解説

Ruby の真実

The Truth about Ruby The Language

松本行弘 （株）ネットワーク応用通信研究所
matz@netlab.jp

Ruby の表と裏

筆者は 1993 年以来，オブジェクト指向スクリプト言語 Ruby を開発している．今年で開発開始以来 10 年になるということで 1 つの節目である．Ruby は純粋なオブジェクト指向である点や，豊富なクラスライブラリにより，従来 Perl などで行われていたテキスト処理やシステム管理を中心とするスクリプトプログラミングを，Perl 以上の読みやすさと同等以上の簡潔さで実現している点が高く評価されているようだ．また，言葉にはしにくいものの「プログラマの気持ちにしっくりくる」という評価 [1] も受けている．また，国内では国産初のメジャープログラミング言語としての期待も受けているようだ．

Ruby を一般公開したのは 1995 年であるが，それ以来，国内外でいろいろな反響を呼び，数多くの書籍が出版されている．2003 年初頭の時点で日本語の書籍が 23 冊，英語の書籍が 6 冊，ドイツ語が 2 冊，フランス語が 1 冊，またロシア語が翻訳中である．

自分の設計した言語が広く使われ，高く評価されるのを見るのは嬉しいものである．しかし，一方親の欲目を抑えて冷静に見てみると，Ruby には欠点もたくさんあり，多くの点でライバルともいえる Perl や Python にも負けている．以下のような点で Ruby は劣っていると考えることができる．

○単純さ

一見シンプルに見える Ruby の文法であるが，その yacc による文法記述を見てみれば，明らかにシンプルとはいえない．その文法の全容の複雑さは Python をしのぎ，Perl に匹敵する．

○美しさ

美しさの基準はさまざまであるが，シンプルさから現れる美しさという点では，Ruby は Lisp にかなうものではない．また，Ruby 以上に美しさを感じさせる言語は数多い．

○簡易さ

習得の容易さという点では，Ruby の文法はとっつきは良いものの，全体を習得するという条件をつけるとより小さい仕様の言語の方が習得が容易であるといえよう．

○性能

Ruby の実装については後述するが，Ruby 処理系の性能は決して優れていない．多くの点で他の言語よりも劣っている．

このような欠点にもかかわらず，Ruby が受け入れられている現状を考えると，プログラミング言語が受け入れられるかどうかは，一般的に良い言語の性質と考えられている上記の点以外に強く影響されると考えられる．

最初のプログラミング言語の登場以来，数千，数万もの言語が設計されてきた．しかし，開発者とその周囲の限られた範囲を越えて使われるようになったものは少数であり，さらに 5 年 10 年の期間を生き延びるものはまれである．ここでは，将来の言語設計者が Ruby に続く，より優れたより広く受け入れられる言語を誕生させることができるように，多くのユーザにアピールした Ruby の真の性質の考察を試みる．

生き残る言語

言語が「死ぬ」のは以下の２つの条件のうちいずれかが成立したときである．

- その言語のユーザがいなくなる．
- その言語の進化が止まる．

これらは相互に関連していて，ユーザがいない言語の進化は止まるし，進化の止まった言語からはユーザが離れていく傾向がある．プログラミング言語は一般的に他のアプリケーションよりも寿命が長い．1950 年代に開発された FORTRAN, COBOL, Lisp はそれぞれいまだに現役である．言語を維持するためには相当長いスパンでの継続的な開発とメンテナンスが不可欠である．また，継続的な開発が期待できない言語はユーザからの信頼を獲得することが難しい．有望でありながら，開発者の都合など諸般の事情で進化が止まってしまい，消え去っていった言語も数多くある．

誰からも使われないプログラミング言語は言語設計者の自己満足である．そうならないためにはプログラミング言語は生き残らなければならない．つまり良い言語の最初の特質は生き残ることである．ここからはいろいろな側面からプログラミング言語が生き残るための原則について考察する．

生き残りの原則

プログラミング言語が生き残るかどうかを決めるものはなんだろうか．一見「その言語で何ができるか」ではないかとも思えるが，実はこれはそれほど重要ではない．チューリング完全な言語では原理的には任意のアルゴリズムが記述可能であるし，そうでなくても最近の言語は他言語インタフェースを持っていて，たとえばＣで書かれたライブラリを利用可能なものがほとんどである．極端な話，理論的にはどの言語を使ってもなんでもできる，というのが実情であろう．

