入門教育向けプログラミング言語に関する調査研究

A Survey on Programming Languages Suitable for Beginners

平　岡　信　之*
Nobuyuki HIRAOKA

１．はじめに

筆者は今般、本学部で複数の非常勤講師の協力を得て担当しているプログラミング入門科目の教材の見直しを行う機会があった。それにあたって、扱うプログラミング言語についても再考することとし、その選定のために、昨今広く使われている主要なプログラミング言語を候補として選び、その特徴と動向について調査を行い、大学でプログラミングの初心者を対象とした授業で使うプログラミング言語という観点からの検討を行ったのでここに報告する。

もとより言語の良し悪しを客観的に比較する絶対的基準は存在せず、数多くある言語の中から最良のものを見つけ出すのは原理的に至難の業である。また、プログラミング言語の選択は、開発の現場では動作環境、要求仕様、開発環境、社会的環境（種としてプログラマの人的資源）といった諸要素の影響から逃れ得ないし、教育の現場でも同様の事情が勘案され、厳格に決定するとすれば非常に複雑なプロセスが必要となるだろう。そのため本報告では筆者なりの状況認識に基づいての指針を提出するところに目標設定を行なっていることを了解されたい。

２．調査検討にあたって

２．１　歴史的流れ

商品としてのコンピュータの初期の頃（IBMによる「アンバンドリング」以前）と、パソコンの初期の頃（ROMに入ったBASICがバンドルされていた、DOSの普及以前の時期）を除いて、概して20世紀のコンピュータでは言語処理系は有償ソフトウェアが主流であったが、世紀の変わり目の前後に２つの大きな動きがあった。１つは、OSの一部であるコンパイラ（CやC++）がオープンソースのOS（主にUnix系）の普及に伴って無償で使えるようになったこと、もう１つは、やはりUnixが関係する話だが、シェルスクリプトによるデータ処理を高度化する道具としてPerl言語が一人の先進的ハッカーにより作られ普及したことである（GNUなどのグループにより既存の言語のフリーな別実装はそれ以前からあったが言語仕様から新たに作る動きとしてはPerlが嚆矢だったと言っていいだろう）。その後、Perlに続くようにして様々な言語がオープンソースによる流通をし始めた。この動きをインターネットの普及が後押ししたのは無論である。

一方、OSの領域での大きな動きも上記とほぼ同時期に起こっている。すなわちWindows95をきっかけにWindowsがPC用OSのデファクトスタ

*企業情報学部准教授

ンダードとなったことである。大学の教育環境ではUnixやMacOSやNeXTStepを使ってその動きに抵抗はしていたものの、社会での動きに少し遅れて追随する形でWindowsが浸透した。生のWindows環境では前述のようなソースで流通しているソフトウェアを、コンパイルするための言語処理系（Unixや後のMacOSでgccやg++といったコンパイラmakeコマンド等のツール群）として決定版と言えるものが用意されてないので、Windows上で動作するソフトウェアはソースではなくバイナリパッケージ（Windows用インストーラ）で流通する必要がある。そのため、一般ユーザのPCのプラットフォームとしてWindowsが優位性を確保していたのに反して、開発系のソフトウェアの開発・流通はUnix/Linux系プラットフォームを前提としたものが先行する傾向がある。サポート体制の弱い大学では、世間でメジャーなWindows以外の環境を学生に提供することは困難だと言うこともあり、比較的最近まで、そういったオープンソースの言語処理系を積極的に採用できる環境になかった。

しかし最近になって言語処理系のWindows用バイナリーを早い時期にリリースするケースが増えてきて、Windowsベースの教育を行うサイトでも、制約が少なくなってきた。そして、こういった動向が総合された結果として、これまでパッケージソフトとしての言語処理系を、予算の制約のもと狭い選択肢の中から選んでいた教育現場においても、ふと気がつけばいつの間にか自分たちが豊かな選択肢を前にしていることを知り、しかしその初めて経験する状況において最適な選択をするための決め手に欠いている、それが我々の現在の状況だというのが筆者の認識である。

２.２　筆者の経験と学部の状況

筆者は約20年になる教員歴を通じて、初級プログラミング科目を断続的にであるが担当してきた。その期間に使用した言語は、以下のように変遷してきている。

a）Logo ～ C

タートルグラフィックスを通じて関数型コーディングを学んだあとCで実用プログラムに誘う、２言語を使った入門～実用への流れ。Logoはパッケージソフト、CはUnixホスト機にtelnetでログインしてccコマンドを用いた。

b）Visual Basic

教室のシステム入替えを機に変更した。基本的制御構造の理解のあと、イベント駆動のプログラミングでGUI型のプログラムを作った。Visual Basicは当初はパッケージソフトを用い、後には安価なアカデミック向けライセンス（媒体なし）で各受講者のノートPCにインストールして用いた。

c）Java

それまでは授業や担当教員によって（初級クラスの間でも）使う言語がまちまちであり、これにより教育の効率が低下するという可能性もあったため、学科内で言語を共通にしようという機運が高まったことと、CodeRallyを初級の教材として採用することになったため、そのCodeRallyで使われているJava言語を採用することになった。CodeRallyは２次元平面上で自動車のラリーを模したゲームにおいて、自分の持ち駒であるラリーカーの挙動をJavaでプログラムし、高得点を狙うというアプローチのプログラミング教材であり、ゲームプログラミングという領域に入門者を誘う面白いアプローチのものであるが、

・プログラムをスクラッチから作る楽しさを得られない
・不確定性に左右され、自分のプログラムの挙動にも再現性がない

という問題もあった。

今般、CodeRallyは使わないこととした。そのため初級プログラミング教育について初心に立ち返って、しかし今の時代に即した内容で新たに教科設計をせねばならない状況になった（機会を得た、という意味でもある）。使用する言語についても、後述のような要因はあるにしても、ともかく必ずしもJavaである必要はなくなった。そこで、扱うべき教材や内容も視野に入れつつあらためてJavaを使うという結論になる可能性もあるにせよ、使用する言語を決定するための準備的調査を行うことにした。これが本研究の動機である。なお筆者は上記のリストとは別に、他の科目との関係などの外部要因のない授業やゼミにおいては、

