オブジェクト指向型モデルに基づく
ロボットプログラミングシステムの研究

松 井 俊 浩

オブジェクト指向型モデルに基づく
ロボットプログラミングシステムの研究

電子技術総合研究所
知能システム部 自律システム研究室
松井 俊浩

1990年8月

目次

表目次

図目次

第 1 章

序　論

1.1　本研究の背景と経緯

ロボットは、感覚し、認識し、計画し、対話し、行動する知能体である。そのプログラミングは、マニピュレータの実時間制御、センサ処理、協調プロセス制御、幾何モデリング、対話的作業教示、動作計画と推論など多岐に亙る。このような大規模な知能システムの構成には、各々のモジュールが高い機能を発揮できるようにすることと、それぞれが有機的に統合されることが必要である。そして、これら両方にとって重要な役割を演ずるのがモデル、すなわちロボット、環境物体、作業内容などに関する知識の計算機内部での記述である。たとえば、センサが環境を認識するには、必ずモデルとの照合を経なければならないし、動作の計画には物体の形状や位置・姿勢のモデルが必要である。人間が対話的に動作を教示する場合にも、やはりモデルを通じた情報の授受が介在する。

ここで、従来の代表的なロボット言語におけるモデルの働きを見てみよう [54]。世界で最初の知能ロボットのプログラミング言語と考えられるのは Stanford 大学の AL[30] である。AL は、座標系フレームを用いてモデルを表現し、マニピュレータのエンドエフェクタと作業対象物体の座標系を用いて動作をプログラムする。言語は Pascal に似た構文を持ち、標準的な動作レベルロボット言語とみなすことができる。物体の把握、取り付けなどにおける物体相互の依存関係 (affixment と呼ばれる) の変化を座標系の連結として管理しており、物体の位置を他の物体からの相対座標系として定義することができる。AL は形状モデルを持たないので、形状から動作に必要なパラメータを計算したり幾何学的推論を働かせることはできない。

Edinburgh の RAPT[37] は、NC マシン用言語である APT にロボット向きの機能を拡張したものである。モデルの状態を、面、エッジなどの空間的関係によって記述するのが特徴である。したがって、RAPT は、マニピュレータの存在を無視して物体の拘束・位置関係によって作業をプログラムすることができる対象物レベルのロボット言語であると言える。また、依存関係を数学的に記述し、数式処理によって拘束の伝播、解消を処理することができる。物体の形状は、面、エッジなどの特徴の集まりとして定義されるが、その定義法

はこみいっているため記述量は増大する。また、3次元の体積を表すことはできない。

IBMのAUTOPASS[23]は、独立したCAD用ソリッドモデラの上で作成された形状モデルをロードして、幾何学的プラニングを行ない、高水準の作業レベルでのプログラミングを達成することを目的としている。AUTOPASSに組み込まれるモジュールの形で把握計画、障害物回避軌道計画などの問題が個別に研究され、幾何モデルに基づく推論の重要性を印象付けた。また、そこで用いられた形状定義法であるB-repが、標準的なモデル表現法として確立された。

このようにロボット言語の発達の過程を概観すると、モデルと言語の機能は密接な関係があり、豊かな表現能力を持ったモデルを用いることで言語の水準がしだいに高まることがわかる。しかし、AUTOPASSをはじめとする上記ロボット言語は、完全なインプリメントがなされたわけではない。実際、現在の産業用ロボットがこれらの高水準のモデルを備えた言語によってプログラムされることは皆無と言ってよく、現実的な応用においては、IBMのAML[46]のような、汎用言語に近い動作レベルの記述を目的とした言語が使われるにとどまっている。また一方でモデルを用いるにしても、どのようにして実世界の物体のモデルを獲得するか、どのようにして作業の計画を完全に自律的に進めるか等、多くの困難があり、人間による対話的な教示も重要な課題として残されている[1]。

対話的なロボット制御法としては、マスタスレーブ法が宇宙、原子力プラント、海洋などにおける遠隔制御に効果を上げている。バイラテラルサーボによる触覚のフィードバックや、異構造スレーブを制御する方法が開発され、精緻な作業を汎用的に処理することができるようになってきている。しかし、これらの方法は環境に関するモデルを一切持たないので機械の知能を働かすことができない。したがって、作業に熟練を要する点、繰り返し作業や危険な作業への対応など人間のオペレータに対する重圧が取り除かれることがない点で限界がある。

1.2 本研究の目的

前節に述べたように、従来の高水準のロボット言語は必ずしも成功を収めたとは言えない。その理由として、ベースとなる言語の機能が不十分で広範な問題に対する記述能力が欠けていたこと、モデルの獲得、作業計画、実行監視に統一的に利用でき、少ない記述量で済む拡張性の高いモデルが構成できなかったこと、モデルを実世界の環境に対応付ける手段が提供されなかったこと、などが挙げられる。これらの問題に対処するために、本研究は、オブジェクト指向型モデルをベースにし、モデルと実世界の対応付けまでを含んだ統一的なロボットのプログラミングシステムを構成することを目的とする。そのためにまず、豊富な機

[1]工場では、今なお熟練工のダイレクトティーチングによるプログラミングが主流となっているが、ここで言う対話的教示とは、オフラインプログラミングにおけるグラフィクスを用いた教示手法を指している。

能を備えAIに幅広く用いられているCommon Lispをベースとして、ロボットプログラミングの核となるオブジェクト指向型言語EusLispを設計する。オブジェクト指向は、ロボットのような実体を伴った世界をモデル化するのに適した考え方であると同時にLispの機能を包含し、個々の機能を抽象化しつつ大きなシステムを統合的に構成するのに適した概念であると考えたからである。

このロボットプログラミング核言語EusLispの上に、ロボットのための幾何モデラと作業記述システムを構築する。従来、幾何モデラは大規模なソフトウェアであり、ロボットプログラムに組み込むことは困難であると考えられていた。オブジェクト指向の持つ階層性、モジュール性、拡張性を最大限に活用することで、複雑な幾何モデラをコンパクトに記述し、幾何モデルを利用したロボットの作業記述プログラムが容易に構成できるようになることを示す。

モデルは、外界に対するシステムの内部表現である。このモデルは現実の世界をなるべく忠実に表す物でなければならず、また、システムの人間が対話するためには、モデルは人間にとってもわかり易い形で表現・呈示されなければならない。そのための表示装置としてマルチメデイアディスプレイ（MMD）を開発する。マルチメディアディスプレイは、文字、グラフィクス、TVカメラ画像3種の画像メディアの重畳表示、マルチウィンドウ、実時間グラフィクス、立体視、を実現するためのディスプレイ装置である。MMDを通じて3次元の実世界、モデル、人間の間の効率良いインタラクションが実現できることを示す。

最後に、前記のモデルとマルチメディアディスプレイを統合し、環境モデルの決定、作業のプログラミング、シミュレーション、実行の監視を行なうロボットの遠隔作業システムを提案する。

1.3 本論文の構成

本研究は次のような要素から構成される。

第2章 オブジェクト指向型ロボットプログラミング核言語EusLisp では、広い範囲のロボットのプログラミングに必要な機能を検討し、これらを効率良く実現するためのCommon Lispの拡張について述べる。システム全体をオブジェクト指向の概念で統一するため、Lisp自体がオブジェクトの上に実現されるのが大きな特徴である。オブジェクト指向は、システムの統合化に重要な機能を多く含んでいるが、効率の良い処理系が実現できるかどうかが問題となる。EusLispでは、高速な型検査法を実装することでメッセージ送信に依存するプログラミングを減らし、メモリ管理にフィボナッチバディを用いることで高い効率を得ている。また、EusLispの基本データ型を拡張する形で、ロボットプログラミングに必須な幾何演算機能、他言語インタフェース機能、ウィンドウシステムインタフェースを効率良く実現できることを示す。

第3章 作業記述のためのオブジェクト指向型幾何モデラ では、EusLisp の上に構築される3次元幾何モデラとロボットの作業をプログラミングするシステムについて論ずる。幾何モデリングは、環境物体とマニピュレータの3次元形状を表現するのに不可欠の機能である。EusLisp の幾何演算機能とクラスの継承によって、プログラム記述量を劇的に減少させることができる。また、オブジェクト指向によってシステムは高い拡張性を発揮でき、従来のソリッドモデラでは困難であった属性の追加が容易に行える。この機能を利用したマニピュレータと環境物体の作業情報のモデリング手法、およびマクロ動作記述と仮想マニピュレータ命令による抽象化によって広範囲のマニピュレータと組み立て作業を記述する枠組について論ずる。

第4章 モデルと実世界の視覚インタフェース、マルチメディアディスプレイ では、視覚を通じてモデルと実世界の対応を取るためのマンマシンインタフェースであるマルチメディアディスプレイ (MMD) について論ずる。まず、3次元のモデルと実世界を扱うのに必要な表示機能を既存のグラフィクス標準やウィンドウシステムと比較して検討する。そして、MMD に特徴的な機能であるマルチメディア機能、マルチウィンドウ、高速グラフィクス、立体視の実現法について述べる。三つの映像メディアは、各々独立し並列動作するパイプラインによって生成され、ウィンドウマッピングメモリによってマルチウィンドウ表示される。さらに、これらのパイプラインの実行を制御するファームウェア、MMD との通信、入出力サービスを管理するホストワークステーション上の基本ソフトウェアについて論ずる。

第5章 マルチメディアディスプレイを用いたロボットの遠隔作業システム では、環境モデルとマルチメディアディスプレイを用いて、実際にロボットの遠隔作業を実施するシステムについて述べる。まず、ディスプレイ上でモデル画像が TV カメラ画像に重なるように対話的にガイドすることで環境物体のモデルを作成する。次にモデルの世界で、作業プログラムを記号的記述する。このプログラムは MMD のグラフィクスによってシミュレーションされ、スーパインポーズされた実際の環境と照らし合わせて不都合のある動作を編集する。完成された動作は実行ステーションに送られ、実際のマニピュレータによって実行される。この場合も、MMD 上でモデルと実際のロボットの動作を重ねて表示することにより、エラーの発見や TV カメラからの画像に遅延があるような遠隔作業環境での作業が容易になることを示す。

最後の第6章では、本論文で述べたシステムを利用して進められている他の研究に触れつつ本研究で得られた成果を総括し、今後のロボット研究を展望する。

第 2 章

オブジェクト指向型ロボットプログラミング核言語 EusLisp

2.1 ロボットプログラミングの特性と言語

ロボットプログラミングは、「3次元幾何モデルに基づく実時間ＡＩプログラミング」と特徴づけられる。そのために必要な言語機能として次のものが上げられる。

(1) **幾何計算とソリッドモデル** 物体の形状を認識したり、形状から導かれる拘束に基づいて動作を表現し計画するためには3次元ソリッドモデルが不可欠である。それらはベクタ、マトリクス演算、交点計算等を多く含む。また位相情報を整然と表現するためリスト処理能力を必要とする。

(2) **ワールドモデルと推論機構** ワールドモデルは、作業環境中に置かれた物体の特性や物体間の結合・依存関係 (affixment) 、それらの時間変化を記述する。ワールドモデルに蓄積された知識からゴールに到達する手順の自動生成のために論理的な推論機構が必要になる。

(3) **モジュラリティと拡張性** ロボットシステムを構成する多くのサブシステムは未だ完成されておらず、改良、拡張が頻繁に起こる。この拡張を容易に進めるには各機能をモジュール化し、その変更、交換、追加が他のモジュールと干渉しあわないように行えることが大切である。適切なモジュール化は、モジュールを組合せて個別の目的に適合したシステムを構成するためにも重要である。

(4) **実時間・非同期プログラミング** ロボットを物理法則によって定まる反応時間で制御するには、高い実行効率と非同期の事象に対する応答性能が要求される。またロボットシステムは非同期に処理を進める多くのサブシステムの融合体として構成される。これらのプロセスの間の効率の良い通信手段が提供されねばならない。[1]

[1] ここではサーボのような特段の高速性を要求されるプログラムを記述することを考えているわけではない。サーボは、いわば周辺装置の機能であり、ロボットプログラミングシステムからはデバイスとしてアクセスされる。サーボのような特段の実時間性を要求される部分を除けば、言語に要求される応答性は数十ミリ秒の単位である。

(5) 対話インタフェース 現段階のロボットの知能は限られており、動作の教示、実行の監視、エラー処理、など相対に人間とのインタラクションに頼る比率が高い。円滑な対話のためには、グラフィクスやマルチウィンドウなどを用いたビジュアルなインタフェースの利用が効果的である。良好な対話インタフェースは、プログラム開発の効率化にも貢献する。

これらの機能は言語の形で提供されるのが望ましい。そのようなロボットプログラミングの核となる高水準の言語の設計には2通りのアプローチが考えられる。最初から目標となる言語の仕様を厳密に定めて、1から完全な専用言語を作成する方法と、既存の一般的な言語に改良・拡張を加える形で必要な機能を実装する方法である。

前者は最終的な言語の仕様を最適に定められればコンパクトで効率の良い処理系が実現できる可能性があるが、そのような固定的な仕様を最初から定めることは不可能である上に、技術の進歩に伴って必要になる機能を柔軟に追加することが困難になり、言語の習熟にも時間がかかる。たとえばAL[30]、RAPT[37]、AUTOPASS[23]、LAMA[25]などのいわゆるロボット言語は、ロボット用に特化しているので、拡張性が限られており特定のマニピュレータとの結合が強い。したがってこれらの言語の機能の上にさらに高次の機能を積み重ねたり、下位モジュールを入れ替えたり、異なった構成のロボットを接続することが困難になっている。このような拡張性の制約のため、先駆的な試みにも拘らずこれらの高水準のロボット言語が実用に供されることはなく、現場では特定のマニピュレータ専用の動作レベル言語が利用されるにとどまっている。

後者の方法には、柔軟な構成が取れる、従来の研究の成果を継承できる、習熟が容易であるなどのメリットがあるが、所望の機能が効率良く達成できるかどうかは、ベースとなる言語の表現能力、抽象化機能に大きく依存する。Cのような比較的低レベルだが万能型の言語では、効率的には満足できても最終的な機能を実現するための記述量が膨大になるために開発、保守に困難が増す。一方、PrologのようなAI言語では、効率、汎用性などに問題がある。

そこで本研究では、ベースとなる記述言語としてはLispを採用する。Lispは、言語の抽象化機能、記述性、効率、汎用性のいずれにおいても高い水準にあり、S式の持つ単純かつ強力な表記法、会話的なプログラミング環境、インクリメンタルな拡張性がロボットのような大規模システムの構築に重要だからである。Lispを核としたロボット用プログラミングシステムには東京大学ののAL/L[57]があり、その上には実験的な作業レベル言語[69]が試作された。これらのシステムは、Lispの上にMinskyのフレーム理論[29]の実現言語であるFRL[39]をインプリメントし、FRLを用いて座標系や物体の関係を記号的に記述している。フレームは、ako(a-kind-of)リンク、デフォルト値、アクティブバリューなどの知識の表現法として注目すべき性質を備えている。しかし、Lisp上のFRLは効率が悪く、記述能力も低いので、幾何モデルを記述するには至らず、積木の世界を抜け出て現実の問題を扱う

のは難しい。本研究では、フレームが発展・一般化した概念であるオブジェクト指向プログラミングを基盤に置くことで、幾何モデル・作業記述に向けたLispの統一的な拡張を可能とし、効率的なシミュレーションの記述・実行を目指した。

2.2 EusLispの設計

2.2.1 Common Lispの拡張

Lispは、Common Lisp[16]によって標準化が図られている。Common Lispの設計目標は次の点にあった。

1. 特定のハードウェアに依存せず、移植性が高い
2. 既存の主要なLispに対して上位互換性を保つ
3. インタプリタとコンパイラに同一の意味解釈を保証する
4. 特にコンパイラが生成するコードが高い効率を発揮できるようにする

一方、並列計算、グラフィクス、プログラミング環境、オブジェクト指向は最初の標準化案には含められていない。また、Common Lispの重要な言語仕様には次のようなものがある。

- 変数、関数束縛のスコープはレキシカルとする
- 引数の評価法によって関数をいくつかの種類に分けることをやめ、関数、マクロ、特殊形式だけとする
- 関数クロージャを厳密に実現する

これらは、従来のLispで問題となっていたインタプリタとコンパイラで動作が異なる現象を解決し、コンパイラの性能を上げたり、処理系が安定に機能することを目指して定められている。次の機能は、言語をより便利に、見易く、汎用性の高いものにする目的で導入されている。

- 関数の結果として複数の結果を返す多値
- 分離されたシンボル空間を与えるパッケージ
- リストとベクタを統一的に扱うためのシーケンス
- キーワードパラメータ

また、データ型として、次のものを含む。

- 整数、実数、複素数、多倍長整数 (bignum)、有理数などの数型
- 配列、文字型、文字列、ハッシュ表、乱数、パス名
- ストラクチャ

このように、Common Lisp は、従来の Lisp 方言の共通部分を標準として定めるのでなく、それらのスーパセットとして多くの種類の問題にすぐに応用できるように十分な機能を網羅し、理想的な Lisp 像を野心的に目指している。しかし、Common Lisp がロボットプログラミングに必要なすべて機能を備えているわけではない。また、汎用性を追求するためにロボットに不要な機能も多く含まれてしまっており、必要以上のメモリを消費したり効率を劣化させる要因となっている。

EusLisp[63, 26] では、Common Lisp に、ロボットプログラミングの記述に重要なもの、効率向上に大きく影響する機能を拡張し、逆にロボットに必須ではなく、効率を阻害する原因になるものを取り除くこととした。2.1で述べた要項と照らし合わせて、ロボットプログラミングの見地から、Lisp に不足していると考えられる機能は、モジュール性、幾何数値演算、実時間性である。一方、効率を犠牲にしてまで導入する必要性の少ないものには、多値、有理数、複素数などがある。EusLisp は、オブジェクト指向を導入することによって、モジュール性を高め、型と手続きの階層的抽象化を図る。オブジェクトによって幾何演算やロボットモデルが Lisp の自然な拡張として記述できるようになる [58]。また、メモリ管理、他言語プログラムや OS との結合方法を工夫することで、実時間性に関しても部分的な解決を図る。

2.2.2 オブジェクト指向

Lisp は、基本的に関数の集合によってプログラムを記述する。会話的なプログラミングを重視しているので、Lisp の関数はグローバルに定義されることがほとんどであり、Pascal のように静的なスコープを持ったブロック構造によってモジュール性を高める方法とは整合性が悪い[2]。

そこで、オブジェクト指向の概念によってこの問題の解決を図る。オブジェクト指向は、データの隠蔽と抽象化といった機能と並んで、シミュレーション、モデリングに適すると評される [28, 78]。この評価は、初期のオブジェクト指向言語である Simula、Smalltalk がシミュレーションを主要な題材として捉えていたことと符合する。ロボットも、3 次元の物体を扱うことを目的とするのであるから、オブジェクトによって物体の振舞いを記述すること

[2]関数より上位の構造としてパッケージがあるが、これはシンボルの空間を区切って名前の衝突を防止するのが主目的である。パッケージは入出力とも密接に関係し、小さな単位のパッケージを多数作成することは混乱を招き易い。

ができれば、直感的理解もシミュレーションの記述も容易になることが期待できる。また、物体に固有の属性をオブジェクトに閉じ込めて表現ができるので、影響の及ぶ範囲を限定してプログラムのモジュール化を促進できる。さらに、クラスの継承によって一般的なクラスから具体的なクラスを派生できるので、大規模なプログラムを段階的に開発するのに適している。

2.2.3 幾何数値演算

Lisp は、記号処理とリスト処理を主な目的としており、数値計算への応用は不得意とされてきた。その理由としては、速度、信頼性、記述性の三つが考えられる。

1. 演算の中間結果をメモリセルに割り付けるので記憶管理に負荷がかかる。また、浮動小数を要素とするベクタ、マトリクスを効率良く表現できない、その演算プリミティブが標準では定義されていない。

2. 定評のある数値計算パッケージを使うことができず、低レベルのサブルーチンからユーザがコーディングする必要がある。

3. 四則演算の表記にも関数型の prefix 記法を用いねばならない。

EusLisp では、浮動小数、整数の表現は 1 語に収まる即値だけに限り[3]、数値演算に伴ってメモリが消費されることがないようにしている。また、型を宣言しておけば、演算の度にポインタと数値との変換が起こるのを省略できる。ベクタ、マトリクスに関しては、中間結果の格納にある程度のメモリを消費することは免れない。しかし、整数、浮動小数のベクタ、マトリクスに適したデータ構造を用意し、その間の、内積、外積、マトリクス積、回転などの演算を組み込み関数で備えることで、中間結果をスタックに置けるようにしている。既存の数値計算ライブラリ等、従来のソフトウェア資産を活用し、Lisp 以外の言語の利点を活かすために柔軟な他言語インタフェースを用意する。数式の表記に関しては、図 2.1に示すような infix 表記を prefix 表記に変換する % リードマクロを設けることで簡潔な表記を可能にしている。

2.2.4 実時間性

実時間性とは、外部で非同期に生じる事象に反応するのに要する遅延の程度を意味している。高い実時間性を発揮するためには外部の事象に即座に反応できねばならず、そのためには無用の計算の中断があってはならない。

[3]EusLisp は、複素数、bignum を持たない。特に bignum は数式処理を除いて必要性が低く、型検査が増えるので効率を阻害する原因になり易い。

```
(defun sin3 (A)
   %(-4 * sin(A) ** 3 + 3 * sin(A)))
      ↓
(defun sin3 (A)
   (let* ((#:g51 (sin A)))
      (+ (* -4 (* #:g51 #:g51 #:g51)) (* 3 #:g51)) ))
```

(a) sin3 の定義とその展開形: sin の呼び出しが 1 回だけになるように最適化されている

```
(defun m+ (A B)
  (let* ((X (array-dimension A 0))
         (Y (array-dimension A 1))
         (C (make-matrix X Y)))
     (dotimes (i X C)
        (dotimes (j Y)
           %(C[i j] = A[i j] + B[i j])))))
```

(b) 配列アクセス、代入の記述法

図 2.1: % リードマクロによる数式記述

Lispは自動メモリ管理を行なうことを特色としている。すなわち、処理に伴ってメモリが消費され、利用できるメモリがなくなるとゴミ集め(ガベジコレクタ)が起動され、不要になったメモリセルを見つけて回収する。このゴミ集めのために、Lisp本来の計算が停止する。いわゆる「実時間ゴミ集め」も開発されてはいるが、ゴミ集めの処理を通常の計算プロセスの中に紛れ込ませているため、特別のハードウェアがない限り全体のスループットは一般的なマーク付け・走査型のゴミ集めより劣る。

EusLispは、実時間ゴミ集めを行うことを諦める代わりに、以下のように、極力ゴミ集めが効率的に行われること、注意してプログラムすればゴミの発生を抑えられるよう配慮している。

