2016 年映像文化特論 1 月補遺 2

1. 細かい補足

まず最初に、忘れないうちに、二、三の細かい補足を。

先日のコメントバックに書いたドラえもんの話ですが、声優交代があったときに、「ドラえもんの声が変わった！」としてあちこちで話題になったことを思い出します。客観的事実としては「ドラえもんの声を演じる声優が替わった」のだけれども、それが「ドラえもんの声が変わった」という事態として語られたわけですね。そしてこの事態、つまり「声が変わった」という事態を、ドラえもんの同一性の崩壊と捉えた人たちは、「これはもはやドラえもんではない」と感知したのでしょうし、他方、この事態はあくまでも客観的には「声優が交代した」ことのあらわれにすぎない、という把握からは、「これはこれでいい」という判断が生まれたのだ、と考えることができるように思います――これは少し図式的すぎる整理かもしれませんが、このように整理すると、例えば映画にトーキーが導入されたときに一部の人々が激しい拒絶反応を示したというできごとも、このドラえもん事件と同様の性質を持ったできごとだったと考えられる気がしてきます。

もうひとつの補足、これは先日の最終回の後で質問があった件で、去年の映像文化論で扱った話題に関してなので、今年はじめて受講した人には関係ないといえばないのですが、ついでなのでご参考までにここで補足します。以前の中間まとめの回で、「画面に姿の映っていない人物の声はその画面を支配する」という話をし、去年の講師・片岡さんの論からの引用として「アクスメートル」という語を紹介しました。フランス語で acousmetre、いわば姿なき声として画面上に権能を振るう主体のことですが、これの対語として片岡さんの論では「ミュエット」という概念も紹介されました。こちらは、姿は画面上に映っているが一切声を発しない存在のことで、今年はこれには言及しなかったのですが、質問は、「アクスメートルという語はネットで検索するといろいろ情報が得られるが、ミュエットのほうはどこにも何も載っていないのはどうしてか、そもそもこの語の出典は何か」というものでした。アクスメートルもミュエットももともとは、今年の講義でも一度言及したミシェル・シオンというひとの映画音声論に出てくる言葉ですが、アクスメートルに関してはシオン自身いろいろ説明していて、映画論の世界ではそれなりに流通した語になっているのに対して、ミュエットのほうはその機能についてシオンはあまり詳しく説明しておらず、業界でも、よくわからないものとされているようです。ただ、「ミュエット」というカタカナ語での呼び方は片岡さんと私とで考えたもので、もとは le personage muet（沈黙の人物）か何かそういう語句で、muet（ミュエ）は形容詞なんですね、でこの「沈黙の人物」というのを、「アクスメートル」と対になるような良い呼び方で呼びたいというので、フランス語のオリジナルな発音とは違うけれども可愛く「ミュエット」と呼ぼうということにしたものです。ですので、「ミュエット」で検索しても何も出てこないのは当然で、le personage muet で検索すれば、フランス語や英語のサイトではなにがしかのことが出てくるだろうかと思われます。

このミュエットの話は、今年はする暇がなかったのですけれども、実は浦野さんの「映像認知の穴」としての黒の話と結びつく、奇妙な現象に関する話題を提供してくれるものです。またの機会にはぜひお話ししたいと思いますが、それよりも、やがて数年のうちには、片岡佑介さんの名前で検索すると何か出てくるような時代が訪れることでしょう。

2. プログラミングと自然言語に関する補足

前回の最終回､最後に時間がなくて話しきれなかった部分を渡邊さんが送ってくれたので､以下に貼ります。

――渡邊さんによる補足――――――――――――――――――――――――――――――――――――――

私たちは自然言語の文章を読むように、順に語句を追っていくだけでは、プログラミング言語を読むのは困難です。読解を容易にするには、ソースコードの文字の視覚的配置を工夫して、プログラムの構造を把握しやすくしなければなりません。

そのようなソースコードに表れている、自然言語が置かれている二重の状況について、簡単なサンプルコードで、説明します。青色の「printf」などは、コンピュータへの命令で、緑色の「print」などはコンピュータが処理するデータとしての文字列を指します。

```
1   #include <cstdlib>
2   #include <cstdio>
3
4   int main(int argc, char* argv[])
5   {
6       printf("print");
7
8       printf("");printf("");
9       printf(");printf(");
10
11      system("pause");
12      return 0;
13  }
```

