EusLisp

EusLisp version 9.00/ irteus version 1.00

リファレンスマニュアル

-ロボットモデリングの拡張-

ETL-TR-95-19 + JSK-TR-10-03

平成 27 年 8 月 18 日

irteus 1.00
東京大学大学院
情報理工学系研究科　知能機械情報学専攻
野沢 峻一 , 植田 亮平, 岡田 慧
ueda@jsk.t.u-tokyo.ac.jp nozawa@jsk.t.u-tokyo.ac.jp k-okada@jsk.t.u-tokyo.ac.jp
〒 113-8656 東京都文京区本郷 7-3-1 工学部 2 号館 7 階 73B2
EusLisp 9.00
通商産業省　工業技術院
電子技術総合研究所 知能システム部
松井 俊浩, 原 功, 中垣 博文 (九州電力)
matsui@etl.go.jp, hara@etl.go.jp, nakagaki@etl.go.jp
〒 305 茨城県つくば市梅園 1-1-4

目 次

第 I 部
EusLisp 基本

1 はじめに

EusLisp は、知能ロボットの研究を目的とした言語で、Common Lisp とオブジェクト指向型プログラミングをベースとしている。ロボットの研究では、センサデータの処理、環境認識、障害物回避動作計画、作業計画などが重要なテーマとして取り上げられるが、これらに共通するのは、ロボットと外界の 3 次元幾何モデルである。EusLisp を開発する動機となったのは、記号処理システムから簡単に使えて拡張性の高いソリッドモデラーの必要性を痛感したからである。既存のソリッドモデラーを調べてわかったことは、その実現にとって最も重要な機能は、数値演算ではなくモデル要素のトポロジーを表現、管理するためのリスト処理能力であった。もちろん、幾何演算も重要ではあるが、ベクトル・行列を操作する関数を組み込めば十分であることがわかった。

こうして、ソリッドモデラーは Lisp の上に実現されるべきであると言う基本方針が得られた。さらに、ソリッドモデラーは、3 次元物体モデルを定義し、動きをシミュレートし、物体の相互関係を表現し、グラフィックスを表示する機能を提供するが、先に述べた各種のロボット問題と結合されなければ意味がない。また、ロボットを完成されたシステムとして実現するには、これらのロボット問題を解くモジュールが効果的に統合されなければならない。EusLisp は、この統合の枠組をオブジェクト指向に求めた。オブジェクト指向は、モジュラープログラミングを促進し、継承機能により既存の機能を段階的に拡張することが容易になる。実際、上記のソリッドモデラーは、物体、面、エッジなどの物理的実体の振舞いをクラスに定義し、ロボット問題に依存する機能は、これらをサブクラスに拡張することで効率よく発展させられる。これは、ソフトウェア資源の再利用にもつながる。

こうして EusLisp は、オブジェクト指向と CommonLisp をベースとして 3 次元幾何モデラーを実現し、複数ロボットの協調動作に必要なタスク間通信機能、マンマシンインタフェースに重要なウィンドウ、グラフィックス、複合的プログラミングに必要な他言語インタフェース等を備え、さまざまなロボット問題への適用を可能にしたプログラミングシステムとして構築された。このほか、メモリ管理にも工夫を凝らし、メモリ容量以外に生じる領域の大きさに関する制約を極力排除し、ガーベージコレクションが効率的に行なわれ、ユーザがメモリ管理に関するパラメータを操作する必要がないように努めた。

このリファレンスマニュアルは、EusLisp の基礎と拡張に分かれ、前者が Common Lisp の機能とオブジェクト指向型プログラミングを、後者が幾何モデル、ロボットモデル、ウィンドウ、画像処理など、よりロボット応用に近い部分を扱っている。アップデート情報は、1.6 節に書かれているとおり、Euslisp のメーリングリストから入手することができる。

1.1 EusLisp におけるオブジェクト指向プログラミング

EusLisp は、その他の Lisp を基礎としたオブジェクト指向プログラム言語（例えば CLOS [?]）と異なり、オブジェクト指向を基礎とした Lisp システムである。以前の考え方として、Lisp はオブジェクト指向プログラミングを実現するための言語として使用され、その中でシステムのデータ型がそれ相応のクラスを持っていなかったので、システム定義オブジェクトとユーザー定義オブジェクトとの間に明らかな区別があった。一方、EusLisp 内の数値を除く全てのデータ構造はオブジェクトで表現されている。そして、内部データ型（例えば `cons` や `symbols`）とユーザー定義クラスとの間に特別な違いはない。これは、システムの内部データ型でさえユーザー定義のクラスによって拡張（継承）できることを意味する。また、ユーザーが内部クラスのサブクラスとして自分独自のクラスを定義したとき、その新しいクラスに対して内部メソッドおよび内部関数を使用することができ、新しいプログラムを記述する量を減らすことができる。例えば、キューや tree やスタックな

どを定義するために、car や cdr と異なった特別な部分を持つように cons クラスを拡張したとする。これらのインスタンスでさえ、cons クラスの内部関数が型の継承を一定時間で認識するため、それらの関数をのロス時間なしで適用できる。したがって、EusLisp はシステムの全ての内部機能（拡張可能なデータ型の形式）をプログラマーに公開している。この画一性もまた、EusLisp の実行のために役に立つ。なぜなら、実行言語の中で **defclass** や **send** や **instantiate** のような僅かな核になる関数を定義した後は、内部データ型の内部構造にアクセスするための大部分の関数を EusLisp 自身で書くことができる。これは、EusLisp の確実性および維持性を改善するものである。

1.2 Euslisp の特徴

オブジェクト指向プログラミング EusLisp は、単一継承オブジェクト指向プログラミングである。数値を除いた全てのデータ型は、オブジェクトで表現され、その動作はそれらのクラスの中に定義されている。

Common Lisp EusLisp は、EusLisp のゴールやオブジェクト指向と一致する限りにおいて、[**?**] や [**?**] に書かれている Common Lisp の文法に従う。次の節に互換性について記述する。

コンパイラ EusLisp のコンパイラは、インタプリタによる実行よりも 5〜30 倍実行速度を上げることができる。コンパイラは、インタプリタと同一の構文を持っている。

メモリ管理 フィボナッチバディ方法は、メモリ制御・GC・ロバスト制御に有効なため、メモリ管理手法として、使用されている。EusLisp は、比較的に平均的な量のメモリを持つシステムで動作できる。ユーザーは、それぞれのデータ型毎のページアロケーションの最適化を考える必要がない。

幾何学関数 数値は、いつも直接データとして表現されるため、数値計算によってゴミは発生しない。任意の大きさのベクトル・行列におけるたくさんの幾何学関数は、内部関数となっている。

幾何学モデラー 立体モデルは、CSG の処理を使用して、基本的な形から定義することができる。質量や干渉チェックや接触判別等を備えている。

グラフィックス 描画時の陰線処理やレンダリング時の陰面処理を備えている。画像をポストスクリプトデータとして出力できる。

画像処理 エッジ抽出機能を備えている。

マニピュレータモデル 6 自由度を持つロボットマニピュレータを簡単にモデル化できる。

Xwindow インターフェース Xlib の関数呼び出しと Xlib のクラスおよび独自の XToolKit クラスの 3 つのレベルの Xwindow インターフェースを備えている。

他言語インターフェース Cや他の言語で書かれた関数を EusLisp の中から呼び出すことができる。EusLisp と他言語間の両方向呼びだしを備えている。LINPACK のようなライブラリ内の関数は、このインターフェースを通して実行される。X toolkit の Call-back 関数は Lisp へ定義することができる。

UNIX 依存関数 ほとんどの UNIX のシステム呼びだしおよびライブラリ関数は、Lisp 関数として揃っている。通信処理や非同期入出力も可能である。

マルチスレッド グローバルデータを分割してマルチ処理を実現するマルチスレッドプログラミングが Solaris 2 オペレーティングシステムの上で可能となった。マルチスレッドは、非同期プログラミングを容易にし、実時間応答を改善する [**?**, **?**]。もし、マルチプロセッサのマシン上で Euslisp を実行するならば、並列プロセッサの高い計算能力を利用することができる。

1.3 Common Lisp との互換性

Common Lisp は、よく本となっていて広く入手できる標準的な Lisp[**?**, **?**] となっている。そのため、EusLisp は Common Lisp の特徴をたくさん採用している。例えば、変数スコープルール・パッケージ・列・一般変数・ブロック・構造体・キーワードパラメータなどであるが、非互換もまだ残っている。実現されていない特徴を次に列挙する。

1. 多値変数: multiple-value-call, multiple-value-prog1, etc.
2. いくつかのデータ型: complex number, bignum, ratio, character, deftype
3. いくつかの特殊書式: progv, compiler-let,macrolet

次の特徴は、まだ完全でない。:

4. closure – 動的範囲のみ有効である。
5. declare,proclaim – inline と ignore は認識されない。

1.4 開発履歴

1986 EusLisp の最初のバージョンが UNIX-System5/Ustation-E20 上で走った。フィボナッチバディのメモリ管理・M68020 のアセンブラコードを生成するコンパイラ・ベクトル／行列関数がテストされた。

1987 新しい高速型チェック方法が実現された。他言語インタフェースと SunView インターフェースが組み込まれた。

1988 コンパイラは、中間コードとして、C プログラムを生成するよう変更された。コンパイラが中央処理装置と無関係となったため、EusLisp は簡単に Ultrix/VAX8800 や SunOS3.5/Sun3 や/Sun4 の上に移植された。ソケットを使用した IPC 機能が追加された。ソリッドモデラーが実現された。Common Lisp の特徴の大部分が追加された。例えば、キーワードパラメータ、再帰的データオブジェクトを扱うための表示フォーマット、一般列関数、readtables, tagbody, go, flet, や labels special forms, 等。

1989 Xlib インターフェースが作られた。C のような数式表現を読み込む%マクロが作られた。マニピュレータのクラスが定義された。

1990 XView インターフェースが稲葉氏により作成された。レイトレーサが作成された。ソリッドモデラーが CSG 処理履歴を保持するよう修正された。非同期入出力が追加された。

1991 動作拘束プログラムが比留川氏により作成された。DEC station に移植された。Coordinates クラスが 2 次元と 3 次元の両方の座標系を扱えるよう変更された。Body 組立関数が接触オブジェクトを扱えるよう拡張された。接触オブジェクトのための CSG 処理が作られた。パッケージシステムが Common Lisp と互換になった。

1992 2 つの平面の結合や交差を求める **face+**や **face***が追加された。画像処理機能が追加された。リファレンスマニュアルの第一版が発行され、配布された。

1993 Euslisp は、全く変更がなかった。

1994 Solaris 2 に移植された。Solaris のマルチスレッド機能を用いて、マルチタスクが実現された。XToolKit が構築された。マルチロボットシミュレータ MARS が國吉氏により作成された。福岡で開催された日本ロボット学会学術講演会において Euslisp のオーガナイズドセッションが開かれた。

表 1: *eusdir*のディレクトリ

FILES	このドキュメント
README	ライセンス・インストール・サンプル実行のガイド
VERSION	EusLisp のバージョン
bin/	実行ファイル (eus, euscomp と eusx)
c/	C で書かれた EusLisp のカーネル
l/	EusLisp で書かれたカーネル関数
comp/	EusLisp で書かれた EusLisp コンパイラ
clib/	C で書かれたライブラリ関数
doc/	ドキュメント (latex と jlatex のテキストとメモ)
geo/	幾何学関数とグラフィックプログラム
lib/	共有ライブラリ (.so) とスタートアップファイル
llib/	Lisp ライブラリ
llib2/	Lisp ライブラリ 2（UTYO で開発）
xwindow/	X11 インターフェース
makefile@	makefile.sun[34]os[34],.vax 等へのシンボリックリンク
pprolog/	tiny prolog のインタープリタ
xview/	xview tool kit インターフェース
tool/	
vxworks/	リアルタイム OS VxWorks とのインターフェース
robot/	ロボットモデルとロボットシミュレータ (MARS)
vision/	画像処理プログラム
contact/	拘束動作解析　比留川氏作 [?, ?, ?]
demo/	デモプログラム
bench/	ベンチマークテスト用プログラム

1995 リファレンスマニュアルの第二版が発行された。

2010 修正 BSD ライセンスに変更され, バージョンが 9.00 となった.

2011 Darwin OS サポートが追加された．モデルファイルが追加された.

2013 Cygwin 64Bit サポートが追加された, MAXSTACK が 65536 から 8388608 へ、KEYWORDPARAMETERLIMIT が 32 から 128 に拡張された.

2014 文書が UTF-8 になった．バージョンが 9.10 となった.

2015 min/max のエラーチェック，vplus 任意長対応, non-ttyp モードでのメッセージ表示消去. バージョンが 9.11 となった.

2015 ARM サポートが追加された．バージョンが 9.12 となった．class documentation が追加された．バージョンが 9.13 となった．assert 関数の API が変更された．message がオプションになった (defmacro assert (pred &optional message)．バージョンが 9.14 となった.

1.5 インストール

インストールの手続きは、README に記述されている。インストールされるディレクトリ（"/usr/local/eus/" を仮定する）は、グローバル変数*eusdir*に設定される。この場所は、**load** やコンパイラが参照する。

*eusdir*のサブディレクトリは 、表 1 に書いてあるとおり。これらの中で、c/, l/, comp/, geo/, clib/, lib/ や xwindow/は、eus や eusx を作成するときの基本ファイルを含んでいる。その他は、付属ライブラリ・デモプログラム・ユーザーからの寄贈品である。

1.6 ライセンス

EusLisp は以下の修正 BSD ライセンスの元配布されている.

なお 8.25 版までは以下のライセンスで配布されていた.

ユーザーは、メーリングリスト (euslisp@etl.go.jp) に登録され、そこにQ＆A・バグ・アップデート情報を流す。この情報は、`*eusdir*/doc/mails` に蓄積されている。

1. EusLisp の著作権は作者（松井俊浩）および電子技術総合研究所に属する。ユーザーは、作者より使用許可を得る必要がある。

2. 軍事目的以外であればどんな目的のために EusLisp を使用してもよい。

3. EusLisp は ftp 経由で電子技術総合研究所から自由に得ることができる。

4. EusLisp はここの条項を守る限りにおいてコピーまたは販売しても構わない。ただし、販売する際、販売者はオリジナルの EusLisp が無料であることを消費者に通知しなければならない。

5. ライセンス取得者が EsuLisp を使用して研究・学習した結果を公表する際、EusLisp の使用を特定参考文献として引用しなければならない。

6. ライセンス取得者は、EusLisp のソースコードを追加・変更してもよい。プログラム結果は、コードの 50%以上が変更されない限り EusLisp であり、変更していない部分については、これらの条項を守る必要がある。

7. EusLisp で開発されたプログラムの著作権は、開発者に属する。しかしながら、EusLisp 本体の著作権に付け加えることはできない。

8. 作者および電総研は使用に際して、どんな保証もしない。

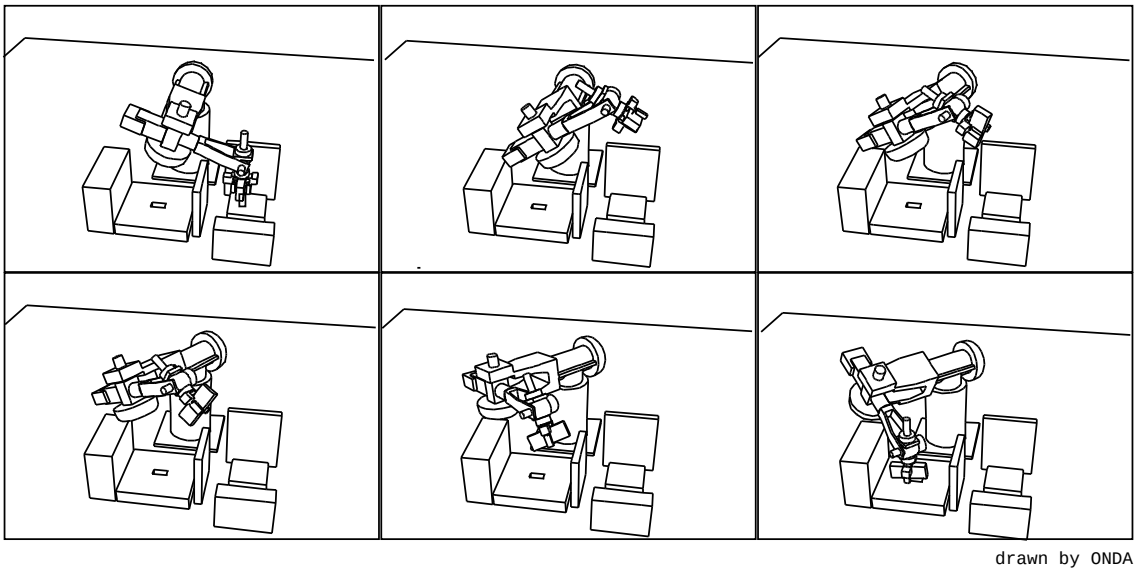

図 1: 衝突回避経路計画のアニメーション

1.7 デモプログラム

デモプログラムは、サブディレクトリ demo の中にある。*eusdir*へ cd した後、eusx 上で実行できる。

ロボットアニメーション eusx より demo/animdemo.l をロードする。約 20 分の計算の後、ETA3 マニピュレータの滑らかなアニメーションが表示される。(図 1)

レートレーシング もし、8 ビットの疑似カラーディスプレイを持っているなら、demo/renderdemo.l をロードしてレイトレーシング画像を楽しむことができる。geo/render.l が先にコンパイルされていることが必要。

エッジ抽出 demo/edgedemo.l をロードすると、サンプル単色画像が表示される。微分オペレータとしきい値を選ぶためのパラメータを入力する。エッジが数秒のうちに探され、元の画像に上書きされる。

2 データ型

他の Lisp と同様に、型の決まったデータオブジェクトは変数ではない。どの変数もその値として、どんなオブジェクトも持つことができる。変数にオブジェクトの型を宣言することが可能であるが、一般的にコンパイラで高速なコードを生成するための情報としてのみ使用される。数値は、ポインタの中で直接値として表現され、そのほかは、ポインタにより参照されるオブジェクトにより表現される。Sun4 で実行する場合、ポインタおよび数値は図 2 で描かれているように long word で表現される。ポインタの LSB の 2 ビットは、ポインタ・integer・float を識別するための tag ビットとして使用されている。ポインタは tag ビットが全てゼロであり、オブジェクトのアドレスとして 32 ビット全て使用できるので、EusLisp は 4GB 以上のアドレス空間を利用することができる。

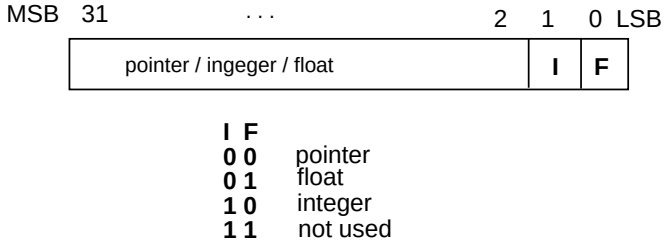

図 2: ポインタと直接値

2.1 数値

数値には、integer と float（浮動小数点）の 2 種類があり、両方とも 29 ビットの値と 1 ビットの符号で表現される。したがって、integer は −536,870,912 から 536,870,911 までの範囲となる。float は、正および負で 4.8E−38 から 3.8E38 までの範囲を表現でき、その有効数字は、十進数で約 6 桁すなわち浮動小数点誤差は 1/1,000,000 程度である。

数値は、いつもオブジェクトでなくポインタで表現される。これは、EusLisp のオブジェクト指向の唯一の例外事項である。しかしながら、数値は決してヒープメモリを無駄にすることがないため、数値を扱うアプリケーションでは、ガーベージコレクションの原因とならず有効に動作する。

EusLisp は、文字型を持たないため、文字列は integer で表現される。文字コード表と無関係なプログラムを書くためには、#\ 読みだしマクロが役に立つ。しかし、文字が読まれるとき、数値表現に変換されるため、プリンタは#\ の表記法に対してどのように再変換すればよいのか解らない。

数値は、図 2 の long word の中に 2 つの tag ビットを持っている。それで、数値計算に使用するときは、シフトまたはマスクすることによりこのビットを消す必要がある。integer は数値シフトにより MSB の 2 ビットを無視し、float はマスクにより LSB の 2 ビットを無視する。VAX のようなアーキテクチャのために Byte swap も必要である。なぜなら、意味を持つ最小の大きさの Byte として右端の 1Byte が使用できないためである。

2.2 オブジェクト

数値でない全てのデータは、ヒープにおかれるオブジェクトで表現される。それぞれのオブジェクトのメモリセルは、オブジェクトヘッダーとオブジェクト変数のための固定数のスロットを持っている。ベクトルは、任意の要素から構成されるため、size スロットをヘッダーのすぐ後に持っている。図 3 はオブジェクトとベクトルおよびオブジェクトのヘッダーを描いたものである。ここに示す *slot* と *element* のワードだけがユーザーからアクセスすることができる。

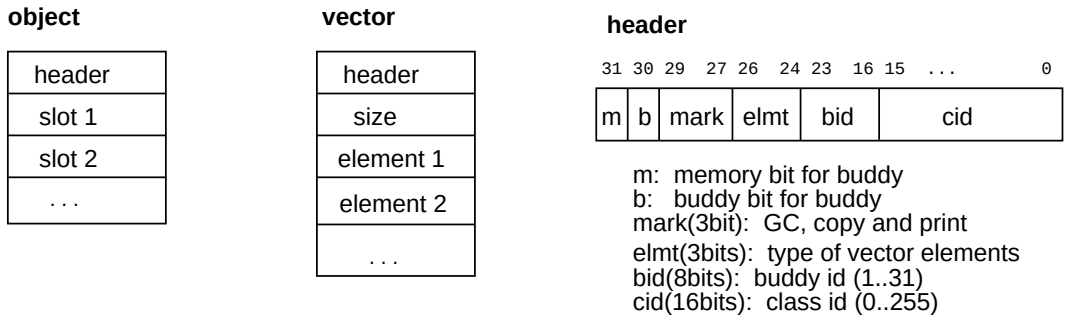

図 3: オブジェクト・ベクトル・オブジェクトヘッダーの構造

ヘッダーは、6 つの部分で構成されている。MSB の 2 ビット m と b は、フィボナッチバディメモリ管理手法の中で、隣接セルの終端を示すために使用される。

mark 部分には、3 つのマークビットがあり、それぞれガーベージコレクタ用のアクセス可能セルの認識、プリンタ用の環状オブジェクトの認識（`#n=`や`#n#`表記法でプリントアウトさせた時）、**copy-object** 用の分割オブジェクトのコピーとして使用される。*elmt* 部分は、ベクトル要素として使用可能な 7 つのデータ型（`pointer, bit, character, byte, integer, float, foreign-string`）のうち 1 つを識別するために使用される。しかしながら、*elmt* はクラスの中で利用可能なため、クラスの構造と無関係なメモリ管理ができ、要素の高速なアクセスができる。*bid* 部分は、メモリセルの物理的大きさを表現する。31 の違った大きさ（16MB 以上）のメモリセルをこの 5 ビットで表現する。下位の short word (16 ビット) は、クラス I D (cid) として使用される。これは、システムのクラステーブルを経由してオブジェクトのクラスを引き出すために使用される。このクラス ID は、伝統的な Lisp の型 tag とみなすことができる。cid は下位 8 ビットのみが使用され、上位 8 ビットは無視される。したがって、クラスの最大数は 256 が限界であるけれども、システムのクラステーブルにもっとメモリを配置するように EusLisp を再構築することによって 65536 まで限界を引き上げることができる。

2.3 クラス継承

オブジェクトのデータ構造はクラスによって定義され、そして、それらの動作はクラス内のメソッドに定義されている。EusLisp において、数ダースのクラスが図 4 に書かれているように木構造化された継承のなかにすでに定義されている。**class-hierarchy** 関数を用いれば、実際の継承構造を見ることができる。左端のクラス object は、EusLisp 内の全てのクラスの根幹となるスーパークラスである。ユーザーが定義したクラスは、これらの内部クラスのどれでも継承することができる。

クラスは、**defclass** マクロか **defstruct** マクロで定義される。

```
(defclass class-name &key :super          class
                          :slots          ()
                          :metaclass      metaclass
                          :element-type   t
                          :size -1
)
(defstruct struct-name slots...)
(defstruct (struct-name [struct-options ...])
               (slot-name1 [slot-option...])
               (slot-name2 [slot-option...])
               ...)
```

```
object
     cons
     propertied-object
          symbol   -----  foreign-pod
          package
          stream
               file-stream
               broadcast-stream
          io-stream ---- socket-stream
          metaclass
               vectorclass
                    cstructclass
          read-table
          array
          thread
          barrier-synch
          synch-memory-port
          coordinates
               cascaded-coords
                    body
                    sphere
                    viewing
                         projection
                              viewing2d
                              parallel-viewing
                              perspective-viewing
                    coordinates-axes
               viewport
          line --- edge --- winged-edge
          plane
               polygon
                    face
                    hole
               semi-space
          viewer
          viewsurface  ----- tektro-viewsurface
     compiled-code
          foreign-code
          closure
          load-module
     label-reference
     vector
          float-vector
          integer-vector
          string
               socket-address
               cstruct
          bit-vector
          foreign-string
     socket-port
     pathname
     hash-table
     surrounding-box
     stereo-viewing
```

図 4: 定義済みのクラス継承

メソッドは、**defmethod** により定義される。**defmethod** は、特定のクラスについて何度でも存在することができる。

```
(defmethod class-name
(:method-name1 (parameter...)  .  body1)
(:method-name2 (parameter...)  .  body2)
...)
```

内部クラスにおける field 定義は、大部分が*eusdir*/c/eus.h のヘッダーファイルの中にある。

クラスは、(describe) 関数によりクラス内の全てのスロット、名前、スーパークラス、スロット名、スロット型、メソッドリスト、などを表示することができる。内部クラスの定義は次の通りである。クラス object はスーパークラスを持たないため、このスーパークラスは NIL である。

```
(defclass object :super NIL :slots ())

(defclass cons :super object :slots (car cdr))

(defclass propertied-object :super object
          :slots (plist))       ;property list


(defclass symbol :super propertied-object
          :slots (value          ;specially bound value
                  vtype          ;const(0),var(1),special(2)
                  function       ;global func def
                  pname          ;print name string
                  homepkg))      ;home package


(defclass foreign-pod :super symbol
          :slots (podcode          ;entry code
                  paramtypes       ;type of arguments
                  resulttype))


(defclass package :super propertied-object
          :slots (names          ;list of package name and nicknames
                  uses           ;spread use-package list
                  symvector      ;hashed obvector
                  symcount       ;number of interned symbols
                  intsymvector   ;hashed obvector of internal symbols
                  intsymcount    ;number of interned internal symbols
                  shadows        ;shadowed symbols
                  used-by))      ;packages using this package


(defclass stream :super propertied-object
          :slots (direction    ;:input or :output, nil if closed
```

```
                buffer      ;buffer string
                count       ;current character index
                tail))      ;last character index


(defclass file-stream :super stream
          :slots (fd       ;file descriptor (integer)
                  fname))  ;file name str; qid for msgq


(defclass broadcast-stream :super stream
          :slots (destinations))       ;streams to which output is e livered

(defclass io-stream :super propertied-object
                  :slots (instream outstream))

(defclass socket-stream :super io-stream
                      :slots (address))            ; socket address

(defclass read-table :super propertied-object
          :slots (syntax       ; byte vector representing character types
                               ; 0:illegal, 1:white, 2:comment, 3:macro
                               ; 4:constituent, 5:single_escape
                               ; 6:multi_escape, 7:term_macro, 8:nonterm_macro
                  macro        ;character macro expansion function
                  dispatch-macro))


(defclass array :super propertied-object
          :slots (entity       ;simple vector storing array entity
                  rank         ;number of dimensions:  0-7
                  fillpointer  ;pointer to push next element
                  offset       ;offset for displaced array
                  dim0,dim1,dim2,dim3,dim4,dim5,dim6)) ;dimensions


(defclass metaclass :super propertied-object
          :slots (name      ;class name symbol
                  super     ;super class
                  cix       ;class id
                  vars      ;var name vector including inherited vars
                  types     ;type vector of object variables
                  forwards  ;components to which messages are forwarded
                  methods)) ;method list


(defclass vectorclass :super metaclass
          :slots (element-type    ;vector element type 0-7
                  size))          ;vector size; 0 if unspecified
```

```
(defclass cstructclass :super vectorclass
          :slots (slotlist))    ;cstruct slot descriptors


(defclass vector :super object :slots (size))

(defclass float-vector :super vector :element-type :float)

(defclass string :super vector :element-type :char)

(defclass hash-table :super propertied-object
          :slots (lisp::key           ;hashed key vector
                  value               ; value vector
                  size                ; the size of the hash table
                  count               ; number of elements entered in the table
                  lisp::hash-function
                  lisp::test-function
                  lisp::rehash-size
                  lisp::empty lisp::deleted ))


(defclass pathname :super propertied-object
          :slots (lisp::host device           ; not used
                  directory                   ; list of directories
                  name                        ; file name before the last "."
                  type                        ; type field after the last "."
                  lisp::version))             ; not used


(defclass label-reference      ;for reading #n=, #n# objects
          :super object
          :slots (label value unsolved next))


(defclass compiled-code :super object
          :slots (codevector
                  quotevector
                  type          ;0=func, 1=macro, 2=special
                  entry))       ;entry offset

(defclass closure :super compiled-code
                :slots (env1 env2));environment

(defclass foreign-code :super compiled-code
          :slots (paramtypes       ;list of parameter types
                  resulttype))     ;function result type

(defclass load-module :super compiled-code
        :slots (symbol-table    ;hashtable of symbols defined
                object-file     ;name of the object file loaded, needed for unloadin
                handle))        ;file handle returned by ''dlopen''
```

2.4 型指定

Euslisp は、**deftype** 特殊書式を持っていないけれども、型名は宣言や結果あるいは中身の型の指定を要求する関数の中で使用される。例えば、**coerce, map, concatenate, make-array** など。一般に、クラス名は `(concatenate cons "ab" "cd") = (97 98 99 100)` のように型指定として使用することができる。このとき、Common Lisp では `cons` の代わりに `(quote list)` を使用する。

Euslisp は、数を表現するクラスを持っていないので、数の型はキーワードによって与える必要がある。**:integer**, **integer**, **:int**, **fixnum**, あるいは**:fixnum** が整数型を表現するために使用され、**:float** あるいは **float** が実数型を表現するために使用される。**make-array** の *element-type* 引数においては、文字列を作るために**:character**, **character**, **:byte** や **b**yte を認識する。**defcstruct**, **sys:peek** や **sys:poke** のような低レベルの関数も、バイト毎にアクセスするために**:character**, **character**, **:byte** あるいは **byte** を認識し、short word 毎にアクセスするために**:short** あるいは **short** を認識する。どの場合においても、キーワードは pname と同じ名前を持つ lisp パッケージの symbol を選ぶべきである。

3 書式と評価

3.1 アトム (atom)

cons 以外のデータオブジェクトは、たとえ複雑な構造をしていたとしても、すべて atom である。空リストとして（）でしばしば書かれる NIL も atom である。すべての atom は、symbol を除いていつもそれ自身評価されている。しかしながら、他の Common Lisp の実行のなかでは、atom の評価に引用符を要求されることがある。

3.2 スコープ

すべての symbol は、値と結び付いている。symbol は、主にくくられた文脈から決定される値によって評価される。ここに 2 種類の変数バインドがある。それは、ローカルまたは静的バインドとスペシャルまたは動的バインドである。ローカルにバインドされた変数は **lambda** 書式または **let** や **let***の特殊書式において special と宣言されない限り外から見ることはできない。ローカルバインドは入れ子が可能で、外側のローカルバインドやスペシャルバインドを隠して、最も内側のレベルで定義されている 1 つのバインドのみ見ることができる。スペシャル変数は 2 つの方法で使用される。1 つは、グローバル変数として、もう 1 つは動的に覗けるローカル変数として用いる。このローカル変数は、バインドの効果の中にある限りローカルスコープの外にいてさえ見ることができる。後者の場合、スペシャル変数は **special** で宣言される必要がある。その宣言は、コンパイラだけでなくインタプリタでも認識される。Common Lisp によると、スペシャル変数は不明瞭なスコープと動的な広さを持っていると言われている。

あるスコープのなかで、ローカル変数が存在するとしても、同じ変数名を内部スコープの中で **special** として再宣言することができる。**symbol-value** 関数は、ローカルスコープに構わず special 値を引き出すために使用することができる。**set** 関数は、スペシャル変数としてのみ働く。すなわち、special として宣言していない限り、lambda や let 変数の値を変更するために使用することはできない。

```
(let ((x 1))
   (declare (special x))
   (let* ((x (+ x x)) (y x))
      (let* ((y (+ y y)) (z (+ x x)))
         (declare (special x))
         (format t "x=~S y=~s z=~s~%" x y z) ) ) )
--> x=1 y=4 z=2
```

symbol は、**defconstant** マクロにより定数として宣言することができる。一旦宣言すると、その後値を変更しようとするとエラーが発生する。そのうえ、そのような定数 symbol は、ローカル変数としてさえ変数名として使用されることを禁じられる。NIL や T は、そのような定数の例である。keyword パッケージの symbol は、いつも作成されるときに定数として宣言される。対照的に、**defvar** や **defparameter** マクロは、スペシャル変数として symbol を宣言する。**defvar** は、symbol がバインドされていない時のみ値の初期化を行い、値が既に割り当てられているときは何もしない。それに対して、**defparameter** はいつも値をリセットする。

symbol が参照され、symbol のためのローカルバインドがなかったとき、その special 値は、引き出される。しかしながら、その special 値にまだ値が割り当ててなかったならば、unbound variable エラーが発生する。

3.3 一般化変数

一般的に、どんな値および属性もオブジェクトのスロット（またはスタック）で表現される。スロットの値を引き出すかまたは変えるときは、2 つの基本的な命令、**access** と **update** で行わなければならない。オブ

ジェクトの全てのスロットに対して 2 つの異なった基本命令を定義する代りに EusLisp では、Common Lisp のように、一般化変数コンセプトに基づいた画一的な更新命令を備えている。このコンセプトのなかで、共通書式は、値のアクセス書式あるいはスロットの位置指定として認識される。したがって、それぞれのスロットに対してアクセスする書式を覚えてさえおけば、更新はそのアクセス書式と **setf** マクロを組み合わせることにより実現できる。例えば、car 値をリストの外に取り出すのと同じ様に (setf (car '(a b)) 'c) として **setf** を使用したとき、(car x) は x の car スロットのなかの値を置き換えることに使用することができる。

この方法は、ユーザーが定義したオブジェクト全てに対して適用できる。クラスや構造体が定義されるとき、それぞれのスロットに対する access や update 書式は、自動的に定義される。それらの書式は、それぞれマクロとして定義されている。その名前は、クラス名とスロット名の連結となる。例えば、cons の car は (cons-car '(a b c)) で処理することができる。

```
(defclass person :super object :slots (name age))
(defclass programmer :super person :slots (language machine))
(setq x (instantiate programmer))
(setf (programmer-name x) "MATSUI"
      (person-age x) 30)
(incf (programmer-age x))
(programmer-age x)   --> 31
(setf (programmer-language x) 'EUSLISP
      (programmer-machine x) 'SUN4)
```

行列要素も同じ手法でアクセスすることができる。

```
(setq a (make-array '(3 3) :element-type :float))
(setf (aref a 0 0) 1.0 (aref a 1 1) 1.0 (aref a 2 2) 1.0)
a --> #2f((1.0 0.0 0.0) (0.0 1.0 0.0) (0.0 0.0 1.0))

(setq b (instantiate bit-vector 10))  --> #*0000000000
(setf (bit b 5) 1)
b --> #*0000010000
```

特定のオブジェクトに特別な setf メソッドを定義するために **defsetf** マクロを用意している。

```
(defsetf symbol-value set)
(defsetf get (sym prop) (val) `(putprop ,sym ,val ,prop))
```

3.4 特殊書式

全ての特殊書式は、表 2 にリストされている。**macrolet, compiler-let,** や **progv** は、該当しない。特殊書式は、文脈の評価および制御フローの管理のための基本的な言語構造である。インタプリタとコンパイラは、これらの構造をそれぞれ正しく処理するために特殊な知識を持っている。それに対して、アプリケーションメソッドは全ての関数に対し画一的である。ユーザーは、独自の特殊書式定義を追加することはできない。

3.5 マクロ

マクロは、言語構造を拡張するために役立つメソッドである。マクロが呼び出されたとき、引数は評価されずにマクロの本体（マクロ拡張関数）へ受け渡される。それから、マクロ拡張関数は、引数を拡張し、新しい

表 2: EusLisp の特殊書式

and	flet	quote
block	function	return-from
catch	go	setq
cond	if	tagbody
declare	labels	the
defmacro	let	throw
defmethod	let*	unwind-protect
defun	progn	while
eval-when	or	

書式を返す。結果となった書式は、マクロの外側で再び評価される。引数のリストにマクロまたは特殊書式を用いるとエラーになる。**macroexpand** 関数は、マクロ展開のために使用することができる。

インタプリタのときマクロはゆっくりと実行されるが、コンパイルすることにより実行速度の向上を図ることができる。なぜなら、マクロ展開はコンパイル時に一度だけ行われ、実行時にそのオーバーヘッドは残らない。しかし、マクロ関数の中における eval あるいは apply の呼出は、インタプリタの実行とコンパイル後の実行との間に違う結果をもたらす。

3.6 関数

関数は、単にリストの最初の要素が **lambda** であるような lambda 書式によって表現される。もし lambda 書式が **defun** を使って symbol を定義するとき、グローバル関数名として参照することができる。lambda 書式は、次の文法で与えられる。

```
(lambda ({var}*
         [&optional {var | (var [initform])}*]
         [&rest form]
         [&key {var | (var [initform]) | ((keyword var) [initform])}*
               [&allow-other-keys]]
         [&aux {var | (var [initform])}*])
       {declaration}*
       {form}*)
```

ここに EXPR,LEXPR,FEXPR などのような型の関数はない。関数への引数は、いつもその関数を実行する前に評価される。受ける引数の数は、lambda-list によって決定される。lambda-list は、lambda 書式のためにパラメータの列を記す。

&optional, &rest, &key や**&aux** はそれぞれ、lambda-list のなかに特殊な意味を持っていて、これらの symbol は、変数名として使用することはできない。&optional や&key パラメータの supplied-p 変数は、サポートされていない。

lambda 書式は、普通のリストデータと区別できないため、**function** 特殊書式を用いて、インタプリタやコンパイラに関数として認識するように知らせなければならない。[1]**function** は、関数の上に環境を固定するた

[1]CLtL-2 のなかで、引用式 lambda 書式は、もはや関数でない。そのような書式の適用はエラーとなる。

めに重要である。そのため、すべてのローカル変数はその関数が違ったローカルスコープの他の関数を通ってきたとしてさえ、アクセスすることができる。次のプログラムは、let の sum が lambda 書式の中に見えるため、インタプリタとコンパイル後のどちらも何もしない。

```
(let ((x '(1 2 3)) (sum 0))
  (mapc '(lambda (x) (setq sum (+ sum x))) x))
```

