リチャード・M・ストールマン　スウェーデン王立工科大学講演

RMS Lecture at KTH (Kungliga Tekniska Hogskolan)

記録：Björn Remseth*　　翻訳：山形浩生†

30 October 1986、1998 年 8 月訳

概 要

1986 年 10 月 30 日に、リチャード・M・ストールマンがスウェーデンのストックホルム市にある王立工科大学で行った講演の講演録[1]。第一部では、1970 年代の MIT AI 研におけるハッカー文化、商業主義によるその破壊と、それに対抗してハッカー文化を再興させるべくフリーソフトの GNU プロジェクト開始をストールマンが決意するまでの過程が述べられている。第二部では、GNU Emacs をはじめとする GNU プロジェクトそのものについて、歴史的な解説とともに、内部での処理系に関する考え方の解説も含めて語られている。第三部では、ストールマンの情報著作権や所有権についての考え方をはじめ、GNU プロジェクトや FSF の背後にある思想が述べられる。情報やソフトは社会効用からみても共有されるべきであり、それを妨げる著作権やソフト隠匿はまちがっている、と主張されている。

[コメント：これは 1986 年 10 月 30 日に、リチャード・M・ストールマンが、学生会 “Datorforeningen Stacken” の招きでスウェーデンのストックホルム市にある王立工科大学で行った講演のテープ起こしに、ちょっと手を入れたものだ。だから変な始まりかたや、口語では自然だけれど字にするとへんてこな語法も入ってる。これを「もとの講演をゆがめることなく」文語にするにはどうすればいいか、よくわかんないのだ。[2]]

目 次

*原文最新版：http://www.gnu.org/philosophy/stallman-kth.html, 翻訳：http://www.post1.com/home/hiyori13/freeware/rmslecture.html

[1]訳注：原文は切れ目のない講演録であり、章分けとその命名および概要の追加は訳者の判断でやった。

[2]訳注：日本語にしちゃえば同じことだけど、講演としての性質を尊重して、ここではなるべくバリバリの口語文体を採用してる。文法的に不正確な日本語もたくさんあるけれど、原文の変なとこを残したせいで生じてる部分がほとんどだから、揚げ足をとらないよーに。

1 イントロ

みんながぼくに話してほしい話題は三つあるみたいだね。一つはまず、このハッカー集団に話すのにいちばんいいのは、昔の MITってのがどんなとこだったのかってことかな、と思った。人工知能研究所のどこがそんなに特別だったのか。でも、今日きてる人は月曜と火曜に会議にきたのとぜんぜんちがう人だから、GNU プロジェクトがどうなってるかとか、ソフトウェアや情報がなぜ所有できないかとかについて話すべきだ、とも言われた。ということはつまり全部で三つの話ってことで、このうち二つがそれぞれ一時間かかったんだから、みんなかなり長時間ここにいることになるわけだ。それで思ったんだけれど、全体を三つにわけて、みんな自分の興味がない部分は外に出られるようにして、それで各部分の終わりにきたら、終わりだって言うから、みんな外にでて、それでぼくはヤン・ライニングにお願いしてほかの人を呼んできてもらうようにするってのはどうだろう。(だれかが “Janne, han trenger ingen mike” (「Janne、この人マイクなんかいらないよ」) と言う) ヤン、ひとっ走りして人を呼んできてもらってもいい？

Jmr: マイクを探してるんですけど、なんかこの鍵のかかった箱の中にあるっていうんですよ。

Rms: うん、昔の AI 研でなら、でっかいハンマーを持ってきてそいつをぶち開けてただろうね。その壊れたドアを見て、みんなが必要とするものを鍵かけてしまいこもうとするようなヤツも少しは勉強になったろう。でも運のいいことに、ぼくは昔ブルガリア歌唱法を勉強したことがあって、だからマイクなしでも全然問題ないよ。

とにかく、話がどの部分か知らせるシステムって要る？　それともずっとすわって全部きいてたい？ (答：そうだそうだ)

2　GNU 前史：MIT AI 研

2.1　MIT AI 研のハッカー文化とその崩壊

ぼくがプログラミングをはじめたのは 1969 年のことで、ニューヨークの IBM 研究所でのことだった。そのあと、コンピュータ科学の学部がある学校があって、まあここはほかと似たりよったり。何をすべきか決める教授たちがいて、だれが何を使っていいか決める連中がいて。ほとんどの人には端末が不足してたんだけれど、教授たちはほとんどみんなオフィスの中に専用端末を持ってて、これは資源の無駄なんだけれど、連中の姿勢からすればまああありがちなことだよね。

MIT の人工知能研に遊びにいったら、そういうのとは爽快なくらいにちがった精神があったんだ。たとえば、そこでの端末はみんなのものだと思われてた。だから自分のオフィスに鍵をかけるような教授は、ドアがぶち破られるというつらい目にあうわけ。ある教授のオフィスのドアをぶち破るのに使った、でっかい鉄のかたまりが載った台車をホントにみせてもらったよ。その頃は端末はほんとに少なくて、システム全部にディスプレイ端末 5 つくらいかな、だからそのうち 1 台がしまい込まれてると、ものすごい損害だったわけ。

その後の数年、ぼくはその考え方に刺激を受けて、何度も天井をのりこえたり床下を這ったりして、みんなが使いたいマシンのある部屋の鍵を開けてきた。そして通常は、ドアに鍵をかけるような身勝手なまねはしなさんな、というメモをおいてった。ドアに鍵をかけるような人は、基本的には自分のことしか考えてないんだ。もちろんその人たちにもそれなりの理由はあるだろう。何か盗まれそうなものがあってそれに鍵をかけときたいとか。でも同じ部屋にある別のものがしまいこまれちゃうと、ほかの人に影響が出るってことは気にもしなかったんだ。これが起きる度に必ずいえることだけど、それで一回はぼくも指摘したんだけど、別の解決策はあるんだ。鍵をかけるかどうかってのは、かれらの勝手じゃないんだ。心配なものをしまっとく場所はほかにもあるし、デスクに鍵をかけることだってできる。でも要するに、みんなそんなことをわざわざ考えようとはしないんだよね。「この部屋はおれのだ、鍵をかけるのもおれの勝手、ほかの連中クソ食らえ」と思ってる。そしてこれぞまさに、持ってはいけない精神なんだってことは教え込む必要がある。

でも、この鍵をかけない精神は、孤立したものじゃなくて、生き様全体の一部だったんだ。AI 研のハッカーたちはいいプログラム、おもしろいプログラムを書くことにすごく熱心だった。そしてもっともっと作業したくてたまんなかったので、端末に鍵がかかってるなんて黙ってられなかったんだ。あるいはその他、役に立つ仕事をじゃましようとして人がやるいろんなこととかも。そのちがいってのは、高いモラルをもってて、自分が何をしようとしてるのか本当に気にかけてる人と、ただの仕事でやってる人との差だ。ただの仕事なら、雇い主が馬鹿で、こっちがずっと何もしないですわってなきゃなんなくたって、だれが気にするもんか。連中の時間だし、連中の金だし。でもそんなとこでは、大した仕事はできないし、そんなとこにいたっておもしろくもなんともない。

あと、AI 研になかったものといえば、ファイル保護。コンピュータにはセキュリティはまったくなかった。そしてこれはきわめて意識的に選んだ道だった。Incompatible Timesharing System (ITS) を書いたハッカーたちは、ファイル保護なんてものは身勝手なシステム管理者が、ほかのみ

んなに対して権力を行使するためのもんだって思ってた。ハッカーたちは、そんな権力を行使されるのはまっぴらだったから、その手の機能は実装しなかった。結果として、システムのどこかがこわれたら、いつも自分でなおせた。いらいらしながらじっとすわってなきゃならないなんてことは絶対許せないよね、こっちはどこがおかしいかずばりわかってて、それなのにだれかが、自分を信用してないからそれをさせないと決めてるなんて。あきらめて家に帰って、朝にだれかがきてシステムをなおしてくれるのを待つなんて、しなくていいじゃない。何をしなきゃいけないのか、自分にはその人の 10 倍もよくわかってるってのに。

あと、どんな作業をやるのかについても、教授だの上司だのには決めさせなかった。だってぼくたちの仕事はシステムの改善だったんだもん！　もちろん利用者とは話をした。だってそうしないと何が必要かわからないからね。でもそれがすんだら、どんな改善が実現可能かいちばんよく判断できるのはぼくたちだったんだ。そしてぼくちはいつも、システムをこう変えたらいいだろうな、とかほかのシステムで、こんないかしたアイデアを見たよ、とか、それを使えないものかな、とか話してた。だから結果として、そこにはなめらかに機能するアナーキーがあった。そしてそこでの自分の経験から、ぼくはそれこそ人が生きる最高のやりかただと確信してる。

残念ながら、そういう形の AI 研は破壊されちゃったんだ。何年ものあいだ、AI 研は MIT の別の研究所、コンピュータ科学研に破壊されるんじゃないか、というのがぼくたちのおそれていたことだった。そこの所長は一種の帝国建設者タイプで、自分の組織をでかくして MIT の中で昇進するためならなんでもやるような人間で、いつも AI 研を自分の研究所に吸収しようとしてたんだ。だれもそいつの流儀でなんか作業したくなかった。そいつは、人は指示に従うべきだとかなんとか、その手のことを信じてたからね。

でもその危険に対してはなんとか防衛できたのに、まったく予想してなかったものによってぼくたちは破壊されてしまった。それが商業主義だった。1980 年代初期になって、ハッカーたちがハッと気がつくと、自分たちのやっていることには商売上の関心がもたれていたわけ。民間企業で働いて金持ちになることが可能になったんだ。必要なのは、自分の仕事をほかの世界と共有するのをやめて、MIT AI 研をぶちこわせばいいだけ。そしてぼくは手を尽くしてそれを止めようとしたけど、でもみんなそれをやっちゃったんだ。

要するに、AI 研にいたぼく以外の有能なプログラマは、みんな雇われてやめちゃって、おかげでそれは一時的な変化じゃすまなくなってた。永続的な変化が生じた。ハッカー文化の連続性がとぎれちゃったんだよ。新人ハッカーはいつだって古参ハッカーに惹かれる。だっていちばんおもしろいコンピュータがあって、いちばんおもしろいことしてる人たちがいて、そして参加したら最高におもしろいスピリットもあったんだから。それがなくなったら、そこを新人に推薦するべき理由もなくなっちゃって、だから新人もこなくなった。ひらめきを受けるような人もないし、伝統を学べる人もいないし。ついでに、いいプログラミングを学ぶ人もいない。いるのは、使えるプログラムを書く方法なんか実は知らない教授や院生がうじゃうじゃいるだけだから、使えるプログラムの書き方なんか習えるわけがない。だからぼくの愛した MIT AI 研はもうなくなった。そして何年か、それをやった人たちと闘って連中に罰を与えようとしてから、ぼくはその精神をもった新しい

コミュニティづくりに精を出そうと決めたんだ。

でも、直面せざるを得なかった問題の一つが、独占ソフト（proprietary software）の問題だった。たとえばハッカーたちが去ってから研究所で起こったことの一つは、ぼくたちが開発したマシンやソフトがもうメンテされないってことだった。ソフトはもちろん動いたし、だれもそれを変えなければずっと動いてたけど、でもマシンはそうはいかない。マシンは壊れるし、だれもなおせるやつがいないから、やがて捨てられる。昔は、ええ確かに保守契約なんてものはあったけど、でもまあ悪い冗談みたいなもんだった。それって単に、AI 研のエキスパート・ハッカーたちが問題を解決してから、パーツを入手するための手段でしかなかった。だって修理担当者になおさせたら、何日かかるやらわかんなくて、そんなのやってられないよね、すぐ動くようにしてくれないと。だから、やりかたのわかってる人間がだまって行ってさっさとなおして、それでそいつらはどんな修理担当者より 10 倍も有能だったから、ずっとましな仕事をしたんだ。それでそうすると、こわれた基板とかが出るから、それをそこにおいといて、修理担当者には「こいつを持って帰って、新しいのをちょっともってきてくださいね」と言うわけ。

すごい昔の頃だと、ハッカーたちは Digital からきたマシンのほうも変更した。たとえば、PDP-10 のてっぺんにのせる呼び出し用ボックスもつくったんだよ。最近だと、ここ（ストックホルム）でもそういうことをする人はいると思うけど、当時はかなり異例なことだった。それにうーんと昔、1960 年代初めとかだと、みんなコンピュータを変更して、いろんな新しい命令を足したり、すっごい TSS 機能を足したりして、だから MIT の PDP-1 は、引退する頃には 60 年代はじめに Digital が届けたときの 2 倍くらいの命令を持ってたし、特別なハードウェア・スケジューラ補助機能や変なメモリマッピング機能なんかもあって、個別ハード装置を特定の TSS ジョブに割り当てたり、その他ぼくでもほとんど知らないようないろんなことになってた。それと確か、一種の拡張アドレッシングモードも組み込んで、インデックスレジスタ修飾とか間接修飾とか追加して、要するに軟弱なマシンだったのをそこそこ使えるものにまで仕立てたんだ。