1. インタプリタが解釈実行の過程でごみを発生しない。

2. ゴミ集めにコピーを避けることで高いゴミ集めの効率を実現する。

3. Lispで記述しにくい部分は他の言語で書かれたモジュールを積極的に利用する。

4. ユーザがゴミと認識できるオブジェクトは明示的にそれらを回収できるようにする。

非同期の事象に反応するための機構は、扱う問題やオペレーティングシステムに大きく依存する。EusLispでは、UNIXのシステムコールをLispの関数として用意している。非同期のプログラミングを実現する機能には、(1) シグナル(割り込み)ハンドラの定義、(2) ストリームからの非同期入力、(3) インターバルタイマーの設定、(4) セマフォ、などがある。さらに、新たなプロセスの生成、パイプ、メッセージキューを用いたローカルなプロセス間通信、ソケットを用いたネットワークワイドなプロセス間通信の手段が実現されている。

2.3 オブジェクト指向言語としてのEusLisp

Common Lisp[16] 上のオブジェクト指向プログラミング機能は、CLOS(Common Lisp Object System)[4, 18] によって標準化されようとしている。CLOSは従来型のオブジェクト指向言語をさらに発展させ、総称関数、多重メソッド、多重継承、メタオブジェクトなどの概念を新たに導入することで柔軟なオブジェクト指向を追求している。これらの機能の有益性の議論を別にすると、CLOSで問題となるのは、簡単な機構でこれらの機能を効率良くインプリメントできるかどうかという点である。

オブジェクト指向言語では、

1. スロットアクセス

2. メソッドディスパッチ

3. インスタンスの作成

の三つが効率を決定する要因となる [20]。CLOS では、多重継承が原因となってスロットアクセスの、多重継承と多重メソッドが原因となってメソッドディスパッチの性能低下を招きやすく、性能低下を抑えるためにコンパイラの処理が複雑になる。また、リスト処理に偏った既存の Lisp の評価機構を拡張することで CLOS を実現しようとするアプローチでは、インスタンス生成能力においても、十分な性能は期待できない。

これらの問題に対して EusLisp は次のようなアプローチを取っている。まず、効率を低下させ、メソッド選択を複雑にする多重継承に代えて、単一継承を基礎に置く。多重継承が必要な場合のために、メッセージフォワーディングの機能を備える。多重メソッドを採用しない代わりに、階層的な型の包含関係を一定時間で処理可能な型判別機構を実装する。この機構は、EusLisp のカーネル部にも適用できる程度に効率的であり、cons や symbol のような Common Lisp の組み込み (built-in) データ型を標準クラスとして実現することが可能になる。組み込みデータ型をユーザデータ型のスーパクラスに指定することで、システム機能の拡張が図れること、および組み込みオブジェクトの生成・管理を EusLisp のプログラムとして記述できるという大きな利点が生まれる。オブジェクト指向言語の多様な大きさのメモリ割り付け要求に応ずるために、メモリー管理にはフィボナッチバディ法を用いる。Lisp 特有の cons の大きな消費に対しては、メモリーの一定領域を併合しないで残す方策を取る。本手法は高いメモリー効率を得ることが可能であり、コピーを伴わないのでポインタの管理が容易になる。

2.3.1 EusLisp のオブジェクト指向プログラミング

クラスは、型の発展した概念である。Lisp とオブジェクト指向の融合を図る場合、Common Lisp のデータ型との関係を考慮する必要がある。Common Lisp は 30 余りのデータ型を定めており、CLOS はそのうちの 17 について対応するクラスを定めている。残りの型は、Common Lisp が supertype/subtype の関係で表現していなかったり、supertype を特定する優先順位にあいまいな部分が残っているため、クラス間の関係を定められない。`typep`、`subtypep` 等の Common Lisp の関数はクラスを型として扱うことができ、また Common Lisp の型はメソッドの型指定子に使うことができる。

Common Lisp に対応する型を持つクラスは、標準クラス (standard class: defclass で定義される)、構造体クラス (structure class: defstruct で定義される)、組み込みクラス (built-in class: 処理系に依存する方法で定義される) のいずれで実現されてもよい。組み込みクラスには、標準クラスに対して次のような制約が加わる。

1. スロットを持たない、

2. インスタンスの作成にはクラス毎に特定の関数を用いなければならない、

3. 他の標準クラスのスーパクラスに指定することができない。

CLOS が、Lisp としての性能を維持するためには、cons や symbol などの基本データ型を標準クラスとして実装することは損失が大きい。それは、現存するほとんどすべての Common Lisp の関数を defgeneric を用いて総称関数として再定義しなければならないこと、またそれができたとしても今度は実行にメソッドディスパッチを伴い、コンパイラによるインライン展開もできないので効率的に不利になるからである。また、PCL(Portable CommonLoops) のように既存の Lisp の上にオブジェクトシステムを搭載する例では、Common Lisp データ型を標準クラスとすることは現実に不可能である。したがって、実際のほとんどの CLOS は Common Lisp 型を組み込みクラスとして実現している。

これに対して、システムの基本データ型を標準クラスとして定義することには、継承によるサブタイプ化を通じたシステムの拡張と統合化ができること、オブジェクトによる Lisp の基本機能の記述ができること、の 2 つのメリットがある。EusLisp では、数値以外のデータをすべてオブジェクトで表現し、Common Lisp のデータ型をクラスで定義する。この場合、組み込みデータ型は、ユーザが定義するサブクラスによって拡張される可能性がある。それらのオブジェクトに対しても Common Lisp の関数が適用できるためには、継承の階層に則った型判別が必要である。そのため、EusLisp は継承構造を単一継承に限り、効率よく型を判別する方法を実装している。また、単一継承の弱点を補強するため、メッセージフォワーディングの機能を付加し、疑似的な多重継承構造を実現している。EusLisp のオブジェクト指向の唯一の例外は、数値を即値による表現だけに限定し、オブジェクトとして扱っていない点である。これは、EusLisp が目的とする幾何モデラの記述における数値演算の効率を特に重視したためである。

2.3.2 継承を用いたシステムの拡張と統合化

Lisp は、古典的な記号・リスト処理言語から、豊富なデータ型を扱う言語に発展してきており、ネットワーク、ウィンドウなどの環境に対応するために、型追加の要求はこれからも続くものと思われる。Common Lisp の組み込み型をクラスで定義することにより、継承を用いた組み込みクラスの拡張が可能になる。このような型の拡張が有効に機能する例をいくつか上げよう（図 2.2参照）。

Common Lisp のストリームには、ストリームの先にストリングが接続されるか、ファイルが接続されるかの区別がある。出力について考えると、両者ともデータをバッファに詰め込む部分は同じであるが、ファイルストリームはストリームの作成時にファイルの open が伴う点と、バッファが満杯になった時にオペレーティングシステムを通じてバッファの

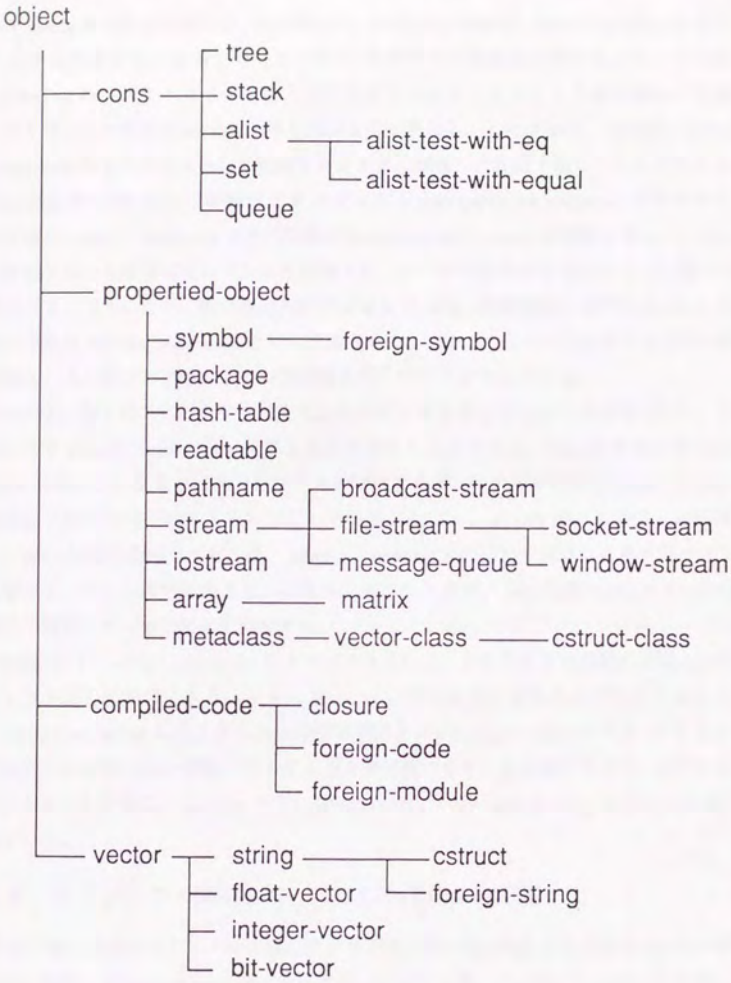


図 2.2: EusLisp の基本データ型の継承構造

内容を flush する点が異なる。したがって、file-stream は、string-stream のサブクラスとするのが妥当である。さらに UNIX でプロセス間通信に用いられるソケットは、read/write のシステムコールを使う点ではファイルストリームと同様の扱いが可能であり、ストリームの作成に connect 等を用いる点が異なる。したがって、socket-stream を filestream のサブクラスとして定義することで、重複した記述を避けることができる。

EusLisp の symbol は、属性付きオブジェクト (propertied-object) のサブクラスとなっており、get、putprop などの関数は propertied-object に定義される。これによって、属性リストの管理がシンボルから分離され、シンボル以外のオブジェクトも属性リストを利用できるようになる。他言語から呼び出せる EusLisp の関数は、通常の symbol のサブクラスである foreign-symbol として定義される。これは、Lisp で使用する通常の関数定義の他に、他言語プログラムとの引数変換を行うコードを含んでいる。

cons は、第 3 のフィールドを加えて逆転ポインタを持ったリストを表現したり、使用目的に応じて alist、set、stack 等を派生させることができる。alist や set は assoc や member 関数による探索を目的としたデータ構造である。これらの関数は:key、:test などの引き数で種々の構造のリストに対応している。そこで、alist 型のコンスには作成時に key、test 関数を登録しておけば、assoc、member を呼び出す時点ではキーワード引き数を省略でき、より汎用性が高まる。これは、ハッシュ表が、test 関数を make-hash-table の時点で指定していることとも整合する。

closure は、compiled-code をスーパクラスにし、レキシカルな環境の情報を加えたオブジェクトとして定義される。また、EusLisp 以外の言語で書かれたプログラムのコードは、compiled-code にリンクのための情報を加えた foreign-code オブジェクトとして作成することにより、Lisp 関数と結合することが可能になる。他言語プログラムのデータ構造にアクセスするために、string の下には cstruct、foreign-string などのクラスが追加されている。

2.3.3 オブジェクト指向をベースにした Lisp の記述

Lisp では、基本的なデータの作成とアクセスは、組み込み関数として実現されるのが普通である。実際、Common Lisp は、リード表、ハッシュ表、パス名、パッケージなど、多くの内部状態を持ったアトムを定義しており、その作成と内部状態へのアクセスのために数多くの専用の関数を用意している。これは、ユーザが定義する structure が、一貫したデータオブジェクト作成法とスロットアクセサを自動的に生成できるのと対照的である。このようになっているのは、実行効率とメモリー効率を確保するために、ポインタとバイナリデータを混在させ、基本データ型毎に異なったデータ構造を持たせており、それらの作成とアクセスは Lisp より下位の言語で記述する必要があるからである。

データ構造をスロットの並びに統一すれば、組み込みクラスを排除でき、標準クラスを用いて一貫したインスタンス作成とアクセスを実現できる。EusLisp ではすべての Common Lisp オジェクトのアクセス関数を EusLisp 自身で記述することが可能である。オブジェクトの作成についても同様にすべてのデータの作成を instantiate 関数だけで行うことができる。

このようなインプリメントでは、car、cdr、rplaca、rplacd、cons などの純 Lisp の基本関数さえも EusLisp の上では全く基本的ではない。これらの関数は、スロットアクセス関数、instantiate 関数を使って次のように定義できる。CLOS が Common Lisp をベースにしてオブジェクト指向機能を付加したのに比べ、EusLisp は、オブジェクト指向をベースにして Lisp を実現していることになる。

```
(defun car (x)
    (if (null x) nil
        (if (consp x) (slot x 'car) (error "not list"))))
(defun rplaca (x a)
    (if (consp x) (setslot x 'car a) (error "not list")))
(defun cons (a d &aux (c (instantiate cons)))
    (setslot c 'car a) (setslot c 'cdr d) c)
```

2.4 EusLisp の構文

2.4.1 クラス定義

クラスの定義は、次の defclass マクロで行う。

```
(defclass classname :super superclassname
                    :metaclass metaclassname
                    :slots (slot-description...))
```

EusLisp は単一継承型なので、スーパクラスは 1 つだけ指定できる。メタクラスとしては、クラス metaclass のサブクラスを指定できる。メタクラスの機能が不必要であれば、デフォルトではクラス metaclass が全てのクラスに共通的に使用される。*slot-description* には、データ型（クラス名）と forward を指定することができる。forward については 2.4.5節で述べる。defclass によってクラスオブジェクト (クラス metaclass のインスタンス) が作成され、それがクラス名シンボルのダイナミック値にバインドされる。属性リスト

クラス名
属性リスト
スーパークラス
クラスid
変数名ベクタ
型ベクタ
フォワードベクタ
メソッドリスト

図 2.3: クラスの構造

に入れることにしなかったのは、クラスオブジェクトは型検査のために頻繁にアクセスされるので、属性リストを探索するのはコストがかかりすぎるのと、2.4.3節で述べるようにクラス名シンボルのダイナミック値を変えることで、インスタンスを作成する関数が参照するクラスを制御するためである。

クラスオブジェクトは図2.3のような構造を持ち、スーパクラスへのポインタ、スロット変数の名前と型を表すベクタ、メッセージフォワーディングの情報、メソッドのリストが入れられる。

2.4.2 メソッド定義

メソッドの定義は、次の defmethod 特殊形式で行う。

```
(defmethod classname
        (methodname1 (lambda-list) . body)
        (methodname2 ...) ...)
```

各々のメソッドの記述は defun による関数定義と同様である。ただし、メソッドの中では、*lambda-list* に書かれたパラメータの他に、スロット変数、self, class が可視になり、各々そのオブジェクトのスロット値、そのオブジェクト自身、そのメソッドが定義されたクラス、にバインドされる。メソッド定義は、クラスオブジェクトのメソッドリストに連結される。

2.4.3 インスタンスの作成

インスタンスの作成には、次の instantiate 関数を用いる。

(instantiate *classname* [*size*])

ベクタ型のオブジェクトの作成には、*size*引数に寸法を与える。instantiateは、スロットに初期値としてNILを詰める。特別の初期化手続きを定義することはできない。例えば、(instantiate cons) の結果は、(NIL . NIL) というドット対になる。list、intern、openなどの組み込み関数がユーザ定義クラスをテンプレートとしてインスタンスを作成するようにするには、次のようにそれぞれspecial変数cons、symbol、streamにユーザのクラスをバインドしておく。

```
(defclass mysymbol :super symbol :slots (myvalue))
(setq x (let ((symbol mysymbol)) (intern "XXX")))
```

オブジェクトの複製を作る方法には、replace-object関数による浅いコピー(shallow copy)とcopy-object関数による深いコピー(deep copy)がある。copy-objectは、オブジェクトのスロットに代入された各々のオブジェクトについても参照のトポロジを維持しつつ再帰的なコピーを行う。このコピーには、オブジェクトヘッダ内のマークビット(図2.7参照)とスタック上に割り付けられたコピー終了表を使用する。新たなオブジェクトがコピーされる度に、ヘッダにマークを記す。そして、旧オブジェクトの第1スロットの値と新オブジェクトのアドレスをコピー終了表にセーブし、旧オブジェクトの第1スロットにはコピー終了表のインデクスを書き込んでおく。マーク済みのオブジェクトに対する別の参照に対してはコピーは行わず、コピー終了表を参照してすでにコピー済みの新オブジェクトへのポインタだけを作成する。コピーが終了すると、もう一度全ポインタをスキャンしてマークを取り除き、第1スロットの値を元に戻す。

このcopy-object関数は、幾何モデラへの応用において形状モデルをコピーしたり、不可逆的な合成操作の前に全状態を保存しておくために不可欠である。このような汎用的な深いコピーを行う関数がないとすると、稜線や面などのモデル要素毎にアドホックにコピー関数を定義しなければならなくなる。

2.4.4 メッセージの送信

メッセージの送信にはsend関数を用いる。

(send *object methodname* [*argument* ...])

*object*のクラスのメソッドリストから*methodname*に相当するメソッドを探索し、それに*argument*を与えて実行し、その結果を返す。*object*にselfを指定することで、自分自

身のメソッドを起動できる。また、send-super マクロによって、あるクラスに定義されたメソッドの中から、さらに上位のクラスのメソッドを探索するよう指示できる。

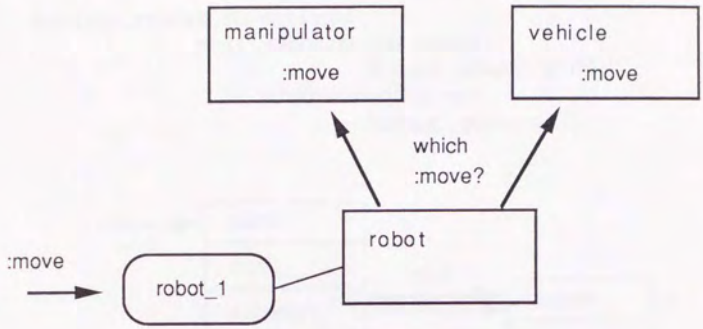
メソッド探索には、すべてのクラスにグローバルなメソッドキャッシュを用いている。メッセージが送信されると、その *methodname* と *object* のクラス名をキーにしてハッシュ表が参照される。そこに該当するメソッドが登録されていればそれが実行される。該当しない場合はクラスに登録されているメソッドリストが線型探索され、見つかるとそれがハッシュ表に登録される。見つからない場合はフォワード指定されたスロット変数のメソッドが探索され、それでも見つからない場合は、:nomethod メソッドが起動される。キャッシュの大きさは 512 である。幾何モデラのアプリケーションでは、99% 以上のヒット率を得ており、10 から 20% の実行速度の改善が達成されている。

2.4.5 メッセージ・フォワーディング

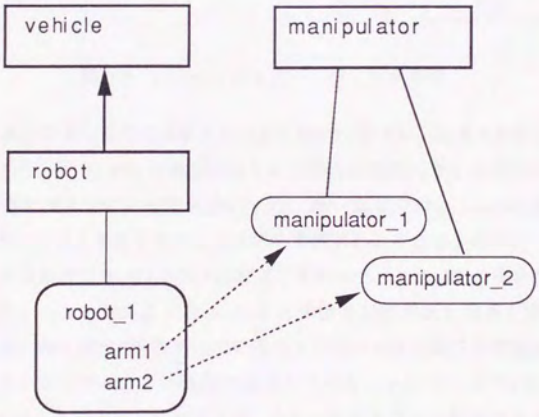
EusLisp の単一継承に対比して、CLOS を含む多くのオブジェクト指向言語は多重継承を許している。しかし、EusLisp の前身となった LEO[67] における多重継承を用いたロボットプログラミングの経験では、多重継承の問題点がいくつか明らかになった。最初の問題は、複数のスーパクラス間でのメソッド名の衝突である。たとえば、robot を、:move というメソッドに反応する vehicle と manipulator のサブクラスとして定義する場合、単純に robot オブジェクトに:move メッセージを送るだけでは、正しい動作を期待することはできない (図 2.4(a))。robot クラスにも:move メソッドを定義し、パラメータに応じてどちらのスーパクラスの:move を起動するかを決めてやる必要がある[4]。次に、要素機能をスーパクラスに実現すべきか、そのインスタンスをスロットから参照すべきかを簡単には決定できない場合がある。上の例では、manipulator は、どちらの方法で定義しても差し支えない。しかし、もし robot が 2 本の腕を持つように定義を変更したとすると、同じクラスを複数スーパクラスにすることはできないので、manipulator は、スロットから参照せざるを得なくなる (図 2.4(b))。このような変更が起こる度に、スーパクラスに定義されていた機能をスロット変数に実現することは大変な労力になる。3 番目の問題は効率である。メソッド選択は、キャッシュの技法を用いることで比較的簡単に高速化が図れるが、多重継承システムでのスロット変数アクセスは、何らかの実行時の表探索を免れない。さらに、2.5.3節に述べるような型検査の方法が適用できないので、あるオブジェクトがあるクラスまたはそのサブクラスから生成されたものであることを検査するのにも探索を伴うようになってしまう。

これらの問題にも拘らず、多重継承型の言語は、多くの問題に有利な特性を持つことを示してきている。Lisp Machine のウィンドウシステム上のすぐれたプログラミング環境が

[4]CLOS では、マルチメソッド (起動される関数が、複数のパラメータの型に応じて決定される) によって解決されている。しかし、メソッドのディスパッチが複雑になることに変わりはない。



(a) メソッド名の衝突



(b) 複数のマニピュレータオブジェクト

図 2.4: 多重継承の問題

```
(defclass person :super object
                 :slots ((name :type string)
                         (age  :type :integer)))
(defclass president :super person
                 :slots ((secretary :type person
                                    :forward (:telephone :mail))
                         (chauffeur :type person
                                    :forward (:go-home))))
```

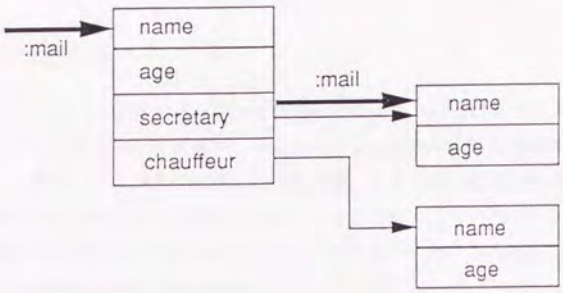


図 2.5: メッセージフォワーディングの例

Flavor[7] の多重継承によるものであることはよく知られている。既存の要素となるクラスを集めてきてソフトウェア IC として組合せることで新たな形式のウィンドウを定義できるので、重複した記述を減らすことができる。また、単一継承では、vector が sequence と array の両方のサブクラスであるといった関係を表現することができない。