予想した結果が得られるためには、次のように書くべきである。

```
(let ((x '(1 2 3)) (sum 0))
   (mapc #'(lambda (x) (setq sum (+ sum x))) x ))
```

#' は、**function** の略語である。すなわち、#'(lambda (x) x) は (function (lambda (x) x)) と同等である。ここは、funarg 問題と呼ばれる別の例を示す。

```
(defun mapvector (f v)
    (do ((i 0 (1+ i)))
       ((>= i (length v)))
       (funcall f (aref v i))))
(defun vector-sum (v)
    (let ((i 0))
       (mapvector #'(lambda (x) (setq i (+ i x))) v)
       i))
(vector-sum #(1 2 3 4)) --> 10
```

EusLisp の closure は、不定な大きさを持つことができない。すなわち、closure はその外側の大きさで可能な大きさまで持つことができる。これは closure が'generators' のプログラミングのために使用されないことを意味する。次のプログラムは何もしない。

```
(proclaim '(special gen))
(let ((index 0))
   (setq gen #'(lambda () (setq index (1+ index)))))
(funcall gen)
```

しかしながら、その同じ目的がオブジェクト指向プログラミングで実現できる。なぜなら、オブジェクトはそれ自身の固定変数を持つことができるためである。

```
(defclass generator object (index))
(defmethod generator
 (:next () (setq index (1+ index)))
 (:init (&optional (start 0)) (setq index start) self))
(defvar gen (instance generator :init 0))
(send gen :next)
```

4 制御構造

4.1 条件文

and,or および **cond** は、Common Lisp においてマクロとして知られているが、EusLisp ではインタプリタ時の効率を改善するために特殊書式として実行される。

and {*form*}* [特殊]
form は、NIL が現れるまで左から右に評価される。もし、全ての書式が non-NIL として評価されるならば、最後の値が返される。

or {*form*}* [特殊]
form は、non-NIL 値が現れるまで左から右に評価される。そして、その値が返される。もし、全ての書式が NIL として評価されるならば、NIL を返す。

if *test then [else]* [特殊]
if は、１つの *then* と *else* 書式のみを持つことができる。そこに多重書式を書きたいときは、**progn** を使ってグループ化しなければならない。

when *test forms* [マクロ]
if と違って、**when** と **unless** は、多重**書式**で書くことを許可している。*test* の評価が non-NIL のとき、**when** は実行され、評価が NIL のとき、*unless* は実行される。もう一方で、これらのマクロは *else* 書式を追加することを許可していない。

unless *test forms* [マクロ]
`(when (not test) . forms)` と同等である。

cond *(test* {*form*}*)* [特殊]
任意の数の条件項は、**cond** の後に続けることができる。それぞれの条件項において、最初の書式 *test* が評価される。もし、non-NIL であったとき、その条件項の残りの書式は、続いて評価される。そして、最後の値が返される。もし、*test* のあとに書式がなかったならば、*test* の値が返される。*test* が失敗したとき、次の条件項は *test* が non-NIL 評価されるかまたは全ての条件項が尽きてしまうまで繰り返される。条件項が尽きてしまった場合、**cond** は NIL を返す。

case *key* {*({label* | *({lab}*) {form}*)}* [マクロ]
key と *label* が一致した条件項について、*form* が評価され、最後の値が返される。*key* と *label* の間の等価は、**eq** または **memq** で行われ、**equal** ではない。

4.2 逐次実行と Let

prog1 *form1 &rest forms* [関数]
form1 と *forms* は、次々と評価され、*form1* から返される値が **prog1** の値として返される。

progn {*form*}* [特殊]
form は次々に評価され、最後の *form* の値が返される。**progn** は特殊書式である。なぜなら、ファイルの最初に現れたとき特別な意味を持つからである。そのような書式がコンパイルされたとき、内部書式はすべて最初に現れたとして見なす。マクロが **defun** や **defmethod** の連続で拡張される場合、それが最初に現われなければならないときに役立つ。

setf {*access-form value*}* [マクロ]
value を一般化変数 *access-form* に割り当てる。

let *([{var | (var [value])}*) {declare}* {form}** [特殊]
ローカル変数を生成する。すべての *value* は評価され、並行して *var* に割り当てられる。すなわち、`(let ((a 1)) (let ((a (1+ a)) (b a)) (list a b)))` の結果は (2 1) である。

let* *([{var | (var [value])}*) {declare}* {form}** [特殊]
ローカル変数を生成する。全ての *value* は次々に評価され、*var* に割り当てられる。すなわち、`(let ((a 1)) (let* ((a (1+ a)) (b a)) (list a b)))` の結果は (2 2) である。

4.3 ローカル関数

flet *([{(fname lambda-list . body)}*) {form}** [特殊]
ローカル関数を定義する。

labels *([{(fname lambda-list . body)}*) {form}** [特殊]
ローカルなスコープとなる関数を定義する。**flet** と **labels** との違いは、**flet** で定義されたローカル関数は、その他の関数を参照または再帰できないが、**labels** は相互の参照を許可する。

4.4 ブロックと Exit

block *tag* {*form*}* [特殊]
return-from によって脱出可能なローカルブロックを作る。*tag* は、ローカルにスコープされ、評価されない。

return-from *tag value* [特殊]
tag によって示されたブロックを脱出する。**return-from** は、関数やメソッドから脱出するときに使用される。関数やメソッドは、その本体をすべて取り囲んだ部分をブロックとして自動的に確定され、その関数またはメソッドの名前を付ける。

return *value* [マクロ]
`(return x)` は、`(return-from nil x)` と同等である。**loop, while, do, dolist, dotimes** は、暗黙的に NIL と名前が付けられたブロックとして確定されるため、これらの特殊書式と組み合わせて使用する。

catch *tag* {*form*}* [特殊]
throw によって脱出または値を返すための動的なブロックを確定する。*tag* は評価される。

全て見える **catch** の *tag* は、`sys:list-all-catchers` から得ることができる。

throw *tag value* [特殊]
catch ブロックから脱出または *value* を返す。*tag* と *value* は評価される。

unwind-protect *protected-form* {*cleanup-form*}* [特殊]
protected-form の評価が終った後、*cleanup-form* が評価される。**unwind-protect** の外側にブロックまたは **catch** ブロックを作っても構わない。

return-from や **throw** でさえ、そのようなブロックから抜け出すためには *protected-form* の中で実行される。*cleanup-form* は、評価されることが保証されている。また、もし *protected-form* が実行されている間にエラーが起こったならば、*cleanup-form* はいつも **reset** によって実行される。

4.5 繰返し

while *test* {*form*}* [特殊]

test が non-NIL と評価されている間、*form* は、繰返し評価される。**while** は、*form* のまわりに NIL と名付けられるブロックを自動的に確定する特殊書式である。**return** は、そのループから抜け出すために使用することができる。

tagbody {*tag* | *statement*}* [特殊]

tag は、**go** のために名付けられる。**tagbody** の中のみ **go** を使用することができる。

go *tag* [特殊]

ローカルにスコープされた **tagbody** のなかに現れる *tag* の直後の書式に制御を移す。ローカルスコープを横切って違う **tagbody** の tag に制御を移すことは禁止されている。

prog *(*{*var* | *(var [init])*}*)* {*tag* | *statement*}* [マクロ]

prog はマクロで、以下のように展開される。

```
(block nil (let var (tagbody tag | statement)))
```

do *(*{*(var init [next])*}*) (endtest [result])*{*declare*} {*form*} * [マクロ]

var はローカル変数である。それぞれの *var* に、*init* は並行に評価され、割り当てられる。つぎに、*endtest* が評価され、もし真のとき **do** は *result* を返す。(そうでないときは、NIL を返す) もし *endtest* が NIL を返したならば、それぞれの *form* は、順番に評価される。書式の評価後、*next* が評価され、その値はそれぞれの *var* に再割当され、次の繰返しが始まる。

do* *(*{*var init [next]*}*) (endtest [result])*{*declare*} {*form*}* [マクロ]

do*は、*init* と *next* の評価と *var* への割り当てが連続的に起こることを除いて、**do** と同様である。

dotimes *(var count [result])* {*forms*}* [マクロ]

forms の評価を *count* 回行う。*count* は、一回のみ評価される。それぞれの評価の中で、*var* は整数のゼロから *count*-1 まで増加する。

dolist *(var list [result])* {*forms*}* [マクロ]

list のそれぞれの要素は、*var* に連続的に与えられる。そして *forms* は、それぞれの値で評価される。**dolist** は、他の繰返しより早く実行される。たとえば、**mapcar** や再帰的関数のようなものより。それは、**dolist** が関数の closure を作ったり適用したりする必要がなく、新しいパラメータのバインドが必要でないため。

until *condition* {*forms*}* [マクロ]

condition が満たされている間、*forms* を評価する。

loop {*forms*}* [マクロ]

forms を永遠に評価する。実行を止めるためには、**return-from, throw** または **go** が *forms* のなかで評価されなければならない。

4.6 述語

Common Lisp の **typep** と **subtypep** はないので、**subclassp** や **derivedp** で疑似実現すること。

eq *obj1 obj2* [関数]
obj1 と *obj2* が同じオブジェクトを指すポインタあるいは同じ数値のとき T を返す。例えば:`(eq 'a 'a)` は T、`(eq 1 1)` は T、`(eq 1. 1.0)` は NIL、`(eq "a" "a")` は NIL である。

eql *obj1 obj2* [関数]
EusLisp の中で数値は全て直接値で表現されるため、**eq** と **eql** は同一である。

equal *obj1 obj2* [関数]
いろんな構造のオブジェクトの等価性をチェックする。オブジェクトは、文字列・ベクトル・行列などで再帰的に参照してないことが保証されなければならない。*obj1* や *obj2* が再帰的に参照されていたとすると、**equal** は無限ループとなる。

superequal *obj1 obj2* [関数]
superequal は、環状参照をチェックするので遅い。しかしロバストな等価が得られる。

null *object* [関数]
object が NIL のとき、T を返す。`(eq` *object* `nil)` を評価する。

not *object* [関数]
not は、**null** と同一である。

atom *object* [関数]
オブジェクトが cons のインスタンスである時のみ、NIL を返す。`(atom nil) = (atom '()) = T`
注意:vectors, strings, read-table, hash-table などに対しては、それらがどんなに複雑なオブジェクトとなっていても **atom** は T を返す。

every *pred &rest args* [関数]
全ての *args* が *pred* について T を返した時のみ T を返す。**every** は、*pred* が全ての *args* に対して効力があるかどうかを検査する時に使用される。

some *pred &rest args* [関数]
args のうちどれか 1 つが *pred* について T を返したとき T を返す。**some** は、*pred* が *args* のどれかに対して効力があるかどうかを検査する時に使用される。

functionp *object* [関数]
object が **apply** や **funcall** で与えられる関数オブジェクトであるなら T を返す。
注意:マクロは **apply** や **funcall** を適用することができない。**functionp** は、*object* が、type=0 のコンパイルコードか、関数定義を持つ symbol か、lambda-form かあるいは lambda-closure であったとき、T を返す。`Examples: (functionp 'car) = T, (functionp 'do) = NIL`

compiled-function-p *object* [関数]
object が、コンパイルコードのインスタンスであったとき、T を返す。そのコンパイルコードが関数かまたはマクロかを知るためには、そのオブジェクトに:`type` メッセージを送り、その返り値が `function` と `macro` のどちらになっているかを調べる。

5 オブジェクト指向プログラミング

オブジェクトの構造と動作は、クラスの中に記述されている。それらは、**defclass** マクロや **defmethod** 特殊書式により定義されている。**defclass** は、クラスの名前・そのスーパークラス・スロット変数名とオプションとして任意の型およびメッセージの前方への送信を定義する。**defmethod** は、メッセージが送られてきたとき呼び出されるメソッドを定義する。クラス定義は、symbol の特殊値として割り当てられる。**クラス**は、Common Lisp の **sutructure** の counter 部分と考えることができる。スロットアクセス関数と **setf** メソッドは、**defclass** によってそれぞれのスロットに自動的に定義される。

大部分のクラスは、内部クラス **metaclass** から派生している。**metaclass** のサブクラスであるクラス **vector-class** はベクトルのためのメタクラスである。もし、class-variables や class-methods を使いたいときは、**meta-class** のサブクラスとして自分独自のメタクラスを作り、メタクラスの名前を`:metaclass` のキーワードで **defclass** に与えればよい。

ベクトルは、その他の record-like オブジェクトと違っている。なぜなら、ベクトルのインスタンスは、任意の数の要素を持っているが、record-like オブジェクトは固定数のスロットを持っている。EusLisp のオブジェクトは、record-like オブジェクトかまたはベクトルであって、両方同時ではない。

要素の型が決められているかまたは要素が入れられないベクトルも **defclass** によって定義することができる。次の例の中で、クラス `intvec5` は 5 つの integer 要素を持つクラスとして定義されている。自動型判定と型変換は、要素がインタープリタによってアクセスされたとき実施される。正しい宣言でコンパイルされたとき、高速なアクセスコードが生成される。

```
(defclass intvec5 :super vector :element-type :integer :size 5)
(setq x (instantiate intvec5))  --> #i(0 0 0 0 0)
```

メッセージがオブジェクトに送られたとき、一致するメッセージを最初そのオブジェクトのクラスから探し、次にそのスーパークラスから探して、スーパークラスが尽きるまで探す。もし、メソッドが定義されてなかったならば、前方のリストが探される。この前方探索は、疑似多重継承によって作られる。もし、探索が失敗したときは、`:nomethod` というメソッド名が探され、メソッドは、全ての引数のリストと一緒に呼び出される。次の例の中で、メッセージ`:telephone` と`:mail` は `person` という型のオブジェクトスロット `secretary` に送られる。そして、メッセージ `go-home` はスロット `chauffeur` に送られる。

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

メソッドにおいて、2 つのローカル変数（**class** と **self**）が使用可能となる。これらの変数は変更すべきでない。もし、変更したならば、システムから供給された変数は隠され、**send-super** と **send self** は正しい動作をしない。

5.1 クラスとメソッド

defclass *classname &key :super object* [マクロ]
*:slots ({var | (var [:type type] [:forward selectors])} *)*
:metaclass metaclass

:element-type t
:size -1

クラスを生成または再定義する。異なったスーパークラスやスロットを持つクラスに再定義したとき、メソッドが新しいスロット配置を仮定するため、以前のクラスを継承する古いオブジェクトは予想できない振舞いをする。

defmethod *classname* {*(selector lambda-list . body)*}* [特殊]
classname の 1 つ以上のメソッドを定義する。それぞれの *selector* は、キーワード symbol でなければならない。

defclassmethod *classname* {*(selector lambda-list . body)*}* [マクロ]

classp *object* [関数]
object がクラスオブジェクトのとき T を返す。そのオブジェクトは、クラス **metaclass** かそのサブクラスのインスタンスである。

subclassp *class super* [関数]
class が *super* のサブクラスであることを検査する。

vector-class-p *x* [関数]
x が、**vector-class** のインスタンスであるとき、T を返す。

delete-method *class method-name* [関数]
method-name のメソッド定義を *class* から除く。

class-hierarchy *class* [関数]
class の下の継承構造を表示する。

system:list-all-classes [関数]
今まで定義されたクラスを全てリストアップする。

system:find-method *object selector* [関数]
selector に記述されたメソッドを *object* のクラスおよびそのスーパークラスの中から見つける。*object* が、*selector* に応じることができるかどうかを知るために使用される。

system:method-cache *&optional flag* [関数]
メソッドキャッシュのヒット率を調査し、ヒットとミスの 2 つの数値のリストを返す。もし *flag* が NIL のとき、メソッドキャッシュは無効になる。もし non-NIL の *flag* が与えられたとき、メソッドキャッシュは初期化されキャッシュ可能になる。

5.2 メッセージ送信

send *object selector* {*arg*}* [関数]
object に *selector* と *arg* で構成されるメッセージを送信する。*object* は、何でもよいが数値はいけない。*selector* はキーワードとして評価されなければならない。

send-message *target search selector* {*arg*}* [関数]

send-super を実行するための低レベル命令である。

send* *object selector &rest msg-list* [マクロ]
send*は、引数のリストに **send-message** を適用する。**send** と **send***の関係は、**funcall** と **apply** あるいは **list** と **list***の関係に似ている。

send-all *receivers selector &rest mesg* [関数]
全ての *receivers* に同じメッセージを送信し、結果をリストとして集める。

send-super *selector &rest msgs* [マクロ]
msgs を `self` に送信するが、メソッドが定義されているクラスのスーパークラスでのメソッドを探し始める。メソッドの外の **send-super** は、エラーとなる（すなわち、メソッド内でなければならない）。

send-super* *selector &rest msg-list* [マクロ]
send-super*は、**send-super** の apply 版である。

5.3 インスタンス管理

instantiate *class &optional size* [関数]
class から新しいオブジェクトを作る低レベル命令である。もし *class* が **vector-class** ならば、*size* がなければならない。

instance *class &rest message* [マクロ]
インスタンスが作られ、そこに *message* が送られる。

make-instance *class &rest var-val-pairs* [関数]
class のインスタンスを作成し、スロット変数を *var-val-pairs* のように設定する。例えば、`(make-instance cons :car 1 :cdr 2)` は、`(cons 1 2)` と同等である。

copy-object *object* [関数]
copy-object 関数は、参照トポロジー（再帰参照でも構わない）を保ったままコピーするために使用する。**copy-object** は、独自性の保存に触れない symbol やパッケージを除いて、*object* からアクセス可能などんなオブジェクトもコピーする。**copy-object** は、オブジェクトの中の全ての参照を 2 度妨害する。1 度が、新しいオブジェクトを作り既にコピーされたオブジェクトのオリジナルにマークを付けるとき、そしてマークを消すときにもう 1 度。この 2 段階の処理は、**copy-object** を **copy-seq** よりも遅くする。もし順番にコピーをしたいならば、**copy-seq** か **copy-tree** を使用することを薦める。

become *object class* [関数]
object のクラスを *class* に変更する。古いクラスと新しいクラス両方のスロット構造は、一致しなければならない。普通、2 要素ベクトル間のクラス変更にのみ安全に使用することができる。例えば、整数ベクトルからビットベクトルへの変更。

replace-object *dest src* [関数]
dest は、*src* のサブクラスのインスタンスでなければならない。

class *object* [関数]
object のクラスオブジェクトを返す。クラス名を得るために、クラスオブジェクトに`:name` メッセージを送る。

derivedp *object class* [関数]

derivedp は、*object* が *class* またはそのサブクラスからインスタンス化されているかどうかを判定する。**subclassp** と **derivedp** 関数は、クラス継承のなかを探索できない。したがって、一定時間内に型のチェックがいつも終了する。

slot *object class (index | slot-name)* [関数]

スロット値の名前かインデックスを返す。

setslot *object class (index | slot-name) value* [関数]

setslot は、内部処理関数でユーザーが使用できない。代りに、**setf** と **slot** の組み合せを使用する。

5.4 基本クラス

object [クラス]

:super
:slots

object は、最も基本のクラスである。それは、クラス継承の最上位に位置する。スロット変数が定義されていないため、**object** はインスタンスを作るために使用しない。

:prin1 *&optional stream &rest mesg* [メソッド]

標準の再読み込可能なオブジェクトフォーマットのなかにあるオブジェクトを表示する。そのクラス名とアドレスは、角括弧でくくられ、符号を前に置く。どのオブジェクトのサブクラスも *mesg* 文字列で説明するマクロ **send-super** を使ってもっと広範囲な情報と一緒にそれ自身を印刷するのにこの方法を使用することができる。オブジェクトは、もし#< で始まるなら、再読み込み可能である。そのクラス名・正確なアドレス・どの Lisp でも読み込可能な情報・>をあとに従えて。全てのデータオブジェクトは数値を除いて、**object** を継承している。この構文で書式の表示が得られる。(symbol や文字列でも構わない) この構文で述べることは、symbol に **setq** し忘れたデータオブジェクトを把握することができる。ただし、表示された後にガーベージコレクションが起こらない限りである。

:slots [メソッド]

変数名のリストおよびオブジェクトの全てのスロットと対になる値を返す。このリストに **assoc** を適用することにより、スロットの詳細値が得られる。しかしながら、それらを変更することはできない。

propertied-object [クラス]

:super **object**
:slots plist

property-list を持つオブジェクトを定義する。他の Common Lisp と違って、EusLisp は、たとえ、symbol でなかったとしても、property-list を持つ **propertied-object** を継承するどんなオブジェクトも許可する。

:plist *&optional plist* [メソッド]

もし *plist* が明記されるならば、このオブジェクトの `plist` スロットに設定する。そのため、以前の `plist` の値はなくなる。*plist* は、`((indicator1 . value1) (indicator2 . value2) ...)` の書式にすべきである。それぞれの `indicator` は、**eq** 関数で等価性をテストされたどのような lisp 書式も可能である。symbol が `indicator` として用いられたとき、内部パッケージを広く実行される等価性のチェックを確実にするためにキーワードの使用を推薦する。**:plist** は、主な `plist` を返す。

:get *indicator* [メソッド]

plist のなかで *indicator* と結び付く値を返す。(send x :get :y) == (cdr (assoc :y (send x :plist)))

:put *indicator value* [メソッド]
plist のなかで、*value* と *indicator* を結び付ける。

:remprop *indicator* [メソッド]
plist から *indicator* と value の組を削除する。さらに、**:get** を試すと value として NIL を返す。

:name *&optional name* [メソッド]
plist のなかの:name 特性を定義し、取り出す。この特性は、表示のために使用される。

:prin1 *&optional stream &rest mesg* [メソッド]
再読み込み可能な書式のオブジェクトを表示する。もしオブジェクトが:name 特性を持っているならば、オブジェクトのアドレスの後に特性を表示する。

metaclass [クラス]

:super	**propertied-object**
:slots	name super cix vars types forwards methods

metaclass は、複数クラスを定義する。独自のクラス変数を持つ複数クラスは、それらのスーパークラスとして **metaclass** を定義しなければならない。

:new [メソッド]
このクラスのインスタンスを生成し、全てのスロットを NIL にした後、それを返す。

:super [メソッド]
このクラスのスーパークラスオブジェクトを返す。一旦クラス定義したスーパークラスを変更することはできない。

:methods [メソッド]
このクラスで定義された全てのメソッドのリストを返す。そのリストは、メソッド名とパラメータと本体を組みにしたリストによって構成されたリストである。

:method *name* [メソッド]
name で関連づけられたメソッド定義を返す。もし見つからなければ、NIL を返す。

:method-names *subname* [メソッド]
メソッド名のなかに *subname* を含む全てのメソッド名のリストを返す。メソッドは、このクラスのなかのみ探索される。

:all-methods [メソッド]
このクラスとその全てのスーパークラスのなかで定義されているすべてのメソッドのリストを返す。言い替えると、このクラスのインスタンスは、これらのメソッドを実行することができる。

:all-method-names *subname* [メソッド]
subname と一致する全てのメソッド名のリストを返す。その探索は、このクラスから **object** まで実行される。

:slots [メソッド]
スロット名のベクトルを返す。

:name [メソッド]
このクラスの symbol 名を返す。

:cid [メソッド]

このクラスと同一であることを示すために、このクラスのインスタンスすべてに割り当てられた整数を返す。これは、システム内部のクラステーブルへのインデックスで、このクラスの下に新しいサブクラスが定義されたとき、変更される。

:subclasses [メソッド]

このクラスの直接のサブクラスのリストを返す。

:hierarchy [メソッド]

このクラスの下に定義された全てのサブクラスのリストを返す。全てのクラス継承の広範囲なリストを得るためには、**class-hierarchy** 関数を呼び出す。

find-method *object selector* [関数]

object のクラスやそのスーパークラスのなかで、*selector* と一致するメソッドを探索する。この関数は、*object* のクラスが不確かで、その *object* がエラーなしにメッセージを受け取ってくれるかどうかを知りたい時に役立つ。

6 数値演算

6.1 数値演算定数

most-positive-fixnum [定数]
#x1fffffff=536,870,911。integer の正の最大値。

most-negative-fixnum [定数]
-#x20000000= -536,870,912。integer の負の最大値

short-float-epsilon [定数]
IEEE の浮動小数点表現形式である float は、21 ビットの固定小数 (うち符号が 1 ビット) と 7 ビットの指数 (うち符号が 1 ビット) で構成されている。したがって、浮動小数点誤差 ϵ は $2^{-21} = 4.768368 \times 10^{-7}$ となる。

single-float-epsilon [定数]
short-float-epsilon と同様に 2^{-21} である。

long-float-epsilon [定数]
Euslisp には、double も long float もないため、**short-float-epsilon** と同様に 2^{-21} である。

pi [定数]
π。 実際には 3.14159203 で、 3.14159265 ではない。

2pi [定数]
$2 \times \pi$。

pi/2 [定数]
$\pi/2$。

-pi [定数]
-3.14159203。

-2pi [定数]
$-2 \times \pi$。

-pi/2 [定数]
$-\pi/2$。

6.2 比較演算関数

numberp *object* [関数]
object が integer か float の時、T を返す。その文字が数字で構成されているときも同様である。

integerp *object* [関数]
object が integer の時、T を返す。float は **round, trunc** および **ceiling** 関数で integer に変換できる。

floatp *object* [関数]
object が float の時 T を返す。integer は **float** 関数で float に変換できる。

zerop *number* [関数]
number が integer のゼロまたは float の 0.0 の時、 T を返す。

plusp *number* [関数]
number が正 (ゼロは含まない) のとき、T を返す。

minusp *number* [関数]
number が負のとき、T を返す。

oddp *integer* [関数]
integer が奇数のとき、T を返す。引数は integer のみ有効。

evenp *integer* [関数]
integer が偶数のとき、T を返す。引数は integer のみ有効。

/= *num1 num2 &rest more-numbers* [関数]
num1 と *num2*、*more-numbers* でどの 2 つの数値も等しくないとき、T を返す。それ以外は NIL を返す。*num1* と *num2*、*more-numbers* を構成する要素はすべて数値であること。

= *num1 num2 &rest more-numbers* [関数]
num1 と *num2*、*more-numbers* がすべて等しいとき、T を返す。*num1* と *num2*、*more-numbers* を構成する要素はすべて数値であること。

> *num1 num2 &rest more-numbers* [関数]
num1、*num2*、*more-numbers* の全要素がこの順に単調減少であるとき、T を返す。*num1* と *num2*、*more-numbers* を構成する要素はすべて数値であること。誤差を含めた数値比較に対しては、14 章に書かれている関数（**eps>**）を使用する。

< *num1 num2 &rest more-numbers* [関数]
num1、*num2*、*more-numbers* の全要素がこの順に単調増加であるとき、T を返す。*num1* と *num2*、*more-numbers* を構成する要素はすべて数値であること。誤差を含めた数値比較に対しては、14 章に書かれている関数（**eps<**）を使用する。

>= *num1 num2 &rest more-numbers* [関数]
num1、*num2*、*more-numbers* の全要素がこの順に単調非増加であるとき、T を返す。*num1* と *num2*、*more-numbers* を構成する要素はすべて数値であること。誤差を含めた数値比較に対しては、14 章に書かれている関数（**eps>=**）を使用する。

<= *num1 num2 &rest more-numbers* [関数]
num1、*num2*、*more-numbers* の全要素がこの順に単調非減少であるとき、T を返す。*num1* と *num2*、*more-numbers* を構成する要素はすべて数値であること。誤差を含めた数値比較に対しては、14 章に書かれている関数（**eps<=**）を使用する。

6.3 整数とビット毎の操作関数

以下の関数の引数は、すべて integer とする。

mod *dividend divisor* [関数]
dividend を *divisor* で割った余りを返す。`(mod 6 5)=1, (mod -6 5)=-1, (mod 6 -5)=1, (mod -6 -5)=-1`.

1- *integer* [関数]
integer − 1 を返す。コンパイラでは、引数を `integer` と仮定する。

1+ *integer* [関数]
integer + 1 を返す。**1+** と **1−** の引数は、`integer` でなければならない。

logand *&rest integers* [関数]
integers のビット単位ＡＮＤ。

logior *&rest integers* [関数]
integers のビット単位ＯＲ。

logxor *&rest integers* [関数]
integers のビット単位ＸＯＲ。

logeqv *&rest integers* [関数]
logeqv は `(lognot (logxor ...))` と同等である。

lognand *&rest integers* [関数]
integers のビット単位ＮＡＮＤ。

lognor *&rest integers* [関数]
integers のビット単位ＮＯＲ。

lognot *integer* [関数]
integer のビット反転。

logtest *integer1 integer2* [関数]
(logand *integer1 integer2*) がゼロでないとき T を返す。

logbitp *index integer* [関数]
integer が NIL でなければ、LSB から数えて *index* 番目のビットが 1 のとき T を返す。

ash *integer count* [関数]
数値演算左シフト。もし *count* が正のとき、*integer* を左にシフトする。もし *count* が負のとき、*integer* を |*count*| ビット右にシフトする。

ldb *target position width* [関数]
LoaD Byte. **ldb** や **dpb** の Byte 型は、EusLisp にないため、代りに 2 個の `integer` を使用する。*target* の LSB より *position* 番目の位置から MSB へ *width* ビットの範囲を抜き出す。例えば、`(ldb #x1234 4 4)` は 3 となる。

dpb *value target position width* [関数]
DePosit Byte. *target* の LSB より *position* 番目の位置へ *value* を *width* ビット置き換える。

6.4 一般数値関数

+ *&rest numbers* [関数]
numbers の和を返す。

- *num &rest more-numbers* [関数]

もし *more-numbers* が与えられたとき、*num* より引く。そうでないとき、*num* は符号反転される。

***** *&rest numbers* [関数]
numbers の積を返す。

/ *num1 num2 &rest more-numbers* [関数]
num1 を、*num2* や *more-numbers* で割り算する。全ての引数が `integer` のとき、`integer` を返し、引数に 1 つでも `float` があったときは、`float` を返す。

abs *number* [関数]
number の絶対値を返す。

round *number* [関数]
number の小数第 1 位を四捨五入し `integer` を返す。`(round 1.5)=2, (round -1.5)=2`.

floor *number* [関数]
number の小数を切捨てる。`(floor 1.5)=1, (floor -1.5)=-2`.

ceiling *number* [関数]
number の小数を切り上げる。`(ceiling 1.5)=2, (ceiling -1.5)=-1`.

truncate *number* [関数]
number が正のときは切捨て、負のときは切り上げる。`(truncate 1.5)=1, (truncate -1.5)=-1`.

float *number* [関数]
number を `float` にして返す。

max *&rest numbers* [関数]
numbers の中から、最大値をさがす。

min *&rest numbers* [関数]
numbers の中から、最小値をさがす。

random *range &optional (randstate *random-state*)* [関数]
0 あるいは 0.0 から *range* までの乱数を返す。もし *range* が `integer` のとき、`integer` に変換して返す。そうでないとき、`float` を返す。オプションの *randstate* は、決まった乱数列で表される。*randstate* に特別なデータの型はなく、2 つの要素からなる 整数ベクトルで表される。

incf *variable &optional (increment 1)* [マクロ]
variable は一般の変数である。*variable* は、*increment* だけ増加され、*variable* に戻される。

decf *variable &optional decrement* [マクロ]
variable は一般の変数である。*variable* は、*decrement* だけ減少され、*variable* に戻される。

reduce *func seq* [関数]
2 変数操作の *func* 関数を用いて、*seq* の中の全ての要素を結合させる。例えば、`(reduce #'expt '(2 3 4)) = (expt (expt 2 3) 4)=4096`.

rad2deg *radian* [関数]
ラジアン値を 度数表現に変換する。#R は同じものである。EusLisp の中での角度の表記はラジアンであり、EusLisp 内の全ての関数が要求する角度引数は、ラジアン表現である。

deg2rad *degree* [関数]
角度値をラジアン表現に変換する。また #D でも実行できる。

6.5 基本関数

sin *theta* [関数]
theta はラジアンで表される `float` 値。sin(*theta*) を返す。

cos *theta* [関数]
theta はラジアンで表される `float` 値。cos(*theta*) を返す。

tan *theta* [関数]
theta はラジアンで表される `float` 値。tan(*theta*) を返す。

sinh *x* [関数]
hyperbolic sine、$\frac{e^x-e^{-x}}{2}$ で表される。

cosh *x* [関数]
hyperbolic cosine、$\frac{e^x+e^{-x}}{2}$ で表される。

tanh *x* [関数]
hyperbolic tangent、$\frac{e^x+e^{-x}}{e^x-e^{-x}}$ で表される。

asin *number* [関数]
number の arc sine を返す。

acos *number* [関数]
number の arc cosine を返す。

atan *y &optional x* [関数]
atan が 1 つの引数だけのとき、arctangent を計算する。2 つの引数のとき、`atan`(y/x) を計算する。

asinh *x* [関数]
hyperbolic arc sine.

acosh *x* [関数]
hyperbolic arc cosine.

atanh *x* [関数]
hyperbolic arc tangent.

sqrt *number* [関数]
number の平方根を返す。

log *number* [関数]
number の自然対数を返す。

exp *x* [関数]
e^x を返す。

expt *a x* [関数]
a の *x* 乗を返す。

7 symbol とパッケージ

7.1 symbol

symbol は、いつでも唯一であることが保証され、パッケージに収容される。1 つのパッケージのなかに、その他の symbol としておなじ print-name を持っていることはあるが、symbol 自体がコピーされることはない。symbol がリーダに読まれるとき、symbol オブジェクトは、自動的に生成され、1 つのパッケージに収容される。そのパッケージは、パッケージ名にコロン（:）を付け加えた接頭語によって記すことができる。そのようなパッケージ接頭語がないならば、symbol は **lisp:*package*** の値にある主なパッケージに収容される。全ての symbol は、1 つのホームパッケージを持っている。もし symbol がそのようなホームパッケージを持っていないならば、収容されていない symbol と言われる。収容されていない symbol は、**gensym** や **make-symbol** 関数によって作ることができ、表示のときは、"`#:`"接頭語がつけられる。これらの symbol は収容されていないので、そのような 2 つの symbol が同じ print-name を持っていた場合、等しいことが保証されない。

ふつう、lisp リーダが symbol に出会ったとき、リーダは自動的に symbol の print-name 文字列を大文字に変換する。したがって、例えば (`symbol-name 'car`) と入力したとすると、EusLisp は、`"car"`の代りに`"CAR"`と答える。(`make-symbol "car"`) は、`car` や `CAR` の代りに |car| を返す。もし、小文字から構成される symbol をリーダに作らせたい場合、\ や |...| のようなエスケープを用いること。

symbolp *object* [関数]

object がクラス symbol かそのサブクラスのインスタンスであったならば、T を返す。

symbol-value *symbol* [関数]

symbol の特殊値を得る。ローカル変数値は、この関数では取り出すことはできない。

symbol-function *symbol* [関数]

symbol のグローバル関数定義を得る。ローカル関数は、この関数で得られない。

symbol-package *sym* [関数]

sym が収容されているパッケージを返す。

symbol-name *sym* [関数]

sym の print-name を返す。**symbol-name** は、**string** がコピーするのに反して、pname 文字列をコピーしない。したがって、もし **symbol-name** で返された文字列を変えるならば symbol は普通の内部手続きを通してアクセス不可能となる。

symbol-plist *sym* [関数]

sym の property-list(plist) を返す。EusLisp の plist は、関連リスト（association-list）と同じ様な書式を与える。それは、属性名とその値の組を点でつないだ構成である。これは、Common Lisp の定義が要求する plist（属性名と値の線形リスト）と非互換である。EusLisp において、plist は symbol に独特なものではない。**propertied-object** を継承するクラスから派生したどんなオブジェクトも、property-list を持つことができる。**propertied-object** の中のこれらの plist を設定したり、取りだしたりするために、**propertied-object-plist** マクロは、**symbol-plist** の代りに使用されるべきである。しかしながら、**get** と **putprop** はどのオブジェクトにも働く。

boundp *symbol* [関数]

symbol がグローバルな値を持っているかどうかを検査する。
注意:ローカル変数やオブジェクト変数として使用される symbol はいつも値を持っているため、*boundp* はこれらのローカル変数の格納状態を検査することができない。

fboundp *symbol* [関数]

symbol がグローバルな関数定義を持っているかどうかを検査する。

makunbound *symbol* [関数]
symbol は、（特殊値を持たないように）強制的に unbound される。ローカル変数は、いつも値が割り当てられ、**makunbound** できない。

get *sym attribute* [関数]
sym の plist の中で *attribute* に関連する値を取り出す。`(cdr (assoc attribute (symbol-plist sym)))` と等価である。

putprop *sym val attribute* [関数]
putprop は、**setf** と **get** の組み合せで置き換えるべきである。

remprop *sym attr* [関数]
属性値 (*attr*) と *sym* の組を property-list から削除する。

setq {*var value*}* [特殊]
value を *var* に割り当てる。*var* は、symbol か点で継った組である。*var* は、ローカル変数・オブジェクト変数・特殊変数の順番にその名前の中から探される。ただし、明確に special と宣言されてないものに限る。

set *sym val* [関数]
val を *sym* の特殊値として割り当てる。**set** は、ローカル変数やオブジェクト変数に値を割当てることはできない。

defun *symbol [documentation] lambda-list . body* [特殊]
symbol にグローバル関数を定義する。ローカル関数を定義するためには、**flet** か **labels** を使用すること。もし、*documentation* が与えられない場合、lambda-list に書かれているデフォルトの documentation 文字列が入力される。

defmacro *symbol [documentation] lambda-list . body* [特殊]
グローバルマクロを定義する。EusLisp は、ローカルスコープマクロ定義の機能を持っていない。

defvar *var &optional (init nil) doc* [マクロ]
もし *var* が特殊値を持っていれば、**defvar** は何もしない。もし *var* が unbound ならば、`special` として宣言し、*init* をその値として設定する。

defparameter *var init &optional doc* [マクロ]
defparameter は、*var* を `special` として宣言し、*var* が既に値を持っていたとしても、*init* をその値として設定する。

defconstant *sym val &optional doc* [マクロ]
defconstant は、**val** を **sym** の特殊値として設定する。**defvar, defparameter** や **setq** と違い、その値は **defconstant** でのみ設定され、これらの書式で変更することができない。もし定数 symbol の値が、変更されようとしたならば、エラーが返される。しかし、**defconstant** は以前の定数値に上書きでき、上書きした場合は注意メッセージが出力される。

keywordp *obj* [関数]
もし *obj* が symbol で、そのホームパッケージが **KEYWORD** のとき T を返す。

constantp *symbol* [関数]
もし symbol が **defconstant** マクロで定数に宣言されているとき T を返す。

documentation *sym &optional type* [関数]

sym のために提示文字列 (documentation string) を取り出す。

gensym *&optional x* [関数]

g001 のような前置文字列と付属数字を組み合わせた新しい収容されていない symbol を作る。収容されてない symbol は、symbol に関連するパッケージがないため、パッケージ前置詞の部分に#:を示す。#:が前につく symbol は、読めない symbol で、リーダではこれらの symbol への参照を作成することができない。*x* は、整数か文字列が可能で、接頭 (prefix) インデックスか接尾 (suffix) 値として使用される。

gentemp *&optional (prefix "T") (pkg *package*)* [関数]

pkg に収容される新しい symbol を作る。ほとんどのアプリケーションにおいて、**gensym** が **gentemp** よりも好まれる。なぜなら、収容されない symbol の方が高速に作ることができ、ガーベージコレクトも可能であるため。

