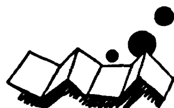
解 説

2. プログラミング・パラダイムと環境



2.1 マルチパラダイム支援環境[†]

竹 内 郁 雄[††]

1. はじめに

気軽に（でもなかったのだが）この解説を引き受けて後悔している．マルチパラダイム支援環境は現在ほとんど未到の研究課題であって，「解説」されるような筋合のものではないからである．だから，ここでは「解説」というより，マルチパラダイム支援環境に関する「雑感」を述べることになろう．

2. マルチパラダイム

プログラミング・パラダイムはおおまかにいって，プログラミングスタイルやプログラミング技法を指す．しかし，これらはよく特定のプログラミング言語と結びつけられ，プログラミング・パラダイムとプログラミング言語がほとんど同一視されることもある．たとえば，Lisp はその独特の個性から，よく言語そのものがパラダイムと呼ばれる．Prolog や Smalltalk もそういった部類だろう．Prolog は「論理型パラダイム」，Smalltlalk は「オブジェクト指向パラダイム」とよく同一視される．しかし，Lisp のパラダイムにはそれを一言でうまく表現するパラダイム名がない．「関数型パラダイム」も「手続き型パラダイム」も間違いではないが的外れである．

パラダイムという言葉の指すレベルはまちまちで，再帰法とループ，束縛と代入などといった比較的低レベルのものも（対立する）パラダイムだとする捉え方がある．しかし，一般にパラダイムとして通用しているのは，上記のほか，「構造化プログラミング」，「制約プログラミング」，「並行プログラミング」，「デーモン（データ指向というのはこれの一部だろう）」，「プロダクションルール（ルール指向と呼ぶこともある）」，「視覚的プログラミング」といったものである．これらのうち，「構造化プログラミング」や「並行プログラミング」などは言語に強く依存しないタイプのパラダイムであり，プロダクションルールはパラダイムと言語が非常に近いものである．（本特集では「日本語プログラミング」もパラダイムのところに分類されているがこれはいかがなものであろうか？）

このようなパラダイムがそれぞれ固有の支援環境をもつことは想像に難くない．たとえば，デバッギングのためのツールはパラダイムやプログラミング言語ごとに大きく異なる性格をもつ．それは本特集の第1部門の解説からも窺えるはずである．

では，これらのパラダイムを融合あるいは統合したような言語や環境，あるいはこれらのパラダイムを接合したような環境ではどのようなことが問題になるか？　これが本解説に与えられた課題であろう．

いま「融合・統合」と「接合」という言葉を使ったが，マルチパラダイム支援環境といったときに，この差は微妙であると同時に本質的である．筆者はマルチパラダイム支援環境という言葉には，二つの意味があると思う．一つは，複数のパラダイムが一つの言語の中に融合されている場合であり，もう一つは異なるパラダイムに対応した言語を複数個つないで使う場合である（図-1）．後者は多言語環境とも呼べるものである（これをマルチパラダイムと呼ぶかどうか議論があるかもしれない）．このどちらを考えるかで，支援環境に要求される技術的課題はかなり異なる．ここでは前者，すなわちマルチパラダイム言語の支援環境について考える．

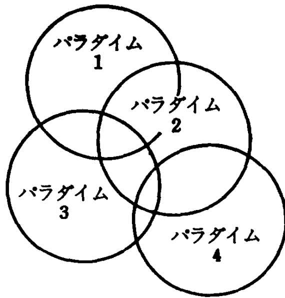
3. マルチパラダイム言語

なにをパラダイムと捉えるかで，マルチパラダイムの判定基準が変わるが，最初にはっきりとマルチパラダイムを謳ったのは Xerox の InterLisp の上に開発された LOOPS であろう[1]．LOOPS は4つのパラダイムを融合したと標榜している（図-2）．すなわち，

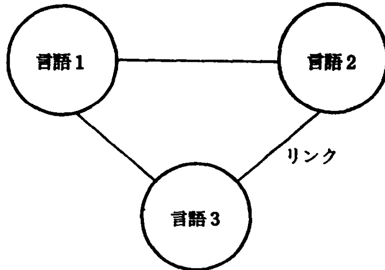
---

† Multiple Paradigm Programming Environment by Ikuo TAKEUCHI (NTT Software Laboratories).