VLSI の欠点の一つは、もうマシンに命令を追加したりできなくなっちゃったことだろうね。

PDP-1 にはまたすごくおもしろい特徴があって、おもしろいプログラムをほんの数命令で書けちゃうってこと。それ以降のどんなマシンよりも少ない命令数でね。確かたとえば有名なディスプレイのハック「マンチング・スクウェア」ってのがあって、四角がどんどんでかくなって、それがたくさんの小さい四角に分裂して、それがまた大きくなってそれがまた小さいのに分裂するんだけど。これって PDP-1 ではなんか 5 命令くらいで書かれてた。それとかいろんな美しいディスプレイプログラムが、ほんの数命令で書ける。

というわけで、これが AI 研の様子だった。でもハッカーたちの文化は、アナキズム以外にはどんなものだったのか？　PDP-1 の頃には、マシンは一度に 1 人のユーザしか使えなかった。少なくとも最初はね。何年かして TSS を書いて、それ用にたくさんハードを追加した。でも初めの頃は、時間枠を予約しなきゃならなかった。さてもちろん、公式プロジェクトの仕事をしてる教授や学生は、いつも昼間にやってくる。だから時間がたくさんほしい人たちは、競争の少ない夜を予約する。これでハッカーたちは夜に働く習慣ができた。TSS が入ってからも、ユーザが少なかったか

ら夜のほうが時間をとりやすい。CPU サイクルは夜のほうが使えたわけ。だから作業をいっぱいしたい人は、相変わらず夜にやってきた。でもその頃になると、それだけじゃなくなってて、それは自分がもう 1 人じゃなくて、ほかにも何人かハッカーたちがいたからで、だから社会的な現象になったのね。昼間にきたら、たぶんいるのは教授や学生とか、マシンをほんとは愛してない連中だけど、夜ならばそこにはハッカーがいる。だからハッカーたちは、自分の文化に加わるために夜にやってきた。そしてハッカーたちはほかの伝統も開始した。朝の 3 時にテイクアウトの中華料理を買ってくるとか。だからぼくは、チャイナタウンから戻ってくる車の中から見た日の出がいくつも記憶に残っている。日の出を見るってのは、すごく美しいもので、だって一日の中ですごく穏やかな時間だからね。寝る準備をするにはすばらしい時間帯だ。ちょうど空が白んできて、鳥が鳴き出す頃に家に歩いてくと、穏やかな満足感が実感としてあるんだ、その晩にやった仕事に対する静謐な気持ちってのが。

ほかにもぼくたちのはじめた伝統は、研究室に泊まること。ぼくがはじめてそこに行ったときから、研究室には少なくとも一つベッドがあった。そしてぼくは、ほかの人よりちょっとばかし研究室で暮らす期間が長かった。1 年か 2 年おきくらいに、あれやこれやでアパートがなくて、だから研究所で数ヶ月暮らしたりしたんだ。いつでもすごく快適で、しかも夏にはすごくすずしくていいんだ。でも、研究所で寝ちゃう人を見かけるのはぜんぜんしょっちゅうで、これもまた熱意のせいだよね。とにかく手をとめたくないからできるだけ長いこと起きててハッキングする。そして完全に疲れきったら、最寄りのやわらかい水平面によじのぼる。すごく形式張らない雰囲気。

でもハッカーたちがみんな研究所を離れたら、これは人口構成の変化を招いた。マシンをほんとは愛してない教授や学生どもは、昔と同じにうじゃうじゃいたから、連中がいまや強い勢力になったわけで、みんな怖がってた。システム管理をしてくれるハッカーがいなくなったので、連中はこう言ったんだ。「このままじゃ大変なことになる、商業ソフトを入れなきゃ」そして「そしたらメーカーがメンテもしてくれるよ」だって。やがて連中がまったくまちがってたことが証明されたけど、でも連中はそれをやったんだ。

2.2 ハッカー文化の再生をめざして

まさにその時に、新しい KL-10 システムがくるはずになってて、そこでの問題は、それで Incompatible Timesharing System を動かすか、Digital の Twenex システムを動かすかってことだった。ハッカーたちはたぶん ITS を使うのを支持したろうけど、それがいなくなってたから、学者タイプの連中は商業ソフトを使う選択をして、これはすぐに目に見えて影響が出てきた。なかにはそんなすぐには出てこなかったものもあったけど、でもこの問題を考えてみた人ならわかるように、遅かれ早かれ出てきたんだ。

一つには、そのソフトはずっと質が悪くて、わかりにくくて、だから本当に必要な変更をするのがむずかしかったってこと。もう一つの影響は、このソフトにはセキュリティがあって、おかげでみんながお互いに協力する機会はどうしても減ってきた。ITS の時代には、だれもがどんなファイ

ルでも見て変えられるってのはいいことだと考えられてた。それなりの理由があったからね。それで思い出したけど、だれかが Macsyma の使い方について助けてくれって言ってきたときに、おもしろいスキャンダルがもちあがったんだ。Macsymaってのは MIT で開発したシンボリック数学プログラム。その人は、Macsyma の作業をしてるある人に、助けてくれってメールを出したら、数時間たって別の人から返事がきた。その人はびっくりしちゃって、「だれそれはあなたのメールを読んでるみたいですよ、ひょっとしてメールのファイルの保護がちゃんとしてないんじゃないですか？」だって。「当然でしょう、ぼくたちのシステムでは保護されてるファイルなんかない。なんかいけませんか？　あなたはすぐに答がもらえたし、何が不満なの？　ぼくたちはお互いのメールを読む。あたりまえじゃん、そうやってあなたみたいな人を見つけて助けてあげるんだから」まったく、自分が得してるのにわかんない人ってのもいるんだからなぁ。

でももちろん、Twenex にはセキュリティがあっただけでなく、それをデフォルトでオンにするし、さらにセキュリティが使われてるという前提で設計されてる。だから、簡単にできちゃってすごいダメージを与えるようなことってのがたくさんあって、うっかりそういうことをしないようにしてくれる唯一の手段がセキュリティなんだ。ITS では、そういうことをうっかりやっちゃわないようにする、ほかのいろんな方法が考案されてた。でも Twenex にはそんなものはない。厳格なセキュリティが機能してて、そういうことをする権限を持つのはボスたちだけだ、と想定してたからだ。だから、そういううっかりミスを難しくするメカニズムはほかにぜんぜんつけてなかった。その結果として、単に Twenex をもってきてセキュリティをオフにするとほしいものが手に入るわけじゃなくなってて、そしてその他のメカニズムを入れ込むハッカーたちもいなかったから、みんなセキュリティを使うしかなくなってた。そしてマシンがやってきて 6ヶ月ほどしたあ、連中はクーデターをはじめた。つまり、最初は AI 研で働いてる人はみんな、全セキュリティ機能をオーバーライドする全権を与えるホイールビットをオンにしてもらえるもんだ、という想定をしてた。でもある日のある午後にやってきてみると、ほとんど全員のホイールビットがオフにされてた。

こんなのがわかったとき、ぼくはそれを打倒した。最初は、たまたまエリート層に属してるある人のパスワードを知ってたので、それを使ってみんなのビットを戻せた。二回目には、そいつはパスワードを変えてて、つまり所属意識も変わってて、いまやそいつはもう貴族社会の一員だった。そこでぼくはマシンを停止させて、TSS でない DDT を使ってつつきまわさなきゃなんなかった。モニタの中でつつきまわして、やがて OS をロードさせてパッチがあてられるようにするにはどうしたらいいかつきとめて、それでパスワードのチェックをオフにして、それからいろんな人のホイールビットを戻して、システムメッセージをポストした。説明しとくとこのマシンの名前は OZ で、だからこんなシステムメッセージをポストした。「またもや権力を奪取せんとする試みが行われた。いまのところ、貴族階級の勢力は打倒された――ラジオ・フリー・OZ」あとで知ったんだけど、「ラジオ・フリー・OZ」というのはファイアサイン劇場で使ってるものの一つなんだね。その頃は知らなかった。

でも状況はだんだんと、どんどん悪いほうに向かってったね。要はそのシステムのつくられかたのせいで、それがみんなもっともっとセキュリティを要求するような性質もものだったってこと。

しまいにやがて、ぼくはそのマシンを使うのをやめなきゃなんなくなってきた。MIT AI 研にパスワードが初めて登場して以来、ぼくは自分の信念のために立ち上がろう、パスワードなんかあってはならないという信念にしたがおう、絶対にできるだけわかりやすいパスワードにして、それをみんなにばらすようにしようと決心したんだ。ぼくはコンピュータにセキュリティをつけるのはいいととは思わないから、セキュリティ支配が続くのを助けるようなこともすべきじゃない。「空のパスワード」を使えるシステムでは使うし、それが許されないシステムとか、それだとほかの場所からは一切ログインできないとか、その手のシステムとかだと、ぼくはログイン名をパスワードにする。これ以上はないってくらい自明でしょ。そしてだれかが、そんなことだと他人がきみになってログインできちゃうじゃないか、と指摘すると、ぼくはこう言う。「うん、まさにそれがねらい。だれかがこのマシンのデータを必要とするかもしれない。そのときに、その人がセキュリティのおかげでバカを見たりすることが絶対ないようにしときたいのよ」

それともう一つやるのが、自分のディレクトリやファイルの保護は全部はずす。だって役に立つソフトをそこに持ってることがよくあったし、もしバグがあったら、みんながなおせるようにしときたいから。

でもこのマシンはまた、「観光」と呼ばれる現象をサポートするようにはできていなかった。さて、「観光」ってのは AI 研ではすごく古い伝統で、ほかの形のアナーキーといっしょに続いてきてて、これはつまり、部外者にも自由にマシンを使わせるってこと。さて昔々、だれでもマシンにやってきて、勝手な名前でログインできた頃には、これはもう自動的にそうなってた。もし MIT に遊びにきてたら、ログインして作業ができる。後にこれをちょっとは形式化した。これは特に ARPAnet がはじまって、みんな全国からうちのマシンにつなぐようになったときにそうで、みんなが認める伝統としてってこと。さて、ぼくたちが期待してたのは、そういう部外者がプログラミングを実際に学んで、OS を変えはじめることだった。この話をほかのどこのシステム管理者にしても、みんな飛び上がるよ。どんな部外者でもマシンが使えるなんて匂わせようものなら、その人は言うだろう。「でもそいつがシステムプログラムを変えちゃったらどうすんの？」でもぼくたちにとっては、部外者がシステムプログラムを変えはじめるってことは、それはつまりその人がコミュニティに貢献するメンバーになろうと本気で興味を示してるってことなんだ。ぼくたちはいつも、そうするよう奨励してる。最初はもちろん、新しいシステムユーティリティを書くところから、それも小さいヤツね、そしてそいつが何をやったか監督して訂正するけど、でもそしたらそれから既存の大きなユーティリティに機能を追加するほうに向かう。そしてそういうプログラムはもう 10 年とか 15 年とか存在していて、一部一部ごとに、職人が次々と新しい機能を足してったものなんだ。

フランスの都市みたいなもんだと言ってもいいかな。ものすごく古い建物に、数百年後に増築されて、それがずっといままで続いてるような。コンピュータの世界だと、1965 年に書きはじめられたソフトってのがそれにあたる。だからぼくたちはいつも、観光客がシステム管理者になってほしいと思ってて、そしたら雇われるかもしれない。もちろんシステムプログラムの作業を始めて、まともな仕事ができることを証明してくれればだけど。

でも ITS のマシンには、これが手に負えなくなるのを防ぐための機能がほかにいくつかあった。

その一つが「スパイ」機能で、だれが何をしてるか、だれでも観察できる機能だった。そしてもちろん、観光客はスパイするのが大好きで、みんなそれがいかしたことだと思って、つまりちょっとイケナイ感じだけれどでも結果として、どっかの観光客が何か面倒を起こすようなまねをしても、必ずだれかがそれを見てるわけ。それでやがてそいつの友だちはすごく怒りだす。観光が続いてくれるためには、観光客が責任ある行動をするかどうかにかかってるのを知ってるから。それでたいがいは、そいつがだれだか知ってるやつがいて、それでそういうことをするなと説明してもらえるんだ。それができなかったら、どうするかっていうと、特定の場所からのアクセスをしばらく完全に切る。そしてそれを戻す頃には、そいつはどっかよそへ行って、ぼくたちのことは忘れちゃう。だからこれが何年も何年も何年も続いたんだ。