7.2 パッケージ

パッケージは、symbol をグループ化するための区分された名前の付いた空間を与える。複数のプログラマが要求されるような膨大なソフトウェアシステムを開発しようとするとき、symbol（関数および変数名）が重複する問題を減少させるために Common Lisp でパッケージシステムが生まれた。それぞれのパッケージは、内部 symbol と外部 symbol を持つ。パッケージの中で symbol が作成されたとき、いつでも内部 symbol となる。**export** を使用することにより外部 symbol にすることができる。異なったパッケージの外部 symbol は、symbol の前にパッケージ名とコロン (:) をつけることにより参照することができる。例えば、x:*display*となる。他のパッケージの内部 symbol を参照する場合には、sys::free-threads のようにダブルコロン (::) を使用する。前にパッケージ名をつけることを省略するためには、**import** を用いる。その上、**use-package** を使用すれば、他のパッケージの全ての外部 symbol を import することができる。symbol を export あるいは import するとき、あらゆるパッケージ内の全ての symbol が独自の print-name を持つ必要があるため、symbol 名の重複を発見することができる。shadow は、パッケージから symbol を仮想的に削除することにより、存在する symbol と同じ名前の symbol を作成することができる。

Euslisp は次の 8 つのパッケージを定義する。

lisp: 全ての lisp 関数、マクロ、定数、など

keyword: キーワード symbol

unix: UNIX システムコールとライブラリ関数

system: システム管理または危険な関数; nicknames=sys,si

compiler: EusLisp コンパイラ; nicknames=comp

user: ユーザー領域

geometry: 幾何学クラスとその関数

xwindow: X-window インターフェース; nickname=x

これらのパッケージとユーザー定義パッケージは、システムの package-list に繋げられている。それは、**list-all-packages** よって得ることができる。それぞれのパッケージは、内部および外部 symbol を探索・位置付けするために 2 つのハッシュテーブルを管理する。また、パッケージは、その名前 (string または symbol) と nick name のリストとそのパッケージが使う他のパッケージリストを記憶している。***package***は、読み込み・印

刷のための主なパッケージを持つグローバル変数である。もし***package***が `user:`でないならば、top-level プロンプトは、現在のパッケージを示すために `mypkg:eus$`のように変更される。

lisp-package [定数]
Lisp パッケージ。

user-package [定数]
ユーザーパッケージ。

unix-package [定数]
UNIX パッケージ。

system-package [定数]
システムパッケージ。

keyword-package [定数]
キーワードパッケージ。

find-symbol *string &optional (package *package*)* [関数]
package のなかで print-name として *string* をもつ symbol を見つける。もし見つかったとき、その symbol が返され、そうでないとき NIL が返される。

make-symbol *string* [関数]
string で示される名前の新しい収容されていない symbol を作る。

intern *string &optional (package *package*) (klass symbol)* [関数]
string と同じ print-name の symbol を見つけようとする。もし探索成功のとき、その symbol が返される。もし失敗したとき、*string* という print-name を持つ symbol が新しく作られ、*package* の中におかれる。

list-all-packages [関数]
今までに作られた全てのパッケージのリストを返す。

find-package *name* [関数]
パッケージの名前または nickname が *name* と同じものを探す。

make-package *name &key :nicknames (:use '(lisp))* [関数]
name で示される名前の新しいパッケージを作る。*name* は、string あるいは symbol でよい。もしパッケージが既に存在している場合、エラーが報告される。

in-package *pkg &key :nicknames (:uses '(lisp))* [関数]
現在のパッケージ (***pacakge***の値) を *pkg* に変える。

package-name *pkg* [関数]
pkg パッケージの名前を文字列として返す。

package-nicknames *pkg* [関数]
pkg の nickname のリストを返す。

rename-package *pkg new-name &optional new-nicknames* [関数]
pkg の名前を *new-name* に変更し、その nickname を *new-nicknames* に変更する。それらは、symbol か string または symbol か string のリストのどれでも可能である。

package-use-list *pkg* [関数]
pkg で使用されるパッケージリストを返す。

packagep *pkg* [関数]
もし *pkg* がパッケージのとき T を返す。

use-package *pkg &optional (curpkg *package*)* [関数]
pkg を *curpkg* の use-list に付け加える。一旦追加すると、*pkg* のなかの symbol は、パッケージの前置詞なしで *curpkg* を見ることが可能になる。

unuse-package *pkg &optional (curpkg *package*)* [関数]
curpkg の use-list から *pkg* を削除する。

shadow *sym &optional(pkg *package*)* [関数]
存在する *sym* を隠すことによって、*pkg* 内に内部 symbol を作る。

export *sym &optional (pkg *package*)* [関数]
sym は、symbol か symbol のリストである。**export** は、*sym* を他のパッケージから外部 symbol としてアクセス可能とする。実際に、*sym* は、*pkg* のなかの外部 symbol として記録される。もし symbol が **export** されると、パッケージマークとして single colon ":"を使ってアクセス可能となる。これに対して、**export** されていない symbol は double colon "::" で得られる。そのうえ、**export** された symbol は、**use-package** を使用したり、パッケージに **import** されたとき、コロンの必要がない。symbol が export されたかどうかは、それぞれの symbol にでなくそれが収容されているパッケージに属性付けられる。それで、symbol は 1 つのパッケージの内部にあり、その他の外部となる。**export** は、*pkg* が使用している他のパッケージの中の symbol 名と *sym* が重複していないかどうかを検査する。もし *sym* と同じ print name をもつ symbol があったならば、"symbol conflict"とエラーを報告する。

unexport *sym &optional pkg* [関数]
もし *sym* が *pkg* の外部 symbol であったならば、unexport され、内部 symbol となる。

import *sym &optional (pkg *package*)* [関数]
sym は、symbol または symbol のリストである。**import** は、他のパッケージで定義された symbol を *pkg* からパッケージの前置詞なしで内部 symbol として見えるようにする。もし *sym* と同じ print-name を持った symbol が既にあったとき、"name conflict"とエラーを報告する。

do-symbols *(var pkg) &rest forms* [マクロ]
pkg において、(内部あるいは外部) symbol に対して繰り返しをする。そのときの *forms* の評価が返される。

do-external-symbols *(var pkg) &rest forms* [マクロ]
pkg において、外部 symbol に対して繰り返しをする。そのときの *forms* の評価が返される。

do-all-symbols *(var [result]) &rest forms* [マクロ]
全てのパッケージにおいて、symbol に対して繰り返しをする。そのときの *forms* の評価が返される。もし、1 つ以上のパッケージの中にその symbol が現れたならば、*forms* は、1 度以上評価される。

8 列、行列とテーブル

8.1 一般列

ベクトル (1 次元行列) とリストは、一般の列である。文字列 (string) は、文字 (character) のベクトルなので、列である。

map, concatenate や **coerce** における結果の型を明記するためには、クラスオブジェクトが symbol にバインドされていないので、引用符なしで `cons`, `string`, `integer-vector`, `float-vector` などのクラス名 symbol を使う。

elt *sequence pos* [関数]

elt は、*sequence* の中の *pos* 番目の位置の値を得たり、(**setf** とともに) 置いたりする最も一般的な関数である。*sequence* は、リストまたは任意のオブジェクト、`bit`, `char`, `integer`, `float` のベクトルである。**elt** は、多次元の行列に適用できない。

length *sequence* [関数]

sequence の長さを返す。ベクトルにおいて、**length** は一定の時間で終了する。しかし、リスト型においては、長さに比例した時間がかかる。**length** が、もし環状リストに適用されたとき、決して終了しない。代わりに **list-length** を使用すること。もし、*sequence* が fill-pointer を持つ行列ならば、**length** は行列全体のサイズを返すのではなく fill-pointer を返す。このような行列のサイズを知りたい場合には、**array-total-size** を使用すること。

subseq *sequence start [end]* [関数]

sequence の *start* 番目から (*end*−1) 番目までをそっくりコピーした列を作る。*end* は、デフォルト値として *sequence* の長さをとる。

copy-seq *sequence* [関数]

sequence のコピーした列を作る。このコピーでは、*sequence* のトップレベルの参照のみがコピーされる。入れこリストのコピーには **copy-tree** を使い、再帰参照を持つような列のコピーには **copy-object** を使うこと。

reverse *sequence* [関数]

sequence の順番を逆にし、*sequence* と同じ型の新しい列を返す。

nreverse *sequence* [関数]

nreverse は、**reverse** の破壊 (destructive) バージョンである。**reverse** はメモリを確保するが、**nreverse** はしない。

concatenate *result-type* {*sequence*}* [関数]

全ての *sequence* を連結させる。それぞれの *sequence* は、なにかの列型である。**append** と違って、最後の一つまで含めた全ての列がコピーされる。*result-type* は、`cons`,`string`,`vector`,`float-vector` などのクラスである。

coerce *sequence result-type* [関数]

sequence の型を変更する。例えば、`(coerce '(a b c) vector) = #(a b c)` や `(coerce "ABC" cons) = (a b c)` である。*result-type* 型の新しい列が作られ、*sequence* のそれぞれの要素はその列にコピーされる。*result-type* は、`vector`, `integer-vector`, `float-vector`, `bit-vector`, `string`, `cons` またはそれらの 1 つを継承したユーザー定義クラスのうちの 1 つである。**coerce** は、*sequence* の型が *result-type* と同一である場合、コピーをする。

map *result-type function seq &rest more-seqs* [関数]
function は、*seq* と *more-seqs* のそれぞれの N 番目 ($N = 0, 1, \cdots$) の要素からなるリストに対して適用され、その結果は *result-type* の型の列に蓄積される。

fill *sequence item &key (:start 0) (:end (length sequence))* [関数]
sequence の *start* 番目から (*end*−1) 番目まで、*item* で満たす。

replace *dest source &key :start1 :end1 :start2 :end2* [関数]
dest 列の中の *start1* から *end1* までの要素が、*source* 列の中の *start2* から *end2* までの要素に置き換えられる。*start1* と *start2* のデフォルト値はゼロで、*end1* と *end2* のデフォルト値はそれぞれの列の長さである。もし片方の列がもう一方よりも長いならば、end は短い列の長さに一致するように縮められる。

sort *sequence compare &optional key* [関数]
sequence は、Unix の quick-sort サブルーチンを使って破壊的に (destructively) にソートされる。*key* は、キーワードパラメータでなく、比較用のパラメータである。同じ要素を持った列のソートをするときは十分気をつけること。例えば、`(sort '(1 1) #'>)` は失敗する。なぜなら、1 と 1 の比較はどちらからでも失敗するからである。この問題を避けるために、比較として`#'>=` か`#'<=` のような関数を用いる。

merge *result-type seq1 seq2 pred &key (:key #'identity)* [関数]
２つの列 *seq1* と *seq2* は、*result-type* 型の１つの列に合併され、それらの要素は *pred* に記述された比較を満足する。

merge-list *list1 list2 pred key* [関数]
２つのリストを合併させる。**merge** と違って、一般列は引数として許可されないが、**merge-list** は **merge** より実行が速い

次の関数は、1 つの基本関数と-if や-if-not を後に付けた変形関数から成る。基本形は、少なくとも item と sequence の引数を持つ。sequence の中のそれぞれの要素と item を比較し、何かの処理をする。例えば、インデックスを探したり、現れる回数を数えたり、item を削除したりなど。変形関数は、predicate と sequence の引数を持つ。sequence のそれぞれの要素に predicate を適用し、もし predicate が non-NIL を返したとき (-if version)、または NIL を返したとき (-if-not version) に何かをする。

position *item seq &key :start :end :test :test-not :key (:count 1)* [関数]
seq の中から *item* と同一な要素を探し、その要素の中で *:count* 番目に現れた要素のインデックスを返す。その探索は、*:start* 番目の要素から始め、それ以前の要素は無視する。デフォルトの探索は、`eql` で実行されるが、*test* か *test-not* パラメータで変更できる。

position-if *predicate seq &key :start :end :key* [関数]

position-if-not *predicate seq &key :start :end :key* [関数]

find *item seq &key :start :end :test :test-not :key (:count 1)* [関数]
seq の中の *start* 番目の要素から *:end* 番目の要素までの間で要素を探し、その探された要素の内、*:count* 番目の要素を返す。その要素は、*:test* か *:test-not* に`#'eql` 以外のものが記述されていないなら、*item* と同じものである。

find-if *predicate seq &key :start :end :key (:count 1)* [関数]
seq の要素の中で *predicate* が non-NIL を返す要素の内、*:count* 番目の要素を返す。

find-if-not *predicate seq &key :start :end :key* [関数]

count *item seq &key :start :end :test :test-not :key* [関数]
seq の中の *:start* 番目から *:end* 番目までの要素に *item* が何回現れるか数える。

count-if *predicate seq &key :start :end :key* [関数]
predicate が non-NIL を返す *seq* 内の要素数を数える。

count-if-not *predicate seq &key :start :end :key* [関数]

remove *item seq &key :start :end :test :test-not :key :count* [関数]
seq の中の *:start* 番目から *:end* 番目までの要素のなかで、*item* と同一の要素を探し、*:count*（デフォルト値は∞）番目までの要素を削除した新しい列を作る。もし、*item* が一回のみ現れることが確定しているなら、無意味な探索を避けるために、*:count=1* を指定すること。

remove-if *predicate seq &key :start :end :key :count* [関数]

remove-if-not *predicate seq &key :start :end :key :count* [関数]

remove-duplicates *seq &key :start :end :key :test :test-not :count* [関数]
seq の中から複数存在する item を探し、その中の 1 つだけを残した新しい列を作る。

delete *item seq &key :start :end :test :test-not :key :count* [関数]
delete は、*seq* 自体を修正し、新しい列を作らないことを除いては、**remove** 同じである。もし、*item* が一回のみ現れることが確定しているなら、無意味な探索を避けるために、*:count=1* を指定すること。

delete-if *predicate seq &key :start :end :key :count* [関数]

delete-if-not *predicate seq &key :start :end :key :count* [関数]
remove や **delete** の *:count* デフォルト値は、1,000,000 である。もし列が長く、削除したい要素が一回しか現れないときは、*:count* を 1 と記述すべきである。

substitute *newitem olditem seq &key :start :end :test :test-not :key :count* [関数]
seq の中で *:count* 番目に現れた *olditem* を *newitem* に置き換えた新しい列を返す。デフォルトでは、全ての *olditem* を置き換える。

substitute-if *newitem predicate seq &key :start :end :key :count* [関数]

substitute-if-not *newitem predicate seq &key :start :end :key :count* [関数]

nsubstitute *newitem olditem seq &key :start :end :test :test-not :key :count* [関数]
seq の中で *count* 番目に現れた *olditem* を *newitem* に置き換え、元の列 *seq* に返す。デフォルトでは、全ての *olditem* を置き換える。

nsubstitute-if *newitem predicate seq &key :start :end :key :count* [関数]

nsubstitute-if-not *newitem predicate seq &key :start :end :key :count* [関数]

8.2 リスト

listp *object* [関数]
オブジェクトが cons のインスタンスかもしくは NIL ならば、T を返す。

consp *object* [関数]
`(not (atom object))` と同一である。`(consp '())` は NIL である。

car *list* [関数]
list の最初の要素を返す。NIL の **car** は NIL である。atom の **car** はエラーとなる。`(car '(1 2 3)) = 1`

cdr *list* [関数]
list の最初の要素を削除した残りのリストを返す。NIL の **cdr** は NIL である。atom の **cdr** はエラーとなる。`(cdr '(1 2 3)) = (2 3)`

cadr *list* [関数]
`(cadr list) = (car (cdr list))`

cddr *list* [関数]
`(cddr list) = (cdr (cdr list))`

cdar *list* [関数]
`(cdar list) = (cdr (car list))`

caar *list* [関数]
`(caar list) = (car (car list))`

caddr *list* [関数]
`(caddr list) = (car (cdr (cdr list)))`

caadr *list* [関数]
`(caadr list) = (car (car (cdr list)))`

cadar *list* [関数]
`(cadar list) = (car (cdr (car list)))`

caaar *list* [関数]
`(caaar list) = (car (car (car list)))`

cdadr *list* [関数]
`(cdadr list) = (cdr (car (cdr list)))`

cdaar *list* [関数]
`(cdaar list) = (cdr (car (car list)))`

cdddr *list* [関数]
`(cdddr list) = (cdr (cdr (cdr list)))`

cddar *list* [関数]
`(cddar list) = (cdr (cdr (car list)))`

first *list* [関数]

list の最初の要素を取り出す。**second, third, fourth, fifth, sixth, seventh, eighth** もまた定義されている。`(first list) = (car list)`

nth *count list* [関数]
list 内の *count* 番目の要素を返す。`(nth 1 list)` は、`(second list)` あるいは `(elt list 1)` と等価である。

nthcdr *count list* [関数]
list に **cdr** を *count* 回適用した後のリストを返す。

last *list* [関数]
list の最後の要素でなく、最後の cons を返す。

butlast *list &optional (n 1)* [関数]
list の最後から *n* 個の要素を削除したリストを返す。

cons *car cdr* [関数]
`car` が *car* で `cdr` が *cdr* であるような新しい cons を作る。

list {*element*}* [関数]
element を要素とするリストを作る。

list* {*element*}* [関数]
element を要素とするリストを作る。しかし、最後の要素は **cons** されるため、atom であってはならない。例えば、`(list* 1 2 3 '(4 5)) = (1 2 3 4 5)` である。

list-length *list* [関数]
list の長さを返す。*list* は、環状リストでも良い。

make-list *size &key (:initial-element nil)* [関数]
size 長さで要素が全て *:initial-element* のリストを作る。

rplaca *cons a* [関数]
cons の **car** を *a* に置き換える。**setf** と **car** の使用を推薦する。`(rplaca cons a) = (setf (car cons) a)`

rplacd *cons d* [関数]
cons の **cdr** を *d* に置き換える。**setf** と **cdr** の使用を推薦する。`(rplacd cons d) = (setf (cdr cons) d)`

memq *item list* [関数]
member に似ている。しかしテストはいつも `eq` で行う。

member *item list &key :key :test :test-not* [関数]
list の中から条件にあった要素を探す。*list* の中から *item* を探索し、*:test* の条件にあったものがなければ NIL を返す。見つかったならば、それ以降をリストとして返す。この探索は、最上位のリストに対して行なわれる。*:test* のデフォルトは`#'eq` である。`(member 'a '(g (a y) b a d g e a y))=(a d g e a y)`

assq *item alist* [関数]

assoc *item alist &key :key :test :test-not* [関数]

alist の要素の **car** が *:test* の条件にあった最初のものを返す。合わなければ、NIL を返す。*:test* のデフォルトは`#'eq` である。`(assoc '2 '((1 d t y)(2 g h t)(3 e x g))=(2 g h t)`

rassoc *item alist* [関数]
cdr が *item* に等しい *alist* のなかの最初の組を返す。

pairlis *l1 l2 &optional alist* [関数]
l1 と *l2* の中の一致する要素を対にしたリストを作る。もし *alist* が与えられたとき、*l1* と *l2* から作られた対リストの最後に連結させる。

acons *key val alist* [関数]
alist に *key val* の組を付け加える。`(cons (cons key val) alist)` と同等である。

append {*list*}* [関数]
新しいリストを形成するために *list* を連結させる。最後のリストを除いて、*list* のなかの全ての要素はコピーされる。

nconc {*list*}* [関数]
それぞれの *list* の最後の **cdr** を置き換える事によって、*list* を破壊的に (destructively) 連結する。

subst *new old tree* [関数]
tree の中のすべての *old* を *new* に置き換える。

flatten *complex-list* [関数]
atom やいろんな深さのリストを含んだ *complex-list* を、1つの線形リストに変換する。そのリストは、*complex-list* の中のトップレベルに全ての要素を置く。`(flatten '(a (b (c d) e))) = (a b c d e)`

push *item place* [マクロ]
place にバインドされたスタック (リスト) に *item* を置く。

pop *stack* [マクロ]
stack から最初の要素を削除し、それを返す。もし *stack* が空 (NIL) ならば、NIL を返す。

pushnew *item place &key :test :test-not :key* [マクロ]
もし *item* が *place* のメンバーでないなら、*place* リストに *item* を置く。*:test*, *:test-not* と *:key* 引数は、**member** 関数に送られる。

adjoin *item list* [関数]
もし *item* が *list* に含まれてないなら、*list* の最初に付け加える。

union *list1 list2 &key (:test #'eq) (:test-not) (:key #'identity)* [関数]
2つのリストの和集合を返す。

subsetp *list1 list2 &key (:test #'eq) (:test-not) (:key #'identity)* [関数]
list1 が *list2* の部分集合であること、すなわち、*list1* のそれぞれの要素が *list2* のメンバーであることをテストする。

intersection *list1 list2 &key (:test #'eq) (:test-not) (:key #'identity)* [関数]
2つのリスト *list1* と *list2* の積集合を返す。

set-difference *list1 list2 &key (:test #'eq) (:test-not) (:key #'identity)* [関数]
list1 にのみ含まれていて *list2* に含まれていない要素からなるリストを返す。

set-exclusive-or *list1 list2 &key (:test #'eq) (:test-not) (:key #'identity)* [関数]

list1 および *list2* にのみ現れる要素からなるリストを返す。

list-insert *item pos list* [関数]

list の *pos* 番目の要素として *item* を挿入する（元のリストを変化させる）。もし *pos* が *list* の長さより大きいなら、*item* は最後に **nconc** される。`(list-insert 'x 2 '(a b c d)) = (a b x c d)`

copy-tree *tree* [関数]

入れこリストである *tree* のコピーを返す。しかし、環状参照はできない。環状リストは、**copy-object** でコピーできる。実際に、**copy-tree** は `(subst t t tree)` と簡単に記述される。

mapc *func arg-list &rest more-arg-lists* [関数]

arg-list や *more-arg-lists* それぞれの N 番目 ($N = 0, 1, \cdots$) の要素からなるリストに *func* を適用する。適用結果は無視され、*arg-list* が返される。

mapcar *func &rest arg-list* [関数]

arg-list のそれぞれの要素に *func* を **map** し、その全ての結果のリストを作る。**mapcar** を使う前に、**dolist** を試すこと。

mapcan *func arg-list &rest more-arg-lists* [関数]

arg-list のそれぞれの要素に *func* を **map** し、**nconc** を用いてその全ての結果のリストを作る。**nconc** は NIL に対して何もしないため、**mapcan** は、*arg-list* の要素にフィルタをかける（選択する）のに合っている。

8.3 ベクトルと行列

7次元以内の行列が許可されている。1次元の行列は、ベクトルと呼ばれる。ベクトルとリストは、列としてグループ化される。もし、行列の要素がいろんな型であったとき、その行列は一般化されていると言う。もし、行列が fill-pointer を持ってなく、他の行列で置き換えられなく、拡張不可能であるなら、その行列は簡略化されたと言う。

全ての行列要素は、**aref** により取り出すことができ、**aref** を用いて **setf** により設定することができる。しかし、一次元ベクトルのために簡単で高速なアクセス関数がある。**svref** は一次元一般ベクトル、**char** と **schar** は一次元文字ベクトル（文字列）、**bit** と **sbit** は一次元ビットベクトルのための高速関数である。これらの関数はコンパイルされたとき、アクセスは in-line を拡張し、型チェックと境界チェックなしに実行される。

ベクトルもまたオブジェクトであるため、別のベクトルクラスを派生させることができる。5 種類の内部ベクトルクラスがある。`vector`, `string`, `float-vector`, `integer-vector` と `bit-vector` である。ベクトルの作成を容易にするために、make-array 関数がある。要素の型は、`:integer`, `:bit`, `:character`, `:float`, `:foreign` かあるいはユーザーが定義したベクトルクラスの内の一つでなければならない。*:initial-element* と *:initial-contents* のキーワード引数は、行列の初期値を設定するために役に立つ。

array-rank-limit [定数]

7。行列の最大次元を示す。

array-dimension-limit [定数]

#x1fffffff。各次元の最大要素数を示す。論理的な数であって、システムの物理メモリあるいは仮想メモリの大きさによって制限される。

vectorp *object* [関数]

行列は1次元であってもベクトルではない。*object* が `vector`, `integer-vector`, `float-vector`, `string`, `bit-vector` あるいはユーザーで定義したベクトルなら T を返す。

vector *&rest elements* [関数]

elements からなる一次元ベクトルを作る。

make-array *dims &key (:element-type vector) (:initial-contents nil) (:initial-element nil) (:fill-pointer nil) (:displaced-to nil) (:displaced-index-offset 0) (:adjustable nil)* [関数]

ベクトルか行列を作る。*dims* は、整数かリストである。もし *dims* が整数なら、一次元ベクトルが作られる。

svref *vector pos* [関数]

vector の *pos* 番目の要素を返す。*vector* は、一次元一般ベクトルでなければならない。

aref *vector &rest (indices)* [関数]

vector の *indices* によってインデックスされる要素を返す。*indices* は、整数であり、*vector* の次元の数だけ指定する。**aref** は、非効率的である。なぜなら、*vector* の型に従うように変更する必要があるためである。コンパイルコードの速度を改善するため、できるだけ型の宣言を与えるべきである。

vector-push *val array* [関数]

array の fill-pointer 番目のスロットに *val* を保管する。*array* は、fill-pointer を持っていなければならない。*val* が保管された後、fill-pointer は、次の位置にポイントを 1 つ進められる。もし、行列の境界よりも大きくなったとき、エラーが報告される。

vector-push-extend *val array* [関数]
array の fill-pointer が最後に到達したとき、自動的に *array* のサイズが拡張されることを除いては、**vector-push** と同じである。

arrayp *obj* [関数]
もし *obj* が行列またはベクトルのインスタンスであるなら T を返す。

array-total-size *array* [関数]

array の要素数の合計を返す。

fill-pointer *array* [関数]
array の fill-pointer を返す。file-pointer を持っていなければ NIL を返す。

array-rank *array* [関数]
array の次元数を返す。

array-dimensions *array* [関数]
array の各次元の要素数をリストで返す。

array-dimension *array axis* [関数]
array-dimension は、*array* の *axis* 番目の次元を返す。*axis* はゼロから始まる。

bit *bitvec index* [関数]
bitvec の *index* 番目の要素を返す。ビットベクトルの要素を変更するには、**setf** と **bit** を使用すること。

bit-and *bits1 bits2 &optional result* [関数]

bit-ior *bits1 bits2 &optional result* [関数]

bit-xor *bits1 bits2 &optional result* [関数]

bit-eqv *bits1 bits2 &optional result* [関数]

bit-nand *bits1 bits2 &optional result* [関数]

bit-nor *bits1 bits2 &optional result* [関数]

bit-not *bits1 &optional result* [関数]
同じ長さの *bits1* と *bits2* というビットベクトルにおいて、それらの and, inclusive-or, exclusive-or, 等価, not-and, not-or と not がそれぞれ返される。

8.4 文字と文字列

EusLisp には、文字型がない。文字は、integer によって表現されている。ファイル名を現わす文字列を扱うためには、9.6 節に書かれている **pathname** を使うこと。

digit-char-p *ch* [関数]
もし *ch* が#\0〜#\9 なら T を返す。

alpha-char-p *ch* [関数]
もし *ch* が#\A〜#\Z または#\a〜#\z なら、T を返す。

upper-case-p *ch* [関数]
もし *ch* が#\A〜#\Z なら、T を返す。

lower-case-p *ch* [関数]
もし *ch* が#\a〜#\z なら、T を返す。

alphanumeric-p *ch* [関数]
もし *ch* が#\0〜#\9、#\A〜#\Z または#\a〜#\z なら、T を返す。

char-upcase *ch* [関数]
ch を大文字に変換する。

char-downcase *ch* [関数]
ch を小文字に変換する。

char *string index* [関数]
string の *index* 番目の文字を返す。

schar *string index* [関数]
string から文字を抜き出す。*string* の型が明確に解っていて、型チェックを要しないときのみ、**schar** を使うこと。

stringp *string* [関数]
string がバイト (256 より小さい正の整数) のベクトルなら、T を返す。

string-upcase *str &key :start :end* [関数]
str を大文字の文字列に変換して、新しい文字列を返す。

string-downcase *str &key :start :end* [関数]
str を小文字の文字列に変換して、新しい文字列を返す。

nstring-upcase *str* [関数]
str を大文字の文字列に変換し、元に置き換える。

nstring-downcase *str &key :start :end* [関数]
str を小文字の文字列に変換し、元に置き換える。

string= *str1 str2 &key :start1 :end1 :start2 :end2* [関数]
もし *str1* が *str2* と等しいとき、T を返す。**string=**は、大文字・小文字を判別する。

string-equal *str1 str2 &key :start1 :end1 :start2 :end2* [関数]
str1 と *str2* の等価性をテストする。**string-equal** は、大文字・小文字を判別しない。

string *object* [関数]
object の文字列表現を得る。もし *object* が文字列なら、*object* が返される。もし *object* が symbol なら、その pname がコピーされ、返される。(equal (string 'a) (symbol-pname 'a))==T であるが、(eq (string 'a) (symbol-pname 'a))==NIL である。もし *object* が数値なら、それを文字列にしたものが返される (これは Common Lisp と非互換である)。もっと複雑なオブジェクトから文字列表現を得るためには、最初の引数を NIL にした **format** 関数を用いること。

string< *str1 str2* [関数]

string<= *str1 str2* [関数]

string> *str1 str2* [関数]

string>= *str1 str2* [関数]
str1 と *str2* を先頭から順番に比較して、比較演算が成立した位置を返す。もし、成立しなければ、NIL を返す。文字の比較は、その文字のコードにに対して行なわれるため、A<Z である。。

string-left-trim *bag str* [関数]

string-right-trim *bag str* [関数]
str は、左（右）から探索され、もし *bag* リスト内の文字を含んでいるなら、その要素を削除する。一旦 *bag* に含まれない文字が見つかると、その後の探索は中止され、*str* の残りが返される。

string-trim *bag str* [関数]
bag は、文字コードの列である。両端に *bag* に書かれた文字を含まない *str* のコピーが作られ、返される。

substringp *sub string* [関数]
sub 文字列が *string* に部分文字列として含まれるなら、T を返す。大文字・小文字を判別しない。

8.4.1 日本語の扱い方

euslisp で日本語を扱いたい時、文字コードは **UTF-8** である必要がある。

例えば **concatenate** を用いると、リストの中の日本語を連結することが出来る。**ROS** のトピックとして一つずつ取ってきた日本語を、連結することで一つの **string** 型の言葉に変換したい時などに便利である。

(concatenate string "け" "ん" "し" "ろ" "う") → "けんしろう"

最初から全ての文字がリストに入っていて、文字を連結したい時はこのようにすればよい。

(reduce #'(lambda (val1 val2) (concatenate string val1 val2)) (list "我" "輩" "は" "ピ" "ー" "ア" "ー" "ル" "ツ" "ー" "で" "あ" "る"))

→ "我輩はピーアールツーである"

coerce を用いて、次のように書くことも出来る。

(coerce (append (coerce "私はナオより" cons) (coerce "背が高い" cons)) string)

→ "私はナオより背が高い"

8.5 Foreign String

foreign-string は、EusLisp のヒープ外にあるバイトベクトルの 1 種である。普通の文字列は、長さとバイトの列を持ったオブジェクトであるが、foreign-string は、長さと文字列本体のアドレスを持っている。foreign-string は文字列であるが、いくつかの文字列および列に対する関数は適用できない。**length**、**aref**、**replace**、**subseq** と **copy-seq** だけが foreign-string を認識し、その他の関数の適用はクラッシュの原因となる恐れがある。

foreign-string は、/dev/a??d??(??は 32 あるいは 16) の特殊ファイルで与えられる I/O 空間を参照することがある。そのデバイスがバイトアクセスにのみ応答する I/O 空間の一つに割り当てられた場合、**replace** は、いつもバイト毎に要素をコピーする。メモリの large chunk を連続的にアクセスしたとき、比較的に遅く動作する。。

make-foreign-string *address length* [関数]

address の位置から *length* バイトの **foreign-string** のインスタンスを作る。例えば、`(make-foreign-string (unix:malloc 32) 32)` は、EusLisp のヒープ外に位置する 32 バイトメモリを参照部分として作る。

8.6 ハッシュテーブル

hash-table は、キーで連想される値を探すためのクラスである（**assoc** でもできる）。比較的大きな問題において、hash-table は assoc より良い性能を出す。キーと値の組数が増加しても探索に要する時間は、一定のままである。簡単に言うと、hash-table は 100 以上の要素から探す場合に用い、それより小さい場合は assoc を用いるべきである。

hash-table は、テーブルの要素数が rehash-size を越えたなら、自動的に拡張される。デフォルトとして、テーブルの半分が満たされたとき拡張が起こるようになっている。**sxhash** 関数は、オブジェクトのメモリアドレスと無関係なハッシュ値を返し、オブジェクトが等しい (equal) ときのハッシュ値はいつも同じである。それで、hash-table はデフォルトのハッシュ関数に **sxhash** を使用しているので、再ロード可能である。**sxhash** がロバストで安全な間は、列や tree の中のすべての要素を探索するため、比較的に遅い。高速なハッシュのためには、アプリケーションにより他の特定のハッシュ関数を選んだ方がよい。ハッシュ関数を変えるためには、hash-table に`:hash-function` メッセージを送信すれば良い。簡単な場合、ハッシュ関数を`#'sxhash` から`#'sys:address` に変更すればよい。EusLisp 内のオブジェクトのアドレスは決して変更されないため、`#'sys:address` を設定することができる。

sxhash *obj* [関数]

obj のハッシュ値を計算する。**equal** な 2 つのオブジェクトでは、同じハッシュ値を生じることが保証されている。symbol なら、その pname に対するハッシュ値を返す。number なら、その `integer` 表現を返す。list なら、その要素全てのハッシュ値の合計が返される。string なら、それぞれの文字コードの合計をシフトしたものが返される。その他どんなオブジェクトでも、**sxhash** はそれぞれのスロットのハッシュ値を再帰的呼出しで計算し、それらの合計を返す。

make-hash-table *&key (:size 30) (:test #'eq) (:rehash-size 2.0)* [関数]

hash-table を作り、返す。

gethash *key htab* [関数]

htab の中から *key* と一致する値を得る。**gethash** は、**setf** を組み合せることにより *key* に値を設定することにも使用される。hash-table に新しい値が登録され、そのテーブルの埋まったスロットの数が 1/rehash-size を越えたとき、hash-table は自動的に 2 倍の大きさに拡張される。

remhash *key htab* [関数]

htab の中から *key* で指定されたハッシュ登録を削除する。

maphash *function htab* [関数]

htab の要素全てを *function* で **map** する。

hash-table-p *x* [関数]

もし *x* が hash-table クラスのインスタンスなら、T を返す。

hash-table [クラス]

:super	**object**
:slots	(key value count
	hash-function test-function
	rehash-size empty deleted)

hash-table を定義する。*key* と *value* は大きさが等しい一次元ベクトルである。*count* は、*key* や *value* が埋まっている数である。*hash-function* のデフォルトは **sxhash** である。*test-function* のデフォルトは **eq** である。*empty* と *deleted* は、*key* ベクトルのなかで空または削除された数を示す symbol(パッケージに

収容されていない）である。

:hash-function *newhash* [メソッド]

この hash-table のハッシュ関数を *newhash* に変更する。*newhash* は、1 つの引数を持ち、`integer` を返す関数でなければならない。*newhash* の 1 つの候補として **sys:address** がある。

9 ストリームと入出力

9.1 ストリーム

定義済みストリームは次のものであり、echo-stream と concatenated-stream は利用できない。

standard-input 標準入力 stdin fd=0

standard-output 標準出力 stdout fd=1

error-output 標準エラー出力 stderr fd=2 bufsize=1

terminal-io ***standard-input*** と ***standard-output*** で作られる入出力ストリーム

streamp *object* [関数]
object が **stream**, **io-stream** かそのサブクラスから作られているとき T を返す。

input-stream-p *object* [関数]
object がストリームで読み込み可能であれば、T を返す。

output-stream-p *object* [関数]
object がストリームで書き込み可能であれば、T を返す。

io-stream-p *object* [関数]
object が読み書き可能なストリームであれば、T を返す。

open *path &key :direction :input* [関数]
:if-exists :new-version
:if-does-not-exist
:permission #o644
:buffer-size 512

open は、*path* で指定されたファイルと結合されるストリームを作る。*path* は、文字列かパス名でよい。*:direction* は、`:input, :output` または `:io` のどれか 1 つでなければならない。いくつかの open オプション `:append, :new-version, :overwrite, :error` と NIL が *:if-exists* のパラメータとして許される。しかしながら、このパラメータは *:direction* が `:input` のとき無視される。*:if-does-not-exist* には、`:error, :create` か NIL のどれか 1 つをとる。`:new-version`, `:rename` と `:supersede` は認識されない。デフォルトとして、*:direction* が `:output` か `:io` でファイルが存在するとき、そのファイルに上書きする。`:input` において、ファイルがないとき、エラーが報告される。ファイルの存在を知るために、**probe-file** を使うことができる。*:buffer-size* のデフォルト値は 512 バイト、*:permission* のデフォルト値は `#O644` である。SunOS4 は、同時に 60 ファイルのオープンを許可している。

with-open-file *(svar path . open-options) &rest forms* [マクロ]
path という名のファイルが、*open-options* でオープンされ、そのストリームは *svar* にバインドされる。それから *forms* が評価される。ストリームは、*forms* の評価が終るかまたは **throw**, **return-from** やエラーで脱出したとき、自動的にクローズされる。**with-open-file** は、**unwind-protect** によって **close** とその内部書式を一緒にして定義されるマクロである。

close *stream* [関数]

stream がクローズされ、成功したら T を返す。*stream* が既にクローズされていた場合、NIL が返される。ストリームは、そのストリームオブジェクトが参照するものがないなら、GC によって自動的にクローズされる。

make-string-input-stream *string* [関数]
string から入力ストリームを作る。

make-string-output-stream *size* [関数]
size 長さの文字列のために出力ストリームを作る。その長さは自動的に拡張される。そのため、*size* は初期化時に配置する文字列のための補助情報である。

get-output-stream-string *string-stream* [関数]
string-stream に文字列を出力する。

make-broadcast-stream *&rest output-streams* [関数]
広報 (broadcast) ストリームを作り、このストリームに書かれたメッセージはすべての *output-streams* へ転送される。

9.2 リーダ (reader)

リーダのグローバル変数は：

read-base 読み込時の基数；デフォルトは 10

readtable reader 構文を決定するカレント読み込みテーブル

Reader のデフォルトマクロ文字は：

(	リスト読み込み
”	文字列読み込み
’	引用符表現読み込み
#	マクロ変換
;	コメント
‘	back-quote
,	list-time eval
@	追加
%	C 言語表記の数式読み込み

エスケープ文字：

\	単一文字エスケープ
|...|	多重文字エスケープ

エスケープされていない symbol が読まれると、全ての構成される文字はデフォルトで大文字に変換され、そして大文字の symbol は内部に蓄えられる。例えば、’abc と’ABC は同じ symbol とみなされる。エスケープは、それらを区別するのに必要である。’|ABC|, ’ABC と’abc は同一であるが、’|abc| と’abc は違う symbol である。デフォルトとして、大文字の symbol を入力したときでさえ、その symbol を表示するときは EusLisp のプリンタが内部の大文字表現から小文字に変換する。この変換は、プリンタによって実行されている。この変換は、`:UPCASE` を***print-case***に設定することにより、禁止される。

10. は整数の 10 として読まれ、実数の 10.0 ではない。’:’ がパッケージマーカーとして予約されているので、’|*g* : *pcube*| のように symbol を構成するものとして使うとき、エスケープ化しなければならない。この制限は、文字’:’ の構文により強制されないが、アルファベット順や letter の意味を決定する属性により強制される。その文字の属性は、リーダから堅く結ばれる (hardwired)。したがって、**copy-readtable** で新しい readtable を作ったり、**set-syntax-from-char** で文字のための構文的意味を組み直したりすることにより、ある文字の構文を変更可能であるが、その属性はどのようにしても変更することができない。その一方で、数字はいつも数字であり、アルファベットはアルファベットで、数値を表現するために’`#$%@`’ の様な文字を使用することはできない。

%は、EusLisp で拡張 read-macro 文字となっている。挿入記述により書かれた数式の前に%を付けることにより、その数式は lisp 用の式に変換される。具体例を上げると、`%(1 + 2 * 3 / 4.0)` は `(+ 1 (/ (* 2 3) 4.0))` に変換され、結果は `2.5` となる。C の様な関数呼出や行列参照も、lisp 形式に変換される。従って、`%(sin(x) + a[1])` は `(+ (sin x) (aref a 1))` として評価される。1 つ以上の引数を持つ関数や 2 次元以上の行列は、`func(a b c ...)` や `ary[1 2 3 ...]` のように記述し、`func(a,b,c)` や `ary[1][2][3]` のように書かない。相対表現や割り当てもまた、正しく扱われる。それで、`%(a < b)` は `(< a b)` に変換され、`%(a[0] = b[0] * c[0])` は `(setf (aref a 0) (* (aref b 0) (aref c 0)))` として得られる。単純な最適化は、

重複した関数呼出や行列参照をなくすことである。`%(sin(x) + cos(x) / sin(x))` は `(let* ((temp (sin x))) (+ temp (/ (cos x) temp)))` のように変換される。

マクロ変換は#文字が前に付いている。数値 (integer) 引数は、#とマクロ変換文字の間に与えられる。これは、どの数字 (0 .. 9) もマクロ変換文字として定義できないことを意味する。リーダの標準のマクロ変換文字は次の通り：

`#nA(..)` 行列

`#B` 2 進数

`#D` [度] から [ラジアン] への変換; #D180 = 3.14

`#F(...)` 実数ベクトル

`#nF((..))` 実数行列; #2F((..) (..)) is matrix

`#I(...)` 整数ベクトル

`#nI((...))` 整数行列

`#J(...)` 一般オブジェクト #J(myclass ....); 古い定義

`#O` 8 進数

`#P` パス名

`#R` [ラジアン] から [度] への変換; #R3.14 = 180.0

`#S(classname slotname1 val1 slotname2 val2 ...)` 構造体 (あらゆるオブジェクト)

`#V(...)` ベクトル #V(vectorclass ...)