通常、クラスの定義後にスーパクラスは変更できないが、スロットの値を変更するのは自由である。つまり、is-a の関係よりも part-of の関係の方が柔軟な構成を取ることができる。EusLisp では、単一継承の簡潔性と高い効率を保ちつつ多重継承の機能を取り入れるため、メッセージフォワーディングの機能を実現している。メッセージフォワーディングとは、あるオブジェクトにメッセージが送られ、それがそのクラスで処理できないことがわかると、:forward 指定されたスロットにメッセージを送り直す機能である。これは、サブクラス、スーパクラスの間で変数を共有しないような多重継承と等価な機能を提供する。図 2.5の例では、president に送られた:telephone と:mail メッセージは secretary に、:go-home メッセージは chauffeur に自動再送信される。フォワードされたメッセージに対してもキャッシュを適用することができれば、効率の劣化も少ない。

メッセージフォワーディングは、4 章、5 章で述べるマルチメディアディスプレイ上のロ

ボットシミュレータが機器独立で稼働するようにするためにも利用されている。マルチメディアディスプレイは、ウィンドウやグラフィクスモデルにid番号を割り当てる。一方、Xwindowなどの一般的ウィンドウシステムはそのような機能を持たず、モデルの管理はすべてアプリケーションプロセスに委ねられる。したがって、MMDを利用するには、EusLispの内部のモデルとMMDのモデルidを対応付ける処理が必要になる。その管理のためにmmdagentと呼ぶ専用のクラスを定義している。MMDを利用するモデルはこのオブジェクトをスロットに持ち、メッセージフォワーディングを指定することで、上位のクラスに変更を加えることなく、何種類かの表示デバイスで同じソフトウェアが動作するようになっている。

2.4.6 スロットアクセス

メッセージがオブジェクトに送られてメソッドの評価が始まると、そのクラスおよびスーパクラスに定義されたすべてスロット変数が、lambdaやletの局所変数と同様の方法で可視になる。スロット位置は、インタプリタによる実行では、クラス内に定義されたスロット変数名表を逐次探索することで見つけられる。コンパイルされると、コンパイル時に定まるオブジェクトの先頭からのオフセットを用いてスロットは直接アクセスされる。このアクセス法に従えば、carの定義は次のように記述することもできる。

```
(defmethod cons (:car () car))
(send '(a b) :car)  --> a
```

スロットにオブジェクトの外部からアクセスすることも可能である。クラスが定義されると、そのすべてのスロットに対して、クラス名とスロット名を連結したアクセスマクロが自動的に生成される。たとえば、(cons-car '(a b))によってconsのcarを取ることができる。setfは、これらのマクロを認識するので、スロット変数は汎変数(generalized variable)として扱うことができる。

スロット名が実行時にしか定まらないときは、slot、setslot関数を用いる。上のアクセスマクロは、実際はこれらの関数に展開される。

(slot *object class slot-name*)
(setslot *object class slot-name value*)

slot、setslotは、第1引数にオブジェクト、第2引数にそのクラスを要求する。クラスは、実行時にオブジェクトから知ることができるので不要であるが、スロット位置を知るために探索が必要となる。*class*と*slot-name*が与えられることにより、コンパイラは表探索をコンパイル時に済ませ、高速なスロットアクセスコードを生成できる。オブジェクト指

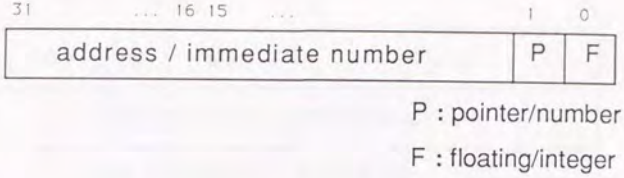


図 2.6: ポインタの構造とタグ

向言語のデータ抽象化の観点からは、スロットに対する外部からのアクセスは不法であるが、Common Lisp の structure の機能との互換性を保つために導入されている。

2.5 オブジェクト、クラスの構造と高速型判別法

2.5.1 ポインタの構造

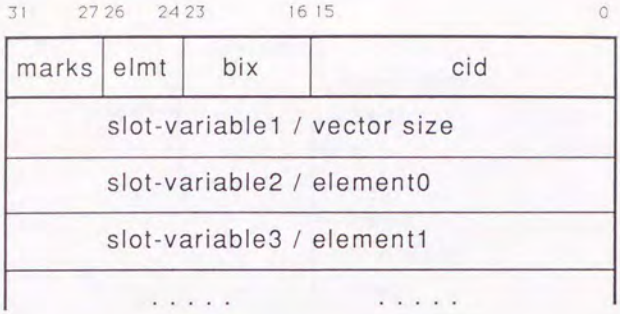
ポインタの表現法は、オブジェクト指向のアプローチおよび Lisp の効率と密接に関係する。オブジェクトの重要な性質に、オブジェクトがどのクラスに属するかを示す情報がオブジェクトの内部に明記されているという点がある。したがって、ポインタ中のタグだけですべてのデータ型を識別する方法は取れない。また、クラスの種類はユーザが定義するにしたがって増加するので、アドレスレンジで型を判別する方法も適用できない。EusLisp では、ポインタ側には数値かポインタかを示す 2 ビットのタグだけを載せ、クラスの情報はオブジェクト内のの class-id によって識別している。

ポインタは 32 ビットの長語で表現される (図 2.6) 。メモリセルは長語境界に置くこととし、上位 30 ビットをアドレスまたは数値、下位 2 ビットをタグに用いる。タグは、00 が整数、01 が浮動小数、10 がアドレスを意味し、11 は使用していない。数値の長精度形式はない。このポインタ表現には次の特徴がある。

1. 4 Gbyte の空間をアドレスできる。

2. 即値で $\pm 10^9$ の整数と、10^{-6} 程度の浮動小数の計算機エプシロンを表現でき、ロボットの実用上十分な精度が確保できる。したがってメモリ消費の大きい長精度形式を用いる必要が少ない。

3. タグを取り除く操作が簡便である。すなわち、オブジェクトにアクセスするときは -2 のオフセットを加えるだけでよく、整数の加減算ではタグを無視できる。



marks : b(buddy), m(memory), g(gc), p(print), c(copy)

elmt : element-type
0 : non-vector 1 : bit 2 : byte 3 : char
4 : integer 5 : floating 6 : foreign 7 : pointer

bix : buddy-index

cid : class-id

図 2.7: オブジェクトの構造

2.5.2 オブジェクトの構造

オブジェクトは、図 2.7に示すような 1 語のヘッダとそれに続くスロットで構成される。ヘッダの各フィールドの意味は図中に示されている。組み込み型のオブジェクトもユーザのオブジェクトも同一の構造であるため、一貫したメモリー管理が可能になる。

オブジェクトは、図 2.8に示すように非ベクタ型とベクタ型に分けられる。非ベクタ型オブジェクトは、スロットの数が固定 (クラスが定める) であり、各スロットはスロット変数名でアクセスされる。ベクタ型は要素の数が可変 (instantiate の時点で定まる) であり、ヘッダの次の語に要素の数を示す数値が入れられ、要素はインデックスでアクセスされる。ヘッダの elmt フィールドが要素の型を表す。ベクタ型オブジェクトだけがポインタ以外のバイナリデータを保持できる。ビット、文字、整数、浮動小数などのバイナリデータの表現は、C などの他言語で書かれたプログラムと互換性があり、相互にデータの受け渡しができる。bix (buddy-index) は、このメモリセルの物理的な大きさを表す指標であり、mark フィールドと共にメモリ管理に用いられる (2.6節参照)。

このオブジェクト構造で注目すべきポイントは、オブジェクトが属するクラスが、ポインタではなく 16 ビットの cid(class-id) として表現されていることである。cid は、実際のクラ

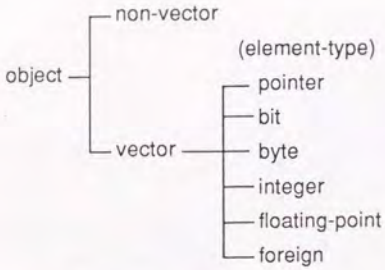


図 2.8: 要素の並びと型によるオブジェクトの分類

スオブジェクトへのポインタが配列されたクラス表へのインデクスを表す (図 2.9参照)。cid 表現により、全オブジェクトがクラスへのポインタを持つことによる無駄な記憶の消費が無く、また、次節で述べるような cid の大小関係を用いた型の包含関係の判別が可能になる。

2.5.3 高速型判別法

Common Lisp は複雑な型の階層を定めている。この階層は、一般的な型に対して定義された関数は、その副型に対しても適用されるが、副型に定義された関数は上位の型には適用できないような関係を保っている。したがって、関数をデータに対して作用させる時には、そのデータの型が、目的とする型またはその副型と一致するかどうかの判定が必要になる。この型判別は処理系の性能に大きく影響する [9]。

多くの CLOS のインプリメントに見られるように、組み込み型がユーザによって拡張されることがない場合は、ポインタとオブジェクトのタグのエンコード法を工夫して最大の効率が得られるようにすることができる。しかし、EusLisp のように基本型の動的な拡張を許す場合には、たとえば `stream` のユーザ拡張クラスである `socket-stream` に対しても `read`、`print` を適用できるようにするためには、タグを静的に固定することはできない。純然たるオブジェクト指向型言語のように、型の判定をメッセージ送信に伴うメソッドの探索に頼る方法も考えられる。CLOS が、総称関数を経由して特定のメソッドにディスパッチするのも、キャッシュ等の改善策はあるにせよ、基本的に探索が伴う処理である。ところが、メソッド探索はオブジェクト指向言語の効率を定める第 1 の要因であり、コンパイラがプリミティブ関数をインライン展開することによって速度の向上を図っている Lisp との間には大きな効率の差が生じる。EusLisp では、型の階層を単一継承に限り、クラスへの参照が class-id としてコード化されていることを利用して次のような効率的な型判別を行っている。

型の判別は、あるオブジェクトが、(1) あるクラス、(2) またはそのサブクラスから、生成されたことを検査する問題である。オブジェクトを生成したクラスはオブジェクト自身に

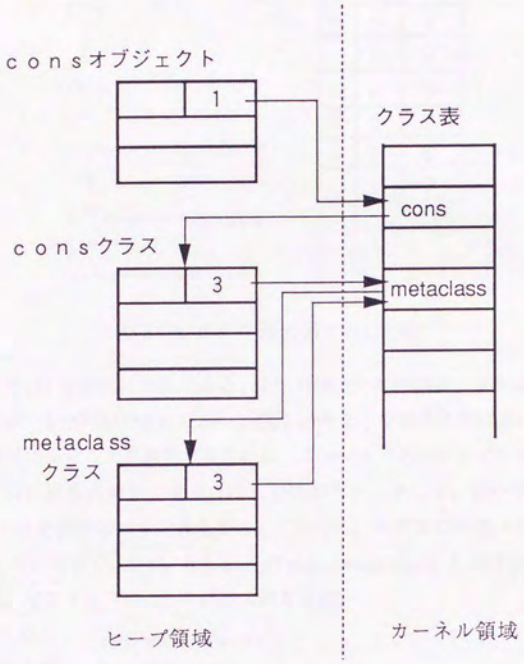
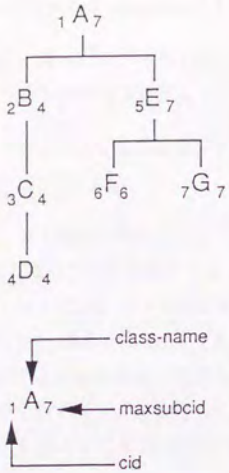


図 2.9: クラステーブルを介したクラスの参照



Class-Table

.	cid	max-subcid
A	1	7
B	2	4
C	3	4
D	4	4
E	5	7
F	6	6
G	7	7
...	...	...

図 2.10: クラス id の割り付けの例

書かれているので (1) は容易に判別できる。(2) の検査のためには、クラスが他のクラスのサブクラスであることが判別できれば良い。図 2.10のような継承構造において、たとえば D が A のサブクラスであることを確認するために、スーパクラスのチェインを D、C、B、A と辿る方法は、特に継承木が長くなった時に能率が悪い。そこで、各クラスには継承木に沿って連続した cid を割り付けることとする。すなわち、クラス C の直下のサブクラスに適当な順序を付けたものを $C_1 \ldots C_n$ とし、その cid、 maxsubcid を以下のように定める。maxsubcid とは、サブクラスの cid 中の最大値を表す。

サブクラスなし　$C_{\cdot maxsubcid} = C_{\cdot cid}$

サブクラスあり　$C_{1 \cdot cid} = C_{\cdot cid} + 1$

$$C_{i+1 \cdot cid} = C_{i \cdot cid} + 1$$

$$C_{\cdot maxsubcid} = C_{n \cdot maxsubcid} + 1$$

このようにして定まる cid と maxsubcid をクラス表に記録しておく。オブジェクト x が C に属することを検査するためには、x の cid が C の cid と maxsubcid の間にあるかどうかを調べればよい。すなわち、$C_{\cdot cid} \leq x_{\cdot cid} \leq C_{\cdot maxsubcid}$ を判定すれば良い。

上記の漸化式の関係を保つため、新しいクラスが登録されると、既存の各クラスおよび全てのオブジェクトの cid の再割り付けが必要になる。例えば、図 2.10の F の下に新たに F' を定義する場合、次のような処理が必要になる。

1. F' の cid として F の maxsubcid+1 を割り付ける

2. F' のすべてのスーパクラスの maxsubcid を 1 増やす

3. F' の cid 以上の cid を持ったクラスの cid、maxsubcid を 1 つずつ増加させると共に、クラス表のエントリを 1 つずつ下にずらす

4. システム中の全オブジェクトを走査して、F' の cid 以上の class-id をすべて 1 増加させる

特に (4) の処理には、ゴミ集めと同様に、オブジェクトの総量に比例した時間を要する。例えば、cons の下に新たなクラスを定義すると、cons の cid はシステム中で最小であるため、cons 以外のすべてのオブジェクトの cid が再計算される。その時間を測定したところ、ゴミ集めに要する時間の 15% から 20% であった。これは、クラスを動的に定義するようなアプリケーションにとっては致命的な問題となり得る。しかし、ロボットではそのようなプログラミングは極めて稀である。定義時にこのような手間をかけておくことで、実行時の型判別がインライン展開によって高速化されるメリットの方がはるかに大きい。実際、次節で述べるように、クラス A がクラス B のサブクラスであるかどうかを調べる subclassp、オブジェクト x がクラス A またはそのサブクラスから導出されたかどうかを調べる derivedp 関数は、継承の深さによらず一定の時間で実行することができる。

2.5.4 型判別の性能

Common Lisp の最も基本的な型判別述語は、consp、symbolp などの組み込み型を判別するものである。また、ユーザが定義する structure に対しては、その structure 名に -p をつけた型識別述語が自動的に生成される。EusLisp の symbolp、consp などは、ユーザによって定義された symbol、ons クラスのサブクラスの判別にも利用できる。また、derivedp 述語は、ユーザ型の判別述語を定義するために用いられる。さらに 2.5.3節でで述べたように、オブジェクト指向型言語ではメッセージ探索によっても型が判別できる。以上の、symbolp、structure-p、derivedp、send の処理時間を表 2.1に示す。実行マシンには Sun3/60 を用い、全てコンパイルされたプログラムの実行速度を測定した。比較の対象には、同じマシンの上で動く代表的な Common Lisp の処理系である KCl[49] を選んだ。時間の測定には get-internal-run-time 関数を用い、CPU の負荷、実メモリの量やごみ集めの差の影響がなるべく出ないように留意した。[5]

symbolp は、EusLisp でも KCl でも in-line 展開されている。KCl では、組み込み型の検査はポインタが指している先のタグの等値を調べるだけで良いのに対し、EusLisp では数値かポインタかの判定とレンジチェックに 2 回の比較が必要なので約 1/2 の速度になっている。もし、ここで M68020 の持つ CHK2 のようなレンジチェック命令が利用できればこ

[5] この節以降に現れる性能比較はすべてこの条件で測定された。

	EusLisp	KCl
symbolp	3.1	1.4
structure-p (1 level)	25.	48.
(derivedp) (2 levels)	25.	77.
(3 levels)	25.	105.
send	80.	–

表 2.1: 型判別の性能
(単位は $\mu sec.$)

の比率はさらに改善できる。derivedp は in-line 展開されないのと、引き数の型検査が必要になるために、組み込みの型判別に比べると性能が低下する。structure の型検査には、defstruct に:include オプションを用いて a(上位)、b(中位)、c(下位) の 3 段の structure を定義し、c のインスタンスが a、b、c のそれぞれに属するかどうかを検査するプログラムを用いた。組み込み型の検査と比べると数十倍の時間を要している。EusLisp の基本型判別述語、derivedp は継承が何段になっても同じ時間で処理が終わるが、KCl では継承が深くなるほど時間がかかるようになるのが判る。send は derivedp に比べてかなり遅くなるが、メソッドキャッシュがヒットしている限り、継承の段数に関係無くやはり一定の時間で処理される。

2.6 メモリ管理

オブジェクト指向型 Lisp には、多くの種類のオブジェクトのための多様な大きさのメモリ要求が頻繁に生ずる一方で、Lisp のプログラムの実行ではやはり cons セルの消費が非常に多いという特性がある。図 2.11に、EusLisp で幾何モデラのアプリケーションを実行した時に発生したメモリ要求の回数と量の表を示す。両方の問題に対処するため、EusLisp では変則型のフィボナッチバディ法を用いたメモリ管理を行っている。

2.6.1 フィボナッチバディ法のアルゴリズム

バディ法の基本アルゴリズムは次のように記述できる（図 2.12参照）。

1. メモリが要求されると大きなセルを分割し、決まった大きさのセルの中から要求を満たす最小のセルを返す

2. ごみ集めによって不要なセルが見つかると、その前後のバディセルを検査し、それらも不要なセルであれば併合する、可能ならば併合を再帰的に繰り返し、より大きなセルにして回収する

```
MACHINE
   sun3/60 with 8MB memory
CPU-TIME
   21:12
GARBAGE COLLECTION
   98 times, 97sec. for marking, 74sec. for sweeping.
MEMORY ALLOCATION
   3.3MB allocated virtually, 2.7MB resident
   2.9MB for heap, 0.7MB free
   totally, 51MB of memory is requested
   in average, 42KB is allocated each second.
```

buddy	size	free	total	total-size	wanted	wanted-size
1	3	14975	68407	205221	1312183	3936549
2	6	14182	30710	184260	1295922	7775532
3	9	251	10093	90837	97124	874116
4	15	11	290	4350	3083	46245
5	24	2	15	360	115	2760
6	39	0	9	351	45	1755
7	63	0	3	189	14	882
8	102	0	14	1428	109	11118
9	165	2	6	990	15	2475
10	267	1	9	2403	15	4005
11	432	1	3	1296	8	3456
12	699	76	77	53823	8	5592
13	1131	0	6	6786	8	9048
14	1830	0	2	3660	5	9150
15	2961	0	11	32571	13	38493
16	4791	0	5	23955	5	23955
17	7752	0	5	38760	5	38760
18	12543	0	2	25086	2	25086
19	20295	0	1	20295	1	20295
20	32838	0	1	32838	1	32838
21	53133	0	0	0	0	0
22	85971	0	0	0	0	0
23	139104	0	0	0	0	0
24	225075	0	0	0	0	0
25	364179	0	0	0	0	0
26	589254	0	0	0	0	0
27	953433	0	0	0	0	0
28	1542687	0	0	0	0	0
29	2496120	0	0	0	0	0

図 2.11: 幾何モデラアプリケーションにおけるメモリ割り付け要求の頻度分布

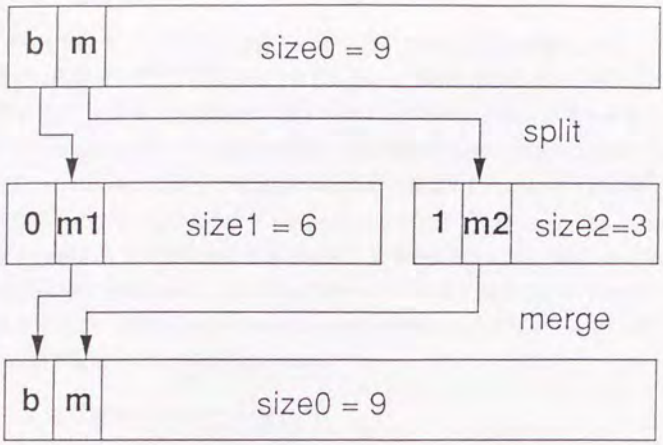


図 2.12: フィボナッチバディ法

分割と併合が簡単に処理できるよう、バディ法ではセルの大きさを 2 のべきやフィボナッチ数などに制限する。前者をバイナリバディ、後者をフィボナッチバディと呼ぶ [8, 36]。

フィボナッチバディの実用的なアルゴリズムは文献 [8] に詳しいが、ここで概略を述べておく。フィボナッチバディでは、異なった大きさのセルがマージされるので、セルの大きさを示すフィールド bix の他にセルの大小関係を表す 2 ビットのタグ b、m が必要である (図 2.7、図 2.12)。$bix = n$ のセルを分割するには、$bix = n - 1$ のセルの b を 0 に、$bix = n - 2$ のセルの b を 1 にセットする。また、親セルの b、m フィールドを子供の m フィールドにセーブする。ごみセルを回収する時は、$b = 0$ であればその右側のセル、$b = 1$ であれば左側のセルを調べ、それが使用されていなければ併合して回収する。そのとき 2 つの m フィールドから親セルの b、m を復元する。

フィボナッチバディはバイナリよりセルの大きさの種類をたくさん用意できるのでメモリ効率が良くなる。Peterson と Norman[36] は、バイナリバディでは約 30% の内部ロス（セルの内部に生ずる無効領域の比率）、フィボナッチ・バディでは約 22% の内部ロスを予測している。EusLisp では、5bit の bix で、図 2.11の第 2 列に示すような 32 種類、最大約 24Mbyte までのセルを実現している。実測では、10 から 20% 程度の内部ロスであることが観測された。

2.6.2 マージの抑止

フィボナッチバディの問題は、分割・併合の手間である。Lisp では cons のような小さなセルの消費が圧倒的なので、ゴミ集めで大きなセルに併合されたとしても割り付けの際には再び極限まで細分化される処理が繰り返されることになり、分割・併合の手間は小さくない。そこで、EusLisp では、ごみ集めの際に一定の割合のメモリは併合しないような管理を行っている。これによってメモリの一定量は、実行の経過にしたがって次第に最小サイズのセルで埋め尽くされる。この最小サイズは cons セルの大きさに等しく、従って cons の割り付けはセルの分割を伴わず高速に処理される。併合されないで残されるメモリの比率は `*gc-merge*` 変数で制御される。この値はダイナミックに変更可能であり、cons の消費が多い場合は大きめに、多様な大きさのオブジェクトが生成されるプログラムでは小さめに設定しておけば効率的なメモリ管理が実施される。

