〔教育研究〕

プログラミング言語教育法についての考察

伊藤 公洋*

―― 目 次 ――

キーワード：情報教育、抽象と隠蔽、多言語プログラミング

1．コンピュータ ＋ プログラム ＝ システム？

かつて「コンピュータ、プログラムがなければただの箱」と言われていた時代があった。しかし、現在ではこのように言われてもピンとこないパソコン・ユーザがほとんどであろう、それは入手可能なパソコンには（一部を除き）当初からオペレーティング・システム（OS）が搭載されており、購入後電源を投入すれ

＊ Kimihiro ITO 本学教授、総合教育研究センター所属

ばともかくも何らかの動作をして、ユーザからの入力を受け付けられるようになっているからである。

さらにそれらのパソコンにはオフィス・スィートのようなワープロや表計算など一般的なアプリケーションがプリ・インストール（購入時に導入済み、という意味）されていなくとも、インターネットでホームページ閲覧や電子メールを行えるようになっているのが普通であるし、音楽や動画を再生でき写真を表示できるアプリケーションが必ず内蔵されている。無償で利用できるOSでもこの状況は変わらない。

また、かつてのコンピュータは「箱形」であったが現在ではスマートフォンやこれから普及するであろう電子ブックなどは既にそのような形状ではないにもかかわらず、コンピュータである。

このような状況が初学者に混乱を与えていると考えられる。すなわち、プログラムの存在が隠蔽され過ぎているのである。もちろんOSもブラウザもオフィス・スィートもすべてプログラムである。しかしそれを意識することはなく、それがプログラマの目標であったように、これからも複雑な構造を上手に隠蔽してユーザに使いやすい環境を提供していくことに変わりはないであろう。反面、そのためにプログラミング学習のハードルは高くなっていく一方である。

FORTRANというコンピュータ用の高水準記述言語が初めて開発されたのが1954年であり、それから様々なパラダイムや方法論、プログラミング言語が登場してきた。しかし、その間にコンピュータそのものの原理は変わっていない。万能チューリング・マシンであり、メモリにプログラムとデータを格納して処理をするストアード・プログラミングのままであるにもかかわらず、プログラミング言語の学習は複雑になっていると考えられている。以下ではその理由について考察をしてみたい。

2．抽象と隠蔽

「理解は、分類することから始まる」（ハイドン・ホワイト[1]）とあるように、共通部分と差異を特定することにより科学技術は進歩してきた。情報科学も同様であるが、ムーアの法則に代表されるように年々指数関数的に向上する処理性能

とユーザや社会からの要望に応えるために、コンピュータで扱える範囲は拡大する一方である。この傾向は今後もかなりの年月に渡って継続すると予想されるが、応用範囲の拡大は複雑さに対処することをコンピュータ科学や工学に強いてきたことを意味している。

複雑さに対処する方法としてわれわれは抽象度を上げることや詳細を隠蔽することを選択してきたのであるが、方法としては間違っていないだろう。もしこの方法を採らなければエンド・ユーザが使いやすいアプリケーションを開発するだけでなく、単機能なプログラムでさえ作成することが一気に困難となる。例えば、あるデータを外部記憶装置（一般的にはハードディスクなど）に保存するには最近のプログラム言語ならほんの数行の処理を記述するだけでいいのであるが、これはそのプログラムとハードウェアの間、つまりOSの役割を隠蔽して抽象化できていることから享受できるのである。

抽象化の利用はこの他にもシステムの分析・設計からアプリケーションの利用にまで幅広く応用されているのであるが、言うまでもなく過度の抽象化は曖昧さを招く。この際に注意しなければならないのは抽象化の対象に対しての粒度である。私の経験では、この粒度の度合いによってプログラミング言語の学習や理解にかなりの差が生じるようである。

同様に隠蔽も過度に行うと、ブラックボックスだけを相手とすることになり背後にある原理の理解や利用への自由度がなくなってしまうという弊害が生まれる。これへの対処は適切なインターフェイスの公開という方法であり、実は公開すべきインターフェイスにも抽象度が必要となる。コンピュータ・システムに限らず一般的な機械や電子機器の場合にも、適切な設計がなされているものは適度に抽象化されたインターフェイスを有しており、操作説明書やマニュアルを読まなくても操作できることが可能となっている場合が多い。

さらにこのように適切に設計されているモノは自然なアフォーダンスをユーザに提供できることになる。アフォーダンスとはそのモノがユーザに対してあることを行うことを促したり連想させたりすることであるが、それはそのモノとコミュニケーションできることをも意味すると考えられる。この意味においては現在のパソコンは自然なアフォーダンスを持たないと言えるだろう。

３．レイヤーと互換性

複雑さと同程度の困難を伴うものとして安定性（堅牢性）があげられる。その対処として導入されたもののひとつにレイヤーがある。レイヤーとは階層構造の「層」のことであり、適切に分類されたある意味を構成する機能群のことを指す。インターネットなどのコンピュータ・ネットワークで利用されているOSIの参照モデルは７階層から構成され、どの層も交換可能となっておりその適切な抽象度と分類により、理解も容易なものとなっている。また最近ならばWebアプリケーションはMVC[(2)]という方法で構築されているがこれも適切なレイヤーの利用形態のひとつであろう。

レイヤーのメリットは上図からもわかるように、自身が接している層に対してのみコミュニケーション、すなわち関心を持てばいい、ということである。伝言ゲームのように前の層から来たメッセージを自分の層で処理し、次の層にそれを送信すればいいのであるが、これが階層構造を持っていなかったならば複雑さが増すばかりでなく、安定性も失われる可能性が高い。また各層は交換可能である、ということは自由度が増すことを意味している。

交換可能ということが、直ちに互換性を意味することはないが、互換性の考慮も複雑さを低減することと安定性を増すことに貢献する重要な概念である。特定

のアプリケーションの互換性については広く周知されているが、以前のバージョンのアプリケーションのデータはその次などのバージョン・アップされたアプリケーションでも利用可能である。この互換性のことを下位互換というが、一般的にはアプリケーションのデータは上位互換ではない。理由は単純で「未来のことはわからない」からである。バージョン・アップされたアプリケーションは過去のことを知っているから互換性は保たれるわけである。しかし互換性は単純にデータに関してだけではなく、操作や概念についての場合の方が重要であり価値も高い。ユーザ・インターフェイスやアフォーダンスにも関係するが、ここではシステムの機能の互換性だけを考えることとする。