これはただ「print」などの特定の文字列を出力する（ディスプレイ上に表示する）だけのプログラムのソースコードです。いわゆる「Hello, world」プログラムとはディスプレイ上に表示される文字列が違いますが、同じような類のプログラムです。5 行目「printf("print");」では、出力される文字列は「print」、つまり、この命令が実行されると、ディスプレイ上に「print」という文字列が表示されるのですが、それを出力するための命令（printf）にもまた「print」という「文字」が含まれています。

文字列を出力するための命令の中にある「print」という部分は、人間にとっては、「印刷する」という自然言語の意味を思い浮かべざるをえないものです。しかしここでは、この青い「print」という「文字」はコンピュータのためにある符号で、コンピュータの言葉（機械語）の代用品でしかありません。この符号（printf）にとって本質的な情報は、コンピュータの命令だということで、そこにくっついている自然

言語の意味は付随的な情報にすぎません（どうして「印刷」なのかと思う方がいるかもしれませんが、「print」という語がディスプレイに表示する命令に含まれているのは、もともとコンピュータの端末がテレタイプ端末（電信みたいなものです）だった頃の名残で、その当時は本当に文字通り紙に「印刷」していたのです）。

その一方、出力される文字列である「print」が、人間が知覚できる「文字」として表示されるためにあるということは、この文字列が人間にとってもつ自然言語の意味は、それ自体、付随的なものではなくて、本質的なものであるはずです。これらの、人間がそれを知覚するために表示されるべき「文字」、つまり人間にとって自然言語あるいはその一部分としてディスプレイ上に表れてほしい「文字」は、ダブルクォーテーションマーク（""）によって囲まれることで、コンピュータへの命令を示す他の文字列から区別されています。ソースコードにおいて人間のための自然言語は、ダブルクォーテーションマークという柵の中に囲われることでようやく存在できます。

そのソースコードから作られたプログラムにおいて、「コンピュータへの命令の一部として、機械語の代替として働く自然言語（青色の部分）」と「コンピュータ内部で命令以外の情報として扱われ、人間への情報伝達のときに必要とされる自然言語（緑色の部分）」、ソースコードではこのような二重の状況に自然言語は置かれているのです（ただし後者は、コンピュータと人間との間での情報伝達にかぎらず、異なるコンピュータどうしの間での情報伝達としても、現在では広く使われています。「print」という文字列において、自然言語としての「印刷」という意味は人間のためだけのものだが、「print」という文字列が「文字（テキスト）」というデータ形式で記述されること自体は、しばしばコンピュータのためでもありえます）。この両者の本質的な役割は違っているように見えますし、それはすごく明確に区別されているようにも見えますが、実はそうではないのではないかと思うわけです。

というのも、ソースコード上で、それらはすべて同じ「文字」でしかないからです。もし、ひとつダブルクォーテーションマークをつけ忘れたり、あるいはダブルクォーテーションマークの囲いの中に、もうひとつのダブルクォーテーションマークをそのままうっかり挿入してしまったりしたら、その時点でコンピュータは、もはや両者をまったく正しく区別できなくなります。記述する人間自身も、しばしば正しく区別できなくて入力ミスをしてしまう。コンピュータの命令に当たる部分やダブルクォーテーションマークで囲われた部分に、エディタによって色がつけられていることが、かえってそれをよく示していると思います。あまりにもうっかり入力ミスしやすいからこそ、こうやって色をつけてわかりやすくしているに他ならないからです。

下の 8 行目は「printf("");printf("");」という行です。これは人間にとってはひとつの行ですが、プログラムとしては 2 つの文です。二つのセミコロン（;）は共に、命令の区切りを表す符号です。「2 つのダブルクォーテーションマーク（""）で囲われている文字列を表示せよ」という命令が 2 つ並んでいるだけのもので、かつ、2 つのダブルクォーテーションマーク（""）の中に何も入っていないので、この命令が 2 つとも実行されても、ディスプレイには何も表示されません。一方、もうひとつ下の 9 行目「printf(");printf(");」だと、これは人間にとってもひとつの行ですが、プログラムとしても 1 つの文で、「);printf(」という文字列を出力せよという命令です。両者の行は、どちらもセミコロンが 2 つあります。コンピュータにとっては、これらの行のセミコロンは、上の 8 行目は 2 つとも、命令の区切りを表

