人気の言語を作るには

Paul Graham 原著, Shiro Kawai 日本語訳

友人がかつて、著明なオペレーティングシステムの専門家に、本当に素晴らしいプログラミング言語を設計したいんだと話したことがある。専門家は、それは時間の無駄だよと答えた。プログラミング言語に人気が出るかどうかはその言語の良さとは関係がない、だからいかに良い言語を作ろうと誰も使ってはくれないよ、と。少なくとも、それが彼がかつて設計した言語に起こったことだった。
何がある言語を人気のあるものにするのだろう？ 人気のある言語はその人気に見合っているのだろうか。良いプログラミング言語というものを定義する試みに意味はあるだろうか。どうやればそれができるだろうか。
これらの質問の答えは、ハッカーを観察し、彼等が何を欲しているかを学ぶことで得られると私は考える。プログラミング言語はハッカーのものだ。プログラミング言語は、ハッカーに好まれてこそプログラミング言語たることができる。そうでなければ表示意味 *(denotational semantics)* やコンパイラ設計の単なる練習課題にすぎない。

1. 人気のしくみ

確かに、大抵の人々はプログラミング言語を、その利点だけを見て選ぶことはしない。プログラマの多くは誰か他の人から使うべき言語を指示される。それでも、そのような外部の要因がプログラミング言語の人気に及ぼす影響は、普通に考えられているほど大きくないと私は思う。むしろ大きな問題は、ハッカーが考える良いプログラミング言語というものと、多くの言語設計者が考えるそれとが同じでないことだ。

ハッカーと言語設計者では、重要なのはハッカーの意見の方だ。プログラミング言語は定理のためにあるんじゃない。それは道具であり、人々のために設計されるべきだ。靴が人間の足のためにデザインされるのと全く同様に、人間の強さと弱さに合致するようにデザインされなくちゃならない。彫刻作品と見まがうほど優美な概観を備えていようと、履いたら足が痛くなるようなら、それは悪い靴でしかない。

プログラマの大部分は良い言語と悪い言語の区別がつかないのかもしれない。が、それは他の道具とて同じことだ。多くの人に見分けがつかないからといって良い言語を設計しようとすることが時間の無駄になるなんてことはない。 優れたハッカー は良い言語を見ればそれとわかるし、それを使おうとするはずだ。優れたハッカーはごく少数派ではあるが、その少数の人々が優れたソフトウェアを生み出し、残りのプログラマがどの言語を使うかに影響を与えて行くのだ。実際のところ、それは影響じゃなくて命令になる場合だってある。優れたハッカーは上司や教員として、他のプログラマにどの言語を使うか指示する立場にいることが多いからだ。

優れたハッカーの意見だけがプログラミング言語の相対的な人気を決めるわけではない。過去の遺産 (Cobol) や大げさな宣伝 (Ada, Java) ももちろん影響する。だが、長期的な影響力としてはやはり優れたハッカーの意見が最も強力だと私は考える。臨界点となる初期ユーザ層と十分な時間があれば、プログラミング言語はその力に相等しい人気を得ることができるはずだ。そして、人気が出れば出る程良い言語は悪い言語との差を拡げて行く。現実の、生きたユーザからのフィードバックが常に発展を促すからだ。人気のある言語がいかに変化していったかを見たまえ。Perl と Fortran が最も極端なケースだが、Lisp でさえ大きく変化したのだ。例えば Lisp1.5 はマクロを持っていなかった。MIT のハッカー達が本物のプログラムを書くために Lisp を 2 年ほどいじった後でマクロは追加されたのだ [1]。したがって、人気が出るためには良い言語でなければならないかどうかはともかくとして、良い言語であるためには人気がなければだめだ。そして、良い言語で有り続けるためには人気があり続けなければならない。最先端のプログラミング言語はじっとしていない。こんにちの Lisp は 1980 年代中頃に MIT にあったものからたいして変化していないが、それはその時代を最後に Lisp は発展に必要な大きさのユーザベースを失ったからだ。

もちろん、ハッカーは言語を使うまえにそれを知らなければならない。どうやって彼等は新しい言語を知るのだ? 他のハッカーからだ。だがそれには、いちばん最初にその言語を使い始めるハッカー達がいなければならない。この最初のハッカー達のグループはどのくらいの大きさだろう。臨界点となるためには何人くらいのユーザが必要だろう。私の頭にぱっと浮かんだ答えは、20 人だ。もし言語に 20 人の独立したユーザ、つまり自分でその言語を使うと決めたユーザがいれば、私はその言語が本

物だと考える。

そこに至るのは簡単ではない。0 から 20 に至るのは、おそらく 20 から 1000 に至るのより難しいだろう。最初の 20 人を獲得する最も良い方法はトロイの木馬だ。人々の欲するアプリケーションを、たまたまその言語で書いて提供するのだ。

2. 外部の要因

まず、プログラミング言語の人気に確かに影響を与える一つの外部要因を認めることから始めよう。人気が出るためには、プログラミング言語は人気のあるシステムのスクリプト言語でなければならない。Fortran と Cobol は初期の IBM メインフレームのスクリプト言語だった。C は Unix のスクリプト言語だったし、後に Perl がそうなった。Tcl は Tk のスクリプト言語だ。Java と Javascript は web ブラウザのスクリプト言語として考えられた。

Lisp は残念ながらものすごく人気のあるシステムのスクリプト言語では無く、したがってものすごく人気のある言語に成り得ていない。現在の Lisp の人気は、 Lisp が MIT のスクリプト言語であった 1960 年代と 1970 年代の頃の名残りでしかない。当時の多くの偉大なプログラマはどこかで MIT とつながっている。そして 1970 年代の初期、C の前に、MIT の Lisp 方言である MacLisp は真剣なハッカーが使いたがる唯一の言語であった。