`#X` 16 進数

`#(...)` ベクトル

`#n#` ラベル参照

`#n=` ラベル定義

`#'` 関数; コンパイルコードあるいは lambda-closure

`#\` 文字

`#,` 読み込み時に評価

`#+` 条件読みだし (positive)

`#-` 条件読みだし (negative)

`#*` ビットベクトル

`#:` 収容されてない symbol

`#|...|#` コメント; 入れ子可能

いくつかのリーダ関数は、*eof-error-p, eof-value* や *recursive-p* のパラメータを持っている。最初の 2 つのパラメータは、リーダが end-of-file に出会ったときの動作を制御する。*eof-error-p* のデフォルトは、T である。これは、eof 時のエラーの原因となる。eof の発生を知りたかったり、snatch control にシステムエラーを渡したくないなら、*eof-error-p* に NIL を指定すること。それで、読み込みの最中に eof が現れたとき、リーダはエラーループに入る代りに *eof-value* を返す。*eof-value* のデフォルトは、NIL である。そのため、実際に NIL が読まれたのか eof が現れたのか判別できない。それらを判別するためには、ストリームに決して現れない値を与えること。そのような特殊データオブジェクトを作るには、**cons** か **gensym** を使用すること。

recursive-p は、read-macro 関数にしばしば使用される。それは、リーダを再帰的に呼び出す。*recursive-p* の non-NIL 値は、読み込み処理がどこかで始まっていて、`#n=`や`#n#`によってラベル付けされる書式の読み込みのために内部テーブルを初期化すべきでないことをリーダに告げている。

read *&optional stream (eof-error-p t) (eof-value nil) recursive-p* [関数]
s 表現を 1 つ読み込む。

read-delimited-list *delim-char &optional stream recursive-p* [関数]
delim-char で終了する s 表現を読み込む。これは、コンマで区切られたリストや`#\]` のような特殊文字で終る数列を読むために役立つ。

read-line *&optional stream (eof-error-p t) (eof-value nil)* [関数]
`#\newline`(改行) で終了する 1 行を読み込む。返された文字列には、最後の改行文字を含まない。

read-char *&optional stream (eof-error-p t) (eof-value nil)* [関数]
1 文字読み込み、その整数表現を返す。

read-from-string *string &optional (eof-error-p t) (eof-value nil)* [関数]
string から s 表現を読み込む。最初の s 表現のみ読み込むことができる。もし、複数の s 表現を持つ *string* からの連続読み込み処理が必要であるならば、**make-string-input-stream** で作られる string-stream を用いること。

unread-char *char &optional stream* [関数]
stream に *char* を戻す。1 文字を越えて連続に戻すことはできない。

peek-char *&optional stream (eof-error-p t) (eof-value nil)* [関数]
stream から 1 文字を読み込むが、*stream* のバッファからその文字を削除しない。これは **read-char** に続いて **unread-char** を実行したものと同じである。

y-or-n-p *&optional format-string &rest args* [関数]
format-string と *args* を画面に表示して、“y か n か” を尋ねる。答えが “y” または “n” で始まらない場合、質問を繰り返す。y なら T そして n なら NIL を返す。 それ以外は起こらない。

yes-or-no-p *&optional stream* [関数]
format-string と *args* を画面に表示して、“yes か no か” を尋ねる。答えが “yes” または “no” でない場合、質問を繰り返す。yes なら T そして no なら NIL を返す。 それ以外は起こらない。

以下に示す readtable の操作関数の中で、readtable のデフォルト値はグローバル変数***readtable***の値である。

readtable-p *x* [関数]
x が readtable なら、T を返す。

copy-readtable *&optional from-readtable to-readtable* [関数]

to-readtable が書かれていなければ、新しい readtable を作る。*from-readtable* のすべての情報が *to-readtable* に移される。含まれる情報は、syntax table, read-macro table と dispatch-macro table でそれぞれ 256 個の要素を持つ。

set-syntax-from-char *from-char to-char [from-readtable to-readtable]* [関数]
from-readtable の中の *from-char* の syntax と read-macro 定義を *to-readtable* の中の *to-char* にコピーする。

set-macro-character *char func [non-teminating-p readtable]* [関数]
char の read-macro 関数として *func* を定義する。

get-macro-character *char [readtable]* [関数]
char の read-macro 関数を返す。

set-dispatch-macro-character *dispchar char func [readtable]* [関数]
dispchar と *char* の組み合せの dispatch read-macro 関数として *func* を定義する。

get-dispatch-macro-character *dispchar char [readtable]* [関数]
dispchar と *char* の組み合せの dispatch read-macro 関数を返す。

9.3 プリンタ (printer)

以下に示すものは、プリンタの行動を制御するための特殊変数である。

print-case この定数が:downcase なら、全ての symbol は小文字で印刷される。しかし、symbol は内部的に大文字で表現されたままである。

print-circle 再帰的参照を残したオブジェクトを印刷する。

print-object 全てのオブジェクトの詳細を印刷する。

print-structure #書式を使ってオブジェクトを印刷する。

print-level 数列の印刷可能深さ

print-length 数列の印刷可能長さ

print-escape 現在使用されていない。

print-pretty 現在使用されていない。

print-base 印刷時の基数；デフォルトは 10 進数

再帰的参照を持つオブジェクトを印刷するためには、再度読み戻しが必要なため、***print-circle***と***print-structure***を両方 T に設定し、オブジェクトを印刷すること。ユーザーが定義するほとんどのオブジェクトは再読み込み可能な書式に表示されるが、クラス, オブジェクトモジュールやパッケージをその方法で dump することはできない。なぜなら、クラスとオブジェクトモジュールは再配置不可能な実行コードを含み、パッケージの再読みだしは、構成される symbol 中に影響があるからである。

print *obj &optional stream* [関数]
prin1 に続いて **terpri** を行う。

prin1 *obj &optional stream* [関数]
書式に沿って s 表現を 1 つ出力する。その出力は、**read** によって再度読み戻しが可能である。書式には、スラッシュ（エスケープ）や引用符を含んでいる。

princ *obj &optional stream* [関数]
エスケープ（escape）や引用符（quote）の追加 (add) がないことを除いて、**print** と同じである。**princ** によるオブジェクト表示は、読み戻しできない。例えば、(princ 'abc) の出力は、(princ "abc") の出力と同じであるため、リーダはそれらを区別することができない。

terpri *&optional stream* [関数]
#\newline(改行) を出力して、*stream* を空にする。

finish-output *&optional stream* [関数]
出力 *stream* を空にする。

princ-to-string *x &optional (l 16)* [関数]

prin1-to-string *x &optional (l 16)* [関数]
文字列への出力ストリームを作り、*x* を書き込む。そして、**get-output-stream-string** を実行する。

format *stream format-string &rest args* [関数]

～A(ascii), ～S(S-表現), ～D(10 進数), ～X(16 進数), ～O(8 進数), ～C(文字), ～F(実数表現), ～E(指数表現), ～G(general float), ～V(dynamic number parameter), ～T(タブ) と ～% (改行) のフォーマット識別子のみ認識する。

```
(format t "~s ~s ~a ~a ~10,3f~%" "abc" 'a#b "abc" 'a#b 1.2)
--->  "abc" |A#B| abc a#b     1.200
```

pprint *obj &optional (stream *standard-output*) (tab 0) (platen 75)* [関数]
obj の最後の空白を除いたものを整形表示する。.

print-functions *file &rest fns* [関数]
file に *fns* の関数定義の"defun"書式を書き出す。

write-byte *integer stream* [関数]

write-word *integer stream* [関数]

write-long *integer stream* [関数]
integer を 1, 2 または 4 バイトにして書く。

spaces *n &optional stream* [関数]
空白を *n* 回出力する。

pf *func &optional stream *standard-output** [マクロ]
関数 *func* を整形表示する。コンパイルされた関数は、印刷できない。

pp-method *class selector &optional (stream *standard-output*)* [関数]
class クラスの中に定義された *selector* メソッドを整形表示する。

tprint *obj tab &optional (indent 0) (platen 79) (cpos 0)* [関数]
表形式で *obj* を印刷する。

print-size *obj* [関数]
印刷のときの *obj* の大体の長さを返す。

9.4 プロセス間通信とネットワーク

EusLisp は、4 種類の IPC 機能（**共有メモリ, メッセージキュー, FIFO** や**ソケット**）を備えている。[2]一般的に、この命令により性能が悪くなる。もし、マルチスレッド機能を使用するならば、12 節に記述されている同期関数も通信手段として用いることができる。これらの機能のうちで使用できるものは、Unix のバージョンや構成に依存する。

9.4.1 共有メモリ

Euslisp は、System5 の shmem ではなく、SunOS の mmap によって共有メモリを提供する。共有メモリは、**map-file** 関数によって配置される。**map-file** は、EusLisp のプロセスメモリ空間内にファイルを配置し、**foreign-string** のインスタンスを返す。データはこの foreign-string に対する文字列関数を用いることにより書き込みと読みだしができる。共有メモリは、システム依存のページ境界に配置されるので、配置アドレスを指定すべきではない。**:share** のキーパラメータが NIL に設定されているかまたは**:private** が T に設定されているファイルを配置することは、ファイルをプライベート (排他的) にアクセスすべきであることを意味する。しかし、メモリの共有化の目的から外れるため、**:share** のデフォルト値は T である。2 人のユーザーでファイルが共有されるとき、読み書きの許可は両方のユーザーに正確に設定されなければならない。残念なことに SunOS は、ネットワークを通して異なったワークステーション間のファイルの共有ををサポートしていない。

64 バイト長のファイルを 2 つの EusLisp で共有するプログラム例を下に示す。

```
;; 64 バイトのファイルを作る
(with-open-file (f "afile" :direction :output) (princ (make-string 64) f))
;; 配置する
(setq shared-string1 (map-file "afile" :direction :io))
;;
;; 他のプロセスの中で
(setq shared-string2 (map-file "afile" :direction :io))
```

その後、`shared-string1` に書かれたデータはすぐに `shared-string2` へ現れる。foreign string への書き込みは、**replace** か **setf** に **aref** を組み合せることにより可能である。

map-file *filename &key (:direction :input) :length (:offset 0) (:share t) (:address 0)* [関数]

filename という名のファイルをメモリ空間に配置する。*filename* は、ローカルファイル、NFS でマウントされたリモートファイル、または`/dev` の中のメモリデバイスのどれでも可能である。この関数の結果として **foreign-string** が返される。その要素は、**aref** によってアクセス可能である。`map-file` によって *:direction=:input* という条件で配置された foreign-string にデータを書き込むことは、segmentation fault の原因となる。

9.4.2 メッセージキューと FIFO

メッセージキューは、**make-msgq-input-stream** か **make-msgq-output-stream** で作られる。それぞれファイルストリームのインスタンスを返す。そのストリームは、ファイルに接続された他のストリームと同じように、読みだしや書き込み処理が許可されている。メッセージキューのストリームの `fname` は、作られるときに、key から設定する。

[2]UNIX における伝統的なプロセス間通信機構であるパイプは、'fork' との組み合せでいつも使用されている。EusLisp は、11.3 節で説明する **piped-fork** 関数を備えている。

FIFO に対するストリームを作るために、最初に **unix:mknod** 関数で、2 番目の引数を *mode*=#o10000 に設定した上で FIFO ノードを作り、ノーマルファイルとしてオープンする。メッセージキューと FIFO は、機械の上でローカルに作られ、機械内での通信チャンネルとしてのみ与えられる。

メッセージキューと FIFO は、自分のプロセスが終了した後でさえもシステムから削除されない。削除するためには、**unix:msgctl** か **ipcrm** コマンドが必要である。

make-msgq-input-stream *key &optional (buffer-size 128)* [関数]

key で示すメッセージキューに繋がる入力ファイルストリームを返す。

make-msgq-output-stream *key &optional (buffer-size 128)* [関数]

key で示すメッセージキューに繋がる出力ファイルストリームを返す。

9.4.3 ソケット

ソケットは、他の通信手段に比べて多才な機能を持っている。なぜなら、UNIX 領域の狭いホスト内とインターネット領域の広いネットーワーク内の両方で機能することができるためである。通信指向のソケット (SOCK_STREAM) と接続されないソケット (SOCK_DGRAM) の 2 つがサポートされている。両方ともまず **make-socket-address** 関数でソケットアドレスのオブジェクトを作らなければならない。**make-socket-address** は、socket-address のインスタンスを返す。UNIX 領域では、ソケットアドレスに UNIX ファイルシステム内のパス名を入れる。インターネット内では、ソケットアドレスにホスト名とポート番号と必要ならプロトコル番号を結合したものを入れる。もし、ポート番号が/etc/services に定義されていれば、service 名によって指定された symbol を通して参照される。**unix:getservbyname** 関数が symbol 化された service 名からポート番号を引き出すために使用される。1024 より小さいポート番号は、root ユーザーのために予約されている。特権のないユーザーは、1024 より大きなポート番号を個人的なソケットとして使用することを推奨する。

接続されたストリームは、両方向通信チャンネルとして供給されるが、接続確定処理は、入力・出力で別々である。片方がサーバーとして参照され、もう一方がクライアントとして参照される。サーバーとなった端 (service access point) は、最初に確定される。これは、**make-socket-port** 関数により作成される。この関数は、socket-port のインスタンスを返す。ソケットポートのオブジェクトは、**make-server-socket-stream** によって 1 つまたは複数のクライアントからの接続を受けるために使用される。**make-server-socket-stream** への呼び出しは、クライアントからの接続要求が実際に起こるまで実行待ち状態となる。クライアントは、ソケットアドレスを指定することによって **make-client-socket-stream** でソケットストリームを複数作ることができる。

```
;;; an example of IPC through a socket stream:
;;; server side
(setq saddr  (make-socket-address :domain af_inet :host "etlic2" :port 2000))
(setq sport  (make-socket-port saddr))
(setq sstream (make-server-socket-stream sport))
;;;
;;; client side
(setq caddr (make-socket-address :domain af_inet :host "etlic2" :port 2000))
(setq cstream (make-client-socket-stream caddr))
```

データベースや移動ロボットの環境シミュレータのようなアプリケーションでは、1 つのサーバーと複数のクライアント間の *multiple connection service*（多重接続サービス）が要求される。この型のサーバーは、

open-server 関数によりプログラムすることができる。カレントホスト名と与えられたポート番号から **open-server** は、接続要求にしたがってソケットポート (service access point) を作る。このポートは非同期なので、**open-server** は遮断されず、直ちに返信する。その後、接続要求はそれぞれ Euslisp のメインループを中断し、ソケットストリーム が非同期に作成される。このソケットストリームも非同期モードで働く。**open-server** の 2 番目の引き数にある非同期入力処理は、新しいデータがこのストリームに現れたときはいつでも呼び出される。30 以上の接続が可能であるため、同時に多くのクライアントがサーバーのデータにアクセスすることができる。

```
;; server side
(defun server-func (s)
   (case (read s) ...    ;do appropriate jobs according to inputs
(open-server 3000 #'server-func)
... do other jobs in parallel
;; client-1 through client-N
(setq s (connect-server "etlmmd" 3000))
(format s "..." ...) (finish-output s) ;issue a command to the server
(read s)                                ;receive response
```

確実な通信チャンネルとして供給される接続指向 ストリームと対照的に非接続 ソケットは、不確実な通信チャンネルである。メッセージがなくなったり、重複したり、故障したりする可能性がある。しかしながら、非接続 ソケットは、それぞれの接続にファイルディスクリプタを割り当てる必要が無いし、また、データやバッファのオーバーフローの読み込みをしないレシーバーでさえ送信処理を中断することができないという利点がある。

非接続ソケットを作るためには、以下の命令を使用する。メッセージは **unix:sendto** と **unix:recvfrom** によって変換される。

```
;;; receiver side
(setq saddr  (make-socket-address :domain af_inet :host "etlic2" :port 2001))
(setq sock   (make-dgram-socket saddr))
(unix:recvfrom sock)
;;;
;;; client side
(setq caddr (make-socket-address :domain af_inet :host "etlic2" :port 2001))
(setq sock (unix:socket (send caddr :domain) 2 0))
(unix:sendto sock caddr "this is a message")
;;;
;;; how to use echo service which is registered in /etc/services.
(setq caddr (make-socket-address :domain af_inet :host "etlic2"
                                 :port (car (unix:getservbyname "echo"))))
(setq echosock (unix:socket (send caddr :domain) 2 0))
(unix:sendto echosock caddr "this is a message")
(unix:recvfrom echosock)  --> "this is a messge"
```

make-socket-address *&key :domain :pathname :host :port :proto :service* [関数]

ソケットアドレスのオブジェクトを作る。

make-socket-port *sockaddr* [関数]

サーバー側のソケットポートを作る。この関数は、クライアントとの接続を確立するために使用される。

make-server-socket-stream *sockport &optional (size 100)* [関数]
クライアントからの接続要求を受けて、双方向ストリームを返す。

make-client-socket-stream *sockaddr &optional (size 100)* [関数]
サーバーのポートと接続をし、双方向ストリームを返す。

open-server *port remote-func* [関数]
インターネット領域内でホスト名と *port* で指定されるソケットポートを準備し、非同期に接続要求を待つ。接続が要求されたとき、それを受け新しいソケットストリームがオープンされる。ソケットポートにメッセージが到着したとき、*remote-func* は、ソケットポートを引き数として呼び出される。

connect-server *host port* [関数]
make-socket-address と **make-client-socket-stream** を連続的に呼び出しを行うための関数である。*host* と *port* で指定されるソケットストリームを返す。このソケットストリームは、クライアントがサーバーと通信を行うためのものである。ポートは、インターネット領域内用に作られる。

9.5 非同期入出力

select-stream *stream-list timeout* [関数]
stream-list の中で、入力処理が準備されているストリームを見つけリストで返す。もし、*timeout* 秒が経つまでにどのストリームも準備が出来ないときは、NIL を返す。**select-stream** は、入力ストリームのリストからアクティブなストリームを選ぶときに役立つ。そのストリームでは、入力処理が非同期で可能となる。*timeout* は、選択処理が失敗するまでの時間を示す。これは、実数でもよい。もし、*timeout* の指定がないときは、最低１つのストリームから入力が到着するまで **select-stream** は続けられる。もし、*timeout* が指定されどのストリームからも入力が現れなかったならば、**select-stream** は失敗し NIL を返す。

def-async *stream function* [マクロ]
stream にデータが到着したときに呼び出される *function* を定義する。*stream* は、ファイルストリームかソケットストリームのどちらかである。ファイルストリームにデータが来たとき又はソケットポートに接続要求が現れたとき、そのストリームを引き数として *function* が呼び出される。このマクロは、SIGIO ハンドラーとして装備され、ユーザーから与えられる実際の入力処理を実行するための *function* に置き換えられる。そして、*stream* が読み込み可能となったとき、非同期に SIGIO を発するために *stream* に関して **unix:fnctl** が使用される。

9.6 パス名

パス名は、OS に依存しないようにファイル名を解析あるいは構成する方法として与えられるものである。典型的なパス名は、次のような構成で成り立っていると仮定されている。host:device/directory1/.../directory-n/name.type.version。Euslisp は、UNIX の上で動作している限り、ホスト・デバイス・バージョンを無視する。**pathname** 関数は、文字列をディレクトリ要素・名前・型に分解し、パス名オブジェクトを返す。そのパス名は、`#P` を先頭につけた文字列として表示される。

pathnamep *name* [関数]
もし *name* がパス名ならば、T を返す。

pathname *name* [関数]
name はパス名または文字列で、パス名に変換される。最後の名前がディレクトリ名を示すために *name* の最後に”/”をつけることを忘れないこと。逆変換は **namestring** で実現される。

pathname-directory *path* [関数]
path からディレクトリ名のリストを返す。”/”ディレクトリは:ROOT と表現される。*path* は、文字列あるいはパス名である。

pathname-name *path* [関数]
path からファイル名の部分を返す。*path* は、文字列あるいはパス名である。

pathname-type *path* [関数]
path からファイルの型の部分を取り出す。*path* は、文字列あるいはパス名である。

make-pathname *&key :host :device :directory :name :type :version defaults* [関数]
directiory,name と *type* から新しいパス名を作る。UNIX 上では、他のパラメータは無視される。

merge-pathnames *name &optional (defaults *default-pathname-defaults*)* [関数]

namestring *path* [関数]
path の文字列表現を返す。

parse-namestring *name* [関数]

truename *path* [関数]
path で名前付けされているファイルの絶対パス名を見つける。

9.7 ファイルシステムインターフェース

probe-file *path* [関数]
path という名のファイルがあるかどうかをチェックする。

file-size *path* [関数]
path という名のファイルのサイズをバイト数で返す。

directory-p *path* [関数]
path がディレクトリならば、T を返す。そうでなかったとき（*path* が存在しなかったときを含める）は NIL を返す。

find-executable *file* [関数]
U *file* という名の UNIX コマンドを探し、フルパス名で返す。**find-executable** は、自分の path-list から実行できるファイルを探す UNIX の which コマンドとほとんど同じ関数である。

file-write-date *file* [関数]
file が最後に変更された日時を整数表現で返す。`(unix:asctime (unix:localtime (file-write-date file)))` で文字列表現が得られる。

file-newer *new old* [関数]
もし、*new* ファイルが *old* ファイルよりも最近に変更されているならば、T を返す。

object-file-p *file* [関数]
もし、*file* がヘッダー内のファイルの magic number を見ることによりオブジェクトファイルであったならば、T を返す。

directory *&optional (path ".")* [関数]
path の中の全てのファイルのリストを作る。

dir *&optional (dir ".")* [関数]
dir で指定されたディレクトリ内のファイル名を表示する。

10 評価

10.1 評価関数

エラーやシグナル (signal) に関する振る舞いを示すために、あらかじめそれぞれ特別の変数***error-handler*** と***signal-handler*** に適当な関数を設定する。修正あるいは続行できるエラーはない。エラーを解析後、現在の実行を **reset** または上位レベルへの適当な **throw** によって停止しなければならない。Euslisp の最上位レベルで `0` と名付けられた catch frame を作成しているので、**reset** は、`(throw 0 NIL)` と同等である。

エラーハンドラーは、*code msg1 form &optional (msg2)* という 3 つあるいは 4 つの引き数を持つ関数として定義しなければならない。*code* はエラーコードで、システムで定義されたエラーを示す。例えば、14 が' 引き数が合わない'、13 が' 関数が定義されていない' となる。これらの定義は、"c/eus.h" の中に定義されている。*msg1* と *msg2* は、ユーザーに示されるメッセージである。*form* は、エラーによって生じた s 表現である。

シグナルハンドラーは、*sig* と *code* の 2 つの引き数を受ける関数として定義されなければならない。*sig* は、1 から 30 までのシグナル番号である。*code* は、シグナル番号の中に定義された補助番号である。

最上位レベルでの^D(*end-of-file*) は、Euslisp の活動を停止させる。これは、Euslisp をフィルターとしてプログラムされているとき役に立つ。

eval-dynamic は、let や lambda 変数として使用される symbol に結び付く動的な変数を捜す関数である。デバッグするときに役に立つ。

identity *obj* [関数]

obj 自身を返す。**idnetity** と **quote** との違いに注意すること。**identity** が関数であるのに対して **quote** は、特殊書式 (special form) である。したがって、`(identity 'abc)` は `abc` と評価されるが、`(quote 'abc) == (quote (quote abc))` は`'abc` と評価される。**identity** は、多くの一般列関数の*:key* パラメータのデフォルト値としてしばしば用いられる。

eval *form [environment]* [関数]

form を評価して、その値を返す。もし、***evalhook***に *form* や *environment* を受ける関数を設定するならば、hook 関数を評価に入る前に呼び出すことができる。

apply *func &rest args* [関数]

args に *func* を適用する。*func* は、関数 symbol か lambda 書式か closure でなければならない。マクロと特殊書式 (special form) は適用出来ない。*args* の最後の要素は、他の *args* が空の引き数であるなら引き数のリストでなければならない。このように、もし、*args* の最後が NIL であったならば、**apply** はほとんど **funcall** と同じである。ただし、**apply** は **funcall** より 1 つ多くの引き数を持つことができる。`(apply #'max 2 5 3 '(8 2)) --> 8`.

funcall *func &rest args* [関数]

args に *func* を適用する。*args* の数は、*func* で要求されている引き数の数と一致しなければならない。

quote *obj* [特殊]

obj 自身を評価する。

function *func* [特殊]

closure 関数を作る。もし、*func* が symbol ならば、その関数定義が検索される。

evalhook *hookfunc form [env]* [関数]

hookfun を***evalhook***に結び付けた後、*form* を一度評価する。

eval-dynamic *variable* [関数]

スタックにある *variable*(symbol) の値を捜す。

macroexpand *form* [関数]

もし、*form* がマクロ call であるなら、それを展開する。もし、展開したものがまだマクロ call を含んでいるならば、マクロ call のない結果となるまでくり返し展開する。

eval-when *situation* {*form*}* [特殊]

situation は `compile, load, eval` のリストである。*form* は、現在の実行モードが *situation* と一致するときに評価される。**eval-when** は、コンパイラでの動作や環境を制御するために重要なものである。もし、`compile` が指定されたならば、*form* はコンパイラによって評価されるので、その結果はコンパイル結果に影響を及ぼすことになる。例えば、**defmacro** はコンパイル時にマクロ call を展開するためにコンパイラで評価されなければならない。もし、`load` が *situation* のリストに与えられたならば、*form* は load 時に load または評価されるためにコンパイルされる。すなわち、load 時にコンパイルされた関数が定義される。これは、コンパイラに期待される一般的な機能である。`load` は、コンパイラの環境を制御するために使用される。もし、`eval` が *situation* のリストに含まれているならば、*form* はソースコードが load されるときに評価される。

the *type form* [特殊]

form を *type* として宣言する。*type* は、`:integer, :fixnum, :float` で示されるクラスオブジェクトのどれかである。

declare *declaration** [特殊]

それぞれ *declaration* は、宣言指定や整数あるいは目的の symbol のリストである。宣言は、コンパイラが高速なコードを生成するために重要である。

special 特殊変数を宣言する。

type 変数の型を宣言する。; `(type integer count)`; 有効な型指定子は *integer*, *:integer*, *fixnum*, *:float* と *float* である。型指定子がここに示したものの 1 つであるならば、**type** キーワードを削除しても良い。そのため、`(integer count)` は正しい宣言である。*float-vector*,*integer-vector* などのような、その他の型（クラス）では、`(type float-vector vec1)` のように **type** を前に付ける必要がある。

ftype 関数の結果の型を宣言する。

optimize コンパイラの*optimize*パラメータに値 (0–3) を設定する。

safety コンパイラの*safety*パラメータに値 (0–3) を設定する。

space コンパイラの*space*パラメータに値 (0–3) を設定する。

inline 認識しない。

not-inline 認識しない。

proclaim *proclamation* [関数]

変数やコンパイラオプションをグローバルに宣言する。同様な宣言は、**declare** 特殊書式によって記述することができる。しかしながら、**proclaim** は、1 つの引数を持つ関数であり、宣言を評価する。

warn *format-string &rest args* [関数]

format-string と *args* で与えられる警告メッセージを***error-output***に出力する。

error *format-string &rest args* [関数]

error-handlerに結び付く現在のエラーハンドラー関数を呼び出す。デフォルトのエラーハンドラー'eu-serror' を***error-output***に最初に出力し *format-string* と *args* を **format** を用いて出力する。その後、新しい最上位レベルのセッション (session) に入る。プロンプトには、エラーセッションの深さを示す。**throw** にその番号を与えることにより、低いエラーレベルのセッションへ戻ることができる。

マルチスレッド Euslisp において、特殊変数はスレッド間で共有され、同じ***error-handler***が異なったスレッドから参照される。この不自由を避けるために、マルチスレッド Euslisp は **install-error-handler** 関数を備えている。その関数は、それぞれのスレッドに対して異なったエラーハンドラーをインストールする。

install-error-handler *handler* [関数]

handler を現在のスレッドのエラーハンドラーとしてインストールする。

10.2 最上位レベルの対話

EusLisp の標準の最上位レベルの入力－評価－出力のループ (loop) は、**eustop** により制御されている。euslisp が呼び出されたとき、**eustop** は、ホームディレクトリから".eusrc"というファイルをあるいは EUSRC 環境変数で指定されたファイルをロードする。それから、euslisp は、引き数リストで指定されたファイルをロードする。これらのロードが終了後、**eustop** は、普通の対話セッション (session) に入る。

standard-inputにユーザーの TTY が接続されたとき、**eustop** は、***prompt-string***（デフォルトとして"eus$"が設定されている）に設定されたプロンプトを出力する。そして、***terminal-io***ストリームから命令を入力する。もし、その入力がカッコで括られた行ならば、**eval** によって lisp 書式として扱われる。もし、入力行の最初の symbol に関数定義があった場合、その行に自動的にカッコが入れられ、評価される。もし、関数定義が見つからなかった場合、その特殊値 (special value) が調査され、その値が出力される。もし、その symbol になにも定義されてないならば、その行は UNIX 命令とみなされ、sh(Bourn's shell) へ渡される。もし、sh が一致する UNIX 命令を捜せなかったとき、"command unrecognized"というメッセージを出力する。このように、**eustop** は lisp のインタープリタおよび UNIX のシェルとして動作する。もし、入力を UNIX 命令として実行したくないとき、行の最初にコンマ',' を付ければよい。これは、対話実行 (interpretive execution) でエラーが発生したとき、動的な値を見るときに役に立つ。Euslisp はローカルなスコープ (lexical scope) を採用しているのでローカル変数の値を special として宣言されていない限りスコープの外から調査することは出来ない。

入力は、それぞれ行番号とともに***history***ベクトルに記憶される。csh の上と同じ様に**!**関数を入力することにより入力の詳細を呼び出すことができる。csh の履歴との違いは、**!**が関数であるため**!**の次に最低１つのスペースが必要である。また、コントロールキーを用いることにより emacs のように対話的に行を編集することができる。

通常^D (EOF) で Euslisp は終了する。上位レベル (普通は csh) に異常終了コードを返すためには、適当な条件コードをつけた **exit** を使用すること。

eustop は、SIGINT と SIGPIPE のためにシグナルハンドラーを設定する。そして、他のシグナルは catch しない。このため、SIGTERM や SIGQUIT のようなシグナルで Euslisp を終了できる。終了を避けたいとき、これらのシグナルを catch するためには、**unix:signal** 関数でユーザーで定義したシグナルハンドラーを設定すること。

- [変数]
　　現在の入力

＋ [変数]
　　直前の入力

＋＋ [変数]
　　２つ前の入力

＋＋＋ [変数]
　　３つ前の入力

***** [変数]
　　直前の結果

****** [変数]
　　２つ前の結果

******* [変数]
　　３つ前の結果

prompt-string [変数]
eustop で使用されるプロンプト文字列

program-name [変数]
この Euslisp が呼び出された命令。例えば、`eus,eusx,eusxview` やユーザーで作った euslisp など。

eustop *&rest argv* [関数]
デフォルトの最上位ループ

eussig *sig code* [関数]
SIGPIPE のデフォルトシグナルハンドラー。**eussig** は、SIGPIPE が到着したり他の最上位レベルループに入るときシグナル番号を出力する。

sigint-handler *sig code* [関数]
SIGINT(control-C) のデフォルトシグナルハンドラー。このシグナルで新しい最上位セッションへ入る。

euserror *code message &rest arg* [関数]
デフォルトのエラーハンドラーで、*message* を出力し、新しいエラーセッションへ入る。

reset [関数]
エラーループから脱出して、最後の **eustop** セッションへ戻る。

exit *&optional termination-code* [関数]
Euslisp プロセスを終了し、プロセスの状態コードとして *termination-code* (0..255) を返す。

h [関数]
historyの中に記憶されている入力履歴を関連する列番号とともに出力する。

! *&optional (seq 0)* [関数]
列番号 *seq* に関連する入力行を呼び出す。*seq* が 0 のとき、もっとも最近の命令が呼び出される。もし、*seq* が負の場合、現在行から相対的な位置にある行が呼び出される。呼び出された行が表示され、その行の最後にカーソルを移動する。次のコントロールキーが使用出来る。control-H (backspace) か control-B で 1 文字戻る。control-F か control-K で 1 文字進む。control-A で行の最初に移動する。control-L で行の最後に移動する。control-C で行の修正をキャンセルする。control-M (carriage-return) か control-J (line-feed) で行修正を終了して、修正した行を評価する。もし、*seq* が番号でなく symbol または文字列の場合、履歴リストを古い方に向かって検索し、symbol または文字列が含まれている命令行を返す。

new-history *depth* [関数]
depth の長さを持つように***history***を初期化する。*depth* 行が記憶される。現在の***history***に記録された入力行は、すべて消滅する。

10.3 コンパイル

Euslisp コンパイラは、Lisp プログラムの実行を高速化するために使用される。実行時間の 5〜30 倍の高速化とマクロ展開によるガーベージコレクション時間の大幅な減少が期待できる。

euscomp は、計算処理とベクトル処理のための最適化を行う。ときどきコンパイラが最適化を効率良く実行するために、固有の型宣言が必要となる。

compile-function は、1 つずつ関数をコンパイルする。**compile-file** は、すべてのソースファイルをコンパイルする。**compile-file** を実行している間、ファイル内のすべての書式が読み込まれ評価される。これは、現在の Euslisp の環境を変化させる。例えば、**defparameter** は symbol に新しい値を設定するし、**defun** はコンパイルされていない関数をコンパイルされた関数に置き換える。これらの予期しない影響を避けるためには、`compile` 指定のない **eval-when** を使用したり、**euscomp** を使用して別プロセスとしてコンパイラを実行したりする。

euscomp は UNIX のコマンドで、普通 **eus** にシンボリックリンクされている。幾つかのオプションを持っている。-O フラグは C コンパイラの最適化を指示し、-O1,-O2, -O3 はそれぞれ Euslisp コンパイラの最適化のレベルを指示する。これは、`(optimize 1 or 2 or 3)` と宣言するのと同等である。-S0, -S1, -S2, -S3 は、**compiler:*safety*** に 0,1,2,3 を設定する。もし***safety***が 2 より小さければ、割り込みチェックのためのコードを発行しない。もし、プログラムが無限ループに入ったとき、制御を失うことになる。もし***safety***が 0 のときは、引き数の数をチェックしない。-V フラグは、コンパイルされている関数名を表示する。-c フラグは、cc の実行や fork を防ぐ。-D は、***features***リストに続く引き数を置く。これは、読み込みマクロ`#-`と`#+` を用いた条件付きコンパイルのために使用することができる。

コンパイラは”xxx.l”という名の Euslisp ソースプログラムを中間 C プログラムファイル”xxx.c” とヘッダーファイル”xxx.h”に変換する。それから、C コンパイラが実行され、”xxx.o”が生成される。中間ファイル”xxx.c”と”xxx.h”はクロスコンパイルの目的のために残される。したがって、違うアーキテクチャーの機械の上で使用したいとき、UNIX 命令の cc で”xxx.c”ファイルをコンパイルするだけでよい。コンパイルされたコードは、’`(load "xxx")`’ によって Euslisp にロードされる。

中間ファイルはそれぞれ、”eus.h”ヘッダーファイルを参照する。このファイルは、`*eusdir*/c` ディレクトリに置かれていると仮定している。`*eusdir*`は、`EUSDIR` 環境変数からコピーされる。もし設定されてなければ、`/usr/local/eus/`がデフォルトディレクトリとして扱われる。

