない主体がなしたことにも、人間という主体のための意味を見いだすことができるのである(7)。以上のことについて、関数の命名とその実装との関係から論じる。

2. ソースコード上の自然言語の文字ないし単語が読み取られることで仮構される主体

人間の視覚にとって自然言語の文字と見えるテキストデータが、ソースコードの記述（プログラミング）で一般的に用いられる。その場合、ソースコード上ではあらゆるものが等しく文字として示されるが、プログラミングでコンピュータに処理させるデータの内には、むろん、自然言語の文字や言葉としてそれを表示させるための文字列データも含まれている。そのような文字列データもまた、テキストデータであるソースコード上では、文字を用いて記述する。つまりコンピュータへの命

（7）本論はプログラミングにおいて生じている人間とコンピュータとの関係を考察の対象とする。よって、考察対象の一方である人間は主にプログラマである場合がほとんどであるが、本論で考察となる両者の関係の基盤は、プログラマが自覚的に認識しているようなことではおそらくない。むしろそれはほぼ無自覚なものとして生じることだからこそ、うまく機能するのだと考える。

令の記述も、コンピュータプログラムで表示させる自然言語の文字や言葉も、等しく文字として人間の目に知覚される。

```
1  #include <cstdlib>
2  #include <cstdio>
3
4  int main(int argc, char* argv[])
5  {
6      printf("print");
7
8      return 0;
9  }
```

ここで挙げたのは、ただ「print」などの特定の文字列を出力するだけのプログラムのソースコードで、いわゆる「Hello, world」プログラムと同じような類のプログラムである。6行目にある最初の「printf("print");」は「print」という文字列を表示するただそれだけのコードであるが、コード上に示されている「print」には2種類ある。一つが二重引用符で括られていない「printf」という文字列の中にある「print」であり、これはコンピュータへの命令の一部として存在している。もう一つが二重引用符で括られている中にある「print」という文字列であり、これはプログラムの実行に際してデータとして処理され、前者の「print」をその一部とする命令が実行された結果ディスプレイ上に表示されるべき文字列であって、こうした文字列はしばしばコンピュータが実行したプログラムの結果を人間が確認したり、人間がコンピュータにやってもらいたい作業を指示できるようにしたりするためにある。同じ文字として表示され、人間にとってディスプレイ上では同じ単語であるように見えたとしても、コードの読解においては両者は異なる役割を与えられたものとして読みとられる。表示形態という観点では文字という同じ位相にあるが、役割という観点では両者は異なる位相にある。

自然言語とプログラミング言語では言語の構造からして大きく異なることは、「printf」の直後に来る丸括弧の括りが――あるいはif文やfor文の直後に来る丸

括弧や中括弧の括りなどが――プログラミング言語の記述にとって文法上不可欠なものであることから明らかである。記号という書記言語特有のものが、プログラミング言語の文法および文の意味の確定には必要なのである。このブロックを示す記号による括りが文法上要請されるということは、これらの記号文字が付随的に用いられる自然言語と違って、プログラミング言語が本来的に書記言語であって音声言語ではないことを示している。自然言語は本来的に音声言語である。ここでいう音声言語とは、声に出して読めるかどうかという話ではなく（自然言語の文字や記号を用いるかぎり発語すること自体は不可能ではないだろうから）、言語が音声を基盤として機能する性質を持っていることである。プログラミング言語は、自然言語のようには音声を基盤とした上で機能するものではない(8)。

printf 関数における「print」や、あるいは if 文および for 文における「if」「for」といった文字列が、自然言語にある単語と同じ視覚的形状の文字だからといって、それが、該当する自然言語の特定の単語を示すためにあるとは単純に言い切れない。「print」や「if」「for」といった文字列が、自然言語の単語としての意味を視覚的に想像させる必要から選ばれているであろうことは容易に予想がつくことであるが、だとしてもそれらの文字列は、自然言語の文章にあるそれらと同じように、自然言語の単語としての機能を果たしているのではない。そもそも「if」や「for」は自然言語の単語とまったく同じ文字列でできているが、printf 関数はすべての文字列を一つの語として考えれば厳密には「printf」という自然言語にはない単語となっている。このことも、これらの、自然言語の単語であるように見える文字列が、単純に自然言語を構成する単語として存在しているわけではないことの現れだと言える。

ソースコードに記述される文字列の自然言語性は、恣意的かつ便宜的な要素である。しかし、プログラマが人間であるかぎり必要不可欠な要素でもある。自然言語の単語がどのように組み込まれるのかは、人間にとっての利便性によって決定される。ただし、どのような利便性が優先されるかは、個々のプログラミング言語によ

（8）プログラミング言語において、音声言語としての側面は必要とされても想定されてもいないのは、音声情報のみで（映像を表示するディスプレイなしで）プログラミングすることが、不可能ではないにせよきわめて困難であるという事態からも十分に考えられることである。

って異なっている。記述されたコードの意味のとりやすさを優先する場合もあれば、入力の簡便さや表記の簡素さが優先される場合もある。printf 関数の「f」は、それだけだと単なる文字としか言いようがないが、それは自然言語とまったく無関係でもない。というのも、この「f」は「formatted（書式付きの）」あるいは「format（書式）」といった語に由来するものだからである。この単なる「f」という文字は、それが単なる「f」でしかないという点で自然言語の単語そのものではないが、それが本来自然言語の単語に由来するものであろうという点で自然言語と関連している。単なる「f」は、それが由来しているとされる自然言語の単語の意味から大きく外れない程度の、おおよその意味合いを、プログラマに思い起こさせれば十分である。つまり、この「f」が「formatted」か「format」あるいは「formatting」かそれに類する意味合いを生じさせるものであって、たとえば「file」の「f」ではない、といったことである。しかし fprintf 関数の「fprintf」という文字列において、頭文字の「f」は「file」であって「formatted」ではない（なお最後の「f」は printf 関数と同様である）という程度には、これらの単なる「f」は暗黙的かつ確固とした意味合いを持っている。

むろんこの事例は多分に C や C++ といった言語の個別具体的な事情に依存しているので、C/C++ と同じくらい省略の命名体系を好む言語もあれば、可能なかぎり省略をせず自然言語の単語をそのまま用いる命名体系の言語もある。しかしここで問題にしていることは、C/C++ といった個別の言語固有のものではない。どのようなプログラミング言語であれ、コードのいかなる符号も、コンピュータの機械語との関係を取り結ぶ上で、便宜上恣意的に決められている。この決定は元になった自然言語にある取り決めに依拠してなされなければいけないものではない。プログラミング言語の記述における（元になった自然言語の単語からの）省略は特定の規則に基づいてはいるが、自然言語（英語）における略語の生成ルールに則っているわけでもない。たとえば strcpy 関数の「str」や「cpy」といった文字列は、それぞれ「string」と「copy」を意味するが、こうした省略のあり方は元になった自然言語のルールに基づいているとは限らない。プログラミング言語の記述は、このような省略をしてもよい程度には自然言語の言語体系から独立している。しかし、元になった自然言語の単語が何であるかが判明していなければならない程度には、自然言語が個々の単語の中に築き上げてきた意味は、プログラミング言語において

も変質することなく必要とされている。

プログラミング言語と自然言語の意味との関連でもうひとつ押さえておく必要があるのは、ソースコードとプログラムとの関係である。プログラムは一般的に、ソースコードの記述に基づいて生成される。両者は互いに密接な関係にあるが、まったく別様な存在である。ソースコードに記述されていることは、平たく言えばコンピュータへの命令あるいは指示であるが、プログラムそのものでもない。ソースコードはデータ形式としてはテキストデータ、つまり単なる文字情報の集まりでしかない。このままではコンピュータ（の CPU）が命令を読み取ることができないため、プログラムの実行に際してソースコードはコンパイラなどによって最終的にはプログラムへ変換（コンパイル）される必要がある(9)。ソースコードがプログラミングにおける最終出力物ではないという点で、書かれたテクストそれ自体が最終出力物となる自然言語とは異なる様相にある。このような様相を考えるにあたって、しばしば人間の視覚にとらえられるためにディスプレイに出力される文字列（コンピュータ内部でデータとして扱われる文字列）がソースコード上で記述されている、そのあり様のよくあるパターンを一度見てみることにする。

```
1  #include <cstdlib>
2  #include <string>
3  #include <iostream>
4
5  int main(int argc, char* argv[])
6  {
```

（9）言い換えれば、ソースコードがプログラムと見なされることで、プログラムが実行可能になる。と同時に、プログラムが実行可能になることでプログラムを書く行為（プログラミング）が可能となる。なお、現在では用いられるプログラミング言語によって、事前コンパイルによって機械語（本論でのプログラム）に変換される場合だけでなく、機械語ではない中間言語を介する場合や、ほかにもインタプリタ方式・JIT 方式など様々な場合があり、ソースコードとプログラムの関係や距離感といったものにはそれぞれの場合で細かな差異はある。これらの差異は本論では直接的には重要でないので、単純化している。

```
 7      std::string stringData{};
 8      std::cout << "文字を入力してください。" <<std::endl;
 9      std::cin >> stringData;
10
11      std::cout <<"入力文字は「" <<stringData <<"」です。"
12                <<std::endl;
13
14      return 0;
15  }
```