となると，むしろ重要なのは「その言語がどれだけ嬉しいか」である．この「嬉しい」は「その言語を知ってると仕事がもらえて嬉しい」とか「その言語を知っているとかっこいい（と見られる）」とか，「その言語でプログラミングしていると気持ちいい」とかである．上記の例のうち，前二者はやや政治的なニュアンスが，後者は心理的なニュアンスが大きい．言語の生き残りには技術的要素は決定的ではないことが分かる．政治的なニュアンスが強い言語としては，米国防総省のお墨付きがあった Ada や，Java, それから膨大な資産を背景とする FORTRAN や COBOL などがある．政治的な側面は本論の範囲を越えるので，これ以上は立ち入らないこととする．

政治的でないプログラミング言語の嬉しさは，普段よく片づける問題がどれだけ簡単に済ませられるかということであろう．ある言語において，コンピュータにやらせたいことを手早く表現できるかどうかということが重要である．

テキスト処理に向いたスクリプト言語に人気があることは，プログラマが普段片づけなければならない仕事の多くがテキスト処理に還元できることを意味している．確かにテキスト処理だけでなく，CGI プログラミングなども結局は文字列処理であるといえないことはない．また，ある言語に自分が最もよく使う分野に対応した（できのよい）ライブラリがそろっていれば，その言語を使うことによる嬉しさは増大するだろう．

結局ある言語が人気を得るかどうかは「なにができるか」より「いかにできるか」が重要であるということが言えるだろう．ある言語が提供する一種のプログラミングスタイルがユーザに対してどれほどアピールするかと言い換えてもよい．

プログラミング言語の役割

では，プログラミング言語の魅力について，基本に立ち返って「プログラミング言語とはなにか」という点から考えてみよう．プログラミング言語の目的は当然「処理の手順をコンピュータに分かるかたちで表現すること」である．この視点から得られる良いプログラミング言語は，コンピュータの性能を十分に引き出すことだったり，曖昧性がなく，より正確に表現できることだったりする．

しかし，見方を変えるとプログラミング言語の目的は「人間の意図を表現すること」と考えることもできる．プログラミング言語によって記述されたものは，もちろんコンピュータによって実行可能ではあるが，他の人間（あるいは３カ月後の自分）によって読まれることもある．この見方では主体はコンピュータではなく，人間である．この視点からは，良いプログラミング言語かどうかはプログラム中の人間の精神にどのような影響を与えるかによって決定される．また，プログラミング言語はマン・マシン・インタフェースの一種としてとらえられ，ユーザインタフェースの原則の多くが適用される．

重要な原則は

- 一貫性
- 簡潔性
- 柔軟性

である．

一貫性とは，同じようなことをするために同じような手順で行えることであり，別の言い方をすれば「常識が通用する」ということである．一貫性のあるシステムでは，常識を身につけたユーザは新しいタスクに対して「きっとこうすればできるに違いない」という推測を行い，その推測は裏切られない．推測が裏切られるシステムではユーザは不快感を覚え，これ以上使いたくないという気持ちが強くなる．

簡潔性とは，ユーザがコンピュータに対して指令を行う場合，できるだけ簡潔に指示できることである．プログラミング言語の場合では，可読性に影響を及ぼさない範囲内でできるだけプログラムが短くなるように，文法・機能の両面で支援するということである．

柔軟性とは，ユーザのニーズにシステムの挙動が適合可能であることである．プログラミング言語の場合，組み込みの機能とユーザ定義の機能の間の差別が少ないことである．

このような原則を考慮して設計された言語はプログラマにやさしい．プログラマにやさしい言語は生き残る可能性が高くなる．

スクリプト言語の世界

Tcl の開発者である John Ousterhout の 1998 の論文[2] では，21 世紀のプログラミング言語は C や C++ のようなシステムプログラミング言語とスクリプト言語に二分され，しかもスクリプト言語の比重が高くなることが予想されている．彼によればスクリプト言語は，システムプログラミング言語によって記述されたコンポーネントを組み合わせるために用いられる．

