Gaucheの開発戦略
―小規模プロジェクトこそ国際化を考えよう―

川合史朗 (Scheme Arts, L.L.C.)

概要 小規模で開発リソースも少ないオープンソースプロジェクトにとって，日本国外への展開は「もっとユーザや開発リソースが増えてから」という，将来のステップのように感じられるかもしれない．しかし，ニッチなターゲットを相手とするソフトウェアならば，対象を国内に限ってしまうことはただでさえ少ないユーザをさらに限定することになる．むしろ最初から国際的に展開しておいた方がプロジェクトの持続に必要なユーザを集めやすい．本稿ではオープンソースのScheme処理系Gaucheの10年間にわたる開発経験から，少ないリソースで国内外にユーザを得る，維持可能な戦略について論じる．

1.はじめに

『Gauche』*は，筆者が10年ほど前からオープンソースで開発を続けているLisp系言語Schemeの処理系である．Schemeの言語仕様R^{5}RS [6]に準拠しているが，それだけでは最低限の言語機能しかないため，実用性を求めて大幅な機能拡張を行っている．デフォルトではSchemeソースコードを読み込み直ちにコンパイルしてVM上で実行するスクリプトエンジンとして動作し，変化の激しい開発現場で手軽にスクリプトを書ける軽量言語処理系としての使い勝手の良さを持ちながら，規模の大きなソフトウェアにも対応可能な処理系を目指しており，筆者自身の開発業務にも使っている．ユーザの絶対数は少ないものの，日本国内／国外問わずに使って頂いているようである．

今回，「世界に飛び出す日本のソフトウェア」特集ということで，国外にもユーザのいるGaucheについて書かせて頂くことになったのだが，実を言うとGaucheの開発にあたって特に「日本から」「世界へ」という意識を持ったことはない．ドキュメントやリリースアナウンスなどは英語版がやや先行しており，開発過程に関してはむしろ世界から日本へ，と言えるかもしれない．

とは言え，これは「世界を席巻する！」というような野望を持ってそうしているわけではない．英語を先行させた日英2ヵ国語でのサポートというのは，少ない開発リソースでニッチな領域を相手にする上で，最も効果的だと判断した戦略なのである．

もともとSchemeは，産業界では決してメジャーとは言えないプログラミング言語である．そこでさらに対象を日本に限定するのは，ユーザ層を狭めるだけであろう．オープンソースの現実的な利点のひとつは，より多くの人の目に晒されることでより多くのバグやアイディアを拾えるということだ．その利点を活かすためにはある程度の大きさのユーザ層が必要になる．ソフトウェアが現場での使用に耐えうると信頼されるまで成熟するには，作者の想定を超越するような様々な環境で使われなければならないが，作者自身および固定した少数のユーザでは偏った環境でしか使われない．多くのオープンソースプロジェクトで開発がアルファ版状態で止まってしまう理由のひとつが，ユーザ層の薄さから来るフィードバックの欠如であると思われる．

個人がオープンソースでささやかに始めるようなソフトウェアプロジェクトの多くは，自分や周辺で使えれば良いという動機で始められるものだろう．ユーザ数を積極的に増やす必要を感じない，あるいは世界へ飛び出そうなんて大げさな，という認識が多いのではないか．本稿では，むしろそのような，開発リソースも少なく，ターゲットも広くはないソフトウェア開発プロジェクトこそ，持続に必要なユーザを得る戦略として，最初から世界を視野に入れることが有効ではないか，と主張したい．そしてGaucheをひとつの事例として紹介する．

なお，国際化のあり方として，日本語と英語で十分かどうかという点には検討の余地があろう．ソフトウェアの対象分野によっては別の言語の需要も大きいかもしれない．それでも日本人開発者が中心の場合は，おそらく英語が最も利便性の高い外国語であり，開発協力者も見つけやすい．そしてまた，英語での利用が可能になっていれば，それを軸として他国の協力者を得ることもできる．筆者は日本語と英語だけで十分だと主張するもので

*http://practical-scheme.net/gauche/

は決してないが，現実的な選択肢としては有用だと考えている．

本稿の構成であるが，次章で簡単に Gauche の開発経緯を紹介し，第 3 章で Gauche が開発の国際化に際して配慮している点を論じる．そして第 4 章にまとめを記す．

2.Gauche の開発経緯

Gauche の開発を始めたのは 2000 年 12 月頃からである．当時筆者は日本のゲーム会社の米国子会社に勤務しており，Common Lisp 製の基幹システムと，C++, Perl, Java, Scheme 等様々な言語で書かれるプロダクションツールを連携させる立場にあった [3,4,5]．Scheme の言語としての素性の良さには注目していたが，システムプログラミングをする上での Perl 並みの多用途性，インタラクティブ CG にも使える性能，日々の作業に耐え得る頑健性を備えた手頃な Scheme 処理系が見当たらなかったというのが，Gauche 誕生のきっかけである．

