プログラミング美学 (仮題)

竹内郁雄 †

概要 「プログラミング美学」というコンセプトについて考える. そのために美学そのものについて簡単に俯瞰し, プログラミングの分野に適用できるかどうかを模索した. 現代の美学はすでになんでもありの状況になっており, プログラミングも十分に美学の対象となるという心証を得た. 最後に, 竹内の最近のプログラミングから「美しい」と言えるかもしれない具体例を 1 つ紹介する.

Programming Aesthetics (tentative)

Ikuo Takeuchi‡

Abstract We discuss the concept of "Programming Aesthetics" briefly and informally. First, we review traditional aesthetics, and then think about its applicability to the field of programming. We are now confident "programming" can be covered by aesthetics because nowadays aesthetics becomes very broad in scope. Finally, we describe a simple example of programs which may be thought of "beautiful".

はじめに

2012 年の, これまでとはかなり異なった開催形態の夏のプロシンでなにか話せというので, お題に沿って「プログラミング美学 (仮題)」というタイトルで申し込んだ. 以下に開陳するように, かなり空振っているので, 結局「仮題」を外さないで本稿を書く. 当日のスライドを少しは補っているが, ほぼスライドからの文章変換版である. なお, もとより, これは論文という性格のものではないので, 気楽に書いている. 文中, 私の個人的知合い以外は敬称を省略させていただいたことをお断りしておく.

1 「プログラミング美学」と言ったものの…

世の中にはすでに「プログラミングの美学」に類した言葉が日本語のページをググっただけでもかなり出てくる. それに倣えばいいのかもしれないが, やはり「美学」とはなんたるものかが多少気になる. 以前, (3 年間も続いた) とある月例の非公開フォーラムで高名な美学者とよくお話をしたのだが, その先生のお話はいつも妙に生々しくて, 結局学問としての「美学」がなんたるかを掴みとることはできなかった.

これを機会にと, 素人にも読めそうな「美学」の入門書を適当に見繕って購入した. 以下の書籍である.

- 野村良雄『改訂音楽美学』 音楽之友社 (1971)
- 佐々木健一『美学への招待』 中公新書 1741 (2004)
- 中井正一『美学入門』 中公文庫 (2010, 1951 初版)
- 谷川渥『美学の逆説』 ちくま学芸文庫 (2003)

最初に買って読んだのが『改訂音楽美学』である. 素人なりに, 通常の美学というのは, 繪画や造形, つまり視覚で捉えられるものを対象としていると想像していたので, 聴覚を対象とする, つまり空間軸ではなく時間軸に着目しているのが音楽美学ではないか, だとすればプログラミングも時間軸に主眼があるから適切かな, と思ったからである. 実は大昔『音楽美学』という本を買って読んだ覚えがあるのだが, どこへ消えたか見つからない. いま思うと多分ブラームス的な非具象音楽の最大のシンパだったハンスリックの有名な著書の翻訳だったようだ. 『改訂音楽美学』を読んで記憶が甦った.

さて, 私の想像するところの時間藝術としての音楽を野村がどのように論じているかに最大の興味があったのだが, それは本の最後の最後になって初めて言及されていて, かつそのような見方をすることで音楽美学に新しい活路が開けるといったような書き方だったのでちょっとがっかりした. ただし, 一般美学の特に古い歴史が正統的に書かれていて, それはそれで参考になった.

しかし, これだけでは心許ないので, 読みやすそうな入門・解説書を 2 冊 (佐々木, 中井) と「逆説」というタイトルに引かれて谷川の本を買った. 残念ながら, 『美学の逆説』は専門家の論述集であり, 素人にはとても歯が立たない. 結局, 最後の解説の部分を読んでサワっただけとなった.

野村, 佐々木, 中井のどれを読んでも同じことは書いてない. 佐々木の本にもあったが, Lisp と同様, 美学も

† 早稲田大学 理工学術院 情報理工学専攻, 東京大学名誉教授

‡ Waseda University, Emeritus Professor of the University of Tokyo

美学者の数だけあるのかもしれない. こうなると入門者としてはあちこちから面白いところを抽出して利用するしかない.

佐々木も中井も美学の対象についてはかなりブロードなのだが, 自然, 技術, 藝術の 3 つが美学の対象となると明確に書いていたのは中井である. 技術も美学の対象になるとは心強い. 素人目にも大胆と見える解説書を書いた佐々木は, 美学の対象が今日非常にブロードになったとは書いているが, 技術そのものが美学の対象になるとは, ご本人は明確に主張していなかったようである.

2 美学の対象となり得るのは?

技術も美学の対象となる. さて, 表題は「プログラミング美学」だが, ここはもう少し考える必要がある. プログラミングの文脈では恐らく次の 3 種類の美学がある.

プログラミング美学: これはプログラミング, つまりプログラムを作るという行動に関する美学である. ソフトウェア工学はそれを工学として扱う. ところで, 野村の本には, 生み出された音楽は美学の対象となるが, 音楽を生み出すプロセスに関することは音楽美学の対象とならないと書いてある. そもそも美学は藝術創出者のためのものではなく, 鑑賞者のための哲学なのであると….

プログラム美学: これは作品としてのプログラム, つまりなにかに対する記述が美しいかどうかに関する哲学である. こちらのほうが美学としては正統派だろう. 本稿のタイトルが仮題になっている一因でもある.

プログラミング言語美学: プログラム美学が作品 (記述) の美学だとすれば, これは技術 (記述法) の美学である. 繪画に関する美学であれ, 音楽に関する美学であれ, 技法というかメティエは十分に美学の対象となっていると思われる. 我々だって, この言語は美しいというようなことを言う. プログラミング言語も, 仕様, 実装を含めて立派な創作的作品だからである (と, 思ってくれない人もいそうだが…).

どの美学入門書を読んでも, 近代以降の藝術のあり方が多様化して, 藝術として不可能なものはなくなったと書いてある. それゆえ, 美学の標準的な目次はなくなったと. だとすれば, 上記 3 つ, どれも立派に新しい美学として創設してよさそうだ.