これらレイヤーと互換性に共通する性格として注目すべきは、方向性を有するということであり取り扱う際にはスカラー的ではなく、ベクトル的になるという点である。そのため、思わぬ落とし穴に陥る危険性もあるが抽象化や隠蔽よりも理解し易いと考えられる。特に適切に設計されたレイヤーや階層構造はユーザの利便性を向上させられる手段として今後も広く応用され、利用されるであろう。

これら４つの方法を採用してコンピュータ・プログラミングは困難を克服してきたのあるが、これらの方法に共通することは間接性の増大であり、これこそがプログラミング言語の学習を困難なものにしていると思われる。そこで、以下ではその対処法について考察する。

４．OSとの対話

最近の一般的なパソコン環境であるGUI（グラフィック・ユーザ・インターフェイス）を採用しているOSでもOSそのものと直接的に対話できる手段が搭載されている。MS-Windowsの場合には「コマンド・プロンプト」、MacintoshやLinuxなどのUnix系OSならば「ターミナル」や「端末」と言われるもので、GUIとは異なりCUI（文字ベースのユーザ・インターフェイス）、すなわちコマンド・ラインでの対話を可能とさせるGUI以前のコンピュータで対話をする方法を提供するアプリケーションである。

プログラミング言語を学習する初心者のほとんどは、このアプリケーションの

存在を知らないために次のようなやり取りと行うことになる。以下はMS-Windowsでの場合である。

◆パソコンが起動したら、「スタートボタン」＝＞「すべてのプログラム」＝＞「アクセサリ」＝＞「コマンド・プロンプト」を選択してください。
◆するとデスクトップ上に真っ黒いウィンドウがひとつ開くでしょう。
◆次に、キーボードから「１＋２」と入力してみましょう。どうなりますか？
◆「‘１＋２’は、内部コマンドまたは外部コマンド、
操作可能なプログラムまたはバッチ ファイルとして認識されていません。」
と表示されますね。決して「３」とはなりません。

この時点で学習者はまだこのコマンド・プロンプトの機能を理解できないかもしれないが、「内部・外部コマンド」と「プログラム」、「バッチファイル」というものがあることを理解するだろう。実はこれらのものを作成する手段がプログラミング言語なのである。

しかしそれより重要なことは、OSに「認識されていない」、つまり実行できないということの意味である。実行可能なプログラムは、まず何よりもOSに認識されていなければならない。そのための手段を提供するものがファイル・システムなのであるが、それは後にしておきコマンド・プロンプトからの入力を続ける。

◆では、次に「date」と入力してみましょう。大文字でも小文字でもいいですが、半角アルファベットで行ってください。
◆はい、今日の日付が表示されますね。もう一度エンター・キーを押してください。最初の状態に戻ります。
◆さて、今度は「notepad」と入力してください。
◆はい、「メモ帳」が別のウィンドウで起動しましたね。

コマンド・プロンプトから他のアプリケーションを起動できることを学習者は理解することになるが、さらにここでもう一つの重要な概念である「マルチタスク」も実現されていることを指摘しなければならない。よりいい方法としてはも

う一つの別のコマンド・プロンプトを起動できることを確認させるとよいだろう。ここでのポイントとして、学習者はGUIを利用してダブル・クリックなどで起動しているアプリケーションが別の方法でも可能であることと、内部・外部コマンドというプログラムが存在していることを理解するようになることだが、これらが普段はOSにより隠蔽されていることを認識してもらうことがこの目的である。

コマンド・ラインでのMS-Windowsの操作がこの小論の目的ではないため割愛するが、外部・内部コマンドをいくつかは理解していないとプログラミング言語を自由に操作できることにはならない。以下では適宜必要なコマンドを記すことにする。ここでは最後にコマンド・プロンプトから「exit」と入力してもらい、コマンド・プロンプト自身が終了することを学習者に確認してもらう。その際に先にそのコマンド・プロンプトから起動してあった「メモ帳」は終了しない、という事実を指摘してマルチタスクを意識させることが必要である。

5．ファイル・システム

コンピュータのOSが認識できるビット列のことをファイルというが、ここには高度に抽象化された概念が横たわっている。GUI環境ではフォルダーとファイルとして理解されているものは階層構造であるし、データをダブル・クリックすると適切に関連付けられたアプリケーションが起動して利用可能となるためその背後にあるものを意識することは少ないだろう。さらに最近のWindowsでは特定のデータは予め設定されている特定のフォルダーに保存「される」ように促されるため、自分でファイルを整理するためにフォルダーを作成することも少ないかもしれない。

状況をより複雑にしているのはデスクトップ上にある「コンピュータ」（ないものもあるが、「スタート」ボタンを押すとメニュー内に表示される）は、初学者でさえ「コンピュータはデスクトップにある」というメタファーを理解しているにもかかわらず実際にはCドライブ（起動ディスク）の個人用フォルダー（Vistaならばc:¥Users¥ユーザ名）の直下にあるDesktopフォルダーが実際のデスクトップの内容を保持している、という事実である。つまり包含関係が逆転ないしは不条理なものになっているのであるが、これはデスクトップというメタ

ファーを重視したための弊害に過ぎない。

現在利用されているOSには必ずこの階層構造、すなわちディレクトリとフォルダーというデザイン・パターンで言うところのコンポジット・パターンを実装しているのであるが、このコンポジットという概念を理解しなければ、ファイル・システムを利用することはできないといっても過言ではないであろう。そこで学習者に対しては以下のようなことを実践してもらうこととなる。