業務ではなく個人のオープンソースプロジェクトにした理由は，社内だけでの限られたユーザによる使用では，開発に投入するコストに対して十分な利益を得られないと考えたからである．そこで開発は自宅で勤務時間外に行い，オープンソースとして成果を公開し，会社では 1 ユーザとしてリリース版をダウンロードして使っていた．

筆者の勤務していた職場では日本語と英語が並行して使われており，両言語を問題なく処理できることは最低条件であった．今でこそ Unicode によって多言語をサポートする Scheme 処理系は多いが，当時きちんとした多言語環境を提供していた処理系はほとんど無く，それもまた自作の動機のひとつであった．当初から日本語圏と英語圏のユーザを対象にしていたため，英語と日本語の 2 ヵ国語対応というのは自然な要請であった．

2002 年から筆者はソフトウェアコンサルタントとして独立し，ソフトウェア開発を請け負う中で Gauche を積極的に利用してきた．2005 年以降は，機能の拡充と，安定性の向上や実用上障害となる実装上の限界の除去とにほぼ同等の比重を置いている．本稿執筆段階でバージョンは 0.9.1 であり，初の安定版 1.0 へ向けた開発の最終段階に入っている．

図 1 に，いくつかの代表的なバージョンにおけるソー

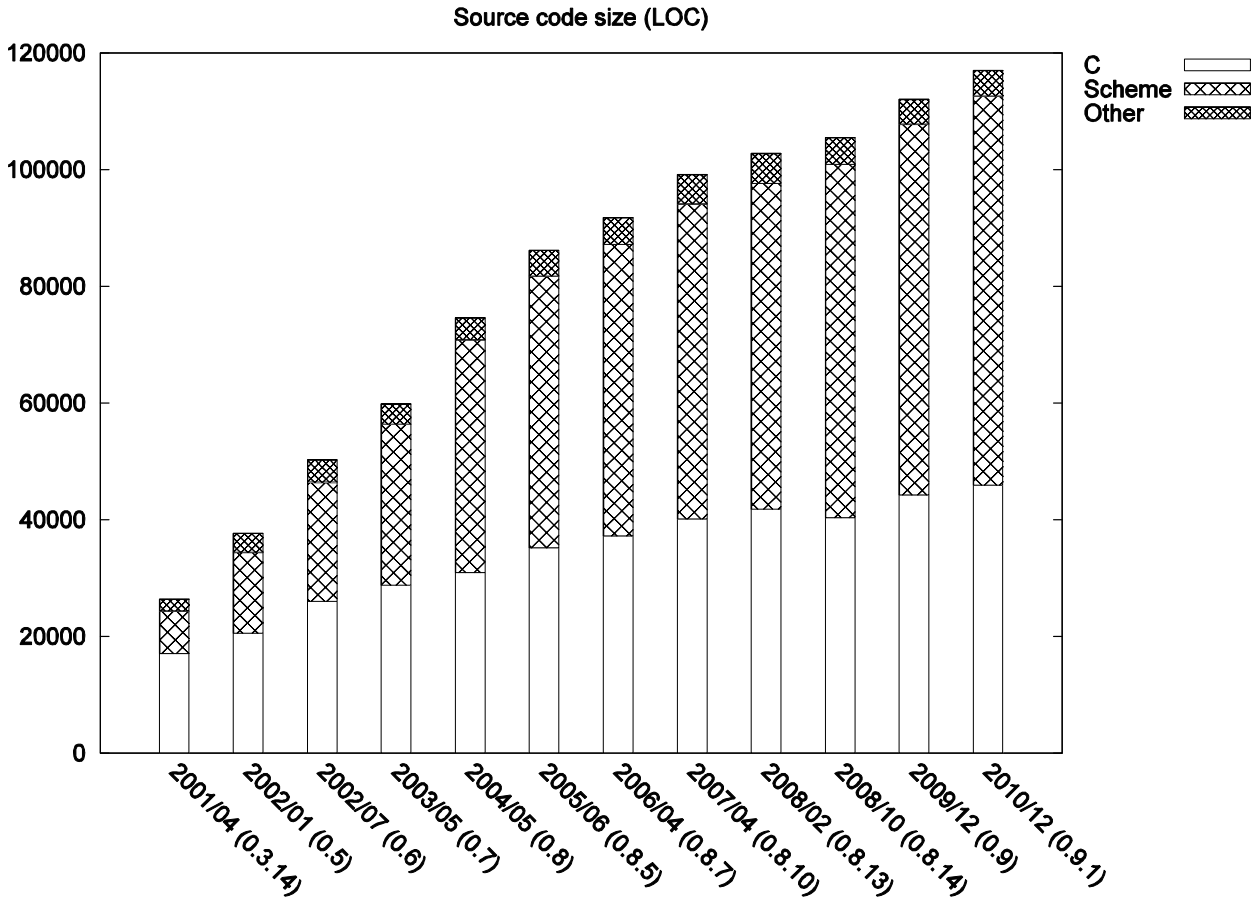

図 1 代表的なリリース毎のソースコード行数の変化．ソースディストリビューション作成時に自動生成されるコード，及び Boehm-Demers-Weiser GC のコードは含まない．