2.6.3 メモリ割り付けとごみ集め

EusLisp のメモリは図 2.13のように割り当てられる。EusLisp は Unix のプロセスとして実行される。Unix のプロセスは、静的かつ共有可能なテキスト、プロセスに固有のデータ、スタックの三つのセグメントからなる。テキスト領域には、約 10,000 行の C プログラムで記述される raweus が入る。オブジェクト、EusLisp がロードするコンパイルされたコード、変数束縛用スタック (バインドスタック) は、すべてデータセグメント内に、EusLisp のメモリマネジャを通して割り付けられる。メモリマネジャは、C の malloc 関数を呼んで大きなメモリのかたまり (チャンクと呼ぶ) を獲得する。データセグメント内には、C のライブラリ関数が割り付けるメモリ領域も散在する。すべてのチャンクは Fibonacci 数の大きさでありチェイニングされている。このチャンクリストが EusLisp のヒープを形成する。各チャンクが分割されてオブジェクト (コンパイルドコードを含む)、スタックに割り付けられる。

獲得済みのチャンク内の領域がすべてオブジェクトに割り付けられた状態で新たなメモリ要求が発生すると、まずごみ集めが起動される。ごみ集めには、古典的なマーク・アンド・スイープ法を採用している。まず、すべてのチャンク内の分割されたセルを走査して必要な (ごみでない) セルのヘッダにマークを付ける。必要なセルには、3 種類ある。一つは、システムの外部から参照される可能性のあるものであり、それはパッケージにインターンされたシンボル以外にない。システムは作成されたパッケージ (高々数十個) を記憶しており、これらのパッケージから参照されるオブジェクトにすべてマーク付けする。二つめはクラスオブジェクトである。各オブジェクトは、クラスをポインタによって直接参照しているわけではないので、単純にポインタをトラバースするだけでは全クラスがマーク付けされるとは限らない。そこで、内部のクラス表をルートにしてマークをつける。三つめは、式の評価の途中で生じる局所変数から参照されるオブジェクトである。これらはすべてバインドスタックに

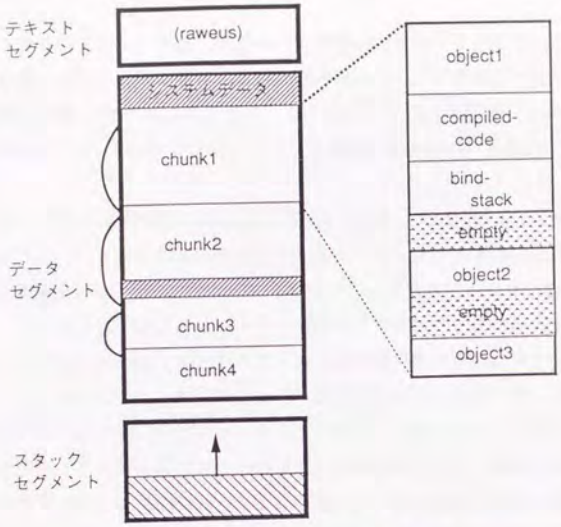


図 2.13: EusLisp のプロセスイメージ

積まれているので、これをルートにしてマーク付けを行う。したがって、バインドスタックにはポインタと見間違うようなバイナリデータを記憶しておくことはできない。

マーク付けが完了すると、全チャンクを走査して、マークのないセルを回収する。空きセルが見つかると、そのヘッダの bix、b、m ビットを参照して、前後のセルとの併合を試みる。ただし、全空きセルの容量が `*gc-merge*` に達するまでは併合を抑止する。ヒープの走査と併合が終了すると、全ヒープと空き領域の大きさを比較し、一定量 (約 25%) 以下の空き領域しか回収できなかった場合は、新たなチャンクを要求し、チャンクリストに加える。

2.6.4 BiBoP 法、コピー法との比較

Lisp でよく用いられるメモリ管理法に、BIBOP(Big Bag of Pages) 法 [17, 49] とコピー法がある [9]。BiBoP 法は、メモリーを多くのページに分割し、1 つのページの中は同じ型もしくは同じサイズのオブジェクトだけになるように管理する。BiBoP 法の特徴は、組み込み型の cons、symbol などのメモリ割り付けはフリーリストからのデキュー操作だけで行えるのでたいへん高速であり、内部フラグメンテーションが生じないこと、またセルが割り付けられたページから、そのオブジェクトの型を知ることができる、などの点である。ところが、これらの利点は EusLisp のようなオブジェクト指向システムにはあてはまらない。組み込み型は拡張されて任意の大きさになり得るし、ユーザが定義するクラス毎にページを用意するのは非現実的だからである。また、BiBoP 法でもベクタのような不定長オブジェクトに

対しては内部ロスを生じるし、特定の型のページが使い尽くされたのに他の型のページに多くの空き領域が残ることによる外部ロスはかなりの量に上る。バディ法では異なった大きさのセルの間で融通がきくので外部ロスがほとんど常にゼロになる。バディのかたまり毎にページを割り付けるので、ダイナミックにメモリを獲得できる点は、BiBoP 法もバディ法も同じである。

コピー法では、空き領域が連続した大きな領域となるような圧縮処理 (compaction) が行われるので、オブジェクトの大きさに拘わらず、メモリ割り付けは簡単に処理される。しかし、コピー用の領域を別に確保しなければならないので、本質的にメモリ効率は 50% を上回ることはない。コピー法は圧縮によって実行時のページング特性を改善するが、ごみ集めはコピーとポインタの付け替え操作が必要になるので余り効率的でない。ポインタの付け替えはスタックやレジスタ中のポインタにもおよび、管理が複雑になる。一方、バディ法ではオブジェクトのアドレスは不変なので、コンパイルドコードも同一のヒープに割り付けることができ、他言語プログラムやライブラリとのリンクが容易に行える。参照が無いことが確認できればメモリをシステムに返却できることを利用して、ごみ集めの発生を抑えることも可能である。バディ法はすべてのページを均質に管理できるのでアルゴリズムの記述が簡単で済む。実際、EusLisp のメモリ管理部はごみ集めも含めて僅か 250 行の C プログラムで実装できた。

2.6.5 メモリ管理の性能

プロセスに割り当てられたヒープメモリの量を同程度にして、cons、長さ 10 のストリング、長さ 3、10、30 のベクタを作成する時間を測定した。EusLisp は、cons だけに限らず多様な大きさのオブジェクトの割り付けに於いて良好な性能を示している。表 2.2の時間は、1 万個のオブジェクトの生成時間から求めており、ごみ集めの時間も含まれている。EusLisp の性能が優れている理由の一つは、ごみ集めの起動される回数が KCl より少ないためである。これは、空きメモリの総量が同一であっても、KCl ではオブジェクトの種類に応じて決まった量のページを割り付けているので、そのページの中で空きが無くなるとたとえ他の種類のページに空きがあってもごみ集めを行わねばならないのに対し、EusLisp ではすべての空き領域をすべてのオブジェクトの割り付けに使用できるからである。ゴミ集めに要する時間は、ほぼヒープの大きさに比例し、約 1 秒／ 1Mbyte の性能が観測された。

2.7 コンパイラと実行性能

Lisp はインタプリタによる解釈実行を一つの特色にしているが、インタプリタによる実行では柔軟性と引換に効率的に大きなロスが生じる。ロボット研究が大規模になるに従って、ソフトウェアの積み重ねと統合のためにコンパイラは不可欠となりつつある。

	EusLisp	KCl
cons NIL NIL	18.4	14.0
make-string 10	103.	380.
vector 1 2 3	80.	860.
vector 1 ... 10	420.	2070.
vector 1 ... 30	820.	6400.

表 2.2: メモリ割り付けの性能
(単位は $\mu sec.$)

2.7.1 コンパイラの効果

コンパイラの最大の目的は実行速度の改善である。そのために、コンパイラは次の処理を行う。

ローカル変数のバインドとアクセスの効率化 インタプリタによる実行では、ローカル変数はスタック上の alist を使った深い束縛によって値を表現する。変数のアクセスには alist の探索を伴う。コンパイルすることによってローカル変数はスタックポインタからの変位で直接アクセスできるようになる。

関数呼びだしの効率化 インタプリタによる実行では、関数呼び出しは必ず funcall を経由する。コンパイラは、目的とする関数を直接呼び出すコードを生成するので、関数呼び出しのオーバヘッドは引数の数の検査だけになる。

特殊形式の直接実行 setq、let、block などの特殊形式は、プリミティブを直接に実行する形式にコンパイルされる。block ラベルのサーチはコンパイル時に行われ、実行時にラベルのリストを管理する必要がなくなる。

コンパイル時のマクロ展開 do、case、prog など全てのマクロはコンパイル時に展開されるので、そのための実行時の展開処理とメモリ消費がゼロになる。

基本組み込み関数の in-line 展開 symbolp、consp などの型判別述語、car、cdr 等のリスト要素アクセス、svref、char などの配列アクセス関数、1+、1-、logand、logior 等の単純数値演算関数を展開する。これらの関数は、引数の型を限定しているので、型宣言がなくても展開できる。

型宣言を利用した変数アクセスと数値計算の最適化 elt、aref などの一般的な列と配列アクセス、+、-、* などの一般の数値に対する演算は、declare 特殊形式による型の宣言があれば in-line 展開できる。

```
(defun fib (n)              ; 宣言なし
  (if (< n 2) n  (+ (fib (1- n)) (fib (- n 2)))))

(defun ifib (n)             ;宣言付き
  (declare (fixnum n))
  (if (< n 2)
      n
      (+ (the integer (ifib (1- n))) (the integer (ifib (- n 2))))))
```

	インタプリタ (i)	コンパイラ (c)	改善率 (i/c)
宣言なし	13.4	1.32	10.5
整数宣言つき	15.9	0.25	64

表 2.3: フィボナッチ数 (fib 20) の計算時間とコンパイラの効果
(単位は秒)

表2.3は、フィボナッチ数の計算を、インタプリタとコンパイラで比較したものである。これらの処理によって、コンパイルされたプログラムは、インタプリティブな実行の数倍から数十倍の性能を発揮できることが分かる。

2.7.2 オブジェクトによるコンパイラの実現

EusLisp のコンパイラは、構文解析を行う部分と、コード生成部の二つのクラスに分けて実現している。構文解析部は、S 式を読み込んで、変数のアクセス、関数呼びだし、クロージャの生成、条件分岐などの、数十種類のプリミティブな操作に分解し、それをコード生成部にメッセージとして送る。コード生成部は、このメッセージに対応するマシン語を生成する。

最初のコード生成部は、EusLisp の最初の実現機械であった M68020 プロセッサのアセンブリコードを生成し、Unix のアセンブラを呼び出して機械語を得ていた。その後、Vax、Sparc などに EusLisp を移植する必要が生じた時点で、このコード生成部は、C プログラムを生成するように書き換えられた。このために、コンパイル速度は犠牲になったが、EusLisp の移植性は飛躍的に高まった。構文解析部との共有データが少なく、インタフェースがメッセージとして明確に分離されていたので、コード生成部の書き換えは容易であった。

構文解析部がオブジェクトとして実現されていることによって、コンパイルの途中での環境の保存が簡単になっている。Lisp では、一つのグローバルの関数の中に、クロージャあるいは flet、labels 特殊形式によって、局所関数が持ち込まれることがある。すなわち、関数の定義が静的にネストする。しかし、EusLisp の目的言語である C にはこのような関数のブロック構造を定義する機能がない。このために、コンパイラは局所関数が定義された時点

での環境をその大域関数の外部にまで保存し、別の大域関数として定義する必要がある。この環境には、ブロック、変数・関数バインド、宣言、などがある。コンパイラは、2.4.3節に述べた copy-object 関数によって簡単に環境をコピーすることができる。

2.7.3 ベンチマーク性能

Gabriel のベンチマーク [9] を用いて、EusLisp の Lisp としての総合的な性能を測定した。測定を行った結果を表 2.4に示す。それぞれ、Tak は再帰呼び出し、Boyer はリスト処理、Derivatibe はコンスの生成速度、FFT は浮動小数点演算、Puzzle は配列のアクセス、Qsort は列のアクセスの性能の指標になっている。

表から判るように、EusLisp はほぼ KCl に匹敵する性能を得ている。Puzzle、Derivative で EusLisp が遅いのは、コンパイラの最適化機能が十分でなく、末尾再帰のループ化などの処理を行っていないためである。FFT で EusLisp が高い性能を示しているのは、浮動小数が即値で表現されているのでメモリ割り付けの処理が不要なためである。幾何演算を高速化すると言う、EusLisp の目標はほぼ達成されていると言える。

3.4.5で述べるように、Common Lisp としての機能を実現するための KCl と EusLisp のソースコードの量を比較すると、図 3.15に示すように、EusLisp は KCl の約 1/3 の記述量で済んでいる。もちろん、コンパイラの質の差、コメントの量、EusLisp に欠落する Common Lisp の機能[6]、幾何演算プリミティブなどで記述量に差が出るのは当然であるが、ソースコードの各部分を比較してみると次のような EusLisp の特徴により記述量が減少していることが確認できた。

- 単一継承により型の検査が簡単になっていること
- 全データをオブジェクトのスロットとして統一的にアクセスできるのでシステムコードの多くの部分を Lisp で記述できること
- 全オブジェクトを単一のヒープに均一に割り付けることによってメモリ管理が簡単になっていること

このように記述量が少なくて済むことは、システムの保守を容易にし、周辺の技術革新に応じて拡張を施すにも都合が良い。

2.8 幾何演算機能

Lisp のロボット応用では、幾何学的演算が頻繁に登場する。Common Lisp では、算術プリミティブ以外の幾何的演算機能を定義していない。これらを Lisp で記述するのでは、記

[6]Common Lisp の機能のうち、多値、多倍長整数、複素数、upward-funarg 問題に対応できる関数クロージャなどが EusLisp には欠落している。

	EusLisp	KCl
Tak	2.9	2.3
Boyer	35.2	41.6
Derivative	11.5	6.6
FFT	47.9	60.1
Puzzle	80.3	53.3
Qsort	1.5	1.8

表 2.4: Gabriel のベンチマーク
(単位は秒)

憶、実行効率上問題が生ずる。EusLisp では、浮動少数点数を要素とするベクタ型、マトリクス型およびその間の演算プリミティブを組み込みにすることで、高い効率を発揮する。

2.8.1 ベクタ、マトリクスの表現

Common Lisp は、配列、ベクタ、文字列、シーケンスを定義している。これらを拡張して、幾何計算用のデータ構造を定義する場合にも、これらとの互換性が保たれるような配慮が必要である。

Common Lisp の配列(array)は、7次元までの任意の次元数を持つことができる。1次元の配列をベクタ(*vector*)と呼ぶ。ベクタは列(*sequence*)の副型でもあり、elt、length、reverse、concatenate、remove、map、find、position、copy-seq、subseq などの操作が、リストと同様に適用できる[7]。配列、ベクタは任意のデータ型のオブジェクトを要素に持てる。データ型が限定されていない配列は一般型(*general*)であると言われる。これに対して、要素の型が数値、文字、ビットのいずれかに特定された配列は特殊型(*specialized*)であると言われる。いずれについても同一の関数(elt あるいは aref)による一貫したアクセスが行なわれる。配列は、他の配列に *displace* することで内容を共有したり、異なった次元の配列としてアクセスすることができる。また *adjustable* を指定することで配列の寸法を動的にに変更することが許される。さらに、ベクタは *fill-pointer* を用いることで、スタックとして扱うことができる。Common Lisp は、効率化のためにこれらの機能を排除した配列を用意することも要請しており、そのような配列は単純(*simple*)であると言われる。read、print の書式は、一般型の配列、一般型のベクタ、文字型の単純ベクタ(ストリング)にだけ、#nA(...)、#(...)、"..." として定義されている。ただし、単純であるかどうかによらず print の書式は同じなので、read によっ

[7]旧来の Lisp では、ベクタとリストは全く別の型であり、ここに上げた関数はベクタあるいはリストの専用の関数であった。Common Lisp では、両者に適用可能な generic 関数として定義されている。

て単純でない配列が作られることはない。文字型配列 (ストリング) は equal で等値性が判定できるが、それ以外のベクタ、配列は判定できない。

特殊型、単純型の配列は、一般型、非単純型の配列があればそのセマンティクスが変わることはないので、実際にそれらが完全にインプリメントされることはまれである。たとえば KCl が特殊型のベクタとして認知できるのは単純型の文字ベクタ (単純ストリング) だけであり、たとえ make-array の:element-type に float 等を指定したとしても一般型の配列が生成される。これは、Common Lisp の仕様には適合するが、浮動小数ベクタを数多く生成する幾何モデラへのアプリケーションではメモリ、実行の両方の効率に問題が生ずる。また、2.9.1節で述べるように、EusLisp は他言語プログラムとのリンクの機能を重視しており、整数、浮動小数ベクタの表現が他の言語と異なることは、それらのプログラムとのデータの共有と受渡しに障害となる。したがって、EusLisp が拡張すべき点は次のようになる。

1. 整数、浮動小数を要素とする特殊化された単純ベクタを用意する。

2. 特殊化された単純ベクタのデータ表現は他の言語と互換性のあるものとする

3. それらの read、print の書式を定義する。

4. 浮動小数のベクタおよび 2 次元配列に対して幾何演算プリミティブを用意する。

インプリメントの方法としては、すべての単純ベクタに要素の型を表す 3bit のタグ (elmt) を付加する (2.5.2節参照)。このタグは、オブジェクトヘッダに含まれ、これにベクタの寸法を表す 1 ワードを加えても単純ベクタのメモリ消費は C、Fortran などの言語と比べて 2 ワード増に抑えられる。単純ベクタのデータの並びは、C 言語の char、long、float の配列と同じ構造になっている。非単純ベクタあるいは 2 次元以上の配列を表現するためには、array オブジェクトを用いる。図 2.14に示すように array オブジェクトは、配列の実体を表す単純ベクタと共に、配列の寸法、fill-pointer、displaced-offset 等を保持する。read、print の書式としては、整数型の配列には #nI(...)、浮動小数型の配列には #nF(...) を用いる。n は次元数を表す。

2.8.2 幾何演算プリミティブ

要素型が浮動小数に特定 (specialized) されたベクタをフロートベクタ (float-vector)、またその 2 次元配列をマトリクス (matrix) と呼ぶ。EusLisp が、組み込みで備えているフロートベクタおよびマトリクスの間の演算プリミティブの代表的なものを表 2.5に掲げる。

これらの関数の多くは、関数の値としてフロートベクタかマトリクスを返す。もし、演算に伴って常に新たなメモリが割り付けられるとすると、幾何演算中にごみ集めが頻発し、実時間性の高いプログラムを開発することが困難になる。そこで、EusLisp では、最後の引数

`float-vector` *f1 f2 ...*	フロートベクタの生成
`v+` *v1 v2* [*rv*]	二つのフロートベクタの加算
`v-` *v1* [*v2*] [*rv*]	フロートベクタの減算、または反転
`v.` *v1 v2*	二つのフロートベクタの内積
`v*` *v1 v2* [*rv*]	二つのフロートベクタの外積
`v.*` *v1 v2 v3*	三つのフロートベクタのスカラ 3 重積
`scale` *s v* [*rv*]	フロートベクタのスカラ倍
`norm` *v*	ノルム
`normalize-vector` *v* [*rv*]	正規化
`distance` *v1 v2*	ベクタ間の距離
`transform` *m v* [*rv*]	マトリクスによるフロートベクタの変換
`unit-matrix` *rank*	正方単位行列の生成
`make-matrix` *n m*	n 行 m 列の行列の生成
`transpose` *m* [*rm*]	マトリクスの転置
`m*` *m1 m2* [*rm*]	マトリクス積
`rotate-vector` *v rad axis* [*rv*]	ベクタの回転
`rotate-matrix` *m rad axis wrt* [*rm*]	マトリクスの回転
`rotation-matrix` *rad axis* [*rm*]	回転マトリクスの生成
`rotation-angle` *m*	等価回転軸と回転角
`rpy-matrix` *roll pitch yaw* [*rm*]	ロール・ピッチ・ヨー角からマトリクスを生成
`rpy-angle` *m*	ロール・ピッチ・ヨー角の取りだし
`euler-matrix` *az1 ay az2* [*rm*]	オイラー角からマトリクス生成
`euler-angle` *m*	オイラー角の取りだし
`matrix` *row ...*	マトリクスの生成
`matrix-row` *m*	マトリクスの行の取りだし
`matrix-column` *m*	マトリクスの列の取りだし
`lu-decompose` *m*	lu 分解
`lu-solve` *lu v*	lu 分解された連立一次方程式の解
`lu-determinant` *m*	lu 分解したマトリクスの行列式
`simultaneous-equation` *m v*	連立 1 次方程式の解
`inverse-matrix` *m* [*rm*]	逆行列

v、*v1*、*v2* はベクタ、*rv* は結果ベクタ、*m*、*m1*、*m2* はマトリクス、*rm* は結果マトリクス、*s*、*rad* はスカラ、*axis* は軸ベクタあるいは軸キーワードを表す

表 2.5: EusLisp の幾何演算プリミティブ

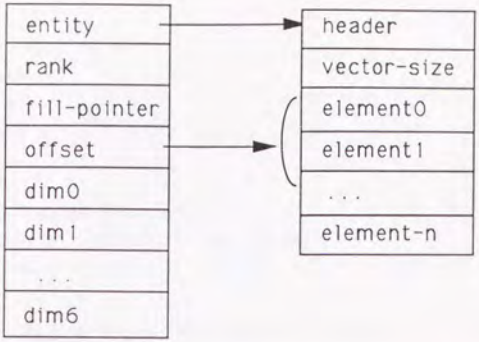


図 2.14: 配列の構造

に結果を受け取るパラメータを指定できるようにしている。2.5に *rv*、*rm* で示したのがそれである。したがって、v+ などの関数では任意の個数の引数ベクタの加算を行なうことはできない。

2.9 他言語インタフェース

Lisp は、汎用言語として広い適合性を持っているが、世の中にすでに多くの言語が普及している以上、すべての問題を Lisp で記述し、Lisp だけの閉じた世界を形成しようとするのは危険である。そのような方針では、これまでに C、Fortran などの言語が蓄積してきた数値演算パッケージ、ウィンドウシステムなどの膨大な資産を利用することができず、今後の技術の進歩を柔軟に吸収することが困難になるからである。EusLisp のローダは、既存のプログラムオブジェクト、ライブラリを実行中の EusLisp にリンクする機能を有し、オブジェクト指向の持つ拡張性を活かして、これらのプログラムの記述言語の違いをあまり意識することなくそれらの機能を利用できるようになっている。