◆デスクトップ上のコンピュータをクリックして、ファイル・エクスプローラを起動してください。それから今開いているコマンド・プロンプトのフォルダーへ移動してください。
（Vistaならば）Cドライブを選択して開き、そこからUsersというフォルダーを開くと自分のユーザ名がついたフォルダーがあります。それを開きます。
◆さて、次に「メモ帳」を起動して適当なファイルを作成して、それを先ほどのフォルダーに「test」と名前を付けて保存してください。
◆すると、ファイル・エクスプローラにそのファイルのアイコンが表示されますね。
今度はコマンド・プロンプト画面から「type test.txt」と入力してください。
先ほど「メモ帳」で記入した内容がそこに表示されるはずです。

ここで重要なのは、上記の３つのプログラムである「メモ帳」、「ファイル・エクスプローラ」と「コマンド・プロンプト」が全て同じディレクトリ（フォルダー）を開いており、その内容を表示しているという点である。学習者はこれらの操作から、OSがファイルを取り扱っていることとファイル・システムの片鱗を理解できるだろう。

この次にはmkdir（ディレクトリの作成）やcd（ディレクトリの移動など）、dir（ディレクトリ内容の表示）とrename（ファイル名の変更）、delete（ファイルの削除）などを学習してもらうわけであるが、その際にも上記の３つのプログラムを起動しておき、それらの操作がどのように各プログラムに反映されるかを観察させることが肝要である。ちなみに操作が即座に反映されない場合は、各プログラムにフォーカスをあて（マウスでクリックする）、Ｆ５のキーを押せば

「最新の状態」に更新される。そして次には以下の内容をコマンド・プロンプトから入力してもらう。

```
〉copy con memo.txt
Hello, world!
This is a message from command prompt.
^Z（これはコントロール・キーを押しながらZを押すことを意味する）
```

すると「一個のファイルをコピーしました」と表示され、ファイルが作成されるのであるが、次にこのファイルを「メモ帳」で開いてもらう。当然、上記の内容がそこに表示されるが、この一連の作業を通じてある程度ファイル・システムを理解するだろう。しかし、最後に前節で述べた「認識されない」ことも実践させる必要がある。そのためにコマンド・プロンプトから既にインストールされているアプリケーション、ExcelやWordなどを起動できるかどうかを確認させるのである。前節では「notepad」と入力することで「メモ帳」が起動したため、「excel」と入力すればExcelが起動すると予想するであろうが、実際には「認識されない」、つまり起動されない。これはパスという概念がファイル・システムにあるからであるが、この説明は難しい部類に属する。今現在いるディレクトリ直下のファイルしか認識されないだけではないからだが、ともかくExcelを起動してもらうために、（Vistaかつオフィス2007の場合なら）コマンド・プロンプトから

```
〉c: ¥program files¥microsoft office¥office12¥excel
```

と入力してもらい、無事にExcelが起動することを確認してもらうことにより、パスの一端を実感してもらうことが大事である。こんなに深い階層構造なんだ！というわけである。

プログラムを起動するためにはそのファイルが存するディレクトリの場所までを絶対指定しなければならないことや、現在コマンド・プロンプトが起動していないディレクトリのプログラムが起動できるためにはパスの設定を行わなければ

いけないことを理解してもらえればほとんどファイル・システムについてのメンタル・モデルが学習者に形成されるはずである。このパスの設定のことを「パスを切る」とよく表現されるが、これはOSに対して「行き先への経路（path）を切り開く」からの連想だと思われる。

しかし、さらに重要なことは名前空間と言う概念があることを提示することである。名前空間のお陰で名前の衝突がなく、異なる階層のディレクトリにおいては同じファイル名が利用可能となるのであるが、私はよくこのことをインターネットのホームページを特定するURLの例を持ち出して説明することにしている。つまりどのサイトのトップページもほぼ「index.html」というファイル名なのであるが、それがどうして別個のものとして把握されるか、ということである。このファイル・システムのパス名も分類をするための方法のひとつなのである。

6．バッチ・ファイル

ここまでで初学者にもOSが提供している「コマンド」や、OSで利用可能である「プログラム」が何を意味するかはわかってきたと思われる。残りはバッチ・ファイルであるが、実はプログラミング言語の入門に一番近いのがこのバッチ・ファイルである。

前節までで「memo.txt」というファイルが学習者のホーム・ディレクトリ（コマンド・プロンプトを起動した際にいるディレクトリ）に保存されており、かつ前述の３つのプログラムが起動しているものとして、以下の内容を学習者に入力してもらうこととする。

```
>copy con test.bat
time
date
ipconfig /all >> memo.txt
^Z（前述のようにコントロール・キーを押しながらZを押す）
```

次にファイル・エクスプローラ上で「test.bat」という見慣れないアイコンが

表示されていることを確認し、それをダブル・クリックして実行すると新たにコマンド・プロンプトが起動して時刻と日付を表示し（新しい内容の入力を促されるがエンター・キーを打鍵することで無視する）、そのまま起動された状態となる。ここで既に起動している「メモ帳」にフォーカスをあててF５のキーを押すことにより最新の状態に更新すると、再びmemo.txtを読み込み、新しい内容を表示する。ファイルの内容は先ほどのものにipconfig /all、つまり操作しているパソコンのネットワークカードの環境設定情報が追加されているものとなっているのだ。

このバッチ・ファイルは単純かもしれないが、コマンドやプログラムの「逐次実行」というプログラミング言語の重要な特性を表している。上記の例にある「リダイレクション」や例にはないが「パイプ」などOSが有する概念も大事であるが、プログラミング言語の学習にはこのように「テキスト・ファイルで指示を記述する」という事実の方が重要なのであると私は考えている。すなわち、テキストという言語ないしは言葉の意味を理解しながらプログラムを構築する作業がプログラミングという作業であることの重要性である。プログラミング言語は言語であり、そこには意味空間が存するのである。

７．プログラムの翻訳者：コンパイラとインタプリタ

コンピュータのハードウェアの知識やブール代数などの基礎理論を疎かにしていい、というわけではない。それらの知識は非常に重要ではあるが、ここまでで初学者でもコンピュータやプログラムというシステムを構成するモノに対してのメンタル・モデルがかなり形成されていると思われる。しかし、プログラムを作成するための基本となるプログラムの翻訳者の役割についてはまだ理解できていないであろう。特にバッチ・ファイルならばそのままテキスト・ファイルに記述するだけで実行が可能だという実例を示したばかりだからである。