ス規模の推移を，ソースコード行数(LOC)で示した．当初はC言語によるコア部分のコードが大半を占めていたが，Gauche自身でブートストラッピングが可能になってからは，コンパイラやランタイムを Gauche 自身で徐々に書き直している．現在ではLOCの約6割がSchemeコードとなっている．

開発プロジェクトのホスティング（ソースコードリポジトリ，ファイル配布，メーリングリスト）には，オープンソース開発支援サイト SourceForge[†]を利用している．現在リポジトリへのコミット権を持つ者は10名いるが，継続的にコミットを行っているのは筆者一人である．パッチ・バグ報告は月数件程度の頻度で寄せられており，メーリングリストの流量もその程度である．決して大きなプロジェクトではない．

現在のユーザ数については具体的な数を把握してはいないが，日本国内では書籍 [2)]が出版される程度には利用者がいる．英語圏では他に多くの有力な Scheme 処理系があり，Gaucheは決してポピュラーとは言えないが，処理系間の比較などの際に取り上げられる程度には認知されている．

3.国際展開上の留意点

では，Gaucheの開発にあたって，国際展開のために留意してきた点を解説してゆこう．

3.1 ドキュメント

ソフトウェアを「人に使ってもらう」ために，ドキュメントの役割は大きい．実際，後発の Scheme 処理系である Gauche が日本で多くのユーザを得た最も大きな理由は，日本語のリファレンスマニュアルであろう．

しかし，プログラマが主に牽引役となるオープンソースプロジェクトではドキュメントはつい後回しにされがちである．ある程度規模が大きければドキュメント専門の担当者をつけることが出来るだろうが，数人の小さなプロジェクトでは難しい．ドキュメントを用意すること自体負担なのに，複数言語版を揃えるというのは無謀に感じられるかもしれない．

Gaucheでとった戦略は2点ある．まず英語ドキュメントから書くこと，そして執筆者が翻訳しやすいように単一のドキュメントソース内に英語版と日本語版を持ち，ビルド時にそれぞれの言語版を作成することである．

筆者も含め日本語を母語とする者にとっては，日本語を英訳するよりも英語を日本語訳する方がはるかに楽である．最初に日本語でドキュメントを書いてしまうと，その英訳の手間は大きな壁となる．

しかも，英語ドキュメントを読む日本語ユーザは少なくないが，非日本語圏ユーザにとっては日本語ドキュメントしかない状態というのはドキュメントが無いも同じである．多少ハードルは高くても，最初に英語ドキュメントを書いてしまえば，その時点で日英両言語のユーザに利用可能になる．

そして，英語ドキュメントを日本語訳してくれる貢献者を日本のコミュニティ内で見つけるのはその逆に比べれば難しくない．実際，Gauche開発初期は英語ドキュメントが主体だったが，2003年頃からドキュメント翻訳の協力者を何人か得て急速に日本語版ドキュメントが整備された．

図2は，図1に示した各バージョンにおける，ドキュメント量の遷移，および，日本語への翻訳が必要な部分のうちどれだけが実際に翻訳されているかをプロットしたものである．公開機能には必ずドキュメントをつけるという方針のため，ドキュメントの総量はソースコードの量と大まかに連動しているのがわかる．当初はほとんど英語のみだったが，バージョン0.6から0.8にかけて何人かの翻訳協力者が得られたことで一気に翻訳が進み，95%以上が翻訳された状態になった．以降は基本的に筆者が日英両言語のドキュメントをメンテナンスしているが，ドキュメント量の増大，及び多くのリライトにもかかわらず，コンスタントに90-97%の翻訳率をキープできている．一度多くの部分を翻訳してしまえば，2ヶ国語のドキュメントを維持するのは決して難しくない．

複数言語版のドキュメントを管理する際に問題となるのは，追加や変更に各国語版を同期させることである．Gaucheでは，ドキュメントのソースに日本語と英語を併記し，プリプロセスによって英語版と日本語版のドキュメントを生成するようにしている．

ドキュメントシステムには Texinfo[‡] を用いており，ソースに特殊タグを入れて言語を切り替えている．通常はパラグラフを単位として英語と日本語を併置しており，次に示すような表記になっている．

```
@c EN
.. 英語版テキスト ..
@c JP
.. 日本語版テキスト ..
```