3 いくつかのパロディ

せっかく普段読んだこともないような本を読むと, パロディをつくってみたくなる.

ギリシャ時代からの歴史に詳しいのは野村の本だが, そこでは (まだ美学という言葉はなく, 哲学として括られていた) プラトンとアリストテレスの音楽美学について比較的詳細に紹介されている. それを私なりにかいつまんでみよう.

プラトンは『国家論』で音楽についてこう論じている. 「音楽は国家のために奉仕しなければならない. だから, 女々しい, かつ陶然たらしむる曲はだめ. 笛のごとき, 乱飲乱舞のための楽器はだめ」さらに「音楽家になるには, まず節制, 勇気, 寛大, 壮大およびそういったことの本質的形体, また同時にそれらと正反対の形態をそれらのあらゆる結合において知らなければならない」とある.

プラトンが現代に生きていて「プログラミング美学」を論じたとしたら, こうなるのだろうか. 「プログラミングは国家のために奉仕しなければならない. だから, 女々しい, かつ陶然たらしむるプログラムはだめ. Lisp のごとき, 乱飲乱舞のための言語はだめ」さらに「プログラマになるには, まず節制, 勇気, 寛大, 壮大およびそういったことの本質的形体, また同時にそれらと正反対の形態をそれらのあらゆる結合において知らなければならない」と続く.

しかし, プラトンの弟子のアリストテレスは, プラトンのような国家主義的な態度はとらない. 野村いわく, アリストテレスは当時の対立的音楽観の正しき中道に立った. 彼が編み出した, 我々にも頼りになる名言は「音楽の本質は閑暇における自由人の高尚な享楽である」だ. ここで音楽をプログラミングに変えた「**プログラミングの本質は閑暇における自由人の高尚な享楽**である」は少なくとも私の琴線に触れる.

「閑暇における自由人の高尚な享楽」に関連して, アリストテレスは, 自ら楽器を手にして学ぶべきかどうかについても論じている. 彼は, 技術的音楽教育を適度に肯定する. それに加わることなくして, その道の優秀な判断者になることは不可能あるいは大難事であるというわけだ. これを勝手に敷衍すれば, (あとでも似たことが出てくるが) プログラムを正しく (広い意味で) 鑑賞するには, 成長過程のどこかでプログラミングについて学んだほうがいいということである. 今日, いまだに紛糾している (?)「教科情報」の価値を予言していたのである?! しかし, そのアリストテレスも, 笛, つまり Lisp はその目的にはだめと言い放ったのであった. 残念.

4 藝術とアート, そして美

藝術は英語の art に対応する言葉であるが, 日本語のカタカナのアートは日本で独自のニュアンスをもつに至った. 日本語では藝術とアートがとても便利に使い分けられている. このあたり, 佐々木の解説書は多くの例を含めて非常にわかりやすく書かれている. たとえば, ネイルアートと言うがネイル藝術とは言わない, など. ついでながら, 博物館でも美術館でもないミュージアムという言葉で指されている施設が今日隆盛を極めているとい

うのも「美学」の概念のある種の変遷を物語っているという.

プログラミングの文脈で art を使った最も有名な本は Donald. E. Knuth 著 “The Art of Computer Programming” (Addison-Wesley, 1968 初版) であろう. この本は原著の改訂に従い, 2 度邦訳されているが, いずれもこの原題は訳されないままになっている. 翻訳が困難だったのであろう.

その後, Leon Sterling, Ehud Shapiro 著 “The Art of Prolog” (The MIT Press, 2nd Edition, 1994) という本が出版されたが, その邦題は『Prolog の伎芸』もとい『Prolog の技芸』(松田利夫訳, 構造計画研究所, 1988) であった. 副題が “Advanced Programming Techniques (Logic Programming)” だったから, それでもいいのかもしれない.

美ということなら, Andy Oram, Greg Wilson がまとめた “Beautiful Code: Leading Programmers Explain How They Think” (Oreilly & Associates Inc, 2007) や Diomidis Spinellis, Georgios Gousios がまとめた “Beautiful Architecture” (Oreilly & Associates Inc, 2009) が多分プログラミング関係の本で初めて “beautiful” をタイトルに掲げた本であろう (邦訳あり). どちらもとても良い本である.

佐々木によると, 実は日本ではそもそも「美しい」という言葉はあまり頻繁に使われないという. 日本語では漢字でも「美女」ぐらいで, あまり使われないという. そうかなぁと思ったが, シンポジウム会場に行く電車の中の女性雑誌の吊り広告では「美肌, 美顔, 美脚」がオンパレードであった. 佐々木は例に挙げていなかったが,「美技」はここで話題にしている「美学」の文脈には合っていそうだ. 佐々木によれば価値観の 3 本柱である真善美のうち,「美」だけが不思議と使われない言葉なのだそうだ. ただし,「うつくしい」はれっきとした日本語で,「いつくしむ」から派生した言葉で元来は可愛いらしいものを形容する言葉だった. 漢字の「美」は (生贄の) 羊が大きいという字の構成となっていて, 立派とか見事とかという意味だったという.

佐々木は, フランス留学中に「美しい」とはなにかについて悟り (?) を開いた瞬間について言及していて, これが面白い. 田舎町のカーニバルを見学していたとき, 父親の肩車に乗った子供が山車を見て, 外国人にもわかる生意気な口調で “Oh, c’est beau!” と叫んだ. 直訳すれば「おー, 美しい!」だが, これはどう見ても「すっごーい!」あるいは「スゲーッ!」だと, そのとき佐々木は「理解」したという. 日本語でいう「美しい」の狭い枠にとらわれない, どちらかというと “wonderful” の語感なのだ. これなら「プログラミング美学」の基準にも使えそうだ.