「cout」はprintf関数のようにディスプレイに出力（標準出力）するための命令(10)をあらわす文字列であり、「c」が「console」、「out」が「output」という意味合いを持っている。「cin」は反対に入力（標準入力）するための命令であり、「c」は同様に「console」、「in」が「input」という意味合いになる。「endl」は「end」と「line」に由来するもので、改行（厳密にはそれに加えてバッファのクリア）を示す。「std」は「standard」に由来するC++で予め用いられている名前空間であるが、本論とは直接に関係がないので詳細は省略する。

先のソースコードとの違いは、このソースコードから生成されたプログラムの実行結果がユーザ（プログラムの使用者）の操作に依存するという点にある。最初の文字列出力である8行目の「文字を入力してください。」は、ソースコード上にある記述それ自体の視覚的様相において通常の自然言語の文章となっている。それは作成されたプログラムが実際に実行されてディスプレイ上に表示されたときにも、

(10) 一種のオブジェクトであるので、命令と呼ぶのは文法上厳密には正確であるとは言いがたい。文法上命令と呼ぶのにふさわしいものを強いて挙げるならばシフト演算子「<<」であるが、この演算子が示す命令・指示は、演算子のオーバーロード（多重定義）によって、ビット列をシフトさせるという本来の意味からは大きく変わっている。ここではそうした詳細な情報は不要と思われるし、また「<<」がシフト演算子であるにもかかわらずプログラマが「コンソールへ出力する」という意味合いを混乱をきたすことなく（この点には異論があるかもしれないが）読み取ることができるのは「cout」という文字列がすぐ隣にあるからだと考えるので、単に命令とだけ表記した。「cin」に関しても同様である。

人間にはそのように見えるべきものである。しかし 11 行目では、二重引用符で括られている文字列は、少なくともソースコード上では自然言語の文章の体をなしていない。自然言語の単語ではあるが、それは自然言語としては不完全である。「**入力文字は「」**という文字列が出力されるわけでも、また「**入力文字は「stringData」です。**」と出力されるわけでもない（「stringData」は変数名であって、出力される文字列そのものではない）。ここで本当に想定されている文字列の出力は、プログラムの実行時にユーザである人間が入力した文字列と組み合わされてできるであろう自然言語の文章である。たとえば、ユーザが「こんにちは」という文字列を入力したのならば、「入力文字は「こんにちは」です。」という自然言語の文章としてディスプレイ上に表示される。このような自然言語の文章は、このソースコードから生成されたプログラムが正しく実行されたときにしか生成されえない。

この事例は簡易なコードではあるが、ソースコードはプログラムそのものではないことを端的に示している。プログラマがソースコードを読み取ったことで把握するものを、（自然言語の読解になぞらえるとすれば）ソースコードの意味内容と呼べるだろう。ソースコードの意味内容とは、たとえば「printf」が、丸括弧の記号とセットで記述されているので、これは予め用意されている printf 関数であるとか、「if」という語があれば条件分岐を表わすコードであり、その右側にある丸括弧の記号で囲われているのが条件式であるといったようなことである。ごく単純化して言えば、記述されているコンピュータへの命令・指示の内容である。しかし、ソースコードは一般的に文字列データの集まりであるテキストデータであり、機械語で構成されているプログラムとは質的に異なるし、ソースコードで記述されているひとつの命令・指示が、ひとつの機械語と一対一できれいに対応しているわけでもない。そのため、ソースコードに記述されている命令・指示の内容は、プログラムの中に含まれているコンピュータへの命令・指示そのものではないという点で仮構的なものである。生成されたプログラムにもとづいて実行されるコンピュータの処理の総体を、（実際の）プログラムの実行内容とするのであれば、このようなソースコードの記述上で示されるかぎりにおいて立ち現れるのは、仮構的なプログラムの実行内容にすぎない。あくまで素朴に考えて、ソースコードの意味内容がコンピュータへの命令・指示であるなら、ソースコードの意味内容は、仮構的なものであれプログラムの実行内容と等しい（あるいはそうなるべきもの）と見なすことができ

る(11)。この見なしが成立することで、ソースコードという本来単なるテキストデータにすぎないものが、そこに書かれているとおりに実行されるプログラムのデータへと変換されていると、プログラマは信じられるのである。むろんこの見なしは、仮構的なプログラムの実行内容が、人間にとって問題が生じない程度には（実際の）プログラムの実行内容と同一であるという見なしの上で成立している。

printf 関数であれ「cout」であれ、ソースコードの意味内容が仮構的なプログラムの実行内容と等しいと見なすことができる局面では、それらの語から生じる意味作用によって、「文字列を表示させる」あるいは「文字列をディスプレイに出力する」というようなプログラムの実行内容を遂行する主体、すなわちプログラムの実行主体がプログラマにとって仮構的に想定される。

ソースコードから読み取られる意味内容に相応するものとして想定されるプログラムの実行内容が仮構的であることが、プログラムの実行主体としてのコンピュータと、コンピュータの使用主体である人間との関係にどのような影響を与えるのかを考えるにあたって、ソースコードとそこから生成されたプログラムの差異に関して、もう少し詳細に（より低水準(12)な範囲を含めて）検討する。通常のコンピュータ使用においてわざわざ気にとめる必要のない、プログラムというものの内実（つまりバイナリ表現）における種々の差異の存在を確かめることで、ソースコードが単にプログラムと異なるというだけではなく、本来的に異なるこの二つのものの間でプログラマが行なっている「見なし」がどのようなものであり、なぜ仮構的なプログラムの実行内容という側面が問題となるのか、よりよく知ることができるからである。

---

(11) ソースコードの意味内容とプログラムの実行内容との同一性はあくまで見なしにすぎないので、ほとんどの場合で問題はないものの、見なしがうまく成立しない（直感的にはしがたい）局面もある。たとえば、ソースコードからプログラムを生成するコンパイラのバグで単純に適切な機械語のコードに変換されないという場合や、コンパイラにバグがなくとも言語仕様上の未定義動作の場合や、あるいはマルチスレッドのプログラムでソースコード上の記述では想像しがたいような意図しない動作が引き起こされる場合などである。

(12) コンピュータの用語における「low level」のことであり、この場合よりコンピュータにおける物理層、ハードウェアの側面に近い水準ということを意味する。

```
 1  #include <cstdio>
 2
 3  int main(int argc, char* argv[])
 4  {
 5      int count = 0;
 6      if(count > 0) {
 7        ++count;
 8      } else {
 9        count = 1;
10    }
11    printf("%d", count);
12    return 0;
13  }
```

上記のような、ある種まったく意味をなさない処理内容があるプログラムのソースコードを例として、ソースコードの意味内容と実際のプログラムの実行内容との間にある隔たりについて話を進める。「ある種まったく意味をなさない」ことがどういうことなのかについては、あとでより具体的に言及することにするが、まずは人間にとっては、このプログラムがただディスプレイ上に数字の「1」を出力するだけのものでしかない、ということを述べておく。

初歩的な知識を有していれば自明のことであるが、5行目から10行目までの大部分は、正確には5行目と9行目を除いた部分は、プログラムにおいて何の役割も果たしていない。つまり、6行目からのif文はif文である必要性が何もない。別の言い方をすれば、9行目さえ残っていれば、ほかはなくともプログラムの実行結果は同一になる（7行目だけ残っていても実質そうであるが）。6行目のif文の条件式は、この場合いかなる実行時でも必ず偽になるからである。

このC++のソースコード(13)にもとづいて作られるプログラムでは、実行に際してどのような処理が行なわれるだろうか。プログラムのバイナリ、すなわち機械

(13) 実質的にはほとんどC言語のソースコードでもある。

語では読むことに（そして記載することにも）困難を伴うので、利便性のため慣例にならって、コンパイラ(14)が出力したアセンブリ言語のコードで代用する。アセンブリ言語は人間が読みやすくするために機械語をアルファベットと記号の文字列で置き換えたものなので、プログラムが実際にコンピュータ上でどのように実行されるのかを示していると言える。なお、すべてのアセンブリコードを記載するときわめて長くなってしまうので、必要最小限の箇所（C++ のソースコードにおける 5 行目から 11 行目あたり）のみ記載する。

```
; Line 5
    mov    DWORD PTR _count$[ebp], 0
; Line 6
    cmp    DWORD PTR _count$[ebp], 0
    je     SHORT $LN2@main
; Line 7
    mov    eax, DWORD PTR _count$[ebp]
    add    eax, 1
    mov    DWORD PTR _count$[ebp], eax
; Line 8
    jmp    SHORT $LN3@main
$LN2@main:
; Line 9
    mov    DWORD PTR _count$[ebp], 1
$LN3@main:
; Line 11
    mov    eax, DWORD PTR _count$[ebp]
    push   eax
    push   OFFSET ??_C@_02DPKJAMEF@?$CFd?$AA@
```