仮にコンポーネントを組み合わせて，目的に適合したシステムを作り出すことをスクリプト言語の定義とすると，面白い命題が導出できる．我々がプログラミングする際に，すべてを自分で開発することはほとんどない．ハードウェア制御は OS とそのドライバに，また多くの処理はライブラリというかたちで提供されるコンポーネントを組み合わせて行う．よって，WWW のスクリプト言語は Perl や PHP（あるいは Ruby など）であり，Emacs のスクリプト言語は Emacs Lisp であり，Perl, Ruby, sh などとともに C もまた UNIX のスクリプト言語であり，スーパーコンピュータによる数値解析のスクリプト言語は FORTRAN である．C や FORTRAN をスクリプト言語と考える人はほとんどいないだろうが，コンポーネントを組み合わせるということに注目するとまさにスクリプト言語である．ただ，スクリプト言語としてはあまりにも劣っているために誰もそう認識しなかっただけである．

2003 年時点では Ousterhout の予想は半分だけ当たっていると考える．つまり，スクリプト言語の比重がますます高くなるという部分である．しかし，すべての言語が程度の多少はあるもののスクリプト言語として振る舞うこと，また多くのコンポーネントが他のコンポーネントを組み合わせて作られるため，コンポーネント開発までがスクリプト言語の対象になる点は彼の予想を超えていたのではないだろうか．

また彼は（おそらくは Tcl の設計者という経歴もあって）スクリプト言語とシステムプログラミング言語との間に決定的な差が存在し続けると考えていたようだが，実際には Perl, Python, Ruby を始めとする最近のスクリプト言語はより普通の言語への道を歩みはじめ，性能上の要求からシステムプログラミング言語を必要とする部分を除いてはシステム全体をスクリプト言語で記述するケースが増大している．

このような観測から，プログラミング言語においてコンポーネントを組み合わせるスクリプト言語的アプローチはますます重要になり，将来登場するメジャーなプログラミング言語のほとんどはおそらくなんらかのスクリプト言語であろう．新たな言語を設計しようとするものはこの点にも注意すべきである．

オブジェクト指向スクリプティング

現在，広く使われているスクリプト言語といえば，Perl，Python そして Ruby くらいであろうか．Ruby はまだ少々マイナー過ぎるかもしれない．

Tcl もメジャーだが，直接扱えるデータタイプが文字列しかなかったり，変数値の取得に特別な記法を要求したりと「普通の言語」からまだ遠いので，他のスクリプト言語と比べると少々古めかしい印象がある．

圧倒的にブラウザ組み込みの用途が多く，他のスクリプト言語とは使われ方のパターンが異なるが，JavaScript も無視してはいけないだろう．

他にもう少しマイナーではあるが，Lua，Pike などがある．Lua は組み込みを意識したスクリプト言語で，やはりオブジェクト指向言語であり，Ruby 同様 Algol 譲りの制御構造が end で終わる文法を持つ．一方，Pike は MUD プログラミング言語を起源とする言語である．これもオブジェクト指向言語で，C に似た文法を持つ．現在は Roxen Challenger という Web Server の拡張言語として知られている．

余談だが，スクリプト言語は英語圏以外で生まれたものが意外に多い．Perl と Tcl はアメリカ生まれだが，日本生まれの Ruby, オランダ生まれの Python, ブラジル生まれの Lua, スウェーデン生まれの Pike などである．現在のソフトウェアのアメリカ中心状態を考えると

少々特異である．

これらの現代のスクリプト言語は例外なくオブジェクト指向機能を備えている．オブジェクト指向機能にはPerlやPythonを始めとして手続き型言語にオブジェクト指向機能を追加したものが多く，すべてのデータがインスタンスであり，すべての手続きがメソッドである純オブジェクト指向言語Rubyはむしろ少数派である．

現代のスクリプト言語が，ハイブリッドか純粋オブジェクト指向かの違いはとりあえずおくとしても，例外なくなんらかのオブジェクト指向言語である点は興味深い．この理由はおそらく，オブジェクト指向プログラミングがコンポーネントを組み合わせるために非常に有効であることだろう．コンポーネントプログラミングを積極的に支援する言語は良いスクリプト言語である．