シンポジウム会場で「このプログラムは美しい」という言い方をする人がどれくらいいるか訊いてみた. すると, 私の予想よりはるかに多くの人が手を挙げたのには驚いた. これはやはり上述の「Beautiful 本」のおかげのような気がする. 素晴らしいプログラムを形容するのに「美しい」という言葉は以前はあまり使われなかったと思う. むしろ「エレガント」とか「カッコいい」とか「きれい」ではなかっただろうか.

どの解説書を読んでも, 近代に至って, 美の絶対的規範がなくなってきたとある. それを象徴的に表わす事件が, 1917 年の Marcel Duchamp の『泉』である. これは既製品の便器に小さなサインを入れただけの「作品」である. それから少し時間が飛ぶが, 1964 年に Andy Warhol が発表した『ブリロ・ボックス』も衝撃的だった. これは洗剤を染ませたタワシのダンボールのパッケージをそのまま木製の直方体で表現したというか, 模倣したものだったからである. 音楽の世界では, Arnold Schönberg の 12 音音楽, それ続いて, 音高のみならず, 音の持続, 強度, 音色すべてを均等に扱う Olivier Messiaen のセリー音楽が登場する. これらは従来「美」と信じられていた規範を覆すものだった.

佐々木によれば, 藝術は自然の模倣というドグマを捨て去り, 藝術がそもそも「非自然」なのものに変わってきたという. こうなると, 誰が作品を評価するのかという問題が生じる. そこで登場するのがコミュニティとコモン・センスの概念である. これを紹介していると長くなるので, 一足飛びに説明すると,「美の絶対基準とは別の, 制作者自身の説明, 誰かの支援が必要になってきた」ということである.

ブリロ・ボックスに哲学的啓発を受けた Arthor C. Danto は “Beyond the Brillo Box” (University of California Press, 1992) という本を書いた. ブリロ・ボックスには従来の藝術と異なる受け取り方が必要, つまり新しい藝術の鑑賞には感性のみならず, 知的活動 (大きな背景知識と技術能力) が必要だというわけである. 詳細の紹介は省くが, 彼の論を敷衍すると, いまここで「美学」の対象にしようとしている, プログラミング, プログラム, プログラミング言語の「鑑賞」にも上記のような知的活動が必要ということになる. これは妥当な考えだろう. アリストテレスではないが, プログラミングの技術背景をまったく知らずして, プログラミングに関わる「美」を評価することはできないだろう. これをさらに敷衍すれば, プログラミング, プログラム, プログラミング言語の 3 つは現代以降の美学の急先鋒になり得る.

5 プログラミングの美学

もう少し, プログラミング固有の話題に移ろう. プログラミングはプログラムという作品を生むための活動だか

ら従来は美学の対象にならなかったが, もうそれは無視.

中井によれば, ギリシャ建築におけるエンタシス (中ほどが少し膨らんだ柱) は, 悲劇に耐え切るあきらめや, 重さに立ち向かう姿勢の顕れである. それに対して, ヘブライ建築における空に向かう尖塔は, 奴隷として抑圧されていた時代の遠い憧れの顕れである. 古代の日本の建築は法隆寺に代表されるようにエンタシスと尖塔が混合している. しかし, 時代が下ると, 桂離宮や茶室のように寂しく, 軽すぎるほど軽く, あっさりとした建築様式が出てくる. それらに代表されるように, 日本の美は「すがすがしく, 清らか, 滑らかで, 軽く, 滞りなく, 明るく, さやけく」といった形容に合う.

ならば, そんな様式の美を具現する日本のプログラミングの例はあるのだろうか? 私の記憶によれば, 「武士道」に通じるイニシエのスタイルのプログラミングの 2 つの具体例に思い至る.

故島内剛一先生のプログラミングは一風変わっている. まだ PC でそのままプログラムを打ち込むという時代ではなかったので, 用意するのは特注の薄青碁盤 1cm マス入りの紙 (それだけだったら今時どこでも手に入るが, 島内式は紙のサイズが長辺・短辺の比とが 3 対 2 という, とても珍しい特注品であった), 完全に尖らせた鉛筆が 20 本, 真空管式アンプのオーディオ装置からはクラシック音楽, 空調はガンガンかける. そして窓を開け放つのである.

いきなりキーボードに触らず, じっと考えて, 手でプログラムを心静かに (音的には静かでないようだが) したためる. これは「プログラミング道」ともいえよう.

その一方で, 某和田英一先生のプログラミングもすごい. もちろん昔話であるが, 混み合った通勤電車の中でもデバグできるように, ラインプリンタ用紙 (といっても今時の若い人は知らないかもしれないが, A3 程度の大きさのページがミシン目でつながった連続紙) に, プログラムの改行のインデントを全部取り除いてコンパクトに印刷していたという.

これは劣悪な環境であっても, 心の目で見ればバグが透けて見えるという極意のプログラミングである.

こういった武士道丸出しのプログラミングもあるが, 私がおつき合いしたプログラミングの天才の中でも飛び抜けているソフトイーサの登大遊君は誰も真似ができないスタイルだ. これは本人の弁だが, プログラミングに集中するとトランズ状態に入り, そうなるとプログラムが自然に頭から湧き出してくるというのだ. まさに Mozart 式プログラミングだ. そんなのあり得ないとも思うが, 彼は未踏ユースの次の年, SoftEther を全面的に作り直した際, 1 年間で 11 万行のプログラムを書いた実績があるのだ.

トランズ状態でのプログラミングと聞くと, 鶴の恩返しという民話を思い出す. 鶴の姿でプログラミングをしているのはきっとかなり美しいと思うが, やはりきっと見てはならない, あるいは見ることのできないものなのだろう.

美しい, とは無関係だろうが, せっかくなので私のプログラミングについてちょっと言及しておく. 私はプログラムは Lisp とマイクロコード以外では書かない. 中途半端は要らないのだ. Lisp のときは, Lisp 自身がわかりやすいのでドキュメントは極小である. ただし, マイクロコードのときは, 少なくとも 20 年近く前の経験では, コードよりもはるかに大量の (コメントを含む) ドキュメントを用意する. 特に, 私としてはいまだに自慢ができる実時間ゴミ集めのプログラムのときは大量の論理式も書いた.