†† NTT ソフトウェア研究所



(a) マルチパラダイム言語環境



(b) 多言語環境

図-1 マルチパラダイム支援環境の二つの意味

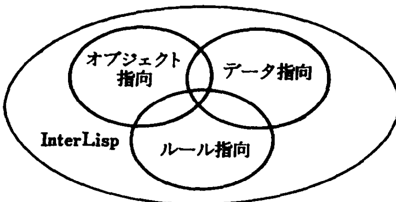


図-2 LOOPS

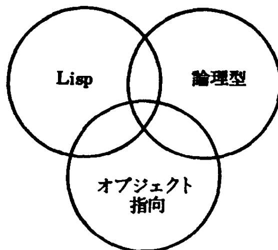


図-3 TAO

手続き型パラダイム，オブジェクト指向パラダイム，データ指向パラダイム，ルール指向パラダイムを InterLisp という Lisp 環境のもとで融合した．われわれが ELIS という Lisp マシンの上に作った TAO という言語[2]は「公式」には三つのパラダイム，すなわち Lisp パラダイム（とりあえずこう呼ぼう），論理型パラダイム，オブジェクト指向パラダイムの三つを融合したことになっている（図-3）．実際にはこのほかに並行プログラミング，Fortran のような番地型計算などもあるから，これをパラダイムとして勘定すれば相当のマルチぶりである．どちらも Lisp をベースにしていることに注意しよう（ただし，実現上，LOOPS は Lisp の「上」に乗っており，TAO はほかのパラダイムと Lisp が横並びに近いという差はある）．新しい言語の実験を行なうのに Lisp をベースにするのが一番やりやすいという事情はここ 30 年間変化していない（これは Lisp がある意味で「メタな環境」を与えていることの証である）．

MIT の Lisp マシンの上に開発された ZetaLisp[3]には，Lisp のほかに Flavors と呼ばれるオブジェクト指向システムがあり，並行プロセスの概念もあった．また，ICOT が PSI という Prolog マシン上に作った ESP[4] も論理型とオブジェクト指向を融合したという意味で，マルチパラダイムと呼べるかもしれない．Common Lisp も言語の中に CLOS[5] というオブジェクト指向システムを取り込もうとしている．しかし，これらの言語をマルチパラダイム言語と呼ぶのに異議を唱える人もいるだろう．それは著者が別の解説[6]でも述べたように，オブジェクト指向が基本的にプログラムの構造化に関する次元のパラダイムであるのに対して，論理型や関数型が計算メカニズムに関する次元のパラダイムであることによる．ESP のような場合，オブジェクト指向は言語の骨格となって論理型を支えているので，パラダイムの取捨選択というマルチパラダイム特有の問題が起こりにくい（と著者は想像する）．もちろん，オブジェクト指向もメッセージ伝達という計算メカニズムにだけ注目すれば，関数型や論理型と横並びのパラダイムとなる．

マルチパラダイムの本来の狙いは，プログラミングの幅，つまり，プログラミングにおける選択の多様性を広げることである．対象としている問題に応じて，それに適した言語あるいはパラダイムがあるが，どんな問題にも万能的に向いた言語やパラダイムがないというのは，経験的な事実である．従来は，問題が与えられたとき，その問題の主要部分にもっとも役に立ちそうな言語を選択してプログラムを書いた．その際，主要でない部分の書きにくさには目をつぶったわけである．しかし，扱う問題が大きくかつ複雑になるにつれ，主要部分が複数化し，それぞれが異なるパラダイム（ここでは，言語特性といってもいいかもしれな

い）を要求するようになってきた．

たとえば，画像理解といったシステムでは，画像の一次処理のための強力な数値計算能力と知能処理のための記号処理能力の比較的密な協調が要求される．その知能処理をよくみると，その中にいくつもの異質な「主要部分」がある．それはある意味で当たり前で，もともと知能自身が単純なパラダイムで割り切れるものではないからである．実際，人間の知能の問題は，一つのモデル化が一つのパラダイムに対応するといってよく，これは経済学のような社会科学と事情が似ている．

また，知能処理といった難しい問題を持ち出さなくても，ひと昔前に比べて，われわれの書くプログラムの複雑性は確実に増大している．プログラミング言語はそれを後追いしている．たとえば，昔は文字列処理言語というものが存在した．それは当時のふつうの言語が文字列の処理に弱かったからである．ところが，今日の言語は（少なくとも最近設計されたものは）文字列処理機能が強化されていて当然である．これは一見データ構造の強化の話にすぎないようにみえるが，立派にパラダイムの話題である．昨今のプログラミング言語のデータ構造の豊かさは，実は（データ構造レベルでの）マルチパラダイムの走りであった．実際，画面エディタのためのテキストデータの内部構造の設計の問題を考えてみればよい．一つの言語に用意されたデータ構造の豊かさゆえに，これをどのようにプログラムするかは大いなる「選択」の問題になる．オブジェクト指向とか論理型といった最近のマルチパラダイム論は，いわばこういう流れの現時点における一断面なのだろう．これは，マルチパラダイム支援環境を考察するのに，一つの背景となる．