コンパイルされたとき、中間の C プログラムは普通元のソースコードよりもかなり大きくなる。例えば、1,161 行の lisp ソースコード”l/common.l”は、8,194 行の”l/common.c”と 544 行の”l/common.h”に展開される。1,000 行の lisp ソースをコンパイルするのは、難しい作業ではないが、10,000 行近い C のプログラムを最適コンパイルすることは、長い時間 (数分) かかるとともに、たくさんのディスク空間を消費する。そのため、もし相対的に大きなプログラムのコンパイルをするならば、/var/tmp に十分なディスクがあるかどうかを確認すること。そうでなければ、CC は死ぬだろう。`TMPDIR` 環境変数をもっと大きなディスク部分に設定することが助かる道である。

リンクがロード時または実行時に実行されるので、eus のカーネルがバージョンアップされても再コンパイルする必要はない。もう一方で、実行時リンクは不便なことがある。２つの関数 A と B が”x.l”ファイルにあり A が B を呼び出していると仮定する。”x.l”をコンパイル後、”x.o”をロードし内部で B を呼び出している A を呼び出そうとする。それから、B の中で bug を見つけると、たぶん B を再定義しようとするだろう。ここで、コンパイルされた A とコンパイルされていない B とができる。再び A を呼び出したとすると、A はまだ古いコンパイルされている B を呼び出す。これは、A が最初に B を呼び出したとき固定的にリンクされるからである。この問題を避けるためには、A を再定義しなおすかあるいは”x.o”がロードされた直後で A を呼び出す前に B を再定義しなければならない。

コンパイルされたコードがロードされるとき、一般的に defun や defmethod の列である最上位コードが実行される。この最上位コードはロードモジュールのエントリ関数として定義されている。コンパイラがそのエン

トリ関数の名前を付け、ローダがこの関数の名前を正確にわからなければならない。状況を簡単にするために、コンパイラとローダの両方ともそのエントリ関数の名前としてオブジェクトファイルの basename と同一のものと仮定する。例えば、もし”fib.l”をコンパイルしたならば、コンパイラは”fib.c”のエントリ関数として”fib(...)”を生成する。そして、ローダはオブジェクトファイル”fib.o”の中から”fib”を探す。最終的にオブジェクトファイルは Unix の”cc”や”ld”で生成されるので、このエントリ関数名は、C 関数の命名ルールを満足しなければならない。したがって、ファイル名として C の予約キーワード（例えば、”int”, ”struct”, ”union”, ”register”, ”extern”など）や”c/eus.h”に定義されているプライベート指示語（例えば、”pointer”, ”cons”, ”makeint”など）を避けなければならない。もし、ソースファイルの名前としてこれらの予約語の内の１つを使用しなければならないならば、コンパイラやローダの*:entry* 引数を別に指定すること。

closure の使用には制限がある。closure の中の **return-from** 特殊書式と **unwind-protect** の中の clean-up 書式はいつも正しくコンパイルされるわけではない。

disassemble は、実現されていない。コンパイルされたコードを解析するためには中間 C プログラムを見るかあるいは **adb** を使用する。

euscomp {*filename*}* [UNIX コマンド]

Euslisp コンパイラを呼びだす。

compile-file *srcfile &key (:verbose nil)* [関数]
(:optimize 2) (:c-optimize 1) (:safety 1) ;optimization level
(:pic t) ;generate position independent code to build library
(:cc t) ; run c compiler
(:entry (pathname-name file))

ファイルをコンパイルする。”.l”が *srcfile* の拡張子として仮定される。もし、*:verbose* が T ならば、コンパイルされた関数やメソッド名が表示される。これは、エラーが発生した箇所を簡単に探すのに役立つ。*:optimize, :c-optimize* と *:safety* は、最適化のレベルを指定する。モジュールが作成中に Euslisp のコアにハードリンクされていないかぎり、*:pic* は、T に設定すべきである。

compile *funcname* [関数]

関数をコンパイルする。**compile** は、最初に関数定義をテンポラリファイルに出力する。そのファイルは、**compile-file** によってコンパイルされ、それから **load** によってロードされる。テンポラリファイルは削除される。

compile-file-if-src-newer *srcfile &key compiler-options* [関数]

srcfile が対応するオブジェクトファイルよりも新しい（最近変更された）ならば、コンパイルする。そのオブジェクトファイルは、”.o”拡張子を持っていると仮定される。

optimize [変数]

コンパイラの最適化レベルを制御する。

verbose [変数]

non-NIL が設定されたとき、コンパイルされている関数名やメソッド名そしてコンパイルに要した時間を表示する。

safety [変数]

安全性レベルを制御する。

10.4 プログラムロード

load *fname &key :verbose* **load-verbose** [関数]
:package **package**
:entry *(pathname-name fname)*
:symbol-input *"/usr/local/bin/eus"*
:symbol-output *"a.out"*
:ld-option *""*

load はソースファイルあるいはコンパイルされたオブジェクトファイルを Euslisp に読み込むための関数である。もし、*fname* で指定されたファイルが存在するとき、Euslisp はロードする。そのファイルがソースとバイナリーの内どちらかは、magic number を見ることにより自動的にチェックされる。もし、そのファイルが存在しないが'.o' の型のファイルが存在する場合、そのファイルをオブジェクトファイルとしてロードする。その他、'.l' のファイルが見つかったならば、ソースプログラムとしてロードする。もし、ファイル名の最初に"/"を付ける絶対パスを指定していない場合、**load** は、グローバル変数`*load-path*`に指定されているディレクトリの中からファイルを検索する。例えば、`*load-path*`が`("/user/eus/" "/usr/lisp/")`であり、*fname* として`"llib/math"`が与えられたとき、**load** は`"/user/eus/llib/math.o", "/usr/lisp/llib/math.o", "/user/eus/llib/math.l", "/usr/lisp/ll`をこの順番に捜す。適当なファイルが見つからなければ、エラーを返す。

:entry オプションは、ロードモジュールを初期化する入力アドレスを指定する。たとえば、`:entry "_myfunc"` オプションは`_myfunc` から実行を始めることを意味する。デフォルト入力アドレスは、10.3 節に記述されているようにロードされたファイル名の basename である。ライブラリモジュール名は*:ld-option* オプション文字列の中に指定することができる。たとえば、suncore ライブラリを使用するモジュールをリンクするために、*:ld-option* には `"-lsuncore -lsunwindow -lpixrect -lm -lc"` を与える必要がある。Solaris システム以外では、ライブラリがリンクされるとき **ld** は 2 度実行される。1 度はサイズを決定するため、2 度目は固有のメモリーに実際にリンクするため。

:symbol-input と *:symbol-output* オプションはあるオブジェクトモジュールから他のモジュールへの参照を解決するため、あるいはライブラリーの 2 重ロードを避けるために使用される。A,B 2 つのオブジェクトモジュールがあり,B が A の中で定義されている symbol を参照しているとする。`:symbol-output ="a.out"`を指定してモジュール A をロードする。このリンクによって生成された symbol 情報は、`a.out` に書き込まれる。モジュール B をロードするためには、B から A への参照を解決する`:symbol-input = "a.out"`を指定しなければならない。

Solaris2 OS において、コンパイルコードのロードは、動的ロードライブラリの中の *dlopen* を呼び出すことにより実行している。*dlopen* の使用は、共有オブジェクトに制限されている。そのオブジェクトは、位置に依存するようにコンパイルされるために、"-K pic"オプションを指定する。また、*dlopen* は同じファイルを 2 度オープンすることができないので、既にロードされているファイルに関しては、*dlclose* を最初に実行する。

load-files *&rest files* [関数]
`:verbose` に T を設定し、*files* を連続的にロードする。

modules [変数]
これまでにロードされたモジュールの名前のリストを持つ。

provide *module-name* [関数]
module-name をすでにロードされたモジュールの名前として`*modules*`の中に加える。*module-name* は symbol あるいは文字列でなければならない。**require** の呼び出しは、完全なモジュールを構成するファイルの最初に存在しなければならない。

require *module-name &optional file* [関数]

`*modules*`の中に *module-name* が見つからなかったとき *file* をロードする。**provide** と **require** は、モジュールの中の依存を制御し、基本モジュールの 2 重ロードを避けるために使用される。1 つの基本モジュール”A”と 2 つの応用モジュール”B”と”C”があったとし、”B”と”C”は互いに依存しないが”A”に依存しているとする。それぞれのファイルの最初に **provide** でモジュール名を宣言する。”A”モジュールがどのモジュールにも依存していないなら、なにも **require** しなくてよい。”B”や”C”の中の **provide** 呼び出しに続いて `(require "A" "a.o")` が実行される。もし”B”をロードするとき、`*modules*`の中に”A”が見つからなければ”a.o”もロードし、`*modules*`に”A”と”B”のモジュール名を追加する。その後、もし”C”がロードされるなら、”A”モジュールはロードされず、”C”が`*modules*`に追加されるのみである。

system:binload *opath qpath &optional (entry (pathname-name opath)) (symfile ”/usr/local/bin/eus”) (symout ”a.out”) (ldopt ””)* [関数]

バイナリファイルをロードしリンクする。

system:txtload *fname* [関数]

10.5 デバッグ補助

describe *obj &optional (stream *standard-output*)* [関数]
describe はオブジェクトの slot ごとの中身を表示する。

describe-list *list &optional (stream *standard-output*)* [関数]
list 内のそれぞれの要素に **describe** を実行する。

inspect *obj* [マクロ]
inspect は **describe** の対話版である。オブジェクトのそれぞれの slot を表示するためにサブ命令を受けたとき、slot の中に深く入ったりあるいは新しい値を slot に設定したりする。'?' 命令でサブ命令のメニューを見ることができる。

more *&rest forms* [関数]
standard-outputにテンポラリーファイルを結び付けて *forms* を評価した後、そのテンポラリーファイルを UNIX の'more' 命令を使用して***standard-output***に出力する。**more** は **describe** のような関数によって発生した長い出力を見るときに役に立つ。

break *&optional (prompt ":: ")* [関数]
break ループに入る。現在バインドされている環境が実施されている間、入力の前に","を付けることによりローカル変数を見ることができる。break を終了したいとき、control-D を入力する。

help *topic* [関数]
help は、*topic* に関して短い説明を表示する。*topic* は、ふつう関数 symbol である。文章は KCL のマニュアルから借りているため、説明が Euslisp 関数のものといつも合っているとは限らない。

apropos *key* [関数]
apropos は、関数や変数の正確な名前を忘れていて、その一部あるいは不確かな名前しか知らないときに役に立つ。symbol-name の中に部分文字列として *key* を含むすべての symbol を表示する。

apropos-list *key* [関数]
appropos と似ているが表示はしない、しかしリストとして結果を返す。

constants *&optional (string "") (pkg *package*)* [関数]
pkg の中に定数として定義され *string* と合う symbol をすべてリスト出力する。

variables *&optional (string "") (pkg *package*)* [関数]
pkg の中にグローバル値として割り当てられ *string* と合う symbol をすべてリスト出力する。

functions *&optional (string "") (pkg *package*)* [関数]
pkg の中にグローバル関数として定義され *string* と合う symbol をすべてリスト出力する。

btrace *&optional (depth 10)* [関数]
depth レベルの履歴を呼び出し表示する。

step-hook *form env* [関数]

step *form* [関数]
step と **trace** は関数の上でのみ正確に働く。マクロや特殊書式では働かない。

trace *&rest functions* [関数]

functions のトレースを始める。*functions* が呼び出されたときはいつでも、その引き数と結果を表示する。

untrace *&rest functions* [関数]
トレースを終了する。

timing *count &rest forms* [マクロ]
forms を *count* 回実行し、*forms* の 1 回の実行に要する時間を計算する。

time *function* [マクロ]
function によって経過した時間を測定し始める。

sys:list-all-catchers [関数]
すべての **catch** タグを返す。

sys:list-all-instances *aclass [scan-sub]* [関数]
すべてのヒープの中から *aclass* で指定されるインスタンスをすべて探し、集める。もし、*scan-sub* が NIL なら、*aclass* の確実なインスタンスをリストする。そうでなければ、*aclass* のインスタンスあるいはサブクラスが集められる。

sys:list-all-bindings [関数]
バインドされるスタックを探し、アクセス可能な値すべてをリストで返す。

sys:list-all-special-bindings [関数]
スタックを捜し、値をすべてリストアップする。

10.6 ダンプオブジェクト

EusLisp のリーダとプリンタは、どのようなオブジェクトも再読みだし可能な書式でファイルに出力できるように設計されている。オブジェクトは相互参照あるいは再帰参照を持っていてもよい。`*print-circle*`と`*print-object*`に T を設定したとき、この特徴はは可能となる。次の関数はこれらの変数を T にし、ファイルをオープンし、オブジェクトを表示する。これらの関数のもっとも重要な用途は、相互参照を持つ 3D モデルの構造体をダンプすることである。

dump-object *file &rest objects* [関数]

dump-structure *file &rest objects* [関数]
再び読み戻しができるような書式で *file* に *objects* をダンプする。

dump-loadable-structure *file &rest symbols* [関数]
symbol にバインドされたオブジェクトを *file* にダンプする。その *file* は簡単にロードすることによって読み戻すことができる。

10.7 プロセスイメージ保存

このプロセスイメージ保存は、Solaris の動的ロード機能に深く依存しているため、Solaris2 の Euslisp においてサポートされていない。Solaris の動的ロード機能は、共有するオブジェクトを `sbrk` 点の上の位置に依存した形でロードする。

save *path &optional (symbol-file "") starter* [関数]
save は、現在の Euslisp 処理の環境をファイルにダンプする。そのファイルは、後で UNIX 命令として呼び出すことができる。もし関数名が *starter* によって指定されているなら、その関数はその命令の実行が始まるときに評価される。それぞれの命令行の引き数は Euslisp の中で強制的に文字列にされ、引き数として *starter* に受け渡される。それで、命令行を解析できる。***standard-input***と***standard-output*** を除くすべてのストリームを確実にクローズしていなければならない。ファイルがオープンされた状態を保存することはできない。また、**mmap** を試そうとしてはならない。**mmap** はインターネットのソケットストリームを作るとき、見えない所で実行されている。Sun のネットワークライブラリは host-by-name のような NIS 情報をメモリ上にいつも展開し、プロセスの最上位に置くため保存できない。そのため、保存されたイメージが実行されるても、ネットワークライブラリへのアクセスはどれも失敗し、コアダンプが生じる。Xwindow もこのライブラリを使用している。それで、一度 Xserver に接続されたプロセスイメージを保存することはできない。

10.8 最上位レベルのカスタマイズ

Euslisp が UNIX から呼び出されるとき、***toplevel***にバインドされる最上位関数によって実行が始められる。この関数は、`eus` の中の `eustop` と `eusx` の中の `xtop` である。**save** の 3 番目の引き数に独自の関数を指定することによりこの最上位関数を変更することができる。

この最上位関数は、任意の数の引き数を受け取れるようにプログラムすべきである。その命令行の上の引き数はそれぞれ、強制的に文字列にされ、最上位関数に渡される。以下に示すプログラムは、最初の引き数に与えられたファイルからくり返し読み込み、2 番目の引き数のファイルに整形表示する。

```
(defun pprint-copy (infile outfile)
  (with-open-file (in infile)
     (with-open-file (out outfile :direction :output)
        (let ((eof (cons nil nil)) (exp))
           (while (not (eq (setq exp (read in nil eof)) eof))
              (pprint exp out))))))
(defun pprint-copy-top (&rest argv)
   (when (= (length argv) 2)
      (pprint-copy (first argv) (second argv))))
```

一度これらの関数を Euslisp の中に定義すれば、(save "ppcopy" "" 'pprint-copy-top) で ppcopy という名の UNIX で実行可能な命令を作る。

Solaris 上の Euslisp において、**save** がないので、最上位評価関数はこの手法では変更できない。代わりに、独自の最上位評価関数を定義するために lib/eusrt.l を修正し、***toplevel***に設定することができる。lib/eusrt.l には、Euslisp の起動時に評価される初期化手続きを定義している。

10.9 その他の関数

lisp-implementation-type [関数]

"EusLisp"を返す。

lisp-implementation-version [関数]

この Euslisp の名前、バージョン、作成日を返す。この文字列が起動時にも表示される。"MT-EusLisp 7.50 X 1.2 for Solaris Sat Jan 7 11:13:28 1995"

第 II 部
EusLisp 拡張

11　システム関数

11.1　メモリ管理

メモリ管理の設計は、オブジェクト指向言語の柔軟性と効率性にたいへん影響を及ぼす。Euslisp は、**フィボナッチバディ法**を基本に統一した方法でオブジェクトをメモリに割り当てる。この方法は、chunk と呼ばれる大きなメモリのプールを小さなセルに分割する。それぞれのセルは、サイズが等しくなく、フィボナッチ数がそれぞれ割り当てられる。chunk メモリは、`symbol`，`cons`，`string`，`float-vector` などのような様々な型のオブジェクトのための同次なデータ容器である。それらのサイズは chunk と一致する長さである。chunk は、固定、動的、再配置可能、交替可能などのようなどんな特別な属性も持っていない。EusLisp のヒープメモリは、chunk の集合である。そして、ヒープは UNIX より新しい chunk を得ることにより動的に拡張することができる。拡張は、動作中に自動的に発生するかあるいはユーザーが **system:alloc** 関数を呼び出すことにより発生する。自動的に処理されるとき、使用可能なメモリサイズは合計のヒープサイズの約 25%に保つ。この比率は、**sys:*gc-margin***パラメータに 0.1 から 0.9 までの値を設定することにより変更することができる。

すべてのヒープメモリを使いきったとき、mark-and-sweep 型のガーベージコレクション (GC) を始める。ルート（パッケージ，クラスやスタック）からアクセス可能なセルは，同じアドレスのままである。他のアクセス不可能なセルは、矯正され free-lists にリンクされる。GC の間にコピーやコンパクト化は発生しない。ガーベージされるセルが矯正されるとき、その隣接セルが free かどうかチェックされる。そして、できるだけ大きなセルを構成するようにマージされる。しかしながら、このマージは、ときどき意味の無いものになる。なぜなら、もっとも頻繁に呼び出されるメモリアロケータである `cons` が、そのマージされたセルを最も小さなセルに分割することを要求するからである。したがって、Euslisp は `cons` の高速化のためにマージされないある特定の量のヒープを残すことを許可している。この比率は、**sys:*gc-merge***パラメータによって決定される。その値のデフォルトは 0.3 である。**sys:*gc-merge***に大きな値を設定することにより、マージされないヒープを多く残す。これは、`cons` が要求されるとき、buddy-cell の分割が滅多に起こらないので、`cons` の性能を改善する。これは、また 3 次元ベクトルのような相対的に小さなセルのアロケーションについてすべて成り立つ。

sys:gc は、明示的にガーベージコレクターを呼び出す。そして、ヒープに配置された空いているワード数と全体のワード数（バイト数ではない）を示す 2 つの整数のリストを返す。

もし、実行中に”fatal error: stack overflow”が報告され、そのエラーが無限ループあるいは再帰処理によるものでないと確信するならば、**sys:newstack** で Lisp のスタックの大きさを拡張すれば回避できる。**sys:newstack** を設定する前には、**reset** を実行しなければならない。なぜなら、スペシャルバインドと **unwind-protect** の整理用の書式が現在のスタックの中からすべて捨てられるためである。

新しいスタックが配置された後、オープニングメッセージを表示するところから実行を始める。デフォルトのスタックサイズは、16Kword である。Lisp のスタックは、システムのスタックと別物である。前者は、ヒープ上に配置され、後者は、オペレーティングシステムによってスタックセグメント内に配置される。もし、”segmentation fault”エラーが発生したならば、システムのスタックが小さいことにより発生したことが考えられる。csh のコマンド `limit` で、システムのスタックサイズを増加することにより、解決できる可能性がある。

sys:reclaim と **sys:reclaim-tree** 関数は、オブジェクトにより占有されているセルをメモリマネージャーに戻す。そのため、ガーベージコレクションを呼び出すことなしにその後も再使用をすることができる。しかし、それらのセルが他のセルから参照されていないことが確実でなければならない。

memory-report と **room** 関数は、メモリの使用に関する統計をセルのサイズやクラスによりソートして

表示する。

address は、オブジェクトのバイト換算したアドレスを返す。このアドレスはプロセスに独自のものであるから、この関数はハッシュテーブルが使用するハッシュ関数に有用である。

peek と **poke** は、メモリから直接データを読み書きできる関数である。アクセスする型は、`:char`, `:byte`, `:short`, `:long`, `:integer`, `:float`, `:double` のどれかにすべきである。例えば、`(SYS:PEEK (+ 2 (SYS:ADDRESS '(a b))) :short)` は、cons セルのクラス ID（ふつう 1 である）を返す。

'**list-all-**' を名前の前に付けている幾つかの関数がある。これらの関数は、システムのリソースあるいは環境のリストを返し、動的なデバッグに有用である。

sys:gc [関数]

ガーベージコレクションを実行する。割り当てられている中で空いているワード数および全体のワード数のリストを返す。

sys:gctime [関数]

3 つの整数のリストを返す。1 つは、GC を呼び出した回数。2 つは、セルをマーキングするために使用した時間（1 ユニットに 1/60 秒）。3 つが、矯正（マーキングを外し、マージする）のために使用した時間。

sys:alloc *size* [関数]

ヒープに少なくとも *size* ワードのメモリを配置し、実際に配置されたワード数を返す。

sys:newstack *size* [関数]

現在のスタックを廃棄し、*size* ワードの新しいスタックを配置する。

sys:*gc-merge* [変数]

メモリ管理用のパラメータ。***gc-merge***は、GC によりマージされずに残すヒープメモリの比率を示す。このマージされない領域は、すぐに **cons** のような小さなセルに満たされる。デフォルトの値は、0.3 である。この値を 0.4 のように大きくすると、マージされない空きヒープが 40%であることを示し、cons のためには役立つが、実数ベクトルやエッジや面などのような大きなセルの確保には、害を及ぼす。

sys:*gc-margin* [変数]

メモリ管理用のパラメータ。***gc-margin***は、全体のヒープに対する空きヒープの比率を決定する。メモリは、UNIX から獲得したものであるため、空き空間はこの比率より小さくならない。デフォルトは、0.25 であり、GC の際に 25%以上の空き空間が維持されることを意味する。

sys:reclaim *object* [関数]

ごみとして *object* を廃棄する。そのオブジェクトは、他のどのオブジェクトからも絶対に参照されないことが保証されなければならない。

sys:reclaim-tree *object* [関数]

object から通過できる symbol を除いてすべてのオブジェクトを矯正する。

sys:btrace *num* [関数]

Lisp のスタックの *num* 番目の深さの情報をトレースして表示する。

sys:memory-report *&optional strm* [関数]

セルのサイズでソートしたメモリ使用のテーブルを *strm* ストリームに出力する。

sys:room *output-stream* [関数]

クラスで整列したメモリ配置の情報を出力する。

sys:address *object* [関数]

プロセスのメモリ空間内にある *object* のアドレスを返す。

sys:peek *[vector] address type* [関数]

address で指定されたメモリ位置のデータを読みだし、それを整数として返す。*type* は、**:char, :byte, :short, :long, :float, :double** の内の 1 つである。もし、*vector* が与えられなかったなら、そのアドレスは、UNIX の処理空間として扱われる。例えば、Sun において a.out のヘッダーは`#x2000` に置かれるため、(`sys:peek #x2000 :short`) は magic number（ふつうは`#o403`) を返す。Solaris2 は、ELF ヘッダーを`#10000` に置くため、(`sys:peek #x10000 :long`) が”ELF”を表現する文字列である`#xff454c46` を返す。もし、*vector* が与えられたならば、それは **foreign-string** であり、アドレスは *vector* の起点からのオフセットとして認識される。(`sys:peek "123456" 2 :short`) は、”34”を表現する short word を返す。(`#x3334(13108)` である)

アドレス位置については十分注意すること。`short, integer, long. float, double word` を奇数アドレスから読み出すと、”bus error”になる。

sys:poke *value [vector] address value-type* [関数]

value を *address* で指定された位置に書き込む。プロセスのメモリ空間内のどこでも書き込むことができるため、特に注意をしなければならない。プロセスの空間の外へ書き込んだなら、確実に”segmentation fault”が発生する。`short, integer, long. float, double word` を奇数アドレスに書き込んだ場合、”bus error”が発生する。

sys:list-all-chunks [関数]

配置されたすべてのヒープの chunk をリストアップする。他に有用な実行関数はない。

sys:object-size *obj* [関数]

obj からアクセス可能なセルとワードの数を計算する。*obj* から参照可能なすべてのオブジェクトが探索される。そして、3 つの数のリストが返される。1 つ目は、セルの数。2 つ目は、これらのオブジェクトに論理的に配置されたワード数（すなわち、ユーザーからアクセスできるワード数)。3 つ目は、物理的に配置されたワード数。これは、ヘッダーとメモリ管理のための特別なスロットを含む。探索は、symbol で停止する。すなわち、`property-list` あるいは `print-name string` のような symbol から参照されるオブジェクトは、カウントされない。

11.2 UNIX システムコール

EusLisp は、UNIX オペレーティングシステムのシステムコールとライブラリ関数とに直接関連する関数を取りそろえている。これらの関数の詳細については、UNIX system interface (2) を調べること。*unix-package* に定義されているこれらの低レベル関数の使用は、ときどき危険をはらむ。できるだけ他のパッケージに定義されている高レベル関数を使用すること。例えば、unix:socket, unix:bind, unix:connect などの代わりに 9.4 節に記述されている IPC 機能を使用すること。

11.2.1 時間

unix:ptimes [関数]
経過時間、システム時間、ユーザー時間、サブプロセスのシステム時間とサブプロセスのユーザー時間の 5 つの要素を持つリストを返す。この関数は旧いので、**unix:getrusage** の使用を推奨する。

unix:runtime [関数]
プロセスのシステムとユーザー時間の合計を返す。単位は、1/60 秒である。

unix:localtime [関数]
現在の時間と月日を整数ベクトルで返す。要素は、秒、分、時、日、年、曜日である。

unix:asctime *tm_intvector* [関数]
整数ベクトルで表現されるローカル時間を文字列表現に変換する。(unix:asctime (unix:localtime)) は、現在の実際の時間の文字列表現を返す。

11.2.2 プロセス

unix:getpid [関数]
このプロセスのプロセス ID(16 ビット整数) を返す。

unix:getppid [関数]
親プロセスのプロセス ID を返す。

unix:getpgrp [関数]
このプロセスのグループ ID を返す。

unix:setpgrp *integer* [関数]
新しいプロセスのグループ ID を設定する。

unix:getuid [関数]
このプロセスのユーザー ID を返す。

unix:geteuid [関数]
このプロセスの使用可能なユーザー ID を返す。

unix:getgid [関数]
このプロセスのユーザーグループ ID を返す。

unix:getegid [関数]

このプロセスの使用可能なユーザーグループ ID を返す。

unix:setuid *integer* [関数]
このプロセスの使用可能なユーザー ID を設定する。

unix:setgid *integer* [関数]
このプロセスの使用可能なユーザーグループ ID を設定する。

unix:fork [関数]
他の Euslisp を作成する。サブプロセスに 0 が返され、親プロセスに fork されたプロセスの pid が返される。パイプで接続されたプロセスを作成するためには、11.3 節に書かれている **system:piped-fork** を使用すること。

unix:vfork [関数]
他の Euslisp を fork し、その新しい Euslisp のプロセスが終了するまで親プロセスの実行を一時停止する。

unix:exec *path* [関数]
Euslisp から他のプログラムへ実行を移す。

unix:wait [関数]
サブプロセスの中の 1 つのプロセスの終了を待つ。

unix:exit *code* [関数]
実行を終了し、*code* を終了状態として返す。ゼロは通常の終了を意味する。

unix:getpriority *which who* [関数]
このプロセスが持つ最大の優先順位を返す。*which* は、0(プロセス)、1(プロセスグループ)、2(ユーザー)のうちの 1 つである。

unix:setpriority *which who priority* [関数]
which と *who* で決定されるリソースの優先順位を *priority* に設定する。*which* は、0,1,2 の内の 1 つである。*who* は、*which* から相対的に解釈される ($which = 0$ ならプロセスを示し、$which = 1$ ならプロセスグループを示し、$which = 2$ ならユーザーの ID を示す)。*who* がゼロのとき、現在のプロセス、プロセスグループ、ユーザーを示す。Euslisp プロセスに低い優先順位を指定することは、大きい値を設定することであり、これはプロセスを不利にする。`(unix:setpriority 0 0 10)` は、優先順位を 10 に設定する。

unix:getrusage *who* [関数]
who プロセスについてのシステムリソースの使用情報のリストを返す。要素は、以下のような順番になっている。もっと多くの情報が、**lisp:rusage** より得られる。

```
float ru_utime (sec.) /* user time used */
float ru_stime (sec.) /* system time used */
int  ru_maxrss;       /* maximum resident set size */
int  ru_ixrss;        /* currently 0 */
int  ru_idrss;        /* integral resident set size */
int  ru_isrss;        /* currently 0 */
int  ru_minflt;       /* page faults without physical I/O */
int  ru_majflt;       /* page faults with physical I/O */
int  ru_nswap;        /* number of swaps */
int  ru_inblock;      /* block input operations */
int  ru_oublock;      /* block output operations */
int  ru_msgsnd;       /* messages sent */
int  ru_msgrcv;       /* messages received */
int  ru_nsignals;     /* signals received */
```

```
int  ru_nvcsw;        /* voluntary context switches */
int  ru_nivcsw;       /* involuntary context switches */
```

unix:system *[command]* [関数]
サブシェルで *command* を実行する。*command* は、Bourn-shell で認識されるものでなければならない。

unix:getenv *env-var* [関数]
env-var の環境変数の値を返す。

unix:putenv *env* [関数]
プロセスの環境変数リストに *env* を追加する。*env* は、`"VARIABLE=value"`のように変数と値の等価を表す文字列である。

unix:sleep *time* [関数]
time 秒間このプロセスの実行を一時停止する。

unix:usleep *time* [関数]
time マイクロ秒間このプロセスを一時停止する。(**u** は、マイクロを表現する。) **usleep** は、Solaris2 あるいは他の System5 系のシステムには実現されていない。

11.2.3 ファイルシステムと入出力

unix:uread *stream [buffer] [size]* [関数]
stream から *size* バイト読み込む。*stream* は、ストリームオブジェクトあるいはファイルディスクリプタ (fd) を表現する整数である。もし、*buffer* が与えられるとき、入力はそこに蓄積される。そうでないならば、入力は *stream* のバッファに蓄積される。したがって、もし *stream* が fd なら、*buffer* は与えられなければならない。**unix:uread** は、新しい文字列バッファを配置しない。**unix:uread** は、実際に読み込まれたバイト数を返す。

unix:write *stream string &optional size* [関数]
stream に *string* の *size* バイトを書き込む。もし、*size* が省略されたならば、*string* の全部の長さが出力される。

unix:fcntl *stream command argument* [関数]

unix:ioctl *stream command buffer* [関数]

unix:ioctl_ *stream command1 command2* [関数]

unix:ioctl_R *stream command1 command2 buffer [size]* [関数]

unix:ioctl_W *stream command1 command2 buffer [size]* [関数]

unix:ioctl_WR *stream command1 command2 buffer [size]* [関数]

unix:close *fd* [関数]

fd で指定されるファイルをクローズする。

unix:dup *fd* [関数]
fd で指定されるファイルディスクリプタを 2 重化して返す。

unix:pipe [関数]
パイプを作成する。このパイプの入出力ストリームを返す。

unix:lseek *stream position [whence 0]* [関数]
stream のファイルポインタを *whence* から *position* の位置に設定する。

unix:link *path1 path2* [関数]
hard リンクを作る。

unix:unlink *path* [関数]
path で指定されたファイルの hard リンクを取り去る。もし、ファイルに参照が残っていないなら、削除される。

unix:mknod *path mode* [関数]
ファイルシステムに inode を作る。*path* は、**pathname** オブジェクトでなく文字列でなければならない。

unix:mkdir *path mode* [関数]
ファイルシステムにディレクトリを作る。*path* は、**pathname** オブジェクトでなく文字列でなければならない。

unix:access *path mode* [関数]
path へのアクセス権利をチェックする。

unix:stat *path* [関数]
path の inode 情報を得て、以下に示す整数のリストを返す。

```
st_ctime ; file last status change time
st_mtime ; file last modify time
st_atime ; file last access time
st_size ; total size of file, in bytes
st_gid ; group ID of owne
st_uid ; user ID of owner
st_nlink ; number of hard links to the file
st_rdev ; the device identifier (special files only)
st_dev ; device file resides on
st_ino ; the file serial number
st_mode ; file mode
```

unix:chdir *path* [関数]
現在のディレクトリを *path* に変更する。

unix:getwd [関数]
現在のディレクトリを返す。

unix:chmod *path integer* [関数]
path のアクセスモード (permission) を変更する。

unix:chown *path integer* [関数]
path ファイルのオーナーを変更する。

unix:isatty *(stream | fd)* [関数]

もし、*stream* が TTY 型のデバイス (シリアルポートあるいは仮想 TTY) に接続されているなら T を返す。

unix:msgget *key mode* [関数]
key によってアドレスされるメッセージキューを作成し、配置する。

unix:msgsnd *qid buf [mtype [flag]]* [関数]

unix:msgrcv *qid buf [size [mtype [flag]]]* [関数]

unix:socket *domain type &optional proto* [関数]
domain に定義されている名前を持ち *type* を抽象型とするソケットを作成する。*type* は、1 (SOCK_STREAM), 2 (SOCK_DGRAM), 3 (SOCK_RAW), 4 (SOCK_RDM), 5 (SOCK_SEQPACKET) の内の 1 つである。

unix:bind *socket name* [関数]
name と *socket* を関連付ける。もし、ソケットが UNIX 領域内で定義されているならば、*name* は、UNIX のパス名でなければならない。

unix:connect *socket addr* [関数]
socket と *addr* で指定される他のソケットを接続する。

unix:listen *socket &optional backlog* [関数]
socket から接続要求を受け始める。*backlog* は、接続の確定を待つためのキューの長さを指定する。

unix:accept *socket* [関数]
socket から接続要求を受け、両方向にメッセージを交換できるファイルディスクリプタを返す。

unix:recvfrom *socket &optional mesg from flag* [関数]
socket からデータを書いたメッセージを受ける。そのソケットは、**unix:bind** により名前を割り当てられなければならない。*mesg* は、入ってきたメッセージが蓄積されるための文字列である。もし、*mesg* が与えられたならば、**unix:recvfrom** は受け取ったバイト数を返す。もし省略されたなら、メッセージを蓄積するための新しい文字列を作成し、その文字列を返す。

unix:sendto *socket addr mesg &optional len flag* [関数]
addr によって指定される他のソケットへデータの書かれたメッセージを送る。*socket* は、名前が割り当てられてないデータ書き込み型のソケットでなければならない。*mesg* は、送るための文字列であり、*len* は文字列の最初から数えたメッセージの長さである。もし、省略されたなら、すべての文字列を送る。

unix:getservbyname *servicename* [関数]
`/etc/services` あるいは NIS データベースに記録されている *servicename* のサービス番号を返す。

unix:gethostbyname *hostname* [関数]
hostname の ip アドレスとアドレス型のリストを返す。(一般にいつも AF_INET==2).

unix:syserrlist *errno* [関数]
errno のエラーコードに対するエラー情報が記述された文字列を返す。

11.2.4 シグナル

unix:signal *signal func &optional option* [関数]

signal に対してシグナルハンドラー *func* をインストールする。BSD4.2 システムにおいて、システムコール処理の間に捕まえたシグナルは、システムコールのリトライに起因する。これは、もしその処理がシステムコールの呼び出しを発行するならば、シグナルを無視することを意味する。もし、*option=2* が指定されたならば、シグナルは system-5 の手法で処理される。そのシグナルは、システムコールの失敗に起因する。

unix:kill *pid signal* [関数]

pid で名前付けされるプロセスに *signal* を送る。

unix:pause [関数]

シグナルが到着するまでこのプロセスの実行を一時停止する。

unix:alarm *time* [関数]

time 秒後にアラーム時計シグナル (SIGALRM 14) を送る。*time*=0 で **unix:alarm** を呼び出すと、アラーム時計をリセットする。

unix:ualarm *time* [関数]

time がマイクロ秒単位であることを除いて **unix:alarm** と同じである。**ualarm** は、Solaris2 あるいは System5 系のシステムに実現されていない。

unix:getitimer *timer* [関数]

timer は、0 (`ITIMER_REAL`), 1 (`ITIMER_VIRTUAL`), 2(`ITIMER_PROF`) の内の 1 つである。秒単位の時間 (timer) と間隔 (interval) の 2 つの要素を持つリストを返す。

unix:setitimer *timer value interval* [関数]

timer に *value* と *interval* を設定する。*timer* は、0 (`ITIMER_REAL`), 1 (`ITIMER_VIRTUAL`), 2(`ITIMER_PROF`) の内の 1 つである。`ITIMER_REAL` は、*value* が終了したとき、SIGALRM を発行する。`ITIMER_VIRTUAL` は、SIGVTALRM を発行し、`ITIMER_PROF` は、SIGPROF を発行する。

unix:select *inlist outlist exceptlist timeout* [関数]

inlist, outlist と *exceptlist* は、ファイルディスクリプタを示すビットベクトルである。そのファイルディスクリプタの入出力イベントは、テストされなければならない。例えば、もし *inlist*=`#b0110` で *outlist*=`#b100` で *exceptlist*=NIL であるなら、fd=1 あるいは fd=2 で読み込み可能かあるいは fd=2 で書き込み可能であるかどうかをテストする。*timeout* は、**unix:select** が待つために許される秒数を指定する。*inlist* で指定されているポートの 1 つに入力データが現れるかあるいは *outlist* に指定されるポートの 1 つに書き込み可能となるかあるいは *exceptlist* で指定されるポートの 1 つに例外条件が起こるとすぐに **unix:select** は、*inlist, outlist, exceptlist* のそれぞれにおいてアクセス可能なポートとして設定されたポートの中で、入力処理可能なポート番号を返す。

unix:select-read-fd *read-fdset timeout* [関数]

入出力の選択は、ふつう入力処理のときのみ意味がある。**unix:select-read-fd** は、`select fdset nil nil timeout` の速記法である。*read-fdset* は、ビットベクトルでなく、設定された読み込み fd を指定する整数である。

