The University of Osaka
Institutional Knowledge Archive

Title	Lispマシン上のオブジェクト指向とその実現に関する研究
Author(s)	大里, 延康
Citation	大阪大学, 1994, 博士論文
Version Type	VoR
URL	https://doi.org/10.11501/3075204
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

Lispマシン上のオブジェクト指向とその実現に関する研究

平成 5年 12月

大 里 延 康

Lisp マシン上のオブジェクト指向とその実現に関する研究

平成 5 年 12 月

大 里 延 康

内容梗概

本論文は，筆者が，日本電信電話公社武蔵野電気通信研究所，日本電信電話株式会社基礎研究所，ソフトウェア研究所，およびヒューマンインタフェース研究所において行なった，Lisp マシンの研究・開発ならびに，その上のオブジェクト指向プログラミングの実現とその応用に関する研究・開発についてまとめたものである。

まず第 1 章緒論と第 2 章以降の各章の第 1 節では，研究の背景，意義および研究の内容について概説している。また，各章の最後の節および全体の結論では，本研究で得られた結果と今後の課題について述べている。

人工知能研究では，プログラミング言語として古くから Lisp が用いられてきた。Lisp は，プログラムとデータがリストと呼ばれる単一のデータ形式で表現され，言語処理機構が `eval` という万能評価関数によって簡潔にモデル化された極めて美しいプログラミング言語である。Lisp は，主として記号を要素とする，動的に大きさの変化しうるリストを処理するので，リスト処理言語あるいは記号処理言語と呼ばれる。また，リストを表現するために，括弧を用いた単純な表現である S 式が用いられ，言語に新たな機能を導入する際にも非常に柔軟性がある。

近年，人工知能研究が急速に発展してプログラミング技術が高度化し，言語処理系に高い性能が要求されるようになった。しかし，Lisp は上記の種々の利点を持つ反面，実行制御が重くメモリ参照も多いため，処理系の負担が大きい。そのため，汎用計算機上の Lisp で大規模な実用プログラムを開発することが次第に困難になってきた。また，汎用計算機上で高性能の Lisp 処理系を実現することも困難である。その大きな理由は，Lisp 処理系の計算機構と汎用計算機のアーキテクチャとがうまく整合しない点にある。しかし，ハードウェア技術が長足の進歩を遂げ，このような不整合を緩和して言語処理系の機構に適したアーキテクチャを持つ専用計算機を実現することが可能となってきた。すなわち，Lisp 専用計算機の実現によって性能上の問題を解決し，加えて，比較的安価に入手できるようになった大容量のメモリを用いて，大量のデータを扱うことのできる実用レベルの Lisp システムを実現することが可能となってきた。

人工知能研究の進展はプログラミング技術の多様化をもたらし，さまざまなアルゴリズムやデータ表現，プログラミング・スタイルが出現した。このため，単一のプログラミング・パラダイムだけを備えた従来の言語処理系では，そうした多様性に十分応えることは困難である。そこで，種々のプログラミング・パラダイムをパッケージやライブラリとして利用することなどが行なわれてきたが，相互のリンクが面倒であったり，概念が整合していなかったりして統一性に欠け，必ずしも使いやすくないことが問題であった。こうした問題は，処理系の機構を抜本的に見直し，プログラミング・パラダイムを言語の核の部分で融合することによって解決することができる。また，そのパラダイムの融合のために必要なオーバヘッドは，専用のハードウェアを持つ計算機によって吸収することが可能である。

第2章では，このような背景のもとに筆者らが開発したLispマシンELISと，その上の言語TAOの設計思想およびその概要について述べている。

ELISは，専用ハードウェアをマイクロプログラムで制御することによって記号処理向きアーキテクチャを実現する。しかし，マイクロプログラミングによってTAOの言語処理系という抽象度の高い機能を記述するには落差が大きく多くの困難を伴うので，ELISの設計では，マイクロプログラマから見えるハードウェアの構造を単純化するという方針をとっている。たとえば，マイクロプログラマにはレジスタ間のデータ転送レベルの操作を見せるようにし，タイミング等を含むゲートレベルの煩雑な操作はハードウェアで処理する。

ELISではタグ・アーキテクチャを採用し，データの識別をハードウェアで支援している。また，大容量のハードウェア・スタックを備える。さらに，Lispのセルを一回のメモリ操作で読み書きしたり，読み出したポインタ・データをそのままアドレスとしてバスに送出できるようにするなど，強力なメモリ・インタフェースを実現している。

ELISのデータパスは比較的単純であるが，シーケンシングは必ずしもそうではない。また，水平型のマイクロ命令であるため，マイクロプログラマが指定しなければならない操作も多い。そのため，マイクロプログラムの開発負担が大きくなることが問題である。そこで，マイクロプログラミングのための高機能の支援系を作成した。この支援系では，マクロ機能を持つアセンブラや，マイクロプログラムの静的なフロー解析に基くWCSへのプログラムの割り付けなどを実現した。また，ELISの主記憶やスタック，レジスタなどの内容を，TAOのデータ構造を反映した記号表現で表示する機能などを備える。この支援系によって，大量のマイクロプログラムの開発が可能となった。第2章ではこの支援系についても述べている。

ELIS上の言語TAOでは，自然言語処理やエクスパートシステムなどの知能処理研究において強く求められていたLispの性能向上の要求にこたえ，さらに新しいプログラミング環境を構築していくための核となる言語を目指した。

まず，TAOではインタプリタによる実行性能を最大にすることを基本とした。その理由は，人工知能のプログラミングでは，プログラムを生成しながら実行する，といったインタプリタ的な実行を，多くの局面で含みうることである。また，試行錯誤を繰り返す会話的なプログラミング・スタイルが中心であり，インタプリタの実行速度はプログラム開発に際してのターン・アラウンドに大きく影響する。さらに，インタプリタは，プログラマの意図した実行環境を保持しながら走行するので，コンパイラ・ベースでのプログラミングよりデバッグ上も有利である。ELISのハードウェア機能を駆使してインタプリタの高速性を追求した結果，TAOでは汎用大型計算機上のLispシステムのコンパイラにも匹敵する速度性能を達成した。また，既存の商用Lispマシンのインタプリタの性能をはるかに凌ぎ，そのコンパイラにも比肩しうるものとなっている。

これに加え，TAOでは，プログラミング・パラダイムの選択をユーザの自由に任せることができるシステムであることを中心思想に据えた。人工知能プログラミングにおける多様な要求に応えうるシステムでは，プログラマができるかぎり自己の持つ問題に集中でき，言語特有のパラダイムに思考を制約されないように配慮することが必要である。TAOでは，Lispの持つ柔軟な特性を活かして，Lisp本来の逐次型パラダイムに，論理型，オブジェクト指向といったパラダイムを，処理系の核の部分でバランスよく融合し，実用レベルの複合プログラミング・パラダイムのシステムを実現した。

第3章では，TAOの複合プログラミング・パラダイムの中でも重要なものの一つであるオブジェクト指向について，その言語仕様ならびに実現法を詳しく述べている。プログラマが問題を計算機

上に表現する際には，データ表現などのモデル化をどうするかがまず問題になる。Smalltalk-80 によって注目を集めたオブジェクト指向の諸概念はこれをサポートする強力な道具立てであり，TAO においてこれをいかにうまく取り込むかは，複合プログラミング・パラダイム言語としての TAO の設計において重要なポイントであった。

オブジェクト指向では，オブジェクトとしてのデータは受動的なものではなく，手続き (メソッド) の付随する能動的な存在である。オブジェクトはメッセージを受信することができ，メッセージによって指定された自分のメソッドを実行する。このような計算機構を実現するためには，データと手続きをうまく関係づける手段が必要になる。特に，受信したメッセージの情報から起動すべきメソッドを効率よく特定する機構が重要である。これは，Lisp のように関数呼び出しなどの形でまず手続きが与えられ，その中で受動的なデータを操作する，といった計算機構にもとづく手続き型のパラダイムとは，根本において異なる点である。また，このような異質な計算機構を，言語の表層においても他のパラダイムと混乱することなく，かつ自然に取り込んで，相互にオーバヘッドなく乗り入れることができるようにする必要がある。また，メソッドの実行本体も，他のパラダイムの実行本体と同等の速度性能を持っている必要がある。

TAO では S 式の特殊な形式を用い，言語の表層でのメッセージ伝達式の自然な融合を実現した。オブジェクト指向の速度性能には，メッセージ伝達式の解釈機構，メソッドの探索と起動，そしてメソッド本体の実行という三つの要因がある。TAO では，ELIS のタグとマイクロプログラムの多重分岐の機能を用いて，Lisp の式の解釈機構にオーバヘッドを持ち込まずに `eval` の中にメッセージ伝達式の解釈機構を埋め込んだ。また，大規模な応用プログラムに対してもメソッド表を大きくしないため，マイクロプログラムによるバイナリ・サーチでメソッド探索を行なう方式を採用した。実測により，メモリ消費の大きいハッシュを用いなくてもメソッド探索は十分高速であることを確認した。さらに，メソッド本体内の変数のアクセス機構には，タグによる前処理とハッシュ表などを用い，基準となる Lisp と比べて遜色のない高速処理を実現した。また，オブジェクト指向をサポートする内部情報を徹底したオン・ディマンドによって作り出す方式を採用し，会話的なプログラミングでのターン・アラウンドを確保するとともに，メモリの消費を最小に押えた。第 3 章ではこれらの詳細を論ずるとともに，ベンチマークや実際の応用プログラムによる総合的な評価を行ない，本研究で用いた実装法の妥当性を明らかにしている。

第 4 章では，TAO のオブジェクトを並行オブジェクトに拡張し，実世界を表現するモデルとしてより自然なパラダイムを提案している。ここでは並行プロセスを伴うオブジェクトをエージェントと呼び，ロボットの高次のプログラミングに応用した結果について述べている。

ロボットの知能化は知能処理研究の中でも重要な分野であるが，ロボットは複合的なシステムであるため要素技術の研究に重点が置かれ，最新のソフトウェア技術を適用してシステム全体を統合する研究は大きく立ち遅れている。特に，近年センサやアクチュエータなどが多様化し，それを組み込むハードウェア構成技術も進歩しており，それに対応したモジュラリティの高い制御システムによってロボットの行動をロバストで柔軟なものにすることが求められるようになってきている。

本研究では，ロボットへの作業教示の中でも，いわゆるオフライン・プログラミングと呼ばれる通常の計算機プログラミングとしての作業記述に，並行オブジェクト指向を適用する方式を提案した。ここでは，作業をプログラミングの視点で分解し，その構成要素の処理をエージェントの担当すべき役割として，陽に定義し割り当てるようなモデルを与える。このモデルでは，ロボットの作業世界はオブジェクト指向で表現され，物理的実体は論理的なオブジェクトとして記述される。そして，一群のエージェントが作業対象物に対応する論理的なオブジェクトを操作することによっ

て，ロボット全体の作業が遂行される。エージェントは，能動性を持つ独立な論理的実体として，並行的に，かつ必要に応じて相互に交信しながら，システムに課された仕事を分担して共同で処理する。このようなプログラムの構成法をとることにより，複合的なハードウェアからなるロボット・システムが，部分的に更改されたり新たなセンサ等の導入が行なわれたりしても，作業遂行方針そのものを変更しない限り，全体のプログラムは影響を受けないようにすることができる。

また，本研究ではエージェントを作業戦略のモジュール化のために利用し，ロボットの行動を段階的に知能化する手法を提案した。すなわち，操作対象物に対応する処理を担当するエージェントに加え，ロボットの行動パターンをそのまま担う役割を持つエージェントを導入することによって，マニピュレータを操作する作業戦略モジュールを与えた。この手法を実ロボットで実験し，困難性を伴う作業を信頼度よく容易に実現することによってその有効性を示している。

また本研究では，エージェントを単なる受動的なデータと見て操作することによってエージェント相互の監視・制御を実現し，ロボットが全体としてある種の内観能力を持つようにすることを提案している。すなわち，単にマニピュレータを操作するだけの下位のエージェントを，その作業目的に即した高位の判断を行なうエージェントが監視し，必要に応じて戦略を変更して下位の作業遂行エージェントに影響を与えうる構成にする。これらのエージェントは独立しているので，ロボット全体として見れば，自分が遂行する作業の目的を，動かしている手先の運動とは別のものとして意識していることになる。このような構成法は，ロボットの行動を作業環境の変動に即して柔軟に適応させるための基本的な機構として有効であると考えられる。本章では，こうした構成法の応用としてXウィンドウによるエージェント監視系を実現し，その有効性を示した。

最後に第5章では，本論文全体のまとめを行なう。オブジェクト指向は，数あるプログラミング・パラダイムの中でも，モジュール化技術として極めて優れた性質を持っている。本研究では，Lispマシン ELIS を開発し，そのマイクロプログラミングによって高性能の複合プログラミング・パラダイム言語 TAO を実現した。ELIS の性能を背景に，TAO の核の部分でオブジェクト指向を融合することにより，オブジェクト指向を実用レベルの速度性能で実現できることを示した。さらに，応用プログラムによる実地の評価を行なって，具体的な実現の諸技術とその妥当性を示した。また，オブジェクト指向に並行性を与えて，実世界を記述するためにより自然なモデルを提案し，ロボット制御に応用してその有効性を示した。

目次

内容梗概 i

1 緒論 1

2 Lisp マシン ELIS とマルチパラダイム言語 TAO 11
2.1 緒論 11
2.2 Lisp マシン ELIS 14
2.2.1 ELIS のアーキテクチャ 14
2.2.2 ハードウェア / ファームウェア開発環境 MIC 18
2.2.2.1 ELIS とホスト計算機とのインタフェース 18
2.2.2.2 マイクロアセンブラ 21
2.2.2.3 マイクロリンカ 21
2.2.2.4 マイクロプログラムのデバッグ 25
2.3 マルチパラダイム言語 TAO 27
2.3.1 インタプリタの重視 28
2.3.2 タグの利用とデータ型 29
2.3.3 変数 30
2.3.4 代入 33
2.3.5 並行プログラミング 33
2.4 TAO の速度性能 34
2.5 結論 35

3 TAO におけるオブジェクト指向とその実現 37
3.1 緒論 37
3.2 TAO におけるオブジェクト指向 38
3.2.1 オブジェクト 38
3.2.2 メッセージ 39
3.2.2.1 メッセージ伝達式 39
3.2.2.2 メッセージ伝達式の評価 39
3.2.2.3 中置記法 40
3.2.3 クラス 40
3.2.3.1 クラスの定義 40

3.2.3.2 クラスの階層と継承 41
3.2.3.3 クラスの順序づけ 41
3.2.4 変数 42
3.2.5 インスタンスの生成 42
3.2.6 メソッド 42
3.2.6.1 メソッドの定義 42
3.2.6.2 メソッド結合 43
3.2.7 スーパー・メッセージ伝達 43
3.2.8 no-method-found ハンドラ 44
3.2.9 リスト・メッセージ 44
3.3 オブジェクト指向の実現 45
3.3.1 クラスの内部表現 45
3.3.1.1 クラスベクタ 45
3.3.1.2 defclass の動作 45
3.3.1.3 component-of-what 46
3.3.2 オブジェクトとしての基本データ 46
3.3.3 オブジェクトの内部表現 46
3.3.4 メソッドとメソッド表 48
3.3.4.1 メソッド 48
3.3.4.2 メソッドの定義 48
3.3.4.3 メソッド表の作成 48
3.3.4.4 メソッド結合の実現法 50
3.3.4.5 メソッド applobj の共有 51
3.3.5 変数のアクセス機構 51
3.3.6 メッセージ伝達のメカニズム 52
3.3.6.1 udo へのメッセージ伝達 52
3.3.6.2 基本データ・オブジェクトへのメッセージ 53
3.3.7 コンパイラ 54
3.4 TAO オブジェクト指向の性能 55
3.4.1 メソッド探索の速度 55
3.4.2 インスタンス変数のハッシュ 55
3.4.3 簡単なプログラムによる Lisp プログラムとの速度比較 57
3.4.4 応用プログラムによる評価 59
3.4.4.1 メソッド表のサイズとメソッドの呼ばれかた 59
3.4.4.2 udo の作られ方と udo の平均的な大きさ 60
3.4.4.3 メソッド表作成方式の評価 60
3.4.4.4 メソッド結合の使用率 62
3.4.4.5 メソッド本体の共有率 62
3.5 結論 63

4 並行オブジェクト指向とそのロボット制御への適用 65
4.1 緒論 65
4.2 エージェント・モデル 67
4.2.1 オブジェクト指向による世界記述 67
4.2.2 エージェント 68
4.2.3 エージェントとプロセス 68
4.2.4 エージェント間の情報交換 68
4.2.4.1 メッセージ 68
4.2.4.2 イベント 69
4.2.5 行動インタプリタ 69
4.2.6 制御の対象としてのエージェント 70
4.3 インプリメンテーション 71
4.3.1 並行プロセスとエージェントの表現 71
4.3.2 行動インタプリタ 71
4.4 実験 73
4.4.1 ハノイの塔 73
4.4.2 ハノイの物理世界の表現 75
4.4.3 機能ベースエージェント 75
4.4.4 行動ベースエージェント 77
4.5 エージェントの監視・制御 82
4.5.1 行動ベースエージェント間の相互作用 83
4.5.2 X ウィンドウによる状態表示 83
4.6 結論 84

5 結論 87

謝辞 91

参考文献 93

A クラスベクタの内部構造 97

B 典型的なメッセージ伝達式のマイクロプログラムのトレース 99

C バイナリ・サーチのマイクロサブルーティン 103

第 1 章

緒論

人工知能 (AI) 分野におけるプログラムは膨大な計算やデータ処理を必要とするものが多く, AI プログラミングのための言語には, 高速で大量の情報を処理する能力が要求される。また, AI 研究の進展に伴ってプログラミング技術が多様化し, さまざまなアルゴリズムやデータ表現, プログラミング・スタイルが用いられるようになってきた。このため, 単一のプログラミング・パラダイムだけを備えた言語処理系では, AI プログラミングの多様性に十分対処しきれなくなってきている。本論文は, AI プログラミングを支援する高性能のプログラミング環境を, オリジナルのハードウェア／ファームウェア／ソフトウェアによって実現した TAO/ELIS プロジェクトにおいて, 特にオブジェクト指向プログラミングに関する研究を中心に述べたものである。

AI プログラミングには古くから Lisp が用いられてきた。Lisp は, 1960 年に John McCarthy によって提案されたプログラミング言語で, Fortran と並ぶ古い歴史を持つ。この言語は, `eval` という万能評価関数によって簡潔にモデル化された計算機構を持ち, データと手続きをリストと呼ばれる統一的表現で取り扱うことができる。リストは動的に大きさの変化するデータの表現に適する。また, その要素として記号を扱えるので, Lisp は記号処理言語であるといわれる。また, リストの表現には S 式と呼ばれる, 括弧を用いた単純な表現が用いられ, 言語に新たな機能を導入する際にも非常に柔軟性がある。こうしたことから, Lisp は AI プログラミングのための基本的な言語としての生命力を維持しつつ, その地位を確固たるものとしてきた。

近年, AI の研究が急速に発展してプログラミング技術が高度化し, 言語処理系に高い性能が要求されるようになってきた。しかし, Lisp は種々の利点を持つ一方, 処理速度が遅くメモリに対する負荷が大きいため, 大規模な実用プログラミングが困難になるなどの問題が生じてきた。こうした問題は, Lisp 処理系の計算機構と, 一般に用いられてきた汎用計算機のアーキテクチャとがうまく整合しない点に大きな原因がある。

これに対し, 近年のハードウェア技術の長足の進歩を背景として, AI をターゲットとする専用の計算機を実現しようとする動きが起こった。**Lisp マシン**と呼ばれる一群の Lisp 専用計算機はその代表的なものである。Lisp マシンの提案は, 1973 年に L. P. Deutsch によってなされた [1]。1974 年には, MIT において, 後の商用 Lisp マシンの原形となった CONS が製作されている [2]。MIT の Lisp マシンは, MacLisp で書かれた AI プログラムを高速実行することを狙っていた。これとほぼ同じ時期に, Xerox 社のパロアルト研究所 (PARC) でも, InterLisp をベースとする Lisp マシンの開発が行なわれていた。その後, MIT の CONS マシンは 1978 年に CADR マシンに継承され, 1980 年には, Symbolics Inc. と Lisp Machine Inc. の 2 社が設立されて, 商用の Lisp マシンとして提供されるに至った。

我が国では，米国にわずかな遅れをとりつつも早い時期からLispマシンの研究が開始され，電子技術総合研究所，青山学院大学，神戸大学，京都大学，大阪大学などで，それぞれ独自のアーキテクチャを持つLispマシンの試作が行なわれた。初期のLispマシンは，制御記憶が高価であったこともあり，Lispのプログラムを一旦中間言語のプログラムに変換し，それをマイクロプログラムによって解釈実行するものが多かった。しかし，神戸大学の瀧らは，ビットスライスのマイクロプロセッサを用いて16ビットのマシンFast-Lispを自作し，`eval`をマイクロプログラムで直接記述して，小さいながらも極めて高い性能が達成できることを示した [3] [4]。

筆者らは日本電信電話公社武蔵野電気通信研究所(当時)において，プログラミングの対象領域の拡大や多様化に対応しうる，柔軟で高性能なAIエンジンを目指して，Lispマシン ELISの研究・開発を1980年に開始した [5] [6]。ELISの設計思想はFast-Lispの影響を受けている。Fast-Lisp自体は実験的な試作機で規模も小さく，そのままで実用に供することは困難である。ELISは，単純なアーキテクチャとマイクロプログラミングによる処理系の高速化というFast-Lispの思想を採り入れ，大型の実用レベルのマシンを開発することを狙いとした。

ハードウェア技術によるLispの性能向上が図られる一方，ソフトウェア技術やプログラミング・パラダイムの面では，AIプログラミングの高度化に応えるため，LOOPS [7] やESP [8] 等のように，複数のパラダイムを言語に採り入れる試みがなされてきた。しかし，プログラミング・パラダイムをパッケージやライブラリとして提供するだけでは，相互のリンクが面倒であったり概念が整合していなかったりして言語としての統一性に欠け，また処理速度も十分でなかったりして，必ずしも使いやすいものにならないという問題があった。こうした問題は，処理系の機構を抜本的に見直し，プログラミング・パラダイムを言語の核の部分で融合することによって解決することができる。筆者らは，問題の性質に応じたプログラミング・パラダイム選択がひとつの言語の中で行なえ，相互に自由に行き来できるような新しい記号処理言語TAO [9] [10] を実現することにした。このようなパラダイムの融合のために生じうるオーバヘッドは，ELISの専用ハードウェアとマイクロプログラミング技術によって吸収することを狙った。

本論文の第2章では，Lispマシン ELISのアーキテクチャを紹介し，そのマイクロプログラミング開発支援環境について述べたあと，TAOの設計思想およびその概要について述べる(関連発表論文 (1) (2) (6) (7) (8) (9) (10) (13) (14) (15) (16))。

ELISでは，Lisp向きのハードウェア資源をマイクロプログラムで操作することによって処理系の高速化を実現する。このような試みが可能となった背景には，WCS[1]やスタックに使用できる高速メモリや主記憶用の大容量メモリが入手可能となったほか，ビットスライスのマイクロプロセッサの出現などによって，実験室レベルで比較的容易に専用計算機を試作することができるようになったことが挙げられる。

TAOの言語処理系という，抽象度が高く大きな機能をマイクロプログラムで実現することには実現上の困難が伴う。すなわち，言語処理系とハードウェアとの機能レベルの落差をいかにして埋めるかが問題となる。この困難を緩和するため，ELISの設計では，マイクロプログラマから見えるハードウェアの構造を単純化するという方針をとっている。たとえば，マイクロプログラマにはレジスタ間のデータ転送レベルの操作を見せるようにし，タイミング等を含むゲートレベルの煩雑な操作はハードウェアで処理する。マイクロプログラムのレベルでは，操作対象として比較的単純な3バス構成のデータパスを意識していればよい。

---

[1]書き換え可能制御記憶，Writable Control Storage: WCS

ELISのデータパスは比較的単純であるが，シーケンシングは必ずしもそうではない。また，64ビットの語幅を持つ水平型のマイクロ命令であるため，マイクロプログラマが指定しなければならない操作の量も多い。そのため，マイクロプログラムの開発負担が大きい。この問題に対処するため，マイクロプログラミングのための高機能の支援系を作成した。この支援系自体を小型のLispで開発し，Lispの特性を活かして会話性を高め，Lispの機能を利用したマクロや，マイクロプログラムの静的なフロー解析に基くWCSへのプログラムの割り付けなどを実現した。また，この支援系はELISの主記憶やスタック，レジスタなどの内容を，TAOのデータ構造を反映した記号表現で表示する機能を備え，ハードウェアを会話的に操作できる。この支援系によって，大量のマイクロプログラムの開発が可能となった。第2章ではこの支援系についても述べている。

ELISでは**タグ・アーキテクチャ**をとり，データの識別をハードウェアで支援する。また，言語の実行管理をサポートする大容量のハードウェア・スタックを備える。スタック・トップのデータは常に専用のレジスタに置き，スタックを内部レジスタと同等の速さでアクセスできるような機構を実現している。さらに，64ビットのLispセルを一回のメモリ操作で読み書きしたり，読み出したポインタ・データをそのままアドレスとしてバスに送出できるようにするなど，リスト操作を支援する種々の機能を持つ強力なメモリ・インタフェースを実現している。また，メモリ・インタフェース用のレジスタをデータパスの中に配置して汎用レジスタとして使用できるようにし，加えて，バイト単位での操作を可能とすることによって，文字列処理やコンパイルされたプログラムのための命令バッファとしても使用できるようにしている。

言語TAOでは，自然言語処理やエクスパートシステムなどの知能処理研究において強く求められていたLispの性能向上の要求にこたえ，さらにAI研究のための高性能のプログラミング環境[2]を構築していくための核言語としての機能を備えた処理系の実現を目指した[11]。

優れたプログラミング環境の第一の要件は速度性能である。いかに高度の機能を実現していても，速度が遅ければ有用性は著しく低いものとなる。TAOではベースとなるLispの速度性能を重視した。特に，インタプリタによる実行性能を最大にすることを目指し，インタプリタで十分実用になる高速性を追求した。その理由は，まず，AIプログラムには，インタプリタ的な実行が少なからず含まれることである。また，AIプログラミングでは，試行錯誤を繰り返す会話的なプログラミング・スタイルが多く，インタプリタの実行速度はプログラム開発時のターン・アラウンドに大きく影響する。さらに，プログラマの意図した実行環境を保持しながら走行するインタプリタは，コンパイラ・ベースでのプログラミングよりデバッグ上も有利である。

インタプリタの高速化のため，インタプリタの本体である`eval`をはじめ，基本関数，基本メカニズムなど速度性能に影響する殆どすべての機能をマイクロプログラムで記述した。これにより，処理系のプログラム自体の主記憶からのフェッチがなくなって大幅に速度が向上する。さらに，ELISの専用ハードウェアを駆使してインタプリタの高速性を追求した結果，充分大きなLispの応用プログラムでDECsystem-2060上のMacLispなどのコンパイラに匹敵する速度性能を達成した。また，商用LispマシンSymbolics-3600のインタプリタの性能をはるかに凌ぎ，コンパイラにも比肩しうるものとなった。

これに加え，TAOではAIプログラミングの多様な要求に応えるため，プログラミング・パラダイムの選択をユーザの自由に任せることができる複合プログラミング・パラダイムのシステムとすることを中心思想に据えた。パラダイムを自由に選択することができれば，ユーザは与えられた言

[2]これをNUE: New Unified Environmentと呼ぶ。

語特有のパラダイムに思考を制約されることが少なくなり，プログラミングの労力を自己の持つ問題に集中することができる。

TAOでは，複数のプログラミング・パラダイムを，専用ハードウェアの支援のもとに，いかに効率よく，かつ自然に融合されたものとして実現することができるかを追求した。そのためにベースとなる逐次型パラダイムであるLisp処理系の本体`eval`を見直し，Prolog [12] に代表される論理型プログラミングの処理機構であるユニフィケーションやバックトラック，Smalltalk [13] に代表されるオブジェクト指向の計算モデルであるメッセージ伝達機構などの本質的部分を，処理系の中心部において同化させた。すなわち，これらの機構は，`eval`という単一の評価機構の核の部分で実現され，結合されている。大雑把には，TAOは，ラムダ計算，Horn節，メッセージ伝達そのほかのセマンティクスを持つS式を提供している，と言うことができる。ユーザは，プログラム・モジュールのレベルのみならず，式のレベルでこれらを混ぜ合わせることができる。TAOは，これらのプログラミング・パラダイムを，単にマクロ展開やライブラリ・パッケージによって提供するのではなく，言語の核において融合している。

こうしたプログラミング・パラダイムの融合にあたっては，AIプログラミングの基本となるLispの基本性能を損わないことに重点を置いた。すなわち，パラダイム融合のためにLisp自体の性能を犠牲にすることは避ける。しかし，それが可能な範囲ではできる限りの機能を盛り込む，という姿勢をとっている。各パラダイムの扱う変数や構造体などはすべてLispのデータ構造を用い，パラダイム間で情報を変換する必要はない。パラダイム間の処理上の分岐などは，ELISのマイクロプログラミングにおいてタグによる多重分岐を用い，Lispへのオーバヘッドをなくした。

複合プログラミング・パラダイム言語では，融合されたパラダイムの性能間に不均衡があることは望ましくない。パラダイム間に大きな性能の差があれば，結果的に性能の悪いパラダイムは使われず，パラダイムを融合する利点は失われる。TAOではパラダイム間の性能比を均衡させることに注意を払い，速度性能，メモリ消費，機能性，あるいはプログラミングの容易さ，といった点で殆ど同じか比較できる程度の差で実現した。従って，ユーザは，目的に応じてこれらのプログラミング・パラダイムを適当に選択し組み合わせることができ，パラダイム毎の性能に合わせて自分のプログラミング・スタイルを調整する必要はない。

TAOではインタプリタの高速化に主眼を置いているが，プログラムをコンパイルすることによって，より高速な実行ができるようになるほか，プログラム自体の占めるメモリ量を削減したり，ソースコードを隠蔽したりすることが可能になる。このため，インタプリタとの両立性を重視したコンパイラも実現している。コンパイラの出力は仮想的なスタック・マシンを操作する中間コードで，そのインタプリタをマイクロプログラムで記述した。

TAOは，Lispマシンのオペレーティング・システムのための基本機能も併せ持ち，マルチプロセス，マルチユーザ，独自のファイル・システムのサポート等を行なっている。従って，TAO/ELISは単なるLisp処理系ではなく，独立した総合的なLispプログラミング環境のためのベースとなっている。Lisp本体の仕様はCommon Lisp [14] に準拠し，全体としてはCommon Lispのスーパーセットである。

TAOの設計・開発とELISのいくつかの部分の開発は相互にオーバラップして進められ，両者は一体となったシステムとして成長した。ELISで効率よく実現可能かどうか，ということが，TAOの言語仕様を決める際の機能選定基準の重要なものの一つであった。また，TAOの必要とする機能のうち，実現の比較的容易なものはELISのハードウェアに反映された。マイクロプログラムのコーディング上，たとえば，メモリ参照の待ちで無駄時間が生ずる場合などには，その待ち時間の

中に埋め込める限り，より高度の機能を導入する，といった方策をとっている。このように，計算機ELISとプログラミング言語TAOは，きわめて緊密な関係にある。

本論文の第3章では，TAOにおけるオブジェクト指向とその実現について述べる。知能処理プログラムを始め，プログラマが計算機上で問題解決を図る際には，データ表現など，対象をどのようにモデル化するかが問題になる。オブジェクト指向の諸概念はこれをサポートする強力な道具立てであり，Lispを始めとする他のプログラミング・パラダイムの中にこれをいかにうまく融合するかは，複合プログラミング・パラダイム言語TAOの設計において極めて重要なポイントであった(関連発表論文(3)(11)(12)(17))。

オブジェクト指向の萌芽は1960年代のSimulaに遡ることができるが，1980年代のSmalltalk-80を契機として大いに注目されるところとなった。Smalltalk-80の持つ**オブジェクト**[3]，**クラス**[4]，**継承**[5]，**メッセージ伝達**といった諸概念は洗練されており，現在もなおオブジェクト指向の考え方の骨格をなしていると言える。Smalltalk-80の思想を採り入れて発展させたのがZetalisp [15]のオブジェクト指向パッケージとして実現されたFlavor Systemである[16]。Smalltalk-80では，概念の抽象化という視点を重視した**単純継承**[6]が実現されていたが，Flaversでは，部品の集積による新たなデータの生成という視点を重視した**多重継承**が採り入れられた。Flaversでは，この多重継承によって徹底したモジュール化の機能を提供することに主眼が置かれている。

TAOのオブジェクト指向は，Flaversを手本としながら独自の機能を加え，単なるソフトウェア・パッケージとしてでなく処理系の基本的なメカニズムとして言語仕様に組み入れた点に特徴がある。Lispの基本データはオブジェクト指向の意味でのオブジェクトである。逆に，ユーザの定義するオブジェクトもLisp自身のデータ構造である。

TAOは，Lisp言語としての性能を最重視している。従って，TAOにLisp以外のプログラミング・パラダイムを融合するに際しては，性能や機能をLispを基準にして比較検討することになる。TAOへのオブジェクト指向の融合では，従来の手続き型パラダイムであるLispとは異なる異質な計算機構を，言語の表層において他のパラダイムと混乱することなく，かつ自然に取り込んで，いかに相互にオーバヘッドなく乗り入れることができるように実現するかが問題であった。受信したメッセージによって指定されたメソッドを自己の文脈において実行するオブジェクト指向の計算機構を実現するためには，データと手続きをうまく関係づける手段が必要になる。特に，受信したメッセージの情報から起動すべきメソッドを効率よく特定する機構が重要である。これは，手続きが先に決定されていて，その中で受動的なデータを操作する従来からの手続き型のパラダイムとは全く逆であり，根本において異なる点である。また，パラダイム間の性能の均衡を重視する立場から，メソッドの実行も，他のパラダイムの実行と同等の速度性能を持っている必要がある。

オブジェクト指向の実現においては，メッセージ伝達式の解釈機構，メソッドの探索と起動，そしてメソッド本体の実行という三つのポイントがある。TAOでは，まず，Lispの関数呼び出しの

---

[3]Object: データとその上の手続きを一体化したもの。オブジェクト内部の実現法は隠蔽され，メッセージと呼ばれるプロトコルのみが公開される。オブジェクトの内部状態はインスタンス変数と呼ばれる変数に保持される。オブジェクトがメッセージを受信すると，その内容に応じてメソッドと呼ばれる内部手続きが起動される。

[4]Class: 同類のオブジェクトをまとめたもの。個々のオブジェクトは，クラスのインスタンスとして生成される。Flavor Systemでは，クラスと呼ばずflavorと呼ぶ。

[5]Inheritance: クラスを定義する際に，既存のクラスをもとに追加・修正等を行ない，新たなクラスを定義する手法。

[6]Single Inheritance: 継承において，上位のクラスを1つだけ許すもの。複数許すものを多重継承(Multiple Inheritance)と呼ぶ。

式としてはこれまで不正であった, car 部が関数でない式を新たにメッセージ伝達式として利用し, Lisp プログラムの中に自由に混在させることができるようにした。関数呼び出しとメッセージ伝達式はプログラマの意識の上でも明確に区別され, プログラミング上の違和感も殆どないので, 言語の表層でのオブジェクト指向の融合は自然なものとなった。この形式は Smalltalk スタイルの中置記法になっており, 算術演算などの可読性が向上した。

インタプリタ eval は, メッセージ伝達式を識別すると, 受信者であるオブジェクトのメソッドの中から所望のものを探索して起動する。TAO では, マイクロプログラムによるバイナリ・サーチでメソッド探索を実現した。この方法はハッシュによる実現と異なり, 単純でメモリ消費が少なく, 探索の深さも, 実用的なプログラムにおいて殆ど速度上の問題にならないことがわかった。

メッセージ処理の本体であるメソッドには Lisp の関数と同じ構造を与えた。従って, 実行上の基本機構は Lisp の関数そのものとほぼ同じである。オブジェクト特有の実行環境である変数のアクセスは, ハッシュ表などを用いて高速化した。その結果, 基準としての Lisp の速度と比べても遜色のない速度性能を実現することができた。

大規模な応用プログラムでは, 継承によるメソッドの重複やメソッド表の肥大化などのためにメモリの使用効率が悪化したり, 種々の内部データを生成する時間が無視できなくなるなどの問題が生ずる。そこで, メソッドの内部表現の共有化を図ったり, 内部的な情報の構築を徹底したオン・ディマンドによって作り出す方式を採用し, 真に必要な情報のみが必要な時点でしか作られないようにするという方策をとった。これによって, プログラム開発時のターン・アラウンドが短縮され, 会話性を確保することができた。また, メモリの消費を最小に押えて大規模な応用プログラムの開発にも耐えうるものとすることが可能になった。