ここで強調しなければならないのは、バッチ・ファイルはユーザが逐一入力してOSに指示を渡す手間をまとめるためにテキスト・ファイルに記述したということであり、プログラム・ファイル（ソースコードというが、これは実行可能な形式になっているファイルとその内容を記述しているファイルとの相違点を明ら

かにするために用いられる。一般には実行可能なプログラム・ファイルは単にプログラムと言われる）も確かにテキスト・ファイルであるが、意味が異なるという点である。特定のプログラム言語で記述されたソースコードはそれを理解する翻訳者によって、OSが理解できる形式に変換されなければならないということだ。またさらに、その翻訳者もプログラムなのである。

実はバッチ・ファイルが単なるテキスト・ファイルであるにもかかわらず直接OSから実行できるのは、OSもプログラムだからである。それはOSが実行できる範囲内で記述しているからであり、「１＋２」のようにOSにとって意味をなさないものは単純に無視される（無視される、ということは重要である。無視されずに別の解釈が行われると弊害が生じる可能性があり、特にOSというそのコンピュータの環境を決定するプログラムの場合には致命的な弊害を生じかねないからである）。OSはたくさんのプログラムから構成されているのである。

意味をなさない、とは解釈可能な意味空間の範疇にないということであるが、プログラミング言語も同様な空間を形成しており、例えばC言語のソースコードはJava言語の意味空間にはなく、結果としてそのソースコードはJava言語のコンパイラでは理解不能となり実行できる形式のファイルを構築できないことになる。逆もまた真であり、これは「特定の言語は特定のコンパイラ（またはインタープリタ）だけが解釈できる」ことを意味する。

従来ならばここで、コンパイラとインタプリタの違いを詳細に説明したであろうが、現在ではそのようなことはあまり関係がないと思われる。どちらにしてもコンピュータが解釈して実行可能な形式のファイルに意味を与えるのであれば同じだと思うからである。この顕著な例はJVMなどの仮想マシンやWebコンテナなどである。詳細は省くがこれらには抽象化と隠蔽、レイヤーの多層的な利用と応用がある。今後はCPUの性能向上などにより、これらの区別がさらに重要視されなくなると予想されるが、どちらにしても単なるテキスト・ファイルをコンピュータで実行可能にするための翻訳者はプログラムであることに変わりはない。留意しなければならない点は、プログラムである以上、その意味空間には制約と限界がある、ということである。

8．プログラミング言語の言語的側面

言うまでもなくプログラミング言語は人工言語であり、特定の目的のために設計されており、その目的とはコンピュータ上で実行可能な手順を記述することである。それならばひとつだけの完全なプログラミング言語だけがあれば、十分ではないかと思われるだろうが、この点も初学者にとっては理解が困難なことのひとつとなっている。確かにFORTRANは科学計算向け、COBOLはビジネス向けのプログラミング言語ということを教われば、自然言語と大きく異なるのはわかるだろうが、なぜそのような区分が必要なのかは理解しづらい。そればかりでなく、自然言語にも「～向け」があるのではないか、との錯覚に陥る可能性すらあるのである。もちろんそのようなことはない。

浅学にして最新の言語学でどのような研究成果があるかは知らないが、ある自然言語が別の自然言語と等価である、との証明はできていないだろう（もし証明されているならば、完璧な翻訳ソフトウェアができているはずである）し、言語や語彙の過多によって思考が規定されるという説を受け入れにしてもある自然言語が特定の目的には適している、とは言えないだろう。しかし、人工言語であるプログラミング言語は異なっているとされているのは記述の容易さ、という点なのである。

コンピュータ上で実行可能な手順を記述する、という意味においてはすべてのプログラミング言語は等価である。どのプログラミング言語を使用しても特定のアルゴリズムを記述でき、作成者の目的を達成することができる。ただし、ゲーデルの不完全性定理によりそれが「完璧」かどうかは別の問題であるがこれに関しては後述する。

どのプログラミング言語を用いても目的を達成できるのならば、どれを選択してもいいはずだし新しいプログラミング言語を作る必要もないはずだが、そうではないのには理由がある。それは記述の容易さは理解の容易さと設計の容易さに通じるからである。いや、逆かもしれない。理解と設計の容易さが記述の容易さになるわけである。このパラダイムの転換がプログラミング言語の進化を促進する原動力になっていると考えられる。この進化とは自然言語の変遷とは全く異なるものである。

この点を強調するために最近の日本語の「全然」を取り上げてみたい。話し言葉では「全然おいしい」は現在では意味をなしているし、理解が可能と思われる(信じられないが)。ところで、通常はこの「全然」は英語では「not at all」と訳されるものであるが、翻訳ソフトウェアならば「It's delicious, not at all!」とは訳してはいけないものだろう。しかし、この「全然」を「すごい、すごく」と解釈すれば、「It's very delicious」となるわけである。この曖昧さが自然言語の多様性と拡張性を表している。ここには論理はないが、意味空間があると考えられる。シニフィエとシニフィアンが一対一の対応をなさない動的な場、という自然言語がそこにある。

それに対してプログラミング言語は別の意味で変遷をする。前述のように抽象化と隠蔽を推し進めることにより、記述を容易にするための進歩なのである。アセンブラなどの低水準言語を別にしてもFORTRANなどの高水準言語が一般的に「～向け」と言われるのは、そのプログラミング言語を設計した際に想定された問題領域があり、その領域特有の解法を記述し易くしたのに過ぎないと言えよう。表層的に言えば、FORTRANには複素数を、COBOLにはデータ構造と帳票類を簡単に記述できる手段が内蔵されている。しかし、だからと言ってCOBOLで複素数を扱えないわけでも、FORTRANにおいてはもちろんのことであるがデータ構造を扱えないというわけではない。