主にRubyを使って来ていることも付記しておく。

２．３　何を教えるか

手段が目的と化してしまう、という現象は我々の周囲でよく見られるが、プログラミング教育においても我々は常にその陥弊と隣合わせにいる。授業の場は、このプログラミング言語「を」教える（学ぶ）のでなくこのプログラミング言語「で」教える（学ぶ）場である筈なのだが、言語（の文法や使い方）を教える（学ぶ）ことの苦労が大きいと、それを乗り越えるだけで達成感を感じてしまうことになる。特に文法的制約の強い言語をサポートの弱い開発環境で使うと、教えられた長いプログラムを正しく入力し、コンパイラを無事にエラーなく通過する、それだけで授業が成立してしまうというようなことがありがちである。こういった作業を通じてコンピュータが融通のきかない機械であることを体験によって知る、という価値や、苦労したあとに喜びがあるというような議論もあるが、授業の設計や環境整備は、その苦労だけに終わらせないように留意し､その言語「で」何を教えるのかを見失わないようにすることが重要である。

その教える内容について論じるにあたって、プログラミング教育（もしくは情報教育全般）の目的意識として、今２つの大きな流れがあると考えている。１つは、実用につながる、技能・技術としてのプログラミングであり、もう１つは、筋道だてて思考を組み立てる技術、いわば教養としてのプログラミングである。その２つは、根っこは同じであり相反するものではないが、現実の技術との距離感の差から話が食い違う危険性もあるので、ここで筆者の立場は示しておきたい。筆者の属する学部学科は入試分類上は文科系には入るようだが情報技術を教育の１つの柱に据えた学科であり、プログラム入門科目を土台として他の科目やゼミナールで様々なプログラミングの営みが教育・研究として行われている。その入口となる科目である以上は、現実の開発に役立つ実用的教育（技術としてのプログラミング）が必要だと考えている。その立場から、入門の段階でどんな内容を扱うべきか、勿論これは今後も継続的に実践に並行しての検討を重ねていくべきものだが、現時点で想定している内容の概要をここに記しておく。

a）まず楽しさを感じる

b）プログラミングの考え方を習得する
（これまでのプログラミングパラダイムの変遷に対応させて）
・逐次型の枠組みの中で、GOTOを使わない構造的プログラミングと、アルゴリズムの考え方
・オブジェクト指向の考え方と、クラスおよびメソッドの活用、クラスの作成
・関数型の（概してコンパクトになる）記述と、再帰の考え方
・パターンマッチングの活用
・高度なデータ構造として、
　連想配列（またはハッシュ）
　伸縮可能なコレクション（Vectorなど）
　タプル
・型の考え方

c）プログラムの使われ方を知る経験として
・CUIベースのプログラム
・GUIを使ったプログラム（ゲーム等もこの分野で扱うことができる）
・多様なプラットフォームを知る
　Webサービス（フレームワークを活用して）
　クロス開発（タブレット端末用アプリ等）

２．４　調査の前提

入門教育に関する研究として、初心者にとっての理解度を深めるため新たな言語および実習環境を設計し実装するというアプローチも数多く行われている。しかし筆者はその方針を取らないこととし､既存の言語の中に候補を求めることとした。使用する言語を決めた上で、実習環境を整備したり、その言語での経験を重ねていくにあたって言語仕様の全体像が大きすぎることによる弊害（初心者が途方に暮れる）を軽減するための段階的提示の工夫など、実装研究テーマは次のステップとして想定されるが、現段階ではまず言語を選ぶことに専念する。

現実には膨大な数のプログラミング言語が現存している（ちなみにWikipediaで数え（させ）てみたところ261項目あった。数えるためにRubyで書いたプログラムを参考として図１に示してお

リスト１　言語の数を数えるのに使用したプログラム（Ruby）

```
#!/usr/bin/ruby -Ks
require 'kconv'
require 'uri'
require 'open-uri'
url='http://ja.wikipedia.org/wiki/'+URI.escape('プログラミング言語一覧'.toutf8)
n=0
open(url) {|f|
  f.each_line {|line|
    break if line.include?('関連項目'.toutf8)
    n+=1 if line =~ /<li>/
  }
}
puts n
```

く）。こういった言語をすべて網羅的に調査することは時間と手間の点で当面の目的に合致しない。そのためここでの調査は（雑駁な基準であるが）「有名な」言語に限定することにする。ここでの目的において非常に優れた言語がその枠外に隠れている可能性はあるにしても、知名度・普及度の低い言語の場合、活用例やトラブル対策に関する情報源が乏しくなり（その逆の因果関係もある）、科目で採用するにはリスクが大きいため、この基準は妥当なものだと考え、普段プログラミング言語に関心を持って暮らしている筆者のアンテナにこの２，３年の間に引っかかって来ていない言語を今般広く収集しなおすことは避けた。その中から、使うべき言語を選択するにあたって、以下のような論点や留意点があるだろう。

a）知名度・普及度

ネットや書籍での情報量。日本語での情報のあることも現実問題としては重要だろう。言語のシェアや人気については、様々な調査があるようだが、調べ方や調査時期によって結果は一様ではないので、参考にはするが依拠しすぎないようにする。また、他学での採用状況には関心は払わないこととした。

b）入門用の言語を用いるか、実用言語で入門するか

入門用に簡易化された言語を用いると、その時点で言語について学ぶべきことは少なくなり、プログラミングに関わる概念理解に注力できるため、学習者が踏み上がるべき階段の段差は低くなると考えられる。しかし実用プログラミングの段階に入ると、入門で学んだ言語だけでは不十分な時期が必ず来るだろう。それは学部教育の間かも知れないし社会に出てからかも知れないが、どちらにしても実用言語に再入門する際に２つめの段差を踏み上がらなければならないことになる。技術としてのプログラミングを習得するために大きな段差を、１回で上がるか２回に分けて上がるか、という選択だと考えることができる。最初から実用言語を選ぶとしても、言語単独の機能特性だけに注目するのではなく、現実的な問題として、卒業後の実際の仕事の場所で使われるメジャーな言語、プログラミングを含む資格試験での選択肢、学部教育の中で（入門後の他の科目で）使う必要性が高い言語、といったいわば社会的要因についても配慮し、２つめの言語への継続性を考慮する必要があるだろう。