第3章ではこれらの詳細を論ずるとともに, ベンチマークや実際の応用プログラムによる総合的な評価を行ない, 本研究で用いた実装法の妥当性を明らかにしている。

第3章で述べたオブジェクトの動作は基本的に逐次実行型であり, 各オブジェクトは実行制御上は独立しておらず, 並行には動作しない。しかし, オブジェクトの自律性を高めれば, 各オブジェクトが並行動作する**並行オブジェクト指向**のモデルに移行することは極めて自然である。実世界を素直に表現する必要性の高い現実の応用分野でも, オブジェクトを独立したプロセスと見て分散方式でモデル化を行ないたい場合が多々ある。本論文の第4章では, TAO のオブジェクト指向に並行処理の機能を付加して並行オブジェクト指向プログラミングに拡張し, 知能ロボットの高次制御に応用してその可能性を検討した。

ロボットはそれ自身物理的な実体をもち, 各種のセンサからの多様な情報を処理したり, アクチュエータを介して実世界に物理的作用を及ぼしたりする計算機システムである。物理世界とのさまざまなインタフェースや, 遂行する作業の多様性に応じて, ロボットは異質な構成要素からなる複合システムとなり, それを制御するソフトウェアも複合的なものとなる。特に, 近年センサやアクチュエータなどが多様化し, それを組み込むハードウェア構成技術も進歩してきた。このため, そうした複合システムに対応した柔軟な制御システムを実現する必要性が高まっている。

分散型の知能ロボットの研究では分散問題解決アルゴリズムなどの個別の問題に力点が置かれたものが多く, ロボットへの教示システムとしてのプログラミングの観点からなされた仕事はあまり多くない。また, 一方で, 知能ロボット研究において, 分散・協調による知能化・ロバスト化などを目指すマルチエージェントによるロボット・システムが注目されているが [17], エージェントと呼ぶ実体は多様で, ソフトウェアの構成法に重点を置いた研究は殆どない。本研究では, ロボットへの作業教示の中でも, いわゆる**オフライン・プログラミング**と呼ばれる作業プログラミングに,

並行オブジェクト指向を適用する方式を提案する。

複合システムであるロボットに作業を教示するには，その構成要素に対応したモジュール構成にもとづいて作業環境や処理手順を明解に表現し，不確定な要因やシステムの変化に容易に対応できるソフトウェア構成をとることが必要である。そうしたモジュール構成をとることによって，複合的なハードウェアからなるロボット・システムが，部分的に更改されたり新たなセンサ等の導入が行なわれたりしても，作業遂行方針そのものを変更しない限り，全体のプログラムは影響を受けないようにすることができる。また，モジュール間に協調的な相互作用を行なわせ，ロボットの行動能力を高度化することも期待できる。

本章では，ロボットへの教示を作業プログラミングの視点で分解し，その構成要素の処理を，**エージェント**と呼ぶ並行オブジェクトの担当すべき役割として，陽に定義し割り当てるようなモデルを考える (関連発表論文 (4) (5) (18) (19) (20))。このモデルでは，ロボットの作業世界はオブジェクト指向で表現され，物理的な実体が，論理的なオブジェクトとして記述される。そして，一群のエージェントが作業対象物に対応する論理的なオブジェクトを操作することによって，ロボット全体の作業が遂行される。このようなロボット・プログラミングの実験システムを，TAO のオブジェクトに並行プロセスとしての性格を与えることによって実現した。この実験システムを MARCS (Multi-Agent Robot Control System) と名づける。

MARCS では，インタプリタをベースとして，ロボット制御の動的な振舞いを実現した。ロボットは，あらかじめ決められたとおりの動作ができるのみならず，物理世界から取得した多様な情報に柔軟に対処しながら，その時点その時点での意思決定にもとづいて行動を変更することができなければならない，という本質的に動的な性質を持っている。MARCS では，このような個々のエージェントの行動レパートリの動的な起動制御を行なうために，行動インタプリタと呼ぶメタプログラムを用いたソフトウェア構成法をとった。

また，エージェントを作業戦略のモジュール化のために利用し，ロボットの行動を段階的に知能化する手法を提案した。すなわち，物理的な対象物やセンサ等にほぼ直接対応した処理を担当するエージェントのほか，特に，**包摂アーキテクチャ** [18] において提案された「振舞い」に相当する役割を担うエージェントを導入することによって，マニピュレータを操作する作業戦略モジュールを与えた。

実ロボットでの MARCS による制御実験では，多関節型ロボットによる組み立て作業の雛型としてハノイの塔のパズルを実行させ，作業世界の記述性を検討した。また，行動ベースエージェントを用いて試行錯誤的な作業戦略モジュールを実現し，困難性を伴う作業を信頼度よく遂行できることを実証した。

本研究では，エージェントを単なる受動的なデータと見て操作することによってエージェント相互の監視・制御を実現し，ロボットが全体としてある種の内観能力を持つようにすることも提案している。単にマニピュレータを操作するだけの下位のエージェントを，その作業目的に即した高位の判断を行なう別のエージェントが監視し，必要に応じて戦略を変更して下位の作業遂行エージェントに影響を与えうる構成にする。これらのエージェントは独立しているので，このような構成を持つロボットは，全体として見れば，自分が遂行する作業の目的を，動かしている手先の運動とは別のものとして意識していると言うことができる。このような構成法は，ロボットの行動を作業環境の変動に即して柔軟に適応させるための基本的な機構として有効であると考えられる。本章では，こうした構成法の応用として X ウィンドウによるエージェント監視系を実現し，その有効性を示した。

最後に第5章では，本論文全体のまとめを行なう。オブジェクト指向は，数あるプログラミング・パラダイムの中でも，モジュール化技術として極めて優れている。本研究では，Lisp マシン ELIS を開発し，そのマイクロプログラミングによって高性能の Lisp 処理系 TAO を実現した。そして，ELIS の性能を背景に，TAO の核の部分でオブジェクト指向を融合し，オブジェクト指向パラダイムが実用レベルの速度性能で実現できることを示した。さらに，応用プログラムを用いた評価を行なうことによってその有効性を実証的に示し，具体的な実現の諸技術も明らかにしている。また，オブジェクト指向に並行性を与えて，実世界を記述するためにより自然なモデルを提案し，ロボット制御に応用してその有効性を示した。

関連発表論文

● 論文誌・国際会議資料

(1) I. Takeuchi, H. Okuno, and N. Ohsato. A List Processing Language TAO with Multiple Programming Paradigms. *New Generation Computing*, Vol. 4, No. 4, pp. 401–444, 1986.

(2) Hiroshi G. Okuno, Nobuyasu Osato, and Ikuo Takeuchi. Firmware Approach to Fast Lisp Interpreter. In *Proceedings of the 20th Annual Workshop on Microprogramming*, pp. 1–11, Colorado Springs, CO., December 1987.

(3) 大里延康, 竹内郁雄. 複合パラダイム言語 TAO におけるオブジェクト指向プログラミングとその実現. 情報処理学会論文誌, Vol. 30, No. 5, pp. 596–604, 1989.

(4) 大里延康. MARCS: マルチエージェントによるロボット制御. 電子情報通信学会論文誌 D-I, Vol. J75-D-I, No. 8, pp. 714–722, 1992.

(5) Nobuyasu Osato. An Action Interpreter of a Robot Control Agent. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems '93*, pp. 1126–1133, Yokohama, July 1993. IEEE/RSJ.

● 研究会資料等

(6) 日比野靖, 渡邊和文, 大里延康. LISP マシン ELIS の設計. 第 21 回全国大会予稿集, 5J-3, pp. 141. 情報処理学会, 1980.

(7) 大里延康, 渡邊和文. Lisp マシン ELIS の開発環境. 記号処理研究会資料, 17-3, 情報処理学会, 1982.

(8) 竹内郁雄, 奥乃博, 大里延康, 渡邊和文, 日比野靖. New Unified Environment (NUE) の基本構想. 記号処理研究会資料, 18-1, 情報処理学会, 1982.

(9) 竹内郁雄, 奥乃博, 大里延康. Lisp マシン ELIS 上の新 Lisp TAO. 記号処理研究会資料, 20-5, 情報処理学会, 1982.

(10) 日比野靖, 渡邊和文, 大里延康. Lisp マシン Elis のアーキテクチャ — メモリレジスタの汎用化とその効果 —. 記号処理研究会資料, 24-3, 情報処理学会, 1983.

(11) Nobuyasu Osato, Hiroshi G. Okuno, and Ikuo Takeuchi. Object-oriented Programming in Lisp. 記号処理研究会資料, 26-4, 情報処理学会, 1983.

(12) 大里延康, 奥乃博, 竹内郁雄. 新 Lisp TAO におけるオブジェクト指向型プログラミングの実現機構について. 第 26 回全国大会予稿集, 4D-5, 情報処理学会, 1983.

(13) I. Takeuchi, H. G. Okuno, and N. Osato. TAO — A harmonic mean of Lisp, Prolog and Smalltalk. *ACM SIGPLAN NOTICES*, Vol. 18, No. 7, pp. 65–74, July 1983.

(14) 竹内郁雄, 奥乃博, 大里延康. TAO における tag の活用法. 第 26 回全国大会予稿集, 4D-6, 情報処理学会, 1983.

(15) 竹内郁雄, 奥乃博, 大里延康. NUE/TAO/ELIS の OS 的側面. 記号処理研究会資料, 24-8. 情報処理学会, 1984.

(16) 竹内郁雄, 日比野靖, 奥乃博, 大里延康, 渡邊和文. ELIS-TAO の性能評価. 第 28 回全国大会予稿集, 3F-2, pp. 221. 情報処理学会, 1984.

(17) 竹内郁雄, 大里延康. No-method-found ハンドラとその応用. 第 38 回 (昭和 64 年前期) 全国大会予稿集, 3P-5, 情報処理学会, 1989.

(18) 大里延康. エージェントをベースとしたロボット・プログラミング. 春季全国大会予稿集, D-231, pp. 1822–1827, 電子情報通信学会, 1991.

(19) 大里延康, 奥乃博, 大原秀一. マルチエージェントシステムとその行動ベースロボット制御への適用. 日本ソフトウェア科学会第 8 回大会論文集, D1-6, pp. 81–84. 日本ソフトウェア科学会, 1991.

(20) 大里延康. ロボット制御エージェントの行動インタプリタ. 電子情報通信学会技術研究報告, AI92-91, 電子情報通信学会, 1993.

第 2 章

Lisp マシン ELIS とマルチパラダイム言語 TAO

2.1 緒論

近年, 自然言語処理やエクスパートシステムをはじめとする多様で高度な人工知能研究が盛んになり, 動的に生成・消滅・増減する大量のデータを扱ったり, 複雑なアルゴリズムを試行錯誤で開発したり, 高速の処理を必要としたりするようなプログラミングが行なわれるようになった。人工知能のプログラミングを支援する言語処理系としては, 古くから Lisp が用いられてきた。Lisp は動的に大きさの変化するリストの扱いや記号処理に適した機構を備え, 関数の概念で統一されたすっきりした言語である。また, 言語に新たな機能を導入する際にも非常に柔軟性がある。

Lisp は, こうした数々の利点を持つ反面, 処理速度が遅いことや記憶域を多く消費することなどから, 大規模かつ高速な実用プログラミングを行なうには不向きであると言われてきた。しかし, 近年ハードウェアが著しく進歩し, Lisp 専用のアーキテクチャを持つ計算機を開発することによって速度上の問題を改善することができるようになった。また, 比較的安価な大容量メモリが入手できるようになり, 大量のデータ処理を必要とする実用レベルの高度なプログラミング環境を実現することが可能となってきた。

筆者らは, このような背景のもとに Lisp 専用計算機 ELIS を開発し [5] [6] [19] [20], その上に, Lisp をベースとする記号処理言語 TAO を開発した [11] [9] [10] [21] [22] [23] [24] [25] [26] [27]。本章では, Lisp マシン ELIS[1] と, その上の言語 TAO の設計思想およびその概要について述べる。

ELIS では, Lisp の処理を支援するハードウェア資源をマイクロプログラムで直接操作することによって高性能の処理系を実現することを狙った。言語処理系のように抽象度の高い大きなシステムを, ハードウェアに近い低いレベルのマイクロプログラムで直接実現する際には, ハードウェアとの落差をいかにして埋めるかが問題となる。ELIS では, マイクロプログラマから見えるアーキテクチャを極力単純化する, という方針でこれに対処した。Lisp 特有の処理をまとめた複雑なハードウェアを備えることよりも, マイクロプログラムの書きやすさの支援のためにハードウェア資源をつぎこんでいる。マイクロ操作には, タイミング等を含むゲートレベルの煩雑なハードウェア操作はない。マイクロプログラマは, 比較的単純な 3 バス構成のデータパスを意識していればよく, レジスタ間のデータ転送や演算を中心とした, 機械語に近いイメージでのプログラミングができる。これによって大量のマイクロプログラミングが可能となり, システムの高性能化を促進することができた。

[1] **E**lectrical Communication Laboratories **List** Processor から名付けられた。

通常, Lisp の関数や変数には型の宣言がなく, データの内容は実行時に判別する。このため, ELIS ではタグ・アーキテクチャを採用してデータの識別をハードウェアで支援している。また, Lisp では言語の実行管理が重いので, 大容量のハードウェア・スタックを装備してこれをサポートする。スタック・トップのデータは常に専用のレジスタに置いてスタックのアクセスが内部レジスタと同等になるような機構を実現している。Lisp の基本データであるリストは, ポインタの対で構成されるセルをベースとして実現されるため, そのアクセスを高速化するためには, 強力なメモリ・インタフェースを持つことが有効である。ELIS では, 64 ビットの Lisp セルを一回のメモリ操作で読み書きできる。また, メモリ・アドレス・レジスタとしてもメモリ・データ・レジスタとしても機能するメモリ汎用レジスタ (MGR: Memory General Register) と呼ぶレジスタを複数備え, 読み出したポインタ・データをそのままアドレスとして再びバスに送出できるようにするなど, リスト操作を支援する種々のハードウェアを装備している。MGR はデータパスの中に配置されているので汎用レジスタとしても使用でき, 読み出したメモリ・データをそのまま ALU[2]での演算対象とすることができる。さらに, バイト単位で MGR の内容を操作可能とするインデックス・レジスタを設け, 文字列やコンパイルド・コードのためのバッファとしても使用できるようにしている。

マイクロプログラマから見た ELIS のデータパスは比較的単純であるが, シーケンシングは必ずしもそうではない。ELIS のマイクロ命令は 64 ビットの語幅を持つ水平型であるためマイクロ操作も多く, マイクロプログラムの開発負担が大きい。そこで, ELIS のフロントエンド計算機上の小型 Lisp で高機能の支援系を開発し, マイクロプログラマの負担を軽減した [28] [29] [30]。このマイクロプログラミング支援系では Lisp の特性を活かして会話性を高め, Lisp の機能を利用したマクロや, マイクロプログラムの静的なフロー解析に基く WCS へのプログラムの割り付け等を実現した。また, ELIS の主記憶やスタック, レジスタ等の内容を, TAO のデータ構造に従って記号的に表示する機能を備え, ハードウェアを会話的に操作することができる。

言語 TAO の開発目的は, Lisp の性能を向上し, さらに AI 研究のための高性能のプログラミング環境 (これを NUE: New Unified Environment と呼ぶ) [11] を構築していくための核言語を実現することであった。

優れたプログラミング環境の第一の要件は速度性能である。いかに高機能であっても速度が遅ければ有用性は著しく低い。TAO では Lisp としての基本的な速度性能を重視した。特に, インタプリタの性能を最大にして, コンパイルしなくても十分実用になる高速性を追求した。その理由はまず第一に, AI プログラムにはインタプリタ的な実行が少なからず含まれることである。第二に, AI プログラミングでは試行錯誤を繰り返す会話的なプログラミング・スタイルが多いことである。従ってインタプリタの実行速度はプログラム開発時のターン・アラウンドに大きく影響する。第三に, プログラマの意図した実行環境を保持しながら走行するインタプリタは, コンパイラ・ベースでのプログラミングよりデバッグ上有利であることである。

インタプリタの高速化のため, その本体である `eval` をはじめ, 基本関数, 基本メカニズムなど速度性能に影響する殆どすべての機能をマイクロプログラムで記述した。これにより, 処理系のプログラム自体の主記憶からのフェッチがなくなって大幅に速度が向上する。さらに, 専用ハードウェアの活用により, Lisp の十分大きな応用プログラムで DECsystem-2060 上の MacLisp 等のコンパイラに匹敵する速度性能を達成した。また, 商用 Lisp マシン Symbolics-3600 のインタプリタの性能をはるかに凌ぎ, コンパイラにも比肩しうるものとなった。

---

[2]算術論理演算ユニット (ALU: Arithmetic Logic Unit)

これに加え，AI プログラミングの多様な要求に応えるため，プログラミング・パラダイムの選択をユーザの自由に任せることができる複合プログラミング・パラダイムのシステムとすることを TAO の中心思想に据えた。これによって，ユーザは与えられた言語特有のパラダイムに思考を制約されず，プログラミングの労力を自己の持つ問題に集中できる。TAO では，Lisp の持つ柔軟な特性を活かし，複数のプログラミング・パラダイムをいかに効率よく，かつ自然に融合できるかを追求した。そのために，ベースとなる Lisp 処理系の本体である `eval` を見直し，Lisp 本来の逐次型パラダイムに，Prolog [12] に代表される論理型プログラミングの処理機構であるユニフィケーションやバックトラック，Smalltalk [13] に代表されるオブジェクト指向の計算モデルであるメッセージ伝達機構等の本質的部分を処理系の中心部に同化させた。従って，大雑把には，TAO は，ラムダ計算，Horn 節，メッセージ伝達そのほかのセマンティクスを持つ S 式を提供している，と言うことができる。ユーザは，プログラム・モジュールのレベルのみならず，式のレベルでこれらを混ぜ合わせることができる。TAO は，これらのプログラミング・パラダイムを，単にマクロ展開やライブラリ・パッケージによって提供するのではなく，言語の核において融合している。

プログラミング・パラダイムの融合に際しては，Lisp の基本性能を損わないことに重点を置いた。パラダイム融合のためにベースとなる Lisp の性能が犠牲になることは避けなければならない。しかし，それが可能な範囲ではできる限りの機能を盛り込む，という姿勢をとっている。各パラダイムの扱う変数や構造体等はすべて Lisp のデータ構造を用いたのでパラダイム間で情報を変換する必要はなく，処理上の分岐はタグによるマイクロプログラムの多重分岐を用いたので，Lisp へのオーバヘッドはない。

このような複合プログラミング・パラダイムの言語では，融合されたパラダイム同士で性能に不均衡があることは望ましくない。もしいずれかのパラダイムの性能が他のものと比べて悪ければ，そのパラダイムは結局使われず，パラダイム融合の利点も得られない。TAO では，速度性能，メモリ消費，機能性，あるいはプログラミングの容易さ，といった性能のバランスの目標を 2 倍程度として実現上の工夫や機能の取捨選択を行なっている。ユーザは，目的に応じてこれらのプログラミング・パラダイムを適当に選択し組み合わせることができ，パラダイム毎の性能に合わせて自分のプログラミング・スタイルを調整することに神経を使う必要はない。

TAO ではインタプリタの高速化に主眼を置いているが，プログラムをコンパイルすることによって，より高速な実行ができるようになるほか，プログラム自体の占めるメモリ量を削減したり，ソースコードを隠蔽したりすることが可能になる。このため，インタプリタとの両立性を重視したコンパイラも実現している。コンパイラの出力は仮想的なスタック・マシンを操作する中間コードで，そのインタプリタをマイクロプログラムで記述した。

また，TAO は，Lisp マシンのオペレーティング・システムのための基本機能を併せ持ち，マルチプロセス，マルチユーザ，独自のファイル・システムのサポート等を行なっている。従って，TAO は単なる言語処理系ではなく，独立した総合的な Lisp プログラミング環境のためのベースとなっている。Lisp 本体の仕様は Common Lisp [14] に準拠し，全体としては Common Lisp のスーパーセットになっている。

TAO の設計・開発と ELIS のいくつかの部分の開発は相互に影響を与えあっており，両者は一体となったシステムとして成長した。TAO の言語機能を選定する際には，ELIS 上で効率よく実現可能であるかどうかを重視した。また，TAO の要請する機能のうち，実現の比較的容易なものは ELIS のハードウェアに反映された。マイクロプログラムのコーディング上，たとえば，メモリ参照の待ちで無駄時間が生ずる場合等には，その待ち時間の中に埋め込める限り，より高度の機能を導

図 2.1: ELIS 一号機

入する，といった方策をとっている。このように，ELIS と TAO は，きわめて緊密な関係にある。

2.2 Lisp マシン ELIS

ELIS の最初の試作機は，ブレッドボードでビットスライスのマイクロプロセッサとショットキ TTL 論理回路を用いて製作された (図 2.1)。ELIS は，市販のミニコンピュータ等をフロントエンドとするバックエンド・プロセッサである。図 2.1において，試作 ELIS(右) の左奥に見えるのは，フロントエンドとして用いた DEC 社の PDP11/60 である。ELIS は，試作機と同じデバイスを用いてプリント基板化された版が約 20 台製作され，電気通信研究所内で使用された。さらにその後カスタム VLSI チップ化して商用化され，約 200 台余が出荷された。

2.2.1 ELIS のアーキテクチャ

表 2.1に，ELIS 試作一号機の諸元を示す。RALU[3] とシーケンサにビットスライスのマイクロプロセッサを用いて CPU を構成し，マイクロプログラムで制御する。特に，`eval` を全面的にマイクロプログラム化することを前提としており，マイクロプログラマのレベルから見えるアーキテクチャを簡単化する方針をとっている。マシンサイクルを 180*nsec* とし，これとのバランスから数 100*nsec* 程度のアクセスタイムのメモリを装備している[4]。メモリの仮想化は行なわず，128M バイトまでのアドレス空間を持つ。仮想メモリを採用しなかったためハードウェアは単純化され，メモリのアクセスタイムを明確に把握し，これを言語設計上の機能選択の判断にまで意識したマイクロ

[3]RALU: レジスタと ALU を合わせたもの。Register ALU

[4]このマシンサイクルの設計値は，1980 年代初期のショットキ TTL を用いた CPU としては標準的なものである。後の VLSI 版では CMOS 論理回路が用いられ，マシンサイクルも短縮された。

表 2.1: ELIS 試作一号機の諸元

プロセッサ	内部バス幅	32 ビット
	内部レジスタ	32 語 (32 ビット)
	メモリ汎用レジスタ (MGR)	4 組 (64 ビット)
	ソース行先カウンタ (SDC)	3 組 (5 ビット)
	スタックポインタ	3 組 (14 ビット)
	スタック	16K 語 (32 ビット / 語)
	制御記憶	16K 語 (64 ビット / 語)
	マシンサイクル	180*nsec* (可変)
	制御方式	マイクロプログラム
	(RALU)	(Am2093 x 8)
メモリ	アドレス形式	バイトアドレス
	読み出し幅	64 ビット
	アドレス空間	2^{27} バイト
	最大実装容量	8M バイト *
	アクセスタイム	380*nsec*
	サイクルタイム	550*nsec*
	誤り訂正符号	16 ビット毎 1 ビット誤り訂正

* 16K ビット / チップ使用の場合。後に商用化された版では 128MB まで実装できる。

コーディングが可能となった。図 2.2に, データパスを中心とするアーキテクチャを示す。ELIS のアークキテクチャ上の特徴を列挙すると, 以下のようになる。

(1) タグ・アーキテクチャ
32 ビット長の語の中で最上位の 1 バイトはタグである。タグによって最大 256 方向までのマイクロプログラムの分岐ができる。図 2.3に, タグを含む ELIS の Lisp ポインタを示す。また, 表 2.2に, タグによる分岐条件を示す。

(2) ハードウェア・スタック
スタティック RAM を用いた大容量 (32K 語) の高速のスタックを持つ。スタックを参照するために 3 つのスタック・ポインタがある。スタック・トップのデータは内部レジスタと同じ速さで参照でき, push や pop は, 1 マシンサイクルで実行される。スタックは 16 のブロックに分割され, ブロックの境界を越えるスタック・オーバフローとアンダフローは, ハードウェアでチェックされる。TAO では,. このスタックを用いて深いバインディングを採用した[5]。また, 高速のプロセス・スイッチが実現されている。

[5]Deep binding: 変数の値を, スタック上の環境の連鎖を辿りながら, 変数名をキーにして探索する。これに対し, 浅いバインディング (Shallow binding) では, 変数名に付随する値のスロット (value cell) を, 環境をキーにして探索する。TAO では, 後述するように, 並行プログラミングの実現性等を考慮して, 深いバインディングを採用している。

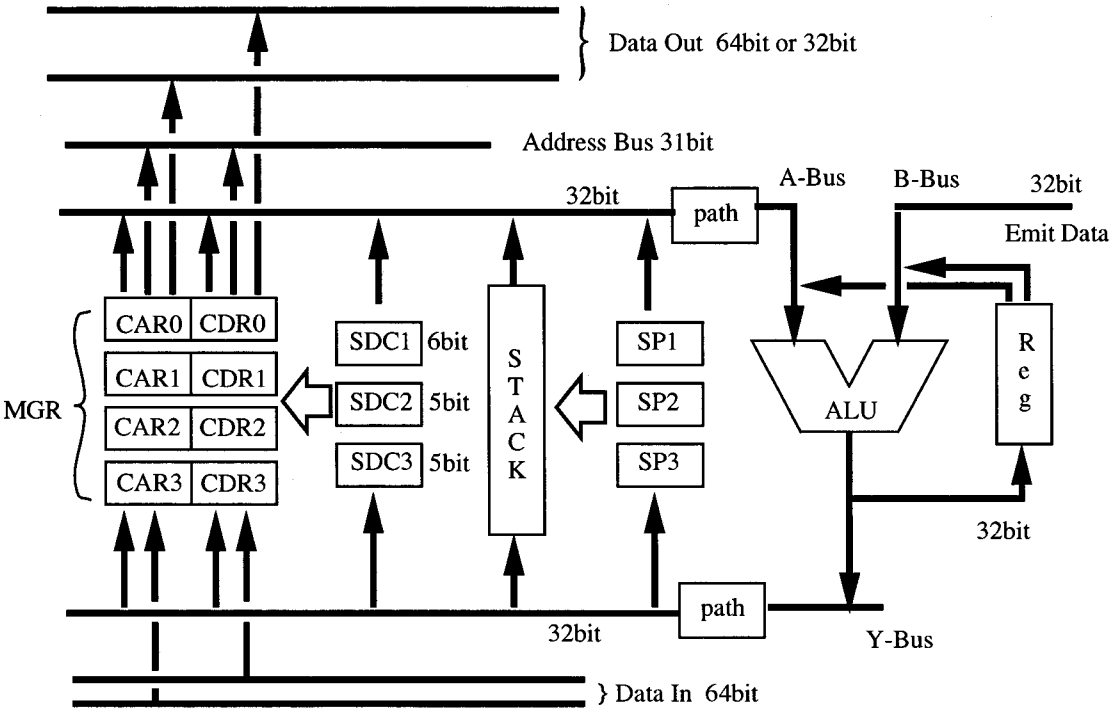


図 2.2: ELIS のデータバス

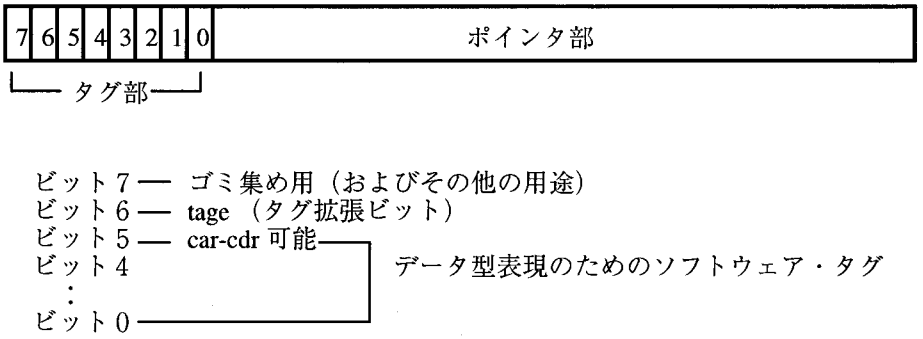


図 2.3: Lisp ポインタとタグ

表 2.2: タグによる分岐

条件 (ニモニック)	意味
tag7	ビット 7 がオンならば分岐。
tag6, tage	ビット 6 がオンならば分岐。
tag5, tagcadbl	ビット 5 がオンならば分岐。
tag5-0	ビット 5-0 による 64 方向分岐。
tagh5-0	33 方向分岐。ビット 4 がオンなら 33 番目のオフセットに飛ぶ。
tagl5-0	33 方向分岐。ビット 5 がオフなら 33 番めのオフセットに飛ぶ。
tag4-0	ビット 4-0 による 32 方向分岐。
tagfil	ビット 5-0 がゼロでないとき分岐。
tagnil	ビット 5-0 がゼロのとき分岐。

(3) 大容量の WCS
64 ビット語長 64K 語の WCS を持つ[6]。スタック・メモリと同じくスタティック RAM を用いている。WCS へのマイクロプログラムのローディングはフロントエンドが行なう。

(4) CPU– メモリ間インタフェース
メモリ・インタフェースは 64 ビットのバス幅を持ち, Lisp の一つのセル (car と cdr) の読み書きは 1 回のメモリ操作で実行できる。マシンサイクルはメモリアクセスの 1/3 なので, メモリ参照を起動してからそのデータが使用可能となるまでの間に, メモリ動作と並列に 3 ステップのマイクロ命令を実行できる。TAO の言語機能の採否には, この car や cdr 等の基本的なメモリ操作の陰で実行可能か否かが重要な考慮点となった。

(5) メモリ汎用レジスタ
メモリ汎用レジスタ (MGR) と呼ぶ 64 ビットのレジスタを 4 つ持つ。各レジスタにはそれぞれ 32 ビットの car 部と cdr 部がある。それらは, 通常の汎用レジスタとしても, メモリ・アドレス・レジスタとしても, また, メモリ・データ・レジスタとしても使用できる。従って, リストを辿る際にも, 次のポインタをアクセスするときに, 読み出したメモリ・データをアドレス・レジスタに転送する必要はない。また, MGR の任意のバイト位置を指すことのできるインデックス・レジスタ SDC[7]が 3 本ある。SDC を用いると, MGR をバイト列のための短いキャッシュとして使用することができる。これによって文字列操作が高速化される。またコンパイルド・コードを実行するときのコマンド・バッファとしても使用できる。

(6) メモリ・アクセスのハードウェアによる検査
メモリ参照のとき, タグの内容が, car や cdr 等の操作が許されるデータであるかどうかを, メモリ起動と並行して検査することができる。もし car や cdr ができないデータの場合には,

[6]WCS は, 当初 16K 語であったが後に 4 倍の 64K 語に拡大された。

[7]SDC: Source Destination Counter (ソース行先カウンタ)。SDC は, MGR の間接指定をするためのレジスタとしてのほか, カウンタとしても動作する。

そのメモリ操作は自動的に中断される。この機能を用いると，典型的なリスト辿りの場合，1セルあたり 1/3 のマイクロステップを節約できる。

2.2.2 ハードウェア / ファームウェア開発環境 MIC

ELIS のような専用計算機を自作するにあたって，開発のための支援ツールとしてどのようなものを用いるかということは，ひとつの問題点である。ハードウェア開発時には，ハードウェアの状態の観測やテストが効率的かつ容易にできる必要があり，マイクロプログラム開発時には，プログラミング支援ツールの充実が重要となる。

ELIS の開発では，フロントエンド側から会話型の高級言語を使って種々の操作を行なうことによって，柔軟で高機能の支援環境を実現するという方針をとった。ELIS の制御や診断をフロントエンド側に行なわせることによって ELIS 自身のハードウェアを簡略化できる。フロントエンド計算機上には ELIS に先立って開発した小型 Lisp, TAO/60 [31] があり，これを用いて，ELIS のハードウェア / ファームウェアの開発支援環境を整えた。この支援環境を MIC と呼ぶ。

多くの並列フィールドを持つ水平型マイクロプログラミングでは，効率のよいコードを生成するマイクロコンパイラの技術は成熟していないので，MIC ではコンパイラを作ることはせず，アセンブラ，リンカ等の単純なツールの能力を高度化した。また，ELIS のハードウェアを操作するツールを充実し，メモリやデータパスの状態を会話的に調べる機能を強化した。

MIC によるマイクロプログラム開発サイクルを図 2.4に示す。ソースファイルのマイクロプログラムはマイクロアセンブラによってアセンブルされ，再配置可能なオブジェクトモジュールになる。マイクロリンカは，このオブジェクトモジュールの各マイクロ命令に WCS の物理アドレスを割り当てて，WCS ロードモジュールを作る。WCS ロードモジュールはローダによって WCS にロードされ，ホストからのコマンドによって走行する。

MIC は当初 TAO/60 で稼動したが，ELIS 上の TAO の稼動開始後に TAO に移植し，ELIS 自身でファームウェアの開発を行なった。ELIS 自身には自己の WCS を操作する機能はないので，WCS ローダ等はフロントエンド側で稼動する[8]。

2.2.2.1 ELIS とホスト計算機とのインタフェース

図 2.5に，フロントエンドのホスト計算機と ELIS とのインタフェース部分の構成を示す。この図は，最初のホスト計算機である PDP11/60 の場合を示しているが，これ以降のホストについても基本的な構成は同様である。ELIS の制御用レジスタの主なものは次のとおりである。

(i) WCSADR, WCSDR0 ～ WCSDR3(WCS インタフェース・レジスタ)
WCS を読み書きするためのレジスタ。ホスト (16 ビット) と ELIS の WCS(64 ビット) の語幅の差のため，データ転送を 4 回に分けて行なう。アドレス・レジスタと 4 個のデータ・レジスタがある。

[8]ELIS の後継機，すなわちプリント基板の版ではフロントエンドが LSI-11 になり，TAO/60 の機能のうち MIC に必要な部分のみを実現した ULISP という小型 Lisp を LSI-11 上に作成して，ELIS ハードウェア制御の機能を引き継いだ。また，VLSI 版の ELIS では，フロントエンドが UNIX 系のワークステーションになり，ULISP が C 言語で書き直された。

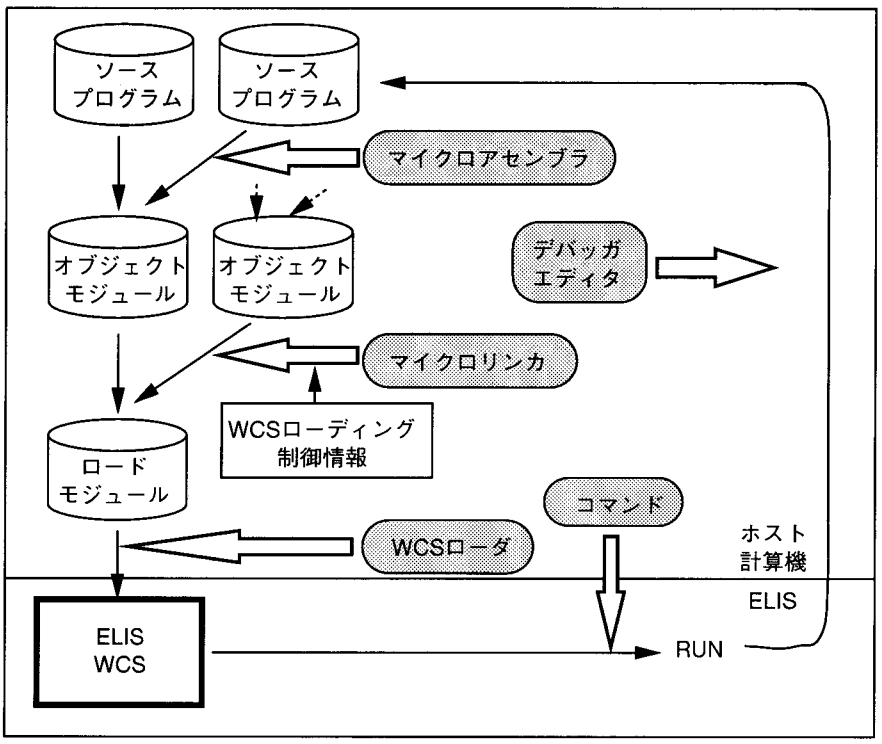


図 2.4: マイクロプログラム開発サイクル

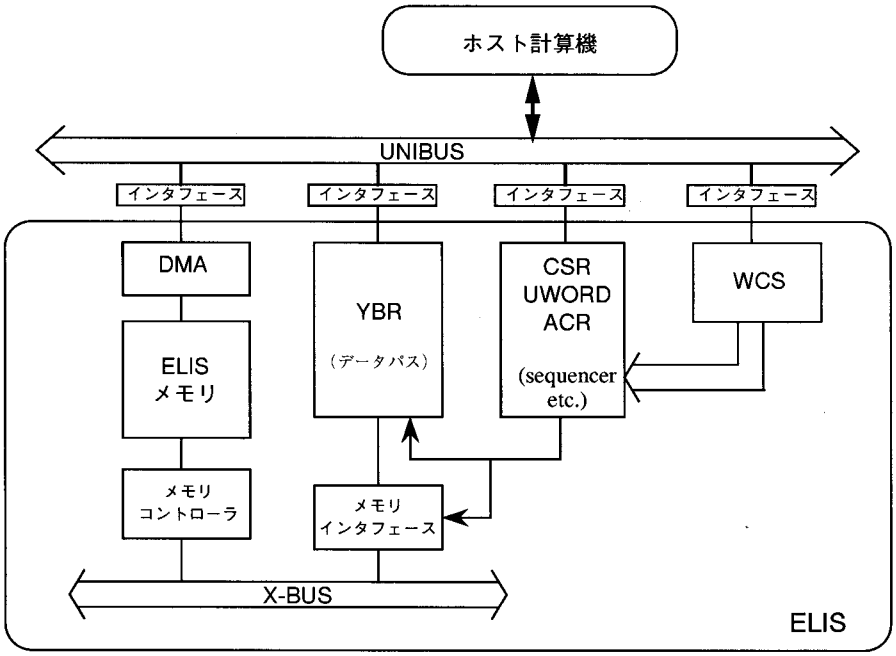


図 2.5: ELIS インタフェース

表 2.3: CSR コマンド

コマンド	ELIS の動作
DEVCLR	ハードウェア初期化
ERRCLR	エラークリア
STOP	停止 (停止していれば Nop)
DME	直接マイクロ命令実行
USTEP	マイクロ命令のステップ実行
STEP	S 式単位のステップ実行
RUN	走行 (現在の PC の場所から)