C言語が登場する1970年代初頭までは、このように様々な数学モデルに基づいた特定分野での解法を容易に記述できるプログラミング言語が開発されて利用されていった。世界に２番目に古い高水準言語とされているLispは、人工知能や言語の構文解析用に様々な改良や変遷があったが今でも利用されているし、初心者の学習用にと開発されたBASICも同様である。状況が変わったのは、C言語の登場からでありそれは特定の分野用に設計されたわけではなかった。敢えて言うならば、それは全ての領域で利用可能な言語ということになるが、当初からそのようなことを狙って開発されたわけではない。開発者（デニス・リッチー[(3)]）が従来にはないものを望んだから開発した、というのが真相であろう。

ところで高水準プログラミング言語には予約語というコンパイラやインタプリタが予め設定している特別な意味を有する単語がある。具体的にはこれらの予約語を変数名や関数名などプログラム作成者が使えない単語類のことをいうが、プ

ログラミング言語の改良や変遷に伴い増減はあるにしても大体は一定数であり、C言語ならば37個でありJava言語ならば50個程度である（バージョン・アップする度に増加している）。すなわち、中学校一年生で学習する英単語の数よりも少ない。これら少数の予約語とそのプログラミング言語特有の文法に基づいてプログラムを作成することになるが、初学者にとってはこの事実が福音ともなるし悪夢のようにも感じられるようである。いわく、「覚えることが少なくていい」とか「少ない部品を組み合わせる方法がよくわからない」などである。

ここで重要な事は、先述の自然言語の例の「全然」のように予約語の意味は文脈や時代によって変化しない、という点である。もちろんその特定のプログラミング言語の言語仕様が更新されることはあるだろうが、下位互換性の確保のためにその意味が全く異なるものを意味するようにはならない。「if」はいつでも「if」として使用できるようにしておかなければならないのである。たとえその特定のプログラミング言語のコンパイラやインタプリタの実装が変更されたとしても、である。そのために自然言語とは大きく違い、文脈における微妙なニュアンスなどはないのであるが、意味空間における微妙なニュアンスは存在しており、そこが学習者にとってハードルとなっていると言えよう。数学とは異なり、答、つまりソースコードの書き方そのものが複数存在する可能性があるし、コンパイラからエラー（文法的なミスの指摘）を受けなくても自分の予定した通りの動作や処理をしない、などが多々あるからである。

このように、自然言語とは大きく性格を異にするプログラミング言語であるが、自然言語と同じように複雑さから解放されるための手段として、イディオムやフレーズといった成句も多数ある。母国語や外国語を学習する場合と同じく、当初はすべて初めて聞いた単語の羅列だと思われるものが段々とあるまとまった単位として利用されていることを理解するのと同じように、プログラミング言語にもイディオムやフレーズは数多くあり、それらを幾つ覚えているか、把握しておくかでソースコードの作成に差ができてくる。実は予約語を適当に組み合わせてもソースコードの基本部分となるわけではなくて、このイディオムやフレーズこそが基本部分となる。さらにこの基本部分がどのようにあるソースコードで使われているのかを見極められるようになると、そのソースコードの作成者の意図も簡単に理解できるようになるのも自然言語と同じことである。

では学習者、特に初学者にプログラミング言語が持つ自然言語との相違点を克服してもらうためにはどのようにすればよいのかを次に考えてみたい。

９．エディタの活用

前述のようにソースコードはすべて人間が可読な状態のファイルであり、テキスト・ファイルというものに保存されてそれをコンパイラに読み込ませて機械可読型、つまりコンピュータで実行可能な形式に変換される。すなわち、ソースコードの作成とはテキスト・ファイルを作成することに他ならない。テキスト・ファイルはWordなどのワープロ・ソフトでも作成可能であるが、一般的にはテキスト・エディタ（以下エディタ）を利用するが、それは特定の単語（キーワード）の検索や置換や複数のファイルを同時に閲覧できるのみならず、キー入力の操作自体を記録させて再び実行できる機能（キーボード・マクロ）などが内蔵されているからである。特に特定のプログラミング言語を使って作業する場合（ほとんどはそうであるが）ならば、そのプログラミング言語の予約語がカラフルに表示されたり、括弧の対応付けをわかりやすく表示できるようになっている機能が重宝されている。また、ワープロ・ソフトとは異なり左側の部分に行番号（ソースコードの構文上の行の番号）を表示できるため、講義で利用する際にはその番号を指摘できるために便利である。

現在ではIDE（Integrated Development Environment）という統合開発環境を提供するアプリケーションがあり、それを利用するとエディタやコンパイラのみならずテストやデバッグ（プログラムの不具合を修正すること）までその環境の中で行うことが可能になっている。しかし私は、特に初学者にはこのIDEの利用を勧めていない。それには二つの理由があり、一つ目はIDEの利用自体が初心者には複雑で修得に時間がかかるということであり、二つ目はIDEがあまりにも便利なためにその背後で何が起こっているのかを理解しづらい、ということである。特に後者は問題だと考えており、プログラミング作業自体がブラックボックス化してしまう危険性がある。とは言え、プログラミング作業の一連の手順（設計、ソースコード作成、コンパイル、実行、テスト、デバッグのサイクル）を理解した後はこのIDEを使う方がよりよいプログラミングの作業環境を手に入

れることになるので、積極的に勧めている。また、一連の手順を理解した後ならばIDEの各機能が何を表しているのかが直感的にわかるようになるため、習得に時間がかかることもないのである。

IDEを利用せずエディタでソースコードを作成する、ということは先述のコマンド・プロンプトを利用してのコンパイルや実行を経験することになるが、これにより一層自分が利用しているOSへの理解を深める結果となる。C言語以来、ソースコードに名前を付ける際にはその拡張子に一定の規則があり（.cや.javaなどにしなければならない）、それをしないとコンパイラが認識しないことなどや実行可能ファイルの拡張子にはいろいろな種類があることなどが理解されるわけである。