c）型付が動的か静的か

一般に動的型付を特徴の１つとする言語がスクリプト言語と呼ばれているため、この問いかけは

スクリプト言語かコンパイラ（向け）言語かという問いに呼び換えても大差はない。変数に代入する値、関数が返す値、関数の引数に与える値などのとるべき型を実行時に処理系が判断するか、予めプログラマが算譜上で指示するかの差異であり、前者はコーディング時の負担を軽減するのに対し、後者は実行効率やプログラムの安全性の点でメリットがある。型の概念を学んだり、ポリモーフィズムによって同名のメソッドが多数ある中から使うべきものを選んだり、型の不一致を回避するためにキャストしたり、といった静的型付言語に特徴的な作業はプログラミングの初学者を混乱させる可能性があり、また、型を考慮するぶんだけソースコード長くなる傾向もある。なお、長くなるソースコードについては、IDEが入力を支援するのでタイピング上のコストは増えないという議論もあるが、初学者には最初にサンプルプログラムを読んで理解を図るというステップがあり、IDEはその読解の助けにはあまりならないと考えられる。これを考慮すると、入門時には動的型付言語のほうが敷居が低いと判断できる。ただし、実用プログラミングに移行する頃には型の概念を学んで積極的に活用することは必要になる。

d）ツールや環境の充実

コンピュータが憧れの対象であった昔とは違って、数式や文字列処理といったいかにも学問的な題材を使った授業に禁欲的に取り組めるような学生は希少になった。授業に学生を惹きつける題材として、ゲーム感覚のプログラム作りは有効である。そのためには、プログラムで図を描けるキャンバスと、インタラクティブなGUI操作のためのインタフェースが平易な文法で用意されていることが望ましい。描画の方式として、xy座標による描画（カーナビで言えばノースアップに相当するイメージ）とタートルグラフィックス（ヘディングアップに相当）の２つが知られているが、その進化したものとしては３Ｄモデルの描画や、スプライトを用いた動く物体の表示支援まで期待したいところではある。また文や式を行単位またはブロック単位でインタラクティブに入力し評価結果を印字する対話環境として、REPL（Read-Eval-Print Loop）が動作することも、初学者には重要な要素だろう。その他、複数のペインに分割された１つの窓で編集から実行までを行える統合環境IDEも必要である。ただし、いきなりプロ仕様のIDEを充てた時に初心者は機能が多すぎて混乱するので、最初は大規模プロジェクト開発の支援などの高度な機能は削ぎ落とした簡易版が望ましい。さらに、多様な環境への移行のサポート（クロス開発など）も行われていることや、パッケージマネージャによりライブラリの利用が楽になっていることも、チェックすべきポイントの１つにあげておく。

３．調査と検討

３．１　プログラミングの現況と分類

１）実行形態

筆者がプログラミングを初めて学んだ頃は、プログラミング言語はコンパイラ言語とインタプリタ言語に大別されていた。前者はソースプログラムから処理系を通して実行形式のファイル（オブジェクトプログラム）を生成するもので、後者はBasicやLispに（当時）代表された、インタラクティブに一行ずつ命令や式を入力し結果を返してくるものである（このリアルタイムなインタプリタを最近は前述のようにREPLと呼ぶ）。その後、ソースプログラムを処理系が即座に実行するスクリプト言語（インタプリタが実行するのでこれも広義のインタプリタ言語に入れられる）や、コンパイラがネイティブなオブジェクトプログラムでなく中間言語や仮想マシンのオブジェクトプログラムを生成する構成のものも台頭してきて、実行形態が多様化し現在はこの単純な二分法は意味をなさなくなった。さらに、コンパイラ言語は概して実行高速だが編集から実行までの操作が煩雑でターンアラウンドタイムが長いという認識があったが、統合開発環境（IDE）の進化によりどの言語でもコンパイル→オブジェクト形式→実行 という流れを意識しないようになり、IDE（Eclipse等）がオンザフライでエラーチェックや中間コードへの変換を行うなどターンアラウンドタイムも短縮されており、実行形態の違いは、使い勝手にはさほど影響はなくなってきた。

２）プログラミングパラダイム

一方のスクリプト言語（日本ではLL言語と分

類されることもある）は、性能に難があると言われていたが、ハードウェアの恒常的進歩と、実行時間よりも開発効率が重視される領域が拡大していることにより、存在感が大きくなってきている感がある（プログラミング言語が①性能、②記述性、③安全性の３条件の適度な妥協点に置かれるとしたとき、②寄りのポジションを占めることになる）。

一方、プログラミングのパラダイム（基本的な概念）に目を向けると、これまでに手続き型、構造化プログラミング、関数型、オブジェクト指向、といった考え方が（概ねこの順に）提唱されてきて（これがすべてではないが本研究で扱わない分類はここでは省略する）、そのうちのどのパラダイムを主眼として作られた言語であるか、に沿った分類は行われて来ている。しかし最近の言語は殆どが何らかのオブジェクト指向をサポートしており、勿論その中に手続き型や構造化や関数型の要素も兼ね備えているため、パラダイムによる大分類も意味を持たなくなった。言語の差異は、属するパラダイムではなく、どのパラダイムのどの要素を強く押し出しているか、というややミクロな視点での差異になってきたと考えられる。

ここでは一例をあげておく。生来の関数型言語は当然として、多くのスクリプト言語が積極的に採用してきたのが、繰り返しや条件分岐の構文が式として値を返すという構文要素である。for式、if式、while式、case式といった呼称が与えられ､それらの式は関数と同等に扱われる｡こういった関数型の機能を充実させた言語では～文という文法要素がなく統一的に～式と呼ばれる。これに高階関数を扱う機能を組み合わせて使うことにより、純粋関数型言語でない言語でも、旧来の手続き型の記述を極力廃したプログラムを作れるようになってきている。

オブジェクト指向言語に関しては、すべての機能が何らかのクラスのメソッドといて実装されていてプログラムを動かすために少なくとも１つのクラスを作成することを要求する厳格なオブジェクト指向言語（Javaがその例）がある一方で、どこからでも呼べるオープンな関数が用意されていて、とりあえず関数を呼ぶだけで（クラスやオブジェクトを１つも生成しなくとも）プログラムが動かせる緩いオブジェクト指向言語もある。
本報告では前者を「閉じたオブジェクト指向」後者を「開いたオブジェクト指向」と呼んでおく。言うまでもないが開いたオブジェクト指向のほうが入門には適している。