す符号として読み取られるものですが、下の 9 行目の行では、最後の 1 つだけがそういう命令として読み取られます。ですが、人間にとってはどちらの行でも、「セミコロンが 2 つある」としか見えませんし、個々のセミコロンの意味するところを区別するにはそれなりの注意力が必要になるくらいには分かりづらいものです。色分けがなければ、かなり苦労すると言ってもよいと思います。苦労しないのは、エディタの色分け機能のおかげにすぎません。そしてエディタはあくまでも（このエディタの場合）、「ダブルクォーテーションマーク 2 つで囲われている文字列は、囲んでいるダブルクォーテーションマークと一緒に緑色で表示する」というルールに従って色分けしているにすぎず、たとえば「print」という「文字」データそのものの中に、色情報が埋め込まれているわけではないのです。ですから、たとえば 8 行目は途中のセミコロンの直後で改行して、2 行にするなど、改行やタブといった「制御文字」や、半角スペースを使って、極力、人間にも区切りが視覚的に分かるようにしなければ、せっかくの「printf」という視覚的な「文字」から受け取れる意味情報も、有効に機能しないわけです（9 行目の改善に関しては、改行やインデントだけでなく、変数を活用するといったことが必要です）。改行やタブといった「制御文字」や半角スペースは、ソースコードにおける視覚的な「文字」が、人間の自然言語を運用する機能を、きちんと働かせる役割を十全に果たすための基盤としてあるわけでしょう。

ソースコードというプレーンテキストには、コンピュータというプログラムを実行する主体と、人間のプログラマーというプログラムを読み書きする主体と、二つの異なる主体を想定せざるをえません。そして一方の主体（コンピュータ）では、テキストを符号として識別する必要があるが、もう一方（プログラマー）は同じテキストを視覚的に把握しなければなりません。それにもかかわらず、プレーンテキストをディスプレイ上に表示する、ということは、多くの異なる役割をもつ符号が、等しく「文字」として操作可能なものとして、存在してしまうことでもあります。そういうややこしい事態が生じているのです。それゆえに、それぞれ二つの異なる主体が、同じプレーンなテキストを、それぞれ違う側面から取り扱うとしても、トラブルができるかぎり生じないように、様々な面倒な工夫が必要とされているわけです。ソースコードはデータそれ自体としてはプレーンなテキストですが、ソースコードの取り扱いを巡る状況はまったくプレーンではないのです。

武村先生のコメントバックに、自然言語が基盤として機能していることへの言及がありましたが、プログラミング言語との関係においても、基盤としてのその力は発揮されているように思えます。

「文字列でできた言葉」が、人間の側で、「正しい」意味であるように視覚的に見えている、これによってプログラミングにおける様々な利便性が成り立っています。人間に「正しい」意味であるように見えるには、コンピュータの「見なし」の仕組みのなかで、文字列がうまく表示される必要があります。そうでなければ、人間の自然言語が、人間とプログラミング言語との関係において、基盤としてうまく機能することはないでしょう。けれども、コンピュータの側としては、「文字」の視覚情報は、邪魔で排除されるほうが都合がよいし、出力されたあとのことは関知しないものでしかない。そういう危うい基盤の上で、これらの利便性は成り立っています。

自然言語や自然言語の「文字」が、人間がソースコードを「正しく」読むことに対して、力を発揮でき

る、そうであるとき、人間がそのプログラムを作るときに、たとえば関数が作られることで、関数名だけ読んだり書いたりすれば関数の中身を読まなくて済むといったかたちで、本来書いたり読んだりしないといけないコンピュータへの命令をいくつか不可視化する（ただし、プログラミング言語が用意している「printf」といったものがコンピュータの言葉、つまり機械語の代替であることも、関数が作られることと本質的には同じことです。「機械語→プログラミング言語→関数」といった「見なし」の繰り返しの中で、前の段階のものが不可視化されていくのです）。それによって、より楽に人間はソースコードを書くことができる。そういう意味で、自然言語や「文字」は、人間にとって「正しく」動作するプログラムをつくるための基盤として、人間とコンピュータとの関係において、必要とされています。