その当時, かなり大きなシステムをつくっていたのであるが, いま振り返っても, 変なやり方だった. まず, アルが入ってから, 一気呵成にマイクロコードを書く. それは実は S 式で書くマイクロコードだったので, プラトンのパロディではないが, 乱飲乱舞のための言語だった. 翌朝, ロールプレイングゲームの乗りで, 机上デバグ, というか書き直しを楽しむ. こういうのを弁証法というのだろうか. いや, きっと弁償法だ. 一粒で二度美味しいプログラミングとも言えよう. アリストテレスは言った. プログラミングとは, 閑暇における自由人の高尚の享楽なのだ. やっぱり, 昔の偉い人はいいことを言う.

そうやっても難物のバグは残る. 私の経験では深夜に及ぶデバグの脂汗と次の日の昼休みのサッカー汗をともに洗い流して「美しい体」に戻すシャワーが最適デバッガの 1 つであった. 「あのハチャニタゴロレ (16 進数で 8C2A560 のこと) がいかんのだ!」とアルキメデスよろしくシャワー室で雄叫びを上げるのだ. プロシン発表の後日, 当日の Twitter を眺めていたら, たしかにシャワーや風呂がいいデバッガだという意見があった.

美しいとはまったく無関係だろうが, このやり方は “Oh, c’est beau!” と言ってもらえるかもしれない.

6 プログラミングのセンス

佐々木の解説書は一般人に馴染みの深い言葉をうまく論って, 美学の本質を垣間見せようとする. 「センスの話」という章もそうだ. センスとは感性とほぼ等しい意味の言葉である. 「美学」という名称の学問の始まりは実は意外に遅い (それまでは哲学の一部門だった). ドイツの思想家 A. G. Baumgarten は, 可知的なもの, すなわち上位能力によって認識されるものは論理学の対象であり, 可感的なものは感性の学 (aesthetica) としての美学の対象である, と定義した.

しかし, 佐々木は感性を肉体レベルの下位能力とはせず, 一歩進んで, 感性とはメタファーとしての感覚, すな

わち決して感覚ではなく, 精神の働きなのだが, 感覚的な働き方をする精神であると言う. つまり, 藝術は感覚を非身体的に用いるのだ.

こう聞くと, プログラミングのセンスとはコンピュータを非機械的に扱うことができることかとも思う. これの正確な意味はともかく, コンピュータの中に単なる冷たい機械的論理しか感じられないようだと, プログラミングのセンスは働きにくいだろう.

ところで, プログラミングという概念自身についても, ちょっと議論しておく必要がある. 閑暇における自由人の高尚の享楽となるためには, 問題をつくるところからプログラミングが始まると思ったほうがよい. 要求仕様が決まってから受注するようなプログラミングには「美」につながる創作活動は関与しにくいだろう.

今年, 情報処理学会が GREE と共催で始めた SamurAI Coding という国際的な (主に学生を対象とする) プログラミングコンテストでは, 多チーム同時対戦のゲームのプログラムの「知的能力」を競う. いろいろなスケジュールが切羽詰まってきたので, それまでの準備をベースに東大の近山隆先生と私の 2 人で実質 3 日程度で一気呵成にゲームのルールをつくってしまった. これは一見プログラミングとは言えないかもしれないが, 広い意味ではプログラミングだったと思う. ルールの妥当性を検証するために, 近山さんは大学内の会議中にサカサカとプログラムを書いて, ある事象の発生確率を調べてくれた. いわゆるゲームバランスの調整にはこういった作業が必要になる. これとて「ゲームのルールを制定する」という発注を受けたことになるのだろうが,「何か面白いゲームのルール」という発注だったから, 閑暇ではなかったが, 自由人の高尚な享楽であったことに間違いない.

こんなのもプログラミングという意味では, 私は, 同名のタイトルの発表を 2011 年の夏のプログラミングシンポジウムで行なっている. 目次だけ紹介しておくと,

1. カレンダープログラム
   プログラミングの大本に立ち返る
2. 確率ゼロの例外に対処するプログラミング
3. 仕様の変更への対処も重要なプログラミング
4. 外部仕様に基づくプログラミング
   関数方程式

である. このうち,「外部仕様に基づくプログラミング」の外部仕様をちょっと紹介しておこう.

有理数から有理数への関数 f で

$$f(f(f(f(x)))) = x^2$$

かつ

$x \to 0$ のとき $f(x) \to \infty$ (x は有理数の範囲)

となるようなものは存在するか? 存在しないならその証明を, 存在するなら具体的にそのような関数 f を書け.

これは私が 2011 年に『数学セミナー』という雑誌の「エレガントな解答を求む」に出したオリジナルの問題であるが, 寄せられた解答を見て印象に残ったことが 2 つあった.

まずは, **仕様を正しく理解するべし**. この問題で不正解が多かったのは, 有理数から有理数への関数を有理関数 (つまり, 分子分母が x の整数係数多項式になっている関数) と誤解した人が多かったことによる. もし, そういう意味なのであれば, 題意を満たす関数が存在しないことは容易に証明できる. もう 1 つ割と多かったのは, 実数から実数への関数をそのまま使ってしまった解答だった.

$$f(x) = 1/|x|^{\sqrt[4]{2}} \quad (x \neq 0), \quad f(0) = 0$$

は確かにそれらしい, かつ美しそうな解答なのではあるが, 有理数 x に対して $f(x)$ が必ずしも有理数にならないのが致命的である. こういう落し穴に引っかかってはいけない. 仕様を正しく理解しなければ「美しい」の門前にもたどりつけない.