こんにちでは、Lisp はやや人気のある二つのシステム、Emacs と Autocad のスクリプト言語である。おそらく、今日の Lisp プログラミングの大部分は Emacs Lisp と AutoLisp で行われているだろう。

プログラミング言語は孤立しては存在しない。「ハックする」は他動詞だ— ハッカーはふつう何かをハックする—そして現実にはプログラミング言語はそれを使って何をハックするかによって判断される。だから人気のある言語を設計しようと思ったら、言語以上のものを提供するか、既にあるシステムのスクリプト言語を置き換えるかしなくちゃならない。

Common Lisp に人気が出ない理由のひとつは、それが孤児だからだ。Common Lisp はもともとはそれでハックするシステムから生まれた：Lisp Machine である。しかし、Lisp Machine は (並列計算機と共に)、1980 年代の汎用プロセッサの増大するパワーによってぺしゃんこにされてしまった。Common Lisp が Unix の良いスクリプト言語であればまだ人気を保っ

ていたかもしれないが、残念ながらCommon Lispはスクリプト言語としては猛烈にひどい出来だ。

この状況を以て、言語はその利点では判断されないと言うこともできるだろう。別の見方として、プログラミング言語は何かのスクリプト言語でなければ真のプログラミング言語になれないということもできる。このことに今気付いた人にとっては、これはアンフェアに思えるかもしれない。私は、それは、プログラミング言語には実装が伴っていなければならないというのと同じようなものだと思う。それはプログラミング言語というものの一部なのだ。

もちろん、プログラミング言語は良い実装を必要とするし、しかもそれはフリーでなければならない。会社はソフトウェアに金を払うだろうが、個々のハッカーは払わない。そして今惹き付けたいのはハッカーなのだ。

また、言語はそれについて書かれた良い本を必要とする。その本は薄くてうまく書かれていて、また良い例がいっぱい詰まっていなければならない。K&Rは理想的だ。現時点で言うなら、言語にはオライリーから出版された本が無ければならないとも言えるだろう。それはハッカーの関心を得るテストになりつつある。

オンラインドキュメントも必要だ。実際、本は最初はオンラインドキュメントとして書かれ始めたっていい。だが物理的な本の必要性はまだ無くなっていないと思う。物理的な本は便利だし、完璧ではないにせよ出版社を通ったということである程度の質は保障される。本屋は依然として新しい言語を学ぶのに最も重要な場所だ。

3. 簡潔さ

どんな言語にも必要な3つの要素—フリーの実装、本、そしてハックすべきもの— が揃ったとして、さてどうやって言語をハッカーの好むものにできるだろう。

ハッカーが好む要素のひとつは簡潔さだ。ハッカーは怠惰だ。数学者や現代建築家が怠惰であるのと同じように。余分なものは嫌われる。これからプログラムを書こうとしているハッカーは少なくとも無意識的に予測される総タイプ量によってプログラミング言語を選んでいる、と言っても、真実からそれほど遠くはあるまい。これがハッカーの考える方法と正確に一致していないとしても、言語設計者はそう仮定することでうま

くやれるだろう。英語に似せた長くてごちゃごちゃした表現でユーザを子供扱いするのは間違いだ。 Cobol はこの欠陥によって悪名が高い。ハッカーは、

```
z = x + y
```

と書く代わりに

```
add x to y giving z
```

と書けなんて言われたら、良くて彼の知能への挑戦、悪くて神をも怖れぬ大罪だと思うだろう。構文は重要なのだ。

プログラムを読みやすくするために、 Lisp は car と cdr の代わりに first と rest を使うべきだとしばしば言われて来た。ま、最初の 2 時間くらいはそうかもしれない。だがハッカーは、car はリストの最初の要素で cdr は残りの要素だなんてことくらいすぐに覚えられる。

first と rest は 50 簡潔さという点では、強力に型付けされた言語は負ける。他の条件が同じなら、誰もたくさんの宣言の並びからプログラムを書き始めたいとは思わない。暗黙にできることは暗黙になっているべきだ。

個々の語句も短いほうが良い。Perl と Common Lisp はこの点で両極端にある。 Perl のプログラムはほとんど暗号とも呼べる程に詰まっており、一方で Common Lisp の組み込みオペレータの名前は滑稽な程に長い。 Common Lisp の設計者は、言語のユーザはそういう名前を自動的に補完してくれるようなエディタを使うことを前提にしていたのかもしれない。しかし、長い名前が問題になるのは書くときだけじゃない。読むときにもコストがあるのだ。長い名前はより大きなスクリーン上のスペースを占有する。

4. ハックしやすさ

ハッカーにとって、簡潔さより大事なことがある。自分のやりたいことがやれることだ。プログラミング言語の歴史をふりかえってみると、「正しくない」と考えられる行いをプログラマがするのを防ぐために驚くべき程の努力が払われて来た。これは危険なほどにおこがましい計画である。プログラマが必要とするであろうことを、どうやって言語設計者はあらかじめ知ることが出来ると言うのだ? 言語設計者は、ユーザを自分

のミスから守ってやらなきゃならないようなまぬけではなく、設計者が考えもしなかったようなことを実現できる天才と考えた方が良いと思う。何をしようがまぬけは自分の足を撃つのだ。他のパッケージの変数を参照するミスを防いでやることはできたとしても、問題設定を間違えた悪い設計のプログラムを延々と書き続けることから救うことはできない。