[†] http://sourceforge.net/

[‡] http://www.gnu.org/software/texinfo

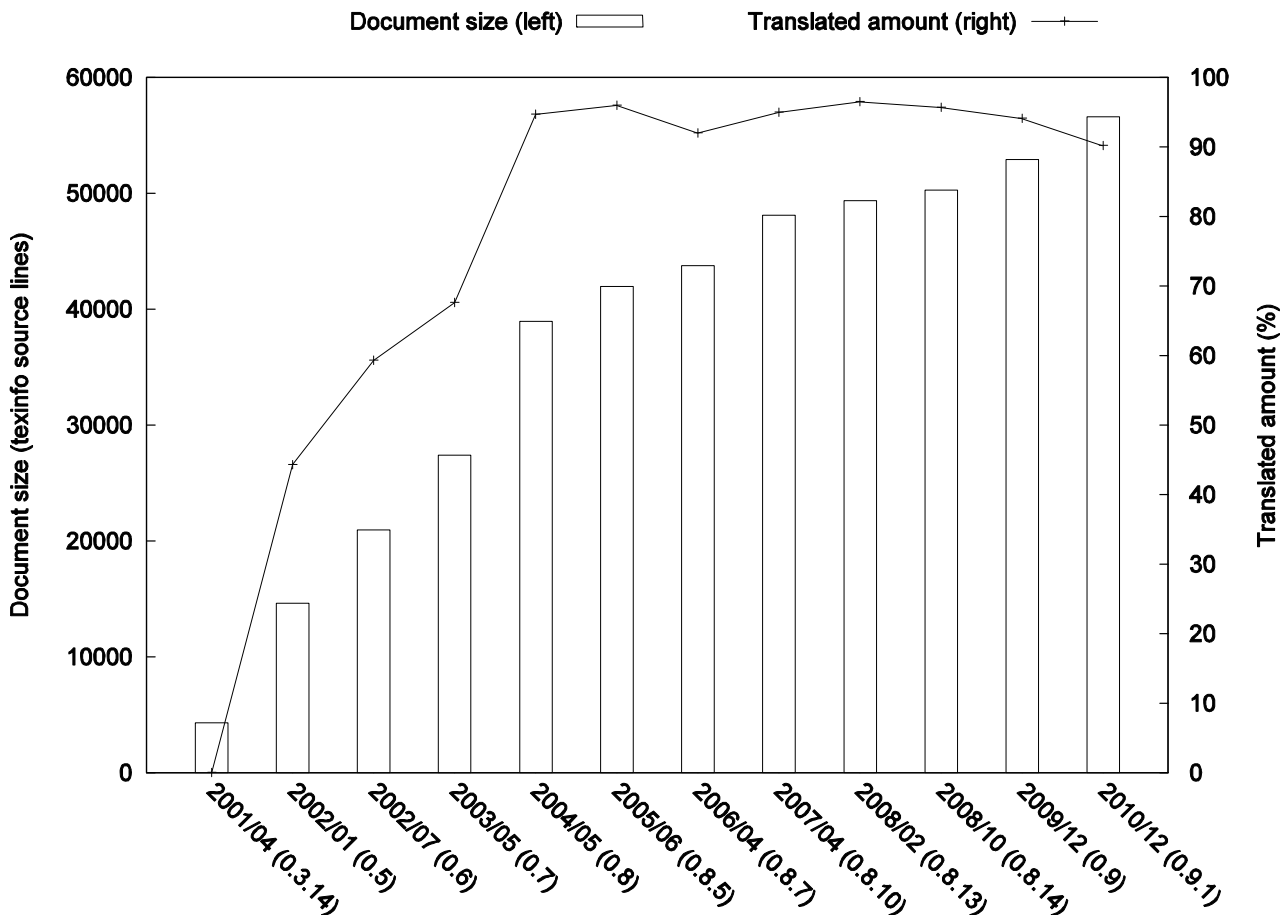

図 2 ドキュメント量および翻訳割合の推移．ドキュメント量は Texinfo ソースのコメントを除いた行数（左目盛）．翻訳割合は，ドキュメントソースのうち，翻訳されるべき部分の行数に対する，実際に翻訳されている行数の割合（右目盛）．

```
@c COMMON
.. 共通テキスト ..
```

クロスリファレンスはすべて英語のノード名で参照し，日本語版生成の際に日本語名に置換する．英語のノード名と日本語のそれとの対応も，次の例のように特殊タグで記している．

```
@node Core syntax, Macros, Programming in Gauche
@chapter Core syntax
@c NODE 基本的な構文
```

ここで@node, @chapter は通常の Texinfo コマンドであり，@c NODE で対応する日本語ノード名を表記している．

これらの特殊タグは Texinfo にとってはコメントになるので，既存の Texinfo 向けの様々なツールがそのまま利用できる．特に，Emacs 上の強力な Texinfo オーサリング支援マクロが使えるのは大きな強みである．

パラグラフの粒度で英語パートと日本語パートを併置するのは翻訳しやすく，また一方を変えた際にもう一方を変え忘れるミスも減らせる．大きな変更で，時間的に両言語を変更するのが難しい場合は，日本語パートを削り英語版を共通パートとすることで，古い情報を残す危険を避けている．