2.9.1 他言語プログラムのロードとアクセス

EusLisp のカーネルはCで記述されている。また、EusLisp コンパイラは Lisp ソースをCに変換する。したがって基本的に EusLisp とCはよく親和する。EusLisp コンパイラの生成するCプログラムと、通常のCプログラムとの主な違いは、(1) 関数のシンボルへの登録（初期化）、(2) 引数と結果の型変換、の 2 点にある。(1) は defforeign によってロード時に、(2) は funcall によって実行時に解決される。

EusLisp がコンパイルしたプログラムは、load によって読み込まれ、compiled-code のインスタンスが作られた後にプログラム中の初期化ルーチンが実行される。これに対し一

```
/* a C function in "sync.c" */
float sync(x) double x;
{ extern double sin();
  return(sin(x)/x);}

eus$ cc -c sync.c
eus$ (setf m (load-foreign "sync.o"))
eus$ (defforeign sync m "_sync" (:float) :float)
eus$ (sync pi)  --> 0.0
```

図 2.15: 他言語プログラムとのリンクの例

般のCプログラムは、load-foreign によって読み込まれ、foreign-module のインスタンスが作成されるが、初期化ルーチンが実行されることはない。foreign-module は compiled-code のサブクラスであり、オブジェクトコードとしての性質の他に、モジュール中に定義された外部シンボル (関数名) のハッシュ表を属性に持っている。このシンボル表を参照して、defforeign マクロが Lisp からC関数へのエントリを付ける。defforeign には、外部関数のパラメータと結果の型を指定する。実際は、Lisp のポインタからは実行時に型の情報が抽出できるので、defforeign の引数の型指定は省略できる。Cから返される値の型は宣言なしには知ることができないので、結果型の情報は省略できない。Lisp の defun がシンボルに compiled-code オブジェクトを登録するのに対し、defforeign はシンボルに foreign-code を登録する。foreign-code はやはり compiled-code のサブクラスであり、関数へのエントリ番地のほか、パラメータと結果の型が記憶される。

funcall は、コードオブジェクトの型を見て、foreign-code であれば、引数を Lisp のポインタからCの要求する型に変換する。型変換に当たって、EusLisp の string、integer-vector、float-vector の内部表現はCや Fortran での表現と同じなので、変換のオーバヘッドは僅かである。図 2.15に、Cのプログラムをロードし、エントリを定義し、実行する例を示す。

Cから EusLisp の関数を呼び出せるようにするためには、次の defun-c-callable マクロを用いる。

(defun-c-callable *funcname*
  ((*param type*)*) *result-type* . *body*)

defun-c-callable は、symbol のサブクラスである foreign-symbol のインスタンスを作成する。foreign-symbol には、Lisp の関数としての属性の他に、Cから呼ばれた場

合にCからLispへの引数の変換を行なうルーチンに分岐する機械語列と、引数と結果の型の情報が格納されている。この機械語列はCからLispの関数を特定するために異なった場所に生成される必要があり、共有することはできない。この機械語列の番地をCプログラムに知らせておけば、CからLisp関数を呼び出すことができる。defun-c-callableの機能は、ウィンドウサーバから送られて来るマウスのイベント処理などに必要となる。

LispからCのstructへアクセスするために、cstructクラスがある。cstructは、stringのサブクラスとして、つぎに示すようなdefcstructマクロによって定義される。structの各フィールドには、*(structname-slotid* cstruct*)*によってアクセスすることができる。

```
(defcstruct structname
            {(slotid :primitive-type) |
             (slotid cstructname) |
             (slotid (:primitive-type [*] [dimension]))}
)
```

さらにCがmalloc等で確保したメモリにLisp側からアクセスできるよう、stringのサブクラスにforeign-stringがある。foreign-stringはLispからはベクタとして扱われるが、実体はLispのメモリ空間外に取られる。

2.9.2 ウィンドウシステム・インタフェース

他言語インタフェースを用いてXwindowインタフェースを作成した。load-foreignによってXlibライブラリをロードし、defforeignで各関数のエントリをつけてやることで、400余りのXlib関数がすべて利用可能になる。これらの関数を元にして、図2.16に示すようなxwindow、pixmap、graphic-context、colormap、fontなどのクラスを定義している。

たとえば、次の式によってウィンドウを作成することができる。

```
(instance xwindow :create
    :x 100  :y 100  :width 500  :height 500  :title "eusx"))
```

このように他言語インタフェースによってライブラリをリンクする方法は、XwindowのプロトコルをCommon Lispで記述したCLXや、インタフェースをCで書く例と比べて開発期間が短くて済み、効率も良く、ウィンドウシステム自体の改訂にも柔軟に対応できるという利点がある。

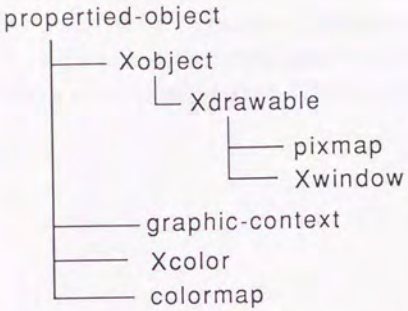


図 2.16: X ウィンドウのクラス

2.10 2 章の結論

ロボットプログラミングの核言語となる EusLisp の機能とインプリメンテーションについて論じた。拡張性、互換性、習熟・プログラミングの容易さなどを考慮して、言語の基本仕様は Common Lisp に従った。Common Lisp に、型の拡張という形でロボット向き機能の拡張を容易に施せるよう、またロボットにとって重要な役割を演ずる幾何モデルが適切に表現できるよう、システムの実現はオブジェクト指向をベースにした。

オブジェクト指向型言語が十分な性能を発揮するためには、一般に、(1) スロット変数のアクセス、(2) メソッドの探索、(3) インスタンスの作成、の効率が問題となる。これに対し、EusLisp は以下のような解決を図った。

1. 単一継承を基本に置くことで、特別な宣言を設けなくてもスロット位置がメソッド、オブジェクトに関わらずコンパイル時に一意に定まるようにする。

2. メッセージ送信におけるメソッドの動的探索を関数をオブジェクトに適用する場合の型判別に置き換える。クラスの階層構造に応じてクラス id を適当に割り当てることにより、型判別はレンジチェック命令によって高速に実行することができる。

3. 単一継承の欠点を補うため、メッセージをスロット変数にバインドされたオブジェクトに自動的に再送信するメッセージフォワーディングの機構を実現する。

4. オブジェクト指向プログラミングにおける多様なサイズのオブジェクトと、Lisp のリスト処理における頻繁な cons の要求の両方に効率良く答えるため、フィボナッチバディ法に基づくメモリ管理を実施する。

これらの方策により、Common Lisp の組み込みデータ型をクラスで記述することに成功し、組み込み型をスーパクラスに指定することによる高い拡張性を実現した。また、Lisp の性能としても、ベンチマークによって、KCl に匹敵する速度が得られていることが確認できた。EusLisp はまた、ロボットに必要な幾何演算機能、他言語インタフェースを備えている。

第 3 章

作業記述のためのオブジェクト指向型幾何モデラ

3.1 オブジェクト指向型 Lisp と幾何モデラ

ロボットプログラミングシステムにおいて基調をなすモデルは、物体の 3 次元的形状表現する幾何モデルである。幾何モデラとは、基本素立体の生成とその集合演算による 3 次元の形状定義、移動・回転などの座標変換、物理的属性の定義と問い合わせ、グラフィクス表示などの機能を持ったソフトウェアである [19]。[1]

3 次元幾何モデラの起源と考えられるのは、1973 年に発表された Cambridge 大学の Braid の Build システム [6, 5] と、北海道大学の沖野らの TIPS-1[32, 52, 53] である。その後、Rochester 大学の PADL、PADL-2、ベルリン工科大学の COMPAC、IBM の GDP[11, 48]、東京大学の GEOMAP[15] 等が開発され、しだいに豊富な形状を表現する能力を備えるようになり、GEOMOD、ROMULUS などの商用のシステムに発展していく。

これらの幾何モデラは CAD を主目的としており、形状定義部分とデータ解析部分は独立したモジュールとして構成されるのが普通である。すなわち、作成された形状データはいったん外部のデータベースに蓄積され、図面作成、有限要素法を適用するためのメッシュの生成、機構解析などのサブシステムがそれを読み込むようになっている。したがって、それらの幾何モデルをロボットに応用しようとする場合は、外部表現に変換された形状データをロボットアプリケーション側で読み込み、幾何モデルを再構築することになる。これではアプリケーション側からオンラインで形状に変更を加えたり、対話的にモデルを定義することが困難になる。その上、たとえ形状定義を外部に委ねることができたとしても、ロボットアプリケーションにとって形状モデルを用いた幾何計算が必要なことには代わりがなく、幾何モデラのほとんどの機能をアプリケーション側にも再プログラムする必要が生ずる。

上記の幾何モデラは、いくつかが Algol68 やアセンブラなどで書かれた他はほとんどが FOTRAN で記述されている。これは、数値計算を重視したため、およびシステムが大規模

[1] 幾何モデラのうち 3 次元の体積が表現できるものをソリッドモデラという。ロボットが対象とするのはもちろんソリッドモデルであるが、この用語は CAD を目的としたシステムに使われることが多いのでここではあえて幾何モデラと言うより広い意味の用語を用いている。

なために歴史的経緯でそうならざるを得なかったためと考えられる。しかし、幾何モデラプログラムが複雑なのはモデル要素間の位相の管理が複雑なためであって、幾何計算が複雑だからではない。実際、これらのプログラムは、かなりの部分をメモリの管理やポインタ操作に費やしており、Lisp のようにそれに適した言語が多く存在する現在では、もはや Fortran 等に拘泥する必要はない。すなわち、オブジェクト指向型 Lisp で幾何モデラを実現することには次のような利点があると考えられる。

1. モデル要素はオブジェクトで、位相関係はリストによって明確に表現できる
2. オブジェクトによってモデルの振舞いが抽象化され、直観的理解が容易になる
3. クラスの継承を用いることで属性の追加が可能
4. モデルの保存・通信（外部表現への変換）が容易
5. メモリ管理から解放され容量的制限が無い
6. Lisp のトップレベルをシェル言語として利用できるので特別のコマンド言語作成の必要がない
7. 幾何モデルとロボットプログラムとの結合が容易

この章では、EusLisp を用いた幾何モデラの実現とその作業記述への応用について述べる。EusLisp は、浮動小数の即値レンジを広く取り、ベクタ・マトリクス演算を組み込み関数で処理できるので、幾何モデラの効率の良い記述言語となり得る。また、頂点、エッジ、面、立体、座標系、などのモデル要素はオブジェクトとして定義される、クラスの継承を用いた機能の拡張が容易に行えるようになる。物体は多面体で近似され、Brep(Boundary-representation) で表現される。以下では、まず座標系の表現、Brep による幾何モデルの表現、CSG 集合演算による形状定義法、グラフィックス等の EusLisp の幾何モデリング機能について述べ、次に、幾何モデルを拡張することで表現されるマニピュレータ、環境物体、作業対象物体、およびマクロ作業記述法と時系列ワールドの定義について述べる。

3.2 幾何モデルの表現法と特徴

Requicha は、幾何モデラの性能を

1. どのような物体を定義できるかという記述能力
2. 正しいモデルを生成できモデルの正当性を保証する能力

3. 一つのモデル表現が必ず単一の物体に対応するかどうかの完全性

4. 物体に対してモデルの表現が一意に定まる単一性

5. 表現の簡潔さ

6. モデル生成の容易さ

などの要素によって評価することを提案している [38]。そして、3 次元の幾何モデルの表現法を、

1. すべての形状をプリミティブとして定義するプリミティブインスタンス法

2. 占有する空間をすべて均等に数え上げる空間要素列挙 (ボクセル) 法

3. 占有する空間を 4 面体などで数え上げる分割法

4. 空間をプリミティブ物体の集合演算で合成する CSG(Constructive Solid Geometry) 法

5. 2 あるいは 3 次元プリミティブを軌道に沿って掃引する掃引法

6. 形状の境界を 2 次元面で表す Brep(Boundary REPresentation)

に分類している。[38]。図 3.1に、Requicha による各手法の評価を示す。

現実のソリッドモデラは、TIPS が CSG 法、COMPAC が掃引法をもとにしている他は、ほとんどが Brep に基礎を置くものであると言ってよい。その理由は Brep の次のような性質によるものと考えられる。

汎用性 他のどの表現からも Brep を得ることができ、表現が汎用的である。したがって、他の方法と適宜組み合わせて欠点を軽減することができる。たとえば、入力 (形状定義) に CSG を組み合わせることで、正当性、簡潔性、生成の容易さの欠点を克服できる。

マスプロパティ 表面積、体積、重心などの CAD に重要なマスプロパティの導出が容易である。

表面処理 一般に、物体の内部と外部を区別して表現できるソリッドモデルは表面形状だけに注目するサーフェイスモデルより高級であるように受け取られているが、物体と外部とのインタラクションはほとんどその界面で起こる。接触の検出、3 次元グラフィクス表示等に Brep は十分な機能と効率を発揮できる。

	記述能力	正当性	完全性	単一性	簡潔性	生成の容易さ
プリミティブインスタンス法		○	○	○	○	○
空間要素列挙法		○	○	○		
分割法	○		○			
CSG	○	○	○		○	○
掃引法		○	○		○	○
Brep	○		○			

図 3.1: 幾何モデル表現法の比較

EusLisp でも、幾何モデルの表現は Brep を基本とする。ただし、形状定義を簡単にするため、形状の入力は CSG 法による形状の集合演算を用いた簡便な方法が取れるようにする。CSG から Brep への変換はシステムが自動的に行う。しかし、Brep から CSG への逆変換は一般には不可能である。一方、画像理解のように、エッジや面の情報から物体を推定する処理では、面が円柱の底面なのか側面なのかというような、記号的な情報が必要になる。このために、変換された後の Brep においても、CSG 定義時のパラメータ、面の種類を同定するための属性を残しておくようにする。

Brep では、頂点、エッジ、面、物体などの要素ごとにデータ構造が必要になる。EusLisp では、これらのモデル要素はオブジェクトとして実現される。そのクラスの継承構造を図 3.3に示す。

3.3 座標系と座標変換

ロボットプログラミングでは、物体や視点の位置・姿勢を表すために座標系が非常にしばしば登場する。座標系の位置を表す手段にベクタを用いることに異論はないが、回転の表現法には、単位 4 元数、バウリスピン、ユニタリ 2 × 2 行列等、多数の方法がある [40]。ロボットで最も一般的に用いられるのは、3 次元の同次座標を用いて、位置と姿勢を単一の 4 次元マトリクスで表現する方法である [35]。この方法には、並進成分と回転成分を同一の方式で表現でき、特に式の上では単純な記述が可能になると言う利点がある。一方、1/4 の要素は冗長であり、そのために変換と変換の合成に 64 回の乗算と 48 回もの加算が必要となると言う欠点がある。さらに、同次座標をマトリクスによる変換の対象として扱うのは都合がよいとしても、ベクタ同士の加減算、内積、外積などの演算の対象とする場合、通常座標と同様に扱うことができず、インプリメンテーションが複雑になる。[2]

そこで、EusLisp では、回転を 3 × 3 のマトリクスで、並進を 3 次元のベクタに分離して表現する方法を取っている。この方法では、変換の合成は、36 回の乗算と 27 回の加算で済ますことができ、メモリの消費も 3/4 に抑えることができる。通常座標表現であるので、ベクタ、マトリクスの構造と演算を一般次元を対象にした汎用的な形式にすることができる。さらに、並進成分と回転成分を独立に扱うことが容易になり、特に人間とインタフェースする場合に直観的な理解が容易になる。

座標系を表すクラスには `coordinates` と `cascaded-coords` がある。図 3.2に、そのスロット変数の内容を示す。`coordinates` は単一の座標系を、`cascaded-coords` は座標系の連結を表現するのに用いる。これらをスーパクラスにすることにより、物体、視点など、

[2]EusLisp の前身となった ETALisp は、同次座標表現を採用していた [56]。ベクタとマトリクスのデータ構造と関数は、3 次元の同次座標だけを対象としており、一般次元の通常の座標表現、座標変換を扱うことができなかった。このため、通常座標にはリスト表記を併用することになり、混乱が生じ易かった。

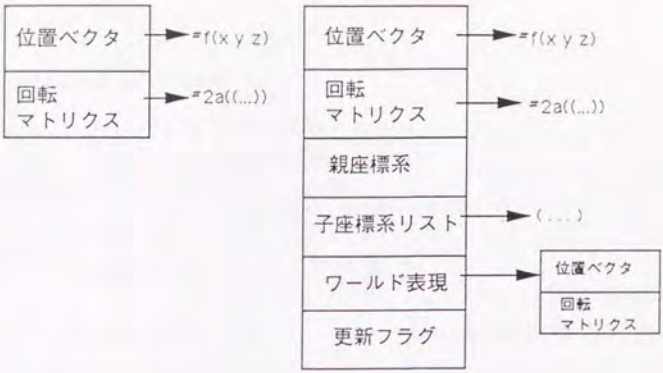


図 3.2: coordinates と cascaded-coords の構造

座標を持ったモデルが統一的に実現できるようになる。

3.3.1 クラス coordinates

クラス coordinates は、座標系あるいは座標変換を表す。図 3.2に示されるように、原点位置を表す 3 次元ベクタと回転を表す 3×3 マトリクスをスロットに持つ。座標系間のベクタの変換、相対変換、逆変換、変換の連結などは表 3.1に示すようなメソッドとして定義されている。3.4節で述べるように、物体のモデルである body、視点を表す viewing などの coordinates のサブクラスはこれらのメソッドを共有する。

ある coordinates に変換を加えるメソッドには、:transform, :move-to, :rotate, :orient, :translate, :locate の 6 種類がある。:transform, :move-to は引数に coordinates を取り、自分自身 (self) に別の coordinates を掛け合わせる。:transform は現在の座標系に対して相対的に変換を加え、:move-to は、絶対的に新たな座標系を指定する。:rotate は相対的な回転を、:orient は絶対的な方向を指定する。:translate は相対的な並進を、:locate は絶対的な座標原点位置を指定する。これらのメソッドでは、wrt(with-respect-to) 引数によって、変換、回転、並進を表現している座標系を指定することができる。wrt 引数には、:local、:world などのキーワード、あるいは他の座標系オブジェクトを取ることができる。現在の座標系を C_1、wrt 座標系を C_2、wrt 座標系で表された変換を T とすると、:transform メソッドでは、C_1 は次の C_1' に変換される。

$C_1' = C_2 T C_2^{-1} C_1$

wrt が:local のとき $C_2 = C_1$ であるので、

$C_1' = C_1 T C_1^{-1} C_1 = C_1 T$

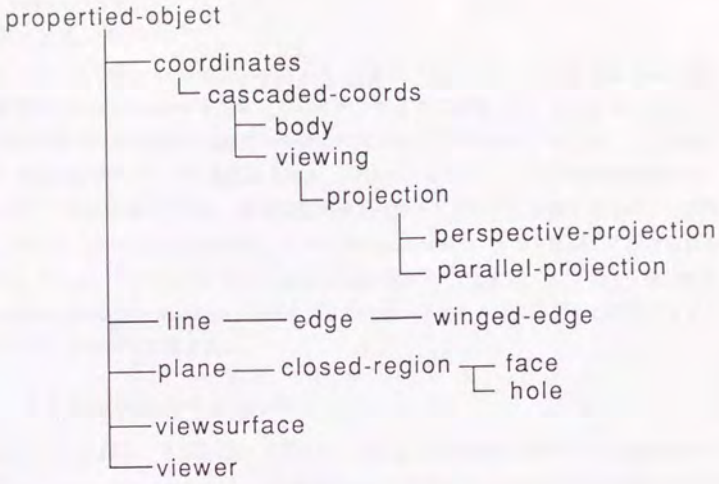


図 3.3: 幾何モデルクラスの継承構造

:rot	座標系の回転要素を表す3×3マトリクスを返す
:pos	座標系の原点を表す位置ベクタを返す
:newcoords	回転と位置を更新する
:coords	この座標系の複製を作成する
:transform-vector	この座標系で表されたベクタを基準座標系表現に変換
:inverse-transform-vector	基準座標系でのベクタ表現をローカルな表現に変換
:transform	座標系に変換を加える
:move-to	座標系を絶対指定する
:translate	座標原点を平行移動する
:locate	座標原点を絶対指定する
:rotate	座標系に回転を加える
:orient	座標系の回転を絶対指定する
:inverse-transformation	この座標系の逆変換を作成する
:transformation	他の座標系との間の変換を作成する
:Euler	回転をオイラー角で指定する
:roll-pitch-yaw	回転をロール・ピッチ・ヨー角で指定する
:4x4	4×4マトリクス表現と変換する
:init	座標系を初期化する

表 3.1: クラス coordinates のメソッド

また wrt が:world のときは、$C_2 = I$ であり、

$C_1' = ITI^{-1}C_1 = TC_1$

と簡単になる。

これらのメソッドは、coordinates のスロット変数である rot と pos を変更する。座標変換の変更は、coordinates を継承しているサブクラスに影響する。たとえば、body では、座標系が書き変わる度に頂点座標や平面方程式を更新しなければならないし、視点座標系を表す viewing では、内部の逆変換を更新しなければならない。これらの更新操作は、各々のサブクラスに定義しておき、座標変換の変更に伴って連鎖的に起動されるのことが望ましい。そこで、rot と pos の変更は、すべて:newcoords メソッドを経由して行なわれるようになっている。サブクラスでは、:newcoords を新たに定義し、まず send-super によって coordinates の:newcoords 処理を済ませた後、そのクラスに特有の処理を行うようにすることでこの連鎖を実現できる。

3.3.2 クラス cascaded-coords

cascaded-coords は、多関節型マニピュレータのような階層的に連結された座標系の列を表現する。cascaded-coords は、coordinates のサブクラスであり、親座標系に対する相対的な変換を保持している。また、一つの親座標系と、複数の子座標系のリストを保持している。

ある座標系の基準座標系 (ワールド) に対する絶対的な座標表現は、すべての親座標系の保持している変換と自分の変換を掛け合わせることで得られる。この変換の連結操作はかなりの計算量を必要とする上に、ある座標系で表現されたベクタをワールドでの表現に変換したいという要求は非常にしばしば生ずる。その度に変換の連結を行なうのは無駄が大きいので、各々の cascaded-coords は、連結された変換を worldcoords スロットに記憶している。

この方法では、今度は、親座標系に何らかの変更を加えた場合に子座標系に貯えられた worldcoords を更新する操作が必要になる。このとき、たとえば親座標系に三つの軸回りの回転と並進とを順番に与えたとすると、子座標系をトラバースして更新する操作は 4 回必要になる。この負担を軽減するため、親座標系が変更されても直ちに変換を連結する処理は行なわず、変更が行なわれた事実だけをすべての子座標系に伝播し、更新フラグに記録する。実際に座標変換の連結処理がなされるのは、cascaded-coords にワールドに対する表現を問い合わせた時、すなわち:worldcoords メッセージが送られてきたときである。このとき更新フラグもクリヤされる。