もう 1 つは, 同じ外部仕様からプログラムをつくっても, **実行性能は千差万別**ということ. 実際に f をプログラムの手順として記述しようとした際, つまり, 存在を納得させるだけではなく, 関数を構成しようとした場合, その記述の複雑さと実行性能が大いに異なってくる. これはアルゴリズムの教科書ではいやというほど学ぶことだが, いざ具体的な問題が与えられると, うっかりと忘れがちである. 記述性能と速度性能がよくてこそ初めて「美しい」と言えるのだ.

7 プログラムの美学

これは前に述べたように記述 (プログラム) の美学である. とすると, 速度性能もさることながら, 記述のエレガンス (これは記述の短さに直結する) が考慮の対象となる.

ところが, そもそもプログラミングに関してはそもそもいろいろなこと, つまり仕様や問題そのものが短く書けないという問題がある. 数学はその点, よく枯れている. 概念形成と語彙がしっかりしているのだ. 少し昔の東大の入試に「円周率が 3.05 より大きいことを証明せよ」という 1 行で済む問題が出た. こういう芸当はプログラミングに関する試験では至難の業だ. 私はかなり前に大学入試センターの情報関係基礎という, ふつうの受験生は多分知らないような問題の作成に関わったことがある. そのとき, 1 時間程度で解かないといけない入試問題なのに, 十数ページにわたってぎっしりと文章や図を書かないといけないことにショックを受けたものだ.

そもそも情報関係基礎の教科書が職業高校の種別によって相当に異なることや, 教科書の記述が, マウスイヤーで変遷する情報技術の発展, というより若年層への普及にまったく追いついていないことも問題だった. その後, 私は某大学院入試のプログラミングという科目の問題に関わったことがあるが, 多少情報科学について学んでいるはずの学生を相手にしても, 問題の記述はどうしても長々となってしまう.

だから, プログラムが短く書けるということは, プログラム自身が実は豊かな情報量・記述量比をもって記述できるということだと私は常々思っている.

1980 年ごろのことだったと記憶するが, ACM の SIGPLAN で, プログラムのインデンテーションについての議論がしばらく盛り上がったことがある. 議論はいま思うに, まさに「インデントの美学」だった. これについては (もう絶版だと思うが) Tsinky というグループが著した『プログラミング・セミナー』(共立出版, 1985) の中の「凸凹プログラム」という章に私は駄文を書いた. 同じ Pascal プログラムについて, これをどうインデントするかについて数名の人々が熱く語っていたのを整理して紹介したのである. 面白かったのは, 最後になって, インデントの情報があれば, いわゆる **begin end** なんか要らないという, いわゆるインデント文法論になったことである. 見た目の美しさ (この場合, わかりやすさと同値?) を追求していったら, 一皮剥けたというか, ワンレベルアップの議論に発展していった面白い例だと思う.

理論の裏付けがあるかどうかをプログラムの美の基準として採用するという考えもあろう. もっともその場合は理論が美しいかどうかというほうに議論がシフトする. 数学や物理学の理論を美しいと評価することは昔からあったように思うが, なぜか美学の対象にはなってこなかったようだ. 美学者には難しすぎたのだろう.

8 プログラムの短さ (エレガンス) について

以上のような抽象的で実体のないことばかり述べていてもインパクトはない. なので, 最近私が書いたプログラムで, これは美のセンスに合致するかもしれないという例を 1 つだけ紹介しよう.

私は現在, 早稲田大学に籍を置いているが, 実際には JICA が創立を支援しているエジプト日本科学技術大学において, プログラミングに関する講義や学生指導を担当している (これに関するエピソードを現在情報処理学会誌に連載中である). 講義は半期 30 コマ, 週 2 回行なう. 後半は Peter van Roy と Seif Haridi の “Concepts, Techniques, and Models of Computer Programming” (The MIT Press, 2004) を教科書として, ほぼそれに沿うが, 前半の 1ヵ月半は, 各種の基本的な計算メカニズムを紹介している. これらは理論というより, さまざまなプログラミング技法というか, プログラミングに関する頭の体操として教えている. 具体的には

- Turing マシン
- Ritchie の `while` プログラム
- Markov アルゴリズム
- λ 計算
- 導出原理 (resolution principle)

の 5 種類である. これらの基本メカニズムをそれぞれ実際に 1 進法の加算のプログラムが書けるところまで, さっと駆け足で紹介するのだ.

Ritchie は UNIX のあの Dennis Ritchie である. 彼の創案した `while` プログラムは 5 種類の文からなる極めて単純な言語である. 変数については, 0 に初期化するか, 別の変数の値をコピーするか, 値を 1 増やすしかできないが, それでも減算ができるのが面白い. Markov アルゴリズムは会場で訊いたらほんの数名しかご存知なかったようだが, 一種の項書き換えシステムで, エキスパートシステムのプロダクションルールの魁を提唱したものである. ちなみにこの Markov は A. A. Markov, 確率論で有名な Markov も A. A. Markov. あれっと思って調べたら, どちらも Andrey Andreyevich Markov だ. まさかと思ったら, この 2 人は親子だった.

λ 計算は数値すら含まない最も原理的な形を紹介した. つまり, 数値も λ 項として定義するやり方である. まず, $[M,N]{\equiv}\lambda z.zMN$ という記法を導入して, $\overline{0}{\equiv}\lambda x.x{\equiv}\mathbf{I}$ (ここで $\mathbf{I}$ は恒等写像のコンビネータ), $\overline{n+1}{\equiv}[\mathbf{F},\overline{n}]$ といった具合に定義していく (ここで $\mathbf{F}$ は 2 引数のうちの後者を返すコンビネータで, 偽を表わすのに使われる). ここから始めると, 単純な足し算を定義するのも立派なパズルになる. 付録 1 に講義スライドの内容の一部を示した. この数値表現に対して, successor 関数 ($\mathbf{U}$) や predecessor 関数 ($\mathbf{D}$), zerop 述語 ($\mathbf{Z}$), if 式 ($\mathbf{?}$) などをコンビネータとして定義していくわけだ. カッコ内に書いたのは, 以下で使うコンビネータの記号である.