この手法の明らかな欠点はふたつある．3 ヶ国語以上になると，パラグラフ単位でも同期を取るのは急速に難しくなること．また，日英混在のソースは，日本語のわからない執筆者には編集しにくいことだ．こういった要求が出てきたら，例えば gettext[§]のように，ドキュメントのソースを単一言語（おそらく英語）で持ち，そこから各国語向け対訳ファイルを生成して，翻訳はそれぞれの対訳ファイル内で行う，というシステムが必要になるだろう．

ただし，日本人が中心となった「日本発」のソフトウェアが英語圏を考慮するという場合については，当面は日英語のみのサポートで十分なことが多いはずだ．何より開発者の士気を保つことが重要であり，Gauche の方式

§ http://www.gnu.org/software/gettext

が応用可能なケースは多いのではないかと考えられる．

なお，「英語版を先に書いて翻訳する」と説明したが，筆者が日本語版を書く場合には翻訳というよりは，異なる言語でひとつのことがらを説明する，という感覚を持っている．異なる言語を使うことにより，複数の視点から対象を眺められるので，対象への理解がより深まることが多い．日本語版でより良い説明を思いついて英語版を直したり，両言語での説明のしやすさの差異を考えているうちにより簡潔明快な説明に至ることも良くある．これもまた，複数言語版のドキュメントを書くことの利点と言えよう．

3.2 コミュニティ

Gauche 開発上の情報交換には以下のチャネルがある．

- プロジェクト公式サイト（英語，日本語）：リリースノート，ドキュメント等
- メーリングリスト（英語，日本語）
- wiki [**]（言語の制約はないが，現在の参加者は日本語圏のみ）
- chat [††]（言語の制約はないが，現在の参加者は日本語圏のみ）
- blog [‡‡]（英語）：開発上の選択や発見，次バージョンに入る新機能の紹介など気軽な話題

wiki とチャットは自然発生的に生じた場であり，当初はメーリングリストをユーザ参加の情報交換の場と想定していた．

プロジェクト公式サイトやリリースノートについては，ドキュメントと同じく，日英併記の単一ソースファイルから両言語バージョンを生成することで，維持の手間を減らしている．

英語圏と日本語圏の両方で開発者・ユーザコミュニティを運営してゆくにあたって憂慮したのは，コミュニティの分断であった．例えばある設計上の決断が，日本語圏だけの議論でなされた場合，英語圏ユーザは疎外感を感じてしまうのではないか，といったことである．

そのため，次のような方針を設けている．

- 公式サイトのアナウンスは，日英同時か，それが間に合わなければ英語を優先する．過去に筆者の多忙のため英語版のリリースノートしか出せないことがあったが，有志による日本語訳に助けられた．
- 一方の言語圏で開発方針の決定に関わるような，あるいは興味深い議論になった場合は，適宜まとめた情報をもう一方の言語側に流す．

これで 10 年運営してみて，これまでコミュニティ間のディスコミュニケーションが問題になったことはない．尤もそれがこの方針のためなのか，単に Gauche のユーザコミュニティの分断が問題になるほど大きくはないだけなのか，については確かめる術がない．

ただ，日英 2 ヶ国語混在の職場環境の経験から言えることは，異言語コミュニティ間のディスコミュニケーションは，それが明らかな問題になってしまってから修復するのには多大な労力を要するということだ．問題は，翻訳されない情報があることではない．情報の存在が知らされないことが積み重なり，いつしか「知るべき情報が他にもあるのではないか」という不信感を持たれてしまうことである．

全ての情報をタイムリーに翻訳して常に同期を取っておくには膨大なコストを要し，多くのプロジェクトでは非現実的だ．経験的に，「他言語側で何が起きているかの概略がタイムリーに漏れ聞こえてくる」程度の情報のやりとりを保つことが重要である．そこでより詳しく知りたいと思った人は質問するなり翻訳するなり，アクションを起こせるからだ．何もかも翻訳しようとするより，「この情報が欲しいけれど翻訳が無い」というユーザの声をトリガとしてその部分だけを翻訳する方が効率も良く，またユーザの要求にも添うことになる．

プロジェクトの主催者はそのような情報の流れが途切れないことを気にかけるべきだが，全て自分でやらねばならないというものでもない．ある程度ユーザコミュニティが育てば，一方で起きていることをウォッチして情報を伝えてくれるユーザや，翻訳を買って出てくれるユーザが現れる．主催者はコミュニティの反応を観察し，足りない部分をフォローする立場に回るのが良いだろう．

Gauche での具体的な経験では，リリースノートを日英両言語で同時に出さないというのは一方のユーザにかなりの不満を与える，ということがわかった．リファレンスマニュアルならば，知りたい情報が絞られているため英語でも許容する，というユーザであっても，リリースノートのように「多くの情報を，ざっと読んで把握する」という目的では，日本語で読みたいという要望が大きい．

一般的な指針としては，「具体的な情報へのポインタとなる，メタ情報」を優先的に両言語でタイムリーに提供することが，うまくコミュニティを回してゆく秘訣のように思われる．

[**] http://practical-scheme.net/wiliki/wiliki.cgi

[††] http://practical-scheme.net/chaton/gauche

[‡‡] http://blog.practical-scheme.net/gauche

3.3 ソースコード

日本語圏内だけで閉じているプロジェクトでは，ソースコード内のコメントやリテラル，識別子に日本語が使われることがある．非日本語圏の開発者にとっては読めないだけでなく，異なるエンコーディングで編集してしまうことにより意図しない変更を加えてしまう危険性もある[§§].