他のメソッドは、coordinates に示されているのと同じである。この表には現れていないが、cascaded-coords では wrt 引数に:parent を取ることができるのと、:world の解釈が coordinates とは異なるので、:transform, :move-to, :rotate, :orient,

:inheritance	接続された子座標系の木構造リストを返す
:assoc	子座標系を接続する
:dissoc	子座標系を切り離す
:changed	座標系が書き換えられたことを記録する
:update	変換を実際に更新する
:worldcoords	祖先の相対変換を集積し自身の基準座標系表現を求める

表 3.2: クラス cascaded-coords のメソッド

図 3.4: 幾何モデル要素のオブジェクト構造

:translate, :locate のメソッドは wrt に対する座標変換が追加されている。

3.4 形状モデル

稜線、面、物体などの形状要素はクラスによって定義される。図 3.3にクラスの継承関係を、図 3.4に、各クラスの構造を示す。

3.4.1 稜線の表現

B-rep の実現法としては、Baumgart の winged-edge[3] 構造体がよく知られており、GEOMAP[14]、Solver[21, 50] などにも踏襲されている。winged-edge は、一つの稜線を中心にその両側の面と隣の稜線を記述する。すなわち、稜線オブジェクトには、二つの頂点、接続する 4 本の稜線、両側の二つの面の 8 個のポインタが記録される。winged-edge では稜線のトラバースが容易になり、エッジベースのコンピュータビジョン、高速の隠線消去アルゴリズムに有効なデータ構造であると言われる。欠点は、記述が冗長で記憶領域を無

駄に消費することとリンクの管理に手間がかかることである。図3.5にSolverにおける稜線と面を表すデータ構造を示す。稜線オブジェクトには、上記の8本のポインタの他に、一つの物体に関連する双方向鎖を加えた10本のポインタが 記録されていることがわかる。GEOMAP、Solverはガベジコレクタを持たず、メモリ管理を簡単にするためにすべてのデータオブジェクトに対してGEOMAPでは8語、Solverでは16語の固定長セルを割り当てている。これに対しEusLispでは、(1)Lispではリストを利用できること、(2)共通の構造をスーパクラスに括り出せること、(3)ウィング情報は冗長であること、などの理由により、Winged-Edgeとは異なった構造をとっている。

EusLispでも、最初はwinged-edge型のデータ構造を採用したが、上記の長所よりも、構造が複雑で冗長であることによる欠点の方が顕在化した。すなわち、複雑な形状の表現に大きな記憶を必要とし、形状合成操作においてポインタの参照関係の整合性を保つために相当量のプログラム記述を強いられる。特に前者の問題は、EusLispのように曲面を多数の平面で近似する場合、単純なプリミティブであっても稜線の数はかなりの数に達し、大きな問題となる。そこで、実際に形状合成、隠線消去の関数をコーディングしてみたところ、稜線のトラバースは、ある面を固定してその面を構成する輪郭稜線をたどるか、物体を構成するすべての稜線について平等に関数を適用するか、の2種類で記述できることが判明した。そこでEusLispでは、winged-edgeから接続稜線へのリンクを取り除き、図3.4に示すような簡略化されたデータ構造をedgeクラスとして定義した。稜線の接続は面を表現するfaceクラスにリストとして記録しておけば、探索によってwinged-edgeが表すすべての情報を取り出すことができ、記憶が節約できる。

しかし、稜線の交差において不可視数を伝播させるAppeleの隠線消去アルゴリズムのように、winged-edge表現が都合が良い問題が存在する。そのような場合に備えて、EusLispでは上記のedgeクラスの簡略形式を基本とし、オプションとしてwinged-edgeを採用できるようにしている。すなわち、winged-edgeクラスはedgeクラスにウィング情報を付加したサブクラスとして定義される。winged-edgeの機能は、edgeクラスに何らの変更も加えることなく、その拡張として定義することができる。これは、従来型の機能拡張が困難なソリッドモデラに比較して、オブジェクト指向の利点を発揮した顕著な例であると言える。

3.4.2 面の表現

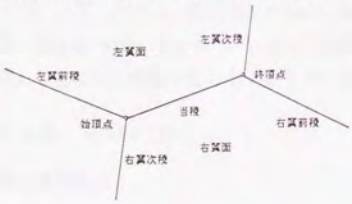
物体は、曲面を含んでいたとしても、すべて平面によって近似される。面は、faceクラスによって表現する。faceのスーパクラスであるplane、closed-regionは、それぞれ平面方程式、輪郭稜線と頂点のリストを表現する。closed-regionはholeクラスのスーパクラスでもあり、faceとholeに共通の性質を実現する抽象クラス(abstract class)となっている。faceは、closed-regionにholeのリストを加えたものと定義されてい

2 稜（Edge）

近似曲面の見掛けの稜
曲稜（予定）
空洞の稜

	0・・・4	・・・8・・・12・・・16・・・20・・・24・・・28・・・
1	00010	＊＊　＊　作業用フラグ
2	⇒	稜　の　双方向鎖
3		⇒
4		左翼面　⇒
5		右翼面　⇒
6		始頂点　⇒
7		終頂点　⇒
8		左翼前稜　⇒
9		左翼次稜　⇒
10		右翼前稜　⇒
11		右翼次稜　⇒
12		
13		
14		
15		〈稜像〉　①　⇒
16		付加　⇒

① 投影体生成時に対応する稜像のアドレスが作業用にセットされて残る．



3 面（Face）

紙
稜線がある
曲面（予定）
穴がある
空洞の面
凸包として生成した立体の「床に置ける面」

	0・・・4	・・・8・・・12・・・16・・・20・・・24・・・28・・・
1	00011	＊　＊＊＊　＊＊　作業用フラグ
2	⇒	面　の　双方向鎖
3		⇒
4		稜数　か　周数
5		稜　か　穴　①　⇒
6		λ　②
7		μ
8		ν
9		d
10		
11		
12		
13		稜線
14		識別番号
15		〈面像〉　③　⇒
16		付加　⇒

① 一つの面には外周が１つだけある（穴はあってもよいが，『島』は別の面になる）．
穴がないときは，その外周の稜のひとつ．
穴があるときは，外周，内周の各代表稜のリストである（要素としての）「穴」を指す．
② 平面方程式：　λ・X＋μ・Y＋ν・Z＋d＝0　（空間座標系での値）．
③ 投影体生成時に対応する面像のアドレスが作業用にセットされて残る．

図 3.5: Solver の稜線と面を表す構造体

る。各クラスに定義されているメソッドの一覧を表3.3に示す。

このような構造によって、交差や属性の実質的な計算アルゴリズムはplaneとclosed-regionに記述し、faceにはその組合せ方を定義するだけで各種の計算が系統的に実現される。たとえば、面と稜線の交差は、次のようにして求められる。

1. planeに定義された平面方程式および:intersectionメソッドを用いて交点の座標を求める。
2. 交点が面の輪郭の内部にあることを輪郭に対する:insidepを用いて検査する。
3. 交点がすべてのholeの外部にあることをholeに対する:insidepを用いて検査する。

また、面の面積、重心は、次のように求められる。

1. closed-regionの:area、:centroidメソッドを用いて輪郭内部がすべて充満している面の面積、重心を求める。
2. 同様に面内のすべてのholeの面積、重心を求める。
3. 1から2を除いて面の面積、重心とする。

曲面は多くの平面に分割されて近似されるので、できあがった面オブジェクトはもともとが平面なのか、曲面なのかが不明になる。これでは、面の関係や拘束を取り扱うのに不便なので、二つのスロットに面の種類が記載される(図3.4参照)。primitive-bodyには、その面を生成したプリミティブ形状モデル(bodyオブジェクト)が記録される。face-typeには、上面(:top)、底面(:bottom)、側面(:side)の区別が記録される。これらの情報から、:face-typeメッセージを送ることにより、(:cylinder :side 1)(円柱の2番目の側面)などの面の種類を取り出すことができる。

3.4.3 形状の定義

素立体の生成

物体はクラスbodyによって表される。bodyはスーパクラスにcascaded-coordsを取り、face-list、edge-list、vertex-list、model-vertices、bounding-box、csgなどのスロット変数を持つ(図3.4参照)。

bodyを作成するには、まず、表3.4に掲げる関数を用いて、直方体、角柱、錐、回転体などのプリミティブbodyを生成する。図3.6にこれらの素立体の形状を示す。これらの関数がbodyを発生させるときは、まず全頂点について頂点-エッジリストを作る。これは、頂点とそれに接続するエッジをリストにしたもので、最初は次のような形式である。

クラス plane のメソッド	
:normal	平面の法線ベクトル
:distance *point*	平面と点の距離
:intersection *point1 point2*	平面と直線との交点
:intersection-edge *edge*	平面と稜線の交点
:foot *point*	*point* から平面に下ろした垂線の足
:project *vector*	*vector* の平面への投影
クラス closed-region のメソッド	
:vertices	輪郭頂点のリスト
:edges	輪郭稜線のリスト
:adjacent-faces	隣に接する面のリスト
:box	閉領域を囲む bounding-box
:insidep *point*	*point* が閉領域の内外判定
:intersection-edge *edge*	平面と稜線の交点
:intersect-face *point1 point2*	平面と直線との交点
:perimeter	周囲長
:area	面の面積
:centroid	面の重心
:invert	面の方向の反転
:visible *vp*	視点 *vp* からの面の可視性
クラス face のメソッド	
:all-vertices	輪郭、穴の頂点のリスト
:all-edges	輪郭、穴の稜線のリスト
:insidep *point*	*point* が面の内部、穴の外部にあることの判定
:area	穴を除いた面積
:centroid	穴を除いた重心
:primitive-body	この面を生成したプリミティブ形状
:face-type	面の種類

表 3.3: 平面を表すクラスのメソッド (抜粋)

convex-hull-3d *points*	点列からの凸包の生成
make-cube *a b c*	直方体の生成
make-cylinder *radius height [segments]*	角柱、円柱の生成
make-prism *bottom sweep*	掃引体の生成
make-torus *points*	トーラスの生成
make-solid-of-revolution *points*	回転体の生成
make-cone *top bottoms*	錐体の生成
make-dodecahedron *radius*	正 12 面体の生成
make-icosahedron *radius*	正 20 面体の生成
make-gdome *hedron*	測地ドームの生成

表 3.4: 形状定義プリミティブ

$((V_1)\ (V_2)\ (V_3) \ldots (V_n))$

次に、各面について外部から見て反時計回りに回る順番で面の周囲の頂点列を与える。この頂点列から頂点を順に二つずつとってエッジを作る。作られたエッジは、頂点 - エッジリストの二つの頂点のリストに加える。たとえば V_2 と V_3 をつなぐ E_{23} が作られると、頂点 - エッジリストはつぎのようになる。

$((V_1)\ (V_2\ E_{23})\ (V_3\ E_{23}) \ldots (V_n))$

1 本のエッジは必ず二つの面に共有されるから、次に V_3 と V_2 をつなぐ E_{32} を作成しようとしたときは、頂点 - エッジリストを探索して E_{23} を見つけ、これを割り当てる。face オブジェクトを作成し、面を構成する全頂点、全エッジを vertex-list、edge-list に入れる。頂点座標から面の方程式を計算し、face-equation に入れる。各エッジの pface と nface スロットには、両側の面へのポインタを登録する。

凸包

点列が与えられれば、convex-hull-3d によって凸包を定義することもできる。凸包を作成するアルゴリズムは、2 次元点列に対しては quick-hull のアルゴリズム、3 次元点列に対しては gift-wrapping 法をインプリメントしている [43]。quick-hull アルゴリズムは、まず平面上の点集合から x 方向に最も隔たった 2 点 (x 座標が最大の点と最小の点) P_1、P_2 を求める (図 3.7参照)。この 2 点は凸包の構成点である。次に、点集合を直線 P_1P_2 によって二つに分ける。この直線から最も隔たった 2 点 P_{11}、P_{21} も凸包の構成点である。以下再帰的に、直線 P_1P_{11}、$P_{11}P_2$、P_2P_{21}、$P_{21}P_1$ について外側の点集合の中から直線に最も隔たった点を求めて分割を繰り返せば目的の凸包が得られる。quick-hull アルゴリズムは、点の総数 n に対して $nlog(n)$ のオーダで凸包を求めることができる。

3 次元の凸包では、このような divide-and-conquer 法が知られていない。

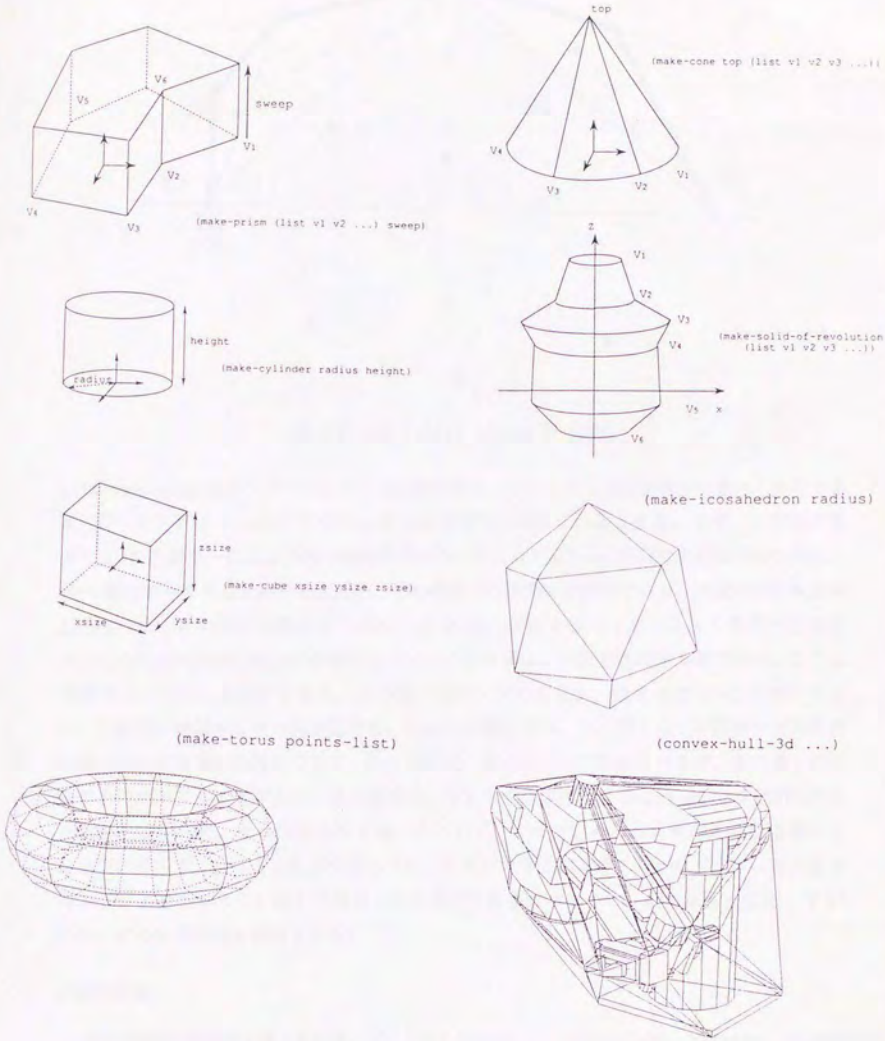

sweep
V1
V2
V3
V4
V5
V6
(make-prism (list v1 v2 ...) sweep)
top
(make-cone top (list v1 v2 v3 ...))
height
radius
(make-cylinder radius height)
z
x
(make-solid-of-revolution
(list v1 v2 v3 ...))
zsize
xsize
ysize
(make-cube xsize ysize zsize)
(make-icosahedron radius)
(make-torus points-list)
(convex-hull-3d ...)

図 3.6: 素立体の生成

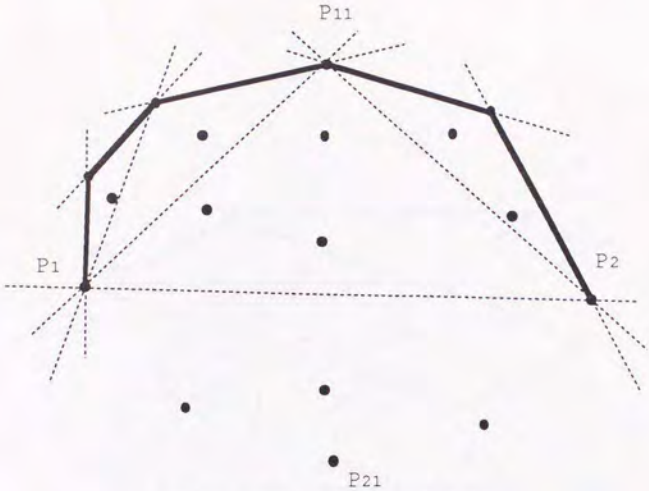


図 3.7: 2 次元の凸包 (Quick-Hull 法)

gift-wrapping 法は、プレゼントを包装紙で包むようにして凸包面を次々に求める方法である (図 3.8参照)。3 次元空間中に n 個の点がばらまかれているとする。まず、z 座標が最大の点 P_1 を探す。これは凸包の構成点である。この点を通る xy 平面に平行な平面を考え、P_1 と他のすべての点とが作る直線が、この平面となす角度を計算すると、角度の最も大きい点 (P_2) はやはり凸包の構成点である。P_1 と P_2、さらに他の 1 点によって平面が決まるが、この平面の片側に他の点が存在しないような平面は、凸包を構成する面である。こうして最初の一つの凸包面が定まる。この面の各エッジの 2 点と、残された $n-3$ 点の一つとで、凸包面に接続する次の面が定まる。これらの面のうち、その面ともとの面のなす角度が最小となる面は次の凸包面である。凸包面は三つの点によって決定されるが、その面上に三つ以上の点が載ることがある。その場合は、それらの点に対して Quick-Hull を適用して 2 次元の凸包を求め、その内点を捨てる。こうして、すでに定められたすべての凸包面の各エッジに対して、次々と凸包面を張っていくことができる。このアルゴリズムは、各凸包面のエッジと他のすべての点とで構成される面の角度を調べるので、点の総数 n に対して n^2 のオーダの計算時間を必要とする。

形状の合成

さらに複雑な形状を定義するには、プリミティブ body に、:translate、:rotate、:transform などのメッセージを送って適当な位置、回転を定めたあと、表 3.5に掲げる関数によって集合演算を行い、形状を合成する。図 3.10には、そのような形状の定義と合成操作の過程が示されている。

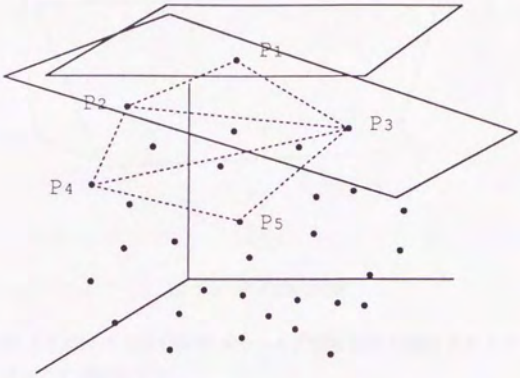


図 3.8: 3 次元の凸包 (Gift-Wrapping 法)

`body+` *bodies ...*	物体の和集合
`body-` *bodies ...*	物体の差集合
`body*` *bodies ...*	物体の積集合
`body/` *body plane*	物体の平面による切断

表 3.5: 形状合成操作

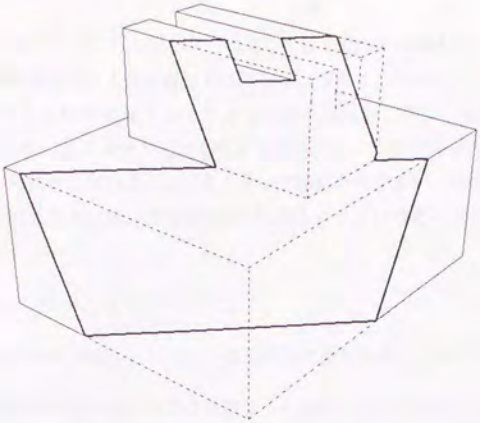


図 3.9: 相貫線の配置

形状合成は次のアルゴリズムにしたがって、まず合成物体を構成するすべてのエッジを作成し、それらを面として再構成する。

1. 物体 A のすべての面と物体 B のすべてのエッジ、物体 B のすべての面と物体 A のすべてのエッジの間の交差を求める。交差のあるエッジは、交点でセグメントに分割される。

2. 分割された各エッジセグメントが相手物体の内部にあるか、外部にあるかを判定する。body+(和集合) の場合は内部のセグメントを捨てる。body*(積集合) の場合は外部のセグメントを捨てる。body-(差集合) の場合は、一方の body の裏表を反転して body+ を行い、結果を反転する。残りのセグメントを分割エッジとして生成する。

3. 次に面と面の交差によって新たに生ずる相貫線を求める。A の m 枚の面 $f_{A_1} \sim f_{A_m}$ と B の n 枚の面 $f_{B_1} \sim f_{B_n}$ の $m \times n$ の組み合わせを考える。f_{A_i} を構成するエッジと f_{B_j}、f_{B_j} を構成するエッジと f_{A_i} の交点は直線上に並び、必ず偶数個ある (図 3.9参照)。これらの交点を直線方向にソートし、奇数番目と偶数番目の交点を結ぶエッジを生成することにより相貫エッジが求められる。

4. 分割エッジ、相貫エッジを、二つの物体の各々の面に由来するエッジにグループ化する。これらのエッジを端点で連続するように連結していくとループが形成される。ループには、輪郭ループと穴ループとがある。ループの包含関係を検査し、最も外側のループを輪郭として face を生成し、残りは hole として face に登録する。

5. 全部の edge、face をまとめて body オブジェクトを作成する。

上記のアルゴリズムは、二つの物体が一般的な状態で交差する場合だけを考慮しており、頂点、エッジ、面が接触するような縮退した場合には適用できない。

CSG による形状合成操作を施す度に、また、頂点の回転、移動、拡大縮小を施す度に body オブジェクトの csg スロットにその履歴を記録する。この情報から、複雑な集合演算を経て合成された形状が、どのようなプリミティブ形状からなるか、再合成するとしたらどのような手続きを経ればよいか、合成操作を一回遡るにするにはどうすればよいかなどを知ることができる。