良いプログラマはしばしば危険で不道徳なことをしたがる。不道徳とは、ここでは言語が呈示しようとしている意味構造の裏口を覗くような行為だ。高レベルの抽象化されたデータの内部表現をいじってみるようなことだ。ハッカーはハックするのが好きで、ハックというのはそもそも物事の内側に入り込んでもとの設計者を後知恵で批判するようなものだからだ。

後から批判され修正されることを恐れるな。どんなツールでも、人々はもとの設計者が意図しなかったように使うものだし、プログラミング言語のように明確な意図のあるものではとくにそうだ。たくさんのハッカーが、あなたが想像だにしなかった方法で意味論的モデルをいじりたがるだろう。そうさせよう。ガベージコレクタのようなランタイムシステムを危険にさらさないぎりぎりのところまで、内部にアクセスできるようにしておこう。私はCommon Lispでしばしば構造体のフィールドをループしたいと思うことがあった。例えば削除されたオブジェクトへのリファレンスを抜きだしたり、初期化されていないフィールドを見つけたりするようなことだ。構造体が内部では単なるベクタで表現されていることを私は知っている。それでも私はどんな構造体に対しても呼べる汎用関数を書くことができない。構造体のフィールドには名前でしかアクセスできないからだ。それがそもそも構造体の意図するところだからだ。

ハッカーが大きなプログラムの中で意図されたモデルを破りたいと思うのは多分1箇所か2箇所くらいのものだろう。それでも、それがどんなに大きな違いをもたらすことか。そして、それは単に問題を解決できるというだけじゃない。そこには一種の快感が潜んでいる。外科医がぞっとするような内臓をのぞき込む時の密かな快感[2]や、ティーンエイジャーがニキビを潰す時の密かな快感と同じだ。少なくとも男の子にとって、ある種の恐怖は魅力的だ。雑誌Maximは毎年ピンナップとぞっとするような事故の写真を集めた写真集を発刊している。彼等は自分達の客を良く分かっている。

歴史的に、Lispはハッカー達に内部をいじくるらせるのがうまかった。Common Lispの政治的に正しくあろうとする態度の方が邪道なのだ。初

期の Lisp は何でもありだった。幸いなことに、マクロにはその精神の多くがまだ残されている。ソースコードを好きなように変換できるということの何と素晴らしいことか。

クラシックなマクロは本物のハッカーの道具だ—簡単で、強力で、危険だ。それが何をするかを理解するのはあまりに容易だ。マクロの引数を以て関数を呼び出し、それが返したものをマクロ呼び出しがある場所に挿入する。健全なマクロは全く反対の原理を表明している。それがやっていることを理解するのを妨げているのだ。健全なマクロを一つの文で説明したものを見たことがない。健全なマクロはプログラマの望むところを規定しようとする危険な思想の古い例である。健全なマクロは、他のいくつかのことと共に、変数の捕獲から私を守ってくれるが、変数の捕獲こそが私がマクロでやりたいことの一つなのだ。本当に良い言語は、綺麗でかつ汚くあるべきだ。綺麗にデザインされ、直交性の高いオペレータと良く理解された小さなコアで構成され、しかしハッカーが好きなことをできるような汚さを備えている。 C がこれに当てはまる。初期の Lisp もそうであった。本当のハッカーの言語は多少野卑な性質を持っているものだ。

良い言語は、「ソフトウェア工学」なんて言葉を使うような人々がこりゃダメだと頭を振るような機能を持っているべきだ。その対極にある言語は Ada や Pascal のようなもので、礼儀正しくて教育には適しているが他のことにはあまり使えない。

5. 書き捨てのプログラム

ハッカーを惹き付けるには、言語はハッカーが書きたがるようなプログラムを書くのに適していなければならない。それは、意外かもしれないが、書き捨てのプログラムを書くのに適していなければならないということだ。

書き捨てのプログラムとは、限定された仕事をするために手早く書き上げる類のプログラムのことだ。何かのシステム管理の仕事を自動化したり、シミュレーションのテストデータを生成したり、データのフォーマット変換をするような。このような書き捨てのプログラムに関する意外な事実とは、第 2 次大戦中に多くのアメリカの大学で建設された「一時的な」ビルディングのように、それがしばしば捨てられずに使われ続けることである。たくさんのそういったプログラムは本物のユーザと本

物の機能を得て、本物のプログラムへと進化してゆく。

私は、最良の大きなプログラムは、フーバーダムのように最初から大きく設計されるのではなく、書き捨てのプログラムから生まれるんじゃないかという気がしている。ゼロから巨大なものを建設するのはおっかないことだ。巨大すぎるプロジェクトに取り掛かるとき、人々は圧倒される。そんなプロジェクトは泥沼にはまるか、貧弱で気の抜けた結果しか出さない。本物のダウンタウンの代わりにショッピングモールになったり、ローマではなくブラジリアになったり、C ではなく Ada になったり。

大きなプログラムを書く別の方法は、書き捨てのプログラムから始めて改善し続けることだ。このアプローチは気力をくじくことがないし、進化することでプログラムの設計を良くして行くことができる。見回してみれば、多くの大きなプログラムはこのようにして作られて来た。そしてこの方法で進化を遂げたものは、依然としてそれが最初に書かれた言語で書かれているだろう。プログラムが他の言語に移植されるなんて、政治的な理由でもなけりゃ起こらないからだ。従って、逆説的であるが、もし大きなシステムで使われる言語を作りたいと思うなら、書き捨てのプログラムを書くのに良い言語にしなくちゃならない。大きなシステムはそこから始まる。