片や、コンピュータとの関係において、自然言語や「文字」は、コンピュータによる「見なし」の仕組みという基盤の上でしか、人間にとっての「正しい」「文字」や言葉として存在することが、そもそもできないものです。コンピュータによる「見なし」の仕組みもまた、コンピュータの内部では、プログラムが動いていることで、その多くが実現されています。たとえば、テキストエディタは、人間にとって「正しく」「文字」を表示するためのプログラムですし、ソースコードをコンピュータの言葉である機械語に翻訳してくれるもの――これをコンパイラなどと言いますが――これもプログラムとして動いているのです。

つまり、自然言語や「文字」は、人間にとって「正しく」動作するこれらのプログラムを、基盤として必要としています。そこでは、0 と 1 の並び（つまりビットの並び）が、何の苦労もなく「文字」として表示され、また、人間があれこれ考えなくても「文字」が機械語に翻訳される。といったように「文字」の「見なし」の仕組みが不可視化される。そこでは自然言語や「文字」のために、「見なし」の仕組みを担うプログラムが、基盤として必要とされています。

つまり、それら互いが必要としている基盤は、当の自分たちを基盤として必要としているはずのものであるわけです。その象徴的な例が、上記の、コンパイラというプログラムで、これは現代ではごく普通に、プログラミング言語を使ってつくられます。プログラミング言語を機械語に翻訳するプログラムが、プログラミング言語を機械語に翻訳するプログラムによってつくられるのです。機械語に翻訳するプログラムでも、機械語ですべてをつくるのは大変であるので、プログラミング言語を通じて自然言語の基盤としての力を借りたいわけです。でも、当然、プログラミング言語において自然言語が力を発揮できるのは、「文字」を機械語に翻訳するコンパイラという基盤が機能しているからです。

一方が存在するために、もう片方を基盤として必要としているが、その基盤として必要とされているもう片方は、一方のほうを基盤として必要としている、ある基盤が自分を基盤として必要としている基盤を必要とする、なんとも倒錯的といいますか、基盤の共依存のような関係が、自然言語とプログラミング言語の結びつきの中で生まれているのです。

現在では、人間にとって自然言語がきわめて基本的で汎用的な基盤になっているように、コンピュータにとって「文字（テキスト）」が基本的で汎用的な基盤になっているのだと思います。そして、両者の基盤が、ひとつの映像上で一緒くたにならざるをえないような局面というのが、プログラミングであり、その産物がプログラミング言語で構築されたソースコードなのです。

――以上、渡邊さんによる補足――

3.「映像的関係」について

現代の私たちは、日常的に多くの言語テクストをディスプレイ上で読みながら、それを「映像」としては把握していません。ときに処理ソフトの違いによって「これはテクスト」「これは画像」と判別したりしながら、それが「テクスト」であれ「画像」であれ、読める限りにおいては普通に「ことば」あるいは「文章」とかそういうものとして、紙媒体で文章を読むときとほぼ同様に読みます。そしてまた時に「やっぱり紙に落とさないと読む気にならない」とか、「ディスプレイで読むのはしんどい」等と感じながら、それらの違和感をあくまでも「紙で読むかディスプレイで読むかの違い」すなわち表示媒体の表面的な違いに帰すのが常です。ディスプレイは目が疲れる、とか、寝そべって読めない、といったような、それはそれで非常に深刻な性質もさることながら、しかし電子媒体において私たちが読む自然言語テクストが紙の上で読むそれとどう異なるのか、果たしてどのくらい異なるのか、という問いの答えは、ディスプレイなり紙なりに表示されているそのあくまでも視覚的な様態においてよりもむしろ、その表示がなされるに至る経緯、すなわち発語の形態にあるであろうことは確かです。そして、ソースコードを読む／書くという特殊な場においては、この発語の形態の輻輳的な性質がそのまま視覚上の輻輳的様態に反映している、そういうことなのではないかと思います。