11.2.5 マルチスレッド

スレッド内でシグナルを生成することはできない。したがって、1 つのシグナルスタックと 1 つのタイマーが Euslisp のプロセスの中で実現されている。Solaris2 の上では、メインの最上位レベルが分割されたスレッド内で実行する。

unix:thr-self [関数]
現在実行されているスレッドの ID(整数) を返す。

unix:thr-getprio *id* [関数]
id で指定されたスレッドの実行優先順位を返す。

unix:thr-setprio *id newprio* [関数]
id で指定されたスレッドの実行優先順位に *newprio* を設定する。*newprio* が小さい数値の場合、優先順位が高いことを意味する。言い替えれば、*newprio* よりも大きな数値を持つスレッドは、CPU にアクセスされることが少なくなる。ユーザーは、実行優先順位をプロセスの値（普通はゼロ）よりも高くすることはできない。

unix:thr-getconcurrency [関数]
並列に実行できるスレッドの数で表現される並列度 (整数) を返す。

unix:thr-setconcurrency *concurrency* [関数]
concurrency の値は、プロセス内の LWP の数である。もし、*concurrensy* がデフォルトの 1 ならば、生成されたたくさんのスレッドがすべて実行可能であったとしても順番に 1 つの LWP に割り当てられる。もし、プログラムがマルチプロセッサマシン上で実行され、同時に複数の CPU を利用したいならば、*concurrency* に 1 より大きい値を設定しなければならない。*concurrency* に大きな値を設定すると、オペレーティングシステムのリソースをたくさん消費する。普通、*concurrency* はプロセッサの数と同じかあるいは小さくすべきである。

unix:thr-create *func arg-list &optional (size 64*1024)* [関数]
size ワードの lisp スタックを持ち、*size* バイトの C スタック持つ新しいスレッドを作成し、そのスレッド内で *arg-list* に *func* を適用する。スレッドは、どんな結果も呼びだし側に返すことができない。この関数の使用は避けること。

11.2.6 低レベルメモリ管理

unix:malloc *integer* [関数]
Euslisp のメモリ空間の外にメモリを配置する。

unix:free *integer* [関数]
unix:malloc で配置されたメモリブロックを開放する。

unix:valloc *integer* [関数]

unix:mmap *address length protection share stream offset* [関数]

unix:munmap *address length* [関数]

unix:vadvise *integer* [関数]

11.2.7 IOCTL

Unix はターミナルデバイスを `ioctl` の 2 番目の引数に命令を設定することにより制御することができるが、Euslisp はインクルードファイルの参照や命令コードとしての引数の論理和を省略するために、関数で備えている。詳しい内容は、Unix の `termio` のマニュアルを参照すること。

ターミナルの入出力制御には、TIOC*と TC*という 2 つの命令系がある。自分のオペレーティングシステムにこれらの関数が実現されているがどうかについて気を付けなさい。基本的に、BSD 系は TIOC*の入出力をサポートし、System5 系が TC*をサポートしている。

SunOS 4.1 TIOC*と TC*の両方サポート

Solaris2 TC*のみサポート

mips, ultrix? TIOC*のみサポート

unix:tiocgetp *stream [sgttybuf]* [関数]
パラメータを得る。

unix:tiocsetp *stream sgttybuf* [関数]
パラメータを設定する。

unix:tiocsetn *stream [sgttybuf]* [関数]

unix:tiocgetd *stream [sgttybuf]* [関数]

unix:tiocflush *stream* [関数]
バッファをすべて出力する。

unix:tiocgpgrp *stream integer* [関数]
プロセスのグループ ID を得る。

unix:tiocspgrp *stream integer* [関数]
プロセスのグループ ID を設定する。

unix:tiocoutq *stream integer* [関数]

unix:fionread *stream integer* [関数]

unix:tiocsetc *stream buf* [関数]

unix:tioclbis *stream buf* [関数]

unix:tioclbic *stream buf* [関数]

unix:tioclset *stream buf* [関数]

unix:tioclget *stream buf* [関数]

unix:tcseta *stream buffer* [関数]
ターミナルパラメータをすぐに設定する。

unix:tcsets *stream buffer* [関数]
ターミナルパラメータを設定する。

unix:tcsetsw *stream buffer* [関数]
出力として列をなす全ての文字を転送した後、ターミナルパラメータを設定する。

unix:tcsetsf *stream buffer* [関数]
出力として列をなす全ての文字を転送し、入力として列をなす全ての文字列を廃棄した後、ターミナルパラメータを設定する。

unix:tiocsetc *stream buffer* [関数]

unix:tcsetaf *stream buffer* [関数]

unix:tcsetaw *stream buffer* [関数]

unix:tcgeta *stream buffer* [関数]

unix:tcgets *stream buffer* [関数]

unix:tcgetattr *stream buffer* [関数]

unix:tcsetattr *stream buffer* [関数]

11.2.8 キーインデックスファイル

近年 UNIX は、キーインデックスファイルの管理のために **dbm** あるいは **ndbm** ライブラリを提供する。このライブラリを使用することにより、キーとデータの組みで構成されるデータベースを構築することができる。以下に示す関数は、`clib/ndbm.c` に定義されている。Sun において、そのファイルは、`cc -c -Dsun4 -Bstatic` でコンパイルし、`(load "clib/ndbm" :ld-option "-lc")` で Euslisp にロードすること。

dbm-open *dbname mode flag* [関数]

dbm-open は、データベースファイルを作るときと、そのデータベースに読み込み／書き込みをするとき、最初に呼び出されなければならない。*dbname* は、データベースの名前である。実際に、ndbm manager は`".pag"`と`".dir"`の拡張子を持つ 2 つのファイルを作成する。*mode* は、ファイルのオープンモードを指定する。0 は読み込み専用、1 は書き込み専用、2 は読み書き用を示す。また、最初にファイルを作成するときは、`#x200` を論理和すべきである。*flag* は、**chmod** で変更されるアクセス許可を与える。`#o666` あるいは`#o664` が、*flag* に適している。**dbm-open** は、そのデータベースを確認するための整数を返す。この値は、他の dbm 関数によってデータベースを確認するために使用される。言い換えれば、同時に幾つかのデータベースをオープンすることができる。

dbm-store *db key datum mode* [関数]

key と *datum* の組み合わせを *db* に蓄積する。*db* は、データベースを確認するための整数である。*key* と *datum* は文字列である。*mode* は 0(挿入) あるいは 1(置き換え) のどちらかである。

dbm-fetch *db key* [関数]

db の中の *key* に関連付けられているデータを呼び出す。

11.3 UNIX プロセス

Euslisp から UNIX 命令を実行ために **unix:system** 関数を使用すること。**piped-fork** は、標準出力を標準入力がパイプを通して Euslisp の双方向ストリームに接続されるサブプロセスを作成する。**piped-fork** はストリームを返す。以下に示す関数は、"wc"を使用することにより、ファイルに含まれる行の数を数えるものである。

```
(defun count-lines (file) (read (piped-fork "wc" file)))
```

次の例は、他のワークステーション”etlic0”の上に eus プロセスを作成し、分散計算をするためのポートを提供する。

```
(setq ic0eus (piped-fork "rsh" "etlic0" "eus"))
(format ic0eus "(list 1 2 3)~%")
(read ic0eus) --> (1 2 3)
```

ソースコードを修正するために、Euslisp から ez を呼び出すことができる。スクリーンエディター ez は、メッセージキューを通して Euslisp と接続する。もし、既に ez プロセスを Euslisp と並列に実行しているならば、**ez** は ez プロセスを再スタートし、ターミナル制御を得る。ez の中で esc-P あるいは esc-M 命令を発行することにより、テキストは戻され、Euslisp で評価される。ファイルに少しの変更を加えたとき、全部のファイルをロードする必要がないので、デバッグするのにこれは便利である。emacs の上でも M-X run-lisp 命令でおなじことが可能である。

cd *&optional (dir (unix:getenv ”HOME”))* [関数]
現在のディレクトリを変更する。

ez *&optional key* [関数]
ez エディターの画面に入る。それから Lisp 書式を読み込み、評価する。

piped-fork *&optional (exec) &rest args* [関数]
プロセスを fork し、両方向ストリームを Euslisp とサブプロセスの間に作る。

rusage [関数]
このプロセスのリソースの使用状況を表示する。

11.4 Cで書かれたLisp関数の追加

ファイルに含まれるCを重く参照したり、行列にしばしばアクセスするようなプログラムにおいては、Euslispで記述するよりはむしろCあるいは他の言語で記述した方が効率が良く、記述もわかり易い。EusLispは、Cで書かれたプログラムをリンクする方法を備えている。

もしCで書かれたEuslispの関数を定義したいならば、Euslispで呼び出しできるCの関数はそれぞれ、3つの引き数を受けるように書かれなければならない。環境へのポインタと受け取る引き数の数とlispの引数領域を示すポインタの3つである。これらの引数は、`c/eus.h`の中のマクロによって参照されるため、`ctx, n, argv`と名付けられねばならない。Cのプログラムは、`*eusdir*/c/eus.h`をincludeしなければならない。プログラマーは、そこに書かれた型やマクロに精通していなければならない。エントリ関数名には、ソースファイルのbasenameを付けなければならない。

任意の数の実数の算術平均を計算するCの関数AVERAGEのサンプルコードは、以下に示す通りである。この例において、引数から実数値を得る方法、実数のポインタを作る方法、特殊変数AVERAGEにポインタを設定する方法やエントリ関数`ave`に関数やsymbolを定義する方法を知ることができる。`'cc -c -Dsun4 -DSolaris2 -K pic'`でこのプログラムをコンパイルする。`c/eus.h`内の正しい定義を選択するために、`-Dsun4`や`-DSolaris2`が必要である。`-K pic`は、ロード可能な共有オブジェクトのために、位置に依存するコードをCコンパイラで生成させるために必要である。その後、コンパイルの結果である'.o'ファイルがEuslispにロードすることができる。もっと完全な例は`*eusdir*/clib/*.c`に置かれている。これらの例は、ここで書かれた方法で定義され、ロードされる。

```
/* ave.c */
/* (average &rest numbers) */
#include "/usr/local/eus/c/eus.h"
static pointer AVESYM;
pointer AVERAGE(ctx,n,argv)
context *ctx;
int n;
pointer argv[];
{ register int i;
  float sum=0.0, a, av;
  pointer result;
  numunion nu;
  for (i=0; i<n; i++) {
    a=ckfltval(argv[i]);
    sum += a;} /*get floating value from args*/
  av=sum/n;
  result=makeflt(av);
  AVESYM->c.sym.speval=result;  /*kindly set the result in symbol*/
  return(result);}

ave(ctx,n,argv)
context *ctx;
int n;
pointer argv[];
{ char *p;
  p="AVERAGE";
  defun(ctx,p,argv[0],AVERAGE);
```

```
  AVESYM=intern(ctx,p,strlen(p),userpkg); /* make a new symbol*/
  }
```

11.5 他言語インターフェース

Euslispとのリンクを考慮していないCの関数もEuslispにロードすることができる。これらの関数は、他言語関数と呼ばれる。そのようなプログラムは**load-foreign**マクロによりロードされる。そのマクロは、**foreign-module**のインスタンスを返す。オブジェクトファイルの中の外部symbol定義は、モジュールオブジェクトの中に登録されている。**defforeign**は、Euslispから呼び出すためのC関数に対するエントリーを作るために使用される。**defun-c-callable**は、Cから呼び出し可能なlisp関数を定義する。呼び出し可能なCの関数は、パラメータを変換し関連するEuslispの関数へ制御を移すために*pod-code*と呼ばれる特別なコードを持つ。**pod-address**は、この特別なコードのアドレスを返す。そのアドレスはCの関数に通知されるべきである。

これは、Cのプログラムのサンプルと Euslispへの関数インターフェースである。

```
/* C program named cfunc.c*/

static int (*g)(); /* variable to store Lisp function entry */

double sync(x)
double x;
{ extern double sin();
  return(sin(x)/x);}

char *upperstring(s)
char *s;
{ char *ss=s;
  while (*s) { if (islower(*s)) *s=toupper(*s); s++;}
  return(ss);}

int setlfunc(f)      /* remember the argument in g just to see */
int (*f)();          /* how Lisp function can be called from C */
{ g=f;}

int callfunc(x)      /* apply the Lisp function saved in g to the arg.*/
int x;
{ return((*g)(x));}

;;;; Example program for EusLisp's foreign language interface
;;;; make foreign-module
(setq m (load-foreign "cfunc.o"))

;; define foreign functions so that they can be callable from lisp
(defforeign sync m "sync" (:float) :float)
(defforeign toupper m "upperstring" (:string) :string)
(defforeign setlfunc m "setlfunc" (:integer) :integer)
(defforeign callfunc m "callfunc" (:integer) :integer)
```

```
;; call them
(sync 1.0) --> 0.841471
(print (toupper "abc123"))  --> "ABC123"

;; define a test function which is callable from C.
(defun-c-callable TEST ((a :integer)) :integer
      (format t "TEST is called, arg=~s~%" a)
      (* a a))    ;; return the square of the arg
;;  call it from C
;;setlfunc remembers the entry address of Lisp TEST function.
(setlfunc (pod-address (intern "TEST")))
(callfunc 12)  --> TEST is called, arg=12  144
```

Euslisp のデータ表現は、以下に示す方法で C のデータ表現に変換される。EusLisp の 30 ビット整数（文字列を含む）は、符号拡張され、スタックを通して C の関数に渡される。30 ビット実数は、倍精度実数 (double) に拡張され、スタックを通して渡される。文字列と整数ベクトルと実数ベクトルについては、その最初の要素のアドレスのみがスタックに渡され、行列自体はコピーされない。Euslisp には、2 次元以上の配列を渡す方法がない。2 次元以上の配列はすべての要素を線形に保持する 1 次元ベクトルを持つ。このベクトルは、**array-entity** マクロにより得られる。もし、2 次元行列を FORTRAN のサブルーチンに送る場合、FORTRAN において列と行が反対となっているためその行列を転置しなければならないことに注意すること。

実数の Euslisp 表現は、いつも単精度であるので、倍精度の実数のベクトルに渡すとき変換を要する。変換関数、**double2float** と **float2double** は、この目的で clib/double.c の中に定義されている。例えば、もし 3x3 の実数行列があり、CF のいう名の C の関数にそれを倍精度実数の行列として渡したいなら、以下のように使用すればよい。

```
(setq mat (make-matrix 3 3))
(CF (float2double (array-entity mat)))
```

C の構造体は、**defstruct** マクロにより定義することができる。**defstruct** は、次のようなフィールド定義書式により struct-name を受け取る。

```
(defcstruct <struct-name>
   {(<field> <type> [*] [size])}*)
```

たとえば、以下に示す構造体の定義は、つぎの **defstruct** によって表現される。

```
/* C definition */
struct example {
   char  a[2];
   short b;
   long  *c;
   float *d[2];};

/* equivalent EusLisp definition */
(defcstruct example
   (a :char 2)
   (b :short)
   (c :long *)
```

```
    (d :float * 2))
```

load-foreign *objfile &key symbol-input symbol-output (symbol-file objfile) ld-option)* [マクロ]

Euslisp 以外の言語で書かれたオブジェクトモジュールをロードする。Solaris2 において、**load-foreign** は*:entry* パラメータに null 文字列を与えた **load** を呼び出す。コンパイルされたコードのオブジェクトが返される。この結果は、後に **defforeign** を呼び出すことによってモジュールの中の関数のエントリーを作ることが必要である。ライブラリーは *ld-option* に指定することができる。しかしながら、ライブラリの中に定義された symbol はデフォルトの symbol-output ファイルで獲得することができない。ライブラリで定義された関数の呼び出しを Euslisp に許可するために、*symbol-output* と *symbol-file* が明示的に与えられなければならない。(もし、*objfile* からのみそれらを参照するならば、これらの引き数は必要ない。) **load-foreign** は、指定されたライブラリとグローバル変数と一緒に *objfile* を Euslisp のコアにリンクし、リンクされたオブジェクトを *symbol-output* に書き込む。それから、*symbol-file* の中の symbol は、検索され、他言語モジュールの中にリストアップされる。*symbol-file* のデフォルトが *objfile* であるので、もし *symbol-file* が与えられないなら、*objfile* に定義されている symbol のみ認識される。*objfile* とライブラリの両方のグローバルエントリーをすべて見るために、**load-foreign** の最初のプロセスリンクの結果であるリンクされた (マージされた)symbol テーブルは確かめられなければならない。このような理由で、*symbol-output* と *symbol-file* の両方に同一のファイル名を与えなければならない。

以下に示されるように、中間の symbol ファイルは **unix:unlink** によって削除することができる。しかしながら、もし同じライブラリを参照する 2 つ以上の他言語モジュールをロードするとき、ライブラリの 2 重化を避けたいなら、*symbol-output* 引き数を使用しなければならない。上記の例として、"linpack.a" のすべての関数をロードしており、次に"linpack.a"の関数を呼び出している他のファイル"linapp.o"を呼びだそうとしていると仮定する。次の **load-foreign** 呼び出しは、"euslinpack"を unlink する前に発行しなければならない (`load-foreign "linapp.o" :symbol-input "euslinpack"`)。**load-foreign** と **defforeign** のもっと完全な例は、`*eusdir*/llib/linpack.l` で見ることができる。

```
(setq linpack-module
    (load-foreign "/usr/local/eus/clib/linpackref.o"
        :ld-option  "-L/usr/local/lib -llinpack -lF77 -lm -lc"
        :symbol-output "euslinpack"
        :symbol-file "euslinpack"
        ))
(unix:unlink "euslinpack")
```

defforeign *funcname module cname paramspec resulttype* [マクロ]

他言語モジュールの中の関数エントリーを作る。*funcname* は、Euslisp に作られる symbol である。*module* は、**load-foreign** によって返されるコンパイルされたコードのオブジェクトである。*cname* は、他言語プログラムの中で定義されている C の関数の名前である。その名前は"_myfunc"のような文字列である。*paramspec* は、パラメータの型指定のリストである。それは、Euslisp から C の関数に引き数を渡すときに、データの型変換と強制 (coercion) を行うために使用される。データ変換がなかったり、あるいは型チェックが必要ないとき、*paramspec* は NIL で構わない。`:integer, :float, :string, (:string n)` の内の 1 つが *resulttype* に与えられなければならない。`:integer` は、C の関数が char,short,int(long) のいずれかを返すことを意味する。`:float` は、返す値が float あるいは double のときに指定する。`:string` は、C 関数が string へのポインターを返すことを意味し、Euslisp は Euslisp の文字列に変更するために string に long-word のヘッダーを追加する。文字列の長さは **strlen** によって見つけられる。string の直前に書き込まれるヘッダーは、悲惨な結果を引き起こすことがあることに注意。もう一方で、`(:string n)` は、安全だが遅い。なぜなら、**n** の長さを持つ Euslisp の文字列が新しく作成され、C の文字列の内容がそこにコピーされるからである。`(:string 4)` は、整数へのポインターを返す C の関数に使用で

きる。FORTRAN ユーザーは、FORTRAN の関数あるいはサブルーチンのあらゆる引き数は、call-by-refference によって渡されることに注意すべきである。したがって、1 つの整数あるいは実数型の引き数でさえ FORTRAN へ渡される前に整数ベクトルあるいは実数ベクトルに置かれなければならない。

defun-c-callable *funcname paramspec resulttype . body* [マクロ]

他言語のコードから呼び出すことができる Euslisp の関数を定義する。*funcname* は、Euslisp の関数として定義されている symbol である。*paramspec* は、**defforeign** の中の型指定のリストである。**defforeign** の paramspec と違い、**defun-c-callable** の paramspec は、関数が引き数を全く受け取らない場合以外、省略することができない。`:integer` は、int,char,short のすべての型に使用すべきである。`:float` は、float と double に使用する。*resulttype* は、Lisp 関数の型である。*resulttype* は、型のチェックあるいは整数から実数に型の強制を必要とする場合を除いて、省略することができる。*body* は、この関数が C から呼び出されるとき、実行される lisp 表現である。**defun-c-callable** で定義されている関数は、Lisp 表現からでも呼び出すことができる。**defun-c-callable** は *funcname* を返す。その返り値は、symbol のようであるが、そうではなく、symbol のサブクラスである **foreign-pod** のインスタンスである。

pod-address *funcname* [関数]

defun-c-callable で定義された C で呼び出し可能な Lisp 関数 *funcname* における他言語と Euslisp とのインターフェースコードのアドレスを返す。これは、他言語プログラムに Lisp 関数の位置を知らせるために使用される。

array-entity *array-of-more-than-one-dimension* [マクロ]

多次元配列の要素を保持する 1 次元ベクトルを返す。これは、多次元あるいは一般の配列を他言語に渡すために必要である。しかし、1 次元のベクトルは直接渡すことができる。

float2double *float-vector [doublevector]* [関数]

`float-vector` を倍精度実数の表現に変換する。その結果は、`float-vector` であるが、最初の引き数の長さの 2 倍になっている。

double2float *doublevector [float-vector]* [関数]

倍精度実数表現が単精度の `float-vector` に変換される。

11.6 VxWorks

ホストと VxWorks との通信機能が”vxworks/vxweus.l”ファイルで提供されている。VxWorks 上に **vxwserv** サーバを常駐させることにより、ホスト上の EusLisp から **vxwserv** にコネクションを張り、vxws プロトコルに従ったコマンドを送ることにより、VxWorks の関数を起動し、引数を送り、結果を受け取ることができる。

VxWorks のソフトは Sun の c コンパイラによって開発することができる上、データ表現が sun3, sun4, VME147 の間で共通であることを利用して、vxws プロトコルは、バイナリモードで動作することができる。

11.6.1 VxWorks 側の起動

VxWorks にログインし、”*eusdir*/vxworks/vxwserv.o”をロードする。その後、**vxwserv** タスクを spawn する。**vxwserv** は VxWorks 上の 2200 番ポートを listen する。2200 が塞がっている場合、2201, 2202, ... を試す。正しく bind されたポート番号が表示される。

```
% rlogin asvx0 （あるいは etlic2 上であれば、% tip asvx[01] も可能）
-> cd "atom:/usr/share/src/eus/vxworks"
-> ld <vxwserv.o
-> sp vxwserv
port 2200 is bound.
```

VxWorks の i コマンドで、**vxwserv** タスクが常駐したことを確かめる。同じ要領で、eus から呼び出したい VxWorks のプログラムを VxWorks 上にロードする。その後、Euslisp と VxWorks とのコネクションが張られると、**vxwserv** を走らせた TTY に、次のようなメッセージが出力される。

```
CLIENT accepted: sock=9 port = 1129: family = 2: addr = c01fcc10:
VxWserv started with 16394 byte buffer
```

11.6.2 ホスト側の起動

任意のマシンの上で eus を起動し、”vxworks/vxweus”をロードする。**connect-vxw** 関数を用いて **vxwserv** に接続する。接続後、ソケットストリームが***vxw-stream***にバインドされる。以下に、コネクトの例を示す。この例では、VxWorks 上の sin, vadd 関数を euslisp の関数 VSIN,VAD として定義している。

```
(load "vxworks/vxweus")
(setq s (connect-vxw :host "asvx0" :port 2200 :buffer-size 1024))
(defvxw VSIN "_sin" (theta) :float)
(defvxw VAD "_vadd" (v1 v2) (float-vector 3))
```

VxWorks 上に作成される関数が、vxws を通じて呼び出されるためには、次の条件を満たさなければならない。

1. 引数は、32 個以内であること、引数に受け取るベクタの容量の合計が connect-vxw の:buffer-size で指定した値を越えないこと
2. struct を引数にしないこと、必ず struct へのポインタを引数にすること
3. 結果は、int, float, double または、それらの配列のアドレスであること
4. 配列のアドレスを結果とする場合、その配列の実体は、関数の外部に取られていること

connect-vxw *&key (:host "asvx0") (:port 2200) (:buffer-size 16384) (:priority 1280) (:option #x1c)* [関数]

:host に対して vxws プロトコルによる通信のためのソケットストリームを作成し、そのストリームを返す。*:host* には、ネットワークにおける VxWorks のアクセス番号あるいはアクセス名を指定する。*:port* には、VxWorks 上の **vxwserv** がバインドしたポートを捜すための最初のポート番号を指定する。このポート番号から、増加方向に接続を試行する。*:option* のコードについては、VxWorks の、spawn 関数を参照のこと。コネクションは、同時に複数張ってよい。

vxw *vxw-stream entry result-type args* [関数]

vxw は、*vxw-stream* に接続されている VxWorks の関数 *entry* を呼び出し、その関数に引き数 *args* を与えて *result-type* で指定された結果を得る。*vxw-stream* には、**connect-vxw** で作成したソケットストリームを与える。*entry* には、VxWorks の関数名をストリングで指定するか、あるいは関数のアドレスを整数で指定する。関数のアドレスを知るには、VxWorks の findsymbol を呼び出す。知りたいシンボルは、通常、"_"で始まることに注意。*entry* がストリングの場合、VxWorks 上でシンボルテーブルの逐次探索が行われる。*result-type* には、結果のデータ型 (`:integer` または`:float`)、あるいはデータを受け取るベクタ型を指定する。ベクタは、`float-vector, integer-vector, string` のインスタンスである。`general vector`(lisp の任意のオブジェクトを要素とするベクタ) は指定できない。結果型は、必ず、実際の VxWorks 関数の結果型と一致しなければならない。*args* には、*entry* に与える引き数を指定する。引数に許される EusLisp データは、`integer, float, string, integer-vector, float-vector, integer-matrix, float-matrix` である。ポインタを含んだ一般のオブジェクト、一般のベクトルは送れない。また、送られたベクトルデータは、一旦 **vxwserv** が獲得したバッファの中に蓄積される。例えば、VxWorks に定義された関数"sin"を呼び出すためには、次のように実行すればよい。`(vxw *vxw-stream* "sin" :float 1.0)`

defvxw *eus-func-name entry args &optional (result-type :integer)* [マクロ]

defvxw は、findsymbol を用いて **vxw** を呼び出して、VxWorks の関数の高速な呼び出しを実現するためのマクロである。VxWorks の関数 *entry* を呼び出すための Euslisp の関数 *eus-func-name* を定義する。このマクロを実行後は、*eus-func-name* を呼び出すことにより、VxWorks の関数を呼び出すことができる。このとき、呼び出しに使用されるソケットストリームは***vxw-stream***に固定されている。ただし、VxWorks 側で、関数をコンパイルし直して再ロードした場合、新しい関数定義が呼ばれるようにするためには、eus 側で、**defvxw** をもう一度実行し直して、最新のエントリアドレスが指定されるようにする必要がある。

12 マルチスレッド

マルチスレッドは、Solaris オペレーティングシステム上の並列プログラミングや非同期プログラミングの機能である。非同期プログラミングは、プログラムの状態と無関係に発生する様々なセンサを経由した外部イベントに応答するためのプログラムに要求される。並列プログラミングは、画像処理や経路計画の干渉チェックのようなプロセス向きの計算の効率を改善する場合に効果的である。

12.1 マルチスレッド Euslisp の設計

12.1.1 Solaris 2 オペレーティングシステムのマルチスレッド

マルチスレッド Euslisp(MT-Eus) は、複数のプロセッサを持った Solaris 2 オペレーティングシステム上で動作する。Solaris のスレッドは、 共有メモリと異なった環境を持つような従来の UNIX プロセスを CPU に配置するためのユニットである。Solaris OS によって提供されるスレッドのライブラリは、それぞれのスレッドを単一の LWP(light weight process) に配置する。このプロセスがカーネルのリソースである。UNIX のカーネルは、それぞれのスレッドに割り当てられたスレッドの優先権に基づいて複数の物理 CPU に LWP の配置を計画する。図 5 は、スレッドと LWP と CPU の関係を表わしたものである。Euslisp の環境およびメモリ管理の設計について、マルチスレッドの能力を引き出すために 2 つの大きな変更がされた。

12.1.2 Context Separation

MT-Eus は、それぞれのスレッドに対し個別にスタックと環境を配置する。そのため、他のスレッドと独立に実行することができる。symbol や cons のようなオブジェクトは、これまでの Euslisp のように共有ヒープメモリ上に配置される。したがって、block label や catch tag やローカル変数のようなスレッドの個別データは、他のスレッドから保護される。ところが、グローバル変数によって示される値（オブジェクト）は、情報の変更が許可されているすべてのスレッドから見ることができる。

環境は C-stack と binding-stack および `lambda`, `block`, `catch`, `let`, `flet` などのローカルブロックに繋がるフレームポインタにより構成されており、新しいスレッドが作成されたときに設置される。複数の環境が本当のマルチプロセッサマシンの上で同時に動作できるので、グローバル変数の中の現在の環境に対して単一なポインタを持つことができない。むしろ、環境のポインタを最上位の評価から低レベルのメモリ管理に変換するために、すべての内部関数にもう一つ引き数を付け加えなければならない。

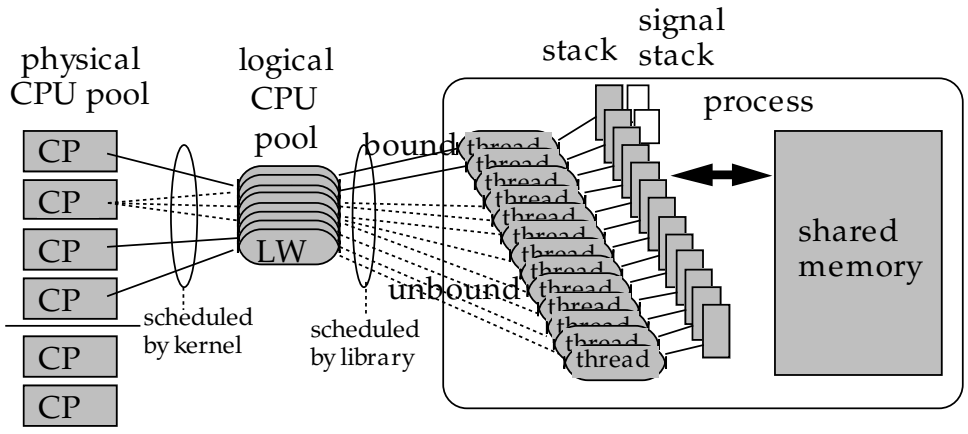

図 5: Solaris オペレーティングシステムのスレッドモデル

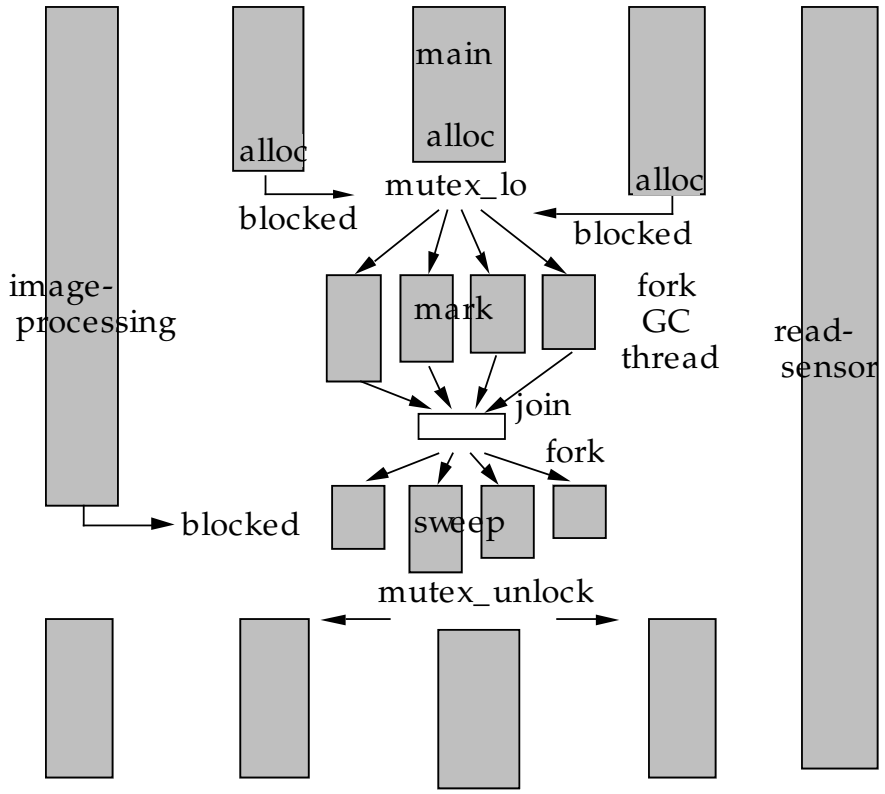

図 6: 並列スレッドのメモリ要求とガーベージコレクトに並列実行

12.1.3 メモリ管理

EusLisp は、すべての型のオブジェクトに対して単一のヒープの中でフィボナッチバディを用いたメモリ管理方法を採用している。異なったメモリ要求を持つプログラムを実行した後、フィボナッチバディが様々な大きさのオブジェクトを等しく高速に配置することができ、コピーなしに素早くガーベージコレクトができ、かつ、高いメモリ利用率（内部損失が 10〜15%で外部損失は無視できる）を示すことを確信した。マルチスレッドのためには、2 つ目のポイントすなわちコピーなしのガーベージコレクトが重要である。もし、オブジェクトのアドレスがガーベージコレクトのコピーにより変化したならば、すべてのスレッド環境のスタックおよび CPU のレジスタ内のポインタを新しいアドレスに書き換えなければならない。しかし、この書き換えは不可能かあるいは大変困難である。

すべてのメモリ配置要求は、低レベルの `alloc` 関数によって処理される。`alloc` は、mutex-locking をする。なぜなら、大きさを持たないリストのグローバルデータベースを扱うからである。ガーベージコレクトの始まる時期およびどのスレッドによってガーベージコレクトが生じるのかを予言できないので、すべてのスレッドは突発的に起こるガーベージコレクトのために準備をしておかなければならない。生きているオブジェクトへのすべてのポインタは、ゴミとして掃除されないように保護するためいつでもガーベージコレクトからアクセスできるよう調整されなければならない。これは、スタックの上に保存されていることを信用する代わりに、それぞれの環境の固定されたスロットの中に極最近に配置されたオブジェクトに対するポインタを蓄積することによって達成される。

図 6 は、スレッドのメモリ要求と fork されたガーベージコレクト内部での marking および sweeping を並列に行っている流れ図を示したものである。メモリ要求およびポインタの処理を行わないスレッドはガーベージコレクトと並列に実行することができ、信号処理や画像獲得のような低レベルのタスクの実時間応答を改善することに注意すること。

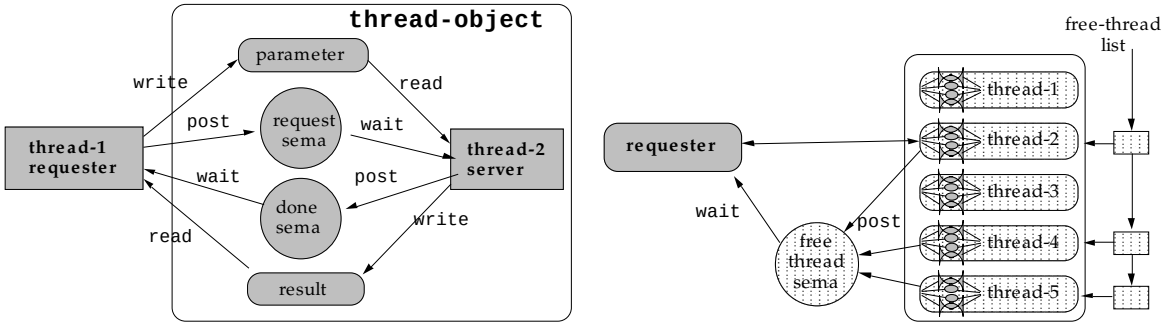

図 7: スレッド間で制御やデータを受け渡すためのスレッドオブジェクト（左）とスレッドプール内に置かれたスレッドの集まり（右）

12.2 非同期プログラミングと並列プログラミングの構築

12.2.1 スレッド作成とスレッドプール

Solaris において複数のプロセッサ上で並列にプログラムを実行するためには、そのプログラムは関数の集まりとして書かれる必要がある。その関数はそれぞれプロセスの中で動的に作成されるスレッドによって実行される。スレッドを作成するために要求される時間は、プロセスを作成するよりも速くなければならないが、スタックを配置しスタックのオーバーフローを発見するためのページ属性を設定した後にスレッドが動き始めるまでに Euslisp において数ミリ秒かかる。この遅れは関数実施と比較して我慢できないため、評価時間におけるシステムコールに対する所要時間を排除する目的で、あらかじめ `make-thread` 関数により十分な数のスレッドが作られ、システムのスレッドプールに置かれる。スレッドプールの中のそれぞれのスレッドは、図 7 で示されるようにスレッド ID と同期のためのセマフォと引き数や評価結果を転送するためのスロットから構成されるスレッドオブジェクトにより表現される。

12.2.2 スレッドの並列実行

スレッドによる並列実行の配置のために、スレッド関数が使用される。スレッドは、スレッドプールから 1 つの空きスレッドを取り、共有メモリを経由して引き数を渡し、図 7 に示されるようなセマフォ信号によりスレッドを立ち上げ、停止することなしに呼出側へスレッドオブジェクトを返す。立ち上げられたスレッドは、呼び出したスレッドと並列に実行され、引き数を評価し始める。呼出側は、fork されたスレッドから評価結果を受けとるために `wait-thread` を使用する。`plist` マクロは、引き数の並列評価を記述するために大変便利な書式である。`plist` は、それぞれの引き数を評価するためにスレッドを割り当て、すべてのスレッドが評価し終わるのを待って結果をリストアップする。

12.2.3 同期の手法

MT-Eus は、**mutex lock, condition variable, セマフォ**と呼ばれる 3 種類の同期手法を持っている。mutex lock は、スレッド間の共有変数の連続アクセスのために使用される。condition variable は、ロックの仮開放あるいは再獲得によって mutex-lock された部分の条件が true になることを待つことをスレッドに許可する。セマフォは、イベントの発生を通知するためあるいはリソースの分割を制御するために使用される。Solaris のカーネルが時間分割スケージューリングを基本として何気なしにタスク切り替えを発生するのと異なり、これらの同期手法は、任意の環境切り替えを引き起こす。

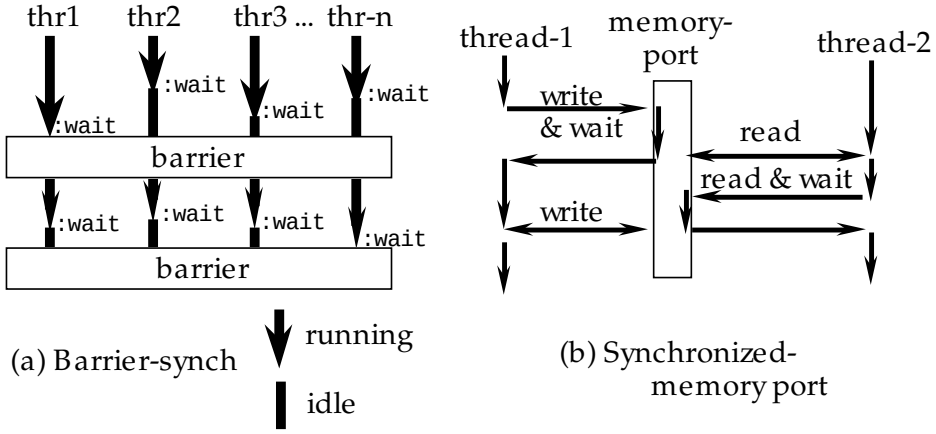

図 8: 同期障壁と同期メモリポート

12.2.4 同期障壁

barrier-synch は、複数のスレッドを同時に同期させるための機構である (図 8)。この目的において、**barrier** クラスのインスタンスが作成され、同期に関係するスレッドがオブジェクトに登録される。その後、それぞれのスレッドは barrier オブジェクトに`:wait` メッセージを送り、そのスレッドは停止する。オブジェクトに登録された最後のスレッドが`:wait` メッセージを送ったとき、待ちが解除され、すべての待ちスレッドが T の返り値を得る。barrier-sync は、マルチロボットシミュレーションのグローバルタイムという重要な役割を演じている。