３）表記のバリエーション

新しい言語を設計するにあたって従来の慣習を無視することは普通はなく、旧来の言語の慣習の中から選択的に踏襲することが多い。それでもその選択のやり方によって表記の方法に様々なバリエーションが発生する。細部にわたるリストアップはスペースの都合上できないがその要点をここに示しておく。

a）文の単位

①セミコロンで終止させることを原則とするもの（C, Javaの流れ）②文末またはセミコロンで終止するもの（多くのスクリプト言の語）に大別される。後者の場合、開いたままの括弧類や、解決されないカンマや２項演算子により複数行に渡る文の継続を示すのが通常である。これに加えて、③インデンテーションにより文の構造を示すもの（Python, Haskell, CoffeeScript）もある。

b）変数名

一般に変数名などの識別子は英文字で始まり２文字目移行は英数字および一部の特殊文字で構成される文字列である。使用できる特殊文字は言語によりまちまちだが、C言語以降の演算子を積極的に活用する言語では特殊文字は演算子に使われるためきわめて限定されることが多い（例えばRubyでは _ のみ）。1970年（UnixとC言語の出現）以前の言語の慣習を継承する言語を除いて、識別子の大文字と小文字を区別する（case-sensitive）言語が殆どで、語頭の大文字を積極的に活用して意味を持たせる（定数、クラス名など）ことが多い。また複数の単語を連結して１つの識別子を構成したときにその単語を読者が認識しやすいよう各単語の先頭文字を大文字にするという慣習は言語を問わず根付いている。他に、識別子の前に特殊文字を１つ付加してその識別子の属するクラスを明示するという仕様の言語もある。PerlやPHPではすべての変数に、Rubyではローカル変数以外の変数に、前置される。Rubyの場合はローカ

ル変数の比率が高いことが多くさほど目立たないが、Perl系言語でこの特殊記号の多い譜面は他の言語に慣れた目からは違和感が大きくなる可能性がある。なお、上記の英文字ルールとは別に、(文字列やコメント中では当然として) 識別子に日本語文字を使える言語も多い。

c) 関数 (メソッド) と引数の関係

①関数名の後に括弧‘( )’を置きその中に引数を列挙する書き方が大勢を占めるが、②関数名の後にスペースを挟んで引数を並べる (もしくは引数の間をカンマで区切る) という記法も関数型言語やスクリプト言語には多い。②の場合、構造の曖昧さを防ぐために括弧の使用も可能なので後者の言語では①②の両方が使えることが多い。他に (関数名 引数 ...) のように全体を括弧で包むLisp系の書き方や、関数名を後置する記法 (Forthの流れを汲むスタック言語) もある。前置記法の慣習が定着した背景には印欧言語の述語+目的語の語順との整合性があると考えられるが、後置記法はパーサを単純化できるメリット以外に、データの流れや処理の順序 (引数を評価してから関数を実行する) を左から右への流れとしてイメージしやすい (日本語などの語順との整合性も強い) 点でもっと大きく扱われていいのかもしれないが、現時点で前述の社会的要因を考えて大勢に従い、あえて後置記法には踏み込まないことにしておく。

d) 型指定

静的型付を行う言語では変数、関数 (の戻り値)、関数の仮引数などの宣言では名前と型を指定する。C, Javaの系列の言語では、型指定を (他の修飾語とともに) 名前の前に置き、Scalaでは間に ' : 'を挟んで後置する。Haskellではそもそも変数の概念がないが、関数のシグニチャは関数定義に先立って独立した行に記述する (ことができる)。Haskellおよび型システムについてその影響を受けたScalaでは、型推論機構があるため多くの場面で型指定は省略することができる。この前置後置の違いは人によって瑣末なことだと受け止めるだろうが、型指定を前置しない文法の場合、関数名やクラス名の開始位置がソースプログラムの定まった位置に揃い、ソース内を目でスキャンする場合に負担が少なくなるというメリットがある。

e) 括弧類

ここでは括弧類として左右同型の引用符も含んで扱う。文字列はほぼ例外なくダブルクォーテーションで囲むのが標準となっている。シングルクォーテーションで何を囲むかは大別して２つのものがあり、①文字を囲んでいる言語 (C言語の流れ) と、②やはり文字列を囲むが、文字列内の置換機能に制限があるという言語 (Unixのshスクリプト言語の流れ) とがある。文字列や正規表現を表現する際にそれを囲む文字や特殊な文字の扱いが複雑になるのを防ぐため、(エスケープ文字は標準的に装備されているとしてその他に) 言語により様々な記法が導入されている。スクリプト言語のヒアドキュメント、Rubyの%記法によるリテラル、Scalaの三重シングルクオートなどがある。

プログラムは入れ子になったブロックとして構成される。前述のインデンテーションにより構造を表現する言語を除くと、そのブロックを囲む表現としてブレース { ... } が使われる言語が多い (C言語の流れ)。RubyではAlgolやPascalに似た do ～ end を併用することができる。

４) 未来に向けての準備

PCのハードウェア技術での最近の大きなトレンドの１つがマルチコア化である。このコア数増の動きは今後も続くだろう。インターネットや無線常時接続の普及によりネットワークアプリケーションの実装も多様化していくことが予想される。マルチコアと分散処理の時代に向けて、プログラミング技術のあり方も変わって来ざるを得ない。今後使われていく言語では、個々の処理を (明示的に逐次的である必要のあるものを除いて) 複数のコア或いは複数のサイトに分散して並行実行することを許すような言語体系が必要になるだろう。その並列実行はコンパイラの判断によるものとプログラマが明示的に指示するものがあるだろうが、前者については、データのコレクション (配列やハッシュなどのデータ集合) に対する、(副作用のない) 関数的処理や、もう少しミクロなレベルでは並行代入などもそういう扱いができる可能性が高い。後者については、マルチスレッドのライブラリは多くの言語ですでに実装されているとして、プロセス間通信のエレガントな記述ができる言語が今後は人気を得ていくことが予想される。

３．２　言語各論

候補とした、若しくは参照したそれぞれの言語について、以下に個別に論じる。掲載の順序としては、文章構成上の都合によって並べたものであり特に意味はない。

１）Processing [*processing.org*]

Proseccingは、言語でもあり実行環境でもある。言語としてはJavaのサブセットとして作られているが「開いたオブジェクト指向」である。記述力としてはさほどめぼしい特徴のないシンプルなものである。処理系はJavaで作られており、Javaのライブラリをimportして使えるが、ライブラリを作る機能はない。描画関係とインタラクションを行う簡易なメソッドが用意されていて、アート系・アミューズメント系のプログラムに簡単に着手できるのが最大の特徴である。スプライトは用意されていないが、最近の版では３Ｄモデルの描画も扱うようになっている。それ以外にデバイス操作のための機能も充実しており教育の場では高い活用性を持つ。