プログラムソース自体は気をつけていても，設定ファイルやビルド補助スクリプト, あるいはコミットログ等, 周辺のテキストは見落としがちだ．プロジェクト初期から，主要なファイルは US-ASCII で編集できるようにしておき，それ以外の文字を必要とする部分を明確に分離しておくのが良いだろう．プロジェクトに途中参加した開発者は，それまでに確立されたプロジェクトの慣習に従おうとするものだ. コミットログが全て英語であれば, 新規参加者にも英語で書く強い動機付けとなる．

Gauche でひとつ引っかかったのは, 変更点を記録した ChangeLog ファイルであった．パッチを採用した場合に貢献者の名前を記録するのだが，様々な国からの貢献者は名前に使われる文字も多彩である．当初は EUC-JP でファイルを編集していたのだが，Emacs 上で ISO-8859-X の文字をコピー&ペーストすると EUC-JP に無い文字でも Emacs の独自エスケープシーケンスで保存できる場合があり，それにしばらく気づかなかった．そのファイルを別のスクリプトで処理しようとして問題になり，UTF-8 に移行した経緯がある．

コメントやコミットログを英語限定にすべきか，という点には議論の余地があろう．的確なコメントやコミットログを書くのはもともと面倒なもので，負担を上げてしまうとなお書くことが億劫になってしまう．日本語であっても無いよりはまし，という考え方もあり得る．筆者は，拙い英語であればむしろ本質的な要点だけに集中できるので，英語で書くのが良いと考えるが，最終的にはプロジェクト参加者のモチベーションを最大化するように決めるべきだ．ただし，ソースを異なるエンコーディングで編集してしまうのは良く起きるトラブルなので対応を考えておく価値はある．現在であれば UTF-8 で統一するという選択は現実的であろう．

第三者からのコードの貢献への対応にも，ソースの言語問題が影響してくる．コードの貢献は，オープンソースプロジェクトを運営していて最も嬉しいことのひとつだ．しかし，日本人開発者からのコード提供に際して日英のドキュメントをも要求するのは，必要以上にハードルを上げることになりかねない．

Gauche では, 公式の機能とするにはコードに加えドキュメントとテストが揃っていることを原則としているが, コントリビュートされるコードに全て揃っていることは要求しない（もちろん揃っていればそれに越したことはないが). 日本語でもドキュメントをつけてもらえれば, プロジェクト側で英訳を起こして収録する．

パッチを取り込む際には，API やドキュメントのスタイルの統一性をとるために全てに目を通し，その際にドキュメントを手直しすることも良くある．そこで小さなプラスアルファの手間をかけて英語版を整備しておくのが，持続的にコントリビューションを取り込んでゆくコツであろう．

3.4 文字と文字列の処理

ここまではプロジェクト運営上の言語の問題を扱ってきたが，本章では作成するソフトウェア自体が扱う言語の問題を考える．

言語処理系やデータベース，OS などインフラに近い層のソフトウェアでは，文字と文字列の表現モデルは重要な設計上の選択の一つである．一度モデルを決め，リリースしてそれに依存するコードが書かれると，互換性を損なうようなモデルの変更は非常に困難になるからだ. そのため，当初多国語対応を考えずに書かれたシステムに，ひどくアドホックな形で多国語対応処理が追加される例は多い．

Gauche の開発を始めた 2000 年前後にはまだこの問題には未解決の部分が大きく，設計上の選択は大きな決断であった．当時既に Java, ECMAScript 等，いくつかの言語は Unicode を言語仕様で採用していた．しかし，筆者の職場環境でほとんどの日本語ドキュメントは EUC-JP か Shift JIS，英語ドキュメントは ISO-8859-1 で作成されており，ツールがデフォルトで Unicode をサポートしていない場合も多かった．

さらに，松本ら [8] が論ずるように，Unicode を使っても全てのエンコーディングの完全なる上位互換にはならない．例えば Unicode を経由して EUC-JP と Shift JIS を変換し，それを再び EUC-JP に戻した場合，変換テーブルの差異により，元とは異なるテキストになってしまう可能性がある. EUC-JP と Shift JIS は同じ符号化文字集合の異なるエンコーディングであり，演算によって情報の欠落無く相互変換が可能である. 扱うテキストが EUC-JP と Shift JIS ならば，文字マッピングの差異から来る問題