(14) ここでは Microsoft Visual Studio 2017 の Visual C++ コンパイラを用いて IA-32 のアセンブリを出力した。

```
    call    _printf
    add     esp, 8
; Line 12
    xor     eax, eax
```

C++ のソースコードに対応する行がコメントアウト（セミコロンで始まる行）で示されている。6 行目の if 文にあたるアセンブリのコードは「; Line 6」の下の行から「; Line 11」の直前までである。今回はアセンブリのコードに関して、詳しい説明は必要ないと思われるのでその点に関しては割愛するが、C++ のソースコードでは明示されていないレジスタの操作やジャンプ命令が見られる。if 文にあたるコードは、複数の機械語の命令で構成されているのであって、「if」という英単語において示される自然言語としての意味内容がプログラミング言語におけるコードとしての意味内容に直接対応していないように、「if」によって示されるプログラミング言語におけるコードとしての意味内容は、機械語の命令に直接対応しているわけではない（むろん直接対応していないといっても、それぞれは無関係にあるわけではないが）。

このアセンブリのコードはコンパイラの最適化(15)を無効にして出力させたものである。そのため、ソースコードでプログラムとして何の必要性もない箇所に該当する命令も、プログラムの動作の中に含まれていることがわかる。しかし、元の C++ のソースコードには何も修正を加えないままコンパイラの最適化を有効にすると、出力されるアセンブリコードの該当箇所は以下のように変化する。

```
; Line 11
    push    1
    push    OFFSET ??_C@_02DPKJAMEF@?$CFd?$AA@
    call    _printf
    add     esp, 8
```

(15) 生成されるプログラムをより無駄なく効率的なものにするための工程。多くはプログラムの実行速度の改善やプログラムサイズの圧縮に寄与する。

```
; Line 12
    xor     eax, eax
```

元のソースコードの5行目から10行目にあたる箇所は、最適化されたアセンブリのコードでは実質すべてないものにされている。代わりに最適化しない場合では、いったん0を置いてそのあとで1に置き換えている場所（スタック）に、そのまま1を置いている。なぜなら、ソースコードにおける「count」変数は結局常に1になるからである。条件によって処理が分岐しないことが明白なif文の部分は、行なわなくとも出力される結果は同一であるので、最適化した場合は、わざわざむだな処理に時間をかけて実行速度が遅くなるようなことはしないように変更されている。

このように、最適化がされていないプログラムと最適化済みのプログラムは、ともに同じソースコードをもとにして生成されるが、実際のプログラムの実行内容はそれぞれ明確に異なる。プログラムの実行内容という観点からすれば、両者のプログラムは同一では決してありえない。別の言い方をすれば、両プログラムのバイナリ（ビットの並び）には差異がある。しかし事実また、両プログラムは同一のソースコードから作られているのであり、ソースコードの同一性を基準にするのであれば、両プログラムは同一である(16)。

「ある種まったく意味をなさない」ということがどういうことかというと、それはその処理をしようがしまいがプログラムの動作上で何の違いも生まない、ということである。5行目から10行目までの部分のうち5行目と9行目を除いた部分のコードから意味内容をプログラマが読み取るかぎり、その意味内容から仮構的なプログラムの実行内容をプログラマは想定するが、実際のプログラムが動作している中ではそれに該当する実行内容があろうがながろうが、人間にはその違いを認識で

(16) むろんここでは、両プログラムの動作が人間の知覚では区別できず、したがって両者の実行結果が人間にとって同一であることを前提としている。なお、コンパイラもプログラムであり、本質的にバグを含むし、全知全能な振る舞いをしてくれる保証はないので、特定の状況下でコンパイラの最適化により、プログラムの実行結果が変わってきてしまうことは、実際ないわけではない（そしてそのような場合に生じるバグは、単純な状況でなければ発見は困難である）。

きない。そのプログラムが実行されたときに、人間が認識したプログラムの動作の様相や実行結果といったもの(17)から判断されるそのプログラムの全体像を、人間にとってのプログラムの意味内容と呼ぶのであれば、動作上何の違いも生まない仮構的なプログラムの実行内容は、このプログラムの意味内容に影響を与えない。最適化されていないプログラムとされているプログラムは、実際の実行内容は異なるが、意味内容からすれば同一だと言ってよい(18)。

とはいえ、実際の実行内容が異なる二つのプログラム間で、意味内容の同一性を保証してくれるものは実は何もない。なぜなら本来それ（意味内容）は、人間つまりプログラマがプログラムのバイナリを適切に読み取ったと確信できているかぎりで、当のプログラマに感知されるものでしかないからだ。しかし、プログラムのバイナリの解析に拠らず、プログラムの意味内容の同一性をずっと容易にプログラマに確信させうるものがあるとすれば、それはソースコードの意味同一性である。

上記では最適化の有無で例示したが、CPU のアーキテクチャが異なったりコンパイラが異なったりすれば、最適化の有無に違いがなくとも、バイナリもアセンブリのコードも様々に異なるわけで、同一のソースコードであれ、バイナリのデータ同一性は容易に崩れ去るものでしかない。そこに現実には様々な困難が伴うことを、それゆえに注意深くならなければならないことを、実際のプログラマはおそらく把握している。だが、完全には困難であっても、同一の仕様による同一の言語で書かれた、同一内容の（そして処理系依存の箇所のない）ソースコードは、プログラムの意味内容の同一性を示すものであることが望まれるはずである。

（17）ソフトウェア設計ではこのようなものに類するものが、しばしば UX（User Experience）と呼ばれる。

（18）最適化は、人間が体感できるかどうかは別として、実行速度に影響を与える。実行速度が異なるが常に同じ結果を出力するであろう二つのプログラムが、はたして人間にとって同じ意味をもつプログラムと言ってよいかどうかは、少しやっかいな問題である。というのも、同一バイナリのプログラムであっても、コンピュータのハードウェアスペックといったプログラムの実行環境が異なれば、実行速度は容易に変わりうるものだからである。
人間による知覚のされ方で、コンピュータプログラムの存在のあり方は大きく規定されるという意味で、このことはコンピュータプログラムが人間にとってきわめて映像的存在であることを示唆するのだろう。

そうした同一のソースコードにおけるプログラムの意味内容の同一性を前提としたとき、よりいっそう興味深くなるのが、上記の例に挙げたような、なかったことにされてしまう（一部の）コードの存在である(19)。仮にそれがコンパイル時の最適化で消されてしまうとしても、ソースコード上ではしかし間違いなく存在しているのである。そのようなコードは、人間であるプログラマが読んでしまいうるかぎり、無意味などではなく意味をもってしまうはずである。そのコードは馬鹿馬鹿しく役立たずで無価値かもしれないが、確かに意味内容はもっている。仮にあるプログラマが、無価値で馬鹿馬鹿しいからといってそのコードを削除してしまったとしても、そのような判断をすることができるのは、該当のコードの意味内容を読み取ることができたからのはずである。そして、ソースコードに（少なくとも文法上）適切な意味内容があるかぎり、そこに書かれたとおりのプログラムとしてコンピュータが実行するであろう物事が、つまり仮構的なプログラムの実行内容が、該当のコードにおいて示されているとプログラマは想定することになる。

ソースコードの記述上で仮構的なプログラムの実行内容が同一性をもっていることが、プログラマをして、実際の実行内容（バイナリ）が異なるはずの二つのプログラムを同一なものと見なさせる。言い換えれば、仮構的なプログラムの実行内容の同一性により、プログラマは、実際のプログラムの意味内容が両者とも同一であると見なす。この見なしが成立するには、仮構的なプログラムの実行内容からプログラマが把握するプログラムの意味内容と、実際のプログラムの実行内容からプログラマ（やユーザ）が把握するプログラムの意味内容が、一つのプログラム（およびそのソースコード）において等しくなければならない。少なくともそれが望まれていなければならない。ところが、それらが等しいことを証し立てるものは、等し

---

(19) なお、例示されたC++ソースコードから、11行目のprintf関数の呼び出し部分を削除して、最適化を有効にしたコンパイルを行なうと、該当箇所（5行目から11行目あたり）では以下のようなアセンブリが出力される。

```
; Line 12
    xor    eax, eax
```

これは本当にほぼ何もしていないプログラムである。そこには1という数すら存在していない。「; Line 12」があるのは、仕様上戻り値を渡さなければいけないからで、そのために戻り値に設定される特定レジスタ（EAX）の値が0になるような操作をする必要があるからにすぎない。