重要な2つのメソッドとして setup()とdraw()がある。setup()は最初に一回だけ呼ばれ、draw()は（コントロール可能な）一定間隔で繰り返し呼ばれる。この2つの関数により暗黙のくり返しやインタラクションを簡易に実現できる仕組みである。この基本構造はイベント駆動の枠組みの中でタイマーが発生させるイベントによるものと類似である。

以下の点で言語としてはJavaと同等である。

・静的型付けの言語であり型指定は省略不可
・制御構造もCやJavaと同じ。ただしfor-each構文はない。
・正規表現 は match()関数によって使う。
・thread 、catch and throw もある。

一方、擬似的多重継承（interfaceなど）はなく、複雑なクラス階層の設計・実装には不向きである。実用言語として使うことを想定したときの問題点としては以下のことが考えられる。

・Processing環境から外に出られない（外で動作するプログラムは作れない。ただしJava AppletとAndoroidアプリケーションの作成は可能になっている）
・言語仕様の不十分さ（末尾再帰の最適化は行なっていない、高階関数による関数プログラミング的記法が用意されていない等）

なお、Processingの教育環境としての価値が広く認識されているため、他言語でのProsessing的処理を行うシステムも以下のようにいくつか流通している。

・Python: NodeBox（Mac限定らしい）
・JavaScript: Prosessing.js（CofeeScriptからも呼び出せる）
・Ruby: ruby-prosessing
・Scala: Spde（他の例がAPIとして提供されているのに対して Spdeは環境として提供されている）

他に prosessing.core パッケージをimportすることでJavaを含むJVM言語からprosessingの機能を呼び出すことができる。

２）C++　　[*cplusplus.com*]

C言語にオブジェクト指向の機能を付加した言語としてC++とObjective-Cがある。後者はMacでの開発言語として（iOS向けクロス開発を含めて）昨今は再度注目を集めているがその普及の経緯（NextStep 〜 MacOS）からMacでのOS以外での環境は乏しい。C++の処理系はフリーのものも含めて多数流通している。CおよびC++は制御系ソフトやOS、言語処理系のベース部分などの基幹ソフトウェアの開発や、アプリケーションの分野でもネイティブコードによるオブジェクトプログラムを生成すべき場面では現在も広く使われている。プロセッサの動作に近い記述ができることと、コンパイラの最適化における長年の技術蓄積もあり、性能的には最も信頼されている言語だろう。しかしそれは裏を返せばオブジェクトや手続の抽象度の低い低水準言語であるということでもある。過去に授業で扱った経験の中でも、ポインタの扱いやメモリ管理などの概念は初心者には難解な事項が多々あり、言語「を」学ぶ授業に陥りやすい。組込み系ソフトウェアを扱うコースやコンピュータの原理を学ぶ課程では有効な言語だが、これからの時代のプログラミング学習はできる限り高水準な言語を使うべきだろうし、型システムの不十分な点からキャストが頻繁に発生するなど、新しい高級言語に比べて安全性にも不安要素はある。ここでは歴史的に重要な意味を持つ（後

発言語に多大な影響のあった）言語としてここで参照するにとどめておく。

3）Java　　[*java.com/ja*]

C言語がネイティブコード生成言語の業界標準であるのに対し、Javaは仮想マシン（Java Virtual Machine または JVM）をベースとした言語の標準の地位を確保している。リリース当初はブラウザHotJavaの中で動くJavaアプレットを生成する言語として知られていたが、その後Webサーバ、デスクトップ環境、組み込み系などに適用分野を広げて、現在はあらゆる所で動くポータブルな言語であり、そのシェアと適用分野の広さを考えた場合、前述の社会的要因としてJava言語の存在を無視することはできない。

その反面、言語としてのJavaは「閉じたオブジェクト指向」であり厳格な静的型付け言語である。そのことから、ちょっとしたプログラムを動かす際にも記述すべきコードの量は多くなる傾向がある。標準入力から入力された文字列をそのまま標準出力に印字することをEOFまで際限なく繰り返すプログラムを例にとると、rubyでは

```
#!/usr/bin/ruby -n
puts $_
```

の２行で記述できるものが、Javaで書くと概ね図２のようになる。膨大なクラスライブラリが整備されていることにより、ほぼ「何でもできる」言語という領域に達していると考えられるが、

・配列がArrayオブジェクトとして実装されていない
・多重代入がない
・制御構造が式でなく文（値を持たない）
・連想配列、ベクター（伸縮可能）などの重要なデータ構造が外部ライブラリにある
・高階関数の扱いが複雑

といった点を考慮すると、多くの場面で「書けるけど複雑」になる傾向もある。記述性の点から見ると、最近の言語に比べるとやや低水準な言語という位置にあるだろう。また、REPLは使えず、そういった点で、初学者のための言語としては不向きな点が多い。

なお、ポータブルな実行環境としてのJVM（その仕様もオープンになっており[*docs.oracle.com/javase/specs/*]）これを活用しつつ、Java言語より

リスト２　やまびこプログラム（Java）

```
import java.io.*;
class Echo {
        public static void main(String[] args) {
                try {
                        String line;
                        BufferedReader reader=
                                new BufferedReader(new InputStreamReader(System.in));
                        while((line=reader.readLine()) != null) {
                                System.out.println(line);
                        }
                        reader.close();
                }
                catch (Exception e) {
                        System.out.println(e);
                }
        }
}
```

も使いやすい言語でプログラミングしようという観点で、Javaと同様にJVMのオブジェクトコードを生成する言語が多数作られている（この後のリストにそのうちのいくつかを挙げる）。それらの言語から、Java言語向けのクラスライブラリをimportして使えるものが多い。従って、JavaはJVM言語群の中心的存在としてライブラリを提供する中核言語の位置は今後も占めていくだろう。

４）ruby　[*ruby-lang.org/ja*]