Perl がこのアイディアの最も印象的な例であろう。Perl は書き捨てのプログラムのために設計されたというだけでなく、Perl 自身が書き捨てのプログラムのようなものだった。 Perl はレポートを生成するためのユーティリティのコレクションとして産声を上げ、人々が書く書き捨てのプログラムが大きくなってきてからプログラミング言語へと進化した。 Perl 5 になってようやく重要なプログラムを書くのに相応しい言語になったが、既にその時には非常にポピュラーになっていたのだ。

書き捨てのプログラムを書くのに良い言語とはどういうものだろう。まず、それはいつでも使える状態になければならない。書き捨てのプログラムとはあなたが 1 時間かそこいらで書きたいと思うものだ。だからその言語は既にあなたのマシンにインストールされていなければならないだろう。使う前にわざわざインストールしなくちゃならないんじゃだめだ。そこに無いと。 C はそこにある。OS についてくるからだ。Perl もそこにある。もともとシステム管理のためのツールで、それゆえに既にインストールされていることが多いからだ。

しかし、使える状態にあるとは、インストールされているというだけではない。コマンドラインインタフェースを持つ会話的な言語の方が、い

ちいちコンパイルと実行をわけなくちゃならない言語より使える。人気の出るプログラミング言語は会話的でなければならず、しかも素早く立ち上がらなくちゃならない。

書き捨てのプログラムに必要なもうひとつのことは、簡潔さだ。簡潔さは常にハッカーを惹き付けるが、1時間で書き上げたいと思うようなプログラムではなおさらだ。

6. ライブラリ

もちろん、究極の簡潔さとは、欲しいプログラムが既に書かれていてそれを呼ぶだけ、というものだ。そこで、私がプログラミング言語の中でますます重要になってゆくと思われることに話を移そう。ライブラリ関数だ。Perl は文字列操作の豊富なライブラリのおかげで人気が出た。このクラスのライブラリは書き捨てのプログラムではとくに重要だ。データを変換したり取り出したりするものが多いからだ。たくさんの Perl プログラムは、最初はいくつかのライブラリコールをつなぎ合わせただけのものから始まるだろう。

この先50年のプログラミング言語の進化は、ライブラリ関数に関するものになるだろうと思う。未来のプログラミング言語は、言語のコアと同じくらい慎重に設計されたライブラリを備えているだろう。プログラミング言語の設計の重点は、強い型付けにするか弱い型付けするかとかオブジェクト思考にするかとか関数型にするかとかそういうことではなく、どうやったら素晴らしいライブラリを設計できるかということになってゆくだろう。型システムの設計みたいなことを考えるのが好きな言語設計者は身震いするかもしれない。ライブラリの設計だなんて、アプリケーションを書くみたいなことじゃないか! 残念でした。言語はプログラマのためのもので、プログラマが必要としているのはライブラリなのだ。良いライブラリを設計するのは難しい。たくさんのコードを書けば良いというものではない。ライブラリが大きくなりすぎると、必要な関数を探し回るより自分で書いてしまった方が早いなんてことが起こる。ライブラリは、コア言語と同様に、小さな直交性の高い操作を使って設計されなければならない。プログラマが、どのライブラリが欲しい機能を実行できるかを推測できるようになっていなければならない。

ライブラリは、Common Lisp が失敗した点のひとつだ。文字列操作にはごく原始的なライブラリしかないし、オペレーティングシステムにア

クセスするものはほとんど皆無だ。歴史的な理由から、Common Lisp は OS なんて存在しないかのようにふるまってきた。だが OS にアクセスできないということは、Common Lisp 組み込みのオペレータだけではまともなプログラムは書けないということになる。実装に依存する何らかのハックをつかわなくちゃならないだろう。そして、そういうハックはあなたの欲しいものを全ては与えてくれない。 Common Lisp に強力な文字列ライブラリと良い OS インタフェースがあったなら、ハッカーは Lisp をもっと好んでいただろう。

7. 構文

Lisp の構文を持つ言語、もっと正確に言えば構文を持たない言語はポピュラーになれるだろうか? この問題の答えを私は知らない。私が考えるのは、Lisp が現在ポピュラーでない主要な理由はその構文ではないということだ。Common Lisp は、見慣れない構文よりもっと悪い問題を抱えている。前置構文を使いこなせるプログラマでも、強力な文字列処理と OS インタフェースのために Perl の方を使う、という例を私はいくつか知っている。

前置記法には 2 つばかり問題があるかもしれない。プログラマにとって馴染みが少ないということと、密度が高くないということだ。 Lisp の世界の伝統的な解釈では最初の問題こそが真の問題とされていた。だが本当にそうだろうか。確かに前置記法は普通のプログラマを狼狽させる。だが、普通のプログラマの意見は重要じゃない。優れたハッカーがどう考えるかが言語の人気を決めるのだ。Perl の構文はほとんど理解不能だが、それが Perl の人気の足を引っ張っているとは思えない。Perl が備えているそういう性質はむしろ Perl 信者を元気づけてきたのだ。

より深刻な問題は、前置記法の散慢さだ。優れたハッカーにとってはこれが本当の問題だ。誰も、a[x,y] と書けるところを (aref a x y) と書きたいとは思わない。

この特定の例に関しては解決策がある。データ構造を、インデックスが渡された時は関数であるかのように振舞わせれば、(a x y) のように書くことができて、これは Perl の書式よりも短い。他の式も似たようなトリックを使って短くできるだろう。

インデントに意味を持たせれば、たくさんの括弧を取り除くか省略可能にできる。プログラマがコードを読む時は結局インデントを見ている

のだ。インデントの示す構造とデリミタの示す構造が食い違っている場合、我々は大抵インデントの方を見てしまう。インデントに意味を持たせれば、このようなよくあるバグの元を減らせると共に、プログラムを短くできる。