くないことを示すものがないのと同様、何もない。

ソースコードは、そこから実行可能なプログラムが生成されうることを基盤として十全にソースコードたりうるが、自身のかたちつまりテキストとしての有り様はその基盤によっては必ずしも一意に決定づけられるわけではない。ソースコードのかたちは、仮構的なプログラムの実行内容と意味内容とをプログラマが関連づける中で決定づけられる。実際のプログラムの実行内容は、人間にとってのプログラムの意味内容を変質させないかぎりにおいて変化してしまってもかまわない。しかしソースコードから読みとられる仮構的なプログラムの実行内容は、プログラムの意味内容と齟齬をきたすことがないものだとプログラマは見なすものである。そのように見なされないと、プログラマがコードを読み書きする中でプログラムを作り出すことは困難になる。なかったことにされてしまうようなコードの興味深さ、もっと踏み込んでいうのであればその存在の不穏さというのは、プログラミング言語によって記述されたソースコードがもつ、奇妙な、あるいは不完全な自律性にある。そしてその自律性を支えている(20)のが、ソースコードの中に含まれる自然言語の単語である。

「printf」にせよ「cout」にせよ、自然言語的な意味内容を含んではいるが、自然言語的な意味そのものを伝達するというよりは、自然言語的な意味内容によって、ソースコードの段階では現れえないプログラムの実行主体を仮構的に想定させるものである。その主体は、自然言語の発語主体として完全である必要はない。したがってその主体が発語すると見なされる言語(21)もまた、自然言語と結びつくことが

(20)「自律性を」「支えている」という時点で（確かにそう言いうるのであれば）それもまた十分不穏である。

(21) プログラムの仮構的な実行主体が行うと見なされ（う）る「発語」については、本文後段とその註にあるようになおその仕組みに関して考察すべき諸問題が多々あるが、例えば次のように理解するとよいかもしれない。「print（表示せよ）」という命令は、プログラマによって記述されるという局面においては、ごく普通の意味でプログラマによる発語に属すが、その記述がプログラムとして機能するだろうという見込み、すなわち「コンピュータが print してくれるだろう」という見込みの上では、記述に従ってエディタ画面に表示されるソースコードの中のこの「print」という命令は、その命令を「理解した」コンピュータが「はい、print します」と応答するそのいわば復唱のような「発語」とも「見なされ（う）る」。そうでなくとも事実、

可能であるような形であれば自然言語の言葉そのものでなくてもよい。他方、そのように自然言語と結びついているかぎり、その主体はプログラム実行時に機械語を実行するとされるコンピュータという主体とも厳密には異なる。

ソースコードは実際に実行されうるものであれば、実際に実行されることはないとしてもよいのである。そしてそこに、プログラミング言語が自然言語の単語を用いて自身を構築することの意味があるように思われる。すなわち自然言語の意味内容を喚起することで生じる、ソースコード（およびプログラム）読解の最適化という働きである。この働きにおいて重要な意味をもつのが、プログラムの実行内容において想定されるプログラムの実行主体と、自然言語において不可避的に想定されざるをえない主体すなわち発語主体との関係である。

ソースコードとプログラムが異なるあり方をしていることは、ソースコードを記述することに関与している主体と、プログラムを実行するという事態において想定される主体が異なっていることを示唆する。（実際の）プログラムを実行するという営みの内実は、決して人間が主体的に行ないうるものでもない（またその必要もない）。数式の計算にしろ、特定の文字列の検索にしろ、プログラムを実行することでそれらを行なうとき、実際にその作業をしているのは人間ではない。自然言語の言葉で簡潔に述べるのであれば、やはりそれはコンピュータと言うほかないものである。確かにコンピュータはただの機械装置であり、それがプログラムの実行の中で行なうべきことは、究極的にはコンピュータではなく人間が決定しているという点では、プログラムの実行におけるコンピュータのあり方に主体性は見られない。しかし、プログラミングができない多くの人々――つまりコンピュータがプログラムの実行の中で行なうべきことをすべて決定しているわけでも把握しているわけでもない人々――もまた、コンピュータを必要十分に使用している。そうした人々がコンピュータを使用するとき、プログラムの作成過程で存在していた人間（プログラマ）の主体性は、その時点で失われているに等しいはずである。「コンピュータが〇〇してくれる」というような物言い、ひいては、事実コンピュータが何かをする、という事態は、まさにコンピュータ自身が主体的にプログラムを実行するかの

ソースコードを記述するそばから打鍵に従ってエディタ画面に刻々と表示される文字列は、打鍵により入力されたデータを刻々と処理するプロセスを経て「コンピュータが表示してくれている」のである。（T）

ようにコンピュータが動作していると、人間が見なしえたそのときに成立する、と言うべきものである。実際に多くの場合人間はそのように見なしているのである。ゆえに、プログラムの実行主体としては、コンピュータという仮構的な主体を想定するのがふさわしいと思われる。

プログラミング言語がそう呼ばれる言語としての力を発揮したと言いうるのは、プログラムが実行されたそのときであって、プログラミング言語が記述された瞬間ではないはずである。もし言語がその効力を発揮するその瞬間を、自然言語の言葉に基づいて発語と呼ぶのであれば、プログラミング言語の記述ではなくプログラムの実行こそが発語ということになる。とすればプログラミング言語において人が可能であるのは、記述ではあっても発語ではない。また仮にプログラムのソースコードに関して、人が自然言語を用いて何かしら発語したとしても、それは現象や操作内容の説明であってプログラムの実行ではない(22)。

データとして記述される文字列の正しさ、ないしふさわしさは、究極的にはすべて人間という自然言語の発語主体のそのつどの判断に依存するが、コンピュータへの命令としての文字列は、それが自然言語の単語そのままであれ、自然言語の単語としては不完全な形をしているものであれ、一度恣意的に決められたのならその通りに記述しなければならないものである。そこで生じる意味作用は、プログラムを実行する主体を人間であるプログラマが思い描くために必要とされているが、ソースコード上で想定されるその仮構的な主体は、決して現実のプログラムを実行することはできない。その仮構的な主体がなすソースコード上での発語――命令としてのコードに含まれる「`if`」であるとか「`print`」であるとか「`out`」であるとか――は、自然言語の文法にそって必要十分な単語を伴っておらず、決して人が原理的に直接には発語できないものを何らかの手段で形づくるために必要とされる。それら発語が意味あるかたちで実際になされる（プログラムが実行される）とき、その発語は、その仮構的な主体が存在するところのソースコードとは質的にまったく異なるところでなされる。数字の大小を比較したり加算をしたり文字列をディスプレイに出力したりするなど、何事もよしなにやってくれる（はずの）主体の発語の様相

(22) おそらく逆のことも言えるだろう。つまりコンピュータは自然言語を表記・表音することができても発語することはできない。

を、人間はほとんどまったく知覚できない。ただプログラムの実行結果という痕跡だけが（それすらない場合もあるが）この主体が何事かをなしたのだと人間に見なさせるにすぎず、また、仮構的なプログラムの実行内容と実際のプログラムの実行内容との同一性は少しも保証されていない。したがって、実際のプログラムの実行上で想定される主体が、ソースコード上の仮構的な主体の存在の様相を明瞭にしてくれることもほぼない。このような仮構的な主体において示されるのは、主体なき発語、としては言い過ぎであるなら、きわめて脆弱である主体の発語である。その者は仮構されることでしか存在できないというだけでなく、想像の中ですら人間にもコンピュータにもなりきれない主体である。

3. 自然言語的な意味内容とプログラムの実行主体との結びつきにおいて生じる主体性の部分的な明け渡し

プログラミングの中で、自然言語の単語が最も便宜的に利用されるのが、プログラマが行なう命名の作業である。コンピュータに行なわせる一連の命令の流れをひとまとめにしておくということが、プログラミングでは現在ごく当然に行なわれている。このようなことを実現するために、たとえば関数やサブルーチン、あるいはメソッドといった用語で呼ばれる構文が、プログラミング言語の言語仕様として予め用意されていることが多い。このような仕組みが言語仕様として用意されているのは、人間であるプログラマにとっての利便性や保守性のためである。しかしそうした利便性や保守性は、プログラマが同種のコードを繰り返し大量に記述しなくてもすむという、単なる作業の効率化をもたらすだけのものではない。人間であるプログラマのソースコード読解の様相とも深く関わっている。

以下ではこの仕組みの話を中心に人間とコンピュータとの関係を検討するが、ここではサンプルとして取り上げるコードで用いられている C++ という言語の用語にあわせて、「関数」と呼ぶことにする。興味深いのは、プログラミングにはしばしば命名という作業がつきまとうものであり、中でも関数の命名は最も気を遣うもののひとつだということである(23)。命名という作業には人間の自然言語が関わっ

(23) ただし関数の命名という作業はまた、前述の「printf」関数がそうであるように、

ている。変数に関しても命名の作業が必要とされ、関数における命名と同様の側面もあるが、ここでは主に関数の命名に焦点をあてる。一連の命令をまとめるという役割とそこで命名が必要とされることに、仮構的なプログラムにおける実行内容と意味内容との曖昧な関係が含まれているためである。

```
bool IsJapaneseCharacter(unsigned char firstByte)
{
    return (0x81 <= firstByte && firstByte <= 0x9F) ||
           (0xE0 <= firstByte && firstByte <= 0xEF);
}
```