みなさんが日常いろいろなウェブサイトを見ているときに、ブラウザのどこかにある「ソースを見る」とか「開発ツール」とかそういうボタンを探しあててクリックすると、そのページの元になっているソースコードを見ることができます。そんなこと始終やるひとはあんまりいないでしょうけれども、そうやって見ることができて、面白いのはそうして「ソースを見る」ボタンを押すと、何かのウィンドウが開いてソースコードが見られる、そのソースコードもまた、それが元になっているウェブサイトと同じディスプレイ上に「映る」という形で見られるのだというところです。そこに映し出されるソースコードは、まさに今働いているコード、というか、まさに今それをコンピュータが解読・解釈・処理してウェブサイトのページを映し出してくれているところのソースコードなんだけれども、それもまた「映っている」形でしか見られない。もちろんそのソースを紙にプリントアウトすることはできるけれども、「今機能している」コードとしては、「映っている」形でしか見ることができないわけですね。

先日のコメントバックで、「フォントにこだわるプログラマーはいない」ということを書きましたが、これは誤解を招くかもしれない、という指摘を渡邊さんにもらったので、少し補足します。最終回をきいた人はおわかりかと思いますが、ソースコードを「読む／書く」に際しても、テクストの「見かけ」、視覚的なありかたはやはり非常に重要なのです。ただその重要さは、書物やサイトの「デザイン」におけるそれとは根本的に性質が違っていて、後者においては、フォントを始めとする種々のデザイン要素は、最終的にできあがって公の閲覧に供されるところのいわゆる「成果物」の構成要素のひとつとして重視される、すなわち、本ならそれを買って読む人の全ての目にそのような形で見えることを目的として重視されるのですが、ソースコードを「書く」のは、最終的にそのコードを多くの人にそういう形で見てもらうことを最終目的とするものではありません（いや、それを最終目的とするケースもあるでしょうが――先日渡邊さんが見せてくれた、飛行機の形をしたソースコード、あれは「難読プログラム」コンテストに出品されたものだそうでしたから、そういう場合はむろんソースコード自体が「成果物」なわけですね）。汎用性の高いプログラムを作ってそれ

を領布するような場合でも、そのとき「成果物」であるプログラムは、あくまでも多くの人に「使ってもらう」ために作られるので、そのプログラムの見かけをその通りの形で見てもらうことが目的なわけではないでしょう。それでもフォントや色分けの色に個々のプログラマーがしばしばこだわるのは、そういう形が「見やすく」「見分けやすく」「作りやすく」「読みやすい」と判断するからです。そして渡邊さんの言葉を借りれば、そういう見やすい形態を選ぶことで自らの「労働環境」をなるべく最良のものにするためである。しかしその労働環境は、つきつめていえばその人とその人のコンピュータのためだけのもので、私が私のパソコンに入れたブラウザでいろいろなサイトのソースを見るときには、どういう環境で書かれたソースであれ、私のパソコンのブラウザの規定のフォントや色分けでもって表示されてしまいます。

これは Word などについても実は同じようなことが言えるでしょう。つまり例えば今私はこの補遺を、MS 明朝＋Times New Roman の組み合わせで、10 ポイントの大きさの字で、48 字×36 行のページ設定で「書いて」いますが、これを、読んでみてくれといって Word ファイルのままで友人の A さんに送るとします。受け取った A さんが仮に MS 明朝が嫌いで、すっごい読みにくい、こんな書式で長いもの読みたくない、と思えば、全く別のページ設定にしてしまって、フォントも別のものに変えて読む、ということができますね。それでも私は私自身の読み書きのために一番いいと思うフォントと字詰め行詰めを選んで書く。そうした設定を、全くいじらないで私が設定したままの形で読んでほしいと思えば、ひとに渡すときには Word ファイルのままにせず PDF にして固定するということがよく行われるらしい。もっとも、この補遺を PDF にして上げておくのは、別に私がテクストの形態にこだわるゆえではなくて、単に、Word ファイルのままだとサーバーに上げにくいからにすぎません（やってみたことはないけど、たぶん上げられない）。もはや私は自分のサイトの記事がリーダーモードで読まれようがどうしようが一向に構わない、むしろそういうことのほうが面白いくらいに思います。