(ii) CSR0, CSR1(コマンド / 状態レジスタ)
ホスト側から見える ELIS の内部状態は, 殆どこの 2 つのレジスタに集められている。また, ELIS に対するコマンド送出やマイクロプログラムカウンタ (UPC) の読み書きも CSR 経由で行なう。

(iii) YBR0, YBR1(ALU 演算結果レジスタ)
ALU の演算結果は, マイクロサイクル毎に YBR という 32 ビットのレジスタにセットされる。YBR は, ホスト側からは 2 つの 16 ビットのレジスタ YBR0 および YBR1 として読める。

(iv) UWORD0 ～ UWORD3(直接マイクロ実行レジスタ)
ELIS は, WCS からマイクロ命令を読み出して実行するだけでなく, ホスト側から直接送られたマイクロ命令も実行できる。この機能を直接実行 (Direct Micro Execution: DME と略す) と呼ぶ。DME を行なうには, この 4 個の UWORD レジスタにマイクロ命令を書き, 続いて CSR0 経由でコマンドを送る。

上記 (i) ～ (iv) 以外に, ELIS とホストとの間には DMA による転送パスおよび, ACR と呼ぶバイト単位の通信レジスタ等がある。これらはすべて, ホスト計算機のシステムバス上のアドレスを割り当てられた, 外部 I/O レジスタになっている。CSR 経由で ELIS に送られるコマンドの一覧を表 2.3に示す。

TAO/60 では, **#num** と呼ぶビット列のデータ型をアドレスまたはデータとして, ホストのシステムバスを直接アクセスする関数 **unibus** を作った。**unibus** は,

(**unibus** *address* {*data*})

という形をしている[9]。*address* は操作対象レジスタのアドレス, *data* は書き込みデータである。*data* を省略すると読み出し操作になる。**#num** に対するビット操作関数と **unibus** を併用することにより, ELIS のハードウェアの操作が, 会話的に極めて容易にできる。ELIS のハードウェアと TAO/60 のインタフェースは, 割り込み以外は, 関数 **unibus** のみであり, マイクロアセンブラ, マイクロリンカ, WCS ローダは, TAO/60 だけですべて記述できた。

[9]{ } は, 随意引数を表わす。

2.2.2.2 マイクロアセンブラ

ELIS のマイクロ命令は図 2.6に示すように Type フィールドに応じて 4 つの型に分類される[10]。I 型はメモリ参照の型, II 型は SDC 制御の型, III 型は定数エミットの型である。

MIC アセンブラでは, ソースファイルをアセンブルできるだけでなく, 単体で与えられたマイクロ命令をアセンブルしてそのオブジェクト語を作ることができるようにし, オンラインで DME 機能を用いたテストを行なったり, WCS の逐語的なパッチが可能なようにした。マイクロ命令を S 式で表現し, Lisp で直接操作するので, こうしたことが非常に容易に実現できた。

マイクロ命令の基本形は, フィールド名とそのフィールドに対する操作とをリストにしたもののリストである。ELIS のマイクロ命令はフィールドの数が多いため, デフォールトパタンを埋め込む機能をアセンブラに実現[11]するとともに, マクロ定義を可能とした。さらに, ビットパタンとしては物理的に可能でも実行するとハードウェア的に問題のあるマイクロ命令等を, アセンブラで検出して排除する。実際のマイクロプログラムは, いくつかのフィールド操作をまとめたマクロが多く用いられ, マイクロ命令やマクロ定義, フィールド定義等のアセンブラに対する命令の並びになる。付録 C にマイクロプログラムの例として, バイナリ・サーチのルーティンを示す[12]。MIC のアセンブラは Lisp で作成したため, フィールドの値の計算を Lisp に戻して行なうなどの高機能化を容易に行なうことができた。

2.2.2.3 マイクロリンカ

ELIS のシーケンサは, WCS 読み出しアドレスのためのソースとして,

1) 外部からの直接入力,

2) 4 レベルまでの深さを持った内部スタック,

3) 通常, 前のアドレスに 1 を加えた値が入る UPC[13],

などから選択する。I, II 型のマイクロ命令では 1) と 2) の機能を, III 型のマイクロ命令では 3) の機能を用いている[14]。条件分岐のためのアドレス修飾は, ワイアド OR で行なわれる。

このように ELIS のシーケンシング機能は多様で, マイクロ命令を WCS の物理的なアドレスに割り付ける作業は大きな負担になる。また, マイクロプログラムの量が多いので, 機能別にモジュール化して開発管理を行なう必要がある。MIC では, マイクロアセンブラで再配置可能なオブジェクトモジュールを出力し, リンカによって実際のアドレスを決めるという, 通常の機械語レベルのアセンブラと同様の方法を採った。しかし機械語レベルのリンカと異なり, マイクロプログラムの

---

[10]第 4 の型は拡張用である。

[11]デフォールトの埋め込みでは, マイクロ命令の型の決定が問題となる。型は Type フィールドで決まるが, 他のフィールド指定から型が決定できる場合には Type フィールドは省略したい。さらに, 型が一意に決まらない場合でも Type を指定するのは煩わしいことが多い。MIC では, 陽に指定されたフィールドから型が決定できないときには, 予め型に順序づけをしておき, それに基づいてデフォールト値を埋め込む。このため, 何をデフォールトとするかはユーザが随時変更できるようにした。

[12]このプログラムは, オブジェクト指向におけるメソッド探索のためのバイナリ・サーチで用いている。

[13]マイクロプログラムカウンタ

[14]III 型のうち, サブルーティン戻り命令のときは, 2) のポップの機能を使う。

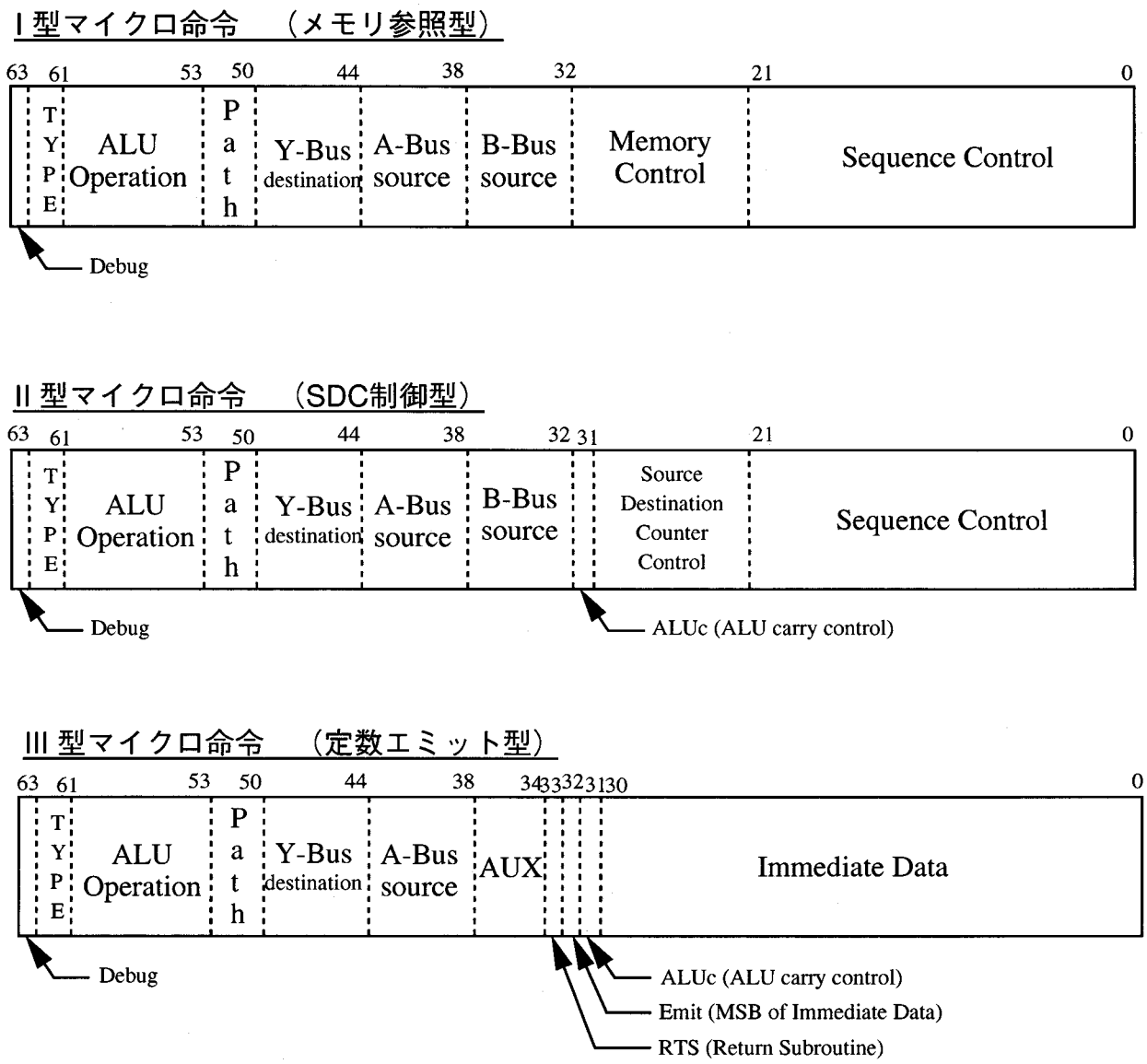


図 2.6: マイクロ命令形式

ソース上の命令の順序は, 実際に WCS にロードされたときの順序とは殆ど対応しない。その理由は, シーケンシングの型によって, アドレス割り付けに以下のような制限があるためである。

(1) III 型 (UPC を自動的にインクリメントする型) のマイクロ命令が連続している。
→ WCS 上の連続番地 (以下これを**ブロック**と呼ぶ) に割り付ける。III 型命令ブロックの次に来る命令は, それが III 型でなくてもその連続番地の末尾に割り付ける。即ち, 連続する n 個の III 型命令があったとき WCS 上で必要な連続番地は $n+1$ 個である。なお, ブロック自体の配置は任意でよい。

(2) n 方向条件分岐の分岐先マイクロ命令。
→ 大きさ n のブロックに割り付ける。ただし, 原則として n は 2 の冪で, かつこのブロック (以下これを**分岐ブロック**と呼ぶ) の先頭番地は n の倍数でなければならない[15]。

(3) サブルーティンジャンプ命令。
→ 上記 2) のスタックを用いたサブルーティンでは, その命令の置かれているアドレスの次のアドレスが戻り先としてスタックに積まれる。従って, サブルーティンジャンプ命令と戻り先の命令は, 連続番地に割り付けられる必要がある。

実際のマイクロプログラムには, これらのものが複合して現われる。たとえば, III 型の命令の次の命令が分岐ブロックの先頭になっている場合や, 逆に, 分岐ブロックの最後のアドレスに置かれた命令が III 型である場合, さらに, それらの両方になっている場合等である。図 2.7の例では, 連続する 3 マイクロ命令 (ラベル **a** のマイクロ命令で始まるもの) が III 型で, その最後のマイクロ命令 (ラベル **b**) の次の命令が 4 方向分岐命令 (ラベル **branch**) の分岐ブロックの先頭 (ラベル **yes**) になっている。これに加え, 同分岐ブロックの末尾のマイクロ命令 (ラベル **no**) は III 型で, その次の命令も III 型である。この場合, 大きさ $9(=3+4+2)$ のブロックが必要で, その中の第 4 番目のアドレスは分岐ブロックの先頭になっているので, 下位 2 ビットが 0(即ち 4 の倍数であるようなアドレス) になっていなければならない。

MIC のマイクロリンカでは, 次のようなアドレス割り付けアルゴリズムを用いた。

(1) プログラムの制御の流れを解析し, 図 2.8に例示するようなフローグラフを作る。すなわち,

(i) アセンブラの出力オブジェクトの各マイクロ命令に, 制御の流れを表わすポインタ・フィールドを設ける。

(ii) 分岐ブロック, III 型命令のブロック, サブルーティンジャンプ命令等, 連続番地を必要とするマイクロ命令群を検出し, 次アドレスの指定があればそのマイクロ命令へのポインタをポインタ・フィールドに格納し, 次アドレスの指定のない III 型以外の命令のポインタ・フィールドにはソース上での次の命令へのポインタを格納する。

(iii) 各群には, 分岐ブロックか III 型のブロックか等の区別や, その大きさ等を記録する。
図 2.8の例では, 4 つの群がある。マイクロ命令 1 は 4 方向分岐, 2 〜 5 は 1 からの分岐ブロック, 6 は単独のマイクロ命令, 7 〜 9 は III 型命令のブロックである。図 2.8の中の矢印は, ポインタ・フィールドの指示を示している。

[15] ワイアド OR による条件分岐を行なうので下位 $\log_2 n$ ビットが 0 である必要がある。

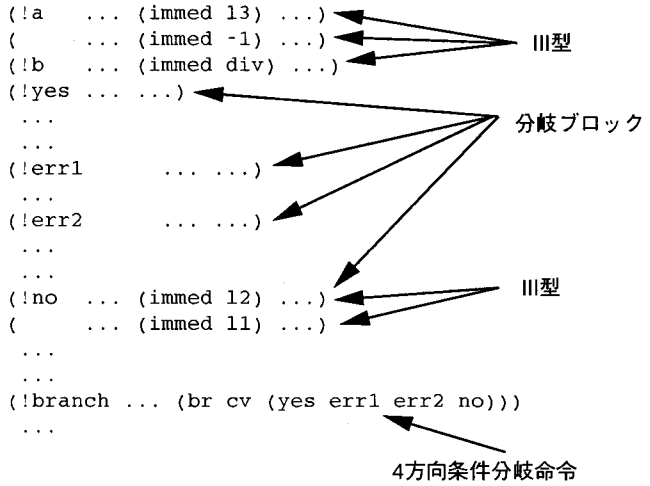


図 2.7: アドレス割り付けに制限のあるブロックが複合している例 (III 型 - 分岐ブロック -III 型の形になっている)

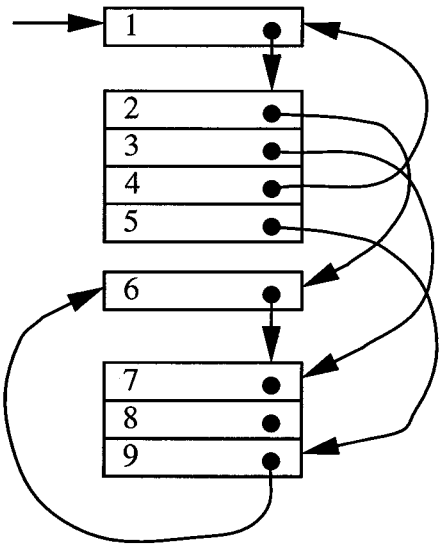


図 2.8: マイクロプログラムのフローグラフ例

(2) (1) で作ったグラフの節点 (各節点はマイクロ命令の塊である) は, WCS アドレス割り付けの単位になる。各節点に, 割り付けの自由性に応じて以下のように順序をつける。

(i) 絶対番地を割り付けるようソースレベルで指示されている節点 (命令)。

(ii) 分岐ブロックを含む節点。大きいブロックのものほど高い優先度を与える。

(iii) III 型命令のブロックである節点と, サブルーティンジャンプ命令に伴うブロックである節点。ブロックの大きいものほど高い優先順位を与える。

(iv) 単一のマイクロ命令の節点。

(3) (2) でつけた順序に従って各命令にアドレスを割り付けていく。このとき, 命令自身の割り付けられるアドレスと, `Sequence Control` フィールド (III 型以外) に埋め込むべき次アドレス, および III 型で `Immediate Data` フィールドに記号番地が書かれているものについて, アドレス割り付けによって確定した値が, それぞれ入る。

アドレス割り付けでは, 各ビットが WCS のアドレスに対応するビットテーブルを用いて空き番地を管理する。分岐ブロックのための 2 の冪の大きさを持つ空き領域は, このテーブル上で **#num** の論理演算を用いてパタンマッチングで探索する。

(4) (1) ～ (3) の手順によって実際にマイクロ命令に割り付けられるアドレスは, もとのソースプログラムの命令の並びから大きくはずれたものとなる。どの命令がどこに割り付けられたかを知ることが, 特にデバッグ時に必要になるため, WCS のマップファイルを作る。

WCS アドレス割り付けではグラフを操作するため, MIC を Lisp で記述したことが非常に有効であった。

2.2.2.4 マイクロプログラムのデバッグ

ELIS は, デバッグのためのハードウェア機能を種々持っている。これをホスト側から制御し, 簡易型のデバッガを作成することができる。いくつかの試験的なトレーサ等を作成し, ハードウェアのデバッグ等に用いた。しかし, TAO のファームウェア開発ではデバッグ対象プログラムのダイナミック・ステップが大きく, トレーサやソフトウェアのシミュレータは殆ど現実的なユーティリティになりえない。

ELIS のマイクロ命令には 1 ビットの `Debug` フィールドがある。ELIS は, デバッグ・モードでは, このフィールドがオンであるマイクロ命令を実行すると停止する。MIC の WCS パッチの機能を用いて適当な命令に `Debug` ビットを立ててトラップを設定し, デバッグ・モードで走行させて停止したときのデータパスの内容を調べる, という方法でプログラムのデバッグを行なった。このように, デバッグではポストモーテムでメモリやレジスタ等に残るデータ, スタックの履歴等の意味を問題にする局面が大半であり, それらを, タグやデータ構造等, 意味に依存する形式でダンプする単純なユーティリティが最も有効であった。

図 2.9に, ホストからのコマンドによって ELIS のレジスタやスタックおよび主記憶の内容を調べている様子を例示する。データパスの内容を見るコマンドは, 主に 1 ステップ・アセンブルによって作ったマイクロ命令を, DME の機能を用いて実行し, その結果を表示している。

スタックポインタレジスタsp1をYBRに出す。

タグと十進および八進でデータを表示する。

スタックの内容を見る。

GCマークビット, tage, タグおよびポインタ部をそれぞれ見やすい形式で表示する。

MGRの内容を表示する。

メモリの内容をダンプする。

4104というアドレスをもつidの印刷名を知るためにダンプする。システムは, idの内部構造にもとづいて印刷名を表わす文字列領域を表示する。この例ではpackageというidである。

```
Fep> mov sp1
         dnil     32239        (#76757)
Fep> stkins #76757 5
  addr       data        ; msb tage   tag       ptr
#76757            #0   ;           dnil         0
#76760            #0   ;           dnil         0
#76761   #3140077132   ;          stkpt#40077132
#76762   #3100477051   ;          stkpt  #477051
#76763            #0   ;           dnil         0
Fep> frmmgr
car0=          dnil        0        (#0)
cdr0=          dnil      512        (#1000)
car1=          cell  3125574        (#4013730506)
cdr1=          dnil        0        (#0)
car2=           udo    36051        (#3200106323)
cdr2=          dnil   142074        (#425372)
car3=           #75    65543        (#7500200007)
cdr3=   tage quoted  8373746        (#14437742762)
Fep> dump 36051 3
addr                tag      car            tag      cdr
 36051           vector    38133  1      shtnum       64
 36052           vector    37743             id     4104
 36053     tage shtnum       17              id     4107
Fep> dump-id 4104
offset = 3, length = 7
addr
 84203   #141 #153 #143 #141 #160 #145 #143   #0 ; akcapec.  ; .cepacka
 84204   #165 #164 #141 #164 #163 #145 #147   #0 ; utatseg.  ; .gestatu
Fep>
```

図 2.9: フロントエンドから ELIS のハードウェアを調べる

2.3 マルチパラダイム言語 TAO

本節では, プログラミング言語 TAO の概要を述べ, 最大限の機能性と最小の実行時コストとを狙いとして, ELIS 上にどのようにして TAO が設計されたかについて述べる。

複数のプログラミング・パラダイムを用いてプログラムを書く場合, 別々の言語で作成したたくさんのプログラム・モジュールを結合したりするより, 必要なプログラミング・パラダイムが融合された単一のプログラミング言語を用いるほうがよい。しかし, そうした言語は, 注意深く設計され, うまい実現法がとられていなければ, 見にくくかつ能率の悪いプログラミングを強いられることになりがちである。

言語にプログラミング・パラダイムを採り入れる際には, 実現性についての考察が重要である。あるパラダイムに, ほかのパラダイムと比べて遅すぎたりメモリを消費しすぎたりするといった実現上の問題があるならば, そのパラダイムはほかのパラダイムとの調和が悪いということになる。TAO の設計では,

(1) 単純で調和のとれた融合をはかること, および,

(2) 融合のためにどのパラダイムに対しても犠牲をもたらさないこと,

に重点を置いている。

TAO では**手続き型**, **論理型**, **オブジェクト指向**をサポートする実用的な AI プログラミング言語を目指している。これら三つのパラダイムについては, いくつかのベンチマークテストに対し, パラダイム間の実行速度性能の比を 1 から 2 までの間に収めることを目標とした。これは, できるだけ 1 に近いことが望ましいが, 実現技術上の検討から, この範囲が妥当な目標点であると判断した。一般に, 性能の比率がこの程度の範囲にあれば, ユーザは, 実行時の速度性能を気にすることなく, 好きなプログラミング・パラダイムを選択できると考えられる。このことは, 複合プログラミング・パラダイム言語における最も重要な点の一つである。

TAO の特徴は, 以下のように列挙することができる。

(1) インタプリタの性能を重視した Lisp 処理系を基本とする。TAO はいろいろな面で ZetaLisp に類似し, Common Lisp に対しては上位互換である。TAO の言語機構はいくつかの点で基本的に Common Lisp と異なるので, そうした局面ではエミュレーションを行なっている。TAO は 1000 以上の基本関数を持つが, そのうちの半数をマイクロコード化した。

(2) タグを活用し, 他の Lisp 言語には見られないデータ型とそれに関連する関数群を備えている。たとえば, 末尾の方でも成長するリストとか, 日本語の文字と ASCII の文字を混在させるのに適した文字列のデータ型等がある。

(3) S 式は, より多くの意味を持ち得るように拡張されている。

(4) 他の Lisp 言語における `setf` に類似した, 一般化された代入機構を持つ。これはマクロ定義ではなく, インタプリタで直接実行している。

(5) Horn 節, ユニフィケーション, バックトラックをベースにした論理型プログラミングをサポートする。これを用いて, 効率のよい Prolog 風の言語が容易に TAO の中に埋め込める。ユニフィケーションと通常の関数呼び出しは互いに入れ子の構造になっていてもよい。ユニフィケーションで作り出される構造は, 通常のリスト処理関数を用いて直接に操作できる。

(6) 本論文第3章で詳しく述べるように, ZetaLisp の Flavors に類似したオブジェクト指向パラダイムを持つ。しかし, Flavors と違って, すべての基本データ型はオブジェクトである。また, Smalltalk 風のメッセージ伝達のシンタックスを与えている。さらに, オブジェクト指向と論理型のパラダイムを融合している [32]。

(7) 並行プログラミングとマルチユーザをサポートしている。プロセスの同期, 排他制御およびプロセス割り込みのためのプリミティブがある。

(8) ELIS のメモリ中の語をオブジェクトとして扱うことによって, アドレスデータ型を組み込んでいる。これにより, Fortran あるいは C 言語風の, ゴミなしの計算が可能である。

(9) ゴミ集めは通常のマーク・スイープ方式である。これもマイクロプログラムで実現され, ELIS が 16 メガバイトのメモリを装備しているときでも, わずか 2 〜 3 秒かかるだけである。ほかのプロセスと並列に走る実時間ゴミ集めも試作された。

2.3.1 インタプリタの重視

TAO の設計において強調すべき点は, インタプリタがコンパイラに優先するという点である。Lisp のインタプリタは, 以下の三つの観点から本質的重要性を持つと考えられる。

- **アプリケーション**
  AI のプログラミングでは, 多くの局面で本質的にインタプリタ的な実行を含みうる。たとえば, エクスパートシステムの中には, ルールのコンパイラがなく, ユーザの定義したプログラムがインタプリタで実行されるものも多い。

- **プログラミング環境**
  インタプリタでの実行は, コンパイラでの実行に比べて会話性が高い。ステッパやエディタ, トレーサ, エラー・ブレイク機能等はインタプリタ・ベースで最も有効に機能する。

- **デバッガ**
  最も有効な Lisp プログラムのデバッガの一つはインタプリタであると言える。インタプリタは, プログラマの意図した実行環境を保持したまま走行するので, ユーザにとって最も容易で明解なツールである。

こうした観点から, TAO では高速のインタプリタを, 単に計算の機構だけでなく, 様々のツール類とともに提供した。インタプリタの速度を遅くするような機能や, 実行時にレキシカルな情報をあまりに多く必要とするような機能は TAO では採用しなかった。これは, Common Lisp や CommonLoops の設計方針と対比されるところである。また, ELIS の性能を生かし, `eval` 全体のほか, 各パラダイムにおける重要な処理をすべてファームウェア化してインタプリタの高速化を図った。

このように, TAO ではインタプリタで実用規模のプログラムが高速に走るようにすることを主眼とした。しかし, コンパイルすることによってプログラムの占めるリスト領域の節約やアクセスの高速化が図れるので, インタプリタとの両立性を重視したコンパイラも実現した [33] [34]。コンパイルされたプログラムも S 式のままのプログラムも混在して走らせることができ, デバッグ情報も, コンパイルしたために大幅に不足するということがないように配慮されている。コンパイラは仮想的なバイトコード・マシンの機械語プログラムを出力し, これをマイクロプログラムによるインタプリタが解釈実行する。

2.3.2 タグの利用とデータ型

TAO では, タグを利用して多様なデータ型や様々な言語機能を実現した。これらのデータ型は, 言語仕様としてユーザに見えるもの以外に, マイクロプログラム・レベルでのみ用いているものも多い。本節では, TAO におけるタグの利用法を示し, 所望の機能と性能を達成するためにどのようなデータ型を実現したかを述べる。

TAO のタグは, データの実体の側についている自己記述標識ではなく, それを指すポインタの側についているポインタ・タグである。図 2.3に示したように, 8 ビットのタグのうち, 下位 6 ビットをソフトウェア・タグと呼び, データ型の識別に用いた。残り 2 ビットのうち最上位のビット 7 はゴミ集め用, ビット 6 は **tage** ビットと呼び, ソフトウェア・タグの機能拡張用である[16]。 **tage** は, 計算の本質にはあまり関係のない, データの小さな相違を表現したりするのに用いている。たとえば, TAO では, 空リストである ( ) と, 論理的な偽を意味する **nil** とを, **tage** ビットを用いて区別している[17]。これらは入出力時に区別されるだけで, 計算機構の上では区別されない。

Lisp データの識別では, あるデータが **car** や **cdr** の操作対象であるか否かの判定が重要である。ソフトウェア・タグの最上位ビット (ビット 5) がこれを表わし, データ型に対するタグの割り当ても, この原則によって行なった。このビットは, メモリ操作のハードウェアが直接参照し, ポインタとしての有効性をチェックする。

TAO におけるタグの利用法は, 以下のようにまとめることができる。

1. データ型の表現と, 内部的なデータ型の実現。

2. インタプリタの高速化とメモリ消費の削減。

3. S 式の可読性の向上。

4. 新しい計算パラダイム実現の機構。

表 2.4に, TAO における主要なタグの利用法とデータ型を示す。

タグの特徴的な利用法にインビジブル・ポインタがある。インビジブル・ポインタは, ユーザからは見えない隠し情報をリスト等の通常のデータ構造に保持する手段である。インビジブル・ポインタは極めて利用価値が高く, インタプリタを高速化したり, プログラムの読みやすさを向上するなど, 様々の目的で使用している。例として, **car** を置き換える **evalcdr** によるマクロ展開式につ

[16] tag extension の意味で **tage** と略称する

[17] TAO では, **nil** はシンボルではなく, データ型の一種である。

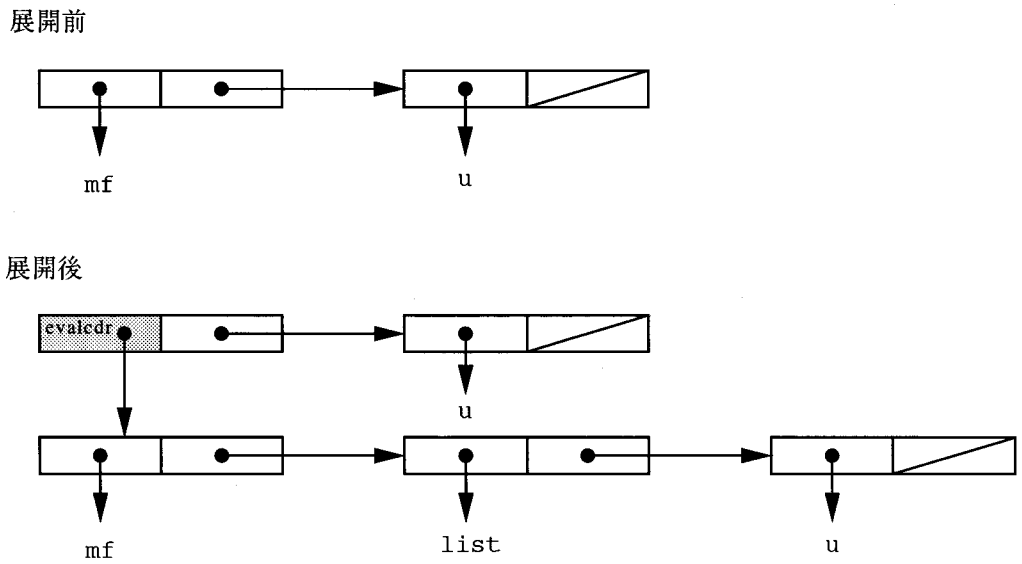


図 2.10: **evalcdr** によるマクロ展開

いて述べる。マクロは一旦展開されると，もとの式の **car** がタグ **evalcdr** を持つ Lisp ポインタで置き換えられる。このポインタは，**car** がもとのセルの **car** の内容を持ち，**cdr** がマクロ展開で作られた新しい式を持つような，新たなセルを指す。**eval** は **evalcdr** を見ると常にそのポインタの **cdr** 側のみを評価すべき式とみなす。たとえば，

```
(defmacro mf (x) `(list ,x))
(defun fn (u) (mf u))
```

というように，関数 **fn** が，マクロ **mf** で展開される本体を持つとする。図 2.10に，マクロ展開前後の **fn** 本体のリスト構造の変化を示す。

通常の Lisp システムでは，マクロ展開される式は一旦展開されるともとの式が失われ，プログラム・テキストとは形が変わってしまう。しかし，TAO では展開後の式のみならず展開前の式もインビジブル・ポインタを用いて保持しているので，デバッグ上も有利になる。

2.3.3 変数

TAO の変数は Lisp 変数と論理変数の二種類に分類される。論理変数はユニフィケーションを高速化するために新たに導入したもので，Lisp 変数とは値の代入機構が異なる。Lisp 変数はラムダ・バインディングまたは **setq** のような代入式によって値を設定されるが，論理変数への代入はユニフィケーションによって行なわれる。論理変数は，初期値として「未定義」を表わす **undef** という特殊な数を持ち，一旦ユニファイされた後，バックトラックによって **undef** に暗黙に巻き戻されたり，関数 **untrail** を用いて陽に巻き戻されたりする。一方，論理変数は Lisp と共存し，代入式によっても値を設定できる。たとえば，ユニフィケーションによって論理変数 **_x** と **_y** が，一旦 **1** と **2** になっていたとする。このとき，

表 2.4: データ型およびインビジブル・ポインタ

データ型または インビジブル・ポインタ	意味
nil	`nil` と ( ) は, 見やすさのために区別される。
shortnum	24 ビットの整数。
bignum	無限多倍長整数。
ratio	比, たとえば 2/3。
float	浮動小数点数。
complex	複素数。
id	シンボル。
keyid	キーワード・シンボル。
sysid	特殊シンボル。
logic	論理型プログラミングのための論理変数。
char	文字。
str	文字列。
fatstr	フォント情報つきの文字列。
vector	ベクタ。
applobj	関数オブジェクト。
cell	セル。
namcell	名前付きのセル *。
bra	ブラケット。メッセージ伝達式に使用。
nambra	名前つきのブラケット。
quoted	クオート。 `'foo` は, `(quote foo)` でなく, `'foo` と出力される。
backq	バッククオート・マクロ展開子。
eval	バッククオート中のコンマまたは, ユニフィケーション前の評価。
icar	セルの `car` 部へのインビジブル・ポインタ (セルの `cdr` は見えない)。
icdr	セルの `cdr` 部へのインビジブル・ポインタ (セルの `car` は見えない)。
splvar	スペシャル変数またはクローズド変数。
evalvar	前処理された変数。一種の **icar**。
evallogic	前処理された論理変数。一種の **icar**。
evalinst	前処理されたインスタンス変数。一種の **icar**。
evalcdr	マクロ展開された式。一種の **icar**。
shadow	前処理された `let`, `prog` の結果の式。一種の **icar**。
comment	コメント。インビジブル・ポインタとして格納される。一種の **icdr**。

* `table(i j k)` の形で入出力されるが, 内部構造は `(table i j k)` と同じリスト。

(setq _x (+ _y)) または (!_x (+ _y 2))

によって, _x の値は 3 になり, また,

(untrail _x _y)

によって, _x と _y の値は **undef** に戻る。これを用いると論理型の繰り返しプログラムが簡単に書ける。これは純粋な論理型パラダイムの観点からは適切でない面もあるが, Lisp 等と組み合わせたマルチパラダイムの観点から見ると便利な機能である。なお, オブジェクト指向で用いるインスタンス変数は, Lisp 変数の一種である。

TAO ではマルチプロセスとマルチユーザ環境をサポートしているので, 変数は通常の Lisp 言語とは多少異なる。マルチの環境では, 様々な相互依存関係を持つ複数のプロセスやユーザが, 同一のプログラムを共有したり変数シンボルを共有したりする。このため, 浅いバインディングではコンテクスト切り替えの処理が重くなって実用的でないと考えられる。また, インビジブル・ポインタを用いて局所変数を前処理する技法を用いれば, 深いバインディングでも浅いバインディングなみの速度が得られる。こうした理由から, TAO のインタプリタでは, 変数のバインディング方式として深いバインディングを採用した。

Lisp 変数には, **局所変数**, **スペシャル変数**, **セミグローバル変数**, **グローバル変数**があり, スコープもこの順で広くなる。一つのプロセスに対し, 単一のスタック上の活性化フレームの連鎖で評価の環境を表現する。個々のフレームには, スコープの切れ目であるかスコープ透過であるか, などの属性が示される。変数がスタックのトップからボトム方向に探索されるとき, 最も内側のスコープ境界となるフレームで探索のモードが変わる。この境界のフレームに達するまでは, すべての変数は無条件にアクセス可能である。この範囲で見つかる変数が局所変数である。この境界を越えると, 式

(special-variables *var* ...)

によってスペシャルであると宣言されている変数のみが探索される。

最内側のスコープ境界フレームを過ぎる際, それ以降のフレームを辿る前に, もしそのフレームがメソッドのフレームである場合にはユーザ定義オブジェクト[18]の中のインスタンス変数が探索される。スタックのボトム付近では, プロセス間で共有される interprocess closure という関数クロージャ[19]が存在している可能性がある。

スタックの中に変数が見つからなければ, セミグローバル変数が探索される。セミグローバル変数はプロセス毎に一つのベクタとしてまとめられ, プロセス・オブジェクトに保持されている。この変数ベクタは変数名のアドレスで整列され, バイナリ・サーチで探索される。セミグローバル変数は, プロセスがリセットされてもそのまま保持されるので, プロセス毎に「局所的」なグローバル変数として使用できる。

セミグローバルでも見つからなければ, その変数シンボルのグローバル変数としての値フィールドが調べられ, もしそれが値を持っていればその値が返される。

スコープを区切る関数であるかどうかは, 関数の種類で区別する。それぞれの関数を定義するマクロが用意される。また, TAO ではクロージャを作ると, セル領域に **splvar** というタグを持つ

(*value* . *var*)

という形のデータがバリュー・セルとして作られる。これは通常の TAO のデータであるので, ス

[18]User Defined Object: 略して **udo** という。第3章で詳しく述べる。

[19]Function Closure: 関数と実行環境を併せ持つもの。

ペシャル変数を関数クロージャの中に閉じ込めることもごく自然である。Common Lisp ではスペシャル変数をクロージャに閉じ込めることを許していないが，並行プロセス間で変数を共有して通信を行なうためには，スペシャル変数をクローズする機能は重要である。上記の interprocess closure がないと，特定の並行プロセス間で変数を共有してプロセス間通信を行なうことが困難になり，その場合，変数はすべてのプロセスに見えてしまうか，個々のプロセスに完全に局所的になるかのいずれかになってしまう。

2.3.4 代入

代入の機構は，TAO の特徴の一つである。TAO では，変数のほか，リストの **car** や **cdr**，配列の要素等，場所を意識することのできるデータをアクセスしたときには rpr, rpf と呼ばれる内部レジスタ[20]に，その場所の情報を設定する。rpr はその場所のメモリ上のアドレスを，rpf はその場所の性質 (**car**, **cdr** 等の情報) を表わす。rpr, rpf が設定されたデータに対しては，TAO における一般的な代入である

(! *assignee new-value*)

が可能である。この式は，**setf** のようにマクロで展開されるのではなく，インタプリタに組み込まれた機構によって実現されているため，マクロ展開では実現が困難な代入にもうまく機能する。たとえば，

```
(!(cond (flag (car x)) (t (cadr x))) new-value)
```

といった代入も可能である。

また，**自己代入**という代入機構もある。これは，たとえば，

(!!*fn* ...(*gn* ...!*assignee* ...) ...)