でも Twenex システムは、この手のこと用には設計されてなくて、やがて連中はみんなにパスワードを知られてるぼくを許してくれなくなった。観光客がいつもぼくの名前で同時に二人も三人もログインしてたわけ。だから連中はぼくのアカウントを消すようになった。そしてその頃には、ぼくはどうせ別のマシンで作業してるほうが多くて、だからやがてあきらめて、もうそっちのスイッチは入れなくなった。それでそれはおしまい。もうあの機械にはずいぶんログインしてないな、自分のアカウントではもう……（この時点で、RMS はすさまじい拍手にさえぎられる）年も。

でも連中が最初にこの Twenex システムを手に入れたら、まずいくつか変更をしたがった。セキュリティの仕組みを変えたがった。それとそのマシンを ARPAnet と MIT-CHAOS ネットワークに両方につなぎたがったんだけど、でも結局はそれができなくて、そういう変更をするだけの能力をもった人が見つけられなかったんだ。それをするだけの才能がもうなくなってたし、そもそもシステムの変更がむずかしかった。このシステムはずっとわかりにくくて、それはダメな書かれかたをしてたからで、もちろん Digital はそんなことしてくれなくて、だから商業システムなら基本的にはシステム管理を会社がやってくれるという考え方は、結局まちがってることが証明されたわけ。システムハッカーに対するニーズは前と変わらなかったのに、でもシステムハッカーをおびきよせる手段がもうなくなってた。そして最近の MIT には、Twenex 上でハッキングしたがる人より ITS でハッキングしたがる人のほうが多いんだ。

そしてそうなる最終的な理由ってのは、Twenex は共有できないってことなんだ。Twenex は独占プログラムで、ソースコードを手に入れるには、ある嫌らしい方法でそれを秘密にしとかなきゃならなくて、これが印象を悪くしてる。その人が無関心でなければ（そしてコンピュータ業界にはそういう人もいて、自分たちだけが楽しければなんでもやって、ほかの人たちと協力してるかなんて一瞬たりとも考えないような人たちだけれど、でもそんなソフトの作業をするのがどんなに悲しいことかわかんないなんて、相当無関心じゃないとアレで、これがさらにマイナス要因だよね）。そしてさらにだめ押しでくるのが、毎年かそこら、新しいリリースってのが出てきて、それが 50,000 行くらいの追加のコードだらけで、それが全部サルが書いたみたいな代物。それは連中が「サル 100 万匹にタイプさせれば、いずれなんか使えるものが出てくる」式システム開発にしたがってるから。

こういう独占システムで何が起きてるか見たとき、ぼくにははっきりわかった。かつての AI 研

のスピリットをぼくらが持つ唯一の方法は、フリーの OS をつくることだってのが。フリーソフトだけでつくったシステムができて、それがみんなで共有できるようになることだって。そうすればみんなに、その改善に参加しようと呼びかけられる。そしてそこから GNU プロジェクトが出てきたんだ。というわけで、この話の第二部にやってきたってことだね。

3 GNU プロジェクト

3.1 はじめの一歩

だいたい 3 年半くらい前に、フリーソフトシステム開発に手をつけるべきだってのは自分ではっきりしてきた。開発すべきシステムとしては二種類の可能性が LISP マシンみたいなシステムで、ちょうど開発されたばかりの MIT LISP システムとまったく同じだけれど、でもフリーで、しかも特別な LISP マシンじゃなくて、ふつうのハードで動くヤツ。そしてもう一つの可能性が、もっと伝統的な OS をつくるってことで、その時にはっきりしてたのは、もし伝統的な OS をつくるなら、それは UNIX と互換性をもたせるべきだってこと。そうすればいろんなとこの人たちが乗り換えやすいから。しばらくして、後者をやるべきだと決めたんだけど、その理由は、ほんとの LISP マシンみたいなものを普通のハードではできないってのが見えてきたからだったのね。LISP マシンは、実行速度を確保して、同時にランタイムで堅牢なエラー探知をするために、特殊なハードと特別な書き込み可能なマイクロコードを使ってるんだ。エラーって、特にデータタイプのエラー。ふつうのシステムで LISP システムを十分にはやく走らせるには、いろいろ仮定をおくようにしなきゃなんない。ある引き数が正しいタイプだって仮定して、で、そうでなければシステムはあっさりクラッシュ。

もちろん外的なチェックを入れることはできるし、やりたきゃ堅牢なプログラムも書けるけど、結局のところは、そのチェックを入れなければ、関数にまちがったタイプの引き数を喰わせたら、メモリアドレッシングのエラーとかが起きちゃうってことだ。

だから結果としては、LISP システムの下で何かが走ってて、そういうエラーを拾ってくれなきゃなんない。そしてユーザがシステムを動かし続けて、起きたことをデバッグできるようにしないと。最後に、もし下の OS をつくるんなら、それはいい OS にしようとも決心した—つまりこれは、OS と LISP か、あるいは OS だけか、という選択だった。だから、まずは OS をやるべきだろう、そしてそれは UNIX 互換にすべきだ。最後にこのシステムの名前として英語でいちばんおもしろい名前を使えるんだと気がついたら、ぼくの選ぶべき道ははっきりした。そのことばはもちろん GNU で、これは「Gnu's Not Unix」の略。この再帰的な略称は、MIT 周辺のハッカー社会ではすごく古い伝統になってる。ぼくの知る限り、これは TINT というエディタではじまったはず。これは「Tint Is Not Teco」の略で、それはやがて「SINE Is Not Emacs」の略で SINE とか、「Fine Is Not Emacs」で FINE とか、「Eine Is Not Emacs」で EINE とか、「Zwei Was Eine Initially（Zwei はもともと Eine でした）」の ZWEI とか[3]、そしていまはそれが GNU まできた

[3]訳注：ドイツ語では Eine は 1 で、Zwei が 2 だというのはもちろんご承知のこととは思いますが……

わけ。

だいたい 2 年半ほど前に実際に GNU の仕事をはじめてから、もう作業の半分くらいは終えたと言っていいかな。このプロジェクトにとりかかろうとしてたとき、まずはすでにフリーで出回ってるものは何があるかを探すとこからはじめた。そこで見つけたのが、おもしろいポータブルなコンパイラシステムで、これは「The Free University Compiler Kit」という名前。こんな名前なら、もらえるのかな、と思った。だから開発した人にメールを送って、GNU プロジェクトにそれをくれないか、ときいてみたら、その人は「いや、free は大学にかかることばで、そこで開発してるソフトはフリーじゃないんです」と言って、でもそれから言うには、かれも UNIX 互換システムがほしくて、それ用のカーネルみたいなものも書いてみたいから、そしたらぼくがそれ用にユーティリティを書いて、そしたらその両方ともかれの独占コンパイラといっしょに配布できて、そしたらみんながそのコンパイラを買いたがるようになるだろう、だって。それでぼくは、ふざけんじゃないよと思って、だからそいつに、ぼくの最初のプロジェクトはコンパイラの開発だ、と言ってやった。

その頃は、コンパイラの最適化について実は大して知らなかった。一度もいじったことがなかったから。でも、その頃フリーだと言われたコンパイラを手に入れられた。それが PASTELっていうコンパイラで、作者たちに言わせると「色の薄い PASCAL」という意味なんだって。

Pastel はすごく複雑な言語で、パラメータ化されたタイプや明示タイプパラメータ（explicit type parameter）とか、いろいろ複雑なものを含んでた。コンパイラ自身ももちろんこの言語で書かれてて、こういう機能の利用を最適化するのに、いろいろ複雑な機能を持っていたんだ。たとえばこの言語では “string” タイプはパラメータ化されていた。特定の長さのストリングがほしければ “string(n)” と書ける。単に “string” と書けば、パラメータは文脈から決まってくる。で、ストリングってすごく大事で、それを使ういろんな構造体を高速に走らせるためにも必要で、ということはつまり、各種のことを検出するようないろんな機能が必要ってことだよね、たとえば、宣言されたストリング長が引数で、それが関数の中ではずっと定数であることを検出するとか、値を保存してそれが生み出すコードを最適化するとか、いろいろややこしいこと。でもこのコンパイラの中では、少なくともレジスタ自動割り当てをどうするかとか、そういうことは読みとれたし、いろいろちがったマシンの扱い方の考え方もわかった。

さてこのコンパイラはすでに PASTEL をコンパイルできるので、必要なのは C のフロントエンドをつけることで、これをやって、それから 68000 用のバックエンドをつけることだった。68000 がぼくの最初のターゲットマシンになるものと思ってたから。でもここで深刻な問題にぶちあたった。PASTEL 言語は何かを使う前に宣言しなくていいようになってたので、だから宣言と使用がどんな順序でもよくて、つまりは、Pascal の “forward” 宣言は使いものにならなくて、おかげでプログラムをまるごと読み込んで、それをコアにいれといて、一挙に処理するしかなかった。結果としてコンパイラ内部で使われる中間記憶、つまり必要メモリ量は、ファイルの大きさに比例して大きくなるわけ。そしてここにはスタック領域も含まれてて、ものすごいスタック領域が必要で、だから結果としてわかったこと：ぼくの手ものと 68000 システムではこのコンパイラは走らない。それはそのシステムが最悪の UNIX で、スタックに 16K ワードとかの上限があって、これってマ

シン自体にはメモリが 6 メガバイトもあるのにだよ、スタックは 16Kw かそこらしかとれないっての。そしてもちろん、一時的な変数値がかちあってないかとか、あるいは同時に生きてるのがどれかとかを見るのに、コンフリクト行列を生成するんだけど、これはビット単位で 4 次元行列とかが必要で、大きな関数になるとそれが何百バイトとか何千バイトにもなる。だからコンパイラは 10 パスくらいかそこらあったんだけど、その最初のパスはなんとかデバッグして、それをそのマシン用にクロスコンパイルして、そしてふたを開けてみたら、二番目のやつはそもそも走らない。

3.2 GNU Emacs

こういう問題をどうしようか考えて、これをなおそうとするのか、それともまったく新しいコンパイラを書こうか考えているうちに、なんやかんやで GNU Emacs の作業にとりかかった。GNU Emacs は GNU システムの配布分の主要部分。拡張できるテキストエディタで、ぼくが 10 年前に開発したオリジナルの emacs とかなり似てるけど、こいつは拡張用言語として本物の LISP を使うんだ。エディタ自身は C で実装されてて、LISP インタープリタも同じく C で実装されてるから、LISP インタープリタは完全に可搬性があって、エディタの外部に LISP システムを持たなくていい。エディタ自身が自前の LISP システムを持ってて、すべての編集コマンドは LISP で書いてあるから、それをお手本として見て自前の編集コマンドを書いたり、何からはじめるかとか、そういうのを変えて、自分が本当にほしい編集コマンドに変えられるんだよ。

その年の夏、いまから 2 年ほど前、ぼくの友だちがゴスリング Emacs の開発初期に手伝ったので、ゴスリングからかれのバージョンのゴスリング Emacs を配布していいよ、という許諾をメールで受け取ったと話してくれた。ゴスリングはもともと Emacs を書き上げて、それをフリーで配布して、たくさんの人が開発に協力して、それはゴスリング自身がマニュアルの中で書いたような、ぼくがもとの Emacs を開始したのと同じ精神にしたがうというかれ自身のせりふに基づいた期待があったからなんだ。そしたらゴスリングはそいつに著作権をつけて、みんなにそれを再配布しないと約束させて、あげくにそれをソフトハウスに売って、みんなを後ろから刺して裏切ったんだ。その後ゴスリングと個人的にやりとりしたけど、この歴史を見て予想されるのと寸分の狂いもないくらい、臆病でふざけたやつだったよ。

まあとにかく、その友たちがこのプログラムをくれて、それでぼくとしてはトップレベルの編集コマンドを変えて、ぼくの慣れ親しんでるオリジナルの Emacs と互換性を持たせたかった。それと数値引き数の組み合わせとかいろいろ扱えるようにして、ぼくのほしいような機能を全部扱えると期待できるようにするとか。でもしばらくやってるうちに、このエディタの拡張用言語、MOCKLISPっていうんだけど、それがこの作業用には力不足だってのがわかってきた。計画してることをやるためには、こいつをすぐにでもすげかえる必要があるってわかった。前にもいつかは MOCKLISP を本物の LISP に置き換えようと思ってたんだけど、でも真っ先にそれをやんなきゃっていうのがわかってきた。それで、MOCKLISP がなぜ「MOCK（にせ）」ってついてるかというと、そこに構造体データタイプがないからなの。LISP リストもない。配列もぜんぜんない。LISP

シンボルもない。これは名前つきオブジェクトね。MOCKLISP ではある名前に大してオブジェクトは一つ敷かなくて、だから名前をタイプすると、いつも同じオブジェクトしか戻ってこないの。こいつのおかげでいろんなプログラムを書くのがえっらくやっかいで、ホントはそんなふうに使うんじゃないようなややこしいストリング操作で、あれやこれやを処理しなきゃなんなかったわけ。