以上のような関数が定義(24)されているとする。最初に記述されている「IsJapaneseCharacter」がいわゆる関数名である。中括弧で括られているコードが、いわゆる関数の実装と呼ばれるもので、該当の関数がプログラムで必要とされたときにコンピュータが実行すると想定される、処理の具体的内容（仮構的な実行内容）が記述される(25)。関数の実装として具体的に何が行なわれることになっているか、つまり該当関数における仮構的なプログラムの実行内容の詳細は、ここではそれほど問題ではないので説明は省く。ただし、関数の実装として示されている仮構的なプログラムの実行内容は、関数名に示されている自然言語的な意味内容を正確に実現するものではないことは、予め指摘しておく。関数名（およびその実装）は上記のようなコードが単に記述されるだけでは、生成されるプログラムにおいて意味をもたない。つくられた関数がソースコードの別の箇所で呼び出される（利用される）ことで、初めてその関数としてまとめられた処理内容がコンピュータによって実際に実行されることになる(26)。つくられた関数が呼び出されるときには、if 文や

プログラミングにおいてしばしばしないで済まされる（あらかじめ済まされている）ものでもある。

(24) C++ の規格に沿って厳密に言えば、関数の宣言（最初の行）と定義（2 行目以降の行）である。

(25) なお、処理の具体的内容を示すコードに加え、それを記述する作業のことをも実装と呼ぶことがある。

printf 関数(27)といった予め用意されている命令・指示であるコードと同様に記述される。

関数名には英単語が用いられているが、コンパイラが正しく識別さえできれば、どのような関数名であっても、コンピュータが実行するプログラムの内容としては差異がない。コンピュータにとっては「IsJapaneseCharacter」は、どのような関数に記述された処理を実行するのかを識別するための単なる符号でしかない。この関数の名前が「IsKanaAndKanji」でも「IsEnglishCharacter」でも「IsAlphabet」でもあるいは「IsMultibyteCharacter」であっても、プログラムが実際に実行される段階ではコンピュータは同じ処理を行なう。関数名が別のものに置き換わったとしても、一貫して同じ文字列で記述されていれば、プログラムの実行内容に違いは生じない。プログラムの実行主体としてのコンピュータに差異を生じさせるのは、関数の実装として記述されているコードだけである。

対して、人間であるプログラマにとって、たとえば英語など自然言語の文字と単語で関数名がつけられているかぎり、関数名の違いは意味内容の差異を生じさせるはずである。「IsJapaneseCharacter」が「IsKanaAndKanji」や「IsEnglishCharacter」や「IsAlphabet」、あるいは「IsMultibyteCharacter」であったりすれば、それらはそれぞれ程度の差はあるとしても異なる意味をもつ。

この関数の命名という作業も、コンピュータにおける見なしのプロセスと関わりがある。関数名は、人間であるプログラマが、ある一連の命令を関数という意味あるひとつのまとまりと見なす符号となるからである。コンピュータにおける二値などの見なしが恣意的かつ便宜的であるように、関数名もまた恣意的かつ便宜的なも

(26) 呼び出されることのない関数をつくることはできるが、そのような関数は、前述の「ある種まったく意味をなさない」コードと同様、プログラムの最適化の過程で消去、つまりなかったことにされてしまう。

(27) C/C++ では、printf 関数が“関数”と呼称されることに示されているように、言語の規格の一部として予め用意されているこれらの関数も、プログラマが自身の裁量で定義する関数と、言語構造上では本質的な違いがあるわけではない（むしろ if 文とは明確な違いがある）。しかし、予め用意されていることでプログラマに与える差異はたしかにあり、そのことについてはあとで言及する。いすれにせよ、ここではprintf 関数といった関数を、コンピュータへの命令という点で if 文と同じようなものとして見なしてもらってかまわない。

のであって、関数名に自然言語の単語を用いなければプログラムが作成できないわけではないが、用いることが強く推奨される。関数名が自然言語的な意味内容を含んだり連想させたりするものであることが、プログラミングにおいて大きな力を発揮するからである。自分が記述したり読解したりして理解したコードでなくとも（つまり他人が書いたコードであっても）、関数名が含む自然言語的な意味作用によって、該当の関数がプログラムにおいていかなる意味内容をもつのかを把握する（より正しく言えば把握したつもりになる）ことができる。そのため、すでにつくられた関数を呼び出すことで、そのコードがプログラムとして実行時に発揮する力の恩恵を容易に受けられる。文字コード（Shift_JIS）の体系についての知識がなくとも、すなわち「IsJapaneseCharacter」という関数における処理の具体的内容を覚えていなくとも、この関数名とその引数を覚えていれば、それを使って関数を呼び出し、ある文字が日本語特有の文字なのか（厳密に言えばShift_JISに特有の文字コードかどうか）を判定できるようになる。

例示された「IsJapaneseCharacter」という関数名は厳密には正確ではない。それはShift_JISの文字コードであることが暗黙的に前提とされているからである。つまり異なる文字コード、たとえばEUC-JPやUTF-8であれば、そこに「日本語の文字」あるいは「マルチバイト文字」と見なされるべきものがあったとしても、人間にとって適切に処理されたとは言いがたい事態が容易に生じる。根本的に見なしを通じてしか構築されえないコンピュータと人間との関係は、異なる文字コードの存在のように、人間による最終的な見なしに至るまでにほんの些細な齟齬が存在するだけで崩れてしまう。しかし関数名と関数の実装は便宜的な関係であるがゆえに、関係の修正という変化も容易に起こりうる。たとえば、もし異なる文字コードに対応したり、あるいは「日本語の文字」であるかどうかをより精密に判定したりする必要があるのであれば、そのためのコードの記述を「IsJapaneseCharacter」という関数の処理として付け加えれたり書き換えたりすればよい。そのとき「IsJapaneseCharacter」という関数を利用するだけのプログラマは、そのままその関数のコードがプログラムとして発揮するであろう力を変わらず享受できる。少なくともその場合、この関数を呼び出す記述をする人間は、関数名にある自然言語的な意味内容を読み取っていればよく、呼び出された関数を実行するコンピュータにとっては関数名が同じ文字列の符号として一貫してさえいればそれでよい。

定義済みの関数を呼び出すという段階では、どのような名前（と引数）であるのかが重要であり、その関数でどのような命令が実行されることになるかという点に関して、関数名が含む自然言語的な意味作用が適切に働いているかぎり、プログラマが十全に把握しておく必要はなくなる。この場合、関数名は、コードを記述するプログラマに対して、該当の関数が人間にとっていかなる意味をもつ関数なのかを提示する。すなわち、その名前がプログラマに対して示唆している自然言語的な意味内容を、プログラムの実行の中で実現させるための符号として機能する。ここに至って、関数名に含まれる自然言語的な意味内容は、関数において実行される仮構的（かつ部分的）なプログラムの実行内容から立ち現れるだろうプログラムの意味内容と同一なものと見なされることになる。

この場合、関数名（仮構的なプログラムの意味内容）と関数の実装（仮構的なプログラムの実行内容）は、本質的には恣意的で便宜的な関連に過ぎない。しかし、恣意的で便宜的な関連であったとしても、人間から見て結果的にその関数名にふさわしいかのように関数が機能するのであれば、プログラマが関数を利用する上では問題ではない。関数を利用する局面においては、関数名は、「if」や「for」といったプログラミング言語の言語仕様レベルで用意されている語と、プログラマがコードを書くという記述的な位相においては差異がなくなるとも言える(28)。つまり、仮構的なプログラムの意味内容を提示するはずの関数名は、関数を呼び出す記述においては仮構的なプログラムの実行内容と同化することになる。この同化のプロセスにおいて、ある関数が別の関数の中で――printf 関数が main 関数の中で呼び出されているように――プログラムの実行内容として埋め込まれるという事態が生じる。

本来的に書記言語であるプログラミング言語では、自然言語の文字だけでなく記号もまた、ソースコードの意味内容に、ひいては仮構的なプログラムの実行内容のあり方に、本質的なレベルで大きく関わるものである。それに対し、本来音声言語である自然言語では、記号において発語主体は現れえない。記号は書記言語で表わす際に文の構造を分かりやすく示すための補助的なものでしかない。そのため、ソ

(28) この二つにおける異質な点と同質な点は、抽象化を経ることによる意味的レベルの階層性と記述レベルでの同質性という観点からより深く考察したいと考えているが、見なしとのつながりも含めて今後の課題である。

ースコードにおいて自然言語的な発語主体が現れうるのは、関数名である。引数の変数名ももちろん人間であるプログラマの読みやすさにとって重要なのは間違いないが、仮引数と実引数(29)で変数名が異なってもよい。仮構的なプログラムの実行内容をプログラマに伝達するという点に加え、定義された関数と呼び出される（使用される）関数との同一性を示すという点でも重要な意味をもつのは、関数名を読み取ることで現れる自然言語的な発語主体である。