4. マルチパラダイム言語に要求されるもの

マルチパラダイム支援環境に要求されるものを考える前に，マルチパラダイム言語に要求されるものを考えよう．まず，プログラミング言語としてまっとうなものでなければならない．これは当然だから，ここでは触れない．次に重要なのが性能である．これもプログラミング言語に対しては当然の要求なのだが，マルチパラダイム言語の場合は，パラダイムの間の性能バランスが重要である．

どんなパラダイムでも，書きやすいとか書きにくいといった相対的な記述力の差はあるだろうが，それで書けない問題があるといった絶対的な能力の差はない．だから，パラダイムの選択は，おおむね記述のしやすさ，すなわち対象としている問題に対するマッチングの良さを基準として行なわれる．この基準は，ある程度主観的なものだが，まったく客観性がないというものでもない．

パラダイムの記述力の差は，同じ仕様をもつプログラムを2通りに書いたときのプログラムの長さの差としてもっとも端的に現れる．また，Aのパラダイムで書いたものをBのパラダイムで書き直す際の手間と，Bのパラダイムで書いたものをAのパラダイムで書き直す際の手間の差としてもある程度定量的に測ることができる．この記述力の差と実際の性能差（速度性能とメモリ性能があることに注意）がかけ離れていると，二つのパラダイムを一つの言語の中に混ぜた意味がなくなってしまう．

たとえば，非常に原始的な意味での対立パラダイムであるループと再帰法でこのことを考えればよい．ループと再帰法が両方備わっていても，再帰法が異常に遅い言語では，再帰法はほとんど使われないだろう．これではパラダイム選択の多様性を真に広げたとはいえない．ループより再帰法が記述力では少し上回る程度だから，許される速度差はわずかなものでしかあり得ない（絶対値といわれれば，ドンブリ勘定で1.3〜1.5倍が限度というところか）．

Lispのような関数呼び出し（関数型というより，手続き型）とPrologのようなユニフィケーションではどうだろう．昔から言い伝えられている有名な逸話がある．二つのクラスの一方にはLispを教え，もう一方にはPrologを教えた．約1週間後，Prologのクラスの学生は　単な機械翻訳のプログラムを書けるようになったが，Lispのクラスでは3カ月後にまだパターンマッチのプログラムを書いていた，という話である．これはユニフィケーションの記述力がLispのリスト処理基本関数の記述力をはるかに上回っているということを意味している．このほかにもユニフィケーションでは，計算の双方向性が保証されるという長所がある．だが，これにはちょっとした落し穴がある．

つまり，上のような意味で記述力が優れているからといって，ユニフィケーションが関数呼び出しより10倍遅くていいという言い訳はできない．事実，広範な範囲のリスト処理をプログラムしてみるとユニフィケーションと関数呼び出しの記述力の差は，それほど大きくない．双方向性の長所は実際にはほとんど使われない．根本的に異なる計算メカニズムをもつ両者

だが，実際に計算機にインプリメントすると，ほぼ同等の性能が出る．逆にこのことは，ユニフィケーションがツボにはまるような分野では，まさにユニフィケーションが（1桁を上回る）圧倒的な強みを発揮することを意味している．

だから，ユニフィケーションと関数呼び出しが同程度の性能を保証されていれば，ふつうのリスト処理プログラムを書くときにどちらを選ぶかは，プログラマの任意の選択になり，いくつかの局面ではユニフィケーションを選ぶことがほぼ絶対になる．ところが，よくみられるように，Lisp の上に Prolog を乗せるような形で実現された複合パラダイム言語☆では，結局ユニフィケーションの美味しいところだけが選択の対象になり，プログラミングにおける選択の多様性は本当の意味では広がらない．もちろん，それで十分という考え方もできるが，ユニフィケーションというもののもつ本来の性能を生かしていないことは事実である．