という形をしている。この式は，*assignee* で設定された rpr と rpf が副作用等で更新されない限り，

(!*assignee* (*fn* ...(*gn* ...*assignee* ...) ...))

と等価であるが，この式よりも効率がよい。後者は *assignee* を二度評価するが，自己代入式では一度しか評価しない。*assignee* の評価が配列のインデックスの計算等を伴う場合等には大きな差が出る。プログラミング経験によると，この自己代入式は，プログラムの読みやすさと効率の両面で，極めて有効であった。

2.3.5 並行プログラミング

TAO では，マルチプロセス / マルチユーザ環境を実現する並行プログラミングをサポートしている。プロセスの状態はクラス **process** のインスタンスを用いて管理し，128 個までのプロセスが存在しうる[21]。各プロセスは，スタック内の連続するスタック・ブロックを割り当てられる。環境スタックは仮想化され，プロセスが走っていないときには主メモリにスワップアウトされうるが，プロセスの走行中は全体がスタック上に存在している。並行プログラミングの核の部分は完全にマ

[20] それぞれ，replace register, replace flag を意味する。

[21] プロセスの状態はオブジェクトに保持されているが，プロセスの実行自体がオブジェクト指向の機構で走行しているわけではない。本論文の第4章では，並行プログラミングとオブジェクト指向を組み合わせたシステムについて述べている。

イクロコード化されており, 実行時の性能は高い。以下では, 並行プログラミング実現上の要点のいくつかを列挙する。

(1) 2.3.3で述べたように, 変数のセマンティクスを変更している。複数のプロセスでプログラムを共有するために深いバインディングを採用し, interprocess closure を用いてスペシャル変数を共有することにより, プロセス間通信を局所化している。

(2) マルチプロセスではプロセス切り替えのオーバヘッドの削減は重要なポイントである。TAO では, プロセス切り替えの高速化のため, プロセスの環境を可能な限り局所化している。たとえば, 論理型プログラミングの機能は単一のスタックで実現されている。

(3) 並行プログラミングのために必要十分なプリミティブ関数を用意している。プロセス同期および相互排除のための任意長のキューを持つセマフォとメールボックスがあり, それぞれ **udo** で実現されている。また, 割り込み処理関数を引数とする `process-interrupt` という関数を, 割り込まれる側のプロセスのもとで実行するプロセス割り込み機能を備える。

(4) インタプリタやコンパイルド・コード・インタプリタの適切な時点で割り込みを検出している。割り込み検出はマイクロプログラムの多重分岐を用いて高速化し, 時間的な損失はない。

(5) プロセス間で `return-from` や `go` を用いた制御の受け渡しをどう扱うかは問題であるが, こうした基本的な制御関数に同期の問題を持ち込むことは得策ではないため, TAO ではこれを禁止している。

(6) プロセス間では, グローバル変数への代入以外にも, `putprop` や `defun` 等, 大域的な効果を持つ危険な操作が存在する。変更を受けるシンボルの **symbol-package** と, 現在のプロセスのパッケージの等価性をチェックすることによって, そうした危険の一部は回避できるが, 問題のすべてが解決しているわけではない。

(7) マルチユーザの環境では, シンボルのパッケージ・システムには種々の問題が生ずる。パッケージ化されたプログラムは複数のユーザや複数のプロセスで共有できる。そうしたプログラムは, 再入可能なコードとユーザ依存のデータとを分離しておく必要がある。共有可能なパッケージへの `use-package` 関数は, それぞれのユーザに特有の環境設定のために何らかの付加作業をする必要がある。また, 共有パッケージは, 誰かが使っている最中にも別の誰かが更新する可能性がある。もしユーザが `logout` したら, そのユーザに局所化されているすべてのシンボルは強制的に `unintern` され, 最終的には回収されることになる。しかし, `logout` してしまったユーザが生成したシンボルのうちの一部が, ほかのどこかから指されているときに, そうしたシンボルをどう扱ったらいいのか, といった問題は残っている。

2.4 TAO の速度性能

TAO は, ELIS のハードウェア性能と処理系実現上の種々の工夫により, インタプリタにおいて極めて高い速度性能を示している。表 2.5は, ベンチマークによる TAO の速度性能を, 商用 Lisp マ

表 2.5: ベンチマークの結果

benchmark	TAO	Symbolics	
	インタプリタ	インタプリタ [1]	コンパイラ [2]
Tarai-5	1.00	44.9	0.17
Tak-18-12-6	1.00	41.8	0.15
List-tarai-4	1.00	36.8	2.52
String-tarai-4	1.00	26.0	3.50
Bignum-tarai-4	1.00	40.8	2.48
Flonum-tarai-4	1.00	30.3	0.26
Bit-A-6	1.00	21.4	0.69
TPU-3	1.00	21.4	1.20
TPU-4	1.00	21.0	1.32
Boyer	1.00	33.8	0.28

[1] Release 5.0 without Instruction Fetch Unit

[2] Release 6.0 with Instruction Fetch Unit and scheduler off

シン Symbolics-3600[22] と比較測定した結果を CPU 時間の比で示す [35]。表から明らかなように，インタプリタ同士では TAO が圧倒的に高速であり，インタプリタを高速化するための様々な手法の有効性を確認できる。Symbolics のコンパイラと TAO のインタプリタの比較では，ベンチマークによってかなり結果に差があり，いずれが速いとも言えない。

TAO は広範な機能を持っているので，このような単体の速度は性能のごく一部である。オブジェクト指向パラダイムに関する性能は第3章で述べる。本論文では割愛するが，パラダイム間での速度性能比の目標とした 1 ～ 2 は，いくつかの性能測定結果により，ほぼ達成されている [36]。また，本論文第 4章で述べるような実験システムではマルチプロセス / マルチユーザ環境での性能も問題になるが，数人のユーザが **login** して 20 ～ 30 個のプロセスが存在している条件でも，使用経験上十分な実用性が確かめられている。

2.5 結論

本章では，Lisp マシン ELIS と，その上の言語 TAO の設計思想およびその実現技法の概要を述べた。ELIS では，Lisp の処理を支援するハードウェア資源をマイクロプログラムで直接操作することによって高性能の処理系を実現することを狙った。そのため，マイクロプログラマから見えるアーキテクチャを単純化し，リスト処理のための基本的な操作をハードウェア化するとともに高機能のマイクロプログラミング支援系を作成して，大量のマイクロプログラムの開発をサポートした。ELIS の主な特徴は，タグをハードウェアでサポートしたこと，大容量のハードウェア・スタックと WCS を備えたこと，およびリスト操作を支援する強力なメモリ・インタフェースを実現していること，などである。

[22]主記憶 8MB。なお，Symbolics-3600 は，Instruction Fetch Unit(IFU) がないと，30 ～ 40% 程度遅くなる。

マイクロプログラミング支援系では，リンカによる WCS アドレスの自動割り付け機能が特に有効であった。シーケンシングとアドレス割り付けとの関係から，論理的な処理の流れとしては一見正しそうでも，アドレス割り付け上は矛盾があって WCS 上に物理的に配置することが不可能である，といった事態は多々生ずる。マイクロ命令の割り付けとともにこうした誤りの検出を自動化したことによって，TAO の大規模なマイクロプログラミングが可能になった。

TAO の言語設計における基本思想は，ユーザが変化するにつれて，言語もまた変化すべきであり，ユーザの多様な要求に言語もまた対応すべきである，というものであった。複合プログラミング・パラダイムは，TAO で与えたひとつの解答である。しかし，ここで与えた TAO は，最終的な言語仕様を示すものではない。実際，TAO の設計と実現は同時並行的に行ない，開発の途中で実現法に発見があると言語仕様が大幅に変わる，といったことが頻繁に生じた。新しい言語機能が必要になれば，それが既存の機能に適合しないものでない限り，実装上の妥当性と有用性を考慮しつつ採り込んできた。この意味で，TAO は今後も進化を続ける言語である。

TAO では実用的な多くのシステム・プログラムやアプリケーションが開発されているが，その多くは TAO のオブジェクト指向パラダイムを用いている。オブジェクト指向は，ある種の大規模なプログラムを明解に，そして扱いやすくするのに不可欠である。TAO はプログラミング・パラダイムの多様性を一つの言語の中に提供した実験的な言語であるが，現在のところ論理型パラダイムの使用は必ずしも十分でなく，他のパラダイムとの結合で使われているものは少ない。しかし，プログラミングにおける自由度の大きさは，複合プログラミング・パラダイムの有効性を裏付けている。ユニフィケーションも，強力なパタン・マッチャとして用いられている。複数のプログラミング・パラダイムを融合する問題は今後も研究課題であり，言語にどのようなパラダイムをどのような実現法で融合していくかは，現在の TAO でもなお結論を下しえない問題であるが，TAO はこうした研究のための基盤を与えている。

TAO のシステム・プログラムやアプリケーションは複数のユーザ環境 (典型的には 3 〜 5 ユーザ) で走っているが，速度性能に対するユーザからの不満は殆どない。当初 DECsystem-2060 上の MacLisp で実装された大規模な知識表現システムが ZetaLisp に移植され，その後 TAO インタプリタに移植されたが，その速度は，それらのシステムより速いか，少なくとも肩を並べる速度である。この理由は，システム定義関数の半数，とりわけ殆どすべての入出力関数と文字列操作関数がマイクロコードで走っていることによると考えられる。インタプリタにおいて，パラダイム間の性能比は当初目標とした許容範囲にほぼ収まっており，この意味でも TAO の速度性能上の目標はほぼ達成されている。

第 3 章

TAO におけるオブジェクト指向とその実現

3.1 緒論

プログラムの開発においては，計算機のアーキテクチャやプログラミング言語など，システムの提供する記述手段に合わせて扱う対象をモデル化し，そのモデルの上で様々の操作を行なうことによって所望の処理を進める。取り扱う問題をモデル化する有力な手法の一つとしてオブジェクト指向の概念が提唱され，Smalltalk-80 [13] や Flavors [15] [16] 等のシステムによって具体化されてきた。

オブジェクト指向によるモデル化の特徴は，対象とする問題に内在する論理的実体に自律性を意識する点にある。オブジェクト指向では，データとそれに対する演算とを一つのまとまりとして捉え，それが自律的な振舞いをする実体 (これを**オブジェクト**と呼ぶ) であると考える。たとえば，整数は 1 とか 2 とかのデータであると同時に，加算や減算などの演算を持つオブジェクトである。オブジェクトの持つ演算を起動するには，そのオブジェクトに**メッセージ**を送る。オブジェクト指向での計算は，一群のオブジェクトがメッセージをやりとりする過程で遂行される。

オブジェクト指向の第一の利点は，対象を計算機上にマッピングする際に利用できるモデルとしての自然さとそのモジュール性にある。オブジェクトは，それ自体の実現法とは切り離した，メッセージというプロトコルのみによる外部インタフェースを与える。このような情報隠蔽によってプログラミング上考慮すべき範囲を制限し，モジュール性を保つことによって，プログラマの労力を軽減することができる。また，類似のオブジェクトをクラス分けし，共通部分を一元管理する継承等のプログラミング機能により，プログラムの再利用性や保守性が向上する。

本章では，TAO における複合プログラミング・パラダイムの一つの柱としてオブジェクト指向を取り入れ，Lisp を始めとする他のプログラミング・パラダイムとの融合を図った研究について述べる [37] [38] [39] [40] [41]。AI プログラミング環境において重要性の大きいオブジェクト指向をいかにうまく取り入れるかは，TAO の設計において極めて重要なポイントであった。

TAO では，ZetaLisp のオブジェクト指向パッケージである Flavors を手本としながら独自の機能を加え，単なるソフトウェア・パッケージとしてでなく処理系の基本機構としてオブジェクト指向を組み込んだ。本研究の意義は，Lisp の言語仕様を拡張する形で，かつその性能を損わずにオブジェクト指向を融合し，実現法を工夫して大規模な応用に耐えるシステムとして提供した点にある。さらに，ベンチマークや実際の使用経験を通じて，その実現手法と性能を評価した。

オブジェクト指向の実現にあたっては三つの問題点があった。まず第一に，ユーザに直接見える言語表層で，いかに自然に Lisp に融合された機能としてオブジェクト指向を提供するかという問

題である。第二に，手続きが先に与えられるのではなく，データと受信メッセージから手続きを決定するという，手続き型のパラダイムとは根本的に異なる計算機構を，既存のLispに遜色のない性能で，かつLispの性能を落とさずに融合する必要がある。第三に，クラス構造などの多くの内部情報管理を効率よく処理し，会話型のプログラミング環境として十分なターン・アラウンドを確保できなければならない。

TAOでは，Lispの式としては不正として排除されていたcar部が関数でない形のS式をメッセージ伝達式として採用し，Lispプログラムの中に自由に混在できるようにした。この式はSmalltalkスタイルのメッセージ伝達式に近く，算術演算などの中置記法[1]を可能にしてプログラムの可読性を改善した。Lispの基本データはオブジェクト指向の意味でのオブジェクトである。また，ユーザの定義するオブジェクトを表わすデータ構造もLispのデータそのものである。しかし関数呼び出しとメッセージ伝達式はプログラマの意識の上で明確に区別できる。プログラミング上も違和感は殆どなく，言語の表層でのオブジェクト指向の融合は自然である。

オブジェクト指向の速度性能には，メッセージ伝達の解釈機構，メソッドの探索と起動，そしてメソッド本体の実行という三つの要因がある。TAOでは，Lispの式の解釈機構の中にマイクロプログラムによるメッセージ伝達式の解釈機構を埋め込んだ。インタプリタはLispプログラムの中に混在するメッセージ伝達式を識別してメッセージの評価機構を起動する。受信オブジェクトとメッセージの情報からメソッドを探索するために，TAOではマイクロプログラムによる高速のバイナリ・サーチを採用した。さらに，メソッド本体の実行にかかわる変数のアクセス機構をハッシュ表等を用いて高速化し，基準となるLisp部と比べても遜色のない高速処理を実現した。

TAOでは速度性能を重視し，性能にかかわる部分では動的にクラス階層を辿る処理などを極力避ける実現法をとった。このため，大規模な応用プログラムでは，継承によるメソッドの重複やメソッド表の肥大化等によってメモリの使用効率が悪化したり，種々の内部データを生成するための時間が無視できなくなる等の問題が生ずる。そこで，メソッドの内部表現の共有化を図ったり，内部的な情報を徹底したオン・ディマンドによって構築する方式を採用し，真に必要な情報のみが必要な時点でしか作られないようにした。これによって，プログラム開発時のターン・アラウンドが大きく短縮され，会話性を確保することができた。また，メモリの消費も大幅に削減できた。

このように本システムでは，Lispマシンのハードウェア性能を背景に，大規模な応用プログラムの開発にも耐えうる性能を持つオブジェクト指向をLispに融合し，複合プログラミング・パラダイムの実現性と実用性を明らかにした。本章ではこれらの詳細を論ずるとともに，ベンチマークや実際の応用プログラムによる総合的な評価を行ない，あわせてオブジェクト指向における問題点も指摘する。

3.2 TAOにおけるオブジェクト指向

3.2.1 オブジェクト

TAOのオブジェクトには，整数や文字列などのような基本データのオブジェクトと，ユーザ定義オブジェクト**udo**[2]の2種類がある。**udo**は内部状態を表わす一群の変数からなる構造を持つ。基本データは，変数としての内部状態は持たない。たとえば，整数オブジェクトは変数としての内

[1] infix notation

[2] User Defined Object

部状態を持たないオブジェクトである。オブジェクトには，一群の名前付きの手続きが付随する。この手続きのことを**メソッド**と呼ぶ。TAO オブジェクトのメソッドは，Lisp の関数と同様に値を返す。後述するように，構造や動作が共通なオブジェクトの集合は，クラスという概念でまとめられる。各オブジェクトはいずれかのクラスに属し，そのクラスの**インスタンス**と呼ばれる。**udo** の内部状態変数を**インスタンス変数**と呼ぶ。

3.2.2 メッセージ

3.2.2.1 メッセージ伝達式

オブジェクト指向モデルでは，オブジェクトが互いにメッセージをやりとりしながら相互作用することによって計算が進行する。オブジェクトはメッセージを受け取ることができ，また，自分自身のメソッドを実行する過程で他のオブジェクトにメッセージを送ることができる。メッセージの送信は，**メッセージ伝達式**と呼ばれる専用の式を評価することによって行なう。TAO のメッセージ伝達式は，一般に次の形をしている。

[*receiver-object message-name argument* ...]

ここで，*receiver-object* は，このメッセージを受信すべきオブジェクト，*message-name* はこのメッセージの名前である。メッセージを受信すると，受け手オブジェクトは *message-name* で指定されたメソッドを自分のメソッドのレパートリから選択して起動する。この意味で，メッセージ名のことを**セレクタ**とも呼ぶ。メソッドには引数を必要とするものもある。*argument* ... は，このようなメソッドのための引数である。これをメッセージの引数ともいう。オブジェクトが外部に対して提供するインタフェースはメッセージのレパートリだけであり，インスタンス変数などの内部状態は，直接にはオブジェクトの外部からは見えない。

3.2.2.2 メッセージ伝達式の評価

メッセージ伝達式は，[ ] を用いた S 式である。[ ] はデータとしては Lisp の通常のリストと全く同じであるが，**bra** というタグが立っており，`eval` がこれをメッセージ伝達式として解釈するという点だけが異なる。`eval` は，この式を次のように評価する。まず `car` 部の *receiver-object* を評価する。その値がオブジェクトならば `cadr` 部の *message-name* をセレクタとみなし，対応するメソッドを起動する。セレクタは評価しない。*argument* 以下はそのメソッドに渡す。メッセージ伝達式は Lisp プログラム中に自由に書かれ，メソッドの値がそのままメッセージ伝達式の値となる。

メッセージ伝達式は，[ ] でなく ( ) を用いて書くことも許される。実行時には `car` が関数でないことからメッセージ伝達式であることがわかるからである[3]。ただし，この場合，字面だけからはそれがメッセージ伝達式であるかどうか決められず，コンパイルしたときに効率的なコードが生成できなくなる。

セレクタは通常は評価されないが，これを評価したい場合もある。このようなときは，メッセージ伝達式においてセレクタにカンマをつける。すなわち，

[*receiver-object ,message-name argument* ...]

[3] メッセージ伝達式にもなりえない場合にはエラーになるが，この場合も「未定義関数」のエラーではない。

とすると，セレクタが評価される。また，Flavors と同様，関数 **send** によるメッセージ伝達が可能である。すなわち，

(**send** *receiver-object message-name argument* ...)

または **apply** に対応するものとして，

(**apply-send** *receiver-object selector-argment-list*)

という形式がある。

3.2.2.3 中置記法

メッセージ伝達式のシンタックスは，関数名が式の先頭に来る Lisp の前置記法とは異なり，手続き名に対応するセレクタが二番目に来る Smalltalk-80 方式である。**car** が関数でない式は従来の Lisp ではエラーとして排除されていた。TAO では，従来の正当な Lisp の式に何ら変更を加えずにこの式にメッセージ伝達としての意味を与え，Lisp のプログラムにオブジェクト指向を導入した。またこの形は，整数の加算や乗算を，**[x + y]** や **[x * y]** といった中置記法で書くことを可能にする。TAO のこの式は，マクロなどのプログラム変換によって Lisp の式に置き換えられるのではなく，直接にシステムのマイクロプログラムで解釈実行されるという点が重要である。さらに，整数の基本演算については **[x + y + z + u]** のような記法もマイクロプログラムで直接解釈実行される。この場合は，最初の **[x + y]** を一つのメッセージ伝達式として実行し，その結果にさらに **+ z** が送られる，といった解釈になる。従って，上記の式は，**[[[x + y] + z] + u]** と等価であるが，入れ子構造がないのでインタプリタでの解釈速度は速い。

3.2.3 クラス

同じインスタンス変数のセットを持ち，同じメッセージに反応できる同一の種類のオブジェクトの集合を**クラス**と呼ぶ。Flavors 等と異なり，TAO では整数や文字列といった基本データ型もクラスである。あるクラスに属する個々のオブジェクトは，そのクラスのインスタンスである。インスタンス変数やその初期化の方法，あるいはメソッドの表など，インスタンスが共有する情報はクラスに属する。TAO では，クラス自体はオブジェクトではなく，Smalltalk-80 にあるようなメタクラスの概念はない。しかし，Smalltalk-80 にあるような**クラス変数**がある。

3.2.3.1 クラスの定義

TAO のオブジェクト指向によるプログラミングは，クラスを定義することから始まる。クラスの定義によって，インスタンス変数のセットなどオブジェクトのデータ型としての構造が決定する。**udo** のクラスは **defclass** マクロで定義される。**defclass** は，

(**defclass** *class-name*
  class-variable-list
  instance-variable-list ———————————— [1]
  {*superclass-list*}
  defclass-option ...)

という形のマクロである[4]。*class-name* は，クラス名を表わすシンボルである。*class-variable-list* と *instance-variable-list* は，それぞれクラス変数，インスタンス変数のリストである。*superclass-list* は，このクラスのスーパークラス名のリストである。スーパークラスについては後述する。最後の *defclass-option* は，このクラスのいろいろなオプション機能を指定する。

3.2.3.2 クラスの階層と継承

整数というクラスは，より一般的な数である有理数というクラスの部分集合であり，逆に，有理数というクラスを特殊化したものと考えられる。このように，数のクラスは抽象化の程度による階層をなしている。一方，整数クラスの持つ性質のうちのいくつかは，有理数クラスの性質から受け継いだものという見方もできる。つまり，整数は，有理数の性質を**部品**として持っていると考えることができる。このように，オブジェクトの階層は抽象化の程度による階層であるという見方と，構成部品と全体という関係による階層という見方の，両方が可能である。

既存のクラスをもとにして，必要な追加・変更分のみを加え，新たなクラスを定義することができる。新たに定義するクラスは，もとのクラスのインスタンス変数やメソッドなどを受け継ぐ。これを**継承**という。新たなクラスは，既存のクラスとの差分のみで定義でき，オブジェクトの共通の属性を一元的に管理することによって，モジュール性の高いプログラミングが可能となる。

TAO では，もとになるクラスを**スーパークラス**，あらたに定義されたクラスを**サブクラス**と呼ぶ。スーパークラスは複数個あってもよい。これを**多重継承**と呼ぶ。TAO の継承に関する用語は Smalltalk-80 にならっているが，機能は Flavors のそれに近い。すべてのクラスは，デフォールトでは **vanilla-class** というクラスを継承する。**vanilla-class** には，あらゆるクラスに共通なメソッドが定義されている。

3.2.3.3 クラスの順序づけ

スーパークラスからの情報を継承するとき，情報間で競合が起こることがある。そこで，クラス間に何らかの順序を与えたり，その中から選択したりすることが必要になる。TAO では，クラスの階層を一定のルールで辿ることによって順序づける。この順序は，起点となるクラスからスーパークラスに向かって，深さ優先で，左から右[5]の順である。たとえば，三つのクラス **a**, **b**, **c** が次のように定義されているとする。

```
(defclass a () (x) (b c))
(defclass b () (y) ())
(defclass c () (y) ())
```

この例では，クラス **a** が最もサブクラス側にあり，ここから見たクラスの順序は，

a → **b** → **c** → **vanilla-class**

となる。**vanilla-class** はクラス **b** のスーパークラスでもあるので本来 **b** の次に来るべきであるが，**vanilla-class** に限ってすべてのクラスの最後に辿る。なお，スーパークラスのネットワークは

[4]{ } でくくられた引数は随意引数 (省略できる引数) であることを，... は，その前に書かれた引数が剰余引数 (最後に並んだ引数をまとめてリストで受け取る引数) であることを，それぞれ表わす。それ以外の引数は義務引数 (必ず与えなければならない引数) である。

[5]`defclass` で指定されたスーパークラスのリスト上での左から右。

木構造に限るわけではないので，単純な深さ優先，左から右という辿りでは同一のクラスに達することもある。この場合，一度辿ったクラスは辿らないようにして重複を避ける。

3.2.4 変数

メソッドはLispの関数に近く，その中では局所変数やスペシャル変数等あらゆるLisp変数を使用できる。オブジェクトに特有の変数にはインスタンス変数とクラス変数がある。インスタンス変数はメソッド内部で自由に参照され，メソッドの実行環境を設定する。クラス変数はクラス自身のデータ・スロットである。ただし，この変数は通常の意味での変数ではなく，専用の関数 **cvar** を用いて参照する。 **cvar** はメソッドの中でのみ有効である。インスタンス変数と異なり，クラス変数は継承されないが，そのクラスとそのサブクラスのメソッドの中からは参照できる。自分のクラスのものでないクラス変数を参照したときは，上位のクラスへクラス階層を辿って探索する。

インスタンス変数をオブジェクト外部から参照できるようにするためには，それらの変数を参照するメソッドを陽に定義する必要がある。 **defclass** マクロのオプション **:gettable** は，これを自動的に行なう。そのセレクタは，インスタンス変数名である。たとえば，クラス **A** のインスタンス変数 **x** が **:gettable** であると宣言されると，**A** のインスタンス **i** に対して，**[i x]** というメッセージ伝達が可能となり，値としてインスタンス変数 **x** の値が返される。

3.2.5 インスタンスの生成

クラス *class-name* に属するインスタンスは **make-instance** というマクロを用いて生成される。**make-instance** は，次のような形式で呼ばれる。

(**make-instance** *class-name init-option* ...)

ここで，*init-option* は，インスタンス変数に初期値を与えるための情報で，インスタンス変数名と初期値の交代リストである。インスタンスは **defclass** 時に宣言された情報を参照して作られる。ただし，**integer** や **vector** など，**udo** 以外のオブジェクトは **read** 関数や専用の生成関数によって作られ，**make-instance** では生成されない。

3.2.6 メソッド

3.2.6.1 メソッドの定義

メソッドの定義は，**defmethod** マクロを用いて行なう。メソッドはクラスに登録され，インスタンスへのメッセージ伝達が起こったときに起動される。 **defmethod** は次のような形式を持っている。

(**defmethod** (*class-name* {*method-type*} *selector*)
  argument-list ———————- [2]
  form ...)

ここで，*class-name* は，このメソッドの属するクラス名である。{*method-type*} は，メソッド結合のときに参照される情報で，省略してもよい。メソッド結合については後述する。 *selector* は，このメソッドのセレクタである。 *argument-list* は引数のリストで，関数定義のラムダ・リストと同

様の役割を持つ。引数リストの先頭には自動的に **self** という義務変数が付加され，本体の実行に入る前に受け手オブジェクト自身に束縛される。*form* ... はメソッドの手続き本体で，Lisp 関数の本体とほぼ同様であり，局所変数のスコープはレキシカルに閉じているが，インスタンス変数は見掛け上自由変数として参照できる。

3.2.6.2 メソッド結合

スーパークラスのメソッドがサブクラス側に継承されるとき，同じ名前を持つメソッドが複数個あると，そのどれを用いるかを明確に指定する必要がある。TAO では，このような場合のメソッドの用い方を**メソッド結合**と呼ぶ方式で組み合わせ，それを新たな手続きとして用いる。**defmethod** で定義されたメソッドはいわば手続きの部品であり，メソッド結合によって合成された手続きがメッセージ伝達のときに実際に起動されるメソッドになる。メソッド結合の方法にはいろいろあるが，以下ではデフォールトで用いる **:daemon** というメソッド結合について述べる。

メソッドは，**defmethod** マクロで指定されたメソッドの型 (*method-type*) を持つ。*method-type* の指定が省略されると **:primary** という型が指定されたものとみなされる。メソッド結合ごとにその結合に参加するメソッドの型は定められている。たとえば **:daemon** 結合に参加するのは **:primary**, **:after**, **:before** の三つである。**:daemon** 結合では，クラス階層を辿って順序づけられた各クラスのメソッドは，その型に応じて次のように起動される。

1. クラス階層をサブクラス側からスーパークラス側への順に辿り，それぞれのクラスの **:before** メソッドを順次実行する。

2. クラス辿りにおいて最初に見つかる **:primary** メソッドを実行する。

3. **:after** メソッドを，**:before** とは逆の順に実行する。

4. **:primary** メソッドの返す値が，結合されたメソッドそのものの返す値になる。

:after や **:before** が存在しないときは Smalltalk-80 の普通の継承と同様になり，サブクラス側のメソッドが有効で，スーパークラス側のメソッドは無効になる[6]。

3.2.7 スーパー・メッセージ伝達

あるメソッドの中から，自分のスーパークラスにある特定のメソッドを起動することができる。たとえば，クラス **a** のメソッドにおいて，**a** のスーパークラス S で定義されているメソッド n を起動したいとする。そのためには **super** という関数を使って，

(**super** *S.n argument* ...) ——————————— [3]

という式を用いる。*S.n* は文字どおりクラス名の S とメソッドのセレクタ n とを "." を挟んで連接して作ったシンボルで，評価されない。*argument* ... は，メソッド n に渡される引数である。**super** の動作は，あたかも **self** が，ある瞬間だけそのスーパークラスのインスタンスになりかわったかのような動作になる。メソッド結合も，そのスーパークラス S を起点として行なう。

[6]Smalltalk-80 で言う override。

3.2.8 no-method-found ハンドラ

送られたメッセージのセレクタが, 受信オブジェクトの内部的なメソッド表に登録されていない場合, **no-method-found** というエラーになる。Smalltalk-80 では, 見つからなかったメッセージを引数とする **doesNotUnderstand:** というメッセージを再び受信者に送る。もしその受信者がメソッド **doesNotUnderstand:** を持っていれば自分でエラー処理ができる。

TAO では, **detach/attach** 機能[7]の実現のために, **no-method-found** ハンドラを導入した。これは基本的に **doesNotUnderstand:** と同じである。**no-method-found** ハンドラは,

(defmethod (*class*) (sel &rest args) *form* ...**)**

のように, 名無しのメソッドとして定義される。引数は, 見つからなかったセレクタと, もとのメッセージの引数のリストである。**no-method-found** ハンドラが定義されているクラスのオブジェクトが, 自分のレパートリにないセレクタを持つメッセージを受け取ると, この **no-method-found** ハンドラが起動される。

3.2.9 リスト・メッセージ

defmethod マクロの式 [2] において, *selector* としてリストを与えることができる。これをリスト・メッセージと呼ぶ。リスト・メッセージは, TAO の論理型プログラミングのための関数の一種である **Hclauses**[8] として, クラスに局所化して登録された述語となる。たとえば,

```
(defmethod (c a) b1 b2 ...bn)
```

は, クラス **c** において宣言された述語で, 意味的に Prolog の

```
A :- B1, B2, ..., Bn
```

と同じである。この述語はクラス **c** のオブジェクトにメッセージを送ることによって起動できる。たとえば,

```
(defmethod (man (How old are you?))  [self age])
```

という定義が行なわれたとする。このとき, クラス **man** のオブジェクト **Einstein** に対し,

```
(== _answer ,[Einstein (How _question are you?)])
```

という式を評価すると, まずヘッダ部のユニフィケーションにおいて論理変数 **_question** が **old** にインスタンシエートされて成功し, 続いて本体 **[self age]** が実行される。ここでオブジェクト **Einstein** がメッセージ **age** を受け取って, たとえば値 76 を返すとすると, それが **_answer** にインスタンシエートされ, 述語 **==** が **t**(ユニフィケーション全体の成功) を返して式の評価が終わる。

なお, リスト・メッセージは, 継承の概念をどのように取り込むかという問題などいくつかの問題点を含んでいたので, セレクタとして **id** を用いるオブジェクト指向論理型プログラミングも別に実現された [32]。

---

[7]ユーザ端末と **login** セッションとを自由に接続したり切り離したりする機能。

[8]Horn 節に対応する。

class	*12*
symbol message vector	version number
list message vector	symbol backpointer
super symbol message vector	super list message vector
class variables	property list
defclass skeleton	make-instance skeleton
(not used)	instance variable hash

図 3.1: クラスベクタ

3.3 オブジェクト指向の実現

3.3.1 クラスの内部表現

3.3.1.1 クラスベクタ

クラスはクラスベクタと呼ぶ **vector**[9]を用いて表現した (図 3.1)。クラスベクタは `class` というタイトルを持つサイズ 12 の **vector** で, クラスの情報の殆どを保持し[10], クラス名をあらわすシンボルの持つ `class` という属性 (property) として登録される。ただし, この **vector** はそれ自身属性リストを持つなど, 通常の **vector** とは多少異なる点がある。なお, **integer** や **id** などの基本データ・オブジェクトのクラスベクタは, 処理の高速化のため, 対応するクラス名シンボルの属性リストからだけでなく, スタックの固定領域からも指されている。

3.3.1.2 defclass の動作

`defclass` マクロを実行するとクラスベクタが作られる。すでに存在しているクラスを再定義すると, 古いクラスとそのサブクラスのクラスベクタはクラス名シンボルの `class` 属性から除去される。そして, クラスの世代番号が 1 だけ増やされた新たなクラスベクタがクラス名シンボルの `class` 属性につながれる。さらに, 後述する `component-of-what` 属性を用いて古いクラスベクタのサブクラスを辿り, 継承されている情報を無効にすると同時にそれらに対してもクラスの再定義と同様の処理を行なう。

この操作によって, 古いクラス階層はその階層構造を保ったまま残る。古いクラスに属するインスタンスはもとのクラスベクタへのポインタを持っているが, 古いクラスに対して `make-instance` を行なうことはできない。新たに `make-instance` したものは新しいクラスに属するインスタンスとして生成される。このように, 古いクラス階層は, スーパークラスの再定義によって皮がはがれるように剝離し, 相互に別のクラス階層となって, クラスの再定義に伴って起こりうるインスタンスの構造上の矛盾を防いでいる。

---

[9] **vector** はタイトル (`vtitle` と呼ぶ) とサイズ (`vsize` と呼ぶ) をヘッダとする Lisp データの一次元配列である。

[10] 付録 A にクラスベクタの各エントリの説明をつけた。

3.3.1.3 component-of-what

各クラス名シンボルは `component-of-what` という属性を持ち，自分がどのクラスのスーパークラスであるかを記録している。すなわち，スーパークラスからサブクラスへ向かう下向き階層構造のリンクは名前を経由する。ただしこのリンクは，マルチユーザを持つTAOの環境ではパッケージについて多少の問題点を含んでいるため，パッケージにまたがる継承関係が生じたときはリンクを作らないことにした[11]。このため，もしシステムのクラスに何らかの変更が加えられても，ユーザ側にはそれが伝わらないという制限が生ずる。しかし，もとよりFlavors流のオブジェクト指向は実行時のクラス変更にあまり馴染まないので，そのようなプログラムに対処する必要は小さい。

3.3.2 オブジェクトとしての基本データ

表3.1に基本データ・オブジェクトのクラスの一覧を示す。これらのオブジェクトはインスタンス変数を持たず，`make-instance` でインスタンシエートすることはできない。クラスの属性としては `abstract-class`[12]になっている。また，これらのクラスは **bas** のパッケージにあって `export` されている。これらのクラスに対してメソッドを定義するときには特権[13]が必要である。

3.3.3 オブジェクトの内部表現

ユーザ定義オブジェクト (**udo**) は，図3.2に示すような特殊な **vector** である。**udo** はTAOの基本データ型の一つで，インスタンス変数とその値のスロットの組の集合からなる。**udo** はベクタそのものではないが，**vector** のデータ型を流用しているので `nthv` などの関数の引数になりうる。このため，**udo** のことを **udo** ベクタと呼ぶこともある。**udo** は `vtitle` のかわりにクラスベクタへのポインタを持つ。`make-instance` マクロは，そのクラスに対して初めてインスタンスを作るときに，まずクラス階層を辿って継承すべきスーパークラスのインスタンス変数の情報を含む内部データを作り，それをクラスベクタの **make-instance skeleton** スロットに登録する。しかるのちそれを参照しながら **udo** を生成し，そのインスタンス変数を初期化する。

udo の中でのインスタンス変数の順序は原則的にはクラスの辿りの順序に従っているが，通常ユーザから見て直接意味を持つことはない。しかし一部のマイクロコードでは，効率化のためにインスタンス変数の内部的な位置を陽に仮定しているものもある。