中置記法は確かに読みやすい。とくに数式ではそうだ。私はプログラマのキャリアとしてずっと Lisp を使ってきたが、それでも前置記法による数式が自然であるとは思えない。だがそれは便利なのだ。特に可変長引数を取るオペレータを使うコードを生成するような場合にだ。もし中置記法を採用するなら、リーダーマクロの一種として実装するのが良いだろう。

Lisp に構文を持ち込むことに対して宗教的な反発を感じるべきではないと思う。少なくともその構文が、ベースとなる S 式へとよく理解された方法で変換されるならば。既に Lisp には結構な量の構文が持ち込まれているのだし、その使用を強制しないのであればさらに構文を追加しても悪いことはなかろう。Common Lisp ではいくつかのデリミタは言語に予約となっているから、少なくとも設計者の一部は将来もっと多くの構文を追加することを意図していたのだろう。

Common Lisp の中で、言語道断に Lisp らしからぬ構文は format 文字列にある。 format はそれ自身の言語を持っており、それは Lisp ではない。Lisp に更に構文を追加するなら format 記述子もそれに含めるべきだ。マクロが他のコードを生成するのと同じように format 記述も生成できたら良い。

ある非常に優れた Lisp ハッカーが、彼の CLtL [訳注 1] は format のところで開く癖がついてしまったと私に語ったことがある。実は私のもそうだ。多分これは改善すべき箇所を暗示している。また、プログラムは I/O をたくさんするものだということも意味している。

8. 効率

皆知っているように、良い言語は速いコードを生成しなければならない。しかし現実には、言語設計において速いコードを出すことが最優先されるべきとは思えない。Knuth がずっと昔に指摘したように、速度は重要なボトルネックとなる箇所のみで問題となる。そして、たくさんのプログラマが経験していることだが、どこがボトルネックかを読み違えるのは非常に良くある間違いだ。

従って、速いコードを得る現実的な方法は、言語を強い型付けにしたりすることじゃなく、非常に良いプロファイラを作ることだ。プログラム中の全ての関数の全ての引数の型を知る必要なんてない。ただ、ボトルネックとなるところの引数の型を宣言してやる必要はある。そうするには、どこがボトルネックかを知る必要がある。

Lisp に関して言われ続けて来た不満は、何が高価な操作かを知るのが難しいというものだ。これはおそらく真実だろう。だが、高度に抽象的な言語を持つ以上それは避けがたいことでもある。そして、良いプロファイリングこそが問題を解決する正しい道だと私は考える。何が高価かはそれで知ることができる。

この問題の一部は社会的なものだ。言語設計者は速いコンパイラを書きたがる。だってそれが彼等の技術を示すものだから。彼等にとっては、プロファイラは眼中にあったとしても附属品にすぎない。だが現場では、実際のプログラムを改善するには速いコードを出すコンパイラよりも良いプロファイラの方が役に立つ。ここでも、言語設計者はそのユーザから乖離している。彼等は問題を解決するために素晴らしい仕事をしているが、その問題は現場の問題とちょっとばかりずれているのだ。

アクティブプロファイラ—プログラマが尋ねた時だけでなく、パフォーマンスデータを常にプログラマに示してくれるようなプロファイラ—は良いアイディアかもしれない。例えばプログラマがソースを編集するときに、エディタはボトルネックを赤く表示するといった具合だ。別のアプローチは、走っているプログラムの中で起こっていることを何らかの方法で表現することだ。サーバーベースのアプリケーションではこれは特に役に立つだろう。たくさんの走っているプログラムを見ていなければならないからだ。アクティブプロファイラは走っているプログラムのメモリのアクセス状況をグラフィカルに表示したりとか、音で知らせるとかできるかもしれない。

音は問題を知らせる良い合図だ。私がかつて仕事をしたところでは、Web サーバの状態を見せるメーターのついた大きなボードがあった。何かあると、メーターの針が小さなサーボモーターで動かされるのでちょっとした音を立てる。私の席からはそのボードは見えなかったのだが、音だけでサーバーに問題が生じた時はすぐにわかった。

効率の悪いアルゴリズムの箇所を自動的に検出するプロファイラを書くことさえも可能かもしれない。ある種のメモリアクセスのパターンが悪いアルゴリズムのサインだと聞いても私は驚かない。コンピュータの

中に我々のプログラムを実行してくれるこびとが走り回っているとしたら、彼は連邦政府の雇用者が語るのと同じくらい、彼の仕事のどこがどれだけ非効率かを語ってくれるだろう。私はよく、今自分はプロセッサに野生のがちょうを追っかけさせるようなことをしているんだろうなと感じることがあるが、実際に何がどうなっているのかを見る良い方法は無かった。

多くの Lisp 処理系はバイトコードにコンパイルしてそれをインタプリタで実行する。これは移植を簡単にするために行われることが多いが、言語の機能としても有用かもしれない。バイトコードを言語のオフィシャルな定義の一部としてしまって、プログラマがボトルネックの部分ではバイトコードをインラインで書けるようにするのだ。そうすればそういう最適化さえも移植可能になる。

エンドユーザが感じるアプリケーションのスピードの性質は変化してきている。サーバーベースのアプリケーションが増えるにつれ、プログラムは I/O に律速されることが多くなって来た。I/O を速くすることには価値があるだろう。言語は、単純にフォーマット出力を速くすることによっても、またキャッシュや永続オブジェクトのような深い内部構造の変更によってもこれをサポートすることができる。