これに対して，LOOPS のようにルール指向と称して，プロダクションシステムを言語の中に取り込んだ場合はやや事情が異なるように思える．プロダクションシステムは関数呼び出しやユニフィケーションのような基本計算メカニズムと同列のものと考えていいかどうか疑問だからである．この場合は，Lisp や Prolog の「上に」乗っていても，美味しいところだけを選択することに意味がある．プロダクションシステムで基本的なリスト処理を行なうことはないだろうからである．もっとも，歴史的にみれば本来の「プロダクションシステム」は計算の理論に出てくるきわめて原理的な計算メカニズムだから，関数呼び出しと同列に考えることも可能かもしれない．

これまでの議論からわかるように，パラダイムにはレベルがあり，パラダイムの性能バランスがとれていないといけないのは，それらのパラダイムが同じレベルにあるときだけである．このことは，オブジェクト指向について考えると浮彫りになる．

オブジェクト指向の基本計算メカニズムはメッセージ伝達である．基本的なデータ型もオブジェクトとして捉えるような複合パラダイム言語を設計する場合，基本データに対する関数呼び出しとメッセージ伝達の性能差を極小にしたい．メッセージ伝達は「操作名」の動的なオーバロードが可能だから，記述力は関数呼び出しに優っている．だから，多少遅くてもいいが，2倍以上も遅いとありがたみが薄れてしまう．しかし，クラス階層の定義やそれに付随する（クラス階層の再構成などの）いろいろな作業は，オブジェクト指向にとって本質的ではあるが，それほどの性能は要求されない（実は，これにも後で述べるように落し穴がある）．意味的には，クラス階層たどりはメッセージ伝達という基本計算メカニズムにとって必要不可欠だが，通常のインプリメントではこれを「仮想化」して実行性能には影響しないようにしている．

☆ パラダイムが二つしかない場合も含めて複合パラダイムと呼ぶことにしよう．

オブジェクト指向におけるメッセージ伝達は，クラス階層とはある意味でレベルの異なるパラダイム要素である．実際，クラス階層がなくメッセージ伝達しかないオブジェクト指向言語も存在し得る．別稿[6]で述べたように，オブジェクト指向が，手続き型や論理型と異なり，いろいろな言語に融け込んでいく秘密がここにある．

手前味噌の実例になるが，われわれの TAO ではまずこの性能バランスのよさを設計の重点課題にした．自分で設計した言語だから，性能バランスの許容誤差の目標はまったくの直観により，手続き型，論理型，オブジェクト指向の間で1.5倍以内とした．結果は2倍弱以内で，目標はほぼ達成されたといっていいと思う．

5. マルチパラダイム支援環境に要求されるもの

さて，いよいよ本題である．Lisp には InterLisp[7] といういわばプログラミング環境の元祖のようなものがあり，オブジェクト指向には Smalltalk-80[8] という偉大な金字塔がある．論理型に関しては，まだ決定版はないようだが，デバッガ，トレーサ，動作の図式表示など論理型固有のツール類の技術蓄積が進行している．すなわち，個々のパラダイムに関するプログラミング環境はそれ相応に存在している．マルチパラダイム支援環境として問題にしなければならないのは，これらの個々の「要素技術」ではなく，これらを組み合わせたときに生ずる，マルチパラダイム特有の問題であろう．

5.1 ターンアラウンド性能のバランス

さきほどの議論のむし返しになるかもしれないが，実はここでも性能のバランスが一番大きな問題となる．ただし，ここでいう「性能」とは，プログラムを書き，走らせ，デバッグするというサイクルに要する

時間（ターンアラウンドタイム）といった，プログラミング環境としての性能である．

われわれの反省を込め，このことを端的に示す実例を一つ紹介しよう．それは TAO のオブジェクト指向に関するものである．TAO のオブジェクト指向は，ZetaLisp (MIT の Lisp マシン上の Lisp) の Flavors に大きな影響を受け，その後，Smalltalk-80 からの影響で大きく軌道修正された．TAO は Lisp をベースにした言語であるから，みかけは Flavors に非常に近い．いくつかの工夫により，関数呼び出しとメッセージ伝達の速度性能差は数パーセントから数十パーセント以内におさまった．だから，手続き型とオブジェクト指向の複合パラダイム言語としての性能バランスはほぼ満足できるものになった．当初，われわれはこれで初期の目標を達成したと考えていたのだが，そのうち「プログラミング環境」としては大きな欠点があることが判明したのである．それはオブジェクト指向のターンアラウンドタイムの遅さである．