---

[11]あるユーザが自分のパッケージで，たとえば **A** というクラスを定義し，そのスーパークラスとしてすべてのユーザが共有するあるシステムのクラス **S** を宣言したとする。すると，**S** の `component-of-what` には，**A** のスーパークラスであることを示すリンクができる。ところが，このユーザが `logout` したとき陽にこのユーザのパッケージのシンボルである **a** を **S** の `component-of-what` から除かない限り，**S** の `component-of-what` は，もはや存在しないユーザのパッケージを押さえ続けることになる。すると，**A** に関連するプログラムやデータは解放されず，メモリを不当に圧迫する。しかし，ユーザが `logout` するたびにそのパッケージのシンボルがクラスかどうか調べて，そのスーパークラスにある (かもしれない) システムのクラスの `component-of-what` からリンクを取り除くのは重い処理である。

[12]インスタンスを生成することができないクラス。

[13]`sys:with-privilege` というマクロを用いる。

表 3.1: 基本データ・オブジェクトのクラス

クラス名	内容	TAO のデータ型
id	シンボル	**id**
codnum	コドナム	**codnum**
integer	整数	**shortnum bignum**
ratio	有理数	**ratio**
real	浮動小数点数	**shortfloat float bigfloat**
string	文字列	**char string**
vector	ベクタ	**vector**
applobj	関数類	**applobj**
cell	リスト	**cell**
dolamb	下向きラムダ	**dolamb**
locbit	locbit	配列 **locbit**
64builoc	64 ビット符号なし整数	**64builoc**
64bsiloc	64 ビット符号つき整数	**64bsiloc**
64bfloc	64 ビット浮動小数	**64bfloc**
dnil	nil データ	**dnil**

class-vector	$2n$ **(udo size)**
value 1	instance variable 1
...	...
...	...
value n	instance variable n

図 3.2: ユーザ定義オブジェクト (**udo**) の構造

3.3.4 メソッドとメソッド表

3.3.4.1 メソッド

メソッドは, 特殊なものを除いて **applobj** というデータを用いて表現される[14]。**applobj** は, 引数リストに関する情報や, スタック・フレームの生成に関する情報, 実行本体などを持つ。実行本体は, インタプリタで動く関数は S 式, コンパイルされた関数は **memblk** というデータ領域を用いて表わされたバイトコードになっている。

3.3.4.2 メソッドの定義

`defmethod` では, メソッド結合の部品となる **applobj** の原形が作られる。メソッド結合の結果できる手続きも **applobj** であって, クラスベクタの中の **id-message-vector** スロットにあるメソッド表に登録される。メソッド表はセレクタのアドレスによってソートされ, メソッドはセレクタをキーとしてバイナリ・サーチされる。

`defmethod` 時に, そのクラスが **id-message-vector** をまだ持っていないならば, **id-message-vector** のスロットには何も影響はなく, クラスベクタの中の **defclass-skeleton** というスロットにこのメソッドのセレクタと S 式の本体が登録されるだけである。この時点では実行可能なメソッドの **applobj** 本体は作られない。もしそのクラスにすでに **id-message-vector** が存在しているならば, そのクラスを起点としてスーパークラスの辿りを行ない, 与えられたセレクタのみをメソッド結合し, 得られた **applobj** を **id-message-vector** に登録する。次に, `defmethod` が行なわれるクラスのすべてのサブクラスに対し, もしそれが **id-message-vector** を持っていたら, その中の対応する **applobj** をメソッドが未作成であることを示すダミーの仮りデータ[15]に変える。

会話的なプログラム開発を支援するため, メソッドは任意の時点で再定義したり追加したりできるようにした。メソッド表はクラスベクタの中の単なるデータなので, それを修正するだけでそのクラスのメソッドのレパートリが変わる。すでにそのクラスで生成された **udo** も, それ以後生成される **udo** も等しく新しいメソッド表を参照してメッセージ伝達式を評価する。

3.3.4.3 メソッド表の作成

メソッド表は, そのクラスのインスタンスに初めてメッセージ伝達が起こったときにオン・ディマンドで作られる。メソッド表を探索しようとしたとき, そのクラスに **id-message-vector** が存在しないと, `construct-id-message-vector` という内部関数を起動してメソッド表を作る。しかしメソッド表の内容全部を作るのではなく, 特にメソッド **applobj** は, そのときに送られたメッセージに対応するものだけを作る。この処理は, 以下に述べる三つのステップからなる。

第一ステップ: メソッド表の枠の作成

インスタンスにメッセージが送られ, そのクラスのクラスベクタの **id-message-vector** スロットを調べる。このメッセージ伝達がそのクラスのインスタンスへの最初のメッセージ伝達ならばそのスロットは **nil** である。そこで, 関数 `construct-id-message-vector` を起動する。この関数は次のように動作する。

---

[14] **applobj** とは, 関数のように `apply` できるオブジェクト, という意味である。TAO の関数の実体は **applobj** である。

[15] **opr** というタグを持つ {opr}0 というデータを用いている。

(1) そのクラスを起点にスーパークラスを辿り，継承すべきメソッド・セレクタを集めたリストを作る。このとき，`defmethod` で定義された手続きだけでなく，`:gettable` や `:settable`，クラス定数なども同様に継承する。

(2) このリストからセレクタのアドレス[16]でソートされた表を **vector** で作る。この **vector** には，`:gettable`，`:settable` およびクラス定数に対するエントリは正しく埋め込まれるが，**applobj** を持つメソッドはまだ作られておらず，{opr}0 になっている。これは，最終的に **applobj** に置き換えられるべきであることを表わすフラグである。

(3) (2) で作った **vector** をメソッド表 (**id-message-vector**) としてクラスベクタに登録する。

第二ステップ: applobj の作成

(4) 起動すべきメソッド **applobj** は最初メソッド表に存在せず，{opr}0 になっている。そこで，この時初めてそのメソッド **applobj** だけを作る[17]。

第三ステップ: 起動

(5) 改めて当該メソッド **applobj** を起動する。

以下，各メソッドは初めて呼ばれる毎に[18]上の (4)，(5) の処理をする。なお，メソッド表や **applobj** の有無のチェックはマイクロプログラムの多重分岐によって行なわれ，まったくオーバヘッドにならない。

このように，メソッド表は，そのクラスのインスタンスに初めてメッセージ伝達が起こったときにオン・ディマンドで作られ，そのエントリであるメソッド **applobj** 自体も，それが呼ばれたときに初めて作られる。こうした徹底したオン・ディマンド方式を採用したのは，大きな応用プログラムにおいて，メソッドを一度に作る方式では表の作成に要する時間が無視できないためである。実際，初期の版ではすべての **applobj** をまとめて作っていたが，メソッドの総数が数千に及ぶ場合，待ち時間が分のオーダになり，デバッグの能率が悪化しただけでなく，まったく呼ばれないメソッドも大量に作られてメモリを浪費することが問題となった。オン・ディマンド方式の採用で，メソッド表の構築時間のみならずメモリ消費も著しく改善された。

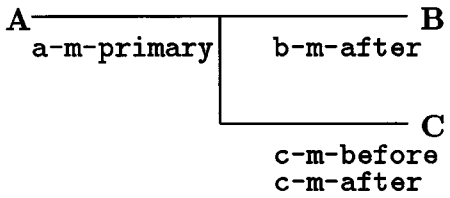
プログラムの開発時には，スーパークラス側でもメソッドの再定義や追加などが頻繁に行なわれる。その変更はサブクラス側に伝えられ，以後のオブジェクトの動作に反映される。そのときに起こるメソッド表の再構成などの処理はユーザからは見えない。また，その場合もメソッド **applobj** の生成はオン・ディマンド方式であるため，実際には殆ど何も行なわず，処理も速い。

---

[16] **id** のタグを含めた 32 ビットのデータを用いる。TAO のゴミ集めではシンボルのアドレスは変化しない。

[17] このときの処理で，クラス階層上のメソッド **applobj** を集め，メソッド結合によって作成した新たな **applobj** で，メソッド表中の対応する {opr}0 を置き換える。

[18] すなわち，見つかったメソッドが {opr}0 ならば



サブクラス側 ← → スーパクラス側

図 3.3: クラス階層とメソッド(メソッドは, その返す値で記述)

3.3.4.4 メソッド結合の実現法

メソッド結合では, スーパークラスにあるメソッドを定められた順番で起動するのではなく, それらのメソッドを静的に組み合わせたまったく別のメソッド (**applobj**) を作って, 継承する側の固有のメソッドとして登録する。継承されたメソッドは, 内部的にはコピーされて下位のクラスに取り込まれる。これにより, 実行時のクラス辿りのオーバヘッドはない。継承された **applobj** が `:daemon` メソッド結合においてどのように使われるかを例で示す。クラスが図 3.3で示したような階層を持ち, メソッドが次のように定義されているとする。

```
(defmethod (a m) () 'a-m-primary)
(defmethod (b :after m) () 'b-m-after)
(defmethod (c :before m) () 'c-m-before)
(defmethod (c :after m) () 'c-m-after)
```

クラス **A** のメソッド `m` は, `:daemon` 結合によって,

`c-m-before` → `a-m-primary` → `c-m-after` → `b-m-after`

の順に実行して `a-m-primary` の値を返すように次の形に合成され, 新たなメソッド **applobj** として **A** のメソッド表に登録される。

```
(progi
    (funcall c-m-before のメソッド applobj)
  ^ (funcall a-m-primary のメソッド applobj)
    (funcall c-m-after のメソッド applobj)
    (funcall b-m-after のメソッド applobj)
```

`progi` は `progn` の拡張版で, `progn` とほぼ同様であるが, 値として "^ " のつけられた式の結果を返す。なお, `funcall` で起動されるそれぞれの **applobj** は TAO のスコープ透過型スタック・フ

レームのもとで評価される。スコープの区切りは, 結合されたメソッドの側, すなわち progi を本体として持つ applobj の側になる。このようにすることで, 継承された applobj の中からの exit 関数による飛び出しが, 結合後のメソッドからの飛び出しになることを保証する。

このようなメソッド結合の方式は, 継承されてくる applobj の独立性を高める。メソッド結合の実現法の代替案として, ソースの S 式を文字どおり append するという方法もある。実際, 当初はその実現法によっていた。この方法によると, applobj の起動がネストせず, 実行が高速になり, スコープの問題も単純になるが, 継承する applobj 間で, 引数の名前まで一致させなければならなくなる。この方法は継承されてくるメソッドの独立性が低いためメソッドのコンパイレーションに難があり, funcall の方式に変更した。

なお, :after や:before などの applobj がないときは funcall の皮をかぶせる必要がない。このときは:primary の applobj そのものがメソッド表に登録される。

3.3.4.5 メソッド applobj の共有

継承したメソッドを自分自身のクラスのメソッド表に持つと, メソッドはおびただしく重複することになる。これらは呼ばれる環境が異なるので本質的に別のメソッドであるが, 実際には全く同じ実体であってかまわないものが多い。そこで, メソッド表へのメソッドの登録時に, 実行本体の S 式をキーとするハッシングを用い, すでに登録済みの他のメソッドと共有可能な applobj 本体はすべて共有する。この共有は S 式のメソッドのみならず, コンパイルされたものに対しても適用する。これは, 大量の applobj を作る応用システムにおいて, メモリ消費の削減に有効である。

3.3.5 変数のアクセス機構

インスタンス変数はレキシカルに宣言された変数ではないので, その参照方式はメソッドの実行速度, ひいてはオブジェクト指向で書かれたプログラムの性能に大きな影響を与える。そこで, インタプリタにおいてハッシュ表を用い, インスタンス変数を udo の先頭からのオフセットで参照するようにして高速化を図った。このハッシュ表は, make-instance が初めて行なわれたとき, クラスベクタの中に作られる。なお, コンパイラでは, インスタンス変数は udo の中でのオフセットがコンパイルド・コード中に書かれており, 参照は極めて高速である。

クラス変数は, クラスベクタの中に, 変数名と値とのペアからなる vector として登録される。クラス変数は TAO の通常の意味での変数ではないので, eval 内部の変数アクセス機構では探索されない。

クラス変数やインスタンス変数をアクセスしたとき, また:gettable のインスタンス変数参照メソッドでも rpr, rpf が設定される。従って, たとえば:gettable の指定されたインスタンス変数 x を持つオブジェクトが変数 i に束縛されているとすると,

(![i x] *value*)

によってインスタンス変数の変更が可能になる。:settable の指定なしでもインスタンス変数の変更が可能であるため, メソッド表を圧迫しないという利点もある。

3.3.6 メッセージ伝達のメカニズム

3.3.6.1 udo へのメッセージ伝達

udo へのメッセージ伝達では以下のような処理を行なう。

(1) **udo** の属するクラスのクラスベクタのメソッド表を調べる。

(2) もしメソッド表がまだないならば, 3.3.4.3で述べたように, メソッド表を作る。

(3) 与えられた式の `cadr`(すなわちセレクタ) をキーとして, メソッド表をバイナリ・サーチする。メソッド表のエントリはメソッド **applobj** など 4 種ある。各エントリに対応する動作を詳しく述べる。

(i) インスタンス変数参照メソッド
`:gettable` や `:settable` で自動的に作られた, インスタンス変数をアクセスするためのメソッド。その実体は特殊なタグ[19]の立った数で, インスタンス変数の **udo** ベクタ内でのオフセットを表わす。この数が正ならば `:gettable` による参照メソッドを, 負ならば `:settable` による代入メソッドをそれぞれ表わし, 絶対値がオフセットである。バイナリ・サーチでこのエントリに達したときは, **udo** の対応するインスタンス変数をアクセスする。関数評価用のフレームは作らない。

(ii) メソッド **applobj**
`defmethod` で定義され, 継承を経てメソッド結合済みの **applobj**。図 3.4に示すような, メソッドを評価中であることを示すスタック・フレームを作る。このフレームには, 環境を伴っている[20]ことを示すフラグや, 変数の探索においてスコープを限ることを示すフラグを立てる。引数は順次評価され, スタックに積まれてメソッド本体が実行される。

(iii) メソッドが未作成であることを示すフラグ
`defmethod` で定義されていてもまだ起動されたことがなければ, **applobj** のかわりにそのことを示すフラグ[21]が入っている。このときは, メソッド **applobj** を作ってメソッド表に挿入し, しかるのち (ii) の場合と同様の処理を行なう。

(iv) それ以外
クラス定数とみなして直ちにその値を返す。クラス定数とは常に定数を返す特殊なメソッドである。

これらの処理において必要な種々のチェックは, ELIS のマイクロプログラムの持つ多重分岐によって並列的に行なわれ, オーバヘッドはない。

メッセージ伝達式を評価する機構は, Lisp の式との識別とメソッドの探索以外は, Lisp の式の評価と基本的に同じである。しかもその差異の部分の処理時間はあまり大きくない。このことは, TAO のオブジェクト指向機能の実用性を保証するものとなっている。付録 B に, メッセージ伝達式の評価にかかわる `eval` のマイクロプログラムの主要な流れの部分を抜き出したものを示す。

---

[19] **empty** というタグを使用した。

[20] **udo** のインスタンス変数は一種のバインディングである。

[21] 3.3.4.2で述べた **opr**。

last arg
...
first arg
self
tf
ef
applobj for the method
message passing form
rtn

- rtn .... このフォームを呼んだルーティン(マイクロプログラム)への戻り先番地。
- message passing form .... 評価中の式(メッセージ伝達式)
- applobj for the method .... このメソッドの **applobj**
- ef .... 環境フレーム(アクセス・チェイン)のリンク。このポインタには,このフレームが環境ベクタ(すなわち **udo**)を持っていることを表わすために tage ビットが立てられる。
- tf .... トップ・フレーム(コントロール・チェイン)のリンク
- self .... このメッセージの受け手オブジェクト。変数 `self` に束縛されている。
- first arg .... このメソッドの第一引数。以下,第二,第三,...の順にスタックに積み上げられる。変数の個数は,**applobj** に書かれている。

図 3.4: メソッド実行のためのフレームの構造

3.3.6.2 基本データ・オブジェクトへのメッセージ

3.2.2.3で述べたように,メッセージ伝達式を利用して算術演算を中置記法で記述することができる。式 `[x + y]` は,**x** が数ならばこの式の **cadr** の + を見て,この式は中置記法で書かれた足し算であると解釈する。もし **x** が数でなく,たとえば **udo** であったとするとそのメソッド表を探索し,+ というセレクタに対するメソッドを(もしあれば)起動する。算術演算の中置記法のような基本的な演算は特別扱いで処理し,メソッド表は経由しない。この場合,セレクタである +[22]のメモリ上のアドレスから簡単なオフセット計算で直接求まる WCS 上のアドレスが,加算メソッドのマイクロルーティンへの入口になっている。加算の実行本体は Lisp の式 `(+ x y)` のそれと共通である。スタック・フレームは作らない[23]。

数以外の基本データオブジェクトに対しても,メッセージ伝達式を利用してさまざまな演算を実現した。たとえば,ELIS のメモリの 1 語(64 ビット)に様々の演算を行なう **locative** オブジェクトのクラスがある。これは,ELIS のアドレスを直接に代表するデータ型である。その一つ,ELIS の語を 64 ビットの符号なし整数とみなす **64builoc** は,たとえば,

```
(unsigned-integer-locatives x)
```

といった式で作られる。この式は,ELIS のあるアドレスを指す **64builoc** のデータを一つとり,変数 **x** に代入する。**64builoc** への代入は,メッセージ `<-` を用いる。このとき,`<-` の引数は逆ポーランド記法に変換され,マイクロプログラムで書かれた仮想的なスタック・マシンによって解釈実行される。たとえば,

[22] この場合,**sysid** という特別の **id**。

[23] ただし,**send** や **apply-send** で起動された場合にはメソッド表を経由する通常のメソッド起動が起こる。すなわち,メソッド表には,+ や - などのようなメソッドも登録されている。この場合にはスタック・フレームを作る。

```
[x <- (x + 1)]
```

という式は, 右辺の (x + 1) が

```
(sys:64pol x x 1 {opr}0)
```

に変換され, sys:64pol という仮想スタック・マシンのシミュレータ関数によって実行される。この関数の第一引数は左辺の **locative**, 第二引数以降は右辺の逆ポーランド記法の式である。{opr}0 は, **opr** というタグを持つ特殊な数で, 仮想スタック上のトップの2つの要素の加算を行なう演算子とみなされる。この計算は ELIS のメモリ上の語をアキュムレータとして使用することによって行なわれるので, 64 ビットまでの計算ならば **bignum** の演算のようなセルの消費も起こらず効率がよい。このように, **locative** を用いた計算は, アドレスと記号とが直接に関連づけられているという意味で Fortran スタイルの計算機構になっている。

3.3.7 コンパイラ

オブジェクト指向のプログラムでは, 変数などの押えているオブジェクトが何であるか実行時にしか決められないため, そのコンパイレーションを最適にするためには, 宣言が欠かせない。しかし, 宣言なしでも正しく動作するコンパイル結果を出力するため, 実行時までコンパイラの出力コードをダミーのものにしておくなどの実現法をとった。インスタンス変数は見掛け上自由変数であり, クラスの階層構造が確定するまではコンパイルに必要な情報が得られない。それゆえ, コンパイル時点で判断できるもの以外はダミーのバイトコードにコンパイルする。そのコードは, 実際にメッセージ伝達が起こって最初にメソッドを作るときに正しいもので置きかえられる。

メッセージ伝達式では, 実際にメッセージが送られたとき, それを受け取るオブジェクトが何であるかということは, 一般には実行してみるまでわからない。たとえば, [x + y] という式は, 何らかの宣言がない限り, 変数 **x** が数であるかどうかの保証はない。**x** は, + というセレクタを持つメソッドの定義されたクラスの **udo** かもしれない。従って, 実行すべき手続きが決められず, コンパイルもできないことになる。そこで, メッセージ伝達式は, 宣言のある場合やコンパイル時に手続きがわかってしまうもの以外は, **メッセージ伝達をする**コードにコンパイルする。すなわち, 一般には次のようなコードにコンパイルする。

- [A B] の形の式
  compile A
  push const B
  send-unary-msg

- [A B C] の形の式
  compile A
  push const B
  compile C
  send-binary-msg

- [A B C D ... Z] の形の式 (一般形)
  compile A

push const B
compile C
compile D
...
compile Z
send-general-msg n ; n は引数の個数 (C から Z までの長さ)

上記において, send-unary-msg, send-binary-msg, send-general-msg は, それぞれメッセージ伝達式の評価のために, メソッドの探索, スタック・フレームの確立, 本体の実行, という一連の処理を行なう命令で, 引数の個数に対応して 3 種類用意されている。

3.4 TAO オブジェクト指向の性能

3.4.1 メソッド探索の速度

メソッドの探索にハッシュ法を用いるかバイナリ・サーチを用いるか, あるいは他の実現法を採るかは一つの重要な選択である。TAO では, メソッドが大量に定義された場合のハッシュ表の占めるメモリ占有量の大きさを嫌ったことと, 高速のバイナリ・サーチが ELIS のマイクロプログラミングによって実現できることの二つの理由からバイナリ・サーチ方式を採用した。バイナリ・サーチを行なうマイクロプログラムのルーティンは, 内部に, 1 回のアドレス計算で表を調べるたびに回る 6 マイクロステップのループを持っている[24]。後述するように, 実測結果によると, メソッド探索はオブジェクト指向の性能を殆ど悪化させない。

3.4.2 インスタンス変数のハッシュ

3.3.5で述べたインタプリタにおけるインスタンス変数のハッシュ表の効果を吟味する。以下のテストでは, 100 個のインスタンス変数を持つクラスを定義し, そのクラスに, インスタンス変数を 10000 回参照するだけの本体を持つメソッドを定義して実行時間を測定した。クラスの定義は,

```
(defclass a ()
    (x1 x2 x3 .....  x49 x50 x51 ....  x98 x99 x100) ())
```

のようになっており, メソッドは

```
(defmethod (a test) (x) (for i (index 1 10000) (null nil)))      (1)
(defmethod (a test) (x) (for i (index 1 10000) (null x1)))       (2)
(defmethod (a test) (x) (for i (index 1 10000) (null x50)))      (3)
(defmethod (a test) (x) (for i (index 1 10000) (null x100)))     (4)
```

をそれぞれ定義している[25]。メッセージの受信オブジェクト自体はグローバル変数 instance に代入し, 次の式を時間測定に用いた。

[24]試作版の ELIS では 1 マイクロステップは 180$n$sec であるので, このループを 1 回まわるごとに約 1μsec かかる。この結果, N 個のエントリを持つメソッド表の探索には, 平均 $log N$ μsec かかる。

[25]同じセレクタでメソッドを定義し, メソッド探索時間の条件を同じにしている。なお, コンパイラでは, for 文中の単体の変数参照を無視するコードを出力するので, ダミーとして関数 null を挿入している。

表 3.2: インスタンス変数参照時間 (μsec)

	インタプリタ		コンパイラ
位置	ハッシュあり	ハッシュなし	(ハッシュは無関係)
1	7.4	5.5	2.4
50	7.4	55.6	2.4
100	7.4	106.7	2.4

```
(speed [(value 'instance) test 1])
```

関数 **speed** は, 引数として与えられた式を実行し, それに要した CPU 時間を返す。この (2)(3)(4) の時間から (1) の時間を引き, それを 10000 で割った値がそれぞれインスタンス変数参照に要する正味時間である。表 3.2に, その正味時間を示す。

この結果から, インタプリタにおいてハッシュが有効なときのインスタンス変数のアクセス時間は一定である[26]。ハッシュ表のないときは **udo** 内部をリニア・サーチするので, インスタンス変数の **udo** 内部での位置によってアクセスの速度が異なる。実際のプログラムでは **udo** のサイズは数 100 に及ぶものもあるので, インスタンス変数のハッシュは十分に有効である。インスタンス変数の個数が 2 個や 3 個などと非常に小さいときには, ハッシュを用いない方が高速であるが, その差は高々 20% 程度であって問題ではない。なお, コンパイラでは, インスタンス変数は **udo** の中でのオフセットがコンパイルド・コード中に書かれており, 参照は極めて高速である。

インスタンス変数の参照時間を他の Lisp 変数の参照時間と比較するため, 次のようなメソッドを定義して同様の測定を行なった。

```
(defmethod (a test) (x) (for i (index 1 10000) (null x)))      (5)
(defmethod (a test) (x) (for i (index 1 10000) (null y)))      (6)
```

ここで, (6) の変数 **y** は, グローバル変数のとき (6-1), スペシャル変数のとき (6-2), およびセミグローバル変数のとき (6-3) のそれぞれについて測定した。これらの変数は, スタックの深まり方に応じて参照の速度が変化するので, 測定条件として TAO の通常のトップレベルで

```
Tao>(!y 1)
Tao>(speed [(value 'instance) test 1])          (6-1) の測定
Tao>(let (y) (declare (special y))              (6-2) の測定
        (speed [(value 'instance) test 1]) )
Tao>(semi-globals y)
Tao>(speed [(value 'instance) test 1])          (6-3) の測定
```

とした。セミグローバルはサイズ 60 のバイナリ表で, **y** は最悪の場合として見つかる。測定結果から空の本体 (1) の時間を引き, 回数 10000 で割った値を表 3.3に示す。

インタプリタでは, インスタンス変数のハッシュ表が存在するか否かでメソッドの中からのスペシャル変数の参照速度が変わる。インスタンス変数は局所変数の次に探索されるので, もしハッ

[26]ハッシュのヒットに応じてこの値は多少変化するが, この測定結果では一定になっている。

表 3.3: インタプリタでの各種変数の参照時間 (単位 μsec)

	変数の種類	ハッシュなし	ハッシュあり
(5)	local	1.68	1.69
(6-1)	special	120.07	20.12
(6-2)	semi-global	120.08	20.12
(6-3)	global	120.47	20.47

シュ表がないとスペシャル変数の探索においても **udo** の中を無駄に探索することになってその探索が遅くなる[27]。しかし, ハッシュ表があるとインスタンス変数であるか否かが直ちにわかり, 不要な **udo** 内の探索は省かれる。

インスタンス変数の探索の優先度を高くしたため, そのアクセスは極めて高速であり, Lisp の変数と比較してほぼ同等かあるいはそれ以上となった。このことは, オブジェクト指向によるプログラミングの実用性を高める上で重要な要因であると言うことができる。

3.4.3 簡単なプログラムによる Lisp プログラムとの速度比較

本節では, fibonacci 数列を計算する再帰的プログラムを用いて, Lisp とオブジェクト指向の速度比較を行なった結果を示す。関数 `fibonacci` の定義は,

```
(defun fibonacci (n)
    (if (< n 2) 1
        (+ (fibonacci (1- n))
           (fibonacci (- n 2)) )))
```

とする。オブジェクト指向では, **integer** のクラスに fibonacci 数列を計算するメソッドを定義し, そのインスタンスである整数にメッセージを送る。メソッドの定義は,

```
(defmethod (integer fibonacci) ()
    (if [self < 2] 1
        [[[self - 1] fibonacci]
       + [[self - 2] fibonacci]] ))
```

とする。`self` には, このメッセージの受け手が代入されている。結果を表 3.4 と表 3.5 に示す。

これらの測定では, メソッド表のサイズを 30 および 100 とし, `fibonacci` のメソッドは最悪の場合として見つかるようにしてある。表のサイズ 30 と 100 とでは, 1 回のメソッド探索について 2 回分のループの時間差が生ずるので, 両者で約 2μsec の差になる。`fibonacci` の再帰的定義では, 引数 19 のときは 13529 回, 20 では 21819 回, 22 では 57313 回, 25 では 242785 回呼ばれる。ゆえに, メソッド表のサイズの差による実行時間の差は, それぞれ 27msec, 44msec, 115msec, 486msec になり, 測定結果とよく一致している。また, コンパイル版では, 変数 **n** または `self` は **fixnum** で

[27]深い変数束縛方式をとっているため。

表 3.4: インタプリタによる fibonacci の実行速度
(単位 msec)

n	*Lisp version*	*Object-oriented version (Method table size)* 30	100
19	521.8	583.5	612.8
20	844.4	944.3	991.6
22	2210.7	2472.3	2596.0
25	9363.5	10472.1	10997.5

表 3.5: コンパイラによる fibonacci の実行速度
(単位 msec)

n	*Lisp version*	*Object-oriented version (Method table size)* 30	100
19	143.2	384.2	413.9
20	231.7	621.7	669.8
22	606.5	1627.7	1753.3
25	2569.1	6894.9	7427.4

表 3.6: 応用プログラムによる評価

	zen	net	jpro
プログラムの規模 (行数)	18386	12654	7199
メソッド表を持つクラスの総数	24	11	12
メソッド表のない (抽象クラスなど) クラスの総数	13	10	3
メソッド **applobj** の総数 (**opr** を含む)	5569	667	330
実際に起動されたメソッドの総数 (典型例)	750 (13%)	218 (33%)	30 (9%)
プログラム中の `defmethod` の総数	453	154	204
継承で作られたメソッド **applobj** の総数	5516	513	126
メソッド継承率	0.92	0.77	0.38

あることを宣言してある。インタプリタ版では, Lisp とオブジェクト指向の速度の差は 20% 以内である。コンパイラでは, Lisp 版の方が 2 〜 3 倍高速である。これは, Lisp 版では再帰部分を分岐命令にコンパイルする等の最適化を行なっているのに対し, オブジェクト指向版はメソッド表経由のメッセージ伝達にコンパイルされていて, 現在のところ最適化が十分でないためである。

3.4.4 応用プログラムによる評価

応用プログラムが実際にどのように作られているかを分析することは, システムの評価では有用である。ここでは, オブジェクト指向によって書かれた emacs 風のエディタ zen [42], zen に組み込まれた日本語入力システムである jpro [43], TAO 上での TCP/IP をサポートする net [44] の三つの応用プログラムをとりあげていくつかのデータを示し, TAO オブジェクト指向の総合的な評価を試みる。なお, ここで用いた応用プログラムはいずれも完成して実用に供され, 非常に多くの使用実績のあるプログラムである[28]。

3.4.4.1 メソッド表のサイズとメソッドの呼ばれかた

適当な使用条件のもとで, 各応用プログラムにおけるメソッド表の平均的な大きさを調べた[29]。

表 3.6において, メソッド継承率は, 各クラスにおいて `defmethod` によって定義されたもの[30]の総個数と, 各 **id-message-vector** の中で **applobj** または **opr** になっているものの総数との比である。この値は, 継承されているメソッドが多いほど 1 に近くなる。メソッド継承率は, 3.4.4.2で述べるインスタンス変数継承率とともに, 継承の利用度をみるための尺度である。

[28]本論文の作成でも, 入力には zen と jpro を使用し, net を介して UNIX マシンに転送して処理した (整形は LaTeX による)。

[29]「適当な使用条件」というのは, そのプログラムの標準的な機能が動作している状態で, というほどの意味である。立ち上げの後, 実用的な使用状態に達したときは, ほぼここでの測定条件に近くなっていると考えられる。また, 呼ばれたメソッドの総数なども, 本節全体を通じて同じにはなっていない。これは, 測定条件を厳密に揃える事にそれほど大きな意味があるとは思えないため, システムの稼働状況がほぼ通常の状態にあると判断される機械の上で, 妥当な稼働時間を経たと思われるときに測定し, 傾向を捉えようとしたためである。

[30]ひとつのセレクタを 1 個と数え, たとえば `:after` や `:before` などはまとめてひとつとみなす。

表 3.7: インスタンス変数の継承

	zen	net	jpro
クラスの総数	37	21	15
インスタンス変数がまだ作られていないか, abstract-class であるクラス	13	10	3
インスタンスの平均サイズ	112	51	27
インスタンス変数継承率	1294/1457=0.89	440/508=0.87	6/81=0.07

3.4.4.2 udo の作られ方と udo の平均的な大きさ

udo の大きさは, スーパークラスから継承されてきたインスタンス変数が加わるため, ユーザ自身が考える以上のものになる。インスタンス変数の継承のようすを調べる尺度として, インスタンス変数継承率を定義する。インスタンス変数継承率は, 各クラスのインスタンス変数の個数を総和し, そのうちスーパークラスから継承してきたものの個数が占める割合である。この値は継承が全くないとき 0, すべてのインスタンス変数が継承したものであるとき, つまり自分のクラスで定義したインスタンス変数が全くないとき 1 となる。

表 3.7に, 三つの応用システムでのインスタンス変数の継承に関するデータを示す。インスタンス変数継承率は, クラスの階層構造を積極的に使っているかどうかのひとつの尺度である。スーパークラスを部品とみなすオブジェクト指向では, 部品が細分化されていればいるほどこの値が 1 に近くなると考えられる。インスタンス変数の宣言とそれを操作するメソッドの定義とは通常同一のクラスで行なわれるので, 当然, 3.4.4.1で述べたメソッド継承率とここで定義したインスタンス変数継承率の値には相関がある。

インスタンスのサイズはプログラミングのスタイルに応じてばらつきがあると考えられるが, 継承の深さが深いほど大きくなる傾向があることは明らかである。しかし, 実行時にインスタンスを大量に生成しない限り, 個々のインスタンスのサイズはあまり問題にはならない。

継承の利用度は, 本来の継承機能のねらいであったモジュール化の程度も反映していると考えられる。この意味で, クラスやメソッドの部品化を強く意識して作られた zen において, メソッド継承率とインスタンス変数継承率が 1 に近くなっていることは, 妥当な結果であると言える。

3.4.4.3 メソッド表作成方式の評価

メソッド表を作成する方式については, 使用経験に基づく変遷があった。最初の実装では, 最初のメッセージ伝達時に, そのクラスのメソッド表のすべてのデータを作っていた。しかし, 応用プログラムの規模が大きくなり, メソッドの個数が増えてくると, **id-message-vector** を作るのに必要な時間が無視できないほどになった。たとえば, zen では, **id-message-vector** の作成に 3 分以上かかるようになり, デバッグの効率上問題になった。その内訳を解析した結果, この時間の大半は, メソッド本体の **applobj** を作る時間であることがわかった。

また, 最初の実装では, スーパークラスや自分のクラスでメソッドの再定義や追加などの変更があるとメソッド表を廃棄し, それ以後の最初のメッセージ伝達時に再構築していた。メソッド表はバイナリ・サーチできるようにソートされた **vector** であるから, そのクラスとそのすべてのサブ

クラスのメソッド表は使えなくなるからである。メソッド表は，たった一個のメソッドの再定義でも再構築されることになる。このことは，メソッド表が小さく，その作成時間が無視できる程度の応用プログラムでは問題にならないが，メソッド表が大きくなってくると問題が顕在化する。特にインクリメンタルにメソッドを追加するようなプログラム開発スタイルでは深刻な問題となる。また，メソッドの **applobj** が大量になり，実際には一度も起動されないにもかかわらずメモリを圧迫するという問題も生ずる。実際，表 3.6に示したように，zen において，5500 個以上のメソッドのうち，実使用で起動されているメソッドは，750 程度である。この理由は，zen は極めて豊富な機能を提供しており，通常の使用ではそのごく一部の機能しか起動されないことと，スーパークラスから不要なメソッドをも継承してきていることによると考えられる。

こうした問題を解決するため，3.3.4.3に述べたように，メソッドの再定義や追加の際のメソッド表への影響を最小限にとどめて「仮り」構造を入れておき，真に必要になったときに必要なものだけを構成するという方式に変更した。また，`defmethod` の動作も，次のように変更した。

(1) まず，クラスベクタの中の **defclass-skeleton** に，`defmethod` の式を字面のまま登録 (あるいは変更) する。

(2) そのクラスが **id-message-vector** を
 (2-1) 持っていないときそのクラスに対しては何もしない。
 (2-2) 持っていたらそのクラスにおいて `bas:update-method`(後述) を行なう。

(3) そのクラスのすべてのサブクラスに対し，もしそのクラスが **id-message-vector** を
 (3-1) 持っていなければ何もしない。
 (3-2) 持っていたらそのクラスにおいて **applobj** を {`opr`}0 に変える。

この処理で用いられる関数 `bas:update-method` は，次のような動作をする[31]。

(1) そのクラスを起点としてスーパークラスを辿り，与えられたセレクタのみをメソッド結合する。こうして得られた **applobj** を `A` とする。

(2) そのクラスの **id-message-vector**(これは **nil** でない) に，当該セレクタを持つメソッドが
 (2-1) 存在するならば，そのメソッドを `A` で置きかえる。
 (2-2) 存在しないならば，`A` を **id-message-vector** に挿入する。
 (これは，`vcons` と `sort` を改めてやりなおすことになる。)

この方式の潜在的な問題として，`defmethod` を一つずつ行なうたびにメッセージ伝達をする，ということを繰り返すと，そのたびに **id-message-vector** の `vcons` と `sort` が起こり，効率が著しく悪くなる。メソッドのファイルを見ながら部分的に再定義と実行を繰り返すようなことを大きなプログラムで行なうと，こうしたことが問題になる可能性もあるが，このようなプログラミング・スタイルは変則的であろう。

`defmethod` 時にメソッドを完全には作らないことにした結果，メソッドの再定義は，殆どの場合瞬時に完了する。さらに，実際には呼ばれない大量のメソッドが作られてしまうという問題も解決