それで LISP インタープリタを書いて、MOCKLISP をすげかえて、その過程で、エディタの内部データ構造を書き直さなきゃならないこともわかった。ぼくはそういうのが LISP オブジェクトになっててほしかったから。LISP とエディタのインターフェースがクリーンであってほしくて、それはつまり、エディタのバッファやサブプロセスやウィンドウやバッファ位置みたいなオブジェクトが、みんな LISP オブジェクトでなきゃだめだってこと。そうじゃないとそれに作用するエディタのプリミティブは LISP データの LISP 関数としてコールできるようにならない。ということは、こういうオブジェクトすべてのデータ形式を設計しなおして、それに作用する関数もみんな書き直して、結果として 6ヶ月後には、ぼくはエディタのほとんどすべてを書き直してたってわけ。

加えて、MOCKLISP でなんか書くのはすごく難しいので、MOCKLISP で書かれたものはみんなきたなくて、だから本物の LISP の力を使えるようにそれを書きなおせば、それをみんなもっと強力でもっと単純でもっと高速にできたんだ。だからそれをやって、結果としてぼくがそれを配布しはじめときには、受け取ったものでそのまま使われてる部分はほとんどなかった。

この時点で、ゴスリングがプログラムを売ったと思ってる相手の会社が、ぼくの友だちがそれを配布する権利にケチをつけはじめて、メールはバックアップのテープに入ってて見つからなかった。そしてゴスリングは、そんな許可は与えてないと否定する。そこでおかしなことが起きた。かれはこの会社と交渉してたんだけど、この会社がいちばん気にしてるのは、自分たちが配布してるのと似たようなものが出回っちゃ困るってことだったようなの。かれもまだ配布してたし、かれの職場—Megatest 社ね—も、かれがぼくにくれたのと同じものを配布してて、それはかれの変更の入ったゴスリング Emacs で、だからかれは連中とそれを配布するのをやめるという合意をして、GNU Emacs を使うのに切り替えて、それでそしたら連中は、かれが実はやっぱり許諾を得てたんだってことを認めて、そうすればおそらくみんな満足ってことになるはずだった。そしてこの会社はぼくに相談をもちかけてて、GNU Emacs を配布させろ、もちろんそれはフリーだけど、でもいろんなサポートや補助を売りたいから、その作業の手伝いにぼくを雇いたいって言う。だからその後、連中の気が変わってその契約にサインするのを拒否して、ネットワークに、ストールマンはプログラムを配布する権利はないというメッセージをポストしたってのは、ちょっと変なんだよね。別に連中は何かするとか言ったわけじゃなくて、単にいずれいつの日か何か手をうたないかどうかははっきりしないとか言っただけ。そしてそれだけでみんなびびっちゃって、もうだれも使わなくなって、これは悲しいことだ。

（ときどき、一生かけてやるのに一番いい仕事ってのは、どっかで商売上の機密になってる独占ソフトのでかい山をみつけて、それを街角で配って歩いて、もう機密でもなんでもなくしてしまうことじゃないかと思って、みんなの手に新しいフリーソフトをわたしたいならそのほうが自分で新しいソフトなんかを書くよりも、ぼくとしてはずっと効率のいいやりかたじゃないかと思うんだけ

れど、みんなそれを受け取るのさえ怖がるほど臆病なんだもんな）

というわけで、残り全部を自分で書き直すしかなくて、だからそれをやって、それに一週間半くらいかかった。というわけで、やつらは大勝利をおさめてさぞ満足だろうよ。そしてぼくは、それ以降はどんな形であってもやつらとは協力なんかするもんか。

3.3 GDB デバッガ

で、GNU Emacs がそこそこ安定してから、というのはなんだかんだで 1 年半ほどかかったんだけど、それからシステムのほかの部分に戻った。GDBっていうデバッガを開発して、これは C コード用のシンボリック・デバッガで、最近配布に入れるようにした。このデバッガは、かなりの部分が DBX の精神にのっとったもので、これはバークレー UNIX についてくるデバッガね。コマンドは、何をしたいかを示すことばと、それに続く引き数で構成される。このデバッガでは、コマンドはみんな短縮形が使えて、よく使うコマンドは 1 文字の短縮形になってるんだけど、独自の短縮形も好きなように使える。充実した HELP 機能もあって、HELP のあとになんでもいいけどコマンドやサブコマンドまでタイプすると、そのコマンドの使い方が詳しく説明される。もちろん C の表現をタイプすれば、どれでもその値を返してくれる。

ほかにも、シンボリック C デバッガでは珍しいことができたりする。たとえば、どのメモリアドレスにあるどの C データ型も参照できるんだよ。値を調べたり、値を入れたりできる。だからたとえばあるアドレスのワードに浮動小数点の値を入れたければ、「これこれのアドレスにある FLOAT 型か DOUBLE 型のオブジェクトをよこせ」といって、それに割り当てればいい。もう一つできるのが、これまで調べてみた値を全部調べられんの。調べた値は全部「値ヒストリー」にのっかる。このヒストリーのどの要素でも、その番号で参照できるし、ただのドル記号（$）を使えば最後の要素を簡単に呼び出せる。こうすると、リスト構造をトレースするのがすごく楽になる。別の構造体を指すポインタを含むような C 構造体があったとするでしょ、そしたらたとえば `PRINT *$.next` みたいなことができて、これはつまり「さっきみせてくれたものの次のフィールドをとってきて、それが指してる構造体を表示しろ」ってことだ。そしてこのコマンドは繰り返せて、そのたびごとにリストの次の構造体が見られる。でも、ぼくがこれまで見た C デバッガだと、毎回もっと長いコマンドをタイプしなきゃなんないんだよね。そしてこの機能と、単に CR を押したら直前のコマンドを繰り返す、という機能と組み合わせると、これはすごく便利。リストの中で自分が見たい要素について、どんどん CR を押してけばいい。

あとデバッガの中で外部設定できる変数もある。いくらでも。ドル記号のあとに名前をつければ、それで変数。こういう変数値をどんな C データ型にでもアサインして、あとで検討できる。これがなんの役にたつかというと、たとえば：もし調べたい値がなんかあったとして、それをたくさん参照するのがわかってたら、ヒストリーの中でその値を覚えとくより、名前をつけちゃったほうがいいかもしれない。あるいは、条件ブレークをセットするときにも使えるかも。条件つきブレークは、シンボリックデバッガにはよくある機能で、「プログラムのここまできたら止まれ、ただし

この条件式が真の場合だけね」と言うわけ。デバッガの変数は、プログラム内の変数と、デバッガ変数内に保存した変数値とをくらべさせてくれる。これはほかにも、数えるのに使える。だって値の割付は要するに C の式でしょ、だから$hoge を 5 増やすには$hoge+=5 でもいいし、あるいは単に$hoge++でやってもいい。これを条件付きブレークでもできるから、このブレークポイントに 10 回目にきたらブレークとかするのにはお手軽でしょ$hoge--==0 をするんだよね。みんな、ついてきてる？　hoge を減らしてって、それがゼロになってたらブレークしろってこと。そしてそれから $hoge を、スキップしたい回数にセットして、それで行ける。これを配列の中の要素を見るのにも使える。たとえばポインタの配列があって、そしたらこんなことする：

```
PRINT X[$hoge++]
```

でもその前にまずこれ：

```
SET $hoge=0
```

オッケー、で、こうしたら（と “Print” 式をさす）、X のゼロ番目の要素が出てくる。で、もう一回やったらそれは一番目の要素で、それでこれがもし構造体へのポインタなら、たぶんここ（PRINT 式の X の前）にアステリスク（*）を入れて、そしたら毎回これはこの配列の要素がさしてる次の構造体をだしてくる。そしてもちろん、改行をおすだけでこれを繰り返せる。これ一つだけ繰り返すんじゃ不十分なら、ユーザ定義コマンドをつくればいい。“Define むにゃむにゃ” とやって、それからコマンドを何行か入れて、“end” とやる。これでもう “むにゃむにゃ” コマンドが定義されて、それが入れた行を実行する。で、こういう定義をコマンドファイルに入れておくとすごく便利なの。ディレクトリごとにコマンドファイルを持って、そこを作業ディレクトリにしてデバッガを起動すると、それが勝手にロードされるようにしておける。だから各プログラムごとに、ユーザ定義コマンドをいろいろ定義しておいて、便利な形でそのプログラムのデータ構造にアクセスできるようにしとける。そういうユーザ定義コマンド用にドキュメンテーションだってつくれるんだよ。そうするともとからのコマンドとまったく同じように、“help” 機能で処理されるんだ。

このデバッガでもう一つ珍しいのが、スタックからフレームを捨てられるってこと。デバッグしてるプログラムで何が起きてるか見られるだけじゃなくて、それを好き放題変えられるってのが大事だと思うんだ。だから問題を一つ見つけて何がダメかわかったら、そのコードが正しかったかのようにあちこちなおしてから、コンパイルしなおさなくても次のバグを見つけにいけるわけだよ。ということは、プログラムのデータ領域を思い通りに直せるだけじゃなくて、コントロールのフローを好きに変えられなきゃならない。このデバッガでは、こんなふうにしてコントロールのフローをすごく直接的に変えられる：

```
SET $PC=<なんか数字>
```

こうやってプログラムカウンタを変えるの。それとスタックポインタもセットできるし、あるいはこういうふうにでも：

```
SET $SP+=<なんとか>
```

スタックポインタをちょっと増やしたいなと思うでしょ。でもついでに、プログラムのどこか途中からはじめたいとして、だからプログラムカウンタをそのソース行にあわせるよね。でもそれで、その関数を呼んだのがまちがいで、その関数呼びたくなかった！　としたら？　たとえばその関数がどうしようもなくいかれてて、実はそこから戻ってきて、その関数のやるべきだったことを手でやりたいとしたら？　それには “RETURN” 命令を使う。スタックフレームを選んで “RETURN” とやると、そのスタックフレームや、その中にあるやつ全部が捨てられて、その関数からすぐに戻ってきたような感じになって、しかもそこから返ってくるはずの値も指定できる。実行は続かないよ。返ってきたようなふりをして、そこでまたプログラムを止めるから、ほかのものもどんどん変えられる。

このくらいいろいろまとまってると、プログラムの中で何が起きてるか、かなり細かくコントロールがきくわけよ。

これに加えて、もう一つちょっとおもしろいこと。C にはストリング定数があって、デバッガ内で計算してる数式にストリング定数を使ったらどうなるか？　デバッグしてるプログラム内にストリングをつくらなきゃなんない。だからつくってくれる。デバッグ中のプログラム内で MALLOC へのコールをやってくれて、MALLOC を走らせて、それからコントロールを取り戻す。こうやって、知らないうちにストリング定数を置く場所を見つけてくれるの。

いずれこのデバッガが本物の GNU システム上で動くようになるときには、その下で動いてる全プロセスの内部状態を調べられるような機能をつけるつもり。たとえばメモリマップの状態を見ようとかね、どのページが存在して、どれが読めて、どれが書き込めるか見て、あと下部プログラムの終了状態（terminal status）を見るとか。もうかなりコマンドはあるんだ。このデバッガは、UNIX のデバッガとはちがって、終了状態をデバッガやデバッグ中のプログラムと完全に切り離してる。だからそのままのモード（raw mode）で動くプログラムも見られるし、割り込み駆動入力をするプログラムでも動くし、それにデバッグ中のプログラムが実際に使ってるものの終了状態についても調べてくれるコマンドがある。一般にデバッガというのは、下部プロセスで起こってることはすべてわかるようにすべきだと思うな。

3.4　gcc コンパイラ

GNU システムのうちすでに存在する主要部分は二つある。一つは新しい C コンパイラ[4]で、もう一つは TRIX カーネルだ。

新しい C コンパイラは、この春から今年いっぱい書いてたもの。ついに PASTEL は捨てようと決めたんだ。この C コンパイラは PASTEL からのアイデアをもらってて、アリゾナ大学ポータブル・オプチマイザからのアイデアももらってる。連中のおもしろいアイデアは、いろんな種類のマシンを単純な命令の生成で扱うことで、ターゲットのマシンが許せばそれをいくつか組み合わせて

[4]訳注：もちろんこの時点ではまだ gcc という名前はなかった。

複雑な命令にするんだ。これを総合的にやるために、命令が数学式の記法で書かれてる。たとえばADD 命令はこんな感じで書かれる：

```
r[3]=r[2]+4
```

これはコンパイラ内部での命令の表現形。2 番レジスタの中身をとって、それに 4 を足して 3 番レジスタに入れろという命令。こういうふうにすれば、あらゆるマシン用にあらゆる命令を表現できる。それでかれらは実際に、すべての命令をこの形式で表現して、それを組み合わせる段階になったら、ある表現式をべつので置き換えて、もっと複雑な式をつくって組み合わせて命令にするわけ。