オブジェクト指向で書いたプログラムが小さいうちはまったく意識されなかったが，1万行を越えるような大きなプログラムを書くと，プログラムの一部の小さな変更を行なってから走行可能になるまで，TAO の場合，とんでもなく時間がかかったのである．Lisp であれば，プログラム全体がどんなに大きくても，関数を一つ修正して再度走らせる手間は一定で，ほとんど瞬時である（インタプリタでもコンパイラでも同じ）．ところが，以前の TAO のインプリメントではメッセージ伝達を速くするために，メソッドを再定義したときなどには，ちょうどプログラム全体を再コンパイルしたのと実質的に同じことをシステムが行なうようになっていた（クラス階層をたどって，メソッドの意味をクラスごとに確定していく作業で，method construction と呼ぶ）．だから，メソッドの小さな修正が実行可能になるまで2分も3分も待つようなことになってしまった．言語システムがもともと Pascal のようにコンパイラベースであれば，「まあ，こんなもんか」ですむかもしれないが，一方におそろしくターンアラウンドタイムの短いパラダイムがあると，この遅さが目立ってしまう．一番恐いのは，ユーザが「TAO のオブジェクト指向は大きなプログラムを対話的に作るのには向いていない」と思い込んでしまうことである．

よく考えればだれでも気づくこんな簡単なことに対して実際の対策が行なわれたのは比較的最近である．米国人で Lisp マシンをよく使っている連中が，われわれのシステムが急に窒息するのを見て，「あ，method construction が始まりましたね」とよく言っていたが，こういう慣れが問題の発見を遅らすものなのである．

いろいろなパラダイムを混合すると，単体では問題にならないようなものがほかとの比較で問題になりだす．よいマルチパラダイム支援環境を作るには，単なる寄せ集めではすまないのである．

5.2 機能面のバランス

ターンアラウンドタイムのような性能面だけではなく，機能面でのバランスも重要である．たとえば，エラー発生時に与えられる情報にパラダイムの間で大きな差があってはいけない．たとえば，エラー時のバックトレースで，Lisp（手続き型）の部分はプログラムの中の変数名がちゃんとみえるのに，論理型では DEC-10 Prolog のようにシステムによって勝手に内部変換された変数名しかみえないのでは失格なのである．オブジェクト指向のメッセージ伝達は結果的には（メソッドという）関数呼び出しになる．だから，バックトレースの中にオブジェクト指向と手続き型が混在していてもあまり異和感がないが，論理型はみせ方に多少注意する必要がある（たとえば，残っている選択肢をどう表示するか）．

また，手続き型ではステッパ（高級言語でのワンステップ実行）が動くのに，論理型やオブジェクト指向では動かないというのも困る．Lisp の式の評価の際にフックをかける evalhook や applyhook，ユニフィケーションをフックする bindhook，メッセージ伝達をフックする messagehook（?）などはそれなりにうまくバランスして機能提供されていなければならない．もちろん，これらがいつも連動していてもダメで，ある特定のメッセージ伝達だけをフックしたいという要望にも応えられないといけない．バランスという意味には，組合せを自由に選べるということも含んでいるのである．だから，上のバックトレースでも，たとえば，ユニフィケーションに関するところを一切表示しないような指定ができることが望ましい．

オブジェクト指向ではブラウザがあるのにほかのパラダイムにそれに相当するものがないというのも，片手落ちだ．もっとも，ほかのパラダイムでのブラウザがどんなものに相当するかというアイデアが必要だ．個々のパラダイムはそれぞれ個性を主張するものだから，あるパラダイム特有のツールがほかのパラダイム

に適用できる必要はないという考え方も成り立つが，人間のアナロジやメタファーの能力は予想外に大きい．環境の統合化という観点から，パラダイムの枠を超えたツール概念の拡張はアタックしてみる価値がある．

要するにマルチパラダイム支援環境と称するものは特定のパラダイムをえこひいきにしてはならないのだ．これはかなり困難な課題である．たとえば，恥ずかしながらわれわれの TAO にはこういう意味での抜けがまだたくさんあることを白状しなければならない．

5.3 マルチパラダイム支援環境に特有なツール

いままでの話はいわれてみれば当然のものばかりである．ただし，ちゃんと実現することはまだまだ今後の課題である．さて，要素パラダイムの間のバランスだけがすべてではない．マルチパラダイム支援環境になってはじめて存在価値がでるようなツール類があるに違いない．しかし，筆者は寡聞にしてそのようなものが実現されたという話を聞いたことがない．ここでは夢と期待を述べることにとどめよう．