[31] この関数は，3.3.4.3で述べた手続きの第二段階からも呼ばれる。

する。ただし，この時間はクラス階層の辿りを含むので，階層が深ければ深いほど，またメソッド結合が複雑であればあるほど長くなる。

表3.6から，**defmethod**されたメソッドが，実際には10%から30%程度しか使われていないことがわかる。このことはオン・ディマンドでのメソッド作成の有効性を裏付けている。継承によって必要なものもそうでないものもサブクラス側に取り込まれるため，こうしたことが生ずると考えられる。zenについて，この方式で節約できる静的なメモリの容量を試算すると，約4300個の**applobj**に対して，各**applobj**(4セル)が，極めて簡単な4セル分のメソッド本体しか持たないと仮定してもおよそ280Kバイトにもなる[32]。**applobj**をオン・ディマンドで作成する方式は，こうした無駄がシステムを圧迫しないようにする手段となった。また一方，クラス階層の設計において継承を細かく制御できるようなメカニズムの導入を考慮する余地があることも示唆している。

3.4.4.4 メソッド結合の使用率

多重継承では，クラス階層の設計はしばしばサブクラス側でのメソッド結合を用いた手続き設計の問題と関連して試行錯誤を余儀なくされる。また，プログラムの性質に応じてメソッド結合の形態もいろいろありうる。このためにいくつかのメソッド結合を提供したが，経験によれば，実際にはデフォールトの**:daemon**以外は，あまり使われていない。

ある時点での測定では，zenではメソッド表にすでに登録されている750個の**applobj**のうちメソッド結合で作られたものは49個あった。そのすべてが**:daemon**結合である。同様のデータをnetについて見ると，実際に呼ばれたメソッド218個のうち4個がメソッド結合で作られていた。jproではメソッド結合はなかった。zenやnetは，継承を多用しているが，メソッド結合の使用率は低く，使われていても**:daemon**程度である。

メソッド結合があまり使われない理由の一つとして，概念自体があまりわかりやすくなく，最終的に作られるメソッドの形をプログラマが管理するのが難しい，という点が考えられる。**defmethod**で手続きを定義するときに，その手続きがどういう結合の部品になるかを予測してコーディングするのが難しい。これは，Flavors流のオブジェクト指向において，オブジェクト同士の情報隠蔽は完全にできるが，継承関係にあるクラス間ではまったく情報隠蔽ができていないためである。この意味で，メソッド結合の概念は問題を含んでおり，今後の研究課題であると言わなければならない。

3.4.4.5 メソッド本体の共有率

3.3.4.5で述べた，ハッシュを用いた**applobj**の共有の効果を示す。まずzenとnetおよびjproについて，適当な時間稼働したあとそれぞれのシステムのメソッド表を調べ，実際に呼ばれたメソッドの**applobj**をすべて取り出した。そして，それらの中でメソッド本体が完全に同じであるもの[33]がどの程度あるかを調べるため，全体の**applobj**の数と，それらから重複を除去して残った**applobj**の数とを求めた。結果を表3.8に示す。実際に削減された**applobj**の数はインタプリタで40%程度，コンパイラで10%〜40%である。コンパイルされたメソッドは，もとの本体がS式として同じであっても，異なるクラスに継承されたときにはインスタンス変数参照のオフセット部分が異なるので異なるバイトコードとなり，共有率は下がる。しかし，この結果から，メソッド**applobj**の共有の有効性を確認することができる。

---

[32] 4300*8*8=275200。実際には4セルのメソッド本体ということはありえないから，この数倍から数十倍になる。

[33] 関数**eq**で比較して**t**になるもの。

表 3.8: メソッド共有率

	インタプリタ			コンパイラ		
	zen	**net**	**jpro**	**zen**	**net**	**jpro**
呼ばれたメソッドの総数	769	250	96	722	218	89
重複除去後のメソッド総数	446 (58%)	144 (58%)	95 (99%)	642 (89%)	137 (63%)	87 (98%)

3.5 結論

本章では, TAO における複合プログラミング・パラダイムの一つの柱であるオブジェクト指向の機能とその実現法について論じた。TAO では, Flavors を手本としながら独自の機能を加え, 処理系の基本的なメカニズムとしてオブジェクト指向を組み込み, Lisp の言語仕様を拡張する形で, かつその性能を損わずにオブジェクト指向を融合し, 実現法を工夫して大規模な応用に耐えるシステムとして提供した。

言語表層では, S 式の意味を拡大してメッセージ伝達式を Lisp プログラムの中に埋め込んだ。この式は算術演算などの中置記法を Lisp の中で可能にした。Lisp の基本データをオブジェクトとして扱う一方, ユーザ定義のオブジェクトを Lisp のデータで表現した。Lisp の関数呼び出しとオブジェクト指向のメッセージ伝達式のプログラミング上の融合は自然なものとなった。

TAO のオブジェクト指向は, 速度性能という点でも一応成功していると考えられる。オブジェクト指向の性能は, (i) メッセージ伝達式の解釈部, (ii) メソッドの探索と起動, (iii) メソッド本体の実行部の三つの要因で決まる。TAO では, 専用計算機 ELIS のマイクロプログラミングによって, (i) 式の解釈部を `eval` の中に埋め込み, (ii) メソッド探索をバイナリ・サーチで実現した。その結果, メソッド本体に至るまでの解釈のステップが高速化できた。また, (iii) メソッドの実行自体はインスタンス変数の参照以外, 関数の実行とまったく同一であるため, メソッド固有のオーバヘッドはない。インスタンス変数はハッシュ表によって高速化することができた。これらにより, インタプリタにおける TAO のオブジェクト指向機能は, Lisp 部と比較して遜色のないものとなった。

TAO では, 性能にかかわる部分での動的なクラス階層辿りを極力避ける実現法をとったので, 大規模な応用プログラムでは静的なメモリ効率が悪化した。そこで, メソッドの内部表現の共有化や徹底したオン・ディマンドによる内部データの構築を行なった。このオン・ディマンド方式はメモリ消費を著しく削減し, メソッド表の構築時間を大幅に短縮するのに極めて効果があった。TAO のオブジェクト指向で書かれたプログラムは, ファイルなどからロードされても実際にはメモリ内に殆ど何も作られておらず, 真に必要になったときに必要になったものだけが作られる。この方式は, プログラム開発時のターン・アラウンドを削減して会話性を保証し, TAO のオブジェクト指向の実用性の上で重要な意味を持つこととなった。

このように本システムでは, Lisp マシンのハードウェア性能を背景に, 大規模な応用プログラムの開発にも耐えうる性能を持つオブジェクト指向を Lisp に融合し, 複合プログラミング・パラダイムの実現性と実用性を明らかにした。

TAO のオブジェクト指向は実応用において多く利用され, その有効性も認識されているが, オ

ブジェクト指向における諸機能の中には，TAOで採用したものも含めて解決すべき問題がいくつかある。たとえば，継承と複雑なメソッド結合がプログラムの了解性を阻害している点もその一つである。情報隠蔽は異なるクラスのオブジェクト間では達成されているが，継承関係にあるクラス間では殆ど達成されない。既存のクラスを部品として使うときにはそのインプリメンテーションを知っていなければならず，ブラックボックスと考えることはできない。このことは，プログラムの設計時のみならずデバッグ時にも，プログラマに余分の負担を強いることとなる。実際，メソッド結合の結果自動的に合成された手続きの動作中にエラーが発生した場合，その個所のプログラム・セグメントがどのメソッドであったのかということが，よりわかりにくくなってしまう。

また，オブジェクト指向的なモデル化のやりやすさも，プログラマを正しいプログラムに一意に導くものではなく，過大に評価することはできない。何をオブジェクトと考えるかというモデル化への指針を，システムが可能な限り提供することも，知能的なシステムでは指向していくべきであろう。この意味では，クラスライブラリを充実したり，それを活用しうるような支援系を実現したりする等の方向が重要である。

TAOのオブジェクト指向や，そのもとになっているFlavors, Smalltalk-80などのシステムは，クラスの構造が実行時に変化するような動的なオブジェクト指向システムを効率よく実現するのは難しい。こうした目的には，別の実現法を考えなければならないであろう。特に，実行時に変化する構造に対しては，そのことを前提としたコンパイラ込みの設計が必要である。こうしたオブジェクト指向の概念の再吟味や新しい機能の摸索もまた今後の課題である。

第 4 章

並行オブジェクト指向とそのロボット制御への適用

4.1 緒論

複合プログラミング・パラダイム言語 TAO では, Lisp をベースとしたオブジェクト指向を実現した。このオブジェクト指向は, 基本的に逐次実行型のパラダイムである。これに対し, 現実の応用分野では, オブジェクトを独立したプロセスと見てモデル化を行ないたい場合も多い。特に実世界を素直に表現するためには, 並行オブジェクトによる分散方式の考え方がより自然である。

本章では, TAO のオブジェクト指向に並行処理の機能を付加し, 知能ロボットの高次制御に応用する問題を取り上げる。ロボットはそれ自身物理的な実体を持ち, 各種のセンサからの多様な情報を処理したり, アクチュエータを介して実世界に物理的作用を及ぼしたりする計算機システムである。物理世界との様々なインタフェースや遂行する作業の多様性に応じて, ロボットは異質な構成要素からなる複合システムとなる。特に, センサやアクチュエータ等のハードウェア構成技術が進歩してきたため, それらを組込んだロボットに作業を教示したり柔軟に制御したりするシステムを実現する必要性が高まっている。

複合システムであるロボットに作業を教示するには, その構成要素に対応したモジュール構成にもとづいて, 作業環境や処理手順を明解に表現し, 不確定な要因やシステムの変化にも容易に対応できるような分散型のソフトウェア構成をとることが有望である。また, そうしたモジュール構成をとることによって, モジュール間に協調的な相互作用を行なわせ, ロボットの行動能力を高度化することも期待できる。このようなプログラムの構成法をとることにより, システムの部分的な更改や新たなセンサ等の導入に対しても, 作業遂行方針そのものを変更しない限り, 全体のプログラムは影響を受けないようにすることができる。

分散型の知能ロボットの研究では分散問題解決アルゴリズムなどの個別の問題に力点を置いた仕事が多くなされてきた。また, 分散・協調による知能化・ロバスト化などを目指すマルチエージェント・ロボットシステムが注目されている [17]。しかし, ロボットへの教示システムとしてのプログラミングの観点からなされた仕事はあまり多くない。また, オブジェクト指向等を含むプログラミング技術上の成果も十分に適用されてこなかった。複合システムであるロボットは多くの要素技術の上に実現されるが, プログラミングは要素技術ごとにアドホックになされ, ソフトウェア構成法の立場からの議論はまれである。本研究では, ロボットへの作業教示の中でも, いわゆるオフライン・プログラミングと呼ばれる作業プログラミングに, 並行オブジェクト指向を適用する方式を提案する。本研究の意義は, TAO のオブジェクトを並行プロセスとして捉え直し, ロボットプログラミングに適用してモジュラー化と知能化のための方向づけを実証的に行なった点にある。

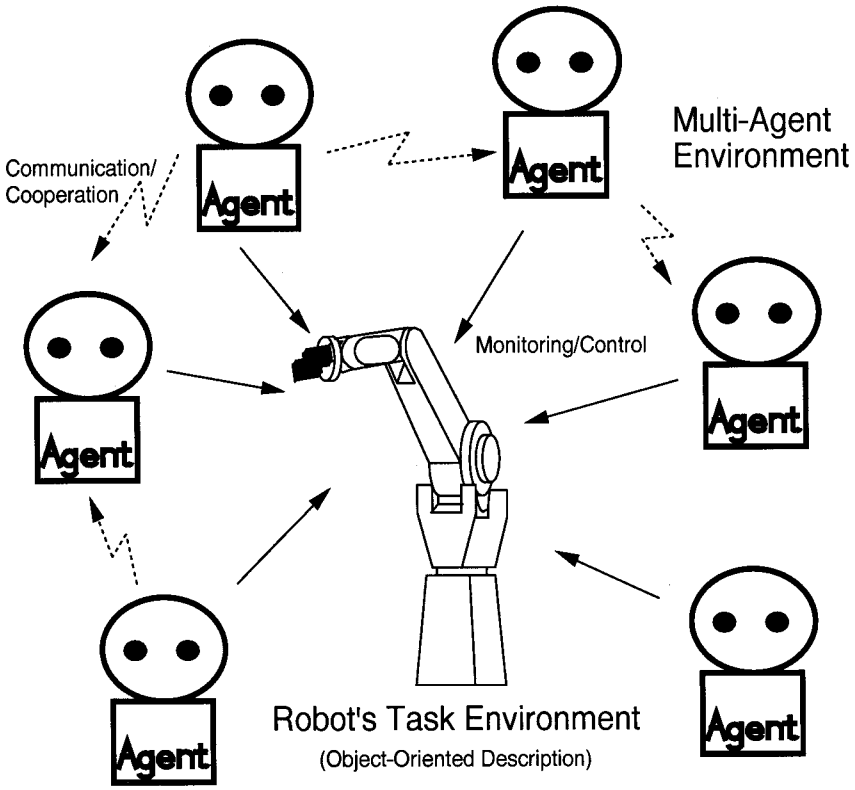


図 4.1: MARCS の概念

本研究で提案した方式では，ロボットへの教示をプログラミングの視点で分解し，その構成要素の処理を，**エージェント**とよぶ並行オブジェクトの担当すべき役割として定義し割り当てるようなモデルを考える [45] [46] [47] [48]。このモデルでは，図 4.1に概念を示すように，ロボットの作業世界はオブジェクト指向で表現され，作業対象物に対応する物理的な実体は論理的なオブジェクトとして記述される。そして，一群のエージェントが対象物に対応するオブジェクトを操作することによって，ロボット全体の作業が遂行される。エージェントは，能動性を持つ独立な論理的実体である。これらは並行的に，かつ必要に応じて相互に交信しながら，システムに課された仕事を分担して共同で処理する。個々のエージェントは，分担する作業に応じた行動をとるためのプログラムを，それぞれ独立のプロセスのもとで実行する。エージェントは，事象の発生やメッセージの交信などによって，新たな活動を始めたり振舞いを変更したりする。個々のエージェントの振舞いは，行動レパートリを実際に起動制御する**行動インタプリタ**と呼ぶメタプログラムによって制御される。

こうしたロボットプログラミングの実験システムを，TAO のオブジェクトに並行プロセスとしての性格を与えることによって実現し，MARCS[1] と名づけた。MARCS では，マルチエージェントをベースとして，操作性のよいロボットプログラミングの実験環境を作るために，

- 物理世界にほぼ直接に対応する論理実体を扱えること，
- 動的で柔軟な操作ができること，

[1]**Multi-Agent Robot Control System**

- インクリメンタルな開発が可能であること,

などを目標とした。物理世界への素直な対応は，オブジェクト指向並行プログラミングによって，極めてよく達成される。また，ロボット制御の処理系が動的な能力を持つことは，実行段階において，物理世界からの多様な情報に柔軟に対処していく上で重要である。ロボットは，予め決められたとおりの動作ができるのみならず，その時点その時点での意思決定にもとづいて行動を変更することもできなければならない，という本質的に動的な性質を持っている。また，インクリメンタルに作業記述を付加することのできる機構は実験的なプログラミング環境では特に重要であり，また，これによってロボットの振舞いの段階的な知能化を目指すことができる。

MARCS 実験システムを用いた多関節マニピュレータによる制御実験では，ハノイの塔のパズル実行させてその有効性を検討した。この中で，エージェントを作業戦略のモジュール化のために利用し，ロボットの行動を段階的に知能化する手法を提案した。すなわち，物理的な対象物やセンサの処理を担当する**機能ベースエージェント**のほか，特に，包摂アーキテクチャ [18] における行動に対応する役割を持つ**行動ベースエージェント**を導入することによって，マニピュレータを操作する作業戦略モジュールを与えた。このエージェントを用いて試行錯誤的な作業戦略を実現し，困難性を伴う作業を信頼度よく遂行できることを示した。さらに，この手法を用いてロボットを段階的に知能化するための展望を示した。この一連の実験から，MARCS によってモジュール性の高い作業記述が可能であることを確認した [49]。

また，エージェントを単なる受動的なデータと見て操作することによってエージェント相互の監視・制御を実現し，ロボットが全体としてある種の内観能力を持つようにすることも提案している。単にマニピュレータを操作するだけの下位のエージェントを，その作業目的に即した高位の判断を行なう別のエージェントが監視し，必要に応じて戦略を変更して下位の作業遂行エージェントに影響を与えうる構成にする。これらのエージェントは独立しているので，ロボット全体として見れば，自分が遂行する作業を，動かしている手先の運動とは別のものとして意識していることになる。このような構成法は，ロボットの行動を作業環境の変動に即して柔軟に適応させるための基本的な機構として有効であると考えられる。本章では，こうした構成法の応用として X ウィンドウによるエージェントの視覚的な監視系を実現し，その有効性を確認した。

4.2 エージェント・モデル

4.2.1 オブジェクト指向による世界記述

ロボットプログラミングでは，センサやマニピュレータ，対象物など外界のさまざまな物理的実体のほか，ロボットの行動を制御するために多様な情報を処理していく必要がある。物理的実体は**物**として意識できる対象であることが多いので，オブジェクト指向による表現がよく馴染む。オブジェクトによって表現されたものは，プログラミング上の論理的実体となる。たとえば，6 軸の力－トルクセンサは，物理的にはさまざまなインプリメンテーションがありうるが，MARCS のオブジェクトとしては，力の xyz 各軸方向の成分 **fx**, **fy**, **fz**, 各軸回りのモーメント **mx**, **my**, **mz** という 6 個のインスタンス変数を持つオブジェクトとして自然に表現される。オブジェクトには，インスタンス変数の参照や変更，あるいはその値の組み合わせから求められる作業遂行上の条件などを計算する手続きがメソッドとして定義される。

4.2.2 エージェント

オブジェクトの中で，ある特別の属性を持ち，ほかのオブジェクトを操作する能動的な性質を持つものをエージェントと呼ぶ。エージェントは，**behavior** という特別のメソッドを持つ。このメソッドは，エージェントが，与えられた状況を解釈し，それに対応した行動をとるための振舞いを制御する。このメソッドを**行動インタプリタ**と呼ぶ。

物理的実体に対応するオブジェクトの多くは受動的であって，何らかのエージェントによって管理される。こうしたオブジェクトのインスタンス変数は，他のエージェントが作業遂行のための情報として参照する。たとえば，力－トルクセンサ・オブジェクトのインスタンス変数は，それを監視するエージェントによって物理センサから読み込まれ，更新される。エージェント群は，物理世界を反映するオブジェクトの世界を共有する。各エージェントの行動インタプリタは，ユーザの与えたプログラムを解釈実行し，他のエージェントとの協調によって作業を遂行する。

エージェントのメソッドの実行環境は，部分的にデータとして参照できる。行動インタプリタはエージェントごとに独立で，個々のエージェントの役割に応じた特異性を容易に取り込むことができる。また，行動インタプリタそのものをデータとして扱うことによって，外部からの行動の変更や，エージェント同士の監視・制御などが実現される。

4.2.3 エージェントとプロセス

並行に実行されるプログラムの単位をプロセスと呼ぶ。MARCS におけるエージェントは，プロセスの付随するオブジェクトである。このプロセスは，付随するエージェントの行動インタプリタを実行する。すなわち，エージェントが生成されると，そのエージェントの行動インタプリタを実行するプロセスが生成され，起動される。

エージェントの行動インタプリタは，作業に応じて自己のメソッドを起動し，実行を制御する。その結果，それらのメソッドは，エージェントに付随するプロセス上で走ることになる。エージェントのプロセスは，メソッドの管理のもとで，必要に応じて他のエージェントのプロセスと同期をとったり通信したりする。また，割り込みによって，強制的に現在の処理を中断し，別の処理を実行することもある。さらに，エージェントのメソッドは，その実行の過程でオブジェクトとしての自分自身や他のオブジェクトを操作する。

4.2.4 エージェント間の情報交換

4.2.4.1 メッセージ

エージェントは，**メッセージ**によって他のエージェントと交信する。各エージェントは，インスタンス変数 **in-coming-mailbox** 等にメールボックスをいくつか持つ。メールボックスはキューであり，投入されたメッセージが到着順につながれる。メッセージは，受信側エージェントのメールボックスに投入することによって送信される。メッセージの持つ意味は，送受信する両エージェント間の行動インタプリタの定める約束によって決まる。

行動インタプリタは，到着したメッセージを順に拾い上げて処理する。メッセージの受信は，エージェントが行動を開始するための動機の一つになる。メッセージを受信するエージェントは，送信側エージェントにとっての**近傍**であるという。

メッセージの送信モードには，以下のように，同期 / 非同期と送信 / 交信がある [50]。

(1) 同期送信では，送信者は，受信者がこのメッセージを受信するまで待っている。受信者は，返書受け付け先に，受信した旨のメッセージを返送しなければならない。

(2) 非同期送信では，送信者はこのメッセージを送りっぱなしにするだけで，その結果には無関心である。

(3) 同期交信では，送信者は受信したかどうかの情報だけでなく，メッセージの処理結果を受け取ることを希望する。送信側は，メッセージの送信時点でブロックしているので，受信側は処理結果を返送しなければならない。

(4) 非同期交信では，(3) と同様，送信者は処理結果の返送を期待しているが，メッセージの送信時点ではブロックせず，結果が真に必要になるまで別の処理をしている。

エージェント間のメッセージは，いつも直ちに処理されるとは限らない。メッセージを受け入れるかどうかの主導権は送信側でなく受信側にある。メッセージの送信は，単に受け手エージェントのメールボックスにメッセージを投入するだけである。受け手エージェントの行動インタプリタは，そのメッセージが受理条件を満たしていると判断したときに，それを処理する。もし条件が整わなければ，そのメッセージの処理は拒否されるかまたは延期される。

4.2.4.2 イベント

エージェント同士は，メッセージによって直接通信するほか，**イベント**を通じた通信を行なう。イベントは，エージェントの世界に発生する論理的な事象である。イベントは，主にそれを担当するエージェントによってグローバルな掲示板にポストされ，それを待っているエージェントによって検知される。イベントは，それらのエージェントの行動インタプリタを活性化する動機となったり，すでに活動中のエージェントの行動を変化させる契機となったりする。各エージェントは，イベントを自分の文脈で解釈する。たとえば，力－トルクセンサを監視しているエージェントが，力がある閾値を越えたことをイベントとしてポストしたとする。すると，たとえばロボットのアームの運動を支配するエージェントは，そのイベントにもとづいて運動のモードを変える。また，別のエージェントは，ユーザのディスプレイ・モニタに，力－トルクの状態変化を表示する。イベントは，メッセージによる通信のような相手を意識した通信ではなく，誰がそれを受け取るかわからない，不特定多数に対するアナウンスである。

イベントの発生を待つエージェントは，(i) そのイベントが発生するまでそこでブロックする，(ii) そのイベントが発生したときに，自分にある特定のメッセージが送られてくることを期待する，(iii) そのイベントが発生したときに自分のプロセスに割り込みが発生する，という三つのイベント待ちモードのいずれかを指定してイベント待ちを宣言する。イベントがポストされると，そのイベントを待っているすべてのエージェントに対してイベントの発生が伝達される。

4.2.5 行動インタプリタ

エージェントの**行動インタプリタ**は，そのエージェントが，他のエージェントや環境との相互作用にもとづいて自分のメソッドを組み合わせ，実行するのを管理することによって，そのエージェ

ント自身の行動の性格を決める[2]。行動インタプリタはメソッドの起動を制御するが，それ自身も **behavior** という名の通常のメソッドである。従って，**defmethod** マクロによってユーザが再定義できる。しかし，以下では，エージェントの標準的な行動様式として，デフォールトで与えている行動インタプリタについて述べることとし，混乱のおそれがないときはこれを単に行動インタプリタと呼ぶ。

デフォールトの行動インタプリタとして，人間の作業者が行なう仕事をヒントに，日常的な仕事(ルーティンワーク)と，必要が生じたときにだけ行なう仕事とを区別して実行するようなものを考える。この行動インタプリタは，通常は(もしあれば)ルーティンワークを実行しつづけているが，メッセージが到着したり，何らかのイベントが発生したりすると，ルーティンワークを中断してそれに対応する処理をする。依頼されたときにだけ実行される仕事を，特別ジョブと呼ぶ。この仕事は，主として他のエージェント等から依頼のメッセージを受信したときに，ルーティンワークを一時中断して実行する。ルーティンワークと特別ジョブの区別は，プログラミング上の便宜のためのものである。実際にエージェントベースのプログラミングを行なってみると，ルーティンワーク的な仕事を通常のジョブと区別して与えるのが便利である。ルーティンワークを持つエージェントは，いわゆるデーモンとかサーバなどと呼ばれるプロセスに類似している。

ルーティンワークもエージェントの一つのメソッドである。どのメソッドがルーティンワークであるかは，インスタンス変数 **routine-work** で指定される。行動インタプリタは，処理すべき他の仕事がなければこのインスタンス変数を参照して，ルーティンワークを実行する。

4.2.6 制御の対象としてのエージェント

エージェントは単なるデータでもあるので，自分自身で，あるいは隠蔽されていない情報であれば外部から，内部情報を参照したり操作したりすることができる。エージェントの動作に関する情報の一部やメソッドのレパートリ自体も操作可能なデータである。従って，こうしたデータを操作すればそのエージェントの振舞いを変更することができる。このような自己参照，あるいは他エージェントからの参照や制御といったメタ動作の機能は，システムを自己監視したり処理系を単純化したりするのに有効である。このとき，参照する側のエージェントは，自分のプロセスの下でその参照動作を行なう。被参照側のエージェントのプロセスはその参照には関知しない。これは，たとえば，他人に聴診器を当てて外部からその人の状態を覗くことに対応する。メソッドやその実行制御に関する変数等の一部も操作対象になるので，行動インタプリタやルーティンワークを変更したり，プロセスへの割り込みによって処理自体を強制的に一時停止したりすることも，エージェント相互で行なうことができる。

メタ制御においては，オブジェクトやエージェントに対してメタオブジェクトやメタインタプリタなどの概念を明確に言語仕様に採り入れるアプローチもある [51] が，ここでは，インタプリタの主要な機能がデータとして操作できる，というにとどめている。メタの概念は一般にわかりやすくなく，通常の自己反映的な操作も含め，ロボットプログラミングで陽に意識する必然性があまりないと考えたからである。しかし，メタプログラミングの能力自体は重要である。

---

[2]これをインタプリタと呼ぶのは，通常のプログラム言語のインタプリタが，入力された式と環境を解釈して計算という振舞いを出力することからのアナロジーで，このプログラムは，受信したメッセージや置かれた状況を解釈してエージェントの振舞いを出力するという意味で，一種のインタプリタであると考えられることによる。

4.3 インプリメンテーション

MARCS の実験系を, TAO/ELIS 上に実現した。TAO は, マルチプロセスをサポートしており, Lisp のレベルでプロセスを容易に扱うことができる [23]。実験システムでは, ELIS に小型の商用ロボットを接続し, シリアルラインを介してロボットにコマンドを送ったり, ロボットの状態を取得したりする。

4.3.1 並行プロセスとエージェントの表現

TAO では, プロセスオブジェクトを生成し, それに初期関数とその引数リストを与えることによって, ほかのプロセスとともに並行に走るプロセスをフォークすることができる。各エージェントにそれぞれ一つの TAO のプロセスを割り当て, その初期関数として行動インタプリタを起動する関数を与える。従って, 各エージェントは, 独立したプロセスのもとで並行に動作する。TAO のプロセスは, ELIS 上のメモリを共有し, 言語のレベルからも変数等を共有できる。MARCS において物理世界を記述するオブジェクトは, プロセス間で共有される大域変数に保持され, すべてのエージェントから参照される。エージェントは, 静的には TAO オブジェクトで表現され, そのクラスは **agent** というスーパークラスを持つ。エージェントは, 名前, **behavior** を走らせる TAO のプロセス, エージェント間交信のためのメールボックス等を持つ。

TAO のオブジェクト指向とマルチプロセスは別個の機能であり, オブジェクトが互いに別のプロセスとして振舞うわけではない。ここでは, TAO の意味でのメッセージを TAO メッセージと呼ぶ。TAO メッセージはプロセス間の通信ではない。これに対し, MARCS におけるエージェント間のメッセージは, プロセス間通信である。エージェント相互のメッセージをエージェントメッセージと呼ぶ。なお, TAO のプロセスは, 単一の共有メモリとしてのセル空間上で動作するので, プロセスというよりはスレッドに近い。

エージェントのメソッドは通常は行動インタプリタによって起動されるが, それ自身 TAO オブジェクトのメソッドでもあるから, TAO メッセージでも起動することができる。この場合, そのメソッドは, エージェントに付随するプロセスとは関係なく, メッセージ伝達式を評価しているプロセスのもとで実行される。これを用いると, エージェントのメソッドを, 別プロセスでなく自分のプロセスのもとでデバッグすることができる。このことは, プログラム開発上極めて有効である。

4.3.2 行動インタプリタ

エージェントには, デフォールトでは図 4.2に概略を示すような行動インタプリタの **behavior** が与えられ, 無限 **loop** でそれを起動する初期関数のプロセスがフォークされる。

この行動インタプリタは, メールボックスに投入されるメッセージで指定された名前を持つメソッドを検索し, その実行条件をチェックした上で実行する。メソッド本体は TAO のプログラムであるから, TAO のデータとしてのオブジェクトや関数を参照でき, 自分やほかのエージェントの内部状態や行動インタプリタの実行状態を監視したり変更したりすることができる。

生成直後のエージェントはルーティンワークも持たず, **in-coming-mailbox** に保持しているメールボックスの待ち状態になる。ルーティンワークを設定するには, インスタンス変数 **routine-work** に, そのエージェントの持つメソッド名のどれかを代入し, ルーティンワークの開始依頼メッセージを送る。

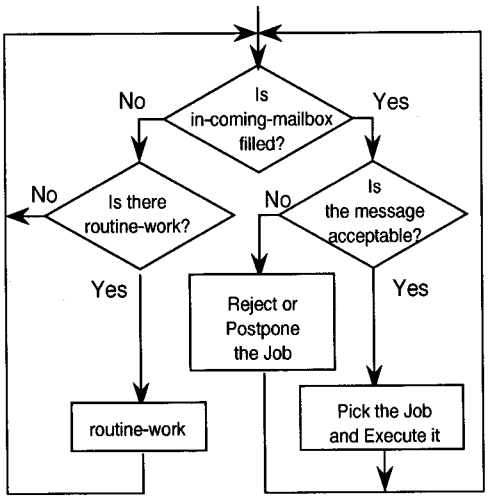


図 4.2: 行動インタプリタ

behavior を再定義してデフォールトでないインタプリタを与えれば，特殊な振舞いをするエージェントを活動させることができる。たとえば，行動インタプリタとして推論エンジンを与え，内部作業変数にルールの集合をもたせれば，ルールベースで動作するエージェントになる。また，現在の **behavior** を退避して別の行動インタプリタを設定することも，メソッド表を操作するメッセージ **change-behavior** によって可能である。たとえば，

```
(send-async taro 'change-behavior 'superman)
```

は，エージェント **taro** の現在の **behavior** メソッドを，それ自身が予め持っている **superman** メソッドに変更する。それまでの **behavior** メソッドは，**superman** になる。

エージェントのプログラミングにおいて，自分自身にメッセージを送信することが必要になることがある。たとえば，再帰的な定義のメソッドを書きたいときにこうしたことがおこる。しかし，並行オブジェクトのメッセージでは注意が必要である。明らかに，相手からの返事を待つためにプロセスがブロックしてしまうモードのメッセージ送信を，自分自身に送ることはできない[3]。

行動インタプリタ **behavior** で再帰を実現するため，自分自身へのメッセージ送信ではメッセージ・オブジェクトをメールボックスに投入する，ということをせず，**behavior** 自身にメソッド名と引数とを与えて，**behavior** を再帰的に呼び出すようにする。すなわち，マクロ

```
(defmacro self-exec
  (message-name message-args)
  `(apply-send self 'behavior
    (list ,message-name ,message-args) ))
```

の呼び出しを，自分自身へのメッセージ送信の代用とする[4]。

[3]これは，再帰のレベルが深まるときに，単に自分自身のサブルーティン・コールをするだけで，プロセスがブロックすることのない逐次型オブジェクトの再帰メッセージとは事情が異なる点である。

[4]なお，**behavior** を経由せず TAO メッセージでも自分のメソッドを起動できるが，そうするとそのエージェントの

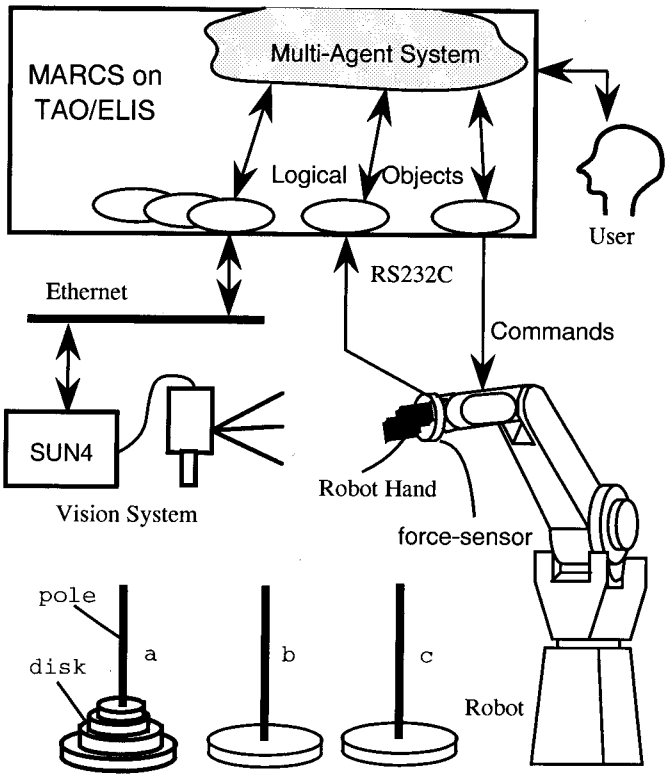


図 4.3: ハノイの塔による実験系

4.4 実験

4.4.1 ハノイの塔

多関節型のロボットを用いた簡単な組み立て問題の雛形として，よく知られたパズルであるハノイの塔を取り上げる。このパズルでは，中央に穴のあいた大きさの異なる円盤を，三つのポール間で移動する。ポールの場所や円盤の位置を検知しながら円盤を移動するのがロボットの作業である。これを，MARCS のマルチエージェントの仕組みを用いて作業記述のモジュール化や作業戦略のインクリメンタルな変更・高機能化のためのプログラミング実験を行ない，実際にロボットを動かした。図 4.3に，この実験系の構成を示す。

この実験系では，3 本のポール (**a**, **b**, **c** とする) の 2 次元的な位置データを，CCD カメラとワークステーション上の画像処理システムからなる視覚センサ系で取得する。初期状態では，図 4.3に示すように，たとえば **a** に 3 枚の円盤が大きさの順に積み重ねてある。ロボットは，この円盤を 1 枚ずつ別のポールに移す。このとき，途中で積み重ねられる円盤の大きさの順が逆転してはならない。この作業は，あるポール X の一番上にある円盤を別のポール Y に移す手続き (**move-disk** と

---

behavior によるメッセージ処理のセマンティクスが維持できなくなる。たとえば，**behavior** の管理下にあるメッセージの受理条件のチェックなどは実行されなくなる。また，自分自身の中で再帰的に処理を続けている間，他のメッセージの処理などを (自分で陽に行なわないかぎり) 行なえなくなる。

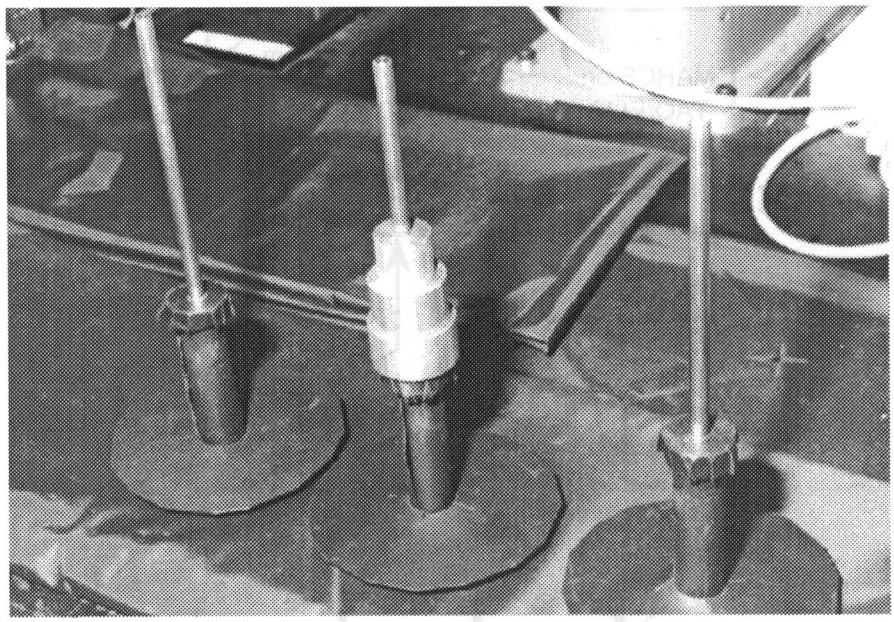


図 4.4: ハノイの塔

する) を用いて, よく知られた再帰的プログラムで記述できる。実験で用いたハノイの塔の写真を図 4.4に示す。また, この系でロボットが実際に作業をしている様子を図 4.5に示す。