3.4.4 ビューイングとグラフィクス

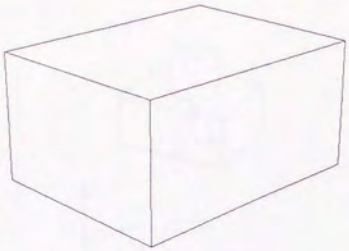
グラフィクス表示のためには、次のような数段階の座標変換が必要である。

1. 物体の固有座標系で定義された頂点座標をワールド (基準) 座標系に変換
2. ワールド座標系から視点を原点とした視野座標系に変換
3. 平行投影、あるいは透視投影変換を加えて NDC (Normalized Device Coordinates) における同次座標表現を得る
4. NDC でビューポートクリッピングを施す
5. NDC からグラフィクス表示装置の物理座標へのビューポート変換

EusLisp には、視野座標への変換を処理するクラス viewing、投影変換を処理するクラス parallel-viewing、perspective-viewing、ビューポートクリッピングのためのクラス viewport、ビューポート変換を行なうクラス viewsurface が用意されている。変換の過程と座標の変化を図 3.11に示す。 projection は、クラス cascaded-coords をスーパクラスに持ち、その機能を用いて座標変換を処理する。

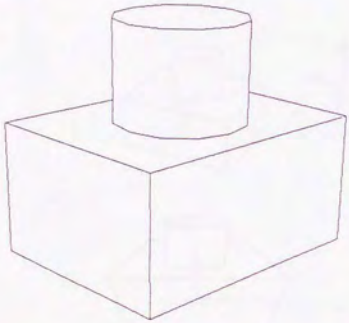
物体の絵を描くには、上記の viewing と viewport、さらに物理的な描画装置を表現する viewsurface オブジェクトが必要である。これら、グラフィクスに関わる座標変換の連鎖を管理するクラスとしてクラス viewer がある。 viewer は、次の関数 view によって作成する。視点、表示位置の異なる viewer を複数生成すれば、マルチウィンドウを利用した多くの絵が同時に表示できる (図 3.12参照)。

```
(view :viewpoint                 視点座標系の原点
      :target                    視線の方向
      :viewdistance, :screen     画角
      :hither, :yon              前後クリップ面距離
      :size, :height, :width     viewsurface の寸法
```

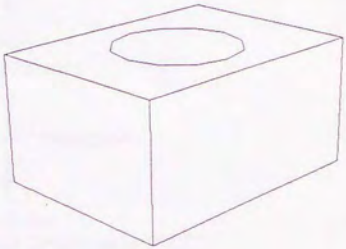


1. a = (make-cube 400 300 200)

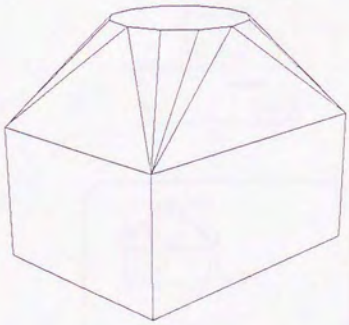
2. b = (make-cylinder 100 300)



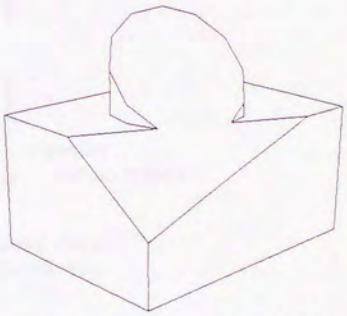
3. c = (body+ a b)



4. d = (body- a b)



5. (convex-hull-3d c)



6. (body/ c (make-plane :point #f(60 70 80)
:normal #f(1 2 3)))

図 3.10: プリミティブ形状の生成と合成操作の過程

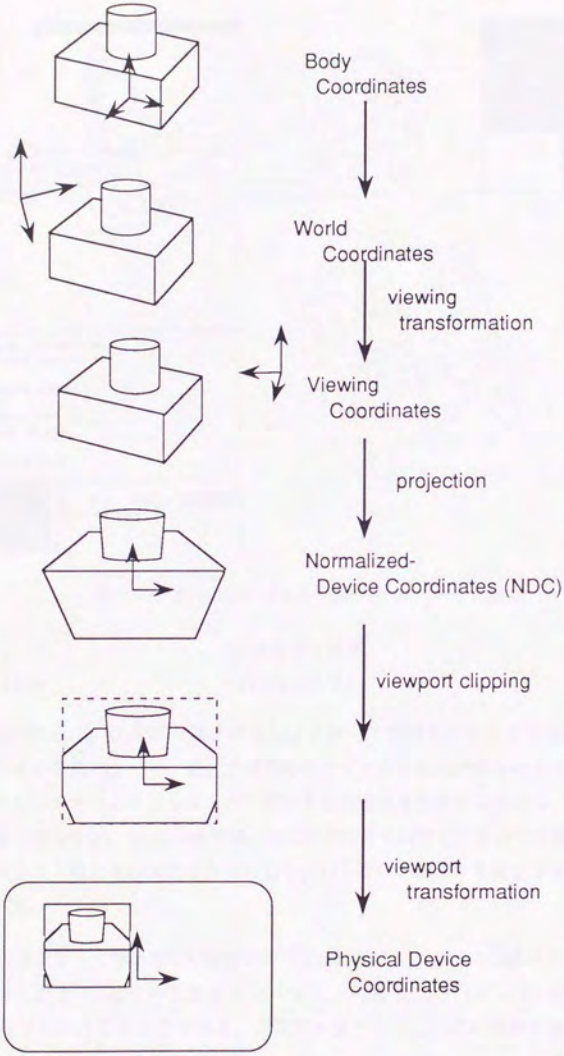


図 3.11: グラフィクス表示の座標変換と処理

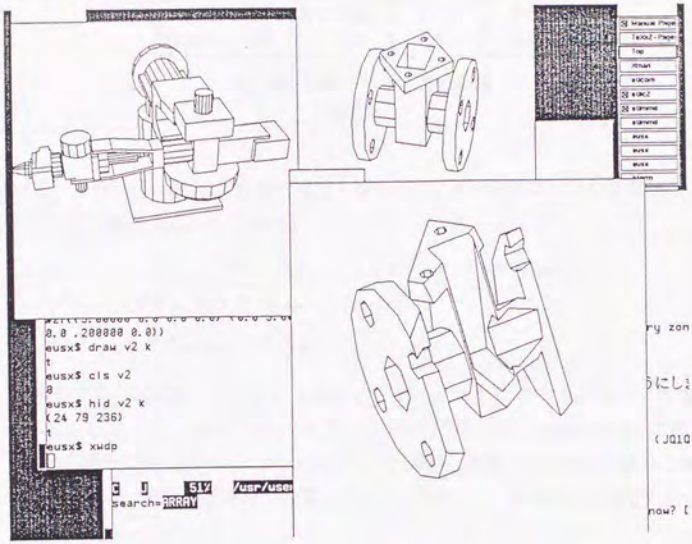


図 3.12: Xwindow によるマルチビューポート表示

:x, :y	viewport 位置
:title	viewport 名)

図 3.12の表示は、隠線消去表示関数 hid を用いて描画したものである。グラフィクスディスプレイの発達によって、最近では実時間でリアルな映像が得られるようになってきている。しかし、ロボットのビジョンが可視面や可視稜線を予測するために、隠線消去機能の重要性はなくならない。EusLisp では、次に示すように線分を交差点で分割し、各々の線セグメントの中点と視点を結ぶ線分が他の面と交わらないかどうかを判定することで可視性を決定している。

1. 面を向きによって潜在的な可視面と不可視面に分ける。面 f の法線ベクトルを f_{normal}、f 上の 1 点を P、視点の位置を V とすると、内積 $f_{normal} \cdot (V - P)$ が正であれば可視面、負であれば不可視面である。可視面の集合を F_{vis}、不可視面の集合を F_{invis} とする。

2. 面 f を構成するエッジの集合を E_f とする。可視エッジの集合を $E_{vis} = \{e; e \in E_f \wedge f \in F_{vis}\}$ とする。可視面 $f \in F_{vis}$ を構成するエッジ $e \in E_f$ につ

	EusLisp	Solver
モデルの作成（集合演算）	116.	38.
隠線消去表示	11.	24.

表 3.6: 幾何モデリングの性能
（単位は秒）

いて、e の両側の面 f_1、f_2 の可視性を調べ、エッジを輪郭エッジの集合 $E_{contour}$ と内線エッジの集合 $E_{internal}$ に分ける。

$(f_1 \in F_{vis} \wedge f_2 \in F_{invis}) \vee (f_1 \in F_{invis} \wedge f_2 \in F_{vis}) \rightarrow e \in E_{contour}$

$f_1 \in F_{vis} \wedge f_2 \in F_{vis} \rightarrow e \in E_{internal}$

$E_{vis} = E_{contour} \cup E_{internal}$ である。

3. 可視面 $f \in F_{vis}$ に対して face-image オブジェクトを作成する。また、可視エッジ $e \in E_{vis}$ に対して、edge-image オブジェクトを作成する。edge-image は図 3.13のような構造を持ち、元のエッジへのポインタ、端点に視野、投影変換を施した NDC での同次座標表現と通常座標表現、可視セグメントのリスト、輪郭エッジを表すフラグをスロットに持つ。

4. 各エッジセグメントの可視性は、NDC で表現されたエッジが輪郭エッジと交差するところで変化する可能性がある。すべての $e_v \in E_{vis}$ と $ec \in E_{contour}$ の交差を計算する。e_v は、この交差によっていくつかのセグメントに分割される。

5. エッジセグメントの可視性を判定する。そのために、エッジセグメントの中点と視点を結ぶ線分を考え、これとすべての面との交差を計算する。交差がなければ可視であり、交差があれば不可視である。交差判定の高速化のため、face-image の bounding-box を用いる。

3.4.5 幾何モデリングの性能

EusLisp の幾何モデリング性能を検証するため、図 3.14に示すような 3 次元モデルの作成と表示に要する時間を Fortran 版幾何モデラ Solver[21] と比較した（表 3.6）。形状の合成操作では Solver が勝るが、隠線消去表示では、EusLisp が高い性能を示している。幾何モデラのような数値演算を多く含むソフトウェアでは Lisp は Fortran 等に劣ると考えられているが、EusLisp は、遜色のない性能を上げていることが分かる。理由としては、幾何演算関数を組み込みとして数値計算の多くを Lisp でコーディングしないで済むようにした

box	境界直方体
face3d	3次元face
edge-images	エッジ像リスト

edge3d	3次元エッジ
homo-pvert	pvertの同次座標
homo-nvert	nvertの同次座標
pvert2	pvertの2次元像
nvert2	nvertの2次元像
segments	分割されたセグメント
contourp	輪郭エッジフラグ

図 3.13: face-image と edge-image オブジェクトの構造

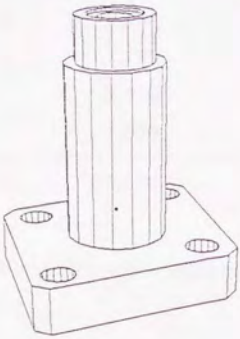


図 3.14: 機械部品の隠線消去表示

記述量の比較

	行数	ファイルサイズ
KCI		
cソース	26,000	500KB
dソース	14,000	290KB
Lispソース	10,000	360KB
コンパイラ	8,700	350KB
ヘッダ	5,000	100KB
	63,700	1,600KB
Solver		
Fortranソース	20,000	540KB
EusLisp		
cソース	14,000	380KB
Lispソース	5,100	165KB
コンパイラ	2,000	73KB
ウィンドウ	1,700	60KB
モデラー	3,000	100KB
ロボット	1,800	80KB
	27,600	860KB

図 3.15: プログラム記述量の比較

こと、edge オブジェクトの構造を winged-edge 構造より簡単にしたことにより構造の一貫性を保つための手続きが簡単化できたことによると思われる。

次に、SOLVER と EusLisp のソースコードの量の比較行った。幾何モデリングに関して、両者はほぼ同等の機能を実現している。図 3.15に示されるように、幾何モデリングに関わる部分のプログラムの記述量は、Solver の約 2 万行に対し EusLisp 版は約 3000 行程度で済んでいる。理由としては、プログラムのほとんどを占めるモデル要素間のトポロジーに関わるポインタ操作において Lisp のリスト処理能力が有効に発揮されたこと、クラスの継承機能により特に座標系の取り扱いにおいて多くのプログラムコードを共有できたこと、メモリ管理、ファイル操作などにおいて Lisp の基本機能が利用できたことなどが上げられる。

3.5 マニピュレータモデル

マニピュレータモデルは関節を表す rotational-joint クラスと、シリーズに連結された関節の順方向、逆方向キネマティックスを解くクラス manipulator で構成される。図 3.16に示すように、rotational-joint は body を、manipulator は cascaded-coords をスーパクラスに持つ。マニピュレータを定義するには、すべての関節を作成した後、manipulator にそれらを統合する。

3.5.1 関節のモデル

クラス rotational-joint が関節のモデルを記述する。クラス rotational-joint は、body をスーパクラスに持ち、その形状モデル、座標系に加えて関節回転軸、回転角度、可動角度範囲、などを管理している。次の defjoint マクロによって rotational-joint のインスタンスが作成され、*joint-name* にバインドされる。parent には、親の関節を指定する。ベースと指には可動軸を指定する必要はない。

```
(defjoint joint-name
        :shape body-object
        :color        color-id              ;0-15 for MMD
        :parent       parent-joint
        :axis         rotational-axis       ; :x, :y or :z
        :offset       trans-from-parent-joint
        :low-limit    joint-angle-limit-low
        :high-limit   joint-angle-limit-hight
        )
```

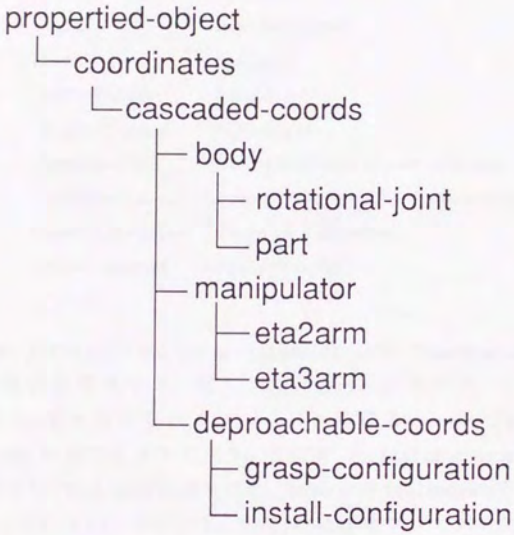


図 3.16: マニピュレータと作業の記述のためのクラスの継承構造

3.5.2 多関節マニピュレータ

マニピュレータモデルはクラス manipulator によって記述される。マニピュレータモデルを作成するには、次の defmanipulator マクロを用いる。

```
(defmanipulator manipulator-name
        :class            manipulator-class
        :base             base-joint
        :joints           list-of-all-joints
        :hand             handjoint
        :left-finger      left-finger
        :right-finger     right-finger
        :handcoords       trans-from-hand-to-armsolcoords
        :toolcoords       trans-from-armsolcoords-to-toolcoords
        :open-direction   finger-open-direction
        :right-handed     righty-or-lefty)
```

manipulator オブジェクトは、base、joints(J1...J6)、handcoords、toolcoords の座標系の連鎖を管理する。図3.17に manipulator オブジェクトのスロット構成を、表3.7に受け付けられるメソッドを掲げる。manipulator クラスは、cascaded-coords のサブクラスであり、やはり、cascaded-coords(または body などのサブクラス)である base に結合され、base から toolcoords(手先座標系)への変換を管理している。したがって、manipulator オブジェクトに対して送られる:translate、:locate、:rotate、:orient、:transform などのメッセージは、手先点に対して作用する。そのとき同時に WRT パラメータを指定すれば、手先は WRT 座標系に対して動く。次のプログラムでは、eta3 を manipulator のインスタンスと仮定している。

```
(send eta3 :translate #f(0 0 -100))          ; 手先を 10cm 引っ込める
(send eta3 :translate #f(0 0 -100) :world) ;10cm 下げる
(send eta3 :translate #f(0 0 -100)
          (manipulator-base eta3))      ; 手先をベース座標系で 10cm 下げる
```

これらのメッセージに対して、manipulator はアーム解を計算して 6 つの関節角度を決定する。一般に解は複数存在するが、right-handed(右手系、左手系)の区別、および現在の関節角度との連続性により適当な解が選択される。しかし、指定された位置、姿勢に対する解が存在しない場合や関節角が限界を越える場合は移動、回転は起こらず、警告が発せられる。

axis	回転軸
offset	bodyの原点から回転中心への変換
high-limit	関節角の上限
low-limit	関節角の下限

(a) rotational-joint オブジェクトの構造

base	基台となる物体
baseinverse	ワールドから基台への変換
joints	関節オブジェクトのリスト
angles	各関節角度のベクタ
right-handed	右手姿勢／左手姿勢
hand	ハンドオブジェクト
handcoords	ハンド座標系への変換
right-finger	右指
left-finger	左指
openvec	指を開く方向ベクタ
max-span	指幅の上限
toolcoords	工具座標系
toolinverse	工具座標系の逆変換
armsolcoords	アーム解を求める座標系
approach	アプローチ姿勢
grasp	把握姿勢
affix	結合された対象物

(b) manipulatorオブジェクトの構造

図 3.17: joint と manipulator の構造

:newcoords	関節角度が限度に収まっていれば座標系を更新する
:armsol-coords	ベースからハンドに至る変換を求める
:tool	工具座標系を返す、または変更する
:set-tool	新しい工具座標系を設定する
:reset-tool	工具座標系を初期値に戻す
:worldcoords	工具座標系のワールド表現を求める
:set-coords	一旦順キネマを計算し、そこからさらに各関節角度を求める
:config	6 つの関節角度を直接に指定する
:park	初期姿勢に戻す
:hand	ハンドオブジェクトを返す
:handcoords	ハンド座標系ののワールド表現を求める
:span	現在の指の間隔を返す
:open-fingers	指幅を相対的、絶対的に指定する
:close-fingers	指を完全に閉じる
:angles	現在の姿勢の関節角度のリストを返す
:right-handed	右手姿勢、左手姿勢を選択する
:get-approach	現在アプローチしている対象を返す
:set-approach	アプローチ対象を設定する
:get-grasp	把握姿勢にある対象物を返す
:set-grasp	把握対象物を指定する
:get-affix	把握している物体を返す
:affix	把握を指示する
:unfix	把握物体を解放したことを指示する
:create	新しいマニピュレータオブジェクトを作成、初期化する

表 3.7: クラス manipulator に定義されたメソッド

アーム解の計算は、実際のマニピュレータに対応した個々の manipulator クラスに定義された:armsol メソッドが行う。マニピュレータがワールド座標系のどこに置かれてもよいように、また、どのような工具を用いてもよいように、アーム解は、base、toolcoords とは独立に、base 座標系中でのハンドの位置、姿勢に対して与えられる。

base、J1、J2、...、handcoords、toolcoords の関係を図 3.18に示す。ワールドから手先への変換を T とすると、T および各部分変換は次のようにして得られる。

$$
\begin{aligned}
T &= base \cdot J1 \cdot J2 \cdot \ldots \cdot J6 \cdot handcoords \cdot toolcoords \\
&= (send\ eta3\ :worldcoords) \\
T_{Jn} &= base \cdot J1 \cdot \ldots \cdot Jn \\
&= (send\ Jn\ :worldcoords) \\
T_{arm} &= J1 \cdot J2 \cdot \ldots \cdot J6 \cdot handcoords \\
&= (send\ eta3\ :armsol-coords) \\
T_{tool} &= J1 \cdot J2 \cdot \ldots \cdot J6 \cdot handcoords \cdot toolcoords \\
&= (send\ eta3\ :copy-coords) \\
T_t &= toolcoords \\
&= (manipulator-toolcoords\ eta3) \\
T_t^{-1} &= toolcoords^{-1} \\
&= (manipulator-toolinverse\ eta3) \\
T_h &= handcoords \\
&= (manipulator-handcoords\ eta3)
\end{aligned}
$$

各関節は、Brep で表現された幾何モデルを保持している。しかし、頂点の座標、平面の方程式は常に現状を反映しているとは限らない。マニピュレータに対する移動、回転などのメッセージでは座標系の更新だけを行い、頂点の座標は変化しない。これは、移動、回転が複数回続けて起こった場合の計算量を減らすためである。更新は、マニピュレータに:worldcoords メッセージを送ることで引き起こされる。

マニピュレータは、手先座標系で動作を指定することを主な目的としている。関節角による指定には :config を用いる。引き数には 6 要素の列を与える。

```
(send eta3 :config (float-vector pi/2 pi/2 0 1 0 1))
```

:config は、各関節角度が可動範囲に収まっていることを検査した後、それらを回転させる。この結果、マニピュレータの管理している座標系と関節角度から定まる実際の手先の位置姿勢とが一致しなくなる。両者を一致させるためには、:set-coords メッセージを送る。:set-coords は、関節角度から順方向のキネマティックスを計算し、最終的な手先座標系に対してさらにアーム解を解く。