ユーザはレスポンスタイムに関心がある。しかし、もうひとつの効率の指標が重要になってきている。プロセッサ当たりのサポート可能なユーザ数だ。近い将来、たくさんの興味深いアプリケーションがサーバーベースで書かれるようになるだろう。そのようなアプリケーションをホストする立場で最も重要なのがサーバー当たりどれだけのユーザを受け付けられるかという点だ。サーバーベースのビジネスの総コストに対して、その値は分母に効いて来る。

長い間、エンドユーザ向けのアプリケーションにとって効率はあまり重要ではなかった。開発者は、いつでもユーザはより強力なプロセッサを手に入れてゆくことを当てにできた。そしてパーキンソンの法則により、ソフトウェアはそこにある資源を使い尽くすように拡大する。サーバーベースのアプリケーションではこの流れは変わるだろう。そこでは、ハードウェアとソフトウェアはセットで提供される。サーバーベースのアプリケーションを提供する企業は、サーバー当たりのユーザ数の違いによって大きな差がつくだろう。ある種のアプリケーションでは、プロセッサが律速となり、実行速度の最適化が主要な課題となるだろう。だが、メモリが律速となることも多い。同時接続ユーザ数は各ユーザのた

めに必要なメモリ量でも決って来る。ここでも言語が助けになる可能性がある。良いスレッドのサポートがあれば、全てのユーザが一つのヒープを共有することも可能かもしれない。永続オブジェクトのサポートや、言語レベルでの遅延ロードなども助けになるだろう。

9. 時間

人気のある言語が必要とする最後の成分は時間である。誰も、消えてしまう言語でプログラムを書きたいとは思わない。そしてたくさんの言語が現われては消えて行く。大抵のハッカーは、言語が作られてから最低でも 2 年間くらい経ってようやく、それを使うことを考え始める。

新しい素敵なものを考え出す発明家はしばしば見落とすのだが、人々にメッセージが行きわたるには時間がかかるのだ。私のある友人は誰かに何かを頼まれても、一回目は何もしようとしない。人々はよく何かを思い付きで頼んで、後からそれが不要だったとわかることが多いと彼は知っているからだ。時間を浪費しないために、彼は 3 回か 4 回目のリクエストが来るまで待つ。そのころにはリクエストを出した人はかなりいらついているが、少なくともその人はそれが本当に必要だということがわかっている。

大抵の人は、耳にする新しいことに対して、似たようなフィルタを身につけている。何かを 10 回は耳にするまで、人々はそのことに注意を払おうとさえもしない。これは全く正当なことだ。新しくてホットなんとやらの大部分は時間の無駄で、消えていってしまうものだからだ。私自身、VRML を学ぶのを遅らせることでそれを学ぶことそのものを避けることができた。

したがって、何か新しいものを発明した人は、人々がその価値に気付くまで、何年も根気良くメッセージを繰り返すことを覚悟しておかねばならない。我々は、私の知る限りでは最初のウェブサーバーベースのアプリケーションを書いたが、人々がそれを使うのに何もダウンロードする必要が無いと納得するまでに何年もかかったのだ。人々が間抜けだったわけではない。彼等は、我々にメッセージを届けさせたのだ。

この事実の良い面は、単純な繰り返しで解決できることだ。とにかく自分の伝えたいことを話し続ければ、いずれ人々は耳を傾けてくれるようになる。人々があなたに注意を払うのは、あなたがそこに居ることに気付いた時じゃなく、あなたがまだそこに居ることに気付いた時だ。

また、勢いを得るまでにも時間がかかるのが普通だ。多くの技術は最初に発表された時から大きく発展を遂げるものだし、プログラミング言語では特にそうだ。新しい技術にとって一番良いことは、最初に採用してくれる少数の人々によって 2〜3 年間使われることだ。最初に使ってくれる人々は知識があり要求も高く、あなたの技術に残っている欠点を速やかに洗い流してくれる。また、ユーザが少数であればあなたは彼等全員と緊密にコンタクトが取れる。最初に使ってくれる人達は、あなたがシステムを改善することには、それが既存のシステムを動かなくするものであったとしても寛容だ。

新しい技術を紹介するには二つの方法がある。有機的に成長する方法と、ビッグバンみたいな方法だ。有機的な成長は、低予算のガレージベンチャーに実例を見ることができる。誰にも知られずひっそりと、2 人ばかりの若者が、新しい技術を創り出す。彼等はそれをマーケティングも無しに発表し、最初はごく少数のユーザしか得られない (でもそのユーザはその技術の熱狂的なファンだったりする)。彼等は技術を改善し続け、同時に彼等のユーザベースは口コミで広がって行く。彼等がそれと気付く前に、彼等は大きくなっている。

もう一つのアプローチ、ビッグバン法は、ベンチャーキャピタルにバックアップされて、派手にマーケティングされたベンチャー企業に例を見ることができる。彼等は製品を出すのを急ぎ、華々しくそれを発表して、(彼等の望みでは) 直ちに大きなユーザベースを得る。だいたい、ガレージ組はビックバン組を妬む。ビックバン組は如才なく、自信たっぷりで出資者からも一目置かれている。彼等は何でも最上の物を手に入れられ、発表の際の広告キャンペーンによって一躍有名人だ。有機的成長組はガレージに座って、貧しく愛されていないと感じる。しかし、有機的成長組は自分のことをそんなふうに考えなくても良いのだ。有機的成長はビッグバンよりも結果的に良い技術と支持者を得ている。こんにち有力な技術を見てみれば、それらの大抵のものは有機的に成長したものだとわかるだろう。

このパターンは企業だけでなく、後援された研究にもあてはまる。Multics と Common Lisp はビッグバンプロジェクトで、Unix と MacLisp は有機的成長プロジェクトだった。