マルチパラダイムでは，言語やシステムがより複雑になる．だから，それなりのガイダンスをシステムが行なってくれる必要がある．思いつくものを少しあげると，

- パラダイムプロファイラ：プログラムの中でどのパラダイムがよく使われているといった使用頻度，実行時間，あるいはパラダイムの混合の粒度などの統計を，静的あるいは動的にとってくれるシステム．これはなにも手続き型と論理型といったパラダイムのレベルでなくても，ループと再帰法の使い分け，データ構造の選択，マクロと関数の使い分けなどのようなレベルの統計でも十分面白い．カラー画面上で，パラダイムごとに色分けされて表示されるというのも悪趣味かもしれないが，楽しいだろう．文章におけるスタイルチェッカーと同様なコメントを与えてくれるような親切なプロファイラができるともっと面白い．これらはプログラムの性能向上や品質向上に役に立つだろう．
- パラダイム変換：大分難しくなるが，あるパラダイムで書かれたプログラムを別のパラダイムに書き直してくれるシステム．プログラム変換の難しい問題を含むから簡単に実現できるとは思えないが，初心者に対しての教育効果ということに的を絞り，簡単なものに対象を限れば，案外うまくいくかもしれない．
- パラダイムアドバイザ：いわば究極の CAI かもしれないが，与えられた問題に対して適切なパラダイムの選択をガイドしてくれるシステム．当然，パラダイム部品データベースのようなものを背後にもつ．ふつうのプログラムデータベースや，プログラミングアドバイザがまだちゃんと実現されていない現在，これは夢のまた夢か．

やや古い職人気質世代の筆者にとって，上記のようなツールは必ずしも個人的な趣味に合わないが，このような挑戦的な問題にアタックすることでまたマルチパラダイムに関する新しい知見が得られると思う．

6. おわりに

まだ見ぬマルチパラダイム支援環境について，我田引水を含めて勝手なことを書いてきた．ソフトウェアに関する問題には未解決なものがやたらと多いのに，またマルチパラダイムといって問題を複雑化していくのは，性能や機能を追求してやまない人間の習性のように思える．たしかにひと昔前には信じられなかったような複雑な言語やシステムを人間は使いこなせるようになった．21 世紀になると，現在マルチパラダイムといっているものが，ふつうのごくありふれた概念になってしまう可能性すらある．だから，マルチパラダイム支援環境といってもなにも特別なことが起こらないかもしれない．

さて，最後にマルチパラダイムといえるかどうかは別として，多言語環境について簡単に触れておこう．多言語環境が必要になるのは，蓄積されたソフトウェア資産を生かしたい場合と，プログラムの主要部分ごとにパラダイムの切り分けが明解で，モジュールと言語の切り分けがうまく一致した場合だろう．こういう例は結構多い．だから，わざわざマルチパラダイム言語を用意しなくても十分だという発想が成り立つ．当事者である筆者にはこれに対して，将来どういう結論がデ・ファクトとして出るか予測ができない．マルチパラダイム言語派がこれに答えるには，パラダイム混合の粒度についてもっと経験を積む必要がある．ただ，プログラミング環境の観点からいえば，マルチパラダイム言語に基づくマルチパラダイム支援環境が優れているとはいえるだろう．本論で述べたように性能・機能のバランスがとれていれば，理想的に統合化されたプログラミング環境が必然的に実現してしまうからである．

参考文献

1) Bobrow, D.G. and Stefik, M.: The LOOPS Manual, Xerox PARC (1983).
2) Takeuchi, I., Okuno, H.G. and Ohsato, N.: A List Processing Language TAO with Multiple Programming Paradigms, New Generation Computing, Vol. 4, No. 4 (1986).
3) Weinreb, D., Moon, D. and Stallman, R.: Lisp Machine Manual, Fifth Edition, System Version 92, LMI (1983).
4) 新世代コンピュータ技術開発機構：ESP 説明書 第2版 (1987).
5) Bobrow, D.G. et al.: Common Lisp Object System Specification X3J13 Document 88-002 R, ACM SIGPLAN Notices, Vol. 23, No. 9 (1988).
6) 竹内郁雄：オブジェクト指向とほかのパラダイムの融合，情報処理，Vol. 29, No. 4 pp. 344～351 (1988).
7) Teitelman, W.: INTERLISP Reference Manual, Xerox (1974).
8) Goldberg, A. and Robson, D.: Smalltalk-80: The Language and Its Implementation, Addison Wesley (1983).

(平成元年1月23日受付)