まったく別の観点から考えても，新しい言語が生き延びるためにオブジェクト指向は有効である．つまり，オブジェクト指向という考え方は元々人間の認知を支援するモデルとしての成り立ちを持つため，人間にやさしい．すでに述べたように人間にやさしい言語は生き延びやすい．この点で他のパラダイム，たとえば関数型プログラミングを採用した言語よりも有利である．筆者は関数型プログラミングの重要性を否定するものではないが，アカデミックな価値と言語の人気や生存率とは独立である．

見かけの重要性

良いプログラミング言語かどうかはプログラム中の人間の精神にどのような影響を与えるかによって決定されるとすれば，言語設計者のとるべき態度も変化する必要がある．この基準に従えば，通常，言語設計者が最も気にする点である性能については，言語仕様として考慮する必要がないことになる．性能というのは実装にかかわる性質であり，言語が良いかどうかには関係がない．しかも，コンピュータの性能そのものが向上しているため，言語仕様による性能の実現にはさほど意味がなく，むしろ性能を向上させるための特質を言語に付与することにより人間に対するやさしさに悪影響を及ぼす可能性がある．

また，言語仕様の小ささもそれほど重要な要素ではなくなる．言語仕様が小さければ，処理系も小さく，かつ効率良くなる可能性もあり，望ましい性質のように思える．しかし，実際には処理系の性質と言語仕様の性質は分けて考える必要があり，処理系の小ささは良い言語とはまったく関係がない．むしろ，小さな仕様の言語で記述されたプログラムは大きくなったり，可読性が低くなったりする可能性がある．処理系の簡潔さよりも記述されるプログラムの簡潔さを重要視すべきである．これ以上ないくらい小さな言語文法を持つLispが不人気である点に留意すべきである．

先に言語の従うべき重要な原則の１つとして柔軟性をあげたが，筆者の個人的な考えでは，プログラミング言語の文法に限って言えば柔軟性はあるべきではない．つまり，ライブラリは柔軟に拡張可能であるべきだが，一方で文法は拡張可能であってはならないと考える．人間は，プログラミング言語の文法を基本的なルールとしてあてはめることにより，その言語で書かれたプログラムを読解する．Lispのマクロのような文法そのものを拡張する機能は，この基本的ルールを変更して，極端な話，プログラムごとに専用の言語を作り出してしまう．これでは文法というプログラム読解のための重要なツールの働きを損ねてしまう．

最も重要な点は複雑さの制御である．プログラムは本質的に複雑である．人間が解決すべきものが複雑だからである．また，人間が使いやすい道具のあるべき姿も複雑である．人間の認知や発想がいびつで複雑だからである．そこで複雑な人間に対応するためにツールであるプログラミング言語は複雑になってしまう．しかし，同時に人間はあまり複雑なものを取り扱うことができない．そこでなんらかのかたちで複雑さを後ろに押し込めて目が届かないようにすることで，見かけの単純さを演出することが大切になる．

１つの方法は複雑さの増加を制御することである．言語が発展するに従って複雑さが増大することは避けられない．問題は言語のある変化による複雑さの増大の質である．複雑さが増大するとき，その増え方には線形増加する性質のものと指数関数的に増加する性質のものがある．たとえば，前述の文法拡張は指数関数的に複雑さが増大する例であり，クラスにメソッドが１つ２つ増えるようなものは複雑さが線形に増大する例である．人間は複雑さの線形増加には比較的容易に対応できるが，指数関数的増加は扱いにくい．複雑さを増大させるときに，この増大はどちらの性質を持つか検討することは重要であり，指数関数的増加を招く変更には特に慎重になるべきである．

もう１つの方法は，複雑な部分をめったに登場しない部分に押し込めてしまう点である．実用的なプログラミング言語はさまざまな応用領域をカバーするために多種多様な機能を持ち，必然的に複雑にならざるを得ない．しかし，すべての機能が均等に重要というわけではなく，ある機能は広い範囲で頻繁に使われるのに対して，別の機能はさほど頻繁には使われないということはよくある．そこで，一般的によく使われそうな機能ほど簡潔かつ単純な外見で提供することで，ユーザが目にする複雑さを低減することができる．これはハフマンエンコー