12.2.5 同期メモリポート

同期メモリポート (synch-memory-port) は、スレッド間でデータを交換するための 1 種のストリームである (図 8)。プロセス内のすべてのスレッドはヒープメモリを共有しているので、もし 1 つのスレッドがグローバル変数にオブジェクトを割り当てた場合、直ちに他のスレッドから見れるようになる。しかしながら、共有メモリはグローバルデータが更新されたというイベントを送るための能力が不足している。同期メモリポートは、共有オブジェクトをアクセスするためのこの同期機能を保証する。同期メモリポートオブジェクトは、1 つのバッファスロットと同期読み書きのために使用される 2 つのセマフォによって構成されている。

12.2.6 タイマー

実時間プログラムは、予定された時間に実行される関数や、特定の間隔で繰り返される関数をしばしば要求する。これまでの EusLisp は、Unix のインターバルタイマーによって定期的に生成される信号によって発生するユーザー関数を実行することができた。MT-Eus において、この実行はデッドロックを引き起こす。なぜなら、割り込みが mutex-lock されたブロック内から発生する可能性がある。したがって、制御は `eval` の最初のように安全な場所で渡されなければならない。上記の同期によって引き起こされる遅れを避けるために、MT-Eus はセマフォを経由して信号通知 (signal-notification) も提供する。言い換えれば、信号関数は呼び出されるかあるいは信号の到着を知らせる関数あるいはセマフォのどちらかをとる。セマフォは、低レベルで告示されるので、同期により隠れているところは最小である。

以下に示すものは、マルチスレッド機能を用いた画像処理のプログラム例である。画像入力スレッドとフィルタースレッドが生成される。samp-image は、33msec 毎に通知される samp-sem を待つことにより、定期的

表 3: マルチプロセッサ上で実行されたプログラムの並列度

processors	1	2	4	8	GC (ratio)
(a) compiled Fibonacci	1.0	2.0	4.0	7.8	0
(b) interpreted Fibonacci	1.0	1.7	2.7	4.4	0
(c) copy-seq	1.0	1.3	0.76	0.71	0.15
(d) make-cube	1.0	0.91	0.40	0.39	0.15
(e) interference-check	1.0	0.88	0.55	0.34	0.21

に画像データをとる。2 つのスレッドは thread-port の読み書きを通じて同期する。filter-image は、フィルターの並列計算のために複数のスレッドを使用している。

```
(make-threads 8)
(defun samp-image (p)
   (let ((samp-sem (make-semaphore)))
        (periodic-sema-post 0.03 samp-sem)
        (loop (sema-wait samp-sem)
              (send p :write (read-image))))
(defun filter-image (p)
  (let (img)
       (loop (setf img (send p :read))
             (plist (filter-up-half img)
                    (filter-low-half img)))))
(setf port (make-thread-port))
(setf sampler (thread #'samp-image port))
(setf filter (thread #'filter-image port))
```

12.3 並列度の計測

表 3 は、32CPU で構成される Cray Superserver の上で測定した並列実行効率を示したものである。コンパイルされたフィボナッチ関数において線形な並列度が得られた。なぜなら、共有メモリへのアクセスがなく、それぞれのプロセッサのキャッシュメモリに十分ロードできるほどちいさなプログラムであったためである。それに反して、同じプログラムをインタープリターで実行したとき、キャッシュメモリを使い果たしたため、線形な高効率を達成することができなかった。さらにまた、頻繁に共有メモリを参照するようなプログラムやメモリ配置を要求するようなプログラムは 1 個のプロセッサで実行したときよりも良い性能を得ることができなかった。これは、頻繁なキャッシュメモリの入れ替えが原因と考えられる。

12.4 スレッド生成

スレッドは、計算を割り当てる単位であり、普通 lisp 書式を評価するための単位である。Euslisp のスレッドは、**thread** クラスのインスタンスによって表現される。このオブジェクトは、内容を表現するスレッド全体というよりはむしろ、実際に引き数と結果を渡すためのスレッドの制御ポートであり、評価を始めるものである。

sys:make-thread *num &optional (lsize 32*1024) (csize lsize)* [関数]

lsize ワードの lisp スタックと *csize* ワードの C-スタックを持つスレッドを *num* 個だけ生成し、システムのスレッドプールに置く。スレッドプール内のすべてのスレッドは、sys:*threads*に束ねてあり、**make-thread** が呼び出されたときに拡張される。**thread** 関数によって、計算はスレッドプールの中で空いたスレッドの１つに割り当てられる。したがって、指定された計算がどのスレッドに割り当てられるか制御できないため、スレッドのスタックサイズを変更する方法が無い。

sys:*threads* [変数]

make-thread によって作成されたすべてのスレッドのリストを持つ。

sys::free-threads [関数]

スレッドプール内の空いたスレッドのリストを返す。もし、結果が NIL ならば、スレッドへのタスクの新しい付託は現在実行されているスレッドのどれかの評価が終了するかあるいは **make-thread** によってスレッドプールに新しいスレッドを生成するまで停止される。

sys:thread *func &rest args* [関数]

スレッドプールから空いたスレッドを 1 つ取り出し、*(func . args)* の評価のためにそれを割り当てる。**sys:thread** は、*args* を展開したリストに func を適用するが、関数の適用結果を受け取らないため、非同期の **funcall** とみなすことができる。むしろ、**sys:thread** は funcall に割り当てられたスレッドオブジェクトを返すので、実際の結果は **sys:wait-thread** によって後から得ることができる。

```
(defun compute-pi (digits) ...)
(setq trd (sys:thread \#'compute-pi 1000)) ;assign compute-pi to a thread
...  ;; other computation
(sys:wait-thread trd) ;get the result of (compute-pi 1000)
```

sys:thread-no-wait *func &rest args* [関数]

空いたスレッドの 1 つに計算を割り当てる。スレッドは、**wait-thread** されることなしに、評価が終了したとき、スレッドプールに戻される。

sys:wait-thread *thread* [関数]

thread に **sys:thread** 関数によって与えられた funcall の評価が終了するのを待ち、その結果を受け取り、返す。もし、スレッドに **sys:thread** によって評価が割り当てられたならば、**sys:wait-thread** は、必須である。なぜなら、スレッドは結果を転送し終わるまでスレッドプールに戻らないためである。

sys:plist *&rest forms* [マクロ]

異なったスレッドにより並列に *forms* を評価し、すべての評価が終了するのを待ち、結果のリストを返す。**sys:plist** は、リスト化されたそれぞれの form が関数呼び出しされることを除いて、*parallel-list* としてみなされるだろう。

12.5 同期

Solaris オペレーティングシステム内には、マルチスレッドプログラムのために 4 つの同期手法がある。Euslisp は、mutex-lock と condition variable とセマフォを提供している。reader-writer lock は実現されてない。これらの手法に基づいて、同期メモリポートや同期障壁のような高レベルの同期機構が実現されている。

sys:make-mutex-lock [関数]

mutex-lock を作り、返す。mutex-lock は、6 つの要素を持つ整数ベクトルで表現されている。

sys:mutex-lock *mlock* [関数]

mutex-lock の *mlock* をロックする。もし、*mlock* が既に他のスレッドからロックされているなら、**mutex-lock** はロックが外されるまで待つ。

sys:mutex-unlock *mlock* [関数]

mlock を解除し、このロックを待っている他のスレッドの内の 1 つが再び実行され始める。

sys:mutex *mlock &rest forms* [マクロ]

mutex-lock と mutex-unlock は、組みで使用されなければならない。**mutex** は、重要な部分をひとまとまりにしたマクロである。*mlock* は、評価する *forms* が評価される前にロックされる。そして、評価が終了したときに、ロックが解除される。このマクロは、以下の progn form に展開される。**unwind-protect** は、*forms* の評価中にエラーが発生したときでさえ、ロックの解除を保証するために使用されることに注意すること。

```
(progn
    (sys:mutex-lock mlock)
    (unwind-protect
        (progn . forms)
        (sys:mutex-unlock mlock)))
```

sys:make-cond [関数]

4 つの要素を持つ整数ベクトルである condition variable オブジェクトを作る。condition variable の返り値としては、ロックされてない状態でである。

sys:cond-wait *condvar mlock* [関数]

condvar に信号が出されるまで待つ。もし、*condvar* が他のスレッドによってすでに獲得されていたならば、*mlock* を解除し、*condvar* に信号が出されるまで待つ。

sys:cond-signal *condvar* [関数]

condvar で示される condition variable に信号を出す。

sys:make-semaphore [関数]

20 の要素を持つ整数ベクトルによって表現されるセマフォオブジェクトを作る。

sys:sema-post *sem* [関数]

sem に信号を出す。

sys:sema-wait *sem* [関数]

sem に信号が来るまで待つ。

sys:barrier-synch [クラス]

:super	**propertied-object**
:slots	threads n-threads count barrier-cond threads-lock count-lock

同期障壁のための構造を表現する。同期を待っているスレッドは、*thread-lock* によって相互に排除される *thread* に置かれる。**barrier-synch** オブジェクトが生成されたとき、*count* は、ゼロに初期化される。同期しているスレッドは、`:add` メッセージを送ることによって、*threads* リストに置かれる。この barrier-synch オブジェクトに`:wait` を送ることは、*count* を増加させることの原因となり、送られたスレッドは待ち状態になる。*threads* の中のすべてのスレッドに`:wait` メッセージが送られたとき、待ちが解除され、すべてのスレッドの実行が再び始まる。同期は、*count-lock* の mutex-lock と *barrier-cond* の condition-variable の組み合わせによって実行される。

:init [メソッド]
この barrier-synch オブジェクトを初期化する。2 つの mutex-lock と 1 つの condition-variable が生成される。

:add *thr* [メソッド]
threads リストの中に *thr* スレッドが追加される。

:remove *thr* [メソッド]
threads リストの中から *thr* スレッドを削除する。

:wait [メソッド]
threads リストの中のすべてのスレッドに`:wait` が配布されるのを待つ。

sys:synch-memory-port [クラス]

:super	**propertied-object**
:slots	sema-in sema-out buf empty lock

1 方向の同期されたメモリポートを実現する。このオブジェクトを通じてデータを転送するために、2 つのスレッドを同期させる。転送制御は、セマフォを用いて実現されている。

:read [メソッド]
この **synch-memory-port** にバッファされているデータを読む。もし、まだ書かれていなかったならば、`:read` は停止する。

:write *datum* [メソッド]
バッファに *datum* を書き込む。1 ワードのバッファのみ存在するので、もし他のデータが既に書かれておりまだ読まれていなかったならば、`:write` は`:read` によってそのデータが読み込まれるまで待つ。

:init [メソッド]
この **sync-memory-port** を初期化する。これには 2 つのセマフォが生成され、`:write` 動作が可能な状態になっている。

13 幾何学関数

13.1 実数ベクトル (float-vector)

float-vector は、要素が実数である 1 次元ベクトルである。float-vector は、どんなサイズでも良い。*result* が引き数リストで指定されているとき、その *result* は float-vector であるべきである。

float-vector *&rest numbers* [関数]
numbers を要素とする float-vector を新しく作る。`(float-vector 1 2 3)` と `#F(1 2 3)` の違いに注意すること。前者は、呼ばれたときはいつでもベクトルが生成されるが、後者は読み込まれたときのみ生成される。

float-vector-p *obj* [関数]
obj が float-vector であるならば、T を返す。

v+ *fltvec1 fltvec2 &optional result* [関数]
2 つの float-vector を加える。

v- *fltvec1 &optional fltvec2 result* [関数]
2 つの float-vector を差し引く。もし、*fltvec2* が省略されているならば、*fltvec1* の符号が反転される。

v. *fltvec1 fltvec2* [関数]
2 つの float-vector の内積を計算する。

v* *fltvec1 fltvec2 &optional result* [関数]
2 つの float-vector の外積を計算する。

v.* *fltvec1 fltvec2 fltvec3* [関数]
スカラー 3 重積を計算する。`(v.* A B C)=(V. A (V* B C))=(V. (V* A B) C)`

v< *fltvec1 fltvec2* [関数]
もし、*fltvec1* の要素が *fltvec2* の対応する要素よりすべて小さいとき、T を返す。

v> *fltvec1 fltvec2* [関数]
もし、*fltvec1* の要素が *fltvec2* の対応する要素よりすべて大きいとき、T を返す。

vmin *&rest fltvec* [関数]
fltvec の中のそれぞれの次元における最小値を捜し、その値で float-vector を新しく作る。**vmin** と **vmax** は、頂点の座標から最小の minimal-box を見つけるために使用される。

vmax *&rest fltvec* [関数]
fltvec の中のそれぞれの次元における最大値を捜し、その値で float-vector を新しく作る。

minimal-box *v-list minvec maxvec [err]* [関数]
与えられた *v-list* に対して `minimal bounding box` を計算し、その結果を *minvec* と *maxvec* に蓄積する。もし、実数 *err* が指定されているならば、`minimal box` はその比率によって成長する。すなわち、もし *err* が 0.01 のとき、*minvec* のそれぞれの要素は *minvec* と *maxvec* との距離の 1%減少する。そして、*maxvec* のそれぞれの要素は 1%増加する。**minimal-box** は、*minvec* と *maxvec* との距離を返す。

scale *number fltvec &optional result* [関数]
fltvec のすべての要素をスカラー *number* 倍する。

norm *fltvec* [関数]

fltvec のノルムを求める。$\|fltvec\|$

norm2 *fltvec* [関数]
fltvec のノルムの 2 乗を求める。$\|fltvec\|^2$=(v. fltvec fltvec)

normalize-vector *fltvec &optional result* [関数]
fltvec のノルムが 1.0 となるように正規化する。

distance *fltvec1 fltvec2* [関数]
2 つの float-vector の距離を返す。$|fltvec - fltvec2|$

distance2 *fltvec1 fltvec2* [関数]
2 つの float-vector の距離の 2 乗を返す。$|fltvec - fltvec2|^2$

homo2normal *homovec &optional normalvec* [関数]
同次ベクトル *homovec* を正規表現に変換する。

homogenize *normalvec &optional homovec* [関数]
正規ベクトル *normalvec* を同次表現に変換する。

midpoint *p p1 p2 &optional result* [関数]
p は実数で、*p1,p2* は同次元の float-vector である。*p1*−*p2* を $p : (1-p)$ の比率で等分した点 $(1-p)\cdot p1+p\cdot p2$ を返す。

rotate-vector *fltvec theta axis &optional result* [関数]
2 次元あるいは 3 次元の *fltvec* を *axis* 回りに *theta* ラジアン回転する。*axis* は、*:x, :y, :z, 0, 1, 2,* または NIL の内の一つである。*axis* が NIL のとき、*fltvec* は 2 次元として扱われる。3 次元空間の任意の軸の回りにベクトルを回転するためには、**rotation-matrix** で回転行列を作り、そのベクトルにかければよい。

13.2 行列と変換

行列は、要素がすべて実数の 2 次元の配列である。ほとんどの関数において行列はどんなサイズでもよいが、**v*, v.*, euler-angle, rpy-angle** 関数では 3 次元の行列のみ扱うことができる。**transform, m*** と **transpose** は、行列を正方行列に限定せず、一般の n*m 行列に対して処理を行う。

result パラメータを受けた関数は、計算結果をそこに置く。そのため、ヒープは使用しない。すべての行列関数は、正規座標系における変換を考慮しており、同次座標系は考慮していない。

rpy-angle 関数は、回転行列をワールド座標系における z,y,x 軸回りの 3 つの回転角に分解する。**euler-angle** 関数は **rpy-angle** と同様に分解するが、回転軸がローカル座標系の z,y,z 軸となっている。角度が反対方向にも得られるため、これらの関数は 2 つの解を返す。

```
; Mat is a 3X3 rotation matrix.
(setq rots (rpy-angle mat))
(setq r (unit-matrix 3))
(rotate-matrix r (car rots) :x t r)
(rotate-matrix r (cadr rots) :y t r)
(rotate-matrix r (caddr rots) :z t r)
;--> resulted r is equivalent to mat
```

3 次元空間の位置と方向の組みを保つために、13.4 節に記載されている **coordinates** と **cascaded-coords** クラスを使用すること。

matrix *&rest elements* [関数]
elements から行列を新しく作る。`Row x Col = (elements` の数) x (最初の `element` の長さ) *elements* は、どの型の列 ((list 1 2 3) や (vector 1 2 3) や (float-vector 1 2 3)) でもよい。それぞれの列は行列の行ベクトルとしてならべられる。

make-matrix *rowsize columnsize &optional init* [関数]
rowsize × *columnsize* の大きさの行列を作る。

matrixp *obj* [関数]
もし、*obj* が行列のとき、すなわち、*obj* が 2 次元の配列でその要素が実数であるとき、T を返す。

matrix-row *mat row-index* [関数]
行列 *mat* から *row-index* で示される行ベクトルを抽出する。**matrix-row** は、**setf** を使用することにより行列の特定の行にベクトルを設定することにも使用される。

matrix-column *mat column-index* [関数]
行列 *mat* から *coloumn-index* で示される列ベクトルを抽出する。**matrix-column** は、**setf** を使用することにより行列の特定の列にベクトルを設定することにも使用される。

m* *matrix1 matrix2 &optional result* [関数]
matrix1 と *matrix2* の積を返す。

transpose *matrix &optional result* [関数]
matrix の転置行列を返す。すなわち、*matrix* の列と行を入れ替える。

unit-matrix *dim* [関数]
dim × *dim* の単位行列を作る。

replace-matrix *dest src* [関数]
行列 *dest* のすべての要素を同一な行列 *src* で置き換える。

scale-matrix *scalar mat* [関数]
mat のすべての要素に *scaler* を掛ける。

copy-matrix *matrix* [関数]
matrix のコピーを作る。

transform *matrix fltvector &optional result* [関数]
行列 *matrix* をベクトル *fltvector* の左から掛ける。

transform *fltvector matrix &optional result* [関数]
行列 *matrix* をベクトル *fltvector* の右から掛ける。

rotate-matrix *matrix theta axis &optional world-p result* [関数]
rotate-matrix で行列 *matrix* を回転させるとき、回転軸（**:x, :y, :z** または 0,1,2）はワールド座標系あるいはローカル座標系のどちらかを与えられる。もし、*world-p* に NIL が指定されているとき、ローカル座標系の軸に沿った回転を意味し、回転行列を左から掛ける。もし、*world-p* が non-NIL のとき、ワールド座標系に対する回転行列を作り、回転行列を右から掛ける。もし、*axis* に NIL が与えられたとき、行列 *matrix* は 2 次元と仮定され、*world-p* の如何にかかわらず 2 次元空間の回転が与えられる。

rotation-matrix *theta axis &optional result* [関数]

axis 軸回りの 2 次元あるいは 3 次元の回転行列を作る。軸は:x,:y,:z,0,1,2,3 次元ベクトルあるいは NIL のどれかである。2 次元回転行列を作るとき、*axis* は NIL でなければならない。

rotation-angle *rotation-matrix* [関数]

rotation-matrix から等価な回転軸と角度を抽出し、実数と float-vector のリストを返す。*rotation-matrix* が単位行列のとき、NIL が返される。また、回転角が小さいとき、結果がエラーとなる。*rotation-matrix* が 2 次元のとき、1 つの角度値が返される。

rpy-matrix *ang-z ang-y ang-x* [関数]

ロール、ピッチ、ヨー角で定義される回転行列を作る。最初に、単位行列を x 軸回りに *ang-x* ラジアン回転させる。次に、y 軸回りに *ang-y* ラジアン、最後に z 軸回りに *ang-z* ラジアン回転させる。すべての回転軸はワールド座標系で与えられる。

rpy-angle *matrix* [関数]

matrix の 2 組のロール、ピッチ、ヨー角を抽出する。

Euler-matrix *ang-z ang-y ang2-z* [関数]

3 つのオイラー角で定義される回転行列を作る。最初に単位行列を z 軸回りに *ang-z* 回転させ、次に y 軸回りに *ang-y* 回転させ、最後に z 軸回りに *ang2-z* 回転させる。すべての回転軸はローカル座標系で与えられる。

Euler-angle *matrix* [関数]

matrix から 2 組のオイラー角を抽出する。

13.3 LU 分解

lu-decompose と **lu-solve** は、線形の連立方程式を解くために用意されている。最初に、**lu-decompose** は行列を下三角行列を上三角行列に分解する。もし、行列が特異値なら、**lu-decompose** は NIL を返す。そうでなければ、**lu-solve** に与えるべき順列ベクトルを返す。**lu-solve** は、与えられた定数ベクトルの解を LU 行列で計算する。この手法は、同じ係数行列と異なった定数ベクトルのたくさんの組に対して解を求めたいときに効果的である。**simultaneous-equation** は、1 つの解だけを求めたいときにもっとも手軽な関数である。**lu-determinant** は、LU 分解された行列の行列式を計算する。**inverse-matrix** 関数は、**lu-decompose** を 1 回と **lu-solve** を n 回使って逆行列を求める。3*3 行列での計算時間は約 4ms である。

lu-decompose *matrix &optional result* [関数]

matrix に LU 分解を実行する。

lu-solve *lu-mat perm-vector bvector [result]* [関数]

LU 分解された 1 次連立方程式を解く。*perm-vector* は、**lu-decompose** で返された結果でなければならない。

lu-determinant *lu-mat perm-vector* [関数]

LU 分解された行列の行列式を求める。

simultaneous-equation *mat vec* [関数]

係数が *mat* で、定数が *vec* で記述される 1 次連立方程式を解く。

inverse-matrix *mat* [関数]

正方行列 *mat* の逆行列を求める。

pseudo-inverse *mat* [関数]

特異値分解を用いて擬似逆行列を求める。

13.4 座標系

座標系と座標変換は、**coordinates** クラスで表現される。4*4 の同次行列表現の代わりに、Euslisp 内で座標系は、高速性と一般性のために 3*3 の回転行列と 3 次元位置ベクトルの組で表現される。

coordinates [クラス]

:super	**propertied-object**
:slots	(pos :type float-vector
	rot :type array)

位置ベクトルと 3x3 の回転行列の組みで座標系を定義する。

coordinates-p *obj* [関数]
obj が **coordinates** クラスかまたはそのサブクラスのインスタンスのとき、T を返す。

:rot [メソッド]
この座標系の 3x3 回転行列を返す。

:pos [メソッド]
この座標系の 3 次元位置ベクトルを返す。

:newcoords *newrot &optional newpos* [メソッド]
newrot と *newpos* でこの座標系を更新する。*newpos* が省略された時は *newrot* には coordinates のインスタンスを与える。この座標系の状態が変化するときはいつでも、このメソッドを用いて新しい回転行列と位置ベクトルに更新するべきである。このメッセージはイベントを伝えて他の**:update** メソッドを呼び出す。

:replace-coords *newrot &optional newpos* [メソッド]
:newcoords メソッドを呼び出さずに `rot` と `pos` スロットを変更する。*newpos* が省略された時は *newrot* には coordinates のインスタンスを与える。

:coords [メソッド]

:copy-coords *&optional dest* [メソッド]
もし *dest* が与えられなかったとき、**:copy-coords** は同じ `rot` と `pos` スロットを持つ座標系オブジェクトを作る。もし、*dest* が与えられたとき、*dest* 座標系にこの座標系の rot と pos をコピーする。

:reset-coords [メソッド]
この座標系の回転行列を単位行列にし、位置ベクトルをすべてゼロにする。

:worldpos [メソッド]

:worldrot [メソッド]

:worldcoords [メソッド]
このオブジェクトのワールド座標系における位置ベクトル･回転行列･座標系を計算する。その座標系は、いつもワールド座標系で表現されていると仮定され、これらのメソッドは簡単に `pos` と `rot` と `self` を返すことができる。これらのメソッドは **cascaded-coords** クラスと互換性がとられている。**cascaded-coords** ではワールド座標系での表現と仮定していない。

:copy-worldcoords *&optional dest* [メソッド]
最初に、ワールド座標系が計算され、*dest* にコピーされる。もし、*dest* が指定されてないとき、新たに **coordinates** オブジェクトを作る。

:rotate-vector *vec* [メソッド]
この座標系の回転行列によって *vec* を回転させる。すなわち、この座標系で表現される方向ベクトルをワールド座標系における表現に変換する。この座標系の位置は、回転に影響を与えない。

:transform-vector *vec* [メソッド]
この座標系で表現される *vec* をワールド座標系の表現に変換する。

:inverse-transform-vector *vec* [メソッド]
ワールド座標系における *vec* をローカル座標系の表現に逆変換する。

:transform *trans &optional (wrt :local)* [メソッド]
wrt 座標系で表現される *trans* によってこの座標系を変換する。*trans* は座標系の型でなければならないし、*wrt* は:local, :parent, :world のキーワードあるいは **coordinates** のインスタンスでなければならない。もし *wrt* が:local のとき、*trans* をこの座標系の右から適用する。もし *wrt* が:world, :parent のとき、*trans* を左から掛ける。もし *wrt* が **coordinates** の型であるとき、*wrt* 座標系で表現される *trans* は最初にワールド座標系の表現に変換され、左から掛ける。

:move-to *trans &optional (wrt :local)* [メソッド]
wrt で表現される *trans* でこの座標系の rot と pos を置き換える。

:translate *p &optional (wrt :local)* [メソッド]
このオブジェクトの位置を *wrt* 座標系で相対的に変更する。

:locate *p &optional (wrt :local)* [メソッド]
この座標系の位置を *wrt* 座標系で絶対的に変更する。もし、*wrt* が:local のとき、**:translate** と同一な効果を生む。

:rotate *theta axis &optional (wrt :local)* [メソッド]
axis 軸回りに *theta* ラジアンだけ相対的にこの座標系を回転させる。*axis* は、軸キーワード (:x, :y, :z) あるいは任意の float-vector である。*axis* は *wrt* 座標系で表現されていると考える。よって、もし *wrt* が:local で *axis* が:z であるとき、座標系はローカル座標系の z 軸回りに回転される。もし *wrt* が:world, :parent であるとき、ワールド座標系の z 軸回りに回転される。言い換えると、もし *wrt* が:local のとき、回転行列はこの座標系の右から掛けられる。そして、もし *wrt* が:world あるいは:parent のとき、回転行列は左から掛けられる。*wrt* が:world あるいは:parent でさえ、この座標系の pos ベクトルは変化しない。本当にワールド座標系の軸回りに回転するためには、回転を表現する **coordinates** クラスのインスタンスを**:transform** メソッドに与えなければならない。

:orient *theta axis &optional (wrt :local)* [メソッド]
rot を強制的に変更する。**:rotate** メソッドの絶対値版である。

:inverse-transformation [メソッド]
この座標系の逆変換を持つ座標系を新しく作る。

:transformation *coords (wrt :local)* [メソッド]
この座標系と引き数で与えられる *coords* との間の変換を作る。もし、*wrt* が:local であるとき、ローカル座標系で表現される。すなわち、もしこの**:transformation** の結果を**:transform** の引き数として *wrt*=:local と一緒に与えたとき、この座標系は *coords* と同一な座標系に変換される。

:Euler *az1 ay az2* [メソッド]

オイラー角 (*az1, ay, az2*) で表現される回転行列を rot に設定する。

:roll-pitch-yaw *roll pitch yaw* [メソッド]

ロール・ピッチ・ヨー角で表現される回転行列を rot に設定する。

:4x4 *&optional mat44* [メソッド]

もし、*mat44* として 4x4 行列が与えられるとき、3x3 回転行列と 3 次元位置ベクトルの座標表現に変換される。もし、*mat44* が与えられないとき、この座標系の表現を 4x4 の同次行列表現に変換して返す。

:init *&key :pos #f(0 0 0)* [メソッド]
:rot #2f((1 0 0) (0 1 0) (0 0 1))
:rpy roll pitch yaw
:Euler az ay az2
:axis rotation-axis
:angle rotation-angle
:4X4 4x4 matrix
:coords another coordinates
:properties a list of (ind . value) pair
:name name property

この **coordinates** オブジェクトを初期化し、rot と pos を設定する。それぞれのキーワードの意味は、以下に示す通りである。

:dimension 2 あるいは 3 (デフォルトは 3)

:pos 位置ベクトルを指定する (デフォルトは #f(0 0 0))

:rot 回転行列を指定する (デフォルトは単位行列)

:Euler オイラー角として 3 つの要素の列を与える

:rpy ロール・ピッチ・ヨー角として 3 つの要素の列を与える

:axis 回転軸 (:x,:y,:z あるいは任意の float-vector)

:angle 回転角 (*:axis* と一緒に使用)

:wrt 回転軸を示す座標系 (デフォルトは :local)

:4X4 4X4 行列 (pos と rot を同時に指定)

:coords *coords* から rot と pos をコピーする

:name :name 値を設定する。

:angle は*:axis* と組みで唯一使用することができる。その軸は*:wrt* 座標系で決定される。*:wrt* と関係なしに*:Euler* はローカル座標系で定義されるオイラー角（*az1, ay* と *az2*）をいつも指定する。また、*:rpy* はワールド座標系の *z, y* と *x* 軸回りの角度を指定する。*:rot, :Euler, :rpy, :axis, :4X4* の中から 2 つ以上を連続で指定することはできない。しかしながら、指定してもエラーは返さない。*axis* と *:angle* パラメータには列を指定することができる。その意味は、与えられた軸回りの回転を連続的に処理する。属性とその値の組みのリストを*:properties* の引き数として与えることができる。これらの組みは、この座標系の plist にコピーされる。

13.5 連結座標系

cascaded-coords [クラス]

:super **coordinates**
:slots (parent descendants worldcoords manager changed)

連結された座標系を定義する。**cascaded-coords** は、しばしば **cascoords** と略す。

:inheritance [メソッド]
この **cascaded-coords** の子孫をすべて記述した継承 tree リストを返す。もし、a と b がこの座標系の直下の子孫で c が a の子孫であるとき、((a (c)) (b)) を返す。

:assoc *childcoords &optional relative-coords* [メソッド]
childcoords は、この座標系の子孫として関係している。もし、*childcoords* が既に他の **cascaded-coords** に **assoc** されているとき、*childcoords* はそれぞれの **cascaded-coords** が 1 つの親しか持っていないなら **dessoc** される。ワールド座標系における *childcoords* の方向あるいは位置は変更されない。

:dissoc *childcoords* [メソッド]
この座標系の子孫リストから *childcoords* を外す。ワールド座標系における *childcoords* の方向あるいは位置は変更されない。

:changed [メソッド]
この座標系の親座標系が変更されていることを通知する。また、もっとあとでワールド座標系が要求されたとき、ワールド座標系を再計算する必要がある。

:update [メソッド]
現在のワールド座標系を再計算するために**:worldcoords** メソッドを呼び出す。

:worldcoords [メソッド]
ルートの座標系からこの座標系までの全ての座標系を連結させることにより、この座標系をワールド座標系で表現した **coordinates** オブジェクトで返す。その結果は、このオブジェクトが持ち、後に再利用される。よって、この結果の座標系を変更すべきでない。

:worldpos [メソッド]
ワールド座標系で表現したこの座標系の rot を返す。

:worldrot [メソッド]
ワールド座標系で表現したこの座標系の pos を返す。

:transform-vector *vec* [メソッド]
vec をこのローカル座標系での表現とみなして、ワールド座標系での表現に変換する。

:inverse-transform-vector *vec* [メソッド]
ワールド座標系で表現される *vec* をこのローカル座標系の表現に逆変換する。

:inverse-transformation [メソッド]
この座標系の逆変換を表現する **coordinates** のインスタンスを作る。

:transform *trans &optional (wrt :local)* [メソッド]

:translate *fltvec &optional (wrt :local)* [メソッド]

:locate *fltvec &optional (wrt :local)* [メソッド]

:rotate *theta axis &optional (wrt :local)* [メソッド]

:orient *theta axis &optional (wrt :local)* [メソッド]

coordinates クラスの記述を参照すること。

make-coords *&key :pos :rot :rpy :Euler :angle :axis :4X4 :coords :name* [関数]

make-cascoords *&key :pos :rot :rpy :Euler :angle :axis :4X4 :coords :name* [関数]

coords *&key :pos :rot :rpy :Euler :angle :axis :4X4 :coords :name* [関数]

cascoords *&key :pos :rot :rpy :Euler :angle :axis :4X4 :coords :name* [関数]

これらの関数は、すべて **coordinates** あるいは **cascaded-coords** を新しく作る。キーワードパラメータについては、**coordinates** クラスの**:init** メソッドを見ること。

transform-coords *coords1 coords2 &optional (coords3 (coords))* [関数]

coords1 が *coords2* に左から適用（乗算）される。その積は *coords3* に蓄積される。

transform-coords* *&rest coords* [関数]

coords にリスト表現されている変換を連結させる。連結された変換で表現される **coordinates** のインスタンスを返す。

wrt *coords vec* [関数]

vec を *coords* における表現に変換する。その結果は (`send` *`coords`* `:transform-vector` *`vec`*) と同一である。

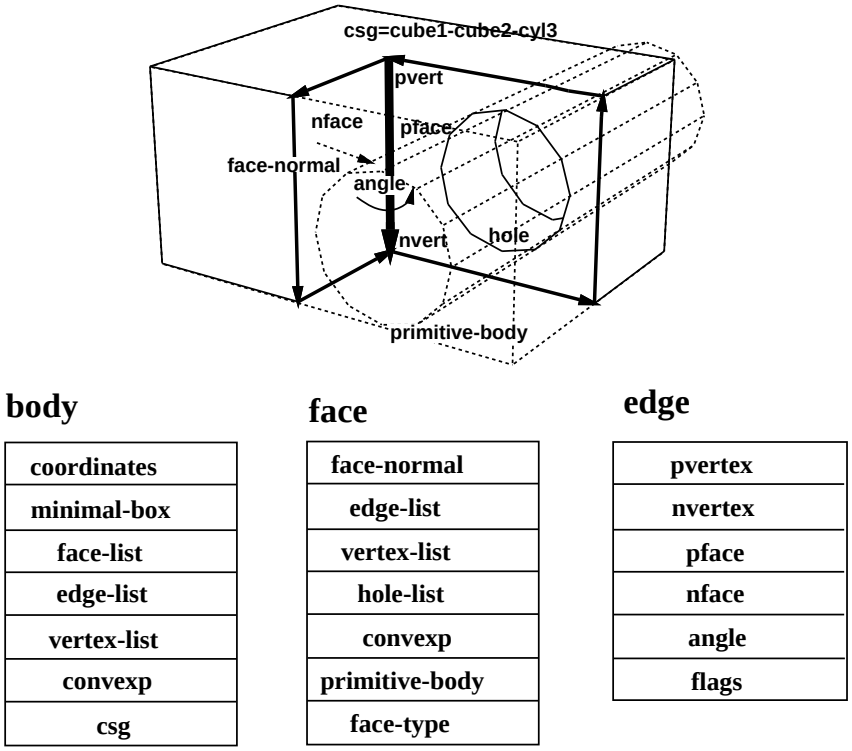

図 9: 頂点とエッジと面の分類

14 幾何学モデリング

Euslisp は、3 次元の幾何学モデルの内部表現として *B*rep（境界表現）を採用している。Brep 内の要素は **edge, plane, polygon, face, hole,** や **body** クラスによって表現される。基本 body の作成関数と body の合成関数は、これらのクラスの新しいインスタンスを作る。もっと属性を持った独自の幾何学クラスを使用するためには、***edge-class*, *face-class***と***body-class***の特殊変数に独自のクラスオブジェクトを設定すること。

14.1 種々の幾何学関数

vplus *vector-list* [関数]

vector-list のすべての要素の合計を実数ベクトルとして新しく作り、返す。**v+**との違いは、**vplus** が 2 つ以上の引数について合計を計算し、結果のベクトルが指定できない点である。

vector-mean *vector-list* [関数]

vector-list の平均ベクトルを返す。

triangle *a b c &optional (normal #f(0 0 1))* [関数]

a, b, c は、2 次元または 3 次元の実数ベクトルである。*normal* は、*a,b,c* が置かれる平面の正規ベクトルである。**triangle** は *a,b,c* で形作られる三角形の領域の 2 倍の大きさを返す。*normal* と同じ方向から見たときに *a,b,c* が時計方向に回転するならば、**triangle** は正である。言い換えると、もし **triangle** が正ならば、*c* は *a-b* の線分の左手側に位置し、*b* は *a-c* の右手側に位置している。

triangle-normal *a b c* [関数]

a b c で定義される三角形に対して垂直方向の正規ベクトルを見つける。

vector-angle *v1 v2 &optional (normal (v* v1 v2))* [関数]
2 つのベクトルの角度を計算する。これは次の式であらわされる `atan(normal·(v1×v2), v1·v2)`。*v1,v2* と *normal* は正規ベクトルでなければならない。*normal* が与えられないとき、*v1,v2* の共通垂線の正規ベクトルが使用される。この場合、結果は 0 から π までの範囲の正の角度になる。符号付きの角度を得るためには、*normal* を指定しなければならない。

face-normal-vector *vertices* [関数]
同じ平面の上にあるベクトルのリストから面の正規化ベクトルを計算する。

farthest *p points* [関数]
3 次元ベクトルのリスト *points* の中から *p* より最も遠い点を捜す。

farthest-pair *points* [関数]
3 次元ベクトルのリスト *points* からもっとも遠い点の組を捜す。

maxindex *3D-floatvec* [関数]
3D-floatvec の 3 つの要素の中で絶対値が最大の要素の位置を捜す。

random-vector *&optional (range 1.0)* [関数]
3 次元デカルト空間の中で同次的に分散されるランダムベクトルを発生する。

random-normalized-vector *&optional (range 1.0)* [関数]
3 次元の正規化ランダムベクトルを返す。

random-vectors *count range* [関数]
range の大きさのランダムベクトルを *count* 個つくり、そのリストを返す。

line-intersection *p1 p2 p3 p4* [関数]
p1, p2, p3, p4 は、すべて 2 次元以上の実数ベクトルである。*p1-p2* と *p3-p4* が平面上の 2 つの線分として定義される。**line-intersection** は、これらの 2 つの線分の交差する点のパラメータ（線分に置ける交点の位置の比率）を 2 要素のリストで返す。3 次元で使用するとき、*p1, p2, p3, p4* は共通平面内になければならない。

collinear-p *p1 p2 p3 &optional tolerance* [関数]
p1, p2, p3 は、すべて 3 次元の実数ベクトルで 3 つの点を表現している。**collinear-p** は、もし $\|((p2-p1)\times(p3-p1)\|$ が `*coplanar-threshold*` より小さければ、*p1-p3* の線分の上に *p2* を投影したときのパラメータを返す。そうでなければ、NIL を返す。

find-coplanar-vertices *p1 p2 p3 vlist* [関数]
p1, p2, p3 は、3 次元の実数ベクトルで、この 3 つのベクトルから平面を表現している。**find-coplanar-vertices** は、その平面内にある点を *vlist* の中から捜す。

find-connecting-edge *vertex edgelist* [関数]
vertex に接続された *edgelist* の中からエッジを捜す。

make-vertex-edge-htab *bodfacs* [関数]
bodfacs は、**body** あるいは **face** のリストである。**make-vertex-edge-htab** は、*bodfacs* の中の頂点を抽出し、それに接続されるエッジの検索ができるハッシュテーブルを作る。

left-points *points p1 p2 normal* [関数]
points, p1, p2 は、正規化ベクトル *normal* で表現される平面内にあるものと仮定する。**left-points** は、

p1, p2 間の線分の左側に置かれている点を *points* の中から捜し、集める。

right-points *points p1 p2 normal* [関数]

points, p1, p2 は、正規化ベクトル *normal* で表現される平面内にあるものと仮定する。**right-points** は、*p1, p2* 間の線分の右側に置かれている点を *points* の中から捜し、集める。

left-most-point *points p1 p2 normal* [関数]

points, p1, p2 は、正規化ベクトル *normal* で表現される平面内にあるものと仮定する。**left-most-points** は、*p1, p2* で決定される線分の左側に置かれている点を *points* の中から捜し、その中でもっとも遠い点を返す。

right-most-point *points p1 p2 normal* [関数]

points, p1, p2 は、正規化ベクトル *normal* で表現される平面内にあるものと仮定する。**right-most-points** は、*p1, p2* で決定される線分の右側に置かれている点を *points* の中から捜し、その中でもっとも遠い点を返す。

eps= *num1 num2 [(tolerance *epsilon*)]* [関数]