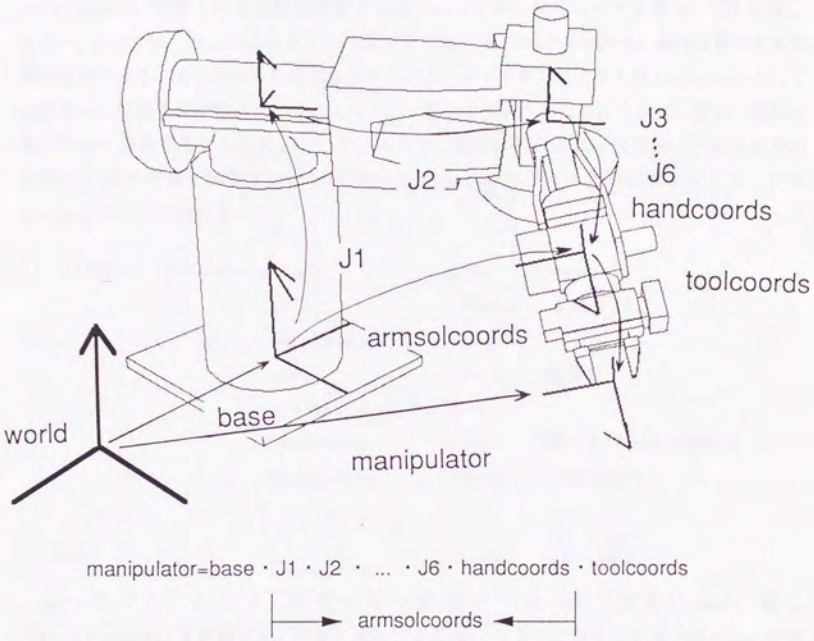


図 3.18: マニピュレータのベース、関節、ハンド、工具座標系の関係

3.6 作業環境のモデリング

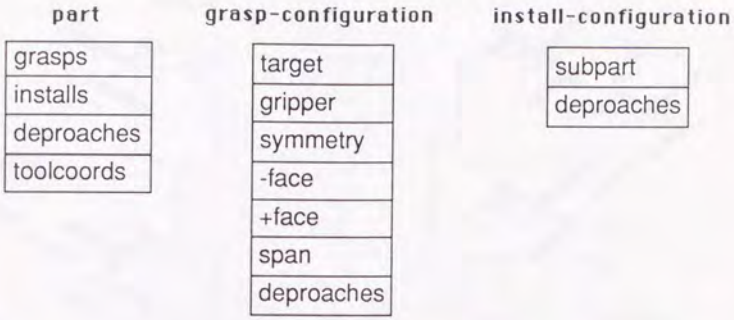
3.6.1 作業対象物体のモデル

ロボットの作業としては、物体の組み立て、分解を対象とする。作業対象物体は部品として、クラス part によって定義される。クラス part は、body をスーパクラスとし把握点のリスト (grasps)、他の物体の装着点のリスト (installs)、持ち上げる時の離脱点 (deproaches)、把握した後の制御座標系 (toolcoords) などのスロットを持つ。(図 3.19)。grasps、installs、deproaches については次節で述べる。toolcoords は、物体に依存する制御座標系であり、その物体を把持するとマニピューレータオブジェクトの toolcoords として設定され、以後その座標系でマニピュレータが動作するようになる。これは、物体の把握は他の物体に装着することを目的としているので、装着に都合の良い座標系、たとえば装着の際に接触する面を制御するために用いる。part オブジェクトを定義するには、次の defpart マクロを用いる。

```
(defpart part-name :shape            body      ;Brep
                   :color            color-id  ; 1 から 15
                   :material         symbol
                   :mass             number    ;Kg 単位
                   :specific-gravity position
                   :departure        coords    ; 持ち上げる時の離脱点
                   :toolcoords       coords    ; 工具座標系
                   )
```

part オブジェクトは、属性リストに材質 (:material)、質量 (:mass)、重心 (:specifig-gravity) を記録する。質量、重心が defpart マクロで与えられない場合は、形状モデルから体積を求め、比重表から材質の比重を知り、質量、重心が自動的に計算される。この場合、物体は均一の材質から構成されていると仮定される。

defpart によって部品が作成されると、それに対して把握点、装着点を定義することができる。把握点、装着点は次に述べるクラス grasp-configuration およびクラス install-configuration によって表現され、そのリストが grasps、installs に記録される。これらの情報から、part オブジェクトに対しては、その把握点、離脱点、どのような部品を取り付けることができるか、現在どの部品が取り付けられているか、何に取り付けられているか、などを問い合わせることができる。



(part は、body のスロットを、grasp-configuration、install-configuration は cascaded-coords のスロットを除いて記述されている)

図 3.19: 作業対象物体、把握点、装着点を表現するオブジェクト

3.6.2 把握点、装着点および接近・離脱点

組み立て作業では、「部品 A を部品 B に取り付ける」といった pick-and-place 動作が基本操作となる。この操作は、図 3.20に示すように、次の六つのステップに分解される。

接近 1 ハンドは何も把持していない。A を把握するために、自由空間で A のアプローチ点へ移動する。把握するための指幅より広く指を開く。

把握 A の把握姿勢へゆっくりと移動する。指を閉じる。

離脱 1 A を把握したまま A の離脱点へ移動する。

接近 2 A を把持したまま、自由空間で A を B に取り付けるための接近点に移動する。

装着 A を B に装着する姿勢へゆっくりと移動する。指を開いて A を解放する。

離脱 2 指を開いたまま A の把握姿勢に対する離脱点に移動する。指を閉じる。

把握動作は、対象物体に付着した固有な座標系への移動として定義することができる。ただし、把握姿勢はひとつの物体に複数存在し、各々が異なる指幅を必要とするので、それらは part オブジェクトにリストとして与えておくのが都合が良い。装着動作は、二つの物体に対してやはり複数定まるので、装着対象物体へのポインタを持ったオブジェクトのリスト

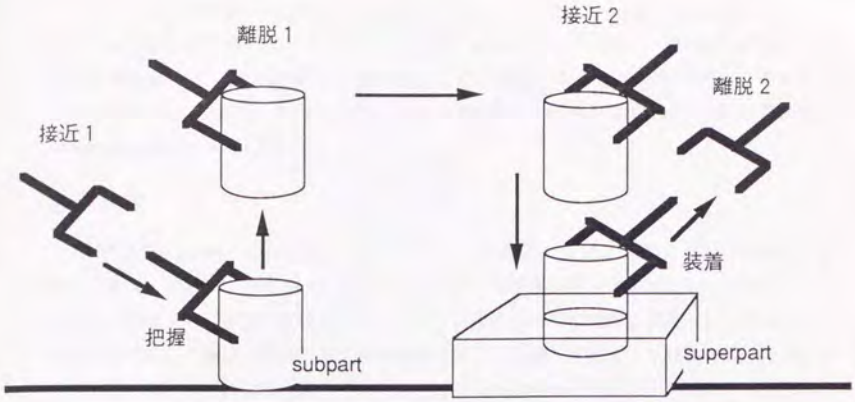


図 3.20: 組み立ての基本操作

として与えなければならない。そこで、把握姿勢、装着姿勢は grasp-configuration、install-configuration クラスによって定義し、そのインスタンスのリストを part オブジェクトに対して与える。

各々に対する接近点と離脱点を考えると、一般に両者は同じでよいと捉えられ易いが、それはハンドが何も把持していない状態で物体を把握に行く時の接近点 (接近 1) と、物体を置いてやはりハンドが何も把持していない状態で離脱する場合 (離脱 2) が同じであるだけである。物体を把握している場合は、把握物体の幾何的性質によって接近点と離脱点が等しくなるとは限らない。したがって、接近・離脱点は必ずしもある物体に固有の姿勢と考えることはできず、何を把握しているか、どういう目的でその物体に接近しているか、という文脈を同時に考慮する必要がある。そこで、離脱 1 は part オブジェクトに与え、接近 1 は把握姿勢の固有座標系、接近 2 は装着姿勢の固有座標系、離脱 2 は把握姿勢の接近姿勢と等しいとするのが合理的である (図 3.21参照)。 pick-and-place 作業では、接近・離脱点の選択が非常に重要であり、逆に適当な接近、離脱点が与えられれば、殊更に多くの move-to 命令を挿入することなく、ほとんどの pick-and-place 作業が簡単なマクロで記述できる。このメカニズムが実際に有効に働く例については、 5.2.3節で述べる。

把握姿勢はクラス grasp-configuration で、装着姿勢はクラス install-configuration で定義する。両者はクラス deproachable-coords をスーパクラスに持つ (図 3.16参照)。さらに deproachable-coords は、 cascaded-coords をスーパクラスにしており、把握対象物体 (part) の子座標系として接続される。したがって、 grasp-configuration オブジェク

ト、あるいは install-configuration オブジェクトに対して:worldcoords メッセージを送れば part の位置・姿勢を連結した把握、装着姿勢のワールドでの表現が得られるし、両姿勢にさらに他の座標系を子座標系として連結することが可能である。そのような子座標系として接近・離脱点があり、その簡便な定義を可能にしているのがクラス deproachable-coords である。

把握点定義

把握点は、grasp-configuration オブジェクトによって記述される (図 3.19参照)。把握点の位置・姿勢は、次の defgrasp マクロによって定義する。マニピュレータのグリッパが物体を把握する姿勢を、物体からハンドの工具座標系への相対的な変換として与える。一つの物体に対して複数の把握点を定義する場合は、:name 引数を変えた複数の defgrasp マクロを用いる。defgrasp によってクラス grasp-configuration のインスタンスが作成され、part-name にバインドされる。

```
(defgrasp part-name  :name      integer    ; grasp-id
                     coords                 ;trans.  from part to grasp
                     :approach  coords )
```

defgrasp によって、把握の姿勢が与えられると、物体の形状モデルを用いて、以下の方法で指幅が決定される (図 3.22参照)。

1. 指が開閉する方向 (把握姿勢の y 軸) に直線をのばす
2. 1 の直線と物体のすべての面との交点を求める。
3. 交点のうちで最も外側にあるペアを選ぶ
4. 3 の交点が乗っている面が指と対抗するかどうか、検査する。
5. 交点の距離を求める
6. 交点を結ぶ中点を把握姿勢の原点に定める

こうして求められた二つの把握面は、grasp-configuration オブジェクトの +face、-face スロットに、把握指幅は span スロットに記録される。

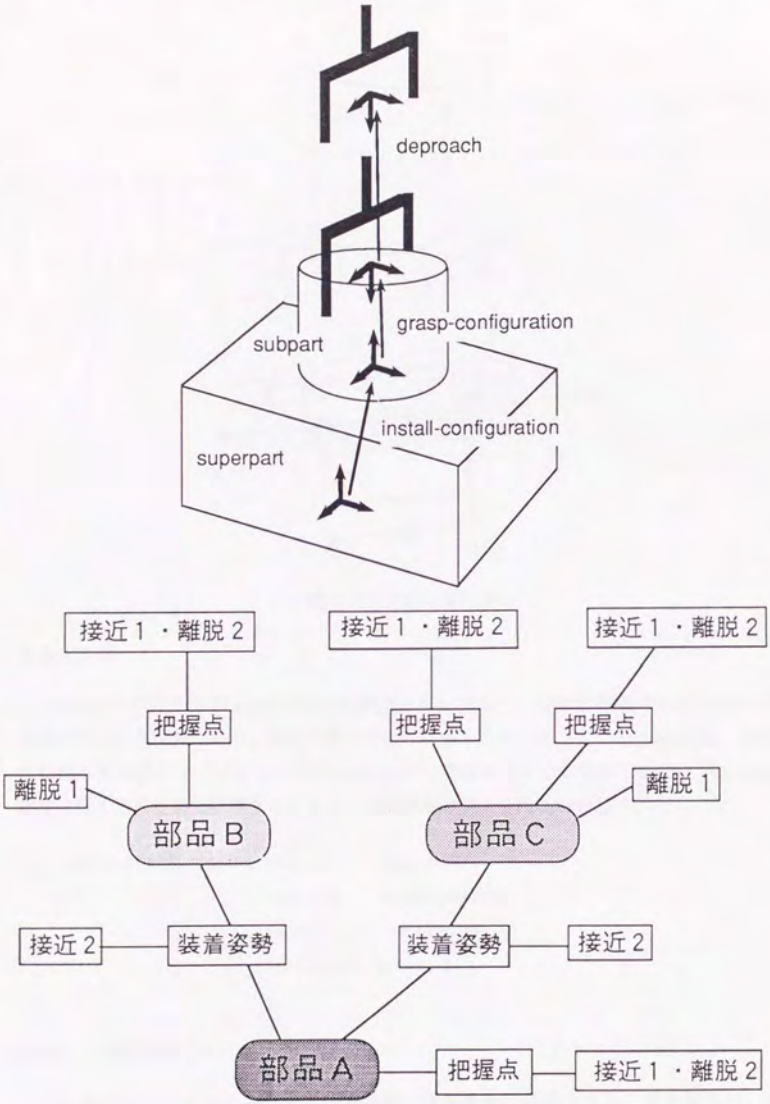


図 3.21: 部品、把握点、装着点と接近、離脱点の関係

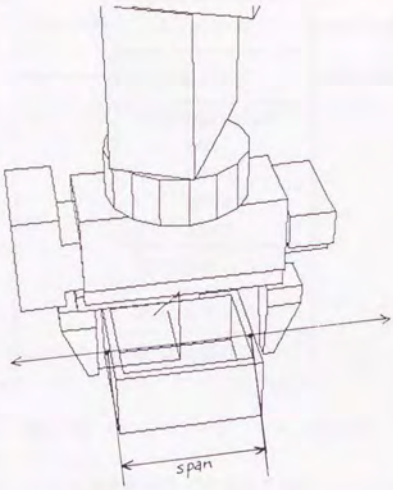


図 3.22: 把握指幅の決定

装着点定義

superpart に対する *subpart* の相対座標系を指定する。この座標系は、マニピュレータの把握姿勢とは無関係であり、純粋に物体同士の関係を記述する。また *approach* は、*subpart* を把握した状態のマニピュレータが *superpart* に接近動作をする姿勢であり、*superpart* を把握に行く時の姿勢とは異なる。これらの座標系の関係を図 3.21 に示す。

```
(definstall subpart :on          superpart
                    :coords      install-position
                    :name        install-name
                    :approach    coords )
```

3.6.3 時系列ワールド

ワールドは、マニピュレータ、対象物体、環境物体で構成される。環境物体は、作業を通じて移動せず、単なる背景として扱える物体を意味する。ワールドは、クラス `manipulator-world` によって記述される。図 3.23に示すように、`manipulator-world` は、マニピュレータと対象物体の位置・姿勢、固着関係、前後のワールドを記録している。

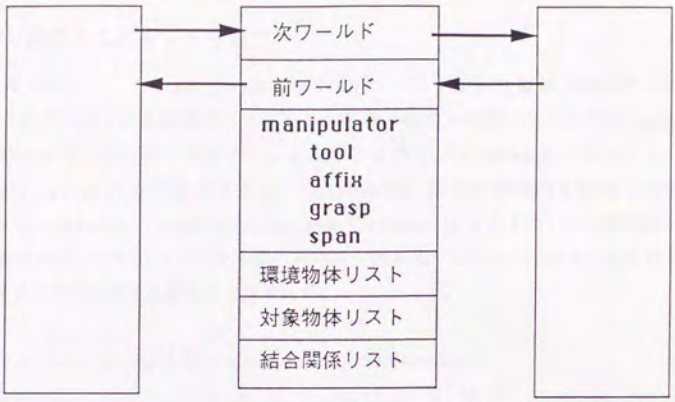


図 3.23: マニピュレータワールドの構成

manipulator-world は、クラス queue のサブクラスであり、クラス queue は、クラス cons のサブクラスである。したがって、manipulator-world は一見リストのように扱うことが可能であり、car を適用すれば先頭のワールドが、cdr を適用すれば次以降ののワールドが取り出せる。manipulator-world は、次の defworld マクロによって作成される。

```
(defworld world-name
                :manipulator      manipulator
                :environment      list-of-environment-objects
                :objects          list-of-handling-objects
                )
```

ワールド中のマニピュレータ、作業対象物を動かすとワールドの状態は次第に変化する。そこで snap-world を実行すると、ワールドのスナップショットが取られ、新しい manipulator-world オブジェクトが作成される。また、reveal によって任意の時刻のワールドが復元できる。この機能は、動作プログラムを対話的に編集する場合に必要となる。

```
(move-to ....)          ;move manipulator
(snap-world kitzw)      ;new-world is kept in the after slot of kitzw
(reveal kitzw)          ;go back to original states
```

3.7 作業プログラム

3.7.1 仮想マニピュレータ命令

仮想マニピュレータ命令は、一般的なマニピュレータが論理的に実行できる動作コマンドである。以下に各仮想マニピュレータ命令の機能を列挙する。引数の trans は、coordinates オブジェクトである。wrt は、任意の coordinates オブジェクト、または:local、:world のいずれかである。:purpose は、動作の意味的な目的を表すキーワードで、:approach、:departure、:grasp、:place、:via がある。この情報は、次の3.7.2節で述べるマクロコマンドを展開する時に付加され、シミュレーションおよび実マニピュレータ命令を生成する段階で利用される。

move-to *purpose object coords wrt :velocity :acceleration*

move-to は、ハンドの手先の座標系を移動させる。*purpose* には、:via、:approach、:grasp、:place、:departure、:park がある。:via は、中間通過点への移動を表す。:approach は、把握、装着の前のアプローチ点への接近動作を、:departure は、逆に把握、装着点からアプローチ点への移動を表す。:grasp は、アプローチ点から把握姿勢への移動を、:place は、アプローチ点から装着点への移動を表す。*object* には、対象となる物体(parts)を指定する。シミュレータにとっては、*purpose* が何であろうと move-to に変わりはないので *purpose* と *object* の情報を無視するが、仮想マニピュレータ命令から実際のマニピュレータ命令を生成する時には、これらの情報を利用して、マニピュレータの制御モードや移動速度が適当に選択される。*coords* と *wrt* は、move-to の目的地を coordinates オブジェクトによって指定する。*wrt* は基準座標系の指定であり、parts オブジェクト、任意の座標系、:local または:world を指定することができる。

gripper *purpose span abs*

gripper は、指の開閉を行なう。*purpose* には、:grasp、:release、:approach、:open がある。*span* は指の間隔(単位 mm)、*abs* は nil のとき現在の指幅に相対的に、t の時絶対指定となる。*purpose* が:grasp のとき、シミュレータは指幅を指定された値にするが、実際のマニピュレータ命令を生成するときは、グリッパが一定の力で把握するような命令が出力される。

affix *subpart superpart*

subpart を *superpart* に結合させる。もし *subpart* がすでにハンドに affix されている場合、ハンドから *subpart* を unfix した後、*superpart* に affix する。この処理は、クラス

cascaded-coords の:assoc、:dissoc メソッドを用いて行われる。

unfix *subpart*

subpart とハンドの結合を切り離す。

set-tool *coords*

マニピュレータに新しい工具座標系 (toolcoords) を設定する。

reset-tool

マニピュレータの工具座標系を定義時のデフォルトに戻す。

3.7.2 マクロ動作記述

作業の記述には、pick-up、place 関数を用いる。

```
(pick-up target :on              superpart
                :tool            toolpart
                :via             list-of-via-points
                :approach        approach-coords
                :at              grasp-position
                :velocity        mm-per-sec
                :acceleration    mm-per-square-sec
                :span            finger-width
                :span-margin     finger-width-margin
                :manual-move     occasion)

(place target   :on              superpart
                :tool            toolpart
                :via             list-of-via-points
                :approach        approach-coords
                :velocity        mm-per-sec
                :acceleration    mm-per-square-sec
                :span            finger-width
                :span-margin     finger-width-margin
                :manual-move     occasion)
```

pick-up、placeは、前節の動作プリミティブを表現するmotionオブジェクトに展開される。基本的には、もし指定されていれば中間通過点への移動、次に接近点への移動、把握または装着点への移動、指の開閉、離脱点への移動といった動作列に分解される。実際にどの物体の装着点を参照するか、またどの点を作業座標系に取るかは、*target*と:on、:tool節の指定によって異なる。また:manual-move節は、センサを持たないマニピュレータに代わってオペレータが位置決めの誤差を補正する場面を指定するのに用い、:approach、:grasp、:release、:departureのいずれかをリストで与える。これによって、各々の動作の前に一旦マスタスレーブモードになるようなコードが挿入される。:toolの指定がなければ、target部品のtoolcoordsがデフォルトで設定される。例を用いた詳細については、5.2.3節で述べる。

全体の作業手順は、*defplan*マクロによって定義する。

(defplan *plan-name initial-world* . *manipulator-commands*)

*initial-world*には、初期状態を表すmanipulator-worldオブジェクトを、*manipulator-commands*には、仮想マニピュレータ命令あるいはpick-up、placeマクロによる作業手順を記述する。*defplan*マクロによって、初期ワールドと作業手順を記録した*taskplan*クラスのインスタンスが作成され、*plan-name*にバインドされる。たとえば、弁の交換作業は次のように記述される。

```
(defplan kitz kitzw
        (pick-up kitzdiaphragmholder)
        (place   kitzdiaphragmholder  :on   kitztable)
        (pick-up kitzhead             :tool olddiaphragm)
        (place   olddiaphragm         :on   kitzbox)
        (pick-up newdiaphragm         :on   kitzrod)
        (place   kitzhead             :on   kitzbody))
```

3.7.3 シミュレーションと教示

taskplanクラスは、作業をシミュレートする:simulateメソッドを定義している。:simulateメッセージが送られると、pick、placeマクロが展開され、その結果得られる仮想マニピュレータ命令がシミュレーションされる。この仮想マニピュレータ命令は、シミュレーションの結果として生ずるmanipulator-worldと共にmotionオブジェクトに

キューとして記録される。このワールドの状態は、MMD 等のグラフィクスディスプレイ上で見ることができる。

さらに、展開された動作を 1 ステップずつ確認しながら実行させ、動作を変更したり、新たな中間通過点を挿入する作業は MMD とマウス、メニューを用いて行なう。シミュレーションを通じて、`*interpolate*` パラメータを T にしておけば、補間動作がシミュレートされる。また、`*interference-check*` パラメータに part オブジェクトのリストをセットしておくと、各補間動作毎にソフトウェアによる part 同士の干渉検査が行われる。

実際のマニピュレータの命令を生成するには、仮想マニピュレータ命令から特定のマニピュレータの命令に変換する機能を持ったクラスを `taskplan` クラスのサブクラスとして定義し、`defplan` 時に `*taskplan-class*` 変数にそのクラスをバインドしておく。たとえば、ETA3 アームの命令を生成するクラスを定義しておけば、`:eta3motions` メッセージを送ると、`motion` オブジェクトが再スキャンされて、ETA3 マニピュレータの動作プリミティブに変換される。完成された eta3motions リストは、そのまま実際のマニピュレータの動作コマンドとして解釈実行することができる。例を用いた詳細については、5.2 節、5.3 節で論ずる。

3.8 3 章の結論

EusLisp を用いて、ロボット用のモデルを記述するシステムについて述べた。システムはオブジェクト指向の概念に基づいて、階層的に定義された座標系、形状モデル、マニピュレータ、作業対象物、作業環境ワールド、タスクの進行を表現するクラスによって構成される。オブジェクト指向によって、高い拡張性を維持しつつ、豊富な機能をコンパクトに実現することができた。

形状モデルは、Brep によって表現される。Brep の要素となる稜線、面、物体がオブジェクトになっている。Lisp のリスト処理の特性を活かすことで冗長な winged-edge 表現を簡略化でき、高い記憶効率が得られると共に、記述量を大きく減少させることができた。形状の入力は、プリミティブ物体の作成と、それらの集合的合成操作によって行う。出来上がった形状モデルに対しては、体積、重心等のマスプロパティの導出、面の種類や形状合成手続きの履歴の取りだしが可能になっており、これらをロボットの作業プログラムに利用することができる。従来の、幾何モデルを利用したロボット言語の多くが、別の言語で記述されたソリッドモデラをオフラインで利用していたのに比べ、EusLisp の上に幾何モデルを持てることは、オンラインでの作業計画、編集、シミュレーションにとって大きな利点となる。

幅広い形態の多関節マニピュレータを表現するためのマニピュレータモデル、組み立て作業における対象物体を表現するモデルが用意されている。これらは、クラスの継承機能を利用して簡単にカスタマイズすることができる。対象物体には、把握点、接近・離脱点、装着

点、を定義することができ、これらのモデルを用いて、記号的な作業記述が行える。このマクロ記述は、シミュレーションによって仮想マニピュレータ命令に展開される。各動作の結果生ずる副作用はワールドモデル中に記録され、任意の時点への復帰が可能となっている。これは、対話的な作業編集を効率良く進めるのに有効である。