初学者向けには教科書などに掲載されているサンプルのソースコードを入力することから始めることになるが、ほぼ100％の学生はコンパイル時にエラーを発生させる。理由はほとんどが入力ミスなのであるが、その時にエディタの機能の一つである予約語の色分けや括弧の強調表示が役に立つことを実感させられることができる。また、エラーが発生するとそれに対するメッセージ（警告など）がコマンド・プロンプト上に日本語で表示されるが、その内容は決して親切なものではなく中級者でも理解が困難なものも多い。さらにたかだか十数行のサンプルのソースコードにおいてたったひとつの文字入力のミスでも十個以上のエラー・メッセージが表示される場合もある。それはたったひとつのミスにもかかわらず、そのミス以降のソースコードの内容に影響を及ぼしているからである。そのようなエラーに多く遭遇することによって、初学者はその特定のプログラミング言語のコンパイラと言語仕様に親しむことができるのである。私の経験では、全くエラーを出さずにサンプルのソースコードを入力できた学習者はその後、自分で作成したソースコードの間違いを訂正するのが困難な場合が多い。そこで私はこのことを「コンパイラから教わる」と表現している。あえてコンパイルが成功したソースコードをわざと手を加えてミスを導入して、コンパイラからどのようなエラー・メッセージが返ってくるかを観察するようにさせている。特に括弧をひとつ無くしてみる、などである。初学者の場合でも自分がどこでわざとミスを発生させているのかわかっているため、安心してエラー・メッセージを表示させてコンパイラが何を「教えてくれる」かを学習している。実はIDEを利用すると多く

の場合においてこのような基本的なミスやミス・スペリングを指摘してくれるし、親切なエラー・メッセージを表示してくれるのであるが、最初からそれに慣れてしまうと「荒れた」ソースコードを書く危険性がある。この「荒れた」ソースコードを書く癖がついてしまうと、他人に理解しやすいソースコードを書けなくなる。ところで「半年後の自分は他人」である。実際、適切に書かれていないソースコードだけでなく自分が半年前に書いたソースコードでさえコメントなどがなければ理解しづらいことがある。

このコメントとはプログラムのロジックには直接影響を与えないメモ的な記述のことであるが、ソースコードを保守するためには重要な役割を持っている。Java言語のように言語機能にコメントをまとめてマニュアル的なものを作成するものもある（JavaDocといい、ある規約を利用することでそれを解析してhtmlファイルを自動的に作成してくれる）が、そのような機能がなくても少し過剰とも思える程度にコメントをソースコードに書いておくことを勧めている。結局、システムは「利用され続ける限りは保守されなければならない」のであるし、保守するための一番いい手段はソースコードを直接読むことなのである。

以上のようにエディタを活用してプログラミング言語を学習していけば、「手が覚え」てくれるようになり、マウスなどのポインティング・デバイスはほとんど利用しなくなる。ミス・スペリングもなくなるなど、IDE活用の前にエディタの活用を強くお勧めする。

10. 声に出して読む（ソースコード・リーディング）

サンプルのソースコードを入力してプログラム作業の一連の手順を学習していくことは重要なステップであるが、私は入力する前にそのソースコードを声に出して読ませるようにしている。日本語の文章を音読するように、である。ただし、プログラミング言語は通常アルファベット表記であり、ほとんどが英語に近いものとなってはいるが英語の文章とは言い難いものであるため、一工夫が必要である。

例えば、C言語はUNIXと縁が深く、その文化を受け継いでいるために略記が非常に多い。多分C言語の設計者は単純にキーボードからの入力回数を低減させ

ることを狙ったのであろうが、UNIXのコマンドやプログラムにあるように「cat=concatenate」、「ls=list up」や「yacc=yet another c compiler」、「uucp＝unix to unix copy program」といった具合である。

一例として、下記のソースコードならば次のように読むように指導している。

#include〈stdio.h〉
 	
main( ) {
　printf("Hello, world!¥n");
　Return 0;
}
	インクルード　スタンダード・アイ・オー
	
	メイン　括弧　ブロック
	プリントフォーマット　括弧　ダブルクオート
	ハローワールド　ニューライン（改行）
	リターン　ゼロ
	ブロック閉じ

同様にJava言語の場合には次のようになる。

public static void main(String[] args)
{
　System.out.println("Hello, world!");
}
	パブリック　スタティック　ボイド
	　メイン　括弧　ストリング配列
	　アーギュメンツ
	ブロック
	　システム　ドット　アウト　ドット
	　プリントライン　括弧
	　ハローワールド
	ブロック閉じ

上記の例のように、予約語が本来意味している内容を補完しながら声に出させて読ませるようにしているわけであるが、文の終わりを示す「;（セミコロン）」は学習の初期段階は読ませるようにしている。ただし、流れが悪くなると感じるためにすぐに省略するようにしているが実際上で困るようなことはなかった。また、C言語やJava言語などで重要な構成要素を表す「{ }」は各々コンピュータ用語である「ブロック」と「ブロック閉じ」と強調するように心がけている。本来なら同様に重要な「( )」も音読すべきであろうが、これらは複雑な式以外は省略しているのが現状である。

難点は特にC言語のように略記が多い場合には読むのが長くなる、ということであるがそれは逆に次のような文を理解するには役立っていると思われる。

FILE *fp; fp = fopen("test.txt", "r");	ファイル　ポインター　エフ・ピー エフ・ピー　イコール　ファイルオープン テスト　ドット　テキスト　リード・モード

この例を字面通りに「ファイル　アスタリスク　エフ・ピー」などと読んでいてもそのプログラミング言語の背後にある概念を理解することにはならないと考えているからであるが、略記や略号の多いコンピュータ用語でもその完全名称を（たとえ覚えなくても）一度は知っておく方が理解に役立つのと同様である。

幸い、Java言語は歴史が浅く変数や予約語の文字数に制限が緩い時代に生まれたものであるためにほとんど自然な英語の文章のようにソースコードを記述できるが、それでも配列やブロックなどがあり、それらを意識して音読するようにしている。このお陰で結果的にエディタの予約語や括弧の強調表示と相まってミス・スペリングがなくなるし、それらの構文的な意味を比較的早く学習できていると思われる。このことは初心者向けのプログラミング言語の教科書は音読できるような内容が望ましい、ということを意味する。優良な教科書とは掲載されているサンプルのソースコードが前述のプログラムの基本部分（イディオムやフレーズ）が多いだけでなく、恣意的な変数名や関数名を用いていないということも重要な要素と言えよう。