以下では, **move-disk** に注目し, この手続きを詳しく論ずる。**move-disk** を遂行するために, 視覚センサは, 3 本のポールを上から見おろし, それらの位置の 2 次元情報 (x 座標と y 座標) を取得する。また, 円盤の z 座標は, ロボット自身がアームをゆっくりと下に降ろしていき, ハンドの先端が円盤に触れて, 手首に取り付けられた力 - トルクセンサが上向きの反力を検知することによって知ることとする。

手続き **move-disk** は, 以下のように記述できる。

(1) ハンドを出発ポール X の真上に持ってくる。

(2) 最上位の円盤に達するまでハンドを下げていく。

(3) グリッパを開く。

(4) ハンドを, 円盤の厚み分だけ下げる。

(5) グリッパを閉じる (円盤をつかむ)。

(6) ハンドを上昇させ, 円盤をポール X から抜き取る。

(7) ハンドを目標ポール Y の真上に持ってくる。

(8) ハンドをポール Y の最上位の円盤か台に達するまで下げる (挿入)。

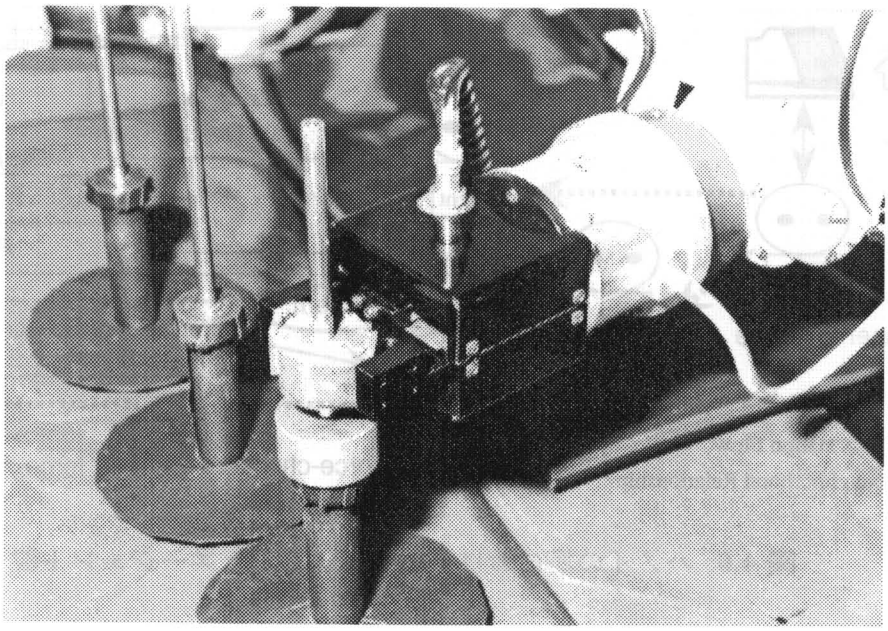


図 4.5: ハノイの塔の作業

(9) グリッパを開く (円盤を離す)。

(10) ハンドを上に動かしてポール Y から離す。

4.4.2 ハノイの物理世界の表現

まず, 物理世界の物体を TAO のオブジェクトで記述する。たとえばポールは, 2 次元的な位置と高さとを持つ,

```
(defclass pole () (x y height) () :gettable )
```

というクラスで定義する。実際に, ポール a というオブジェクトを作るには,

```
(make-object a (make-instance 'pole))
```

とする。ポール b, c も, 同様である。マクロ **make-object** は, オブジェクトを世界記述として登録する。登録されたオブジェクトは, 各エージェントのメソッドの中で名前で参照できる。ロボットの手首に取り付けられた力 - トルクセンサは, 6 つのインスタンス変数を持つ **force-sensor** オブジェクトとして定義される。ロボット自体もオブジェクトである。オブジェクト **robot** は, 動作コマンドや状態の問い合わせに対応する一群の TAO メッセージを受けつける。ロボットに動作コマンドの TAO メッセージを送ると, マニピュレータが実際に運動を起こす。

4.4.3 機能ベースエージェント

これらのオブジェクトを定義したのち, システムを構成するセンサやアームの駆動系などの各機能に対応する, 次のようなエージェント群を生成する。

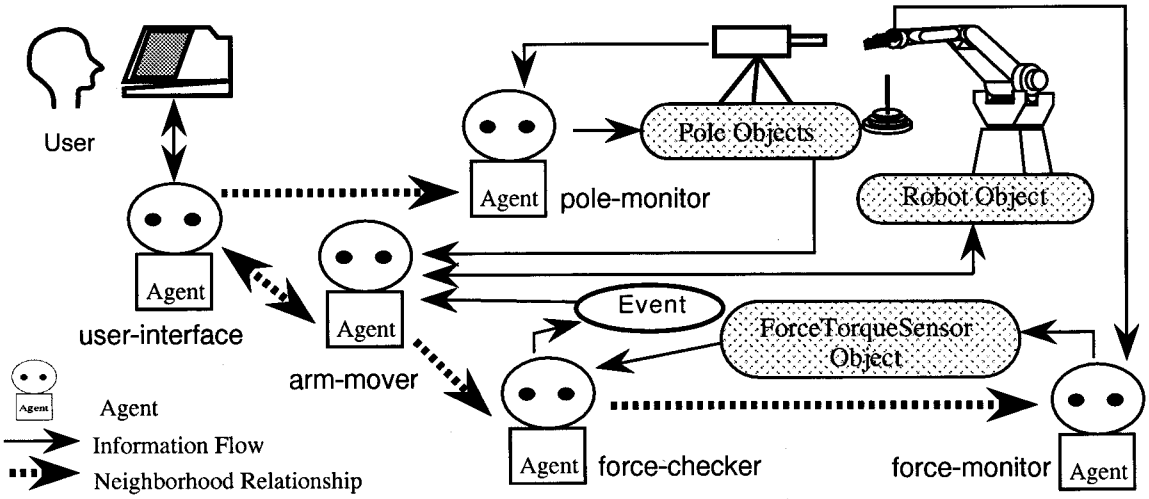


図 4.6: ハノイの塔におけるオブジェクトとエージェント群

- **force-monitor** は, 力－トルクセンサを監視し, オブジェクト **force-sensor** のインスタンス変数を更新する。

- **force-checker** は, **force-sensor** オブジェクトの値を監視し, **excess-threshold** というイベントの発生条件をチェックする。条件が成立したらそのイベントをポストする。

- **pole-monitor** は, ポールの位置を常時監視し, オブジェクト **a**, **b**, **c** の値を更新する。ワークステーション上の画像処理システムとは TCP/IP プロトコルで通信する。

- **arm-mover** は, オブジェクト **robot** を直接駆動する。このエージェントは, ハノイの塔を解く再帰プログラムを解釈し, ロボットへのコマンド列を生成した上, オブジェクト **robot** に TAO メッセージとして送出する。

- **user-interface** は, 人間のユーザとマルチエージェント系との間を仲介する。

これらのエージェントと, その近傍関係, オブジェクト群を図 4.6に示す。ハンドをポールの真上に位置決めするために参照される **pole** オブジェクトの情報は **pole-monitor** によって常時更新されている。また, **arm-mover** は, 円盤の z 方向の位置を知るためにハンドを下向きに動かしている間, **excess-threshold** イベントを意識している。もしそれが検出されると, **arm-mover** は, ハンドが台かまたは最上位の円盤に達したことを知って, ハンドの動きを止める。ロボットがエラーを起こすなどの異常事態が発生したときには, **arm-mover** は **user-interface** と通信を行ない, 人間のユーザに, どのようにすべきかを尋ねたりする。すると, ユーザは, このエージェントを介して異常状態から手動で抜け出すといったことができる。エージェントは独立しており, たとえ他のエージェントが動作中であっても変更したり置き換えたりすることができる。たとえば, **force-checker** の閾値の条件などをロボットの動作中に変更したりすることができる。

ここで定義したエージェントは，それぞれがロボットの部分機能に対応する，いわゆる機能ベースエージェントである。機能ベースでエージェントを定義することは，ロボットの物理的なモジュールとエージェントとの対応がよいため，システムのモデル化の手段としてごく自然である。

エージェントの具体的なプログラムを arm-mover について例示する。arm-mover は agent のサブクラスとして，

```
(defclass arm-mover ()
  ((force-sensor (object 'force-sensor)))
  (agent manipulator-commander-mixin)
  :gettable )
```

のように定義される。このエージェントは，

```
(make-agent arm-mover)
```

で実際に生成され，プロセスが起動されて活動状態になる。ハノイの塔のトップレベルの実行プログラムは，メソッド定義

```
(defmethod (arm-mover hanoi) (n a b c)
   (cond ([n = 1] (self-exec move-disk a b))
         (t (self-exec hanoi [n - 1] a c b)
            (self-exec move-disk a b)
            (self-exec hanoi [n - 1] c b a) )))
```

である。ここで，n は円盤の枚数，a, b, c はポールを表わすオブジェクトである。このメソッドは，先に述べた self-exec による再帰的メッセージ送信の例になっている。メソッド hanoi は，より単機能のメソッドである get-disk および put-disk を呼ぶ move-disk

```
(defmethod (arm-mover move-disk) (pole1 pole2)
   (self-exec get-disk pole1)
   (self exec put-disk pole2) )
```

によって記述されている。get-disk と put-disk は，力センサをモニタする force-monitor エージェントに依頼して閾値のイベントを頼りに円盤の移動を行なう。これらは，arm-mover エージェントのメソッドである。put-disk メソッド (図 4.7) は円盤を挿入する手続きで，単純にハンドを降下させていき，下端に達したことを force-monitor の発生するイベントによって検知するナイーブなプログラムである。

4.4.4 行動ベースエージェント

ハノイの塔の例題において，図 4.7のようなナイーブなプログラムでは視覚センサの精度が不十分な場合，円盤をポールに挿入することは大変困難になる。実際，この実験系では視覚センサの精度が 1.5mm 程度であるのに加え，ロボットの位置決め精度もせいぜい 1mm 程度である。このため，円盤の挿入は，円盤の穴の径とポールの径との間のクリアランスが小さい (実験では 0.8mm) 場合，計測された情報だけに頼って遂行することは，殆ど不可能になる。図 4.8に，円盤挿入時のハンド部分の状態を模式的に示す。

```
;;; put-disk メソッドは，ロボットハンドが持っている円盤を，目標ポールに
;;; 挿入する。このとき，挿入作業の終了は，ポールの下端にハンドが到達した
;;; ときの反力の発生で判断する。
(defmethod (arm-mover put-disk) (pole)
    ; まず，目標ポールの真上にハンドを持ってくる。
    (self-exec move-above pole 'close)
    ; 次に，反力検出の閾値設定のため，現在の力センサ (y軸まわりの
    ; モーメント my) の出力値の安定性をにチェックする。
    (let ((x [force-sensor my]))
      (loop (send-async force-monitor 'update-data)
            (sleep 0.1)
            (:while (eq x [force-sensor my])) ))
    ; 力モニタエージェント force-monitor に閾値の監視を依頼する。
    (send-sync force-monitor
          'set-routine-work
          'simple-thresh-check
          t
          (list (- [force-sensor my] 30)) )
    ; 反力を意識しつつハンドを下げ，円盤を挿入する。
    [self with-event-accept
          'force
          'move-hand-downward ]
    ; グリッパを開いて円盤を放す。
    (self-exec command 'go)
    (send-async force-monitor 'suspend-routine-work)
    (self-exec move-vert 20)
    (self-exec move-above pole 'open)
    (inc [pole disks]) )
```

図 4.7: **put-disk** メソッド

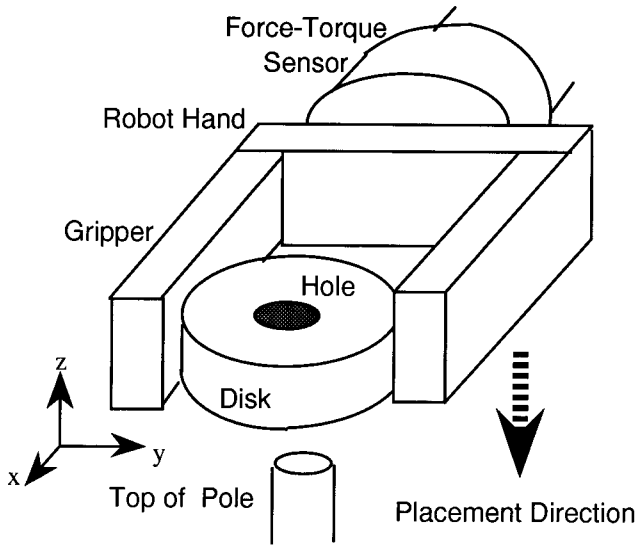


図 4.8: 挿入作業におけるハンド部分の模式図

このような場合に人間のとる戦略を観察すると, 円盤の穴をポールの上のだいたいの位置に置き, 穴にはいるまで円盤を不規則に動かして, 試行錯誤的な位置合わせを試みるのが普通である。そこで, これと同様の解決法をロボットにも採用する。この戦略は, 次の 2 つの意図をその要素として含むと考えられる。

(1) 円盤を下向きに動かそうとする意図, および

(2) 円盤をグリグリと動かしてみようとする意図

(1) は, 挿入という作業目的を比較的はっきりと反映しているが, (2) は, そうした目的からは少し離れている。(2) の意図による行動は, (1) の意図の目指す目的を遂行するために必要な条件を整えるためにとられる。これら 2 つの意図要素にもとづく行動を, それぞれ MARCS におけるエージェントの行動に移しかえる。

まず, 専らアームを降下させる役割を担うエージェント **move-down** を導入して **put-disk** のアーム降下部分を置き換える。**move-down** は, 位置決めに失敗して反力が発生したら降下を中断する。次に, 反力の検出によって **move-down** がアームの降下を中断したら, それに代わってアームを試行錯誤的に動かし, 正しいポールの位置を探索するようなエージェント **joggle** を導入する。これらのエージェントは, **arm-mover** の **put-disk** の実行を乗っ取り, 位置決め精度が不足している場合でも, 協力して円盤の挿入成功のために活動する。

Brooks は, 行動という観点からタスクを分割する**包摂アーキテクチャ**[5]という興味深い提案をし

[5]Subsumption Architecture: センサからの入力に対して振舞いが直接的に出力される反射的な**行動**モジュールからなる階層としてロボットの制御システムを構成するアーキテクチャ。下位の低レベルの反射モジュールは, 上位のより高度なモジュールに**包摂** (subsume) される。ある意味でモデルを持っておらず, モデル生成のためのセンサ情報処理や, いわゆる AI 的な行動計画をしない。従って, 素早く動き, 構造も単純である。Brooks は, このアイデアを従来型の AI に対するアンチテーゼとして, 移動ロボットにおいて提案した。

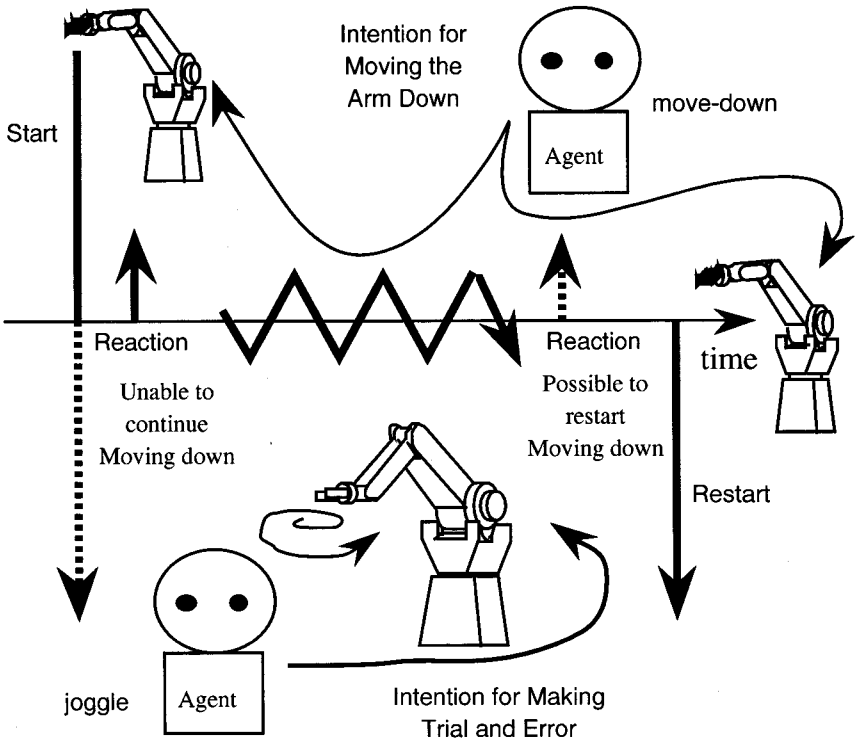


図 4.9: 行動ベースエージェント: **move-down** と **joggle**

ている [18]。上記の円盤挿入タスクにおける 2 つのエージェントの役割は, 包摂アーキテクチャで言う行動の階層にほぼ対応している。**move-down** と **joggle** は, 力 - トルクセンサによる反力イベントを入力として直接にロボットの行動を出力する行動ベースエージェントである。図 4.9 に, これらのエージェントとその動作を示す。各エージェントの役割は, 詳しくは次のとおりである。

- **joggle** は, 下位の行動として, ロボットのハンドをグリグリ動かしつづける。これは, 作業目標を直接には反映しない反射的な行動である。動かし方の厳密な指定は重要ではなく, いつかは円盤がポールにはいることを期待できる範囲で, プログラマによって与えられる。また, 上位の行動によって抑止されうる。

- **move-down** は, 上位の行動として, ロボットの作業目的を知っている。上向きの反力が検出されない限り, このエージェントは下位の **joggle** エージェントの働きを抑止して, ロボットのハンドを下向きに動かそうとする。

これら 2 つのエージェントが活動状態になると, 円盤を手にしたロボットは, まず **move-down** によって腕を下に動かし始める。その間, **joggle** からロボットへのコマンド出力は, **move-down** に抑止されている。もし円盤がポールの先端に当たって上向きの反力が検出されると **move-down** は自分のコマンド出力を止め, **joggle** の出力を許可する。**move-down** からの抑止が解かれた **joggle** は, ロボットへコマンドを出力して, ハンドを動かし始める。実験ではこの運動として, 水平方向の螺旋状の動きを与えた。この運動の間, アームと, 力 - トルクセンサに内在する微小なたわみに

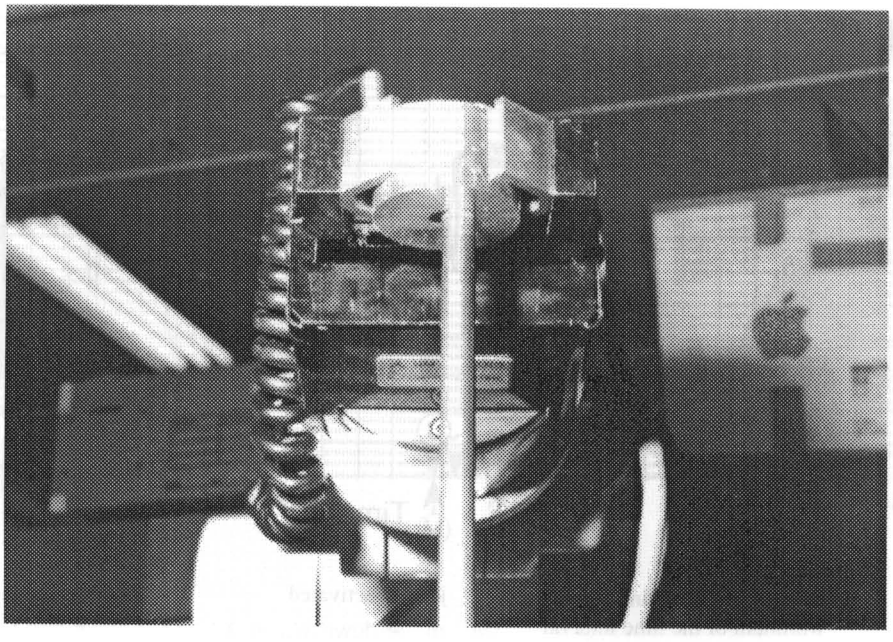


図 4.10: **Joggle** による試行錯誤

よって，一定以上の垂直方向の反力がアームにかかりつづけている。このとき，たまたま円盤の穴とポールの位置が一致すると，円盤はわずかにポールの中に入り，上向きの反力が消失する。その結果 **move-down** が活動を再開して **joggle** を抑止し，コマンド送出を始める。こうして **move-down** はハンドを下向きに動かし，最終的に円盤を挿入する。この間，反力は **force-checker** によって監視され，**joggle** と **move-down** との間の主導権の交代は，**force-checker** がポストするイベントによって制御される。図 4.10 は，この戦略によって **joggle** がアームを試行錯誤的に動かし，円盤の位置決めを試みているようすを示す。

move-down と **joggle** の導入は，もとの **arm-mover** の持っていた **put-disk** メソッドの働きの一部を置き換えるだけであり，それ以外の既存のプログラムには何の変更もない。これは，インクリメンタルなプログラミングによってロボットの行動を知能化する簡単な例である。

図 4.11 は，実際の作業履歴の例 (円盤 2 枚の作業とした) である。力センサ出力 (反力として y 軸回りのモーメントを監視する) の各時点の小期間 (1 秒) ごとの最大最小値の履歴と，行動ベースエージェントの活性化時点を示す。センサ出力 -80 付近で変動している三つの谷が **joggle** による試行錯誤の期間である。それらの期間の始めで反力が変動し，**joggle** と **move-down** 間での主導権が一時的に何度か交代する現象が見られるが，**move-down** が主導権をとると反力が増大するので，結局は **joggle** が主導権を奪い返している。グラフの途切れている部分は，**force-monitor** が休止している期間である。

実際にロボットを動かした結果，この行動ベースエージェント群によってプログラムされた系では，視覚センサやロボット・アームの制御が十分な精度を持たなくても，手探りによってポールの位置が発見され，高い信頼度で円盤を挿入できた。また，この作業では，試行錯誤法はより一般性を発揮する。たとえば，挿入に成功したあと円盤を下に降ろしていく途中で，円盤がポールに食いつ

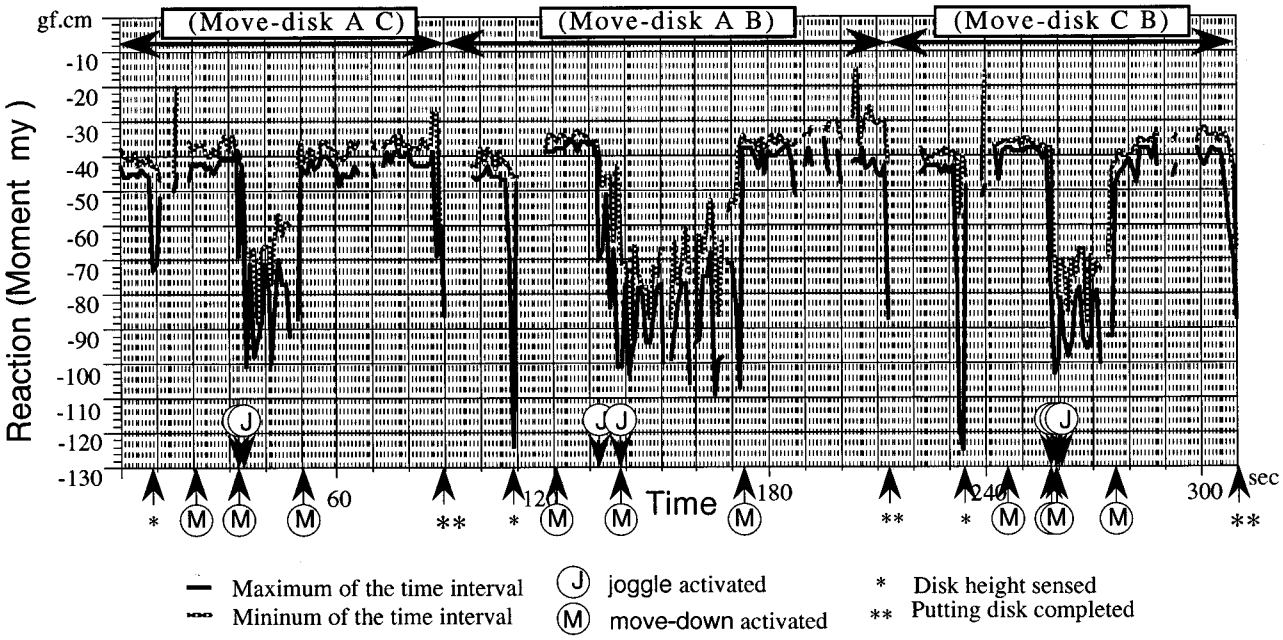


図 4.11: ハノイの塔のタスクの実行履歴の例(2 枚の円盤をポール A からポール B へ移す)

きを起こすことがある。この場合にも上向きの反力が検出され, **joggle** エージェントが活性化されてアームを動かし, 食いつきを解消する。

この実験では, 組み立て作業に手探り行動を導入する目的で包摂アーキテクチャを適用した。マニピュレータを包摂アーキテクチャで制御する試みは幾つかある [52] [53] が, ここでの **move-down** と **joggle** のような, 組み立て作業での微妙な制御の目的に効果的に用いた例はあまりない。また, 目的作業とそのための条件を整える戦略とがエージェントによって独立して与えられている点が重要である。すなわち, エージェントによる作業戦略のモジュール化は, オブジェクト指向の新しい応用になっていると言うことができる。

4.5 エージェントの監視・制御

4.2.6で述べたように, MARCS ではエージェントを単なるオブジェクトと見てそのインスタンス変数を参照したり, エージェントの行動の基本単位であるメソッドを外部から操作したりすることができる。TAO のプロセス割り込みと **catch-throw** を用いて, エージェントのプログラム実行の流れを外部から制御することも容易である。

システム全体がどのような計算状態にあるかをリアルタイムで監視し, 必要に応じてそれを修正したり停止したり再実行したりすることが自由に行なえることは, プログラミングやロボットの操作環境として極めて有効である。マルチエージェントの枠組みでは, ロボットの実行制御やセンサの制御を司るエージェントを, 別のエージェントで監視・制御することによってこうしたことを実現できる。これは, ロボット全体として見れば一種の内観能力とも言え, 自律性の高い知能ロボットにおいて重要であることが指摘されている [54]。図 4.12に, MARCS におけるロボットの実制御

とその実行状況の監視系の概念を示した。

4.5.1 行動ベースエージェント間の相互作用

4.4.4で述べた行動ベースエージェント間の相互作用を実現するために，MARCSでは，特定の運動をアームに与えるコマンドを出力しているエージェントを別のエージェントが監視し，そこでトラップして別の行動を起こさせる，といった機能を用いた。包摂アーキテクチャでは，ここで一つのエージェントとして実現した行動は，いくつかのAFSM[6]の結合によって構成され，行動間の相互作用は，AFSM単位での抑制や情報の注入などによって実現されている [55]。これに対し，MARCSでは，行動インタプリタが実行するプログラムの変数束縛の環境や実行の継続の一部を，外部のエージェントから操作することによって相互作用を行なっている。ハノイの塔における行動ベースエージェントは，自分がロボットへのコマンド出力ができるかどうかを示すフラグ変数を持っており，この変数を上位のエージェントが外から操作する。

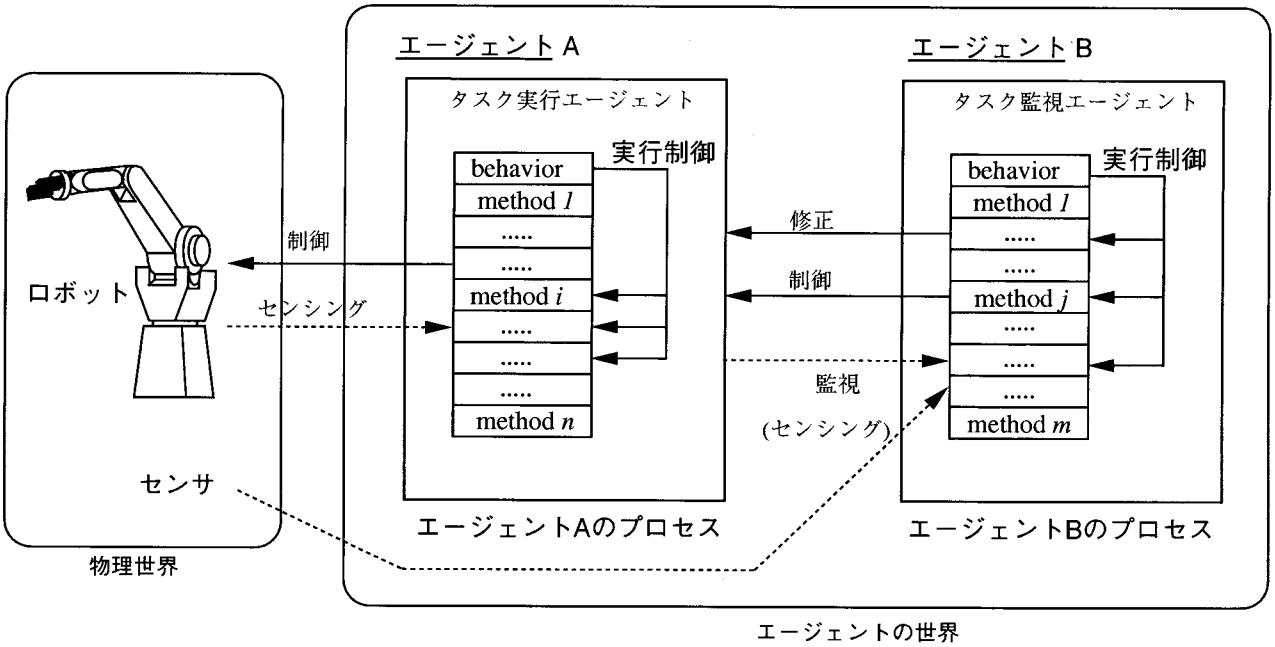


図 4.12: エージェントによるエージェントの監視・制御

4.5.2 X ウィンドウによる状態表示

エージェントを監視するエージェントの別の応用として，X ウィンドウを用いたユーザインタフェースのシステムについて述べる。この応用では，ロボット作業を遂行するエージェント群を監視し，その状態を表示する X ウィンドウのクライアントを作成した。このクライアント自身もエージェント (**x-manager** と呼ぶ) である。**x-manager** は，X ウィンドウのサーバが発生する種々の X のイベントを取り込んで処理し，エージェントの状態やオブジェクトの状態を表示するプログラムをルーティンワークとしている。また，X ウィンドウを介したユーザからのコマンドにより，各

[6]Augmented Finite State Machine の略。

エージェントにメッセージを送ったり割り込んだりして，エージェントの世界を制御する。このXクライアントは，TAOの上に実現されたオブジェクト指向によるXクライアント・ライブラリであるNueX [56] を用いている。図4.13に，**x-manager** によるマルチエージェント系の監視・制御の概念を示す。

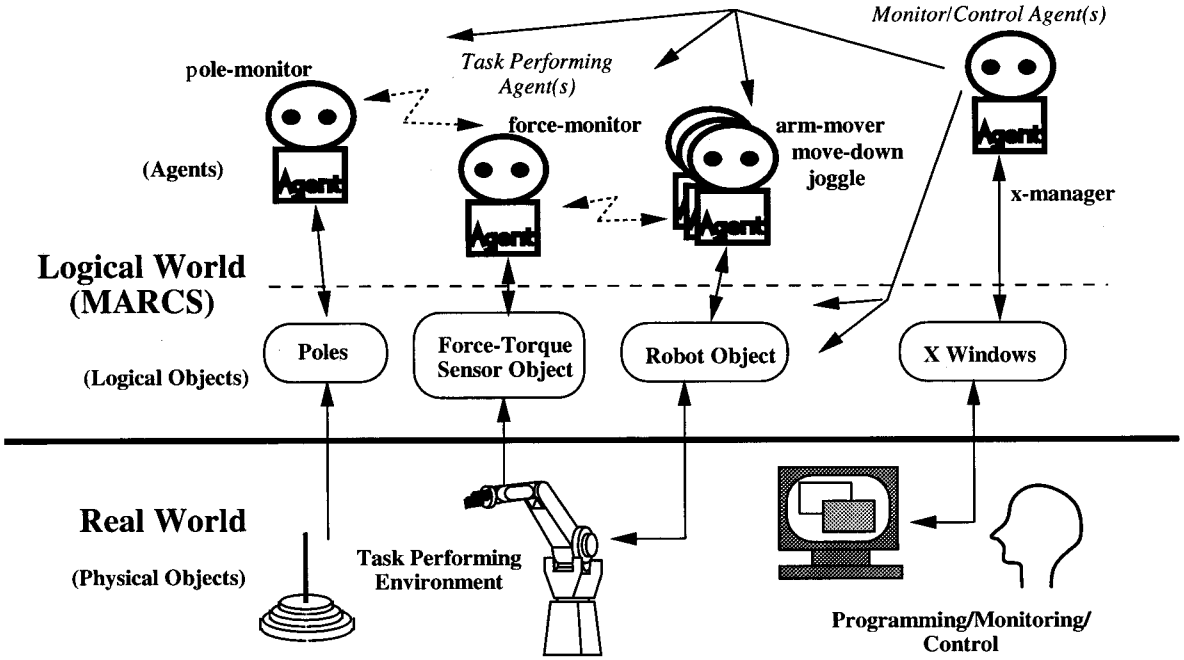


図 4.13: Xウィンドウを用いたエージェント系の監視・制御

図4.14に，ハノイの塔のタスクを実行しているときの，**x-manager** によるXウィンドウの表示例を示す。Xウィンドウのサーバを UNIX ワークステーション上で起動し，TAO/ELIS 上で稼動する MARCS の状態を示している。この図に示したウィンドウには，オブジェクトの状態やエージェントの動作状態をリアルタイムで表示している。MARCS はマルチプログラミング環境であるので，エージェントやオブジェクトの振舞いを視覚化することは，ユーザとシステムとの情報交換にきわめて有効である。

x-manager は，エージェント群全体をモニタしながらXのサーバに情報を送信することが主たる役割である。従って，他のエージェントからメッセージを受信することはないものとすることもできる。この場合には，先に述べたルーティンワークを通常の行動インタプリタの管理のもとで実行するのでなく，**change-behavior** によって，それ自身を行動インタプリタとしてしまうことができる。

4.6 結論

本章では，実世界を表現するモデルとして，TAO のオブジェクトに並行プロセスとしての性格を与えた，エージェントをベースとするロボットプログラミングを提案した。また，その実験システムを用いて多関節マニピュレータによる作業を行なった結果を述べた。

ロボットの知能化は知能処理研究の中でも重要な分野であるが，ロボット自体が極めて複合的な

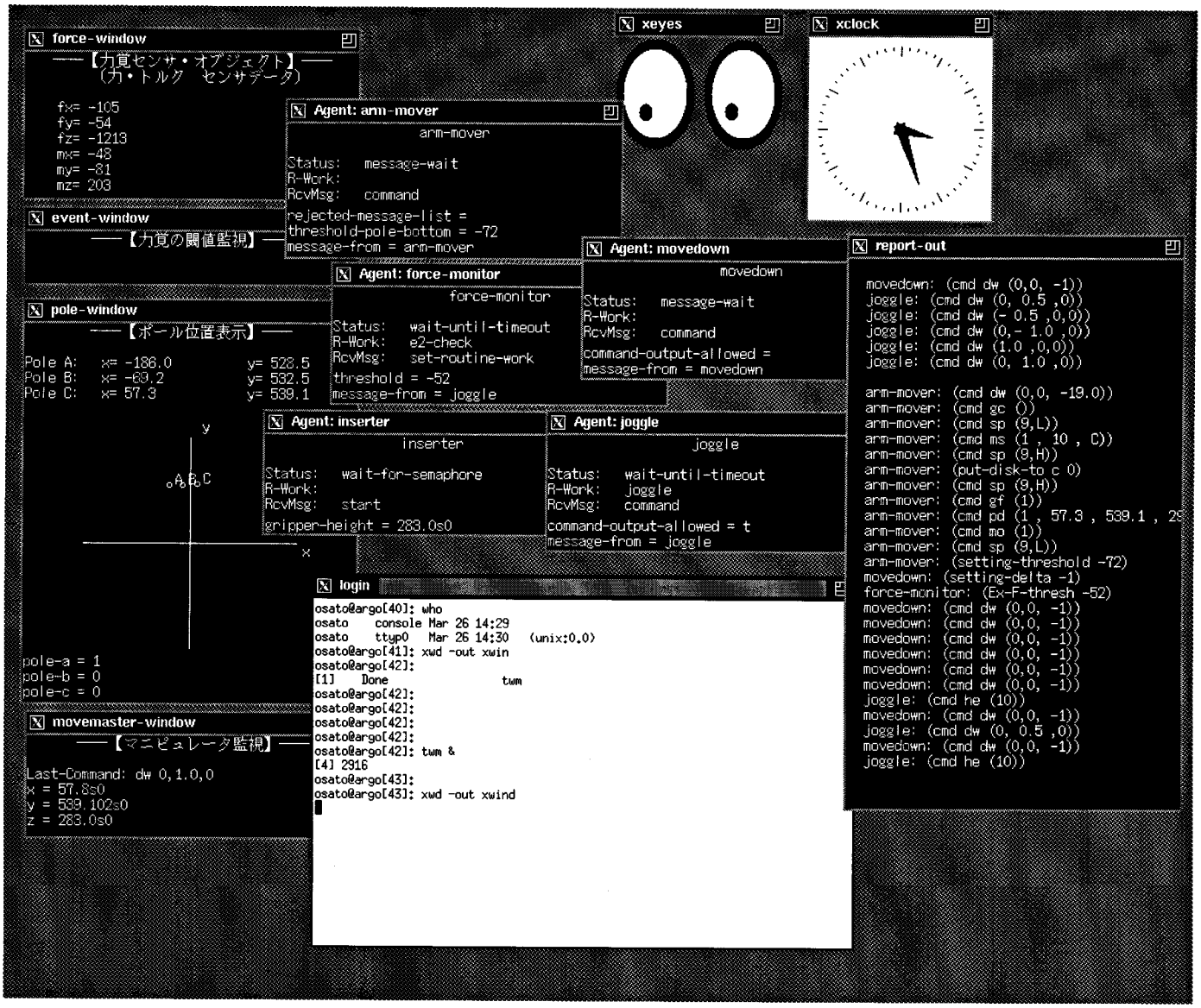


図 4.14: X ウィンドウによる表示例

システムであるため要素技術の研究に重点が置かれ，最新のソフトウェア技術を適用してシステム全体を統合する研究は立ち遅れている感がある。本研究は，オブジェクト指向並行プログラミングを適用して，ロボットの知能化をプログラミングの視点から支援しようとするものである。