10. 再デザイン

「最も良い書き方は書き直すことだ (The best writing is rewriting)」と E. B. White は書いた [訳注 2]。どんな作家もこれを知っている。ソフトウェアについてもこの命題は真実だ。デザインの最も重要な段階は再デザインである。プログラミング言語は特に十分に再デザインされていない。

良いソフトウェアを書くには、二つの相反する考えを頭に置いておかなくちゃならない。若いハッカーの、自分の能力に対する無邪気な信頼と、ベテランの懐疑主義だ。「こんなの簡単だよ」と頭の半分で考える一方で、「こんなのうまくいきっこない」ともう半分で考えるのだ。

ポイントは、実はこの二つには矛盾は無いということだ。あなたは異なるものについてそれぞれ楽観的と懐疑的になっているのだ。問題を解決できる可能性に関する楽観と、得られた解決の価値に対する懐疑だ。

素晴らしい仕事をする人はしばしば、自分がやっていることには価値が無いと考えてしまう。他の人が見ればその仕事はまさに驚異的なのに、それを創った当人は心配でいっぱいなのだ。そうなるのには理由がある。その心配こそが、仕事を良くするものなのだ。

希望と心配のバランスをうまく取れれば、それはあなたが二本の足で自転車をこぐように、プロジェクトを前へ押し進めてくれる。発展の二サイクルエンジンを考えてみよう。最初のフェーズでは、自分はその問題を解決できるという確信に押されて、あなたは狂ったように問題に取り組む。第二フェーズでは、あなたは自分のやったことを冷たい朝日の下で眺め、その欠陥をはっきりと目にする。だが批判的な精神が希望を押し潰さない限り、あなたは自分のシステムが不完全であることを認め、考えるのだ。あとこんだけ頑張ればできるかな。そしてサイクルを続けるのだ。

二つの力をバランスさせておくのは難しい。若いハッカーでは楽観主義が支配しがちだ。彼等は何かを創り出し、それが素晴らしいものだと信じて疑わず、よりよくして行こうと思わない。年老いたハッカーでは懐疑主義が支配し、野心的なプロジェクトに手を付けようとさえしない。

再デザインのサイクルを回し続けられるものなら何であれ良い。散文は、作者が満足するまで何度も何度も書き直される。しかしソフトウェアは一般的に十分に再デザインされない。散文には読者がいるが、ソフトウェアにいるのはユーザだ。作家がエッセイを書き直したところで、古い版を読んでいた読者が新しい版は前と互換性がなくなったから自分の

考えが動かなくなったなんて文句を言って来ることはない。

ユーザは諸刃の剣だ。彼等はあなたの言語を改善してゆく助けになるが、同時に改善を止める要素にもなる。だからユーザは慎重に選ばなければならず、ゆっくりと増やして行かねばならない。ユーザを持つことは最適化のプロセスとも似ている。賢いやりかたは、それを後回しにすることだ。また、一般的なルールとして、どんな時でも、思った以上に変更はきくものだ。変更を入れることは、包帯をはがすのに似ている。痛みを感じたと思ったらそれはもう過去の思い出だ。

誰もが、言語は委員会で設計してはいけないと知っている。委員会は悪い設計をする。だが私の意見では、委員会の最大の危険は、再デザインをやりにくくすることだ。誰もが現状でどうにか満足しているときに新しい変更を入れるのには莫大な労力を要する。委員会が決めたことは、たとえ大部分のメンバーがそれを本当は気に入ってないにしても、そのままになりがちだ。

たった二人からなる委員会でさえ再デザインの妨げになる。別々の人が書いた二つのソフトウェア部品のインタフェースでよくそれは起こる。インタフェースを変更するには、両者がそれに合意して同時に変更しなければならない。そのせいで、インタフェースは変更されずに残ることが多く、結果としてどんなシステムでも後付けの問題として抱えることになる。

一つの解決法は、システムのインタフェースを水平でなく垂直に設計することだ— モジュールが常に抽象化の階層として上下に積まれるようにする。そうするとインタフェースはモジュールのどちらかが責任を持つことになりやすい。下のモジュールが、上のモジュールが利用する言語だとすれば、下のモジュールがインタフェースに責任を持てるし、下のモジュールがスレーブであれば、上のモジュールがインタフェースを支配できる。

11. Lisp

ここまでの議論が暗示するのは、新しい Lisp に希望があるということだ。ハッカーにその欲するものを与える言語なら何でも希望があるし、Lisp もそれに含まれる。 Lisp の奇妙さをハッカーが嫌ったと結論づけるのは間違いだったんじゃないか。この仮説の心地よい幻覚は、我々の目を Lisp の本当の問題から逸して来た。少なくともハッカーがやりたいこと

を駄目にする Common Lisp についてはそうだ。ハッカーの言語は強力なライブラリとハックすべき物を必要とする。 Common Lisp にはどちらも無い。ハッカーの言語は簡潔でハック可能でなくてはならない。Common Lisp はそうではない。

良いニュースもある。これは Common Lisp が駄目だってことで、 Lisp が駄目だってことじゃない。本物のハッカーの言語となるべき新しい Lisp を作ることができれば、ハッカーはそれを使うんじゃないかと私は思う。ハッカーはやりたいことにちゃんと使える言語なら何であれ使うだろう。我々のやるべきことは、新しい Lisp がいくつかの重要な用途で他の言語よりもうまく動くようにすることだ。

歴史が勇気を与えてくれる。次々と現われる新しいプログラミング言語は、 Lisp から機能を次々と採り入れている。新しい言語が Lisp になるために欠けている機能はもうあまり残っていない。最も新しいホットな言語 Python は中置記法を使いマクロを除いて Lisp を薄めたものだ。新しい Lisp はこの進歩の自然な一歩になるだろう。