日本製のスクリプト言語であり、日本では人気が高い。国際的には知名度の点でPythonに遅れを取っていたが、2012年に国際規格にも認定され、今後の普及・浸透が予想される。開いたオブジェクト指向で、プログラムのブロックを引数として受け渡すことにより高階関数を使えるなど、関数型言語の記述にも適している。また動的型付けの言語で、クラスがオープンである（既存のクラスにメソッドを追加したり変更したりもできる）等、言語の内部への変更も自由にできる柔軟性の高い言語である。リスト１に実例を示したように、ちょっとした操作を短いプログラムで実現可能な、記述性の高い言語だと言える。REPLとしては、irbが標準添付されている。Windowsの処理系として、初学者にどれを勧めるべきか最近までは迷うような状態であったが、現在はrubyInstaller でほぼ落ち着いている。パッケージマネージャとしてRubyGems（コマンド名はgem）があり、このコマンドだけで依存関係の管理も行なっている。GUIツールキットとして、様々なライブラリが提供されてきていて、決め手に欠ける状態が長かったが、Ruby1.9では Ruby/Tkが標準添付となった。ただしGUIアプリケーションは今も重い傾向がある。Webアプリケーションを作る環境としては、Webフレームワークの見本となって他の言語のコミュニティにも影響を与えたRuby on Rails [*rubyonrails.org*] があり、タブレット端末向けアプリを作る環境として[*rhomobile.com/products/rhodes*]がリリースされている。学習環境としては Hackety Hack [*hackety.com*]や Shoes [*shoesrb.com*] が使える。Ruby 自体は和製言語であるがこういったアプリケーション群は海外製であり、表記や文書が英語であることは日本の学生に負担になるかも知れない（他の言語でも状況に大差はないが）。その他、rubyでゲームを作るための環境整備も（これは主に日本で）行われている。しかし有志によるプロジェクトが多く（例えば Miyako [*www.twin.ne.jp/~cyross/Miyako*]）その継続性は未知数である。なお、rubyの実装は現在は提唱者の手を離れて、複数のプロジェクトが存在しており、ネイティブアプリケーションとしての処理系以外に、JVMベースのJRuby [*jruby.org*]や.NETベースの IronRuby [*ironruby.net*] もある。

４．１）その他のスクリプト言語

ここではスクリプト言語を１つだけ選んで論評したが、ここで選ばなかった言語についても若干の補足をしておく。

a）Perl [*perl.org*]

スクリプト言語の元祖である。後の版でオブジェクト指向も採用し、今も進化の途上である。最新のPerl6では静的型付もオプションとして採用されている。Perlでは変数の先頭には必ず$または@といった文字が付加され、その文字によって変数がスカラなのかベクタなのかを区別している。さらに関数の動作はその結果が代入される先によってスカラコンテキスト、ベクタコンテキストに区別される。これがPerlの特徴の１つだが、その使い分けは便利だが解りにくく、初学者にとってはトラップの１つになる可能性がある。

b）PHP [*php.gr.jp*]

Webアプリケーションに特化した言語である。その領域での手厚いライブラリ群の整備により、Web開発では人気が高いが、言語としてはさほどの先進性はなく、ユーザが手元で使うようなアプリを作る環境は整っていない。

c）Python [*www.python.jp/Zope*]

前述のようにスクリプト言語としては世界的に認知度の高い言語であり、Sketch [*bohemiancoding.com/sketch*]、Sphinx [*sphinx.pocoo.org*] など Pythonベースのアプリケーションも多数世に出ているが、記述力としてはRubyとほぼ同等だと考えられる。ここでは日本での普及度と、Pythonのインデンテーション文法が他言語との距離を感じさせる点を考慮して、スクリプト言語の代表としてはRubyを第一候補としておいた。

５）Haskell [*haskell.org*]

関数型言語の中ではこのHaskellを代表として考察する。Haskellは純粋関数型言語に分類される。すなわち、関数の評価は副作用を伴わず、変数は代入後の値の変更ができない（他の言語では定数として扱われるものだがHaskellでは変数と呼ばれている）。Haskellの特徴として３点をあげておく。

a) 遅延評価：　無限リストを関数の引数として与えることができる
b) 型システム：　Haskellは厳格な静的型付言語であるが、型推論を装備しているため、プログラマへの負担は軽い。
c) モナド：　実用プログラムとして副作用を引き起こす必要のある箇所（入出力など）を扱う特殊なクラス

この特徴により、シンプルな文法でいながら高度なプログラムを記述できる言語である。ただし、従来のプログラミングの概念に慣れた頭には、発想の転換が必要で、理解しづらい点が多々ある。初学者が最初に出会う言語として純粋関数言語が適切かどうかという議論はあるだろうし、教え方の実例も経験も少ないという問題は現時点では不可避である。実用上の問題点としては、大規模なプログラムになった際の名前空間の適切な管理方法が（筆者の調べた範囲では）不明で、混乱を招く可能性があることであろう。関数型言語にありがちな性能上の問題も指摘されているが、簡易な性能実験報告を眺めている限りでは良好な結果を得ているようである。プログラミング教育においてすぐに採用できる状態にはないが、プログラミング言語としては完成度の高い体系のものであり、学ぶことの多い言語であり、未来のプログラミング言語の標準の１つとなるだろう。

なお、関数型言語としては、他に以下のような言語もあるが、ここではHaskellに注目した。

・Erlan [*erlang.org*] ： 動的型付けに基づく純粋関数型言語。1992年頃から使われている、VMベースの言語としては先発になる。Hasellとは違い、正格評価を行う。並列プログラミングのためActorモデルが実装されており、将来的にこういったモデルの有用性は高くなると思われる。
・OCaml[*caml.inria.fr/ocaml*]：　関数型言語でありオブジェクト指向言語でもある。

６）Scala [scala-lang.org]

JVMベースの言語の１つ。様々な言語のいいとこ取りを狙ったような仕様が特徴だが、その中でも顕著なものを挙げるとすると、①Haskellに倣った型推論のある静的型付け言語、②開いたオブジェクト指向でREPLを持ちスクリプト言語に相当する記述性を持つこと、③高階関数を型システムの中で取扱うことのできる関数型言語、④正規表現およびオブジェクトに関する幅広いパターンマッチの機能の４点になるだろう。Scalaの型システムでは、CやJavaの型システムの不完全な点（返すべき値がないことをNULLで表現することの問題点）を、Option型（概ねHaskellのMaybeに相当する）で解決し、CやJavaで頻繁に見られた括弧入りの型名前置によるキャスト記法の必要性を少なくしている。このような特徴により、高い記述性を持つ言語となっている。Javaの後継言語という見方をする人も多い。前述のようにScalaでも他のJVM言語と同様に、javaのクラスライブラリをほぼシームレスに利用できる。従って、Swingを使ったGUI、Applet、JavaFXの利用、といった連携も可能となる（若干の工夫が必要なケースもあるが）。