話がそれました。

例えば「a」のキイを押すと「あ」と表示される、という状況の奥には、渡邊さんが話してくれたような幾重もの「見なし」のプロセスがあります。「あります」と書くだけでも、たいへんなことですね。それは口頭で「あります」と言ったり、鉛筆で紙に「あります」と書くよりもはるかに複雑なプロセスなのですが、それが、今ではほとんど何の遅滞もなくクリアされて、ペンで書くよりもキイボードで打つほうがはるかに楽、とすら思えるような状況になっている、そしてそういうことを全く意識することなく、あたかも軽く話しかけるようにメールを書いたり SNS に書き込みをしたりできる。そこでは「平板」（→「ノイズ・デザイン論」）どころではない発語の営みが行われており、興味深いことのひとつは、そこには必ず「翻訳」が介在するということかもしれません。私が Word で何を書こうとあるいは FB に何を書き込もうと、そこには、自然言語→機械語→自然言語という、翻訳→訳し返し、のプロセスが入っている。例えば村上春樹の小説が英語に訳されたものをまた誰かが日本語に訳し直したとして、それが「村上春樹の書いた小説」だと言えるのかどうか疑わしい、という意味では、私がこうして「書いている」のが本当に私が書いていると言えるのかどうか疑わしい、ともいえるのですが、しかし、自然言語→機械語→自然言語の翻訳プロセスを経て映し出されるテクストは、私が脳内出力していたテクストと、類似ではなく同一のものであらねばならない、という前提がこの一連のプロセスには置かれています。同一のものでないものが、例えば「侮ﾌ」などというものが出てきたらそれはバグっているのである。そういうことも含めて、私がキイボードを打っているときに目の前に

映し出されているテクストは私が書いたものである、それは私の発語であると「見なし」うる、そういう仕組みが、現代のネットコミュニケーションを支えているわけです。「いいね！」を押して何かを「共有」する、と言われるとき、そこで共有されているものは、「いいね！」という自然言語のタームであると同時に、これらの一連の多層的な「見なし」の構造そのものなのでしょう。

不可視のインフラストラクチャ、と私たちが今回呼ぼうとしてきたものには、二つの側面があります。ひとつは、インフラそのものが基本的に不可視であるという側面、もうひとつは、そのインフラによって何事かが不可視化されているという側面。ともに、そこで不可視なものが不可視であることによって、何かしらの上部構造が成立可能になっているという、そういう性質のものをインフラストラクチャと呼ぼうとしているのですが、私たち自身、このことに関してまだ詰めきれていないところがたくさんあります。「見なし」の輻輳的構造は、普通にコンピュータを使って日常の業務や遊戯や通信を行うという局面においては、それ自体が不可視化されているものですが、ではこの仕組みによって何が不可視化されているのかというのは、おそらく、自然言語そのものの成立にかかわる問いなのではないかと思っています。例えば私たちは今では、ほとんどのひとが、日本語で「書く」ときにもアルファベット入力をすると思うのですが、「書記言語は音声言語の代替にすぎない」という西洋的な音声中心主義にそろそろ疑問が呈されるようになってきた今になって、アルファベット入力という形であたかも再度、音声中心主義的メソッドが「正しく」活用されているかのようである。速読するときにことさら「声」が脳裏に響くことはないのかもしれないにしても、キイボードでアルファベット入力でテクストを「書く」ときには、それこそ誰しも、音声を脳内出力せずにはすまされないはずです。しかしそれでいて同時に、機械にとっては、音声が入力されるわけではなく、私が「おもしろい」と入力しながら頭の中で「おもしろい」という音声を響かせていたとしても、機械が受け止めてくれるのは、o-m-o-s-h-i-r-o-i という順番でキイが打たれた、ということだけなのです。そしてその入力データを解釈して、「おもしろい」と正しく表示してくれる。いったい何がどうなっているのでしょう！　しかしそもそも我々が自然言語と呼んでいる言語であっても、実は同様の「見なし」の仕組みで成り立っているのかもしれないのです。言語モジュールという言葉を今回、本田さんからお借りして何度か使いましたが、ひとくちに「言語化」といっても、その「言語化」がどのようにして言語モジュールの中で、ときに非言語脳と連動しながら行われているのかは、まだ何もわかっていないのです。