マーケティングのためには、新しい Lisp を進歩した Python だと呼んだほうがいいんじゃないかと考えることもある。Lisp とか言うよりもずっとカッコいい。多くの人にとって、 Lisp は括弧だらけの遅い AI 向け言語だ。Fritz Kunze の正式な経歴からは L の付く単語が慎重に除かれている。だが、新しい Lisp を Lisp と呼ぶことを恐れてはならないだろう。 Lisp は未だに、最高のハッカー達—例えばコース 6.001 [訳注 3] を履修して理解したような人達から隠れた尊敬を集めている。彼等こそが我々が惹き付けなければならない人々だ。「ハッカーになるには」の中で、エリック・レイモンドは Lisp をラテン語かギリシャ語のように形容している。実際には使わないかもしれないが、教養のために学ぶべきだと：LISP は、それをものにしたときのすばらしい悟り体験のために勉強しましょう。この体験は、その後の人生でよりよいプログラマーとなる手助けとなるはずです。たとえ、実際には LISP そのものをあまり使わなくても。

私が Lisp を知らずにこれを読んだとしたら、質問したくて仕方無くなっただろう。私をよりよいプログラマにしてくれるだって? (それもその後の人生で?) 言語は道具のはずだ。その言語が自分をよいプログラマにしてくれるってことは、それが何かを意味するとしたら、その言語がプログラミングによりよいってことじゃないのか。実際、レイモンドの言ってることは、彼自身は自覚していないかもしれないが、それを暗示している。

こんな風に思われている限りにおいては、ハッカーは新しい Lisp を、たとえ Lisp と名がついていても、受け入れるんじゃないかと思う。但しこの Lisp はハッカーの言語じゃなくちゃだめだ。1970 年代の Lisp がそうであったように。簡潔で、単純で、ハック可能な。そして今日のハッカーがやりたいようなことを実現する強力なライブラリを備えている。

ライブラリに関して言えば、Perl や Python といった言語に対して彼等の土俵で勝てる余地はあると私は思っている。これから書かれる新しいアプリケーションの多くは サーバベースのものになるだろう。だから、新しい Lisp が Perl と同じくらい良い文字列ライブラリを備え、さらにサーバベースのアプリケーションに必要な強力なライブラリを備えていたら、それは人気が出るんじゃないか。本物のハッカーは、難しい問題をほんの数個のライブラリコールが解決してくれるようなツールを軽蔑したりはしない。そう、ハッカーは怠惰なものなんだ。

コア言語がサーバベースのアプリケーションをサポートしていればなお良いだろう。例えば、マルチユーザのプログラムを明示的にサポートしたり、タイプタグのレベルでデータのオーナシップをサポートしたりするようなことが考えられる。

サーバベースのアプリケーションはまた、新しい Lisp で何をハックすべきかという質問に答えてくれる。Unix のスクリプト言語とするだけでも Lisp は良いものになるだろうが (これ以上悪くするほうが難しい)、既存の言語と競争するのに勝ちやすい領域というのはあると思うのだ。Tcl のモデルをフォローするのが良いんじゃないか。サーバベースのアプリケーションをサポートする完全なシステムを作り、Lisp をそれと一緒に提供するのだ。レキシカルクロージャは、UI が単なる Web ページの連なりである場合にサブルーチンの機能を提供できるし、S 式は HTML に綺麗にマップされるし、それを生成するのにマクロは最適だ。サーバベースのアプリケーションを書くのに良いツールが必要とされており、一方新しい Lisp も必要とされている。この両者を一緒にやればうまく行くんじゃないか。

12. 夢の言語

まとめの代わりに、ハッカーの夢の言語というのを考えてみよう。夢の言語は美しく、明快で、簡潔だ。素早く立ち上がる会話的なトップレベルを持つ。よくある問題を解くプログラムはほんのわずかなコードで

書ける。どんなプログラムでも、書かなくちゃいけないのはアプリケーションに特有な部分だけだ。他の全てはあなたのために既に作られている。その言語の構文は極端に短い。必要のない文字をタイプする必要はほとんどなく、シフトキーさえほとんど使わない。

高度な抽象化を使って最初のバージョンは極めて速くに書き上げられる。後に、最適化したくなったら、非常に良いプロファイラが提供されていて、どこに注目したらいいかを教えてくれる。必要とあらばインラインでバイトコードを書くことさえできて、内部のループを目もくらむほど速くすることができる。

学ぶためにたくさんの良い例が提供されていて、さらに言語は直観的なので例からほんの 1〜2 分で使い方を学ぶことができる。マニュアルを見る必要さえあまりない。マニュアルは薄く、注意すべき点や制限はほとんどない。

その言語は小さなコアと、コアと同じくらい慎重に設計された強力で直交性の高いライブラリからなる。ライブラリはお互いがうまく協調して動く。良いカメラのように、言語の全ての部分はぴったりとかみあって動くのだ。使わない方が良い機能とか、互換性のために残された機能なんてものは無い。オペレーティングシステムや他の言語で書かれたアプリケーションとコミュニケートするのも容易だ。

その言語はいくつもの層から構成されている。抽象化の高い層は抽象化の低い層から透過的に構成されており、必要とあらば抽象化の低い層を直接叩くこともできる。

絶対に必要である箇所以外は、何事もプログラマから隠されてはいない。その言語の抽象化機能は、プログラマにどうプログラムすべきかを教えるのではなく、プログラマの仕事を楽にするためだけに提供されている。実際、その言語はプログラマを言語の設計に参加するものとして扱い、プログラマが書いたものは可能な限り、言語組み込みのものと同様の地位を得る。