ディングと類似の発想で複雑さを制御していると考えることもできる．

巨人の肩に乗る

コンポーネントを組み合わせるプログラミングスタイルは既存の資産を最大限に活用するという点で，まさに「巨人の肩に乗る」ことを実践しているわけだが，プログラミング言語の設計においても巨人の肩に乗ることは必要である．ほとんどのプログラマ（言語ユーザ）は複数のプログラミング言語を知っており，実際に複数を切り替えて使っている．ユーザが新しい言語を習得するためには，当然学習が必要であり，それはコストがかかる．別の言い方をすれば心理的な抵抗がある．この心理的な抵抗をその言語を学ぶうれしさが上まわるとき，その言語は新しいユーザを獲得する．このときプログラマの過去の経験や常識が通用すれば心理的抵抗を軽減することができる．心理的抵抗はユーザ獲得の最大の障害なので，この抵抗を削減することは生き残りに重要である．これは言語仕様レベルでの一貫性の実現と考えることもできる．しかし，あまりにも既存の言語に似すぎていると，こんどは学習する動機を失わせる結果になるので，注意が必要である．

Prolog のようにパラダイムから異なる言語は学習コストおよび心理的抵抗が高い．Prolog が各方面で高い評価を受けた優れたパラダイムを提供し，熱狂的ユーザを持ちながら広く使われない（そして最近あまり話題にならなくなった）理由はここにあると考えられる．

Java は類似性によって成功した言語である．Java の言語仕様に独自性は少ないが，C++ に十分似ていること，それでいて C++ の複雑すぎる部分（多重継承，リファレンス型，テンプレートなど）を排して，逆に Lisp などで有効性が示されていた機能（ガベージコレクションや例外）を取り込んだことが成功要因（の 1 つ）であろう．

入手可能性

過去，新しい言語をひっさげて登場したベンチャー企業は数多くあるが，ほとんど成功したものはない．プログラミング言語はどんなにその言語が優れていてもビジネスにはならないように思われる．例外はマイクロソフトの VB のように圧倒的な市場占有率を背景にした場合くらいではないだろうか．

一般的に，新しいプログラミング言語を学んでみようと思った時点ではその言語がものになるかどうかは分からない．新しい言語を学ぶことは知識を広げることにはなるが，多くの場合は知的好奇心を満足させるだけに終わる．悲しい現実だ．単なる知的好奇心を満足させるためにどれだけ投資できるか，ということを考えると新言語の商用処理系にはほとんど望みがないように思われる．

生き残るためには広く使われなければならず，広く使われるためには間口を広げなければならない．そのためには無料の処理系が必要だ．処理系がオープンソースである（ソースコードを公開する）必要はないかもしれないが，過去に生き残った言語とその処理系がオープンソースであったかどうかにはかなり強い相関があるようだ．オープンソースであれば，たとえば OS に最初から添付されたり，CD-ROM などの媒体で配布されやすいこと，コミュニティの力で多くのプラットフォームに移植され，より広く入手できることなどが影響するのかもしれない．

コミュニティ

プログラミング言語において，オープンソースソフトウェアにおいて，そしておそらくはほとんどのソフトウェアプロダクトについても，コミュニティの存在は重要である．コミュニティはそのソフトウェアの発展を支援し，ユーザ同士での助け合いを実現し，そのソフトウェアの生き残りに貢献する．

Ruby についていえば，Ruby を中心に日本語と英語のコミュニティが存在していて，それぞれ周辺プロダクト（クラスライブラリなど）の開発，ドキュメントの整備，Ruby 処理系の保守，WWW サイトの運用など多くの役割を担っている．彼らが無給でこれだけの働きをしていることは驚嘆に値する．コミュニティのやりとりは主にメーリングリストを通じて行われる．メーリングリストのやりとりはしばしばフレームと呼ばれる不毛な論争を起こしやすいことが知られているが，Ruby に関するメーリングリストでは一般に参加者は驚くほど紳士的・友好的で，このコミュニティの態度のゆえに Ruby を選んだと公言する人もいる．

運の善し悪し

生き残りに関する最後の要素である「運」について考察してみる．科学的な考察は不可能ではあるが．

Ruby の発展にはさまざまな「運」が大きく作用した．最も重要だと思われるのは，人との出会いである．Ruby は幸運にも重要な節目での出会いに恵まれた．1994 年の公開直後にメーリングリストの開設を申し出てくれた人，またメーリングリスト上でアイディアを提供してくれたり，開発・デバッグに協力してくれた人，1997 年の Perl Conference でのスピーチに