ときどき、最初の命令の結果がそれ以上使われるかどうかに応じて、割りつけ演算子を二つ持つような組み合わせ命令をつくんなきゃならない。この値（と何かを指さす）用に一つと、こっちの値（と何かを指さす）用に一つ、こっちは二番目の命令からきた値と差し替えるのね。でもこの値を一回しか使わないなら、差し替えたあとは捨てちゃえる。もうそれで計算する必要はないんだから。だから。途中で入ってくる命令とかがこういう値を変えたりしないかとか、そういうのをちゃんとチェックして、差し替えをきちんとやるのはなかなかややこしい。自動インクリメントや自動デクリメント・アドレッシングみたいなものをサポートするなら（ぼくはしてるけど）、値を保存するんじゃないような状況についてチェックするのに、いろいろチェックもしなきゃならない。

でもこういうの全部チェックしたら、差し替えた組み合わせ式をパターン・マッチャーに通して、それが選んだターゲットマシンで有効な命令を全部認識してくれる。それで認識されたら、その二つの命令を組み合わせ命令と置き換えて、そうでなきゃそのままにする。そして連中の技法ってのは、データフローで関連づけた命令 2 つ 3 つをこうやって組み合わせることなんだ。

アリゾナコンパイラでは、いろんなものをこういう文字列で表現してて、だから連中のコンパイラはとてつもなく遅い。最初はこのコンパイラをそのまま使ってちょっと変えるだけにしようかと思ったけれど、でもぼくの求めるスピードを出すには完全に書き換えるしかないのは明らかだったので、こういう式すべてにリスト構造の記法を使うように書き直した。たとえばこういうの：

```
(set (reg 2)      (+ (reg 2)    (int 4)))
```

なんか LISPっぽいけど、でもこいつの意味はそんなに LISP してない。ここの各シンボルは特別に認識されるものだから。こういうシンボルの集合が特別に定義されていて、必要なのは全部そこにある。そしてそれぞれが特定の引数のパターンを持ってる。たとえば：“reg” は必ず integer で、それはレジスタに番号がついてるからだけど、“+” は必ずサブの式を二つ持ってて、とかね。そしてそれぞれの式にはデータ型があって、それが基本的に、それが固定か浮動か、バイト長はどんだけか、なんてことを指示する。必要なら、ほかの物を扱えるように拡張することもできる。

それでぼくが自動レジスタ割り当てをやる方法ってのは、ぼくが最初にコードを生成するとき、組み合わせとかいろいろやるときだけど、レジスタに入りそうな変数にはすべて、自称疑似レジスタ番号ってのを割り振って、これは 16 とかなんとか、そのターゲットのマシンで実際のレジスタになるには大きすぎる番号からはじまる数字だ。だから本物のレジスタは 0 から 15 までとかなん

とかに割り当てられてて、そのあとに疑似レジスタがくる。それで、コンパイラの最後のところで何をやるかというと、ずっと見てって、疑似レジスタを本物のレジスタに変えてくんだ。またもやここでコンフリクトグラフをつくって、どの疑似レジスタ同士が同じ時点で生きてるかを見て、もちろんそれを本物のレジスタの同じところに入れるわけにはいかないので、だからなるべく疑似レジスタを固めてなるべく本物のレジスタに入れるようにして、しかもその重要度に応じて並べるわけ。

そして最後に、いろんな問題に対応してコードを訂正しなきゃならない。たとえば本物のレジスタにおさまりきらない疑似レジスタがあって、それをかわりにスタックのスロットに入れなきゃならないとする。一部のマシンだと、これが起きると命令の一部は無効になっちゃうかもしれない。たとえば68000だと、レジスタの内容をメモリに加算したり、メモリの内容をレジスタに加算したりはできるけれど、メモリの内容同士の加算はできない。だからADD命令があって、68000を相手にしてて両方の値がメモリにおさまっちゃったら、この命令は無効になる。だから最後のパスでは、ずっと見てって、必要に応じていろんなものをレジスタにコピーしたり、戻したりして、こういう問題を片づける。

インデックスレジスタも問題になる。何かをインデックスにしてアドレスを決めるとき、そのインデックス値がメモリに入ってたら、そのコードはほとんどの場合役にたたなくなる。ただし、間接アドレッシングでそれができるマシンは別だけど。インデックスレジスタに自動インクリメントとかかけてるときには、その値をレジスタに入れて、命令をやって、インクリメントした値をほんとにあるべきメモリのスロットに戻してやらなきゃならないかもしれない。

まだまだいろいろ小細工の余地はあって、ぼくもまだ十分に効率よくなるほどの細工は実装しきってない。

このコンパイラは、Cのコードをとって、それを実質的にはCデータ型の注釈がついた構文ツリーに変えるパーサーを持つことでいまは動いてる。それから次のパスではそのツリーをながめて、こんな（LISP状の）コードを生成する。それから最適化パスがいくつかある。一つには、ジャンプからジャンプ、ジャンプへのジャンプ、.+1へのジャンプなんかを扱うもので、こんなのはみんなすぐに簡素化できる。それからよくある副次式の認識、さらに基本ブロック探し、それからデータフロー分析をして、どの命令にどの値が使われて、その後まったく使われないのはどれかがわかる。そしてそれぞれの命令を、それが使う値の生成場所とリンクさせて、だから疑似レジスタR[28]を生成するある命令があって、別の命令があとでR[28]を使って、それがR[28]を使う最初の場所だったら、二番目のやつが最初のやつを戻って指すようにして、このポインタが、命令を組み合わせようとするときの制御に使われる。隣り合った命令を組み合わせるんじゃなくて、ある値を使う命令と、それを生成する命令とを組み合わせるの。間にほかの命令がはさまっていても、ここでは関係なくて、単に介入してきたりしないかどうかチェックしなきゃならないだけ。それで組み合わせ機能が動的レジスタ割り当てをやって、そして最後に、アセンブリコードにする部分がくる。

アリゾナコンパイラでは、命令認識部分はLEXで生成されてた。マシン記述はただのLEXプ

ログラムで、LEX はそれを C 関数にして、有効な命令を文字列として認識する。ぼくがかわりにつけたのは、特別な専用デシジョン・ツリーで、まるで LISP みたいなこの構文で書かれたマシン記述から生成される。そしてこの認識部分は、コンパイラ内のいろんな部分のためのサブルーチンとして使われてる。

いまんとこ、このコンパイラは PCC くらいの速度で走る。レジスタ割り当ての小細工をするなといえば、目に見えて速くはなって、それだと PCC とレジスタの割り当ては同じ。超小細工モードだと、PCC よりずっと上手にレジスタ割り当てをやって、ぼくの見立てでは VAX だと VAX 用のどの C コンパイラよりもいいコードを生成してくれるね。

68000 だとコードはまだ理想的とはいえない。前段で、十分に先を見通してないから、最高とはいえないようなことをしてる部分が見える。前段では選択の余地があって、だからそいつが一番いいと思うことをやるんだけれど、でも別のやりかたをしてくれたら後段のほうが頭がいいから、もっといい処理をしてくれたはず。だけど前段は、後段がそんなことしてくれるとは知らないので、だからそういうのでもっと手を入れないと。

ときどきこのせいで、レジスタが無用に解放されちゃう。だって、何かがメモリのほうにおさまって、それをレジスタにコピーしなきゃならなかったら、まずはそれをコピーするためのレジスタが必要になる。これはつまり、すでに割り当てたレジスタを持ってきて、一時的な値をスタックのスロットに蹴り出すってことだ。もちろんそういうのがレジスタからメモリにいっちゃうと、さらにほかの命令が無効になるかもしれなくて、だから何度も何度もチェックが必要になる。ときどきこいつは、何かをレジスタにコピーしなきゃと思うんだけど、でもそんな必要がなかったりして、すると必要以上のレジスタを解放しちゃって、使えるレジスタを使い切らなかったりするんだ。

（質問: 32000 用のコードジェネレータはありますか？）まだだけど、でも言っとくと、要るのはコードジェネレータじゃなくて、ただのマシンの記述だけね。そのマシンの命令が全部こんなふうに（LISPっぽい形式で）リストしてあればいい。だから実際問題として、どの引数がレジスタに入れてとかどのレジスタにとかいう制約条件の考え方を実装するところは別にして、これは 68000 には必要だけど VAX では不要だった話なんだけど、でもそれを別にすれば、このコンパイラを VAX から 68000 に移植するのはほんの数日しかかからなかった。だから、すごく簡単に移植できるんだ。

コンパイラはいまはアセンブラ・コードを生成して、デバッグ情報を DBX の求める形式でも出せるし、GDB の特別な内部形式でも出せる。ぼくに言わせれば、このコンパイラで手をいれなきゃいけないのは、あとたった 3 分野だけ。

1. 「プロファイリング」機能をつけなきゃならない。UNIX のコンパイラにあるやつ。
2. レジスタ割り当てをもう少し賢くしないとダメ。出力にバカな代物が出てこないようにする。
3. 最後に、いろんなバグがあるし、ちゃんと扱えてないものがある。自分自身はコンパイルはできてるんだけどね。

これだけやるのに、まあ数ヶ月もあればってとこで、そしたらコンパイラをリリースする。

3.5 TRIX カーネル

システムの中ですでに存在する大きな部分がカーネル。(質問：休みは？) ああ、うん、そういや休憩を忘れてたね。とりあえずカーネルの話だけ終わらせてよ。5 分ほどですむから。そしたら休憩にしよう。

で、カーネルには TRIX （ぼくの知る限りでは、何の略でもないみたい）っていうシステムを使う予定[5]。これは MIT の研究プロジェクトで開発されたもので、リモート・プロシージャ・コールに基づいてる。だからプログラムはドメインと呼ばれてる。各ドメインはアドレス空間といろんな機能（capability）で、機能（capability）ってのはまさに、ドメインを呼び出す能力なんだ。どのドメインもそれを呼び出す “capability ports”（機能ポート）がつくれて、そしてシステムコールとほかのユーザドメインをコールするのとがまったく同じ。どっちをしてるのかすら区別できない。だからほかのユーザプログラムでデバイスを簡単に実装できる。ファイルシステムも、透過的にユーザプログラムで実装できる。さらにネットワークごしに通信するのも透過的。ほかのドメインを直接呼んでるつもりでも、実はネットワークサーバのドメインを呼んでるかもしれない。コールで与えた情報をとって、それをネットワーク越しに別のサーバプログラムに渡して、それがこんどはあなたの話そうとしてるドメインを呼び出す。でも、あなたとその相手のドメインには、そういう動きはまったく見えずに起こる。

TRIX カーネルは動くし、ごく限られたかたちで UNIX と互換性もあるんだけれど、でもまだまだだね。いまのところ、ディスク上では古くさい UNIX ファイルシステムが使ってるのと同じ構造を使ったファイルシステムを持ってる。おかげでデバッグはやさしいよ。ファイルを UNIX でセットして、それを TRIX で走らせたりできるから。でもこのファイルシステムは、ぼくが必要だと思う機能をぜんぜん持ってない。

どうしても追加するべきだと思う機能としては、バージョン番号、削除ファイルの復活、ファイルがいつどこでテープにバックアップされたかの情報、ファイルの詳細更新（atomic superseding of files)。UNIX でいいと思うのは、ファイルが書き込まれているときには、いつでもどうなってるか見られるってことね。たとえば “tail” を使ってどこまで進んだか見るとか、あれっていいよね。それでプログラムが、ファイル書きかけで死んだりしたら、どこまで行ったかも見られる。こういうのっていいんだけど、でもこの書きかけの出力が、いずれ期待してた完全な出力にまちがえられるようなことは、絶対にあってはならない。その前のバージョンもちゃんと見られて、新しいバージョンが完全に正しくできるまでは、それを使おうとする人みんなに使われるべきでしょ。ということはつまり、新しいバージョンはファイルシステムの中で見えなきゃダメだけれど、でも名前は予定されてた名前じゃいけないってことだ。作業が完了してはじめてリネームされるようにしないと。これはまさに ITS がそうで、だけどここでは、各ユーザプログラムがそれを明示的にやる必要があった。ユーザプログラムに UNIX と互換性を持たせるには、これを目に見えない形で

[5]訳注：不詳。きいたことない。でも、すでにマルチサーバ式の OS らしきものは構想されているのがわかる。なおここでは、この 1986 年の時点でカーネルがまもなくできそうな雰囲気だが、実際に GNU のカーネルである HURD がまがりなりにもリリースされたのは、1997 年になってからのことだった。