Javaとの間の表記上の相違点は幾つかあるが、型指定子が変数名、仮引数名の後ろに置かれ、関数の返す値の型指定は関数の仮引数リストを囲んだ右括弧の後ろに置かれることが特に見かけ上の大きな違いとなっている。そういったいくつかの相違に対処できれば、Scala→Javaの移行はさほど大きいものにはならないだろう（もっとも、既存資産としてのコードを保守するような場面でなければ移行の必要もない可能性が高いが）。

Scalaにおけるパッケージマネージャとしては、Sbazが標準添付されている。Scala上での学習環境としては、kojo [*kogics.net/kojo*] および、前述のProsessingの機能を実現した環境として Spde [*spde.technically.us*] がある。また、Scalaをサポートしているwebフレームワークとしては、Lift [*liftweb.net*] と Play! [*playdocja.appspot.com*] の名前を挙げておく。JavaによるAndroid向け開発は例えばEclipse等のIDEをベースとして行うが、この

Eclipseに（他のIDEでも対応している）ScalaIDEプラグインを差し込めば、Scala言語で同じ事ができるとされている。

なお、JVMベースの言語としては、最近の書籍の発行数や情報源の量から判断してScalaを第一候補としているが、他にJavaに取って代わる可能性のある言語としては Groovy [*groovy.codehaus.org/Japanese+Home*] の名を挙げておく。こちらは動的型付のオープンなオブジェクト指向言語という位置づけのものである。その他のJVM言語については、[*matome.naver.jp/odai/2133273419683789401*] がよくまとまっている。

７）CoffeeScript [*coffeescript.org*]

JavaScript [*developer.mozilla.org/ja/JavaScript*] は、当初はブラウザの画面表示を豊かにするプログラミング言語としてブラウザに組み込まれたが、その後機能を増強し、各種JSライブラリが作られ、またサーバでも動作するプラットフォームが整備されたことにより、クライアント/サーバの両方で動作する言語として有力な存在となった。JavaScript自体は、プロトタイプベースのオブジェクト指向と、動的型付けを特徴とする軽量な（日本でいうLL言語ではなくコンピュータ負荷が軽量という意味で）言語であるが、記述力に不足な点がある。これを補うための各種ライブラリであったが、ライブラリとは別のアプローチでJavaScriptに対する糖衣（シンタックスシュガー）として作られた言語の１つがこのCoffeeScriptである。CoffeeScriptによるコードはJavaScriptにトランスレートされてそれぞれの（サーバまたはクライアントの）JavaScriptエンジンで実行される。正規表現、連想配列（これはJavaScriptからある）、配列内包といった機能を持つことでスクリプト言語に近い記述性を持つ言語が、双方の環境で動くことは、Webアプリケーションを１つの言語で作れる道が開けたということになる。ただしCoffeeScriptはまだ日本語での情報源がさほど多くないのが難点である。

なお、この双方で同じ言語を使うための、別のアプローチとして、Google社による Dart言語 [*dartlang.org*] がある。全貌はまだ未知数であるが（これまでのところ言語としての斬新な点はあまり見られていないが）、今後の動きに注目はしていたいところである。

８）日本語によるプログラミング

日本語でプログラムを作るという試みは古くからある。ここでは３つのプロジェクトを挙げておく。

a）ドリトル [*dolittle.eplang.jp*]

教育用言語および環境として開発されたもの。日本語に近い文法による表記とタートルグラフィックスがその特徴。

b）なでしこ [*nadesi.com*]

教育に限定せず実用的な日本語プログラミングを指向したもの。

c）PEN（言語名としてはxDNCL）[*www.media.osaka-cu.ac.jp/PEN*]

ここで使われるxDNCL言語（のもとになったDNCL言語）は、大学入試センター試験の「情報関係基礎」で用いられる言語である。いずれもキャンバスを含む簡易IDEにより編集～実行が１つの画面で行えるようになっている。本稿では前述の社会的要因を考慮して、ここで紹介するにとどめておく。

９）ビジュアル言語

文字を使わずにプログラミングを習得しようという試みも盛んに行われている。ここでもこれまでの調査対象としたものを挙げておくに留める。

a) Scratch [*etoys.jp/scratch/scratch.html*] は Smalltalkの実装の１つである Squid [*squeak.org*] をベースに作られた幼児向け環境。

b) VISCUIT（ビスケット）[*viscuit.com*]　これも子供向けの環境である。

10）ゲーム言語、アニメーション言語

前項と若干重なる部分があるが、言葉の分かる年齢層以上を対象にした場合は多少とも言語的学習活動を要求するようになる。そのぶん、楽しみの要素をどこに見出すか、についてそれぞれ工夫を行なっている。

Tonyu [*tonyu.jp*]　日本で作られている、RPG作成のための言語および環境。並列処理をサポートしているとこは先進的である。

Alice [*alice.org*]　３Ｄアニメーションを作りながらオブジェクト指向プログラミングを学ぶシステム。言語としてはJavaのサブセットに近いが、そのプログラミングを、ほぼタイピングなしにマウスクリックだけで進めていけるのが１つの特徴。また、「Do Together」という文法要素で並列処理もプログラムできる。

なお、これと同名の（ケース違いの）ALiCEという言語 [*kazpyon.iza-yoi.net/alice.html*] もあるが、これは研究中の、Basicに似たスクリプト言語であり、混同しないよう注意されたい。

３．３　総合評価

様々なアプローチがあることがわかった。初学者への入門教育として、これらすべてを試してみたいところであるが、現実的課題としては当面の言語を１つ選択することになる。これまでに述べた評価基準を考慮して、「Java言語にいずれかの時点でつないで行くことを考慮しつつ、実用言語で入門する」といった方針に沿って言語を選ぶとすればこれまで見てきた中では、RubyとScalaが現実的に有力な選択肢として浮上してくる。ちなみにこの２言語は、共通点としては、①def で関数宣言する、②関数呼び出しの括弧は省略可、③開いたオブジェクト指向、④ジェネレータによるfor-each的な繰り返し構文を持つ、が挙げられる。差異として最も大きいものは、Rubyが動的型付けであるのに対し、Scalaは静的型付け言語であることだろう。Rubyの世界にいる限りは型指定子の存在に出会わないですむ。或いは型指定子の概念を学ぶ機会がないとも考えられる。Scalaで始めるにしても最初は型指定子のことを知らずに（型推論任せで）コーディングは進めて行けるが、ある程度高度なことを考え始める段階になってScalaの型指定子に出会うことになる。「次の言語で初めて出会うまで型指定子を見せない」か、「最初の言語で徐々に型の話をしていく」か、これは教育者の方針次第だろう。本稿ではその最後の結論は名言しないで置くが、これまでRuby使いだった筆者もScalaの型推論システムを見て、その有効性に刮目する思いだったことは付記しておきたい。