しかし、自然言語とプログラミング言語にはきわめて大きな乖離があるので、人間がコンピュータにやってほしいようなことが自然言語の発語というレベルではきわめて簡潔に現れうるものであっても、それをプログラミング言語で予め用意されている基本的な命令の単語による簡潔な記述で、実行可能なプログラムとして実現するのは多くの場合難しい。人間に実用的なレベルで意味のあるプログラムを開発する場合、複数人で分担して数多くの関数（オブジェクト指向の言語であればクラスも）を定義し、それらを複雑に組み合わせていくことになる。関数が作成され利用されるプロセスが繰り返されるにつれて、複雑な処理を実行する関数を定義することが可能となる。そうした関数が呼び出し（使用）可能となることで人間がコンピュータにやってほしいと考えていること（を自然言語で発語した場合のその発語が意味すること）により近い関数名を与えることが可能となり、関数が含む仮構的なプログラムの実行内容は、仮構的なプログラムの意味内容へと近似していく。この仮構的なプログラムの意味内容は、関数名がもつ自然言語的な意味内容として示されているのであるから、この近似は、仮構的なプログラムの実行内容と命名において示されるソースコードの自然言語的な意味内容との近似でもある。コンピュータの使用主体が人間であるという前提上、プログラミングはいわば人間にとって何らかの意味があるようにコンピュータへの命令を組み立てる営みであり、その営みにおいて自然言語的な意味作用にもとづいて関数名を決めることは、すなわち関数名においてプログラムの実行主体を（自然言語の発語主体として）仮構することである。

(29) 仮引数は、関数の宣言において関数名とともに記述されている変数で、その関数が呼び出されたときに呼び出し元から渡されたデータ（実引数）を受け取って関数内での処理に使用するもの。反対に、実引数は、関数の呼び出し時に、呼び出された関数へ渡されるデータとして指定されたもの。

この関数名において仮構されるプログラムの実行主体の有り様を考えてみるとき、コンピュータと人間との間で築かれる関係でどのようなことが生じているのか、そしてその関係においてプログラミング言語の中にある自然言語がどのような存在であるのかが見えてくるように思われる。

関数の命名が関数の実装とは別様なプロセスの中で恣意的かつ便宜的に決定せざるをえないものだという点で、「IsJapaneseCharacter」という関数名がもつ自然言語的な意味内容はこの関数にとって厳密に正しいことを証しえない。関数の実装を修正することで、「IsJapaneseCharacter」がもつ自然言語的な意味内容が、人間にとってよりふさわしく見えるようになることは確かにありうる。しかし絶えざる修正よってどれだけよりふさわしく見えるようになったとしても、それはそのように「見える」だけでしかない。つまり、よりふさわしく「見える」ことを越えて、この関数名がもつ自然言語的な意味内容がよりふさわしく「なる」ことは決してない。見えるものでしかないのは、人間とコンピュータの関係が見なしのプロセスによって成立するものであり、あくまで（プログラムの実行を通じて最終的には人間が）そうと見なしうるかぎりにおいて、この関数名のふさわしさが示されるからである。見えるということはそのように見なすことが結果的にできたということである。

しかしまた、関数名が果たす役割は、このふさわしさがそうと見えるものでしかないこと、つまり見なしのプロセスの上で築かれた関係において果たされるものであることによって、むしろその効力を支えられている。コンピュータにとっては関数名がただ識別のためにあり、それが含む自然言語の意味は何ら効力をもたないという点によって、関数名がもつ自然言語的な意味はプログラミング言語において価値あるものになっている。人間がプログラムの実行主体にはなれないように、コンピュータもまた、直接的には人間の自然言語の発語主体にはなることはできない。そのため、自然言語の意味が、人間の恣意的な設計なしに、リースコードにおいて示されるべきプログラムの実行内容と結びつくことはない。しかし、結びつくことがないからこそ、逆説的に人間の側の都合にあわせて名をつけ、プログラムの実行主体と自然言語の発語主体にある乖離を見えないようにすることができる。そして、乖離が見えないようになっているときには、見なしのプロセスの中で仮構的なプログラムの実行主体が不可避的に想定されるのである。

関数名に限らず、プログラミング言語には自然言語の単語はしばしば含まれる。「if」や「for」あるいは「while」といったプログラミング言語の基本的な言語機能を司る語にも、自然言語の単語がそのまま用いられている。それゆえ関数名において、そこに含まれる単語それ自体がそれらの基本的な語よりも自然言語に近いということはない。では、関数名に含まれる自然言語の単語が「if」や「for」などといった語とは何が異なるのかと言えば、単語が並べられ命名されることで、人間が見なしたいもの（プログラムの意味内容）により一層近づく意味作用をもたらすことが望まれているという点である。

関数の実装コードに示される仮構的なプログラムの実行内容が、関数名において仮構的なプログラムの意味内容に近づくということが、すなわち仮構的なプログラムの実行内容と（ソースコードの）自然言語の意味内容との近似が、いかなるものであるのか、関数名（や変数名）がソースコードの形成の中で様々に組み合わせられる段階を見ることで、もう少し検討する。

以下は、入力されたテキストデータを加工して HTML のテーブルを生成するプログラムのソースコードである。

```
#include <vector>
#include <string>
#include <iostream>

using TextTable = std::vector<std::vector<std::string>>;

std::string CreateStartTag(const std::string& element)
{
    return "<" + element + ">";
}

std::string CreateEndTag(const std::string& element)
{
```

```
    return "</" + element + ">";
}

std::string CreateTag(
    const std::string& element,
    const std::string& text)
{
    return CreateStartTag(element) + text + CreateEndTag(element);
}

std::string CreateTags(const std::string& element, const
std::vector<std::string>& texts)
{
    std::string html{};
    for (auto&& text : texts) {
          html += CreateTag(element, text);
    }
    return html;
}

std::string Indent(size_t size, const std::string& html)
{
    constexpr auto indent = '¥t';
    return std::string(size, indent) + html;
}

std::string BuildTableRow(const std::vector<std::string>& row)
{
    constexpr auto lineFeed = "¥n";
    constexpr auto rowElement = "tr";
```

```
    constexpr auto cellElement = "td";
    auto cellHtml = CreateTags(cellElement, row);
    auto rowHtml = CreateTag(rowElement, cellHtml) + lineFeed;
    return Indent(1, rowHtml);
}

std::string BuildHtmlTable(const TextTable& data)
{
    constexpr auto tableElement = "table";
    constexpr auto lineFeed = "¥n";
    std::string htmlText{lineFeed};
    for (auto&& row : data) {
          htmlText += BuildTableRow(row);
    }
    return CreateTag(tableElement, htmlText) + lineFeed;
}

TextTable Input(const std::string& path)
{
    return {
          {"cell0-0", "cell0-1", "cell0-2"},
          {"cell1-0", "cell1-1", "cell1-2"},
    };
}

void Output(const std::string& path, const std::string& htmlText)
{
    std::cout << htmlText << std::flush;
    }
```

```
int main(int argc, char* argv[])
{
    auto texts = Input("textPath");
    auto htmlText = BuildHtmlTable(texts);
    Output("resultPath", htmlText);

    return 0;
}
```

いくつかの関数が定義されているが、生成されたプログラムが起動された際に最初に実行されるコード箇所（エントリーポイント）は、最下部にある「main」関数である（なお、「main」関数に関しては、起動時に最初に実行される箇所であることを指示するため、予め決まった名前になっておりプログラマが自由に命名（宣言）できるものではない）。下の２行ほどのコードは、実行結果確認のため便宜的にあるものや「main」関数の決まり上あるものである。最後の return 文はこの関数の終了を指示するものであり、プログラムを実行するコンピュータがコードのこの箇所まで処理を進めた場合、この関数で行なわれる処理は終了（または中止）になる。そして「main」関数が終了した場合、プログラムは終了（プログラムの実行で行なわれる処理が完了）したこととなる。すなわち、プログラムの実行内容という観点からすれば、「main」関数で行われるだろう処理の総体は、このソースコードから生成されるささやかなプログラムで実行される処理の総体と同義である。

「main」関数は実質的に上から３行ほどの命令で構成されているように見える(30)。まず最初に「Input」関数の呼び出しがあり、その関数の実行結果（戻り値）が「texts」変数に代入される。次に「BuildHtmlTable」関数の呼び出しがあり、実行結果が「htmlText」変数に代入される。最後に「Output」関数が呼び出される。「Input」関数と「Output」関数はデータの入出力処理を行なうための関数である。関数の宣言はファイルからの入出力を想定したものになっているが、簡略化のため、

(30) コードの書き方によっては、一時オブジェクトを変数に代入することをやめて１行（１文）で記述することもできる（無理のない範囲の内容である）が、構文をできるだけ複雑にしないために、３行（３文）に分けて記述してある。