やる必要がある。

バージョン番号を、いまの UNIX のユーザプログラムにフィットさせるための、すごい小細工っぽい方式を考えてある。それでこれって、ファイル名をそのまま指定してバージョン番号を略すと、ふつうの形で名前を指定するってこと。でもファイル名をはっきり指定したければ、たとえばはっきりどのバージョンを使いたいか指定するとか、あるいはぜんぜんバージョンを使いたくなければ、ファイル名の最後にピリオドをつける。だからもし “HOGE” っていうファイル名を与えたら、これはつまり「HOGE のバージョンを全部見て、最新のヤツをもっといで」という意味。でも “HOGE.” といえば、「純粋に HOGE という名前だけのファイルを持っといで、それ以外はいらないよ」ってこと。“HOGE.3.” ってのは「ずばり HOGE.3 って名前のファイル」で、これはもちろん HOGE のバージョン 3 そのもの。出力では、単に “HOGE” といったら、これはいずれ “HOGE” の新しいバージョンをつくるけれど、でも “HOGE.” と指定すれば、純粋に “HOGE” という名前のファイルに書き込む。

さて、細かいところを全部つめて、問題が残ってないかとか、UNIX がファイル名にピリオドをつけたらいかれるかかなんとか、同じ行動をさせるようにするには、いろいろまだハードルが残ってはいる。

出力用に、名前がピリオドで終わるファイルを開いたら、その名前をすぐに開いて、だから同じ UNIX の振る舞いが得られるようにしたい。書きかけの出力がそのまま見えるようになって、でもピリオドで終わらない名前に出力したら、閉じたときには新しいバージョンがあらわれて、しかもそれを明示的に閉じないと新しいバージョンにならない。システムがクラッシュしたとかなんとかで、ジョブが死んだためにファイルが閉じたら、それは名前が変わる。

そしてこのアイデアは、「スター・マッチング」に結びつけられる。つまり、ピリオドで終わらない名前はバージョン番号のついてない名前とマッチするようになるってわけ。だからあるディレクトリにこんなファイルがあったとしよう：

```
ho.1    ho.2    ge.8
```

ここで “*” と言ったら、それは：

```
ho    ge
```

に相当する。名前をとって、バージョン番号をそこから除いて、それで区別がつくものを選ぶから。でも “*.” といえば、絶対名を全部とってきて、それにピリオドをつけて、それに対してマッチしたのをさがす。だから存在する個別バージョンがすべてあてはまる。同じようにして “*.c” と “*.c.” のちがいもわかるよね。こいつ（最初の）は基本的にバージョンなしの “.c” ファイルすべてをさすんだけど、こいつ（二番目）は全バージョンをさす……わけじゃないな、それだと “*.c.*.” とやんなきゃいけないのか。まだ細かいところはつめきってないんだ。

もう一つ、ユーザからは見えない機能でしかも確実に互換性があるのが、ファイルシステムのフェイルセーフさってこと。つまり、全情報をディスクにちゃんとした順序で書けば、うまくそれ

ができれば「停止」ボタンをいつ押しても、それでディスク上のファイルシステムがいかれるようなことは絶対ないふうにできる。これのやりかたはよく知られてる。なんでみんなそれを無視するのか、想像もつかないよ。もう一つのアイデアは、さらに情報の冗長性をもたせる。これをやるかどうかはよくわからないけど、でも各ファイルに名前を全部入れて、だからディスク上のディレクトリが壊れても、それをディスクのほかの中身から再構築することが可能になるようにする方法について、ちょっと考えがあるんだ。

あと、ファイルの任意の一部を細かく（atomically）更新できるようにするにはどうしたらいいか、ぼくはわかってるつもりだ。つまりファイルの一部分を新しいデータで更新して置き換えるときに、それだとファイルを読もうとすれば、見えるのは古いデータだけとか新しいデータだけとかね。できると思う。しかもファイルをロックしたりとかもしなくてね、ぜんぜん。

ネットワークのサポートとしては、いずれこのシステム用に TCP/IP を実装するつもり。それと実質的に UUCP に相当するものとして、KERMIT がつかえると思う。

シェルは確かもう書き上がってるはず。二つのモードがあって、一つは BOURNE シェル風で、同じプログラムが別のモードでは C シェル風になる。まだこいつはぼくの手元には届いてないから、どのくらい手をかけなきゃなんないかもわかんないや。それ以外にもたくさんユーティリティがある。MAKE はあるし、LS も、あと BISONっていう YACC にかわるものもあって、配布されてる。LEX にかなり近いものもできてるんだけど、完全に互換性はないので、ちょっと作業が必要。そして全体として、これからやんなきゃならないことは、もう済んだことよりはずっと少ないんだけれど、でもまだまだたくさん手伝いがいるんだ。

みんながしょっちゅうきくのが「いつになったら完成するの」ってこと。もちろんぼくだって、いつできんのかなんてわかりゃしないけど、これはぼくにすべき正しい質問じゃない。もしその人がそれにお金を払う気なら、そりゃずばりどんなものをいつ手に入れられるか知りたがるのは当然だ。でも、お金を払うことにはならないんだから、きみたちがきくべき正しい質問は「もっとはやく完成させるために、どんな手伝いをしたらいいですか」なんだ。プロジェクトのリストがあって、MIT のファイルにおいてあるんだけど、手伝いたい人はこのインターネットアドレスにメールをくれれば、プロジェクトのリストを送ってあげる。（こいつ、うまく動くかな（とチョークを見つめて言う）。これ、読める？「RMS@GNU.ORG」だよ（カラオケ風に色が変わる通り）。で、ここでちょっと休憩といこうか。そして休憩が終わったら、ぼくはすごい問題発言をいろいろするからね。いま帰っちゃだめだよ。いま帰ったら、ホントの山場をみのがすからね。

[ここで 15 分休憩]

4　情報、ソフトと著作権

GNU ソフトの入手法を言っとくようにと言われた。で、一つの方法はもちろん、もしそれを持ってる友だちがいたら、それをコピーすればいいけど、でもそういう友だちがいなかったら、そしてインターネットにもつながってなかくて、だから FTP も使えないなら、配布テープを注文して

フリーソフトウェア財団（FSF）にお金を送ってくれればいい。もちろん、フリーソフトは無料配布ってのとは別物なんだよ。この話はまたあとで詳しくやる。

ここに取り出しましたるは EMACS のマニュアルだけど、きれいに印刷製本したやつ。写真製版してオフセット印刷してある。Emacs の配布パッケージに入ってるソースから自分で印刷してもいいし、こっちのやつを FSF から買ってくれてもいい。あとでここにきて、こいつを見てみて、それとこれには注文書もあってあとでそれを写したりとか、あとこの（表紙の）絵もおもしろがってもらえたりする。こいつ（と、絵の中でヌーにまたがった RMS に追いかけられてる人物を指さす）は、びびってるソフト隠匿者。あとでこいつの話もしよう。

ソフトウェアってのは比較的新しい現象だ。人がソフトを配布し出したのは、せいぜい 30 年前かな。だれかがそれを商売にしようと思いついたのは、たった 20 年ほど昔だ。人がどうするかについて、なんの前例もないし、だれが何の権利を持ってるかもわからない世界だった。そして、アナロジーでほかの分野の伝統を持ち込めたんだけど、その際のアイデアがいくつかあった。

ヨーロッパの教授たちがお気に入りのアナロジーは、プログラムと数学とのアナロジー。プログラムってのは一種の大きな数式みたいなもんだ。さて伝統的には、だれも数式を所有したりはできない。だれでもそれを写して使える。

一般の人にいちばん意味があるアナロジーは、料理のレシピだろう。考えてみれば、日常生活でプログラムにいちばん近いものっていえばレシピなんだよ。ちがいといえば、レシピは人が従うもので、機械が自動的にやることじゃないってこと。確かにレシピにはソースコードとオブジェクトコードの差はないけれど、でもいちばん近いのは事実。そしてだれもレシピを所有することは認められていない。

でも選ばれたアナロジーは、本とのアナロジーで、本には著作権がある。そして、なぜそれが選ばれたのか？　それは、この選択をすることでいちばん利益を被る人が選択権を与えられたからだ。プログラムを使う人じゃなくて、書く人が決定を認められて、そしてみんなはまったくの利己的な決断をくだして、おかげでプログラミングの世界は醜いところに変わってしまった。

ぼくがこの世界に入ってきたとき、1971 年に MIT で働きだしたとき、ぼくたちの開発したプログラムが共有されないかもなんて考えは、議論すらされなかった。そしてスタンフォードも CMU もみんな、DIGITAL でさえそうだった。当時の DIGITAL からの OS は無料だった。それにぼくも、PDP-11 クロスアセンブラとかあれこれプログラムをもらうことがよくあって、それをポートして ITS で動くようにして、いろんな機能も加えた。そのプログラムには版権はなかったんだ。

こいつが変わりはじめたのは、やっと 1970 年代後半に入ってからのことなんだよ。ぼくはぼくらの共有精神にはすごく感激した。ぼくたちは、願わくば役に立つことをやっていて、人々がそれを使えれば幸せだった。だから最初の EMACS を開発したとき、MIT の外の人がそれを使いたいな、と言い出したとき、ぼくはそれが EMACS “コミューン” のものだ、だから EMACS を使うにはそのコミューンの一員じゃなきゃだめで、それはつまり、自分が改良を加えたらそれを貢献する責任があるんだという意味だよ、と言った。オリジナルの EMACS への改良はすべてぼくに送らなきゃならなかったんだ。そうすれば新しいバージョンの EMACS にそれを入れて、コミュニティ

の全員がメリットを得られる。

でも CMU で SCRIBE が開発されて、それが会社に売られたときに、これが破壊されはじめた。これはいろんな大学のぼくたちみんなにとって大問題だった。みんなの目の前に大きな誘惑がおかれて、協力をやめるのがすごく儲かるようになって、協力を信じているぼくたちは、協力するように説得する材料がなにもなかった。はっきりと、一人また一人と、寝返って社会との協力をやめて、ぼくたちの中ですごく良心の強い人だけが協力を続けるだけになるだろう。そしてまさにそうなったわけ。

プログラミングの世界は醜いところになっちゃったよね。みんなシニカルで、同じ分野の人たちやユーザに意地悪にしたらいくら儲かるかな、と考えてる。

ぼくは、ソフトを所有するという慣行は物質的にも無駄が多く、精神面でも社会的に有害で邪悪なものだと断言したい。この 3 つはすべて相互に関係しあってる。なぜ精神的に有害かと言えば、それはコンピュータに触れるあらゆる社会人にかかわるからで、その接触を明らかに他人に対して無駄の多い形でやらせる慣行だからだ。そして自分だけの利益のために何かをして、しかもそれが自分の助かるよりも他人に害を与えるほうが大きいことを知ってやるたびに、心の中でそんなことを正当化するために、きみはシニカルになるしかない。そしてそれは、意図的に社会で行われた作業を無駄にするものであり、社会の退廃を招いているがゆえに邪悪なんだ。

まず、ソフトウェアとか、その他一般的に役に立つ情報を所有しようとすることで、どんな害が生じるかを説明しよう。それからその慣行を弁護するような議論に反論して、この現象とどうやって闘えばいいか、そしてぼくがどうやって闘ってるかを話そう。

4.1 ソフトの所有とその害

情報の所有という考え方は、3 つの異なるレベルで有害だ。三つのレベルで物質的に有害だし、それぞれの物質的な害は、対応する精神的な害を伴う。

最初のレベルでは、それは単に、利用者がそのソフトを使うのをじゃまするんだけど、実は使う人が少なくてもプログラミングの作業が減るわけじゃない。プログラムの利用にお値段がついたらこれはみんながプログラムを使わないインセンティブ、というのはこの手のソフト隠匿者どもが大好きなことばなんだけど、使わないインセンティブになって、これは無駄だ。たとえばあるプログラムにお値段がついて、だから使う人間の数が半減しちゃったら、プログラムの半分は無駄になったことになる。同じだけの作業が、半分の富しかつくりださなかったんだから。

さて実は、プログラムが使いたい人みんなに出回るようにするには、特別なことは何もしなくていい。みんなコピーくらい自分でちゃんとできるし、だからいずれはみんなの手に入る。プログラムを書いたら、あとはすわってて、みんなのしたいようにさせればいいの。でも、そうはならない。かわりにだれかが意図的にプログラムの共有を妨害しようとする。そしてそれどころか、そいつが妨害しようとするだけじゃなくて、ほかの人を手伝わせるよう圧力かける。ユーザが守秘契約にサインしたら、その人は要するにほかの仲間のユーザたちを売り渡したってことだ。黄金律にし

たがって「ぼくはこのソフトが好きだ、ぼくの隣人もこのソフトが気に入るだろう、だから二人ともそれが持てるようにしよう」というかわりに、そいつはこう言ってるんだ。「いいよ、おれにくれよ。隣人なんか地獄に堕ちろ！　おれもそいつを隣人の手に入らないようにするのに手を貸すからさ、とにかくおれにくれ！」　そしてこの精神が、精神的に有害なんだ。この「隣人なんか地獄に堕ちろ、**このおれに**コピーをよこせ」という態度が。