[§§] もちろん日本語に限ったことではなく，筆者は仏語でコメントが入っている ISO-8859-1 エンコーディングのソースを編集後 EUC-JP で保存，コミットしてしまって困ったことがある．

を避ける意味でも，性能的にも，Unicode を経由せずに処理できることが望ましい．

Gauche では，軽量なスクリプト処理系という目標から，ランタイムに複数のエンコーディングの切り替えを許して処理系が複雑化するのを避け，コンパイル時に決定されるエンコーディングで内部処理を統一することにした．処理系自体，現在の計算機環境からすれば大きくはないので，必要に応じて別々の内部エンコーディングでコンパイルしたバージョンを持っておけば良い，という割り切りである．選択可能な内部エンコーディングとしては，UTF-8, EUC-JP (JISX0213:2000), Shift JIS (JISX0213:2000), および ISO-8859-X をサポートしている．

また，JIS の文字集合の範囲では演算によるエンコーディング変換ルーチンを持つことで，Unicode を軸としてレガシーエンコーディングを変換する際の文字マッピングの問題を回避した[***]．EUC-JP と Shift JIS が混在する環境であっても，例えば EUC-JP を内部エンコーディングとした Gauche を使い，内部ルーチンで変換を行うことで，情報の欠落無く両エンコーディングの文書を処理できる．

文字エンコーディングを巡る状況はこの 10 年でも大きく変化した．現在では Unicode で作業フローを統一することはそれほど難しくは無くなっている．Gauche は当初 EUC-JP をデフォルトの内部エンコーディングにしていたが，2005 年からデフォルトを UTF-8 へと変更した．2007 年に制定された Scheme の新言語仕様 R^{6}RS [7] では Unicode に準拠する文字列処理が要求されており，今後も Unicode 主流の動きは揺らがないであろう．そして，Unicode で十分な応用範囲もまた，広がってゆくであろう．実際，英語圏での議論では多国語化の話題が「Unicode で良いだろう」で済ませられる傾向がある．

もちろん Unicode がアプリケーションの要求を満たせるならばそれで良い．ただ，言語処理系はプログラマの思考の根本を支える道具であり，新しい問題の解決に取り組む際に携える武器となるものである．文字と文字列に関して，未知の問題にも柔軟であるためには，早過ぎる最適化に対する警戒も必要だ．

文字の扱いのデザインは，大きな文字集合と何種類ものエンコーディングを扱ってきた日本語圏のノウハウを役に立てるチャンスでもある．

Gauche の文字コード変換ライブラリには，日本語対象の統計的文字コード判別ルーチンが付属しており，「日本語であることは分かっているがどのエンコーディングかはわからない」ような入力を直接オープンできる仕組みがある[†††]．

次のコード片はファイル input.txt をオープンして文字列として読み込むものであるが，Scheme の標準関数 `call-with-input-file` が拡張され，入力エンコーディングを指定する `:encoding` オプションが使えるようになっている．さらにエンコーディング名として「日本語だが具体的なエンコーディングは推測せよ」という意味の"`*jp`"を与えている．

```
(call-with-input-file "input.txt"
  port->string
  :encoding "*jp")
```

統計的エンコーディング判定は 100%の精度ではないので，頑健なソフトウェアを書く場合は誤判定に対する対応も必要だが，日常のスクリプトを書く場合にはこれで十分なことが多い．そして，エンコーディングを推測しなければならないケースというのは，日本語圏であれば日常的に遭遇するものであるのに，その重要性がなかなか英語圏では理解してもらえないことの一つである．

こういった，他国語圏に説明しにくい事例を理解してもらう際にも，それを実装したソフトウェアが英語圏から見える形で存在していることは役に立つ．

3.5 英語圏でのプレゼンス

ここまで書いてきた，ドキュメントの整備や日英両語圏に配慮したコミュニティ運営などは，英語圏ユーザへのアプローチの必要条件にすぎず，これらを整えても直ちにユーザが英語圏に広がるわけではない．

どの程度積極的に英語圏へプロモートするか，というのはプロジェクトの方針によるだろう．ユーザが増えれば反応も増えるが，サポートの負担も同時に増える．

Gauche は，安定版になるまでは非互換な変更を持ち込む可能性を留保したいと考えている．そのような場合は，あまり広まってしまうとかえって不自由になる．従って

***なお，JIS 符号化文字集合と Unicode との変換も処理系内部で独自に行っている．EUC-JP/Shift JIS と UTF-8 を直接変換することで，変換テーブルはコンパクトな trie にパックされている．それ以外の文字エンコーディングは iconv を用いて変換している。