さて, これらをスライドや白板だけで説明しても迫力がない. そのため, 実際に動かすためのインタプリタを自分で書いた. これは学生のレポートの採点とフィードバックのためにも必要だった.

上に述べたように, こんなのはさかさかと Lisp で書くに限る. 仲間たちと一緒に昔開発した TAO (いまは DAO と呼んでいる) のエミュレータが UNIX の中で動いているので, それを利用した. それぞれの計算メカニズムは独自の文法をもっているが, 面倒なのでそのあたりは可能なかぎり S 式に変換して, それで入出力してしまうことにする. こうして作成したインタプリタの行数 (コメントだけの行はカウントしない) は下記のとおり

である．なお, 導出原理は Web から簡単に入手できる Prolog 処理系にお任せすることにした．

計算メカニズム	行数
Turing マシン	53
`while` プログラム	16
Markov アルゴリズム	13
λ 計算	253

Turing マシンは 4 組タイプで, ステップ実行しながらテープ表現を見ることができるものである．`while` プログラムも Markov アルゴリズムも, 入出力をサボって, S 式そのもので書く．だから, やたらと行数が少ない．S 式にしてしまうとメカニズム自体は非常に簡単に書けてしまうのだ．

λ 計算インタプリタはちょっと長いが, このうちコンビネータや $[M, N]$ の入出力が 112 行と半分近くを占める．それでも長いと言えるが, 変数名の変換 (α 変換) や β 縮約などはそれなりに複雑な処理を必要とする．この λ 計算インタプリタでは β 縮約可能な, いわゆる β-redex を対話的に選択して計算を進める．講義に間に合わせる即席だったので, 全体として「美しいか?」と言われると自信はないが, 機能のわりには短く書けたと思う．

λ 計算の入出力を S 式で行なうのはさすがに辛い．それに上記のような数値の定義もそのままの表記で行ないたい． しかし, 内部表現はあくまでも S 式である．$\lambda x_1 \cdots x_n.M_1 \cdots M_m$ は `((& x1 ... xn) M1 ... Mm)` と内部表現する．`&` は苦しまぎれの λ 代用品である．

足し算を再帰関数として定義するために, いまや西海岸にもある有名な **Y** コンビネータ † を利用するが, これは同値の λ 項を計算するだけなので, 付録 1 のスライドにある Θ コンビネータ $\Theta \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))$ を使う．こうすると足し算は

$$\Theta(\lambda fxy.?(\mathbf{Z}x)y(f(\mathbf{D}x)(\mathbf{U}y)))$$

と書ける．なお, Θ コンビネータは **H** で表わすことにする．こうして, このインタプリタに

```
(H(&fxy.?(Zx)y(f(Dx)(Uy)))[F,I][F,[F,I]])
```

という式を与えると $1+2=3$, つまり, `[F,[F,[F,I]]]` が求まるわけである．ただし, β 縮約のための redex を対話的に適切に選んでいく必要がある．

前置きが長くなってしまったが, この λ 計算インタプリタを書いていて,「ああ, これはエレガントに書けた!」というのが, コンビネータの出力である．たとえば, 内部表現で

```
((& p q) q ((& x y) y) p)
```

となっているものを successor 関数を表わす **U** (これは Up から来ている) 1 文字に変換して出力するのに, 図 1 に示す関数を使った．パターンマッチで定義済みのコンビネータかどうかをチェックしている．

```
(defun combinator-p (lmd)
    (cond
     ((let (_x) (== ,lmd ((& _x) _x))) 'I)
     ((let (_x _y) (== ,lmd ((& _x _y) _x))) 'T)
     ((let (_x _y) (== ,lmd ((& _x _y) _y))) 'F)
     ((let (_x _y _z) (== ,lmd ((& _x _y _z) _x _y _z))) '?)
     ((let (_x _u _v _z) (== ,lmd ((& _x _z) _z ((& _u _v) _v) _x))) 'U)
     ((let (_x _u _v) (== ,lmd ((& _x) _x ((& _u _v) _v)))) 'D)
     ((let (_x _u _v) (== ,lmd ((& _x) _x ((& _u _v) _u)))) 'Z)
     ((let (_x _y _z) (== ,lmd ((& _x _y _z) _x _z (_y _z)))) 'S)
     ((let (_x _y _z) (== ,lmd ((& _x _y _z) _x (_y _z)))) 'B)
     ((let (_x _y _z) (== ,lmd ((& _x _y _z) _x _z _y))) 'C)
     ((let (_f _x) (== ,lmd ((& _f) ((& _x) _f (_x _x)) ((& _x) _f (_x _x))))) 'Y)
     ))
```

図 1. 内部表現をコンビネータ記号に変換するプログラム

ここで, アンダースコアのついた変数は Prolog 的な論理変数で, == はユニフィケーションである．成功したら真値を返す．これがコンビネータの定義をそのまま翻訳しただけのプログラムになっているところが肝である．こんなに簡潔に, しかも直接的に書けるというのは, さすがユニフィケーションの威力だと, このとき感じ入った．私は Prolog はあまり趣味ではないが, ユニフィケーショ

† 成功した人たちが次に成功する人を生み出していくという意味で実に適切なネーミングである．こういう面白さを鑑賞するためには知的背景が必要という面白い例でもある．

ンは好きだ. なお, 肝心の Θ, つまり **H** コンビネータがここに出てこないのは, これが 2 つの λ 項が並んだコンビネータであるために, 別の形のパターンマッチが必要だからである (それは省略).

ちなみに, さきほどの $1+2$ の計算の実行を画面コピーした学生に配ったものを付録 2 に示しておく. 発表当日にデモしたものを少しだけ見やすくして印刷したものである.