守秘契約とかなんとかにサインしたからってんでぼくに何かのコピーをくれないような人に出くわしてから、ほかのだれかがぼくにその手のものにサインしろって言ったときには、そんなのがまちがってるのはわかった。自分がやられてあんなに腹がたったことを、このぼくが人にやれるわけないでしょ。

でもこれは、害の中で最初のレベルにすぎない。第二のレベルは、人がプログラムを変えたいときに生じる。だって、使いたい人みんなに完全にあったプログラムなんてないもの。人は料理に手を加えるでしょ。塩を減らすとか、ちょっとピーマンを入れるとかさ、同じように、プログラムだって自分のほしい効果を得るには手直しがいるんだ。

さて、ソフト所有者たちは、人がプログラムを変えようがどうしようが、実はぜんぜん気にしてないんだけれど、ただ連中の目的のためには、人にそんなことをさせないほうが都合がいい。ソフトが独占ソフトだと、ソースは手に入らないし、変えられないし、おかげでプログラマにとってはすごく無駄な作業が増えて、ユーザもいらいらがつのる。たとえば、何ヶ月も銀行でプログラマやって、新しいプログラムを書いてた友だちの話なんだけど、それにはほとんど大丈夫な市販ソフトがあったんだけれど、でもかれらの求めるものそのものってわけじゃなくて、その差があるおかげで、その市販ソフトは使いものにならなかったんだ。そこの部分だけ変える労力はほんのわずかですんだだろうね。でもそのプログラムのソースがなかったから、それは不可能だった。彼女はゼロからはじめるしかなくて、だからたくさん仕事を無駄にした。世界でこんな風に時間を無駄にしているプログラマが、どのくらいいるのかは神のみぞ知る、だよね。

それと、あるソフトがその場しのぎには使えるけれど、でもしっくりこないことがある。たとえば MIT にはじめてグラフィックプリンタがきたとき、ソフトは自分で書いて、いろいろすてきな機能をつけたんだ。たとえば自分のジョブが印刷し終わったらメッセージを送ってくれるとか、自分のジョブがキューに入ってる時に紙切れになったら教えてくれるとか、ほかにもいろいろほしいような機能を入れた。その後、もっとずっといいグラフックプリンタが入って、最初のレーザプリンタだったんだけど、でもそのソフトは Xerox のもので、それは変えられなかった。で、そういう機能とかも入れてくれないし、ぼくたちが足すこともできなかったし、だから「その場しのぎ」の代物で我慢しなきゃならなかった。自分たちには、それをなおす意志もやる気も能力も十分にあることがわかってるのに、それが許されてないってのは、すごくいらだたしかったね。これって妨害工作だよ。

それと、コンピュータを使ってるのに、コンピュータはわけわからん、仕組みがわからんと言う人がたくさんいる。うん、そりゃわかりっこないよね。自分の使ってるソフトが読めないんだもん。プログラムの正しい書き方を学ぶ唯一の方法、そしてプログラムがどう動いてるのかを知る唯

一の方法は、ソースコードを読むことなんだもん。

だから思うんだけど、コンピュータをただの道具としてしか考えない利用者ってのは、実はソースコードを秘密にしとく習慣のせいで生まれた、ニワトリか卵か式の悪循環なのかもしれないよ。

さてこの種の物質的害に伴う精神的な害は、自分さえよければという精神だ。人が長時間コンピュータシステムを使ってると、そのコンピュータシステムの設定はその人が住まう都市になる。ちょうど自分の家や家具の配置が、その中ですむぼくたちの生活を規定するように、ぼくたちの使うコンピュータシステムもそうで、自分たちにあうようにコンピュータシステムを変えられなければ、ぼくたちの生活は実は他人に支配されてることになる。そしてこれに気がついた人は、ある意味でモラルが低下するんだ。「こういうのを変えようとしても無駄だよ、ずっとこうやってひどいままなんだから。不満をいうのさえ無駄だ。とにかく時間をつぎ込んで … これがすんだらぼくがどっかへ行こう、そしてもうこのことは考えないようにしよう」　公徳心ってのもがあるのに物事の改善が許されないと、結果としてこの手の精神、この手のやる気のなさが生じるんだ。

第三のレベルの害は、ソフトウェア開発者自身の間のやりとり。あらゆる知識分野は、人が他人の成果の上に積み上げられるときにいちばんはやく進歩する、でも情報の所有権は、まさに他人がそうしないようにするためにつくられてる。もし人が他人の成果に積み上げられたら、その所有権ははっきりしなくなって、だからみんな、その分野への新規参入がゼロからはじめるしかないように手を打って、おかげでその分野の進歩が大幅に後れる。

だからわかるでしょ。表計算ソフトが別々の会社からたくさん出てるけど、みんなそれがそれまでどうやってきたのか見てみるという恩恵なしにやってる。そりゃ確かに、最初に書かれた表計算ソフトは完璧じゃなかった。たぶん、一部のコンピュータでしか動かなくて、なにかをするときにもいちばんいい方法ではやらなかった。だから、それを部分的に書き換えたい人が出てくる理由はいろいろあったろう。でも、自分の改善したいところだけを書き直せばいいんだったら、作業量はずっと少なくて済んだよね。システムのある部分をよくする方法は見えるかもしれないけれど、同じシステムの別の部分なんかぜんぜんマシにできないかもしれない。いや、同じくらいのレベルに達するのだって、えらく苦労するかもしれない。自分の好きなところをとって、自分がひらめいた部分だけをやりなおせたら、あらゆる面で前より優れたシステムが手に入って、まったく新しいシステムを書くよりもずっと作業は減る。そりゃシステムをゼロから書き直すといいこともあるのはみんな知ってる。でもそれは、古いのをまず読んでからの話だ。

だからプログラミング業界の人たちは、時間をたくさん無駄にする方法を編み出したわけで、おかげで表面上は、本当に必要なのよりもずっとたくさんのプログラマが必要になったように見える。なぜプログラマ不足なんてことが言われるのか？　知的所有権のおかげで、プログラマたちは自分のやる仕事の半分を無駄にするような仕組みにしちゃったからだよ。だからみんなが知的所有権システムを指さして「ほら、こんなに雇用を生み出してる、産業がこんなにでかくなってるじゃないか」なんて言うけど、それが証明してるのは、実はみんながお金と時間をたくさん無駄にしてるってことなんだ。プログラマの生産性を上げる話をするときでも、高度なツールがどうのこうのといえば、連中は喜ぶけど、でもプログラマのやってるよけいなことを削ることで生産性をあげ

るって話になると、とたんに反対する。そうなったら雇われてるプログラマの数が減るからって。これってちょっと、分裂症じみた議論だと思わない？

そしてこのレベルの物質的害に対応する精神的な害は、科学的な協力精神に及ぼす害だ。これは昔はすごく強くて、戦争してる国同士の科学者ですら協力を続けてた、自分たちがやってるのが戦争とは何にも関係なくて、人類の長期的なメリットのためなんだってのがわかってたから。最近じゃみんな、もう人類の長期的なメリットのことなんか気にもしない。

プログラムの利用をじゃまするのがどんなことか理解するには、仮にここにサンドイッチがあって、食べられるけれど、食べてもなくならないとしよう。あなたが食べて、別の人がその同じサンドイッチを食べて、何度でも食べて、それでも毎回もとのと同じだけ栄養がある。

それでやるべきいちばんいいこと、このサンドイッチを使って何をすべきかといえば、おなかのすいた人たちがいるところにそれを持ってくことだ。なるべく多くの口にそれを与えて、なるべく多くの人の腹を満たすようにすることだ。このサンドイッチを食べることに、値段なんか絶対つけちゃいけない。値段をつけたら金がなくて食べられない人が出てくるからで、そうしたらそれは無駄になる。

プログラムってのはこのサンドイッチみたいなものなんだけれど、でももっといいもので、なぜかといえば同時にいろんなところにあって同時に食べられて、どんどん別の人に使ってもらえるから。このサンドイッチがみんなをあらゆる場所で永遠に食べてもらえるのに、だれかがそれを自分の所有物だと決めたせいで、それが実現できないでいるんだ。

4.2　ソフト所有肯定論とその反駁

さて、プログラムを所有できると信じてる人たちは、ふつうはそれを正当化するのに 2 つの議論を持ち出す。最初の議論は、「おれが書いたんだ、おれの魂の申し子だ、わが心、わが魂がここにはこもってる。それを他人が奪い取るなんて？　どこまでいってもこいつはおれのもんだ、おれのおれの**おれのぉっ！**」というやつ。でも不思議なことに、こういうこと言う人のほとんどは、そのソフトは勤め先の会社のものだっていう合意書にサインしてるんだよね。

だからこれって、簡単に自分をごまかして何かが重要だと思いこんでしまえることの一つだと思う。同じくらい簡単に、そんなのぜんぜん大事じゃないと自分を説得することもできるんだ。

こういう人はこの論法を使って、人がこのプログラムを変えるやりかたまでコントロールする権利を要求するんだ。「だれにもわたしの芸術作品をめちゃくちゃにさせてなるものか」って。じゃあ、これからつくろうと思ってる料理を発明した人が、それはその人の芸術作品だからってきみの料理の仕方をどうこうする権利があったらどうなると思う？　塩を使わないようにしようと思っても、そいつは「ダメダメ、わたしがこの料理を設計したんだから、これはこんだけ塩をいれなきゃダメ！」「でも、ぼくは医者に塩をとめられてるんですよ、どうしましょう？」

明らかに、プログラムを使ってる人のほうが、その現場に近いんだ。プログラムの利用はその人に直接はねかえってくるけど、それを書いた人には、一種のごく抽象的な関係しかない。したがっ

て、人々に自分自身のくらしをできる限り好きなようにさせるという趣旨からして、そういうことを決めるのはそのユーザであるべきなんだ。

連中の二番目の議論は、経済的なものだ。「プログラムを書いても報酬が得られないじゃないか」と言うわけで、ここには確かにホントの問題がちょっとはある。でも連中のいうことのかなりの部分は混乱してる。そしてどこが混乱してるかっていうと、「たくさんの人がプログラミングするようにしたいなら、ほかの手段で生計をたてなくてもいいような仕組みをつくらなきゃ」というのと、その一方では「いまのシステムじゃなきゃダメだ、プログラミングで金持ちにならなきゃ」というのとでは、話がまるっきしちがうってこと。生きてけるだけの賃金を得るのと、少なくとも最近のアメリカのプログラマがもらってるような給料をもらうってのとでは、話がぜんぜんちがう。みんながいつも言うのは「じゃあおれはどうやって喰ってけばいいんだ？」ってことだけれど、実際には「こいつは食えるか」ってことじゃなくて「こいつは寿司[6]が食えるか」ってことで、「雨つゆがしのげるか」ってことじゃなくてホントに言いたいのは「高級マンションが買えるか」ってことなんだ。

いまの方式は、最大限の儲けを得るためにソフト開発に投資しようと思った人の選んだシステム。システム開発をサポートするための唯一の方法だからこうなってるわけじゃないんだ。実は、ほんの 10 年とか 15 年とか前のごく最近までは、ソフト開発をほかの方法でサポートするのがあたりまえだったんだ。たとえば DIGITAL の OS は 1970 年代初期までフリーだったけれど、フリーの OSってのは、70 年代初期ですら、ちゃんと給料もらってる人が開発してたんだよ。役に立つプログラムの多くは大学で開発されてる。最近だとこういうプログラムは売られることが多いけど、15 年前ならふつうはフリーで、それでもみんな給料がもらえてたんだ。

プログラムみたいなものだと、これは無限サンドイッチとか、道とかみたいなもので、一回はつくらなきゃダメだけど、一度つくったらもうそれを何回使おうが関係なくて、使うのにコストもかかんないなら、使うのに値段なんかつけないほうが一般的にいい。そして、いまでもぼくたちがつくって、作った人に金を払うようなそういうものってのはたくさんある。たとえばそこらの道がそうだよね。お金を払わずに道を造ってくれる人を見つけるのは無理だ。道をつくるのはプログラミングとちがってクリエイティブじゃないし、楽しくもない。でも、世の中にはたくさん道があって、それに支払うだけの金も捻出されてる。そしてそのほうが、ずっといいよね。でもこういうことだってできる。「企業に道をつくらせて勝手に料金所をつけさせて、街角を曲がるたびに通行料を支払うようにしよう。そしていい場所に道を敷いた企業は儲かって、そうでないのは倒産するようにしよう」だれかが、何かを隠匿することで大金を儲ける方法を編み出すと、変なことが起きる。それまでは、たぶんその分野にすごく情熱をもっていて、熱心に働く人たちがたくさんいたんだ。唯一の問題は、その人たちはそもそも喰っていけるのか、ということだけ。たとえば数学者を考えてみると、純粋数学者に支給されるお金よりも、純粋数学者志望者のほうがずっと多い。そして支払われたとしても、大した額じゃないし、あまりいい暮らしもできない。ミュージシャンとなるともっとひどい。時間の大半を費やしてミュージシャンになろうとしてる人が、平均的なミュー