マルチエージェントによるプログラミング実験を通じて，ロボットの仕事を並列オブジェクト指向システム上で実現することが有効であることが確認できた。エージェントによるロボットの作業プログラミングは，エージェントの独立性によるモジュラー性の高さと，エージェントの相互作用による遂行作業の高度化という二つの利点を持っている。プログラマは，ロボット作業の構造を分析し，それを複数のエージェントが役割分担する仕事として定義する。エージェント間の通信を抽象度の高いレベルで行なうことによって通信量を抑え，ロボット作業遂行システムの高いモジュール性と再利用性を確保することができる。本研究では，ロボットの作業世界の個々の要素の機能を代表する機能ベースエージェントに加えて行動ベースエージェントを導入し，遂行すべき作業が困難性を増すにつれて新たなエージェントがそれを解決するために投入される，という枠組みを与えた。この方式によってロボットに試行錯誤的な動きをさせ，簡単な組み立て問題の雛形を例題として信頼性の高い作業遂行を実現した。

さらに，エージェント自体を単なる受動的なオブジェクトと見て操作し，エージェント相互の監視・制御を行なうことを提案した。これにもとづいて，エージェント系の動作状態をXウィンドウを介して表示するシステムを試作し，その有効性を示した。ロボットシステムが知能的であると言えるためには，ロボット自身に自己の行動を評価しながら行動できる能力が備わっている必要がある。こうしたロボット自身の内観能力に相当する概念は，ソフトウェアの研究分野でもリフレクションなどの概念で関心が持たれており，近年研究の機運が盛り上がってきている。ロボットでは，これを作業信頼性向上の基礎的問題として考えていくことが必要であると考えられる。マルチエージェントによるシステム構成法は，ロボット内部に存在する独立なエージェントの相互作用を用いてこのような自己評価能力を持つロボットを実現しやすい枠組みである。

将来，遂行する作業が高度化するに伴って，ロボットは多様なハード要素，ソフト要素の複合体になるであろう。こうした複雑なシステムを効率的かつ容易に制御するには，それに応じたプログラミングシステムが必要であり，マルチエージェントの概念は，そのための単純で強力なモデルとして有力である。

本研究で実現したシステムは，動的能力を重視したインタプリタ型のシステムであるため，処理速度上の問題がある。ここでは，エージェントが比較的高次の役割を担うことを想定し，処理速度が直接に問題になる低レベルの運動制御や知覚機能は別のハードウェア・モジュールによって構成することを念頭に置いて，リアルタイム性には当面目をつぶっている。実際，小型のマニピュレータなどの運動能力に対しては，実験システムにおいても速度の問題は殆どない。しかし，リアルタイムの要求を満たすシステムは将来的な研究課題である。

本研究で述べた実験系は，共有メモリを持つマルチプロセスのLispシステムであるTAO/ELIS上に構築したプロトタイプであるが，負荷の軽減や信頼性の向上など，分散システムの持つ利点を享受するには，通信のコストが重要な要素となるような物理的な分散環境に拡張していく必要がある。これも将来の研究課題である。

第 5 章

結論

本研究で得られた成果と今後の研究課題をまとめる。

第 2 章では, 本論文の中心テーマであるオブジェクト指向のベースとなった Lisp マシン ELIS と言語 TAO について, 設計思想・実現技法・性能の概要を述べた。 ELIS は, 言語処理系を全面的にマイクロプログラムで記述して専用ハードウェアを制御することにより, 高性能を実現した。ハードウェアに近いマイクロプログラムで抽象度の高い言語処理系を直接記述する困難を克服するため, ELIS ではマイクロプログラマから見えるアーキテクチャを単純化し, また, 高機能のマイクロプログラム開発支援系を作成した。この支援系では, リンカによる WCS アドレスの自動割り付け機能が特に有効であった。また, ELIS のハードウェアを会話的に制御できる環境も TAO における大規模なマイクロプログラムのデバッグにおいて有効であった。

TAO の処理系の実現にあたっては, 優れたプログラミング環境の第一の要件は速度性能であるという観点から Lisp 処理系としての性能を重視した。その一方, マイクロプログラムによる実現上, Lisp の性能を犠牲にしない範囲においてできる限りの機能を盛り込む, という姿勢をとった。また, 実行時にもプログラムの字面の情報をできるだけ保持することが, デバッグにも有効であり, プログラミング環境として重要であるという観点から, インタプリタの性能に重きを置いた。

TAO の高速化には, ハードウェアでサポートされたタグを活用したこと, ハードウェア・スタックによって言語処理に伴う情報管理を行なったこと, およびリスト操作を支援する強力なメモリ・インタフェース等が大きく貢献した。

TAO では, ユーザの多様な要求に言語もまた対応すべきである, という基本思想を据え, プログラミング・パラダイムの選択をユーザの自由に任せることができるシステムを目指した。すなわち, 実用レベルの複合プログラミング・パラダイム言語の実現を目指し, Lisp 本来の逐次型パラダイムに, 論理型, オブジェクト指向といったパラダイムを, 処理系の核の部分でバランスよく融合した。これによってプログラミングの自由度は極めて大きくなり, 様々な実用的な応用プログラムが TAO 上で実現された。 TAO における複合プログラミング・パラダイムの有効性は実使用経験を通じて裏付けられていると考えられるが, 言語にどのようなプログラミング・パラダイムをどのような実現法で融合していくのがよいかは, 現在の TAO でもなお結論を下すことはできない。複数のプログラミング・パラダイムを融合する問題は, 今後も引き続き研究課題である。 TAO はそのような研究のための基盤を与えており, またその実績も積まれている。

TAO のインタプリタの速度性能は, ELIS のハードウェア性能を背景として, 他のシステムのコンパイラに匹敵する。また, 複合プログラミング・パラダイムのシステムとして重要な, パラダイム間の性能の比率もまた, インタプリタにおいて当初目標とした許容範囲に収まっている。こうし

た意味で, TAO の速度性能上の目標はほぼ達成された。

第3章では, TAO における複合プログラミング・パラダイムの一つの柱であるオブジェクト指向の機能とその実現法を述べた。TAO では, Lisp の言語仕様を拡張し, 処理系の基本機構としてオブジェクト指向を融合し, 実現法を工夫して大規模な応用に耐えるシステムとして提供した。

まず言語仕様の面では, メッセージ伝達式を Lisp プログラムの中に埋め込み, Lisp の基本データをオブジェクトとして扱う一方, ユーザ定義のオブジェクトを Lisp のデータで表現した。速度性能の観点では, (i) メッセージ伝達式の解釈部, (ii) メソッドの探索と起動, (iii) メソッド本体の実行部の三つの要因がある。TAO では, ELIS のマイクロプログラミングによって, (i) 式の解釈部を `eval` の中に埋め込み, (ii) メソッド探索をバイナリ・サーチで実現した。その結果, メソッド本体に至るまでの解釈のステップが高速化できた。また, (iii) メソッドの実行自体はインスタンス変数の参照以外, 関数の実行とまったく同一であるため, メソッド固有のオーバヘッドはない。インスタンス変数はハッシュ表によって高速化することができた。これらの手法の結果達成した性能を簡単なベンチマークを用いて評価し, インタプリタにおいて TAO のオブジェクト指向が, Lisp と比較して遜色のないものであることを示した。

TAO では, 性能にかかわる部分での動的なクラス階層辿りを極力避ける実現法をとった。これに伴う大規模な応用プログラムでのメモリ使用効率の悪化を改善するため, メソッドの内部表現の共有化や徹底したオン・ディマンドによる内部データの構築を行なった。その結果, システムは必要最小限の処理時間とメモリ使用量で稼動するようになり, プログラム開発時のターン・アラウンドを十分実用に耐えるレベルにすることができた。また, こうした実現法の妥当性を実際の応用プログラムを用いて評価した。

このように, 本研究では, Lisp マシンのハードウェア性能を背景に, 大規模な応用プログラムの開発にも耐えうる性能をもつオブジェクト指向を Lisp に融合し, 複合プログラミング・パラダイムの実現性と実用性を明らかにした。

しかし, TAO で採用したいくつかの機能, たとえば, 継承における情報の競合を解消する手段としてのメソッド結合が必ずしも使いやすくなく, プログラムの了解性を阻害するなど, 継承機能の提供のしかたの中には, 今後の課題として解決すべき問題もある。既存のクラスを部品として使うとき, そのインプリメンテーションを知っていなければならない点もプログラムの設計やデバッグにおいて問題となる。

また, TAO の現在の実現法では, クラスの構造が実行時に変化するような動的なオブジェクト指向システムを効率よく実現するのは難しい。動的なオブジェクト指向は, 今後重要性が増すと考えられるが, こうした目的には, 別の実現法を考えなければならないであろう。こうした概念の再吟味や新しい機能の摸索もまた今後の課題である。

第4章では, TAO のオブジェクト指向に並行処理の機能を付加し, 知能ロボットの高次制御に応用する問題を議論した。ロボットは, 物理世界とのさまざまなインタフェースや遂行する作業の多様性に応じて, 異質な構成要素からなる複合システムとなり, それを制御するソフトウェアも複合的なものとなる。本研究では, ロボット作業をプログラミングの視点で分解し, その構成要素の処理を, エージェントとよぶ特別なプログラミング上の論理的実体の担当すべき役割として, 陽に定義し割り当てるようなモデルを与えた。

マルチエージェントによるプログラミング実験を通じて, ロボットの仕事を並行オブジェクト指向システム上で実現することが有効であることを示した。エージェントによるロボットの作業プログラミングには, モジュラー性の高さと遂行作業の高度化等の利点がある。本研究では, 機能ベー

スエージェントに加えて行動ベースエージェントを導入し，ロボットの遂行すべき作業が困難性を増すにつれて新たなエージェントがそれを解決するために投入される，という枠組みを与えた。この方式によってロボットに試行錯誤的な動きをさせ，簡単な組み立て問題の雛形を例題として信頼性の高い作業遂行を実現した。

また，エージェントを受動的なオブジェクトと見て操作し，エージェント相互の監視・制御を行なうことを提案した。ロボットシステムが知能的であると言えるためには，ロボット自身に自己の行動を評価しながら行動できる能力が備わっている必要がある。ロボットでは，これを作業信頼性向上の基礎的問題として考えていくことが必要である。マルチエージェントによるシステム構成法は，ロボット内部に存在する独立なエージェントの相互作用を用いてこのような自己評価能力をもつロボットを実現しやすい枠組みであり，本研究では，行動ベースエージェントの実現やエージェント系を監視・表示するシステムの実現の中にこの考えを適用し，その有効性を示した。

本研究で実現したシステムは，動的能力を重視したインタプリタ型のシステムであるため，処理速度上の問題があり，ロボットのように本質的にリアルタイムの制御が要求されるシステムでは問題となりうる。低レベルの運動制御や知覚機能をも含めてすべてマルチエージェントの枠組みでプログラミングを行なうことのできるシステムは将来的な研究課題である。また，本章で述べた実験系は，単一の TAO/ELIS 上に実現された共有メモリ型のマルチプロセスをベースとしているため，真の意味での分散システムではない。通信のコストが重要な要素となるネットワークなどのような分散環境にシステムを拡張していくことも将来に残された研究課題である。

謝辞

本論文をまとめるにあたり，あたたかいご指導を賜わり，お励ましを頂いた大阪大学基礎工学部情報工学科の谷口健一教授に心から深謝致します。また，有益なご助言を頂いた同情報工学科の都倉信樹教授，首藤勝教授，大阪大学情報処理教育センターの松浦敏雄助教授に深謝致します。

筆者に本論文をまとめることをお勧め下さり，またその契機をお与え頂いた，筆者の東北大学大学院在学中の恩師である，三重大学工学部の那須正和教授，東北大学工学部の丸岡章教授に深謝致します。

本研究の TAO/ELIS プロジェクトは，日本電信電話公社武蔵野電気通信研究所，同基礎研究所，横須賀電気通信研究所，ソフトウェア研究所等において行なわれました。この間，ハードウェア ELIS の研究・開発は，北陸先端科学技術大学院大学 (現) の日比野靖教授のご指導のもとに行なわれ，またソフトウェア TAO の研究・開発は，NTT 基礎研究所の竹内郁雄グループリーダのご指導のもとに行なわれました。筆者は，それぞれの卓越したご指導を受けて本研究を行なう機会を得ました。ここに深く感謝致します。また，この TAO/ELIS プロジェクトが可能となるには，そのルーツにおいて，電気通信大学の故池野信一教授が主宰された池野特別研究室の，比類のない研究環境があったことを併せ付し，池野先生に心から感謝を捧げます。

日比野教授とともに ELIS 開発の主力となり，筆者にいろいろご教示頂いた NTT ヒューマンインタフェース研究所の渡邊和文主任研究員，および，TAO の研究・開発において竹内リーダとともに筆者をご指導頂いた NTT 基礎研究所の奥乃博主幹研究員に感謝いたします。特に，奥乃主幹研究員には，TAO オブジェクト指向の性能向上に関して重要な契機となった，スタンフォード大学知識工学研究所でのソフトウェア移植に際して公私に渡り極めてあたたかいご援助を受けました。

TAO オブジェクト指向については，多くの方々に使って頂き，貴重なご意見を賜わりました。NTT 基礎研究所の村上健一郎主任研究員，天海良治主任研究員の両氏には，それぞれネットワークシステム，ZEN エディタの開発にオブジェクト指向を使用していただき，バグ出しや改良のための助言を頂きました。同山崎憲一主任研究員は，オブジェクト指向を論理型と融合する仕事を行なわれました。また，ヒューマンインタフェース研究所の神尾視教主幹研究員は，オブジェクト指向を含む TAO のコンパイラを作成されました。

TAO/ELIS をロボットの制御に適用する研究では，NTT ヒューマンインタフェース研究所知能ロボット研究部の諸兄に実験環境を提供して頂く等のご援助を頂きました。とりわけ，大原秀一主任研究員に大変お世話になりました。また，NTT 基礎研究所の高田敏弘研究主任には，TAO 上の X ウィンドウ・クライアントのプログラムを使わせて頂きました。

本論文をまとめるにあたっては，ヒューマンインタフェース研究所知能ロボット研究部の酒井高志部長，立石和義グループリーダをはじめとする各位のあたたかいご理解を頂きました。

TAO/ELIS の研究・開発は，ブレッドボードによる試作機から商用機に至るまで，NTT，(株)NTT

インテリジェントテクノロジ, 沖電気工業(株), 岩通アイセル(株)等の, ここにご芳名を挙げきれないほどの極めて多くの方々のご努力によって遂行されました。筆者は, この膨大な仕事のごく一部を担ったものであり, 多くの研究者・技術者の方々のご支援なしでは本研究をなし得なかったことを付し, 各位のご厚情に, 深甚なる謝意を表わします。

参考文献

[1] L. P. Deutsch. A LISP Machine with Very Compact Programs. In *Proceedings of 3rd IJCAI*, 1973.

[2] R. Greenblatt. The Lisp Machine. Working Paper 79, MIT AI Lab., November 1974.

[3] 瀧和男, 金田悠紀夫, 前川博俊. LISP マシンの試作 — アーキテクチャと LISP 言語の仕様 —. 情報処理学会論文誌, Vol. 20, No. 6, pp. 481–486, 1979.

[4] 瀧和男, 金田悠紀夫, 前川博俊. LISP マシンの試作 — インタプリタの構造とシステムの評価 —. 情報処理学会論文誌, Vol. 20, No. 6, pp. 487–493, 1979.

[5] 日比野靖, 渡邊和文, 大里延康. LISP マシン ELIS の設計. 第 21 回全国大会予稿集, 5J-3, pp. 141. 情報処理学会, 1980.

[6] 日比野靖, 渡邊和文, 大里延康. LISP マシン ELIS の基本設計. 記号処理研究会資料, 12-15. 情報処理学会, 1980.

[7] D. G. Bobrow, K. Kahn, G. Kizales, and et al. COMMONLOOPS, Merging Common Lisp and Object-Oriented Programming. Technical Report ISL-85-8, Xerox Parc, August 1985.

[8] T. Chikayama. ESP Reference Manual. Technical Report TR-044, ICOT, February 1984.

[9] 竹内郁雄, 奥乃博, 大里延康. Lisp マシン ELIS 上の新 Lisp TAO. 記号処理研究会資料, 20-5. 情報処理学会, 1982.

[10] I. Takeuchi, H. G. Okuno, and N. Osato. TAO — A harmonic mean of Lisp, Prolog and Smalltalk. *ACM SIGPLAN NOTICES*, Vol. 18, No. 7, pp. 65–74, July 1983.

[11] 竹内郁雄, 奥乃博, 大里延康, 渡邊和文, 日比野靖. New Unified Environment (NUE) の基本構想. 記号処理研究会資料, 18-1. 情報処理学会, 1982.

[12] L. Periera, F. Periera, and Warren D. *Users Guide to DEC system-10 Prolog.* Dept. AI Research, Edinburgh Univ., 1978.

[13] A. Golberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison Wesley, Reading, Massachusetts, January 1983.

[14] Guy L. Steele Jr. *COMMON LISP The Language.* Digital Press, 1984.

[15] D. Weinreb, D. Moon, and R.M. Stallman. *Lisp Machine Manual.* LMI, 1983.

[16] S. E Keene. Flavors: Object-oriented Programming on Symbolics Computers. Technical report, Symbolics, Inc., 1985.

[17] 日本ロボット学会. 〔特集〕マルチエージェントロボットシステム. 日本ロボット学会誌, Vol. 10, No. 4,, 8 月 1992.

[18] R. A. Brooks. A Robust Layered Control System For A Mobile Robot. *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, pp. 14–23, 1986.

[19] 渡邊和文, 日比野靖, 大里延康. LISP マシン ELIS におけるハードウェアの繰り返し構造. 第 23 回全国大会予稿集, 4E-6. 情報処理学会, 1981.

[20] 日比野靖, 渡邊和文, 大里延康. Lisp マシン Elis のアーキテクチャ — メモリレジスタの汎用化とその効果 —. 記号処理研究会資料, 24-3. 情報処理学会, 1983.

[21] 竹内郁雄, 奥乃博, 大里延康. TAO における tag の活用法. 第 26 回全国大会予稿集, 4D-6. 情報処理学会, 1983.

[22] 竹内郁雄, 奥乃博, 大里延康. TAO – Lisp, Prolog, Smalltalk の調和平均. プログラミング・シンポジウム予稿集. 情報処理学会, 1983.

[23] 竹内郁雄, 奥乃博, 大里延康. NUE/TAO/ELIS の OS 的側面. 記号処理研究会資料, 24-8. 情報処理学会, 1984.

[24] 竹内郁雄, 日比野靖, 奥乃博, 大里延康, 渡邊和文. ELIS-TAO の性能評価. 第 28 回全国大会予稿集, 3F-2, pp. 221. 情報処理学会, 1984.

[25] I. Takeuchi, H. Okuno, and N. Ohsato. A List Processing Language TAO with Multiple Programming Paradigms. *New Generation Computing*, Vol. 4, No. 4, pp. 401–444, 1986.

[26] Hiroshi G. Okuno, Nobuyasu Osato, and Ikuo Takeuchi. Firmware Approach to Fast Lisp Interpreter. In *Proc. of the 20th Annual Workshop on Microprogramming (MICRO-20)*, pp. 1–11, Colorado Springs, December 1987. ACM and IEEE.

[27] 竹内郁雄, 奥乃博, 大里延康, 山崎憲一. 知能処理用マルチパラダイム言語 TAO. 研究実用化報告, Vol. 37, No. 2, pp. 137–143, 1988.

[28] 大里延康, 日比野靖, 渡邊和文. ハードウェア・デバッグ支援システムとしての Lisp の有効性について. 第 23 回全国大会予稿集, 4H-12. 情報処理学会, 1981.

[29] 大里延康, 渡邊和文. Lisp マシン ELIS の開発環境. 記号処理研究会資料, 17-3. 情報処理学会, 1982.

[30] 大里延康, 渡邊和文. Lisp マシン ELIS のマイクロプログラミング支援システムについて. 第 24 回全国大会予稿集, 3D-9. 情報処理学会, 1992.

[31] 大里延康, 竹内郁雄. 会話型 Lisp システム TAO/60. 研究実用化報告, Vol. 31, No. 11, pp. 2139–2152, 1982.

[32] 山崎憲一. TAO における論理型パラダイムとオブジェクト指向の融合. 第 34 回全国大会予稿集, 1P-3. 情報処理学会, 1987.

[33] 神尾視教. ELIS 上の Common Lisp コンパイラ. 第 34 回全国大会予稿集, 1P-4. 情報処理学会, 1987.

[34] 神尾視教, 竹内郁雄. ELIS の Common Lisp とコンパイラ. 研究実用化報告, Vol. 37, No. 2, pp. 153–157, 1988.

[35] Hiroshi G. Okuno. The Report of the Third Lisp Contest and the First Prolog Contest. 記号処理研究会資料, 33-4. 情報処理学会, 1985.

[36] Hiroshi G. Okuno, Ikuo Takeuchi, Nobuyasu Osato, Yasushi Hibino, and Kazufumi Watanabe. TAO: A Fast Interpreter-Centered System on Lisp Machine ELIS. In *Conf. Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pp. 140–149, Austin, Texas, August 1984. ACM.

[37] 大里延康, 竹内郁雄. 複合パラダイム言語 TAO におけるオブジェクト指向プログラミングとその実現. 情報処理学会論文誌, Vol. 30, No. 5, pp. 596–604, May 1989.

[38] Nobuyasu Osato, Hiroshi G. Okuno, and Ikuo Takeuchi. Object-oriented Programming in Lisp. 記号処理研究会資料, 26-4. 情報処理学会, 1983.

[39] 大里延康, 奥乃博, 竹内郁雄. 新 Lisp TAO におけるオブジェクト指向型プログラミングの実現機構について. 第 26 回全国大会予稿集, 4D-5. 情報処理学会, 1983.

[40] N. Osato and I. Takeuchi. Object-Oriented Programming in Multiple-Paradigm Language TAO and Its Implementation. *Journal of Information Processing*, Vol. 14, No. 4, pp. 501–507, 1991.

[41] 竹内郁雄, 大里延康. No-method-found ハンドラとその応用. 第 38 回 (昭和 64 年前期) 全国大会予稿集, 3P-5. 情報処理学会, 1989.

[42] 天海良治. オブジェクト指向による画面エディタの部品化. 第 33 回全国大会予稿集, pp. 771–772. 情報処理学会, 1986.

[43] 杉村利明, 竹内郁雄, 奥乃博, 天海良治. ELIS の日本語処理. 研究実用化報告, Vol. 37, No. 2, pp. 145–152, 1988.

[44] 村上健一郎. LAN プロトコルのオブジェクト指向による実現. 第 34 回全国大会予稿集, 1P-7. 情報処理学会, 1987.

[45] 大里延康. エージェントをベースとしたロボット・プログラミング. 電子情報通信学会春季全国大会予稿集, D-231, pp. 1822–1827. 電子情報通信学会, 1991.

[46] 大里延康, 奥乃博, 大原秀一. マルチエージェントシステムとその行動ベースロボット制御への適用. 日本ソフトウェア科学会第8回大会論文集, D1-6, pp. 81–84. 日本ソフトウェア科学会, 1991.

[47] 大里延康. ロボット制御エージェントの行動インタプリタ. 電子情報通信学会技術研究報告 [人工知能と知識処理], AI92-91. 電子情報通信学会, 1993.

[48] Nobuyasu Osato. An Action Interpreter of a Robot Control Agent. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems '93*, pp. 1126–1133, Yokohama, July 1993. IEEE/RSJ.

[49] 大里延康. MARCS: マルチエージェントによるロボット制御. 電子情報通信学会論文誌 D-I, Vol. J75-D-I, No. 8, pp. 714–722, 1992.

[50] 石川裕, 所真理雄. オブジェクト指向並行プログラミング. 情報処理, Vol. 29, No. 4, pp. 325–333, 1988.

[51] 渡部卓雄, 松岡聡, 米澤明憲. 並行オブジェクト指向計算における自己反映計算の一方式. 並列処理シンポジウム (JSPP'91) 論文集, pp. 421–428. 情報処理学会, 1991.

[52] J. H. Connell. A Behavior-Based Arm Controller. *IEEE Transactions on Robotics and Automation*, Vol. 5, No. 6, pp. 784–791, 1989.

[53] Brian Yamauchi and Randal Nelson. A Behavior-Based Architecture for Robots Using Real-Time Vision. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pp. 1822–1827, 1991.

[54] 佐藤知正, 平井成興, 松井俊浩. 自律ロボットの構成法に関する一考察. 日本ロボット学会第7回学術講演会予稿集, pp. 257–258, 1989.

[55] R. A. Brooks. A Robot that Walks: Emergent Behaviors from a Carefully Evolved Network. In P. H. Winston and S. A. Shellard, editors, *ARTIFICIAL INTELLIGENCE AT MIT Expanding Frontiers*, pp. 29–39. The MIT Press, Cambridge, Massachusetts, 1990.

[56] 高田敏弘. – NueX – オブジェクト指向による X Window System インターフェイスの実現. In *WOOC'89*, 1989.

付録 A

クラスベクタの内部構造

(1) **version number**
このクラスの世代番号を示す。クラスの世代番号は, 0 から始まって, このクラスが再定義されるごとに 1 ずつ増えていく。

(2) **id-message-vector**
このクラスのオブジェクトが受けつけるメッセージ・セレクタと, それに対応するメソッドの表である。バイナリ・サーチができるように, セレクタのアドレスでソートされている。この表は, このクラスに直接に定義されたメソッドのみならず, スーパークラスから継承してきたメソッドも含んでいる。

(3) **symbol backpointer**
このクラスの名前を表わすシンボル。このクラスベクタは, このシンボルの `class` という属性の値になっている。

(4) **list message vector**
リストの形をしたメッセージ・セレクタに対する **Hclauses** を保持する。

(5) **super symbol message vector**
3.2.7 で述べた関数 `super` が参照するメソッド表。

(6) **property list**
クラスベクタは属性を持つことができ, その属性リストはシステムが内部的に使用しているが, 場合によってはユーザが使うこともできる。TAO の属性処理関数 `putprop` や `remprop` は, `vtitle` が `class` であるベクタを識別してこの属性を操作する。クラスの属性は, システムプログラムにおいて, あまり速度を必要としないデータを保持する目的で用いている。

(7) **class variables**
クラス変数のベクタ。形式はインスタンスとほぼ同じである。クラス変数は, TAO の通常の意味での変数ではなく, 専用の関数である `cvar` を用いてアクセスする。

(8) **make-instance skeleton**
インスタンスが生成されるときに参照される情報で, 継承したものも含むインスタンス変数名のリスト, 初期値のリスト, 生成時に評価する式等を持つ。

(9) **defclass-skeleton**
クラスを定義する `defclass` マクロの情報を，ほぼそのまま保持する。

(10) **instance variable hash**
3.3.5で述べたインスタンス変数のハッシュ表。

付録 B

典型的なメッセージ伝達式のマイクロプログラムのトレース

以下のプログラムは, **udo** へのメッセージ伝達を行なう式を評価するマイクロプログラムの制御のメイン・ストリームのソース上でのトレースを示したものである。評価される式 (**form**) を

```
[obj msg args ...]
```

とする。コメントではこれらの記号を用いている。このプログラムの入口での条件は, 通常の **eval** であるから, スタックのトップが次の形になっている。

```
<sp>   => form
<sp+1> => return address
```

```
; eval の入口。ここはメッセージ伝達式のみならず，すべての式の評価の入口である。
(!!eval (and <sp>+ gmc car0)
        (boc car0 mdr1)                          ; form を mdr1 に読む。
        (br nhap (evi ev)) )                     ; 割り込み条件のチェック
(!!ev   (br tag5-0 eval-disp)                    ; form のデータ型で分岐
        (com sysmode) )                          ; evalhook のチェック
(.case !eval-disp dtyp#
 ((list bra nambra)                              ; form は [ ] である。
  (mov car0 -<sp>)                               ; form を保存して...
  (br y6 (hmessage message)) ))

; 以上の eval のステップで，メッセージ伝達式であることが識別された。
; 以下では，まず form の car を一度評価して，それが udo であることを
; チェックする。

(!!message
        (mov cdr1 -<sp>))                        ; form の cdr を保存。
(       (mov m1 -<sp>))                          ; obj を評価するため戻り番地
(       (and car1 gmc car0) (boc car0 mdr1)      ; m1 を保存する。
        (br nhap (evi ev)) )                     ; obj の評価のため eval へ。
```

```
(!m1     (and <sp>+ gmc r1)                    ; 評価結果を r1 へ。
         (brc tag5-0 evaled-receiver) )        ; そのデータ型で分岐。
(.case evaled-receiver dtyp#                   ; ここでは obj は udo であった
 ((list udo)                                   ; とする。
  (mov <sp>+ r0) (goto objudo) ))              ; form の cdr を r0 へ。

; obj の評価が終了し，それが udo であることが識別された。以下では udo に
; 対するメッセージ伝達式として form の cdr 以降を調べる。

(!!objudo
         (mov r1 car0) (bo car0 mdr2) )        ; udo の頭部を mdr2 へ
                                               ;  car2 に class-vector
                                               ;  cdr2 に udo のサイズ
(        (mov r0) (jsrc tagh5-0 delinv))       ; form の cdr を delinv
(        (mov r0 car0) (bo car0 mdr0)          ; (msg args ...) を mdr0 へ
         (br tagcadbl (u1 objud)) )            ;
(!!objud(and car0 gmc r0))                     ; r0 = msg
(        (mov cdr0 -<sp>)                      ; args をスタックに保存。
         (jsr tagh5-0 delinv) )                ; msg を delinv する。
(        (mov car2 r5)                         ; class-vector を r5 へ
         (br tagcadbl (u-id u-list)) )         ; msg の型をチェック。 id か?
(!u-id   (+ car2 1 car0) 24tc (bo car0 mdr1))  ; class-vector の頭部を mdr1 へ
                                               ;  car1 にメソッド表。
(        (- r0 :&))                            ; instance fact のチェック
(        (br neq (u-id& u-id')) )
(!u-id'  (+ car1 1 r6) 24bwt (bo car1 mdr0))   ; メソッド表の頭部を mdr0 へ
                                               ; メソッド表の先頭番地を r6 へ
(        (- r6 1 r2) (br tagnil (u-id0 u-id1')))
(!u-id0 (mov cdr0 cdr0) 24tc sra)

; msg をキーにして id-message-vector をバイナリ・サーチする。バイナリ・サーチ
; は [付録 C] に示すように bins という入口をもつサブルーティンである。

(!!u-id0' (+ cdr0 r2 r2) 24tc (jsr no bins))   ; メソッド表の最後番地を r2 へ
                                               ; r0 をキーに，バイナリ・サーチ
(        (mov car1) 24bw                       ; メソッドが見つかったときは
         (br tag7 (u-id00 u-id01)) )           ;  car1 にそのエントリ。
(!u-id00(mov car1) 24bw                        ; エントリの型で分岐。
         (br tag5-0 what-method) )
(.case !what-method 64                         ; 見つかったメソッドは
 (applobj                                      ; applobj だった。
```

```
        (mov <sp>+ car0)
        (bo car1 mdr2) (goto appl) ))          ; applobj の vtitle を car2 へ

; 以下では，見つかったメソッド applobj の内容に対応するスタック・フレームを
; 作る。メソッド applobj のフレームは，スコープの制限をするとともに udo という
; 環境を持っていなければならない。すなわち，一種のバリュー・セルの集合を包み
; こんだクロージャのようなものである。フレームの構造は本文中に図示した。

(!appl  (mov car1 -<sp>))                      ; applobj を保存。
(       (mov ef -<sp>))                        ; ef チェインを保存。
(       (mov car2))
(!appl2 (mov tf -<sp>) (br y3-0 applc))        ; tf チェインを保存。
(.case applc 16                                ; applobj の種類 (引数の個数)
 (#!expr-simple                                ;  で分岐。 expr-simple だった。
        (+ car1 2 car1) 24tc (bo car1 mdr2)    ; 引数の個数を cdr2 に
        (goto appl5) ))
(!appl5 (+ sp                                  ; tf に applobj の属性表示。
         ^(+ (getsym 'tagstkpt)                ;  局所変数バイアス = 0
         #22000001)                            ;  環境フレーム・フラグ
         tf ))                                 ;  スコープ制限ビット
(       (mov r1 -<sp>))                        ; self(udo) を環境 vector と
                                               ; してスタックに保存

; 以上でメソッド評価のためのスタック・フレームが作成された。このあと，メソッド
; applobj の情報を参照しながら，args... を順次評価し，スタックに積む。

(       (- cdr2 1 r2) htf)                     ; 変数の個数をチェック
(       (mov car0) (br z (ap7 ap8)))
(!ap7   (br tagcadbl (ap1 ap2)))
(!ap2   (jsr no bothd))                        ; args... を mdr1 に読む。
(       (mov cdr1 -<sp>))                      ; args... の尻尾を保存
(       (- r2 1 -<sp>))                        ; 引数の個数をカウンタ r2 で
(       (mov ap3 -<sp>))                       ;  数えながら eval を呼んで
(       (and car1 gmc car0) (boc car0 mdr1)    ;  順次評価する。
        (br nhap evi) )
(!ap3   (and <sp>+ gmc r0))
(       (mov <sp>+ r2) 24bw)
(       (mov <sp>+ car0) (br z (ap4 ap5)))     ; ここまでが評価のループ。
(!ap4   (mov r0 -<sp>) (br tagcadbl (ap1 ap2)))
(!ap5   (+ tf 1 sp2) (br tagcadbl (ap10 ap11))) ; sp2 が applobj を指す。
(!ap10  (+ <sp2> 2 car1) (bo car1 mdr2))       ; applobj の本体を car2 に。
```

```
(       (mov r0 -<sp>))                          ; 最後の引数をプッシュ。

; 以上でメソッドの実行環境が整った。以下，applobj の本体が S 式であるか memblk
; であるか（コンパイルされているか）に応じて seq(関数 seq の実行本体) に飛ぶ
; か enter-lap(コンパイルド・コードの実行本体) に飛ぶかで実際にメソッドを
; 実行する。

(!ap5'  (mov exit-frame -<sp>))                  ; 式の評価後の戻り番地。
(       (mov car2))
(!!app13(mov tf ef) (br tag5-0 ap-disp))         ; applobj 本体の種類で分岐。
(.case ap-disp 64
 (cell  (mov car2 -<sp>) (call seq))             ; メソッドは S 式である。
 (memblk (+ car2 1 car0) 24tc (bo car0 mdr3)     ; メソッドはコンパイルされ
        (goto msglap) ))                         ;  ている。
(!msglap (+ tf 1 sp3) (goto enter-lap))
}
```

付録 C

バイナリ・サーチのマイクロサブルーティン

```
; binary search for id-message
; r6 = head position of the current table
; r2 = tail position of it
; car0 = position to be examined
; r0 = key
; all relevant addresses are nilnum!
; returns car1 = corresponding method if not nil

; acrobatic version with nVv and z

(!!bins (+ r6 r2 car0) sra (bo car0 mdr1))
(!lp    (- r2 r6))
(       (- car0 1 r3) (br n (cont not-found)))

(!cont  (- cdr1 r0))                          ; no care for gm
(       (+ car0 1 r4)                         ; note id's are sorted as 32bit
        (br gel (big found small)) )          ; number (with tag, id/sysid)

(!found (and car1 #17777777777 car1) rts)

(!big   (+ r6 r3 car0) sra (bo car0 mdr1))
(       (mov r3 r2) (goto lp))

(!small (+ r4 r2 car0) sra (bo car0 mdr1))
(       (mov r4 r6) (goto lp))

(!not-found
        (mov #20000000000) rts)               ; gcmark tarari(not-found code)
```