３．４　考察

Java言語は何かと気難しいところがあり、Javaでの入門は、いわば大人（プロ）の作法を子供（アマ）に求めるような感覚になる。プログラミングの最初の段階は、なるべく「ゆるゆる」から始めたいものである。強い型付のある言語は、その「ゆるゆる」を許さないのが入門教育の上で問題になる。HaskellやScalaにあるような型推論はその入門時の敷居を下げるのに効果的だろう。このいずれの言語も、REPLにt:（正式には type:）コマンドがあり、式（が返すことになっている値）の型を表示してくれるので、それを見ながら型の概念を少しずつ学んで行けるのではないだろうか。

人が使う言語（ここでは自然言語）が思考（ものの考え方）に影響を与えるというのはよく言われることである。プログラミング言語もまた人の思考に影響を与えるだろうし、言語が変わるとそこで教えるべき内容も変わってくる。例を１つあげておく。変数aと変数bの内容を入れ替えたいときに、逐次型言語の場合（例えばCだと）int tmp=a; a=b;b=tmp; のように、仮置きのための一時的変数（ここではtmp）が必要なことを、大抵は鉄道の引き込み線の図を書いて説明したものである。されにその手順を簡単にすますための上記のコードを関数swap()の中に包み混んで、swap(a,b)のように使いたいと思ったときに、変数a,bがintならいいが他の型だと使えないね、と問題提起をするような流れの授業をかつてやっていた。今はこの議論そのものが意味を持たなくなっている。昨今は多重代入（或いは平行代入）が一般化したため、(a,b)=(c,d);のような代入文で交換が可能である。関数swapを作るにしても、動的型付の言語や、型パラメータのある言語だと前述の課題とは無縁である（したがってRubyでもScalaでも悩まなくていい）。こうして楽になった時間をどのように生かし、どんな内容でより深い或いはより広い学習につなげていけるか、教える立場の人間は言語の進歩に沿って次のステップの模索を続けなければならないことになる。

ハードウェア環境の変化もまた、プログラミングのあり方の変化或いは多様化を後押ししてきている。開発効率と実行効率の間での優先度において、開発>>実行、となるような領域が増えてい

る。いかに楽に書くか、すなわち人間にとってのLight Weightな（軽量な）言語がより強く求められるようになってきたのだろう。

従来の「変数」の概念は、CPUの「レジスタ」に対応する。レジスタが一杯だとメモリ上に配置（実際はレジスタから必要に応じて退避）する必要があり、アセンブラ時代はそれをプログラマがやり繰りしていたが、コンパイラ時代になるとコンパイラ任せですむようになった。一方、配列はメモリ上の連続した領域に対応する。といった具合に、データ処理の際にデータの実体を意識しなければならなかったのは資源の量的制約があったからだが､今はそこからほぼ開放されている筈。ファイルを処理する際に、少しずつメモリに読み込む旧来のスタイルではなく､ファイルの中身をごそっと読み込む、または読みこまなくてもプログラマがイメージしやすいデータ構造にマッピングさせて、そのデータ構造上の操作を行うようなプログラムを書くことは可能である。そんなプログラミングスタイルの変化が起きると、変数というものも、これまでのようにデータを保持する場所として認識せずに（授業では「箱」のイメージによる説明を今でも多用しているが)、むしろ関数から関数へ受け渡しされるオブジェクトを、その受け渡しの途中でトラップしてどこかに仮置きしておくような、例えば押しピン若しくはフックのようなイメージで捉えることが妥当になってきている。（実際に多くの言語での「変数」は現実には値でなく参照を保持している。）言語を考察することにより、プログラミングのあり方もまた考え直す、そういう時期に来ているような気がする。

４．おわりに

４．１　まとめ

初級プログラミング授業で使う言語の選定を目的としてプログラミング言語に関する調査を行った。結論としては型に関してのスタンスの違う２つの言語を最有力といて候補に残すことになった。

この課程で以下のようなことを論じてきた。

a) プログラミング言語の大きな動き
・高水準言語へのシフト
・今後は関数型パラダイムの重要性が増す
b) 現実との折り合い
・関数型パラダイムは徐々に取り込むべし一気にパラダイムを変えてしまって、社会で使われるパラダイムから離れた思考をする、言わばミュータントを送り出したとしてもそれを受け入れる現場がない
c) 入門のための必要条件
・開いたオブジェクト指向
・REPLによるインタラクティブ実行
・グラフィック機能とインタラクションの平易な活用手段
d) 型制約の概念
・型推論により静的型付言語も入門教育に導入可能になってきた
・型制約との出会いについての考え方によって、早い時期から→Scala　後からでいい→Ruby

今般、あらためて調査をしてみて感じたことは、多様な選択肢が我々の目の前にあるという事である。（大学教育だけが影響力を持つとは考えないが)、大学教育＝＞実社会（での技術の現場)、という流れの中で実社会を意識して教育内容を整えるという後ろ向きの影響と大学教育の結果が実社会で働く若者の思考として反映されるという前向きの影響とが、いわばニワトリと卵の関係になっている。そんな中で、単純に多勢に合わせるだけの発想でなく、どんな発想法を身につけた卒業生を社会に送り出すかを模索していくことが大学人の責務だと、あらためて感じた次第である。

４．２　今後の課題

この入門教育に関しては、この言語選定のあとの課題としては、

１　興味を持続させる教材の整備
２　コンパクトな見せ方のできる（言語仕様のサブセットを見せるような）環境の模索
３　DSLの機能を活用し後置表現による言語（処理の流れを左～右に統一）の可能性の模索

を考えていきたい。また、最終物に選定した言語での教育実践を重ねつつ、それと並行して、他の場所（ゼミナールや他学への出講）において、他の言語での実践からのフィードバックも、今後の研究に反映させていきたいと考えている