9 塑像的 vs 彫像的プログラム, および結び

はこだて未来大学の木村健一さんは, 東京藝術大学で塑像を学んだ人である. 彼から聞いた面白い話がある. 造形には塑像と彫像がある. 塑像屋は彫像屋に対して, 「あんたらの作品には骨がない」と悪口を言う. 塑像屋は, まず人体の骨格をつくり, それに肉となる粘土をくっつけていく. 彫像屋は塊を削っていくわけだから, どうしても骨格がずれやすい. 逆に彫像屋は塑像屋に対して, 「あんたらは, つけたりはがしたり, 一発勝負に賭ける気迫が足りない」と言う. これもなるほどである.

これをプログラミングに敷衍すると, 塑像的プログラムというのが世の中には多そうだ. つまり, モジュールをくっつけてはくっつけていくやり方で出来たプログラムだ. 現実的なのだろうが, 建増し建増しの温泉旅館的な構造になりやすい. 逆説的だが, そのうち骨も怪しくなる. これに対して彫像的プログラムとは, なにやら無駄を削っていった究極の美的プログラムにつながるようにも思える. もっとも, そんなプログラムはなかなか書けないような気がする.

というところまで書いてきて, 本稿自身が実は悪い意味で塑像的であることに気がついた. 読み返すと, 建増し温泉旅館的構造をしている. さらには竜頭蛇尾, いや鵺頭蛇尾ですらある. このへんで結んでしまうのが吉である.

なお, 参考文献等はすべて文中に書いた.

[質疑応答]

小川 (名古屋市工業技術研究所): 発表の中で「受注では美の創作活動は起こりにくい」という話があったが, 旧来, 音楽も美術も貴族からの発注ではなかったのか.

竹内: その場合の発注は仕様の詳細のない, きわめて曖昧模糊としたもので, この楽器編成でつくってくれとか, 今度の宮中晩餐会に相応しいのをつくってくれというようなものだった. だから創作の十分な余地があったと思う. もちろん, 詳細な仕様の決まった給与計算システムの発注を受けて, 外部仕様はともかく, 内部的に非常に「美しい」プログラムを書いたら, その人は偉いなぁと思う.

落合: 竹内さんのプログラミングに「享楽」とあったが, それと美学の関係がよくわからない.

竹内: アリストテレスの音楽に関する哲学の中での「閑暇における自由人の高尚な享楽」という文脈で「享楽」という言葉を使った. 実際に私のプログラミングスタイルが一般的な意味で「享楽」と言えるかどうかは微妙だ. 夜を徹してドラクエをやるのは, 苦痛が享楽と実は相通ずることを示唆している. 私もデバグなどではかなり消耗することがある. それでもそれを享楽だと思えるようになれば, 一皮剥けることになるのではないだろうか.

河内谷 (IBM): 藝術作品と異なり, ユーザにとっては出来上がったプログラムの内部構造が美しいかどうかというのは見えないし, 判断のしようがない. プログラムと藝術を同じ土俵で論じるのは難しいのではないか.

竹内: たしかにそうとは言える. しかし, 例えば音楽の場合, 楽譜がプログラムであり, 実際にはそれを演奏家が演奏しないと鑑賞できない. プログラムも実行したら, たとえばその UI がいいとか悪いとか言える. その UI からプログラムの美しさにダイレクトにつながらないのは困ったことではあるので, 仰るとおりかもしれない. [補: だからこそ, プログラムの美学には, プログラミングに関する深い知的背景が必要になる. それが現代以降の美学のココロであろう.]

付録 1　講義スライドの一部

Combinator: λ term which has no free variable.

$\mathbf{I}\equiv\lambda x.x$
$\mathbf{K}\equiv\lambda xy.x\equiv\mathbf{T}$
$\mathbf{F}\equiv\lambda xy.y$
$\mathbf{S}\equiv\lambda xyz.xz(yz)$
$\mathbf{B}\equiv\lambda xyz.x(yz)$
$\mathbf{C}\equiv\lambda xyz.xzy$
$\mathbf{Y}\equiv\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
$\Omega\equiv(\lambda x.xx)(\lambda x.xx)$

[ex] How is $\mathbf{SKK}\equiv(\lambda xyz.xz(yz))(\lambda xy.x)(\lambda xy.x)$ transformed by β reduction?

Now we are ready to write an addition program!

We define λ terms corresponding to non-negative numbers:

$[M,N]\equiv\lambda z.zMN$
$\overline{0}\equiv\lambda x.x\equiv\mathbf{I}$
$\overline{n+1}\equiv[\mathbf{F},\overline{n}]$
(We may write n instead of $\overline{n}$ for brevity.)

[ex] Examine succ$\equiv\lambda x.[\mathbf{F},x]$ is a successor function.
[ex] Examine pred$\equiv\lambda x.x\mathbf{F}$ is a predecessor function except for pred $\overline{0}$.
[ex] Examine zerop$\equiv\lambda x.x\mathbf{T}$ is a predicate that becomes $\mathbf{T}$ for $\overline{0}$, and become $\mathbf{F}$ otherwise.
[ex] Examine if$\equiv\lambda xyz.xyz$ means **if** x **then** y **else** z.

How about addition? The idea is to define "add" as

add x y $\sim$
if (zerop x) **then** y **else** (add (pred x) (succ y))

Remaining problem is how to define this recursive function only by using λ term.

[**Fixed Point Theorem**] For any λ term G, there is a λ term X such that $GX=X$.

[proof] Let $W\equiv\lambda x.G(xx)$. Since $WW\longrightarrow G(WW)$, we can choose WW as X. It is easy to examine

$$X\equiv WW\equiv(\lambda x.G(xx))W\longrightarrow G(WW)\equiv GX$$

QED.

This theorem says there must be a fixed point X for any "mapping" G.

For $\mathbf{Y}\equiv\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, $\mathbf{Y}G=G(\mathbf{Y}G)$; that is, $\mathbf{Y}$ combinator is a function to get a fixed point of any λ term G. This is an important scaffolding for defining recursive functions.