呼んでくださった人，1999年の最初のRubyの本を出版することに協力してくれた人などなどである．その中でも重要な出会いは1999年のRubyとPragmatic Programmersとの出会いである．彼らはソフトウェア開発の有益な指針を多く載せた「Pragmatic Programmers」という著書のある二人組のソフトウェアコンサルタントで，Rubyに関する英語での最初の書籍「Programming Ruby」を著した．この書籍と彼らの活動によって，海外でのRubyの普及が確実なものとなった．

Rubyの実装とその課題

先にプログラミング言語において，実装の質は最重要ではないと述べたが，現実には実装がない言語は幻に過ぎない．そこで現在のRubyの実装について述べておく．

現在，Ruby言語を処理できるインタプリタの実装は3種類ある．1つは筆者の開発しているC言語で記述されたもので，1993年に筆者が開発を開始したものの直系の子孫であり，一般にRubyインタプリタと呼ぶときにはこれを指す．現在のバージョンは1.8.0である．もう1つはJavaで記述されたJRubyと呼ばれる処理系である．これはRuby 1.6互換を目指しており，またJavaのクラスを参照するなど独自の機能も備えている．最後のものはNETRubyと呼ばれるもので，C#で記述されている．NETRubyはクラスライブラリ部分をマイクロソフト.NETのクラスライブラリに依存しており，文法的にも機能的にもRuby言語のかなり小さなサブセットである．以降では筆者の開発したRuby処理系についてのみ述べる．

Rubyのソースプログラムはyaccによって記述された構文解析器によって構文木に変換される．実際のプログラムの実行は，この構文木を入力とした構文木インタプリタが行う．このようなソースコードを最初に内部表現にコンパイルし，その内部表現をインタプリタで実行するコンパイラ・インタプリタ型の処理系は，昨今のインタプリタ型言語では一般的な実装である．

処理系はオブジェクト管理用にガーベージコレクタを持つ．Rubyが採用しているガーベージコレクタは古典的なマーク・スイープ法である．PerlやPythonなど他のスクリプト言語が採用しているリファレンスカウント法に比べるとオブジェクトの相互の参照が複雑になることの多いオブジェクト指向プログラミングに向いていると考えられる．このガーベージコレクタはC言語のスタック領域をスキャンする「保守的」なガーベージコレクタであるため，Cなどの言語で記述されインタプリタに機能を追加する拡張ライブラリにおいて参照管理の必要がなく，拡張ライブラリの開発効率を向上させている．

現在のRubyの処理系は非常に素朴な実装になっている．実際，性能向上のための工夫といえば，メソッド検索の効率化のためのメソッドキャッシュがインタプリタに組み込まれている程度である．このメソッドキャッシュは簡単なものではあるが，典型的なケースでは90%以上のヒット率を達成し，性能向上に貢献している．

このようなナイーブな実装のままであった理由としては，開発を開始した時点での想定では，スクリプト言語であるRubyの対象分野はテキスト処理などが主体となり，性能への要求はそれほど高くないだろうと考えていたからである．しかし，Rubyの適用分野が広がるにつれ，プログラム，扱うデータの両方の規模において想像もしなかった領域に挑戦するユーザが現れはじめた．ここにおいて今までの実装では性能への要求に対応できなくなりつつある．

もう1つの課題としては，Rubyインタプリタの他のプログラムへの組み込みである．現在のRubyでも組み込みは不可能ではないが，やはり得意なのはRubyが中心になり，他のライブラリやプログラムを制御するスタイルである．しかし，たとえばゲームやアプリケーションのスクリプトエンジンや，Apacheを始めとするサーバの一機能としてRubyインタプリタを組み込みたいというニーズも増大しつつある．