「Input」関数は決まった値を返す（データを取得する）だけになっており、「Output」関数はファイルへの出力ではなくディスプレイへの出力を行なうものとしている。「BuildHtmlTable」関数なるものは、その関数名を自然言語的に読解した結果を信じるのであれば、おそらくはその関数が呼び出されることで「texts」変数にあるデータからHTMLのテーブルコードが生成されるものなのであろう。

これらの３つの関数は、プログラミング言語の仕様として予め用意されたものではないので、ソースコード上のどこかにそれらの定義が記述されている必要がある。このサンプルの場合、それらの定義は「main」関数より上のコードにある。しかしながら、定義されている関数はこの３つ以外にも存在している。この３つ以外の関数は、「main」関数のコードを読むかぎりでは、それらが呼び出されていることを確認することはできない、すなわち「CreateTag」や「BuildTableRow」などいった関数の存在を見つけることはできない。それは別の言い方をすれば、「CreateTag」や「BuildTableRow」などいった自然言語由来の関数名から生じる、ソースコードの意味内容（仮構的なプログラムの意味内容の一部）は「main」関数内のコードからは失われている。というより、見えているものだけを文字通りに読み取るなら、そもそも存在していない、というほうが正確ではある。少々くどい言い回しになってしまったのは、仮構的なプログラムの実行内容と実際のプログラムの実行内容との間を隔てる仮構的なプログラムの意味内容、という三者の様相に関係しているためである。

「main」関数の中で呼び出されていない（利用されていない）からといって、コンパイラが行なっていた最適化のように、「CreateTag」や「BuildTableRow」などいった関数の定義を消去してしまったとしたら、このソースコードは正常にコンパイルされることは不可能になってしまう。なぜなら、「main」関数の中で呼び出されていない関数は、別の関数、とくに「BuildHtmlTable」関数が行なう処理の中で利用されているからである。「CreateTag」や「BuildTableRow」といった関数名は、「main」関数の中には記述されていないという点で、仮構的なプログラムの意味内容としては存在していない（あくまで「main」関数のみを読み取ったかぎりでは、という条件はつくが）。ただし「BuildHtmlTable」という名が意味する内容が適切に機能するようになっているのであれば、「CreateTag」や「BuildTableRow」と命名された関数の中で行なわれることになっている処理は、仮構的なプログラムの意

味内容としては存在していなくとも、書かれているとおりに実行されるはずである。実際のプログラムの実行内容としては存在しているはずでなければならないものが、ソースコード上の自然言語の意味からは消失している。ソースコードの意味内容から仮構的なプログラムの実行内容が読み取られ想定されるとして、ソースコード上で自然言語の意味内容の比重が高まるほど、実際のプログラムの実行内容から乖離していくことになる。仮構的なプログラムの実行内容と自然言語の意味内容との近似は、仮構的なプログラムの実行内容と実際のプログラムの実行内容との乖離ということを不可避的に意味する。

とはいえ「main」関数において、「CreateTag」や「BuildTableRow」などいった直接的に呼び出されない関数の名前がもつ意味内容が示されないとしても、この関数内から読み取られる仮構的なプログラムの意味内容に不足がある、ということを必ずしも意味するわけではない。このささやかなプログラムで行なわれることが期待されているのは、「入力されたテキストデータを加工して HTML のテーブルを生成する」ことであって、それは「Input」「BuildHtmlTable」「Output」という 3 つ関数名で十分に示されていると考えられるからだ。生成されたプログラムが期待どおりに実行されるのであれば、実際のプログラムの意味内容（入力されたテキストデータを加工して HTML のテーブルを生成する）と「main」関数から読み取られるところの仮構的なプログラムの意味内容（「Input」「BuildHtmlTable」「Output」）は、それほど乖離していないし、十分に近似しているだろう。

この近似は、適切に命名されることで適切にコードが書かれうることも意味している。関数の命名と実装は本質的には便宜的なものであるから、不適切な命名であっても適切にコードを書くことは不可能ではないが、命名が不適切であればあるほど、書くことはいっそう困難になっていく。「BuildHtmlTable」が「Vdssfhiuo」だったり「BuildTableRow」が「tAHLDnsfc」だったりと、意味のない文字の羅列になってしまったら、仮に自分自身で記述したコードであっても、プログラミングは相当困難になる。それらの関数を適切に呼び出すために、それらの関数で何が行なわれているのかを、そのつどより実際のプログラムの実行内容に近い局面で把握していないといけないからである。実用的なプログラムではサンプルよりはるかに膨大な量とはるかに複雑な構造のコードを必要とし、また複数人が開発に関わることになるわけであるので(31)、それらの関数が意味不明な名前であることは、困難さを

より一層大きくする。

プログラムの単純な使用者であるなら、「入力されたテキストデータを加工してHTMLのテーブルを生成する」ということだけ理解していればよく、「タグをつける」という作業のことを知っている必要はない。同様に「main」関数を実装するプログラマも、「BuildHtmlTable」関数がすでに何者かによって正しく実装されているのであれば、結局のところ「BuildHtmlTable」関数を呼び出すようにするだけでよいということであり、「CreateTag」「CreateStartTag」「CreateEndTag」といった関数のことやそれらの名前が示す仮構的なプログラムの意味内容を把握していなくともよい。しかし、そういったことが可能になるのは、「main」関数を実装するプログラマが把握していなくともよい「CreateTag」「CreateStartTag」といった関数が、「BuildHtmlTable」関数を機能させるための基盤となっているからである。そして、「BuildHtmlTable」関数を実装するプログラマにとっては「CreateTag」「BuildTableRow」といった関数の名前が示す仮構的なプログラムの意味内容は重要である(32)。

関数の呼び出しは、プログラムのソースコード全体としては階層的な構造を形成していくのが通例である。階層は必ずしもきれいに区分されて形成されていくとは限らないが、きれいに区分されたほうが理想的ではある。たとえば「CreateStartTag」や「CreateEndTag」といったプリミティブな部品を構成するような関数がもっとも下位にあって、それらを組み合わせて一セットのタグで囲われた文字列を作るものとして「CreateTag」があり、さらにこの「CreateTag」が組み合されて別の関数が作られ、最後に「BuildHtmlTable」がもっとも上位の階層としてそのほかの関数を呼び出すことで作られる、といったようなことである。「BuildTableRow」関数内で「CreateTags」と「CreateTag」が並列して呼び出されているように、実際

(31) このあたりで述べられている事象は、いわゆるオブジェクト指向型プログラミングにおいて想定されている分業体制において顕著に表れるものではあろうが、次註にもあるように、実際に分業が行われなくとも（またオブジェクト指向型でなくとも）事態は本質的には同じであると考えられる。註 (5) をも参照のこと。(T)

(32) ここでは「main」関数を実装するプログラマと「BuildHtmlTable」関数を実装するプログラマが別の主体であるかのように述べたが、それぞれ別の主体であると分かりやすく現れるというだけで、同じ主体であったとしても同様のことは言える。

的な面では関数の呼び出しの階層関係が厳密なわけではない。しかし、関数がそれぞれの中で階層関係にあることで、理想的には、各関数を実装するプログラマは自分が実装する関数より一つ下の階層に位置する関数がどういう役割なのかを知っているだけでよくなる。言い換えれば、一つ下の階層である関数をどのように呼び出せばよいのか、そしてそれらの関数がもつ仮構的なプログラムの意味内容がいかなるものであるのかを適切に把握していればよい。

関数の宣言における命名と呼び出しの際に現れる、仮構的なプログラムの意味内容のこのような様相は、いわばソースコード読解の最適化のプロセスでもあると言える。この最適化のプロセスにおいて不要なものはなかったことになり、予め実行可能なものは実行されひとつの出力結果へとまとめられる。仮構的なプログラムの実行内容と仮構的なプログラムの意味内容との近似は、仮構的なプログラムの実行内容と実際のプログラムの実行内容との乖離を意味する。最適化が進むほど、仮構的なプログラムの実行主体は、実際のプログラムの実行主体から自然言語の発語主体へと遠ざかり、実際のプログラムの実行主体は見えなくなる。関数名に示される仮構的なプログラムの意味内容は、おおよそプログラム全体の意味内容の一部であって、階層的に使用されていくことで、しだいにプログラム全体の意味内容へと近づく。こうした階層的な関係において、下位の階層の関数は上位の階層の関数内で呼び出される（使用される）ことで、自身は仮構的なプログラムの実行内容へと埋め込まれる。このプロセスによって、下位の階層の関数が上位の階層の関数を機能させる基盤となり見えなくなることで、新たに命名されたより上位の関数名はよりプログラム全体の意味内容へと近づく。

関数の実装だけでなく、関数の宣言も、人間であるプログラマが熟慮や創意を凝らした結果、主体的に注意深く構築されるものではある。だが、関数の命名で示される主体性なるものが仮にあるとしても、それはまた別の（上位の）関数の実行主体の礎となるために、その主体へと明け渡され、見えなくなる(33)。