For brevity, we denote add as h. We want make the following equality:

$hxy=$ if(zerop x)y(h(pred x)(succ y))

Abstracting both side by x and y, we get

$h=\lambda xy.$if(zerop x)y(h(pred x)(succ y))

Further abstracting the righthand side by h (renamed to f) and giving it h as an argument, we get

$h=\underline{\lambda fxy.\text{if (zerop } x)y(f(\text{pred } x)(\text{succ } y))}h$

and let G be the underlined λ term. This is of the form $h=Gh$. By the fixed point theorem, we can obtain an h that satisfies the equation. Namely $h=\mathbf{Y}G$.

Finally we get
add $\equiv\mathbf{Y}(\lambda fxy.$if(zerop x)y(f(pred x)(succ y))).

Let us examine whether this add is really an addition:

add 1 2
$\equiv\mathbf{Y}(\lambda fxy.$if(zerop x)y(f(pred x)(succ y))) 1 2
$=(\lambda fxy.$if(zerop x)y(f(pred x)(succ y))) add 1 2
(because $\mathbf{Y}G=G(\mathbf{Y}G)$)
$\longrightarrow(\lambda xy.$if(zerop x)y(add(pred x)(succ y))) 1 2
$\longrightarrow^*$ if(zerop 1)2(add(pred 1)(succ 2))
$\longrightarrow^*$ if $\mathbf{F}$ 2(add(pred 1)(succ 2))
$\longrightarrow^*$ add(pred 1)(succ 2)
$\longrightarrow^*$ add 0 3
$\equiv\mathbf{Y}(\lambda fxy.$if(zerop x)y(f(pred x)(succ y))) 0 3
$=(\lambda fxy.$if(zerop x)y(f(pred x)(succ y))) add 0 3
$\longrightarrow(\lambda xy.$if(zerop x)y(add(pred x)(succ y))) 0 3
$\longrightarrow^*$ if(zerop 0)3(add(pred 0)(succ 3))
$\longrightarrow^*$ if $\mathbf{T}$ 3 (add(pred 0)(succ 3))
$\longrightarrow^*$ 3

$\mathbf{Y}G=G(\mathbf{Y}G)=G(G(\mathbf{Y}G))=G(G(G(\mathbf{Y}G)))=\ldots$ shows that $\mathbf{Y}$ is a magical device to generate G by $\mathbf{Y}G$. But note that it creates G as an equivalent λ term. However, there is a combinator Θ which really creates G by β reduction such that $\Theta G\longrightarrow^* G(\Theta G)$ [Of course, $\Theta G=G(\Theta G)$].

$\Theta\equiv(\lambda xy.y(xxy))(\lambda xy.y(xxy))$

[ex] Examine the above.

This is the **principle of recursive call**.

Note that from only application expression and abstraction expression, we can construct arithmetics without any number constants! Not only theoretically interesting, it is also attractive from programming technique viewpoint.

付録 2 λ計算インタプリタの実行結果

Here, `H` is Theta combinator `?` is if combinator
`Z` is zerop combinator `D` is pred combinator (Down)
`U` is succ combinator (Up)

We now compute `add 1 2`, or `add [F,I] [F,[F,I]]`, where `add` is defined as `H(&fxy.?(Zx)y(f(Dx)(Uy)))`, thus, `add 1 2` is `H(&fxy.?(Zx)y(f(Dx)(Uy))) 1 2`.

At first, there are five redexes:

`H(&fxy.?(Zx)y(f(Dx)(Uy)))`, `?(Zx)y(f(Dx)(Uy))`, `Zx`, `Dx`, `Uy`

In the following, chosen redex is underscored. And for easy reading, blanks are inserted between sub-lambda expression if adequate.

```
H(&fxy.?(Zx)y(f(Dx)(Uy))) [F,I] [F,[F,I]]
-------------------------
(&fxy.?(Zx)y(f(Dx)(Uy))) (H(&fxy.?(Zx)y(f(Dx)(Uy)))) [F,I] [F,[F,I]]
-----------------------------------------------------------------
? (Z[F,I]) [F,[F,I]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(D[F,I])(U[F,[F,I]]))
   ------
? ([F,I]T) [F,[F,I]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(D[F,I])(U[F,[F,I]]))
   ------
? (TFI) [F,[F,I]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(D[F,I])(U[F,[F,I]]))
   ---
? F [F,[F,I]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(D[F,I])(U[F,[F,I]]))
-------------------------------------------------------------
F [F,[F,I]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(D[F,I])(U[F,[F,I]]))
-----------------------------------------------------------
H(&fxy.?(Zx)y(f(Dx)(Uy))) (D[F,I]) (U[F,[F,I]])
                          ------
H(&fxy.?(Zx)y(f(Dx)(Uy))) ([F,I]F) (U[F,[F,I]])
                          ------
H(&fxy.?(Zx)y(f(Dx)(Uy))) (FFI) (U[F,[F,I]])
                           ---
H(&fxy.?(Zx)y(f(Dx)(Uy))) I (U[F,[F,I]])
                            ----------
H(&fxy.?(Zx)y(f(Dx)(Uy))) I [F,[F,[F,I]]]
-------------------------
(&fxy.?(Zx)y(f(Dx)(Uy))) (H(&fxy.?(Zx)y(f(Dx)(Uy)))) I [F,[F,[F,I]]]
-------------------------------------------------------------------
? (ZI) [F,[F,[F,I]]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(DI)(U[F,[F,[F,I]]]))
   --
? (IT) [F,[F,[F,I]]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(DI)(U[F,[F,[F,I]]]))
   --
? T [F,[F,[F,I]]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(DI)(U[F,[F,[F,I]]]))
-----------------------------------------------------------------
T [F,[F,[F,I]]] ((H(&fxy.?(Zx)y(f(Dx)(Uy))))(DI)(U[F,[F,[F,I]]]))
---------------------------------------------------------------

[F,[F,[F,I]]]

beta-normal-form!
```