[6]訳注：高くて高級なぜいたくな飯という意味。

ジシャンでどのくらい稼いでるかっていう統計を見たことがある。マサチューセッツでは、確か州平均所得の半分以下だったんだよ。かつかつで暮らしてけるくらいで、つらいよね。でも、そうしたがる人はたくさんいる。そこへ、何かをすることですごくたくさん稼げるようになったとしよう。するとそういう人たちは消えて、みんなこう言うんだ。「そのくらい儲からなきゃ、だれもやりゃしないよ」

そしてぼくは、これがプログラミングの分野でおこるのを見てきた。AI 研で働いてて、大した給料もなかったのに仕事が大好きだった人たちが、いまでは年に 5 万ドルはもらわなきゃとても働けないよ、なんて言う。どうしちゃったわけ？　人の前に大金を儲ける可能性をぶら下げてやると、似たようなことをしてるほかの人たちが、そんだけの金を稼いでるのを見ると、みんな自分たちも同じくらい稼ぐべきだという気になって、だから昔ながらのやりかたを続けようという人はだれもいない。そしてこれが起きちゃったら、人に大金を払うしか手がないんだと思いこむのは簡単なんだけれど、でもそりゃちがう。もし大金を儲ける可能性がなかったら、ちょっとのお金でもそれをやろうって人が出てくるだろう。特にそれがクリエイティブでおもしろいことなら。

4.3　ソフト隠匿との闘い

AI 研のユニークな世界が破壊されるのを見てきて、ソフトを売るのがその核心なんだってことも見てきたし、さらにはさっき説明したように、ああいう社会を手に入れるにはフリーソフトが必要なんだってことも見てきた。でもそれからいろいろ考えてみて、ソフト隠匿がいろんな形で社会全てを傷つけることに思い当たったんだ。特に、人々に隣人を売り渡すよう圧力をかけて、それが社会の退廃を招いてるってことに。道で人が刺されてるのを見ても、黙って用とするのと同じ精神だよ。そこらじゅうの企業がしょっちゅう示してるのがわかる、あの精神だよね。そして、ぼくは自分が道を選べるのがはっきりしてた。その世界の一部となって、自分の送ってる人生について不幸に感じ続けるか、それともそれと闘うか。だからぼくは闘うことにした。ぼくはキャリアを捧げて、ソフト共有コミュニティの再建に努めてきたし、一般にとって有用な情報を員臆するという現象を終わらせようと努力してきた。そして GNU システムは、この目的達成のための手段なんだ。社会的な目的のための、技術的な手段だ。GNU システムによって、ぼくはソフト隠匿者どもの脅しに対してユーザたちにワクチンをあげたいんだ。

いま現在、この隠匿者どもは基本的に、人のコンピュータを粗大ゴミにしてしまう力を持ってる。アメリカでは、だいたい 50 年くらい前に、マフィアとかでそういう連中がいた。店やバーに行って、特にもちろん違法なバーだよね。それでこう言う。「ここらへんじゃあ、最近火事が多いですなあ。あんたんとこも、火事なんかになったらいやだよねえ。おれたちなら、あんたを火事から守ってやれるんだよ、月に千ドル払ってくれるだけで、ここで火事が起きないようにしてやるよ」これがいわゆる「保護恐喝」ってやつ。さていまはだれかがこう言う。「なかなかいいコンピュータをお持ちですな、それでいろいろソフトも使ってらっしゃる。さて、そのソフトに消えてほしくなければ、警察に追われる身になりたくなければ、わたしに 1000 ドル払いなさい。そしたらこの

プログラムをライセンス付きで売ってあげよう」これが人呼んで「ソフト保護恐喝」。

ホント、連中がなにやってるかといえば、ほかの人がやるべきことをしようとすんのをじゃましてるだけじゃん。なのに、連中はわれわれに対しても自分自身に対しても、自分たちがなにか有益な機能を果たしてるようなふりをしてみせてる。で、ぼくの願いは、ソフトウェア・マフィアの連中がやってきて「そのプログラムがコンピュータから消えてもいいのか？」と言ったら、利用者たちが「おまえなんかもうこわくないぞ、ぼくにはこのフリーの GNU ソフトがあるんだ。もうおまえたちは手も足も出ないんだぞ」と言えることなんだ。

さて、ソフト所有を正当化する議論として出てくるのが、みんなにものをつくるインセンティブを与えるってやつだ。ぼくは私企業の考え方には一般的に賛成だし、ほかの人が喜んで使ってくれるものをつくってお金を得たいな、と思うのにも賛成だけれど、ソフトの分野ではこれが収拾つかない状態になってきてる。独占ソフトをつくるのは、同じプログラムをつくってそれをフリーにするのとでは、社会への貢献度がぜんぜんちがう。社会の富への貢献が起こるのは、そのプログラムが使われたときだけなんだ。プログラムが使われるのを妨害したら、その貢献は起きないんだ。だから社会が必要としている貢献は、みんながえらくインセンティブをもってつくってる独占ソフトではなくて、ぼくたちが本当に求めてる貢献はフリーソフトで、ぼくたちの社会が収拾つかなくなっているのは、それがあまり役にたたないことをするインセンティブを人に与えて、役に立つことをするインセンティブをぜんぜん与えてないからだ。だから私企業の基本的な考え方がここではおかしくなっていて、だから社会自体が神経症気味とさえ言えるかもね。だってさ、個人が他人に対して、その個人自身にとってよくない行動を推奨するとき、その人は神経症なんだから。ここでは社会がそういうふうに行動してる。プログラマに、社会にとってよくないことをするように奨励してるんだから。

ぼくは変わり者なんだ。自分が社会のよき一員で、何かを社会に貢献していると感じていたいんだよ、うまいこと社会をカモにしてるなんて感じるよりはね。だからぼくは、いまの自分の道を選んだ。でもだれでも、自分が実は役にたってないことでお金をもらってるという気分は、多少なりともすっきりしないものを感じてるんだ。だから、こんなまちがったことをするためのインセンティブなんて考え方を擁護するのはやめて、みんなに正しいことを奨励するような仕組みを考えつく努力くらいはしようではないの。その正しいことってのは、フリーソフトをつくることなんだ。

ご静聴どうも。

5　質疑応答

[このあと、RMS は一時間ばかり質問に答えた。ここには質疑応答のほんの一部しか入れていない。テープが悪かったし、そのすべてをまともに書き出す暇がなかったので、悪しからず]

Q：だれかあなたを訴えたりとか、面倒を起こそうとしたことはありますか？

A：そういうもめごとを唯一起こそうとしたのは、ゴスリング Emacs の持ち主、というか自称持

ち主、詐称持ち主どもだけだな。それをのぞけば、あやをつける手がかりがないから、大して何もしようがないよね。ところで、あることを考えさせて、あることを考えなくさせるためにことばが使われてることに、みんなもっと注意してほしいなと思うんだ。いまこの分野での用語は、自称詐称ソフトウェア所有者たちが選んだもので、なるべくソフトウェアを、所有物になる物質的ものと同じ物だと思わせて、そのちがいを見過ごすようにしてあるんだ。これのいちばん明白な例が「海賊」ってことば。よき市民として隣人とソフトを共有しようとする人間を表現するのに、「海賊」ということばを使うのは拒否してほしいんだ。

話しそこねてたけど、著作権の考え方は印刷術の後に発明された。古代には、著者はおたがい自由にコピーしあってたし、それがまちがってるとも思われなかったし、すごい役にたったりもした。一部の著者が後世に伝えられたのは、その一部が断片的ではあっても、ほかの著作の中でまとまって引用されてたおかげなんだ。

これは、本というものが一部ずつ筆写するものだったからそうなった。10 部つくるのは、1 部つくるのの 10 倍手間がかかった。そこへ印刷術が発明されて、だからといって人は本の筆写ができなくなったわけではないけれど、でも印刷するのに比べたら、筆写はえらく面倒で、実質的に不可能といってよくなったわけね。

本が大量生産でしかつくれなくなったら、著作権が意味を持つようになってきて、そしてそれは読書大衆の自由を奪ったりもしなかった。印刷術を持っていない一般大衆の一員である人は、どのみち本をコピーできなかったから、著作権ができたって、別に自由を失うわけじゃなかった。だから技術的な変化のために著作権が発明されて、道徳的にも筋が通ってたわけ。いまはその反対の変化が起きてる。情報の個別コピーがどんどんよくなってきてて、やがて技術が究極にまで進歩すれば、どんな情報でもコピーできるようになるというのが見えてきた。 [テープを裏返すので中断]

だからぼくたちは、著作権なんか無意味だった古代世界と同じ状況に戻ってるんだ。

ぼくたちの所有物の概念を考えてみると、それは物質的なものからきている。物質的なものは、ほぼ保存則にしたがう。うん、確かにチョークは半分に折れるから、正確にはちがうし、すりへるし、消費もされる。でも基本的にこれは（と椅子を指さす）、椅子一脚だ。指をパチンとならしてこれを二つにするわけにはいかない。これをもう一つ手に入れるには、最初のやつをつくったのと同じようにつくるしかない。原材料がもっといるし、もっと労働もいるし、だからぼくたちの所有物の考え方は、そういう事実にあてはまるよう、道徳的に納得がいくかたちで発展してきたんだ。

だれでもコピーできる情報なら、話はちがってくる。だからそれにあてはまる道徳的な態度もちがう。ぼくたちの道徳的な態度というのは、何かをしたら人がどれだけ助かるか、人がどれだけ害を被るか考えることで決まってくる。ものだと、この椅子をとってくことはできるけど、これをコピーすることはできない。そしてきみがこの椅子をもっていったら、それで何かが生まれるわけじゃないから、これはまったく正当化できない。だれかが「おれは

この椅子一脚をつくる作業をしたんだし、こいつを持てるやつが一人しかいないんなら、それはおれだろう」と言ったとしたら、確かにみんな「うん、そりゃ筋が通ってる」と言うだろう。だれかが「おれはこのディスクにビットを刻んだんだ、だからこいつおれから奪ったら承知しないぞ」と言ったら、うん、これも確かに筋が通ってる。そのディスクを一人しか持てないんなら、じゃあそれはそのディスクの所有者でいいよ。

でもだれかがやってきてこう言ったとする。「あんたのディスクはこわさないよ、ただまったく同じものを魔法で作り出して、それを持ってくから、あんたはいままでどおりディスクを使えばいいんだよ」　そしたらこれは「ぼくは魔法の椅子コピー機を持ってて、きみはいままで通り椅子を楽しんですわったりできて、いつでも手元においとけて、でもぼくもその椅子が使えるんだ」というのとおなじで、これはいいことだ。

つくんなくていいんなら、指をパチンと鳴らせば複製できるんなら、すばらしいことだ。でもこの技術の変化は、個別コピーを所有して個別コピーで金を儲けようとする人にはお気に召さない。連中の考えは、保存する物体にしか通用しない。だからなんとかプログラムを実体のある物質みたいにしようとする。ソフト屋にいってプログラムを買うと、なんか本みたいなものについてくるでしょう。不思議じゃない？　あれはみんなに、自分が買ってるのが物体なんだと思ってほしくて、本当はデジタルのコピー可能なデータを手に入れたんだと気がつかないでほしいからなの。

コンピュータって結局のところ、万能マシンでしょ。たぶんみんな、ユニバーサルチューリングマシンについては勉強してるよね。ほかのどんなマシンでも模倣できるマシンね。万能マシンがすばらしいのは、ほかのマシンを模倣できるだけじゃなくて、その指示をコピーして変えられるってことだ。これってまさに、物質ではできないことだ。そしてこれがまさに、ソフトウェア隠匿者どもがみんなにやらせるまいとしてることなんだ。万能マシンという技術変化のメリットは享受したがってるくせに、一般社会にはそのメリットを手に入れさせたくないんだよ。

基本的に連中は「物の時代」を温存しようとしてるんだけれど、でもそんな時代はもう終わってる。だからぼくたちも、正しいとかまちがってるとかの考え方を、ぼくたちが住む世界の実状にあわせてシンクロさせるべきなんだ。

Q：つまり結局のところ、情報の所有権の問題なんですね。情報を所有することが正しいような場合というのは、あるんでしょうか。どう思います？

A：一般的な利用価値のない情報なら、あるいは個人的な情報なら、所有オッケーだと思う。言い換えると、なにかのやりかたじゃなくて、それでどうするつもりかという情報。他人にとっての価値が疑わしいもの、あなたからお金をむしり取るには使えても、それで何かを作り出したりはできないような情報。ぼくに言わせれば、そういうものを秘密にして統制するのはまったく問題ない。でも創造に係わる情報、人が使って楽しめる情報、そしてそれを持ってる

人が多ければ多いほど、もっと利用されて楽しまれるような情報、そういうのは必ずコピーを推奨すべきなんだ。