このようなニーズに対して，現在の処理系では以下のような課題が残されている．

○インタプリタ初期化

現在のRuby処理系は一度動作したインタプリタを再初期化する手段を提供していない．

○インタプリタ構造化

現在のRuby処理系はプログラム中に複数のインタプリタのインスタンスを用意する方法がない．

○ネイティブスレッド対応

現在のRuby処理系はネイティブスレッドを利用したプログラム中に組み込むと，保守的ガーベージコレクタが異常終了を引き起こす（スレッドセーフでない）．

性能と組み込みは，将来のRuby処理系の実装に対する大きな課題である．

将来の実装

このような課題に対して現在の実装では解決しきれない現状がある．現在の実装は，この 10 年の間，段階的に発生したプログラミング言語の進化に比較的アドホックに対応するかたちで発展してきた．その結果，構造が複雑化し相互依存や影響が大きすぎるので，安易に大規模な改善にとりかかることができなくなりつつある．そこで近い将来，インタプリタの大規模な再実装を行う予定である．Rite というコードネームを持つこの再実装の主たる目的は性能の改善と C からの API の整理による組み込み対応の改良である．Rite の開発はまだ始まっていないので，ここで述べる内容はあくまでも目標と予定である．

性能については，インタプリタのバイトコード化と新たなガーベージコレクションの導入によって実現するつもりである．

昨今のインタプリタ型言語がコンパイラ・インタプリタ型処理系をとることはすでに述べた．最近ではコンパイル結果の内部表現として，Ruby の用いているような構文木ではなく，命令をバイト列として並べたバイトコードと呼ばれる形式が流行である．バイトコードが流行している理由は，Smalltalk や Java での実績が大きいと思われる．インタプリタの性能を考えるとき，バイトコードにはさらに，命令フェッチのコストが低い，効率の良い実装に関するノウハウが蓄積されてきている，ピープホール最適化を始めとする最適化を行いやすい，などの利点がある．これらの利点を踏まえて，Rite ではインタプリタ部分はバイトコードによって実現されるだろう（厳密には後述の理由により「ワードコード」となる）．

Ertl らの研究[3] によれば，ソフトウェアによって実装されるバイトコードインタプリタ（仮想マシン（VM）とも呼ぶ）においては，命令フェッチのコストが高いため，1 つの命令がたくさんの処理を行う方が効率が良い．そこで Rite では命令の単位をバイトから CPU が最も効率良く扱える単位であるワードに拡張する．このことによって命令の扱いの効率を高めるだけでなく，命令空間を拡大することによって，多様な命令を備えたり複数の命令を 1 つの命令にまとめたりする最適化の余地を広げることができる．

ガーベージコレクタについては，大量のデータを扱うプログラムに対応するため，世代別ガーベージコレクタを導入する．それと同時に，ネイティブスレッドと共存できる保守的ガーベージコレクションを実現させる．この共存の詳細については実装完了後，別の機会で報告したい．

C 言語から Ruby を呼び出す API に関しては，インタプリタを C 言語のオブジェクトとして独立させ，現状のコンテキストの保存を大域変数に頼った（そして結果としてプロセス中に唯一のインタプリタしか持てない）構成を払拭したい．また，組み込みも最初から視野に入れた設計を実現したい．

Rite の開発を宣言してから 2 年になるが，いまだに実装にとりかかれていない．これは既存のインタプリタの整備に時間がとられているせいだが，今年こそは Rite を実現させる年にしたいと考えている．

まとめ ー非本質が本質

Ruby の設計原則についても，あるいは思想的背景であるオブジェクト指向プログラミングも，「できないことをできるようにする」という性質のものではない．むしろプログラマの気持ちにどう影響するかという心理的側面のほうが大きい．よってそれらは本質ではあり得ないという考え方もあるだろう．しかし，プログラミング言語に限って言えば，「言語は人間が使うもの」である以上，プログラミング言語の「良さ」は「見かけ」とか「使いやすさ」とか「気持ちよさ」といったような非本質に宿ると断言できるのではないだろうか．

参考文献

1) Ruby Testimonial:http://www.ruby-lang.org/en/20020110.html
2) Ousterhout, J. K. : Scripting: Higher Level Programming for the 21st Century, IEEE Computer Magazine, pp. 23-30 (Mar. 1998).
3) Ertl, M. A. and Gregg, D. : The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures, Lecture Notes in Computer Science, 2150 (2001), citeseer.nj.nec.com/ertl01behavior.html

(平成 15 年 4 月 1 日受付)