†††アーキテクチャ上は日本語に限らず，複数の判別アルゴリズムを持てるようになっているが，現在実装されているのは日本語向け判別のみである．

新しいリリース時にも，プロジェクトの Web サイトやメーリングリストでのアナウンス程度で，特に Scheme コミュニティや言語コミュニティに積極的にリリース情報を流すということはしていない．

それでも，関係する英語コミュニティでの議論に参加し発言していれば，自ずと存在は知られるようになってゆくものである．そこで関心を抱いてくれた英語圏の少数のユーザがプロジェクトにアクセス可能であることが，英語圏でコアユーザを獲得する重要な鍵であろう．

4. おわりに

Gauche の開発経験をもとに，小規模なオープンソースプロジェクトが日本語圏と英語圏にまたがるユーザ層を得るために考慮すべきことを論じた．

ドキュメントにせよ Web サイトにせよ，日英 2 ヶ国語版を用意するよりは日本語版だけ用意する方が楽である．しかしドキュメントがある程度大きくなってから英訳を用意したり，日本語コメントが散らばったソースを整理したりするのは，新機能を実装したりバグを修正したりする作業に比べれば単調で，モチベーションを維持するのが大変である．大きくなってから英語圏へのサポートを追加するのはコストが高くつくのだ．

最初から日英両言語に対応しておけば，インクリメンタルな積み重ねで日英両言語のサポートを維持でき，負担感はそれほど大きくならない．そしてそのコストの見返りは，ユーザを探す領域が日本から世界へと広がることである．

当初から英語でのサポートも必要とされていた，という点で，Gauche は他の多くの国内のプロジェクトとは異なるかもしれない．それは国際化の重要性について早い段階から考えることに役に立った．しかし，そのために Gauche が特殊な事例になったとは考えていない．当初の対象ユーザが日本国内だけだったとしても，おそらく英語版ドキュメントを書いていただろう．稀少な Scheme プログラマを発掘するのに，探索範囲をわざわざ狭める必要はないからだ．

そして 10 年プロジェクトを運営してきて思うのは，プロジェクトはユーザに支えられるということである．見知らぬユーザからのバグレポート，パッチ，そして単なる感想さえも，プロジェクトに新鮮な光を当ててくれるものだからだ．

大きな，注目を集めるプロジェクトであれば，そのプロジェクトの影響力の大きさが開発者を駆り立てる力ともなろう．しかし数人の人間が，隙間時間に，少しづつ育ててゆく．そんなソフトウェアが，いつしか成熟したものになるには，時間が必要だ．そんな小さなプロジェクトをネットの片隅で諦めずに続けてゆくには，ネットの反対側の隅から届く，ユーザからのフィードバックが，何よりも貴重なのである．

続けること，そしてアクセス可能であること．そうすればいつか，ソフトウェアはそれを必要としている人に届く．ドキュメントは未だ見ぬ新たなユーザへのメッセージだ．

It's not when people notice you're there that they pay attention; it's when they notice you're still *there.—Paul Graham, "The Dream Language"* [1)]

参考文献

1) Graham, Paul: Hackers and Painters, O'Reilly, 2004.
2) Kahua プロジェクト，川合史朗：プログラミング Gauche, O'Reilly Japan, 2008.
3) Kawai, Shiro: Shooting a moving target－An experience in developing a production tracking database, in the Proceedings of Japan Lisp User Group Meeting 2000, Tokyo, 2000.
4) Kawai, Shiro: Tracking assets in the production of 'Final Fantasy: The Spirits Within', in the Proceedings of Game Developers Conference 2002, San Jose, 2002.
5) Kawai, Shiro: Gluing things together－Scheme in the real-time CG content production, in the Proceedings of the First International Lisp Conference, San Francisco, 2002.
6) Kelsey, Richard, Clinger, Will, and Rees, Jonathan (eds.): Revised5 Report on the Algorithmic Language Scheme, Higher –Order and Symbolic Computation, Vol. 11, No.1, 1998.
7) Sperber, Michael,Dybvig, R. Kent, Flatt, Matthew,andvan Straaten, Anton (eds.): Revised6 Report on the Algorithmic Language Scheme, Journal of Functional Programming, Vol. 19, Issue S1, pp.1-301, 2009.
8) 松本行弘，縄手雅彦: スクリプト言語 Ruby の拡張可能な多言語テキスト処理の実装，情報処理学会論文誌 Vol.46, No.11, pp.2633-2642, 2005.

川合　史朗（非会員）
E-mail: shiro@acm.org
Scheme Arts, L.L.C.代表．動的プログラミング言語を用いて，コンテンツの創作過程を支援する技術のコンサルティングを行っている．2002 年までは Square USA Inc.にて CG 映画，ビデオゲームの製作と技術研究開発に従事．1996 年東京大学大学院工学系研究科博士課程電子工学専攻修了．博士（工学）．

投稿受付：2010 年 12 月 31 日
採録決定：2011 年 2 月 16 日
編集担当：竹内　郁雄（早稲田大学）
　　吉野　松樹（日立製作所）