11. 変数、型、メモリ

ここまでで初学者にもかなりプログラミング作業という一連の手順へのメンタル・モデルが形成されていると考えられるが、次の段階はこれまでは隠蔽されていた「メモリの利用」という概念を理解してもらうステップとなる。ハードウェアの知識を突然提示されて戸惑う学習者もいるかもしれないが、ハードウェアの基本は「中央処理装置、記憶装置、入力装置、出力装置、拡張装置」の五つを抽象化して教えることができる。それぞれの役割は単純化されているため、理解す

るのも困難ではない。

ところがここまでのプログラミング手順の学習は、ともすればその中央処理装置への指示だけを行ってきたという錯覚を学習者に引き起こす可能性がある。確かにプログラムとは中央処理装置への指示を記述しているものであるが、中央処理装置だけですべてを処理できるわけではない。重要なのは記憶装置、特に内部記憶装置（メイン・メモリ、以下メモリ）なのである。このメモリをどのように効率的に扱えるかで、そのプログラムが有効な処理をするかが決定すると言っても過言ではない。このことをメモリ管理というが、初心者の域を脱するとこのことがプログラム作成の一番の難点となる。コンピュータ・ウィルスや複雑なバグをもたらしているのもこの点であり、一般的にはメモリ・リークという言葉で知られている。それは自分の利用したメモリを管理できていないことから生じるのである（リークとは漏れる、という意味）が、後片付けができていないわけである。

これを克服するために最近のプログラミング言語では「ガーベージ・コレクション」という不要になった部分を自動的に処理する機能が備わっている。このお陰でメモリ管理を気にすることはなくなっているが、それでもメモリの効率的な利用は重要である。

初心者向けの教科書でよく見受けられる記述に、「変数とはデータを格納する箱です」というものがある。その上で、その箱に宣言された型の大きさのデータを入れて各種の操作が可能になることを説明しているのであるが、これはこれで正しい。そうでなければ「X＝X＋１」という数学的には理解不能なことを受け入れることはできないだろう（数学的には両辺からXを差し引くと「０＝１」になる）。これはXという変数に１を足す、という解釈を意味する。プログラミングではある変数に1を足す、ということはしばしば出現するため、数学的な意味を排除するために「x＋＋」という記述が考えられたのは自然であろう。

ところで「型」とはこれら変数を規定するプログラミング言語上の概念である。C言語やJava言語などは「静的な型付言語」と言われ、明示的に宣言された型を一貫して取り扱うように設計されている。そのためソースコードをコンパイルする際に型の矛盾を指摘され易くなっている。つまり、違う箱を利用しようとしてもだめですよ、と警告されるわけである。これは重要なことあり、でもしコンパ

イル時に警告されなければ実行時にエラーとなるわけである。事前に警告を出される方がいいのは当然であろう、実行するまで不具合がわからないよりは状況を改善する手立てが講じられるからである。

それに対して最近のプログラミング言語（RubyやGroovyなど）は「動的な型付け言語」と言われ、ソースコードの文脈から使用されている変数の型をいわゆるダック・タイピングにより判断するようになっている（ダック・タイピングとは「あひるのように鳴き、あひるのように動くのはあひるだろう」ということから、そのように振る舞うのは「それなのだ」と類推すること）。この場合もどちらがいいかはトレード・オフの問題となり、記述の容易さと後からエラーを修正することがそれに当たるが、基本は取り扱う変数や型がどのようにメモリに格納されているかに帰着する。この意味ではもう変数や型は「箱」にはなっていない。箱を意味するメタ・データを意識しなければならなくなっているのであるが、その箱がメモリ上に格納されてそれを操作していることに変わりはない。そのため、プログラミングとはメモリを操作する、という点を強調することは重要である。極論すれば、プログラミングとはメモリ上に意味空間を構築することなのである。特に最近のオブジェクト指向言語（C＋＋やJava）においては、プログラム作成者が独自に設定した型（クラス）を用いてそれらの実体（インスタンス）間においてメッセージをやり取りすることで機能を実現するのであるが、それらの実体はメモリに存在するためにこの点がますます重要になっている。

12.「これまで」と「これから」

私がこの小論を書こうと思ったきっかけは、「学習や理解とはインクリメンタルなものであるか？」ということであった。すなわち、ステップ・バイ・ステップで学習や理解は進むものであるかということであるが、私の経験では、そうではないと結論付けなければならない。ある段階から次の段階に進むのにステップ・バイ・ステップのこともあるだろうが、多くの場合はそこに大きな断絶があり、そのハードルを越えるために我々人類は様々な手段を講じてきたということである。特にコンピュータの分野では前述の「抽象化、隠蔽、レイヤー、互換性」を多用してきたのであるが、その反面、理解するのが困難になっていると思ったか

らである。これらのハードルを隠蔽することは決して学習には役立たない。意識せず利用するのに役立つだけである。確かにそれはそれで正しいだろう、すべての詳細を理解していなければならないとすれば、何もできないことになる。そこで、これらのハードルを「扉」に変換する作業が必要であると考えた次第である。扉ならば開けることも不必要ならば閉じたままにしておくことも可能だからだ。問題はその変換方法と順番である。

多少繰り返しになるが、抽象化には粒度を少し下げることで、隠蔽にはインターフェイスを適切に設定して公開することで対処し、レイヤーや互換性にはその利用方法と方向性があることを指摘することで、ハードルを扉に変換できるはずである。それに対して順番の方は一筋縄ではいかない。ここまで述べてきたような方法がベストである、と言い切る自信はない。が、扉を開けたままにして次の段階に進むのは問題であることは理解している。問題に対処するためにはある時はある扉だけを開ける必要があるだろうし、また別の場合にはその扉を閉めて他の扉を開けておくことが大事なのかもしれない。しかし、すべての扉をいつも開けておくのは複雑さを導入する結果になり賢明な方法でない。一度に複数の扉を開けたり閉めたりしながら、複雑さに対処することが肝要なのである。プログラミング言語を教えるとは、どの扉を開けてどの扉を閉めているのかに留意することだと考えるべきである。