そうでなければ、プログラム（全体）の実行内容を遂行する主体（プログラムの実行主体）が仮構されることはないのであり、全体がひとつのプログラムとして統

(33) ここで問題になっている仮構的な実行主体と関数名の関係、たとえば関数名は関数における実行主体なのか、実行主体の名なのかについては、今後の課題のひとつである。

合性を持つことはないのである。「入力されたテキストデータを加工して HTML のテーブルを生成する」プログラムであるというとき、そのような自然言語の文においては、HTML を加工して生成する者としての何らかの主体が前提として想定されてしまうのであり、そしてその主体は人間ではない（それはその関数名で呼ばれるところの何者か、というより、その関数名が呼ばれ関数が呼び出されることで働くその働き以外の何物でもないが）。「Input」「BuildHtmlTable」「Output」という関数が「入力されたテキストデータを加工して HTML のテーブルを生成する」プログラムとして統合的に機能するのは、これら以外の諸々の関数がそれぞれ別個の小さなプログラムとして無関係に存在するのではなく、別個の小さなプログラムとして存在するというそれぞれの主体性を自身以外の上位の関数に明け渡し、その基盤となっているからにほかならないのである。

プログラミングにおいて、コードの記述という側面では、関数名において示されるような自然言語的な（プログラムの）意味内容というより高い階層は、コンピュータへの命令というより低い階層にある記述の連なりが基盤となって示される、という図式になっている。しかし、プログラムのコードを読解する、すなわちプログラムの意味内容を理解するという側面では、自然言語的な（プログラムの）意味内容こそプリミティブな基盤であり、いかなるコンピュータへの命令も、この基盤の上で仮構的に把握される。つまり自然言語が最も低い階層にあり、コンピュータへの命令というものがより高い階層に位置する図式となる。命名の恣意的かつ便宜的な決定可能性が、本来整合できないはずの両側面を、整合されているかのように見なさせる。

仮構的なプログラムの実行内容と自然言語的な意味内容との恣意的・便宜的同一化が成立する、すなわち関数がその名にもとづいて正しく機能しているように見えるかぎりでは、その名（たとえば「CreateTag」という名）は、人間にとっては、ある特定の文字列を前後に結合するというプログラムの実行主体としてのコンピュータではなく、自然言語的な意味内容を実現してくれる主体として立ち現れる。関数名が名前でありながら述語を含んだ命令形のかたちで名づけることが、現在では比較的多くのプログラミング言語において慣用的なルール(34)となっているのは、プログラム実行主体としての主体性を、ほかでもない関数名においてより明確に現

れさせるためでもあるのだろう。「○○せよ」という自然言語による命令を受け取り、その命令を実行することのできる「○○する者」としての主体がそこにあらわれる。このとき、関数名に含まれる自然言語的な意味内容は、コンピュータというプログラムの仮構的な実行主体を自然言語の仮構的な発語主体へと近づけさせる。人間がコンピュータを、このような自然言語の発語主体へより近づいた存在として問題なく見なしうるとき、プログラミングにおいて、それ以前には人間が主体的に行なわなければならなかった見なしのプロセスを成立させるための作業を人間は部分的に忘却しうる。

このプログラムの仮構的な実行主体は、関数の呼び出しというプログラムの実行内容と直接には関わりのない自然言語という基盤の上で成立している。人間はそこに自然言語を見つけるやいなや、意味を見いださずにはいられない。自然言語それ自体もまた、人間に見なしを可能にさせる仕組みだと言えるが、自然言語は人間という主体ときわめて密に結合した関係で結ばれている。しかしながら、プログラミングにおいて自然言語的な意味内容から仮構される主体は、この関数の呼び出しを含むプログラムが生成され実際に実行されることによってその仮構の正当性が保証されるにすぎない。あるプログラムの仮構的な実行主体によって可能となる見なしそのものの正当性の保証が、常に別の見なしを経由してしか示されない。あるプログラムにおける人間にとっての（自然言語的な位相で形成される）意味は、仮構的な実行主体が提示する見なしという基盤の上でしか存在しない。見なしのプロセスを形成する営みもまた、見なしのプロセスを通してなされるのであり、それはつまり、人間の代わりに何かをやってくれることになっているコンピュータという仮構的な実行主体の助けを借りて、また別のコンピュータという仮構的な実行主体を作り出しているということである。人間が自然言語的な意味内容を通じて見いだす主体は、人間がコンピュータという機械装置に見なしのプロセスを任せる、言い換えれば何かを別の何かと見なすという主体的な営為がコンピュータに明け渡される結果生まれるのである。

---

(34) 関数の戻り値が真偽値（Boolean 型）のときは、三人称単数の動詞（とくに is）から始まることが多い。また、動詞を含めない場合もある（その場合は as、from、to といった前置詞で始まる命名が多い）。

4. おわりに

見なしの主体性をコンピュータというプログラムの仮構的な実行主体へと明け渡す、あるいは明け渡されるような主体性を問題なく措定することができるという事態（それは無自覚になされるものなのであろうが）において、プログラムのコードを読み書きできるような理解というものが起こり、ソースコードを媒介とした人間とコンピュータとの関係が成立する、かのようなことが生じる。自然言語の発語主体としては決して同一化することのないはずのプログラムの実行主体に、人間という主体のための意味が見いだされたとき、コンピュータという、本来的には機械装置であって決して人間という主体の代わりにはならないものが、仮構的な実行主体としてその代わりをしてくれたかのようなことが実現される。その者は、むしろ人間とは根本的に異なるがゆえに、膨大な量の計算であれ、資料の検索や整理であれ、様々なことで人間よりはるかにうまく代わりをしてくれる。

コンピュータと人間の関係が根本的に見なしを通じてしか構築されえないのは、その関係の脆弱さの現れでもある。どれほど仮構的なプログラムの実行内容と自然言語的な意味内容とが近似していこうとも、ソースコードが自然言語そのものになることは決してない。つまりコンピュータの言語が自然言語へと至ることはありえない。ただ限りなく近似していくことが可能なだけだ。しかし、そのことは積極的に肯定するようなことでもなければ別段否定的に捉えることでもないように思われる。また、限りなく近似していくということ自体、ある意味当然のことではある。

というのもおそらく、自然言語というもの自体がそもそも人間に「見なし」を行なわせるものであるからだろう。自然言語が可能にする「見なし」がどのような性質のもので、その「見なし」が人間にとってどのような意味をもつのかは、本論で検討できなかったことであり、今後の大きな課題の一つである。ただ、人間が何かを人間が理解しうるようなあるものとして認識できるとき、それは自然言語を通じてその対象をそれと見なすことができるときではないか、そして自然言語という存在なしには人間という主体における「見なし」は成り立ちえないのではないか、と考えている。であるならば、自然言語が人間を発語主体として不可避的に想定するものである以上、プログラミングにおいて「見なし」の主体性をコンピュータに明け渡すということは、少なくとも文字通りの意味では不可能なはずである。だから

こそ、プログラミング言語が自然言語に近似していくことに必然性があるのだろうし、また、「見なし」の主体性を明け渡すことはほぼ無自覚なかたちでのみ可能であるのだろう。「見なし」という仮初めの基盤の上で構築された人間とコンピュータの関係は齟齬の潜在的な可能性を含みつつ、それが忘却されていることで、あるいはされうるかぎりで、成立している。その基盤で生じる理解のプロセスそのものはプログラムのコードに記述化されてはいないのであり、それはまた人間が言語でもってプログラミングを理解しようとする際の困難をも生じさせているように思われるのである。

参考文献

アラン・シャロウェイ、ジェームズ・R・トロット、1999、『オブジェクト指向のこころ』、村上雅章訳、ピアソン・エデュケーション。

エーリヒ・ガンマ、リチャード・ヘルム、ラルフ・ジョンソン、ジョン・ブリシディース、1999、『オブジェクト指向における再利用のためのデザインパターン　改訂版』、本位田真一・吉田和樹訳、ソフトバンククリエイティブ。

大須賀節雄、2010、『言語と知能――言語はどのようにして創られたか？――』、オーム社。

デイビッド・A・パターソン、ジョン・L・ヘネシー、2006、『コンピュータの構成と設計　第3版』、成田光彰訳、日経BP。

西垣通編著訳、2010、『思想としてのパソコン』、技術評論社。

深沢千尋、2011、『文字コード「超」研究　改訂第2版』、ラトルズ。

マーティン・ファウラー、2000、『リファクタリング　プログラミング体質改善テクニック』、児玉公信・友野晶夫訳、ピアソン・エデュケーション。

マーティン・レディ、1999、『C++ のための API デザイン』、ホジソンますみ訳、三宅陽一郎監修、ピアソン・エデュケーション。

矢野啓介、2010、『プログラマのための文字コード技術入門』、技術評論社。