プログラミング言語は、自然言語とはまったく異なる力学で進化する。それは省力化という推進力なのである。古くはアセンブラから高水準記述言語への記述と理解の容易さへの省力化であったし、C言語のようなキーボード入力と構文理解への省力化や、Java言語の自由度は下げるがエラーを回避するべく曖昧性を排除するための省力化とRuby on Rails[4]の理念のひとつである「convention over configuration[5]」という複雑な設定の回避などである。

これらの省力化も抽象化と詳細の隠蔽を導入する方向で進んできた。最近開発されてきているプログラミング言語は従来とは異なり新たな設計パラダイムは導入するものの、他の言語から完全に独立したものではない。例えば、groovyやScala、JRubyという言語があるがこれらは最終的にはJavaに変換されてJVM上で稼働する。それらの言語で記述されたソースコードはコンパイルされた後にはお互いを呼び出しあって協調して一つのシステムを構築することが可能なのであ

る。つまり特定の言語体系に束縛されることがないようになっているのである。

従来のプログラミング言語でも「車輪を二度発明しない[6]」ために、頻繁に利用される機能をサブ・ルーチンやライブラリ、クラス・ライブラリなどに、さらにはフレームワークの利用ということで省力化と安定性を向上させてきたのでる。それらはやはり特定のプログラミング言語を利用するということが前提であった。その状況が変化する兆しが登場してきた。実際私も過去にCOBOLやBASIC言語からC言語への翻訳を行った経験があるが、そのような不毛な作業が不必要な時代になったわけである。あるプログラミング言語でシステムを記述する、ということは時代の要請や流行があったのだが、それは本質的なことではなかった。プログラミング言語の等価性がより多くの交換可能性と自由度を獲得したと言えよう。これは抽象化とレイヤーによる、上手な隠蔽と省力化の見本なのである。

ここで重大な疑問が生じるだろう。つまり「プログラミング言語を複数、それも多数修得しなければならないのだろうか」ということである。答は「その通り」と、私ならば言うであろう。しかしそのことがプログラミング言語の学習のハードルを上げることを意味するわけではない。まずは特定のプログラミング言語をより深く理解することである。自然言語とは異なりプログラミング言語は、記述の方法や表現方法は異なっていても根本はほぼ同じであるからである。「分類することにより、理解は深まる」のであって共通部分と差異を理解することにより、どの言語が自分の解こうと思っている分野に適しているかを判断することは意外と簡単なことなのである。ここでもツールの選択（つまりプログラミング言語の選択）へのセンスが重要となる。どのようなツールが存在するかを知らなければ選択することはできないし、自分が立脚する場がなければ選択する基準がないからである。そのために複数のプログラミング言語への理解は必要であるし、立脚している場である特定のプログラミング言語を使いこなせることも必要である。つまり自分の「母国語」となるプログラミング言語の習得が重要となる。それが手続き型言語でもオブジェクト指向言語でも関数型言語、さらにはSQL[7]のような問い合わせ言語でもかまわないと私は考えている。要は基準となるプログラミング言語を使いこなせるようになることが重要なのである。

20世紀の初頭に人類の知を限定するような二つの発見と証明があったと言われる。一つは物理学のハイゼンベルクの不確定性原理であり、基本的な意味では未

来は予測できないという原理の発見である。もう一つは数学でのゲーデルの不完全性定理の証明であり、ある論理系が無矛盾かつ完全であることは証明できないということである。どちらか一方だけしか証明できない。これらの発見や証明があったにもかかわらず科学技術が進歩したのは周知の事実であるが、それはこれらが制限的に働きだす範囲が、前者は非常に微細な世界であることと、後者ならば複雑な論理体系であるからであった。そのために実際上は無視できたわけである。ところが、OSなどのように高度に複雑化されたシステムにおいてはこの不完全性定理が徐々に影響を及ぼすようになってきている。つまりそのシステムを作成したものが意図した機能を実現できれば必ずバグがあり、バグがないならば機能に欠陥があることを意味するからである。しかし、それでも人類は月に人跡を刻むことができたし慣性航法だけで無人探査機ボイジャーは木星と土星を観察して現在は太陽系外への航行を続けている。これらの偉業は古典物理学の範囲で計算され、様々な可能性を考慮して計画されたものである。つまりプログラミングされたわけである。もちろん、様々な可能性とはすべての可能性ということではない。それでも特定の目的を実現するための方法論は存在するわけである。

これからもプログラミング言語は進化するだろうし、開発されていくであろう。もしかすると究極のプログラミング言語は自分が解決したい特定の分野における問題を特定できて解決方法を記述できるプログラミング言語を作成可能なようにするものかもしれない。人工知能の開発は現状では実現されることを疑問視されているが、このようなメタ・プログラミング言語は開発可能であろう。そのメタ・プログラミング言語はやはり意味空間を有する「言語」なのである。究極のアプリケーションとも言える、つまり自分独自の問題を記述できて解決可能な言語を自然言語で記述できるようになるためにも、プログラミング言語の教育は必要であると考える次第である。

注

(1) Hayden White(1928－) 比較文学及び文芸評論家。metahistoryの提唱者として有名。

(2) ロジックやルールを表すモデル、表示部分を担うビュー、それらを管理するコントローラーからなる構造。もともとはGUI環境を構築するために考案された。

(3) Dennis Ritchie（1941－）：UNIXとC言語の設計・開発者。C言語の古典的名著「プログラミング言語C」の著者の一人である（もう一人はケン・トンプソン）。

(4) Ruby言語によるWebアプリケーションの開発を容易にするフレームワークの名称。

(5) ある命名規約に則ることにより、複雑な設定ファイルを不要にする方法のこと。

(6) 既に存在し、立派に機能しているものを再び作ることの無駄を戒める言葉の意味。

(7) データベースで利用されている問合せ言語で、以前は「Structured Query Language」の略とされていた。

(8) 1977年に打ち上げられた惑星の探査機。ボイジャー１号、２号があり両方とも同年に打ち上げられた。