2 つの実数 *num1* と *num2* を比較して、*torelance* の誤差範囲内で等しいかどうかを返す。

eps< *num1 num2 [(tolerance *epsilon*)]* [関数]

num1 が明らかに *num2* よりも小さいとき T を返す。すなわち、`num1<num2-tolerance` である。

eps<= *num1 num2 [(tolerance *epsilon*)]* [関数]

num1 が多分 *num2* よりも小さいときあるいは等しいとき T を返す。すなわち、`num1<num2+tolerance` である。

eps> *num1 num2 [(tolerance *epsilon*)]* [関数]

num1 が明らかに *num2* よりも大きいとき T を返す。すなわち、`num1>num2-tolerance` である。

eps>= *num1 num2 [(tolerance *epsilon*)]* [関数]

num1 が多分 *num2* よりも大きいときあるいは等しいとき T を返す。すなわち、`num1>num2+tolerance` である。

bounding-box [クラス]

:super **object**
:slots (minpoint maxpoint)

xy-,yz-や zx-平面に平行な面を境界とする最小の四角柱を定義する。**bounding-box** は、初期に与えられるベクトルの次元によって、どんな次元でも使用することができる。**bounding-box** は、surrounding-box の名前で定義されていた。

:box [メソッド]

この **bounding-box** のオブジェクト自身を返す。

:volume [メソッド]

この **bounding-box** の体積を返す。

:grow *rate* [メソッド]

この **bounding-box** のサイズを *rate* 率で増加または減少させる。*rate* が 0.01 のとき、1%拡大される。

:inner *point* [メソッド]

point がこの **bounding-box** 内にあれば T を返し、そうでないときは NIL を返す。

:intersection *box2 &optional tolerance* [メソッド]
この **bounding-box** と *box2* との共通 bounding-box を返す。もし、*torelance* が与えられたならば、この **box** はその誤差で拡大される。もし、共通部分がなければ、NIL を返す。

:union *box2* [メソッド]
この **bounding-box** と *box2* を結合した bounding-box を返す。

:intersectionp *box2* [メソッド]
この **bounding-box** と *box2* との間に共通領域があれば T を返し、そうでなければ NIL を返す。このメソッドは、**:intersection** よりも速い。なぜなら、新しい **bounding-box** のインスタンスを作らないためである。

:extreme-point *direction* [メソッド]
この **bounding-box** の 8 つの頂点の中で、*direction* との内積が最大のものを返す。

:corners [メソッド]
この **bounding-box** のすべての頂点のリストを返す。もし、この **box** が 2 次元であれば、4 点が返される。同様に 3 次元の場合、8 点が返される。

:below *box2 &optional (direction #(0 0 1)* [メソッド]
この **bounding-box** が *box2* に対して *direction* の示すベクトルの下の方向にあれば T を返す。この **boundign-box** が *direction* の方向に動かされるとき、2 つの box に共通部分でできるかどうかをチェックするために使用される。

:body [メソッド]
この **bounding-box** によって内包される立方体を表現する **body** を返す。

:init *vlist &optional tolerance* [メソッド]
minpoint と maxpoint スロットを *vlist* から設定する。もし、*torelance* が指定されたなら、この **bounding-box** はその量で増大される。

make-bounding-box *points [tolerance]* [関数]
points のリストの中から最小と最大の座標値を見つけ、bounding-box のインスタンスを作る。

bounding-box-union *boxes [tolerance *contact-threshold*]* [関数]
boxes の結合で表現される bounding-box のインスタンスを作る。その結果は、*tolerance* によって拡張される。

bounding-box-intersection *boxes [tolerance *contact-threshold*]* [関数]
boxes の共通領域を表現する bounding-box のインスタンスを作る。その結果は、*tolerance* によって拡張される。

14.2 線とエッジ

頂点の順番やエッジの順番の向きは、**body** を外から見たときに反時計方向に整列するように定義される。pvertex や nvertex や pface や nface は、pface が外から見たときエッジの左側に位置しているとき、pvertex から nvertex に向かう方向にエッジを定義する。

line [クラス]

:super **propertied-object**
:slots ((pvert :type floatvector) (nvert :type floatvector))

pvert と nvert の上を通る線分を定義する。線分は、*pvert* から *nvert* に向かう方向を持つ。t · pvert +(1-t)nvert

:vertices [メソッド]
pvert と nvert のリストを返す。

:point *p* [メソッド]
この線分の上で *p* パラメータで示される位置の 3 次元のベクトルを返す。p · pvert + (1-p)nvert

:parameter *point* [メソッド]
この線分の上の *point* に対するパラメータを計算する。これは、**:point** メソッドの逆メソッドである。

:direction [メソッド]
pvert から nvert へ向かう正規化ベクトルを返す。

:end-point *v* [メソッド]
この線分の他の端点を返す。すなわち、もし *v* が pvert に等しいとき、nvert を返す。もし *v* が nvert に等しいとき、pvert を返す。それ以外のとき、NIL を返す。

:box [メソッド]
この線分の **bounding-box** を作成し、返す。

:boxtest *box* [メソッド]
box とこの線分の **bounding-box** の共通部分をチェックする。

:length [メソッド]
この線分の長さを返す。

:distance *point-or-line* [メソッド]
この線分と *point-or-line* の間の距離を返す。もし点からこの線分におろした垂線の足が pvert と nvert の間になければ、最も近い端点までの距離を返す。このメソッドを使うことにより、2 つの線分の間の距離を計算することができるため、2 つの円柱の間の干渉をテストすることができる。

:foot *point* [メソッド]
point からこの線分へおろした垂線の足である点を示すパラメータを見つける。

:common-perpendicular *l* [メソッド]
この線分と *l* とに垂直な線分を見つけ、2 つの 3 次元ベクトルのリストとして返す。2 つの線分が平行で共通な垂線が一意に決定できないとき、:parallel を返す。

:project *plane* [メソッド]
plane に pvert と nvert を投影した 2 つの点のリストを返す。

:collinear-point *point &optional (tolerance *coplanar-threshold*)* [メソッド]
collinear-p を用いて *torelance* の誤差範囲内で *point* がこの線分と一直線上にあるかどうかをチェックする。もし、*point* がこの線分と一直線上にあるとき、その線分のその点に対するパラメータを返す。そうでなければ、NIL を返す。

:on-line-point *point &optional (tolerance *coplanar-threshold*)* [メソッド]
point がこの線分と一直線上にあり、`pvert` と `nvert` との間にあるかどうかをチェックする。

:collinear-line *ln &optional (tolerance *coplanar-threshold*)* [メソッド]
ln がこの線分と共通線上にあるとき、すなわち *ln* の両端がこの線分上にあるとき T を返し、そうでないとき NIL を返す。

:coplanar *ln &optional (tolerance *coplanar-threshold*)* [メソッド]
ln とこの線分が共通平面上にあるかどうかをチェックする。この線分の両端と *ln* の 1 つの端点で平面が定義される。もし、*ln* の他の端点がその平面上にあるとき、T を返す。そうでなければ、NIL を返す。

:intersection *ln* [メソッド]
ln は、この線分と共通平面上にあるとする。**:intersection** は、これら 2 つの線分の交点に対する 2 つのパラメータのリストを返す。パラメータは 0 から 1 までの実数である。これは、両端で区切られた線分の内分点を示す。2 つの線が平行であるとき NIL を返す。

:intersect-line *ln* [メソッド]
ln は、この線分と共通平面上にあるとする。交点のパラメータが`:parallel, :collinear` や`:intersect` のようなシンボル情報と共に返される。

edge [クラス]

:super	**line**
:slots	(pface nface
	(angle :type float)
	(flags :type integer))

2 つの面の間の交差線分として定義されるエッジを表現する。`pface` と `nface` がスロットの中に定義されているが、それらの解釈はこのエッジの方向によって相対的に決まる。例えば、このエッジが `pvert` から `nvert` に向かっていると考えたとき、`pface` が正しい pface を表現している。そのため、**:pface** や **:nface** メソッドで適当な面を選択するためには、pvert と nvert の解釈を与えなければならない。

make-line *point1 point2* [関数]
point1 を `pvert` とし、*point2* を `nvert` とする **line** のインスタンスを作る。

:pvertex *pf* [メソッド]
pf をこのエッジの `pface` とみなした `pvertex` を返す。

:nvertex *pf* [メソッド]
pf をこのエッジの `pface` とみなした `nvertex` を返す。

:body [メソッド]
このエッジを定義する **body** オブジェクトを返す。

:pface *pv nv* [メソッド]
仮想的に *pv* と *nv* をこのエッジの pvert と nvert に解釈したときの pface を返す。

:nface *pv nv* [メソッド]

仮想的に *pv* と *nv* をこのエッジの pvert と nvert に解釈したときの nface を返す。

:binormal *aface* [メソッド]
このエッジと *aface* の正規化ベクトルに垂直な方向ベクトルを見つける。

:angle [メソッド]
このエッジでつながった 2 つの面の間の角度を返す。

:set-angle [メソッド]
このエッジでつながった 2 つの面の間の角度を計算し、それを `angle` スロットに置く。

:invert [メソッド]

:set-face *pv nv f* [メソッド]
f を `pface` とし、*pv* を `pvertex` とし、*nv* を `nvertex` として設定する。このメソッドは、このエッジの `pface` あるいは `nface` を変更することに注意すること。

:contourp *viewpoint* [メソッド]
もし、このエッジが輪郭エッジであれば、すなわち、このエッジの `pface` あるいは `nface` のどちらかが *viewpoint* から見え、もう一方が見えないなら T を返す。

:approximated-p [メソッド]
このエッジが円柱の側面のような曲面を表現するための近似エッジであるならば、T を返す。近似エッジは部分直線で曲線を表現するのに必要である。

:set-approximated-flag *&optional (threshold 0.7)* [メソッド]
Euslisp では、どんな曲面もたくさんの平面で近似される。`flags` の LSB は、このエッジの両側の面が曲面であるかどうかを示すために使用される。

もし、2 つの面の間の角度が *threshold* より大きいなら、**:set-approximated-flag** は、このフラグを T に設定する。

:init *&key :pface :nface :pvertex :nvertex* [メソッド]

14.3 平面と面

plane オブジェクトは、その平面の正規化ベクトルと座標原点から平面までの距離で表現される。2 対の正規化ベクトルと距離が **plane** オブジェクトに記録される。1 つは、変換後の現状を表現し、もう 1 つが平面を定義したときの正規化ベクトルと距離を表現する。

plane [クラス]

:super	**propertied-object**
:slots	((normal :type float-vector)
	(distance :float))

平面方程式を定義する。平面は境界がなく、無限に広がっているものとする。

:normal [メソッド]
この平面の正規化ベクトルを返す。

:distance *point* [メソッド]
この平面と *point* との間の距離を計算する。

:coplanar-point *point* [メソッド]
もし、*point* がこの平面の上に置かれているなら T を返す。

:coplanar-line *line* [メソッド]
もし、*line* がこの平面の上に置かれているなら、T を返す。

:intersection *point1 point2* [メソッド]
point1 と *point2* を端点とする線分とこの平面との交点を計算する。その線分の上の交点に対するパラメータを返す。もし、線分とこの平面が平行であるなら、:parallel を返す。

:intersection-edge *edge* [メソッド]
この平面と *point1* と *point2* で表現される線分あるいはエッジとの交点のパラメータを返す。

:foot *point* [メソッド]
この平面上に *point* を直角に投影した位置の 3 次元ベクトルを返す。

:init *normal point* [メソッド]
point を通り *normal* を面の正規化ベクトルとする平面を定義する。*normal* は、正規化されていなければならない。$|normal| = 1$

polygon [クラス]

:super	**plane**
:slots	(convexp edges vertices
	(model-normal float-vector)
	(model-distance :float))

polygon は、平面の上の輪で表現される。convexp は、その輪が凸面であるかどうかを示す論理フラグである。edges は、この輪の輪郭や頂点のリストである vertices で形成されるエッジのリストである。

:box *&optional tolerance* [メソッド]
この多角形のための **bounding-box** を返す。

:boxtest *box2 &optional tolerance* [メソッド]
この多角形のための **bounding-box** を作成し，その **boundign-box** と *box2* との共通領域を返す。もし，共通領域がなかった場合，NIL を返す。

:edges [メソッド]
この多角形のエッジのリストを返す。そのリストは，この平面の正規化ベクトルに沿ってその多角形を見たとき，時計方向の順番になっている。もし，正規化ベクトルをねじと考えると，そのエッジはねじを入れる方向に回転させる向きの順番になっている。多角形または面が立体オブジェクトの面を表現するために使用されているとき，その正規化ベクトルはその立体の外側に向かっている。多角形をそのオブジェクトの外側から見たとき，エッジは反時計方向の順番になっている。

:edge *n* [メソッド]
エッジの *n* 番目の要素を返す。

:vertices [メソッド]
この多角形の頂点をエッジと同じ順番にならべたものを返す。最初の頂点は，そのリストの最後に重複してコピーされているため，そのリストは実際の頂点の数より 1 だけ長くなっていることに注意すること。これは，頂点のリストを用いてエッジへの変換を簡単にするためである。

:vertex *n* [メソッド]
頂点の *n* 番目の要素を返す。

:insidep *point &optional (tolerance *epsilon*)* [メソッド]
この領域に対して相対的に置かれた *point* の位置にしたがって`:inside`,`:outside` あるいは`:border` を返す。

:intersect-point-vector *point vnorm* [メソッド]
point と正規化方向ベクトル *vnorm* によって定義される擬似線分との交点を計算する。

:intersect-line *p1 p2* [メソッド]
p1 と *p2* で指定される線分との交点を計算する。その結果は、交点がなければ NIL を返し，交点があればその交点の位置のパラメータのリストを返す。

:intersect-edge *edge* [メソッド]
edge で指定される線分との交点を計算する。その結果は，交点がなければ NIL を返し、交点があれば交点の位置のパラメータのリストを返す。

:intersect-face *aregion* [メソッド]
もし，この領域が *aregion* と交差しているなら，T を返す。

:transform-normal [メソッド]

:reset-normal [メソッド]
この多角形の現在の `vertices` リストから面の正規化ベクトルを再計算する。

:invert [メソッド]

:area [メソッド]
この領域の面積を返す。

:init *&key :vertices :edges :normal :distance* [メソッド]

face [クラス]

:super **polygon**
:slots (holes mbody primitive-face id)

穴を持った面を定義する。*mbody* と *type* は、基本 body と body 内の面の属性 (`:top`, `:bottom`, `:side`) を表現する。

:all-edges [メソッド]

:all-vertices [メソッド]
この面および内部ループ（穴）の輪郭のエッジあるいは頂点をすべて返す。**:edges** と**:verticies** メソッドは，輪郭を構成するエッジと頂点のみを返す。

:insidep *point* [メソッド]
point がこの面の内部にあるかどうかを決定する。もし *point* がこの面の外側の輪郭の中にあり，どれかの穴の範囲内にあるならば，外側として分類される。

:area [メソッド]
この面の面積を返す。これは，外側のエッジで囲まれる面積から穴の面積を引いたものである。

:centroid *&optional point* [メソッド]
この面の重心を表現する実数と実数ベクトルのリストを返す。もし，*point* が与えられないならば，最初の数はこの多角形の面積を表わし，2 番目のベクトルがこの多角形の重心の位置を示す。もし，*point* が与えられたならば，この多角形を底面としその点を頂点とするような多角錐を考え，その体積と重心のベクトルを返す。

:invert [メソッド]
この面の向きをひっくり返す。正規化ベクトルが逆方向とされ，エッジループの順番も反転される。

:enter-hole *hole* [メソッド]
この面に穴 *hole* を加える。

:primitive-body [メソッド]
この面を定義する基本 **body** を返す。

:id [メソッド]
`(:bottom)`, `(:top)` や `(:side seq-no.)` の中の 1 つを返す。

:face-id [メソッド]
基本 **body** の型とこの面の型をリストで返す。例えば，円柱の側面は `((:cylinder` *`radius height segments`*`)` `:side` *`id`*`)` を返す。

:body-type [メソッド]
この面を定義する基本 **body** を返す。

:init *&key :normal :distance :edges :vertices :holes* [メソッド]

hole [クラス]

:super **polygon**
:slots (myface)

穴は，面の内部ループを表現する多角形である。**face** のオブジェクトは、自分の `holes` スロットの中に **hole** のリストを持っている。

:face [メソッド]

この **hole** を含む面を返す。

:enter-face *face* [メソッド]

この **hole** を囲んでいる面 *face* へリンクを作る。このメソッドは、**face** クラスの `:enter-hole` メソッドと共に使用されるものである。

:init *&key :normal :distance :edges :vertices :face* [メソッド]

14.4 立体 (body)

body [クラス]

:super **cascaded-coords**
:slots (faces edges vertices model-vertices box convexp evertedp csg)

3次元形状を定義する。

:magnify *rate* [メソッド]
この **body** のサイズを *rate* で変更する。拡大は，csg リストの中に記録される。

:translate-vertices *vector* [メソッド]
モデルの頂点を相対移動する。*vector* はローカル座標系で与えられなければならない。変換は csg リストに記録される。

:rotate-vertices *angle axis* [メソッド]
モデルの頂点を *axis* 軸回りに *angle* ラジアン回転させる。回転は csg リストに記録される。

:reset-model-vertices [メソッド]

:newcoords *rot &optional pos* [メソッド]
座標系を *rot* や *pos* を用いて変更する。*pos* が省略された時は *newrot* には coordinates のインスタンスを与える。

:vertices [メソッド]
この **body** のすべての頂点のリストを返す。

:edges [メソッド]
この **body** のすべてのエッジのリストを返す。

:faces [メソッド]
この **body** を構成するすべての面のリストを返す。

:box [メソッド]
この **body** の **bounding-box** を返す。

:Euler [メソッド]
この **body** のオイラー数を計算する。これは，faces+vertices−edges−2−holes である。これは，−2rings と等しくなるべきである。

:perimeter [メソッド]
すべてのエッジの長さの合計を返す。

:volume *&optional (reference-point #f(0 0 0))* [メソッド]
この **body** の体積を返す。

:centroid *&optional (point #f(0 0 0)* [メソッド]
この **body** が均質な立体と仮定し，重心の位置を返す。

:possibly-interfering-faces *box* [メソッド]

:common-box *body* [メソッド]
この **body** と他の *body* の共通な最小の **box** を返す。もし，2 つの **body** が干渉しているならば，その交差部分はこの共通 **box** の中に存在するはずである。

:insidep *point* [メソッド]
もし，*point* がこの **body** に属するなら，`:inside` を返す。もし，*point* がこの **body** の表面上にある場合，`:border` を返す。そうでなければ，`:outside` を返す。

:intersect-face *face* [メソッド]
もし，この **body** の面と *face* の間に干渉がある場合，T を返す。

:intersectp *body* [メソッド]
この **body** と他の *body* との間の交差部分を返す。

:evert [メソッド]
すべての面とエッジの方向を反転させる。そのため，この **body** の内部は外部になる。

:faces-intersect-with-point-vector *point direction* [メソッド]
point から *direction* の方向に伸びるベクトルと交差する面をすべて集める。

:distance *target* [メソッド]
target は，実数ベクトルあるいは平面オブジェクトである。**:distance** メソッドは，*target* から最も近い面を見つけ，その面と距離のリストを返す。

:csg [メソッド]
body が構築された履歴である `csg` スロットを返す。

:primitive-body [メソッド]
この **body** を構築する基本 **body** のリストを返す。

:primitive-body-p [メソッド]
もし，この **body** が 14.5 節で示される関数の内の 1 つから作られた基本 **body** であるなら，T を返す。

:creation-form [メソッド]
この **body** を作るための Lisp 表現を返す。

:body-type [メソッド]
もし，この **body** が基本 **body** あるいはこの **body** の表現が複雑（に構成された）**body** なら，作成パラメータのリストを返す。

:primitive-groups [メソッド]
2 つの要素をもつリストを返す。最初の要素は，この **body** を構成するために追加 (body+) された基本 **body** のリストである。次の要素は，差し引かれた基本 **body** のリストである。

:get-face *body &optional face id* [メソッド]
body は，この **body** を構成している **body** のインスタンスであり，基本 **body** 型の 1 つである。例えば，`:cube`, `:prism`, `:cone`, `:solid-of-resolution` などかあるいは NIL である。もし，*face* も *id* も与えられないならば，*body* に一致する面をすべて返す。もし，*face* が与えられたなら，その上にフィルターが実行される。*face* は，`:top`,`:bottom` と `:side` の内の 1 つでなければならない。(`send abody :get-face :cylinder :top`) は，`abody` を構成する円柱の上面すべてを返す。もし，*face* が `:side` なら，*id* で番号付けされた面を取り出すことができる。(`send abody nil :side 2`) は，*id* が 0 から始まるため，`abody` を構成する **body** の側面から 3 番目の面をすべて返す。

:init *&key :faces :edges :vertices* [メソッド]

:faces よりこの **body** を初期化する。*:faces* は，必要な引き数である。*:faces*,*:edges* と *:vertices* は完全な立体モデルを定義するために矛盾のない関係を持っていなければならないので，矛盾した引き数でこのメソッドを呼び出すことは，意味の無いことである。**body** を作るために，14.5 節で書いている基本 **body** の作成関数と 14.6 節の **body** 合成関数を使用する。

:constraint *b* [メソッド]

この **body** が *b* に接触しているとき，この **body** の拘束を返す。このメソッドの詳細な説明は 14.8 節を参照すること。

14.5 基本 body の作成関数

make-plane *&key :normal :point :distance* [関数]
point を通り，*normal* の方向を向いた **plane** オブジェクトを作る。*point* を与える代わり *distance* を指定することもできる。

xy-plane [変数]

yz-plane [変数]

zx-plane [変数]

make-cube *xsize ysize zsize &key :name :color* [関数]
x,y,z 軸の方向に大きさが *xsize,ysize,zsize* である，直方体を作る。この直方体の原点は **body** の中心に置かれる。

make-prism *bottom-points sweep-vector &key :name :color* [関数]
sweep-vector に沿った *bottom-points* により定義される形状を積み上げることにより角柱を作る。もし、*sweep-vector* が実数ベクトルでなく数字であれば、*z* 方向の角柱の高さとして扱われる。*bottom-points* は，この **body** の底面を定義する順番になっていなければならない。例えば，`(make-prism '(#f(1 1 0) #f(1 -1 0) #f(-1 -1 0) #f(-1 1 0)) 2.0)` は，高さ 2.0 の直方体を作る。

make-cylinder *radius height &key (:segments 12) :name :color* [関数]
半径 *radius* と高さ *height* で指定される円柱を作る。底面は，xy-平面に定義され，座標系の原点は底面の中心に置かれる。

make-cone *top bottom &key (:segments 16) :color :name* [関数]
頂点が *top* で底面が *bottom* である角錐を作る。*top* は，3 次元ベクトルである。*bottom* は，底面の頂点のリストあるいは半径である。もし，頂点のリストなら，順番を慎重にしなさい。`(make-cone #f(0 0 10) (list #f(10 0 0) #f(0 10 0) #f(-10 0 0) #f(0 -10 0)))` は，正方形の底面を持つ四角錐を作る。

make-solid-of-revolution *points &key (:segments 16) :name :color* [関数]
points は，z 軸まわりの時計方向に回転される。もし、*points* のリストの 2 つの端点が z 軸上に置かれてないならば，曲面を作る。したがって，`(make-solid-of-revolution '(#f(0 0 1) #f(1 0 0)))` は、円錐を作り、`(make-solid-of-revolution '(#f(1 0 1) #f(1 0 0)))` は、円柱を作る。*points* は、順番が重要であり、*z* 軸の高い方から低い方へ整列しておくことが望まれる。

make-torus *points &key (:segments 16) :name :color* [関数]
ドーナッツのような torus 形状を作る。*points* は，断面上の頂点のリストである。

make-icosahedron *&optional (radius 1.0)* [関数]
正 20 面体を作る。それぞれの面は正三角形である。

make-dodecahedron *&optional (radius 1.0)* [関数]
正 12 面体を作る。それぞれの面は，正五角形である。

make-gdome *abody* [関数]

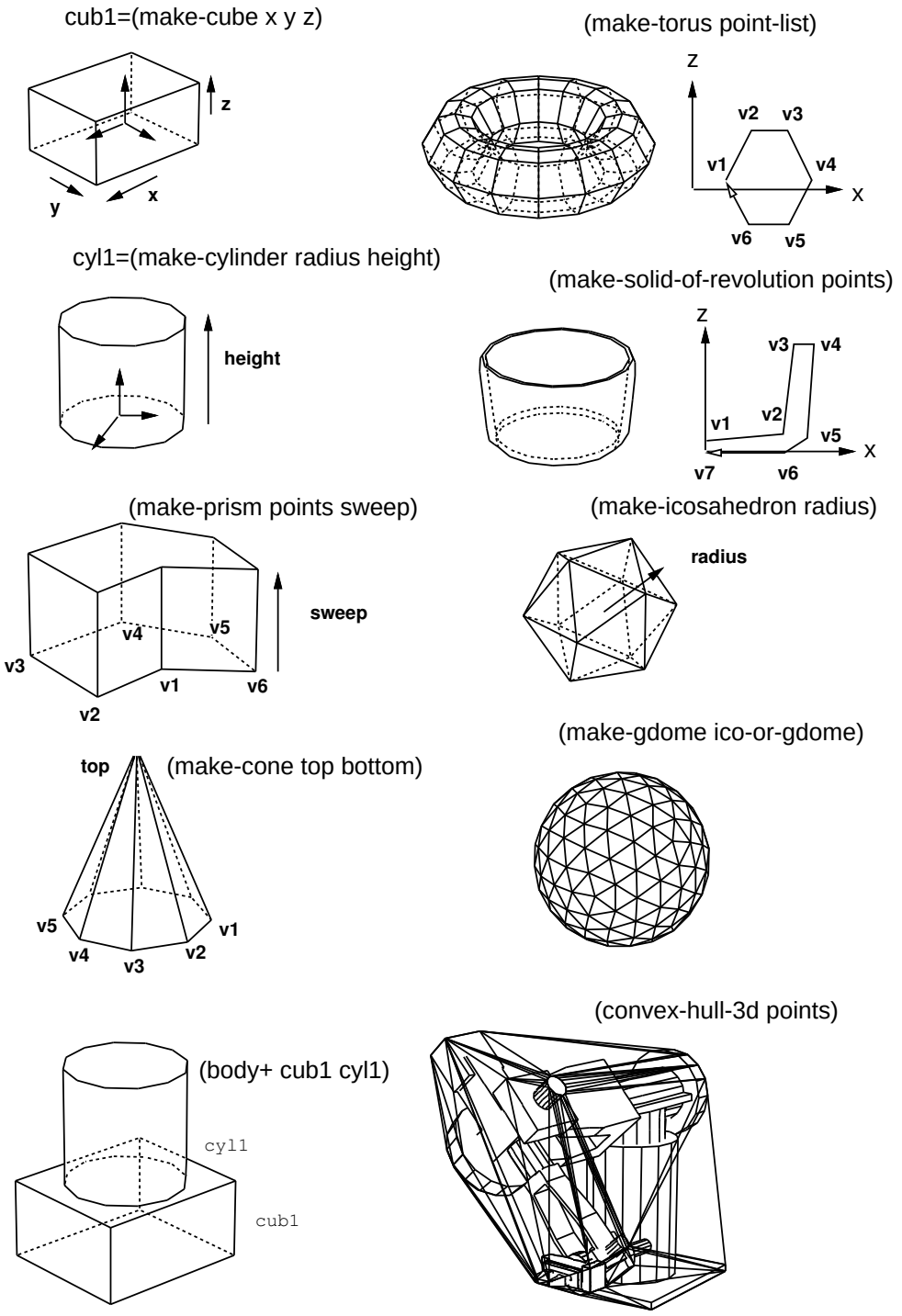
cub1=(make-cube x y z)
z
y
x
(make-torus point-list)
Z
v2
v3
v1
v4
X
v6
v5
cyl1=(make-cylinder radius height)
height
(make-solid-of-revolution points)
Z
v3
v4
v1
v2
v5
X
v7
v6
(make-prism points sweep)
sweep
v4
v5
v3
v1
v6
v2
(make-icosahedron radius)
radius
(make-gdome ico-or-gdome)
top
(make-cone top bottom)
v5
v1
v4
v2
v3
(convex-hull-3d points)
(body+ cub1 cyl1)
cyl1
cub1

図 10: 基本 body

abody の三角面を 4 つの面に小分けすることにより測地ドームを新しく作る。*abody* は，最初正 20 面体とすべきである。それから，**make-gdome** の結果を再帰的に **make-gdome** に与えることができる。それぞれの呼び出しで，測地ドームの面の数は，4 倍に増加する。すなわち，20, 80, 320, 1280, 5120 などになる。

```
(setq g0 (make-icosahedron 1.0))        ; 20 facets
(setq g1 (make-gdome g0))               ; 80 facets
(setq g2 (make-gdome g1))               ; 320 facets
...
```

grahamhull *vertices &optional (normal #f(0 0 1))* [関数]

Graham のアルゴリズムを用いて，2 次元上で凸状の覆いを計算する。**quickhull** よりも遅い。

quickhull *vertices &optional (normal #f(0 0 1))* [関数]

2 分探索法を用いて 2 次元上で凸状の覆いを計算する。

convex-hull-3d *vertices* [関数]

gift-wrapping 法を用いて 3 次元上で凸面の覆いを計算する。

make-body-from-vertices *vertices-list* [関数]

矛盾しない順番になっている面のループを定義する頂点のリストから **body** を返す。

14.6 body の合成関数

face+ *face1 face2* [関数]

face* *face1 face2* [関数]

face1 と *face2* は，3 次元上で共通平面上にある。**face+**は，これらの面の結合を構築し，面のオブジェクトとして返す。もし，交差領域がないなら，元の 2 つの面が返される。**face***は，これらの面の交差領域を返す。もし，交差領域がなければ，NIL を返す。

cut-body *body cutting-plane* [関数]

body を *cutting-plane* で切断し，その切断面に作られる面のリストを返す。

body+ *body1 body2 &rest more-bodies* [関数]

body- *body1 body2* [関数]

body* *body1 body2* [関数]

2 つあるいはそれ以上の **body** の和，差あるいは積を計算する。それぞれの body は，**body+**, **body-**, **body***の処理を行う前にコピーされ，元の body は変更されない。その結果の body の新しい座標系の位置・姿勢は，ワールド座標系のものと一致している。もし，しきい値パラメータ`*coplanar-threshold*`,`*contact-thres`
を正確に設定するなら，2 つの **body** が面同士で接触している場合でもこれらの関数は正しく働くであろう。しかしながら，もし **body** の頂点が他の **body** の頂点あるいは面に接触している場合，どの処理も失敗する。

body/ *body plane* [関数]

make-plane で作られた **plane** クラスのインスタンスである *plane* で *body* を切断する。新しく作られた **body** が返される。

body-interference *&rest bodies* [関数]

bodies の中で 1 対 1 の組み合わせにおける干渉をチェックし，交差している 2 つの **body** のリストを返す。

14.7 座標軸

coordinates-axes クラスは，画面上に表示可能な 3 次元座標軸を定義する。それぞれの軸と z 軸の頂点の矢印は，**line** オブジェクトで定義される。このクラスは，**cascaded-coords** を継承しているので，このクラスのオブジェクトは，**body** のような他の **cascaded-coords** を元とするオブジェクトに付けることができる。このオブジェクトは，**body** の座標軸あるいは他の座標の相対座標系を見るために使用される。

coordinates-axes [クラス]

:super	**cascaded-coords**
:slots	(size model-points points lines)

表示可能な 3 次元座標軸を定義する。

14.8 立体の接触状態解析

この節のメソッドおよび関数は、次のファイルに記述されている。**contact/model2const.l, contact/inequalities.l, contact/drawconst.l**

constrained-motion c [関数]
拘束 c を満たしている動作のリストを返す。

constrained-force m [関数]
拘束されている **body** から拘束している **body** に加わる力を返す。m は、**constrained-motion** から返される動作のリストである。

draw-constraint c [関数]
拘束 c を描く。

draw-motion m a b [関数]
a が b に接触しているときに取り得る動作を描く。リターンキーを押すことにより描画を始める。

Example

```
;;
;;      peg in a hole with 6 contact points
;;
(in-package "GEOMETRY")
(load "view")
(load "../model2const.l" :package "GEOMETRY")
(load "../inequalities.l" :package "GEOMETRY")
(load "../drawconst.l" :package "GEOMETRY")

(setq x (make-prism '(#f(50 50 0) #f(50 -50 0) #f(-50 -50 0) #f(-50 50 0))
                    #f(0 0 200)))
(setq x1 (copy-object x))
(send x1 :translate #f(0 0 -100))
(send x1 :worldcoords)
(setq a1 (make-prism '(#f(100 100 -150) #f(100 -100 -150)
                       #f(-100 -100 -150) #f(-100 100 -150))
                     #f(0 0 150)))
(setq ana (body- a1 x1))
(send x :translate #f(0 -18.30127 -18.30127))
(send x :rotate -0.523599 :x)
(send x :worldcoords)

(setq c (list (send x :constraint ana)))
(setq m (constrained-motion c))
(setq f (constrained-force m))

(hidd x ana)
(draw-constraint c)
```

```
(draw-motion m)
```

拘束の例を次の図で示す。図の小さな矢印は，ペグに対する拘束を示す。

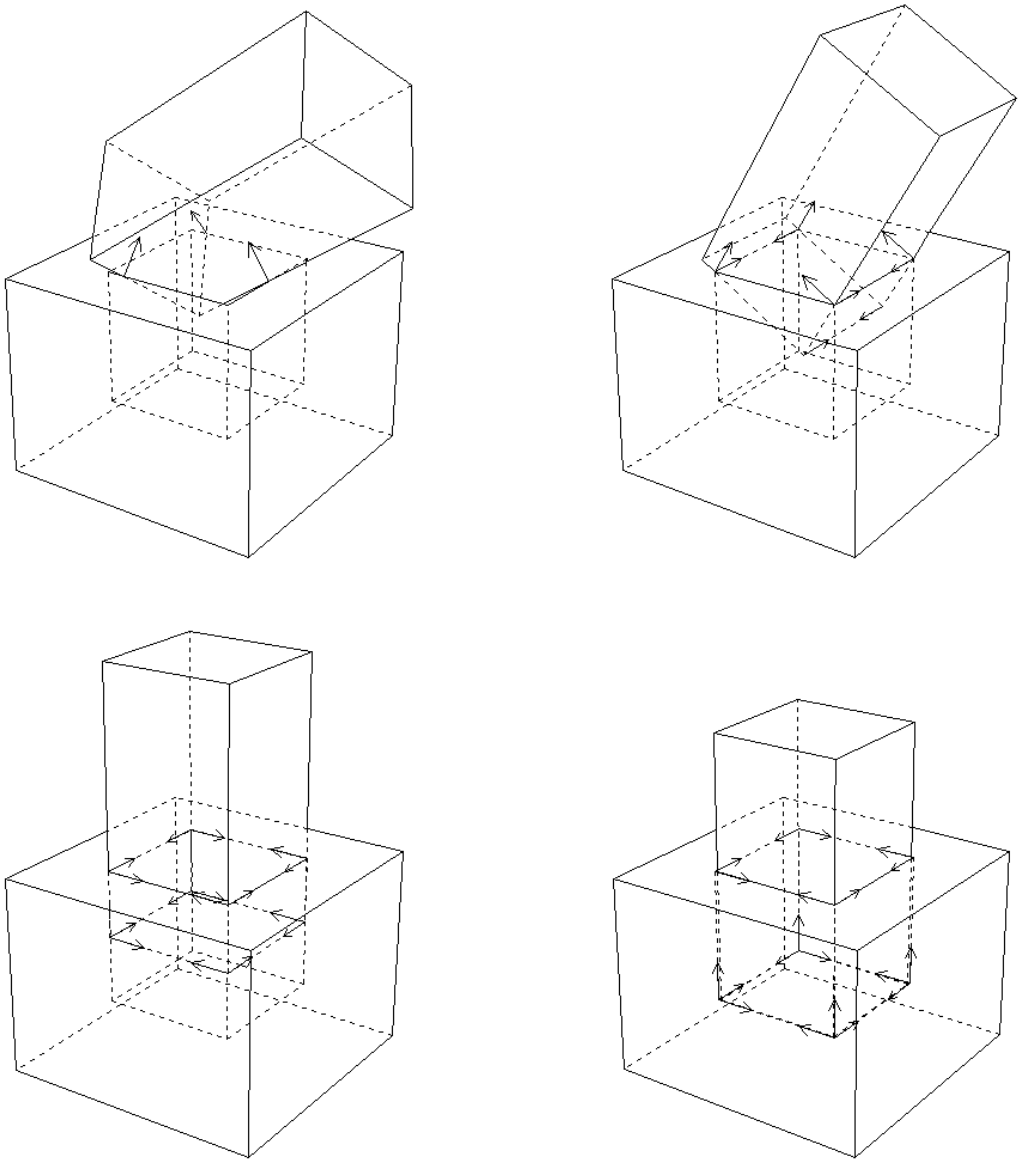

図 11: Constraints for a peg in a hole.

ペグを穴に入れる作業において取り得る動作の例を次の図で示す。この例は，上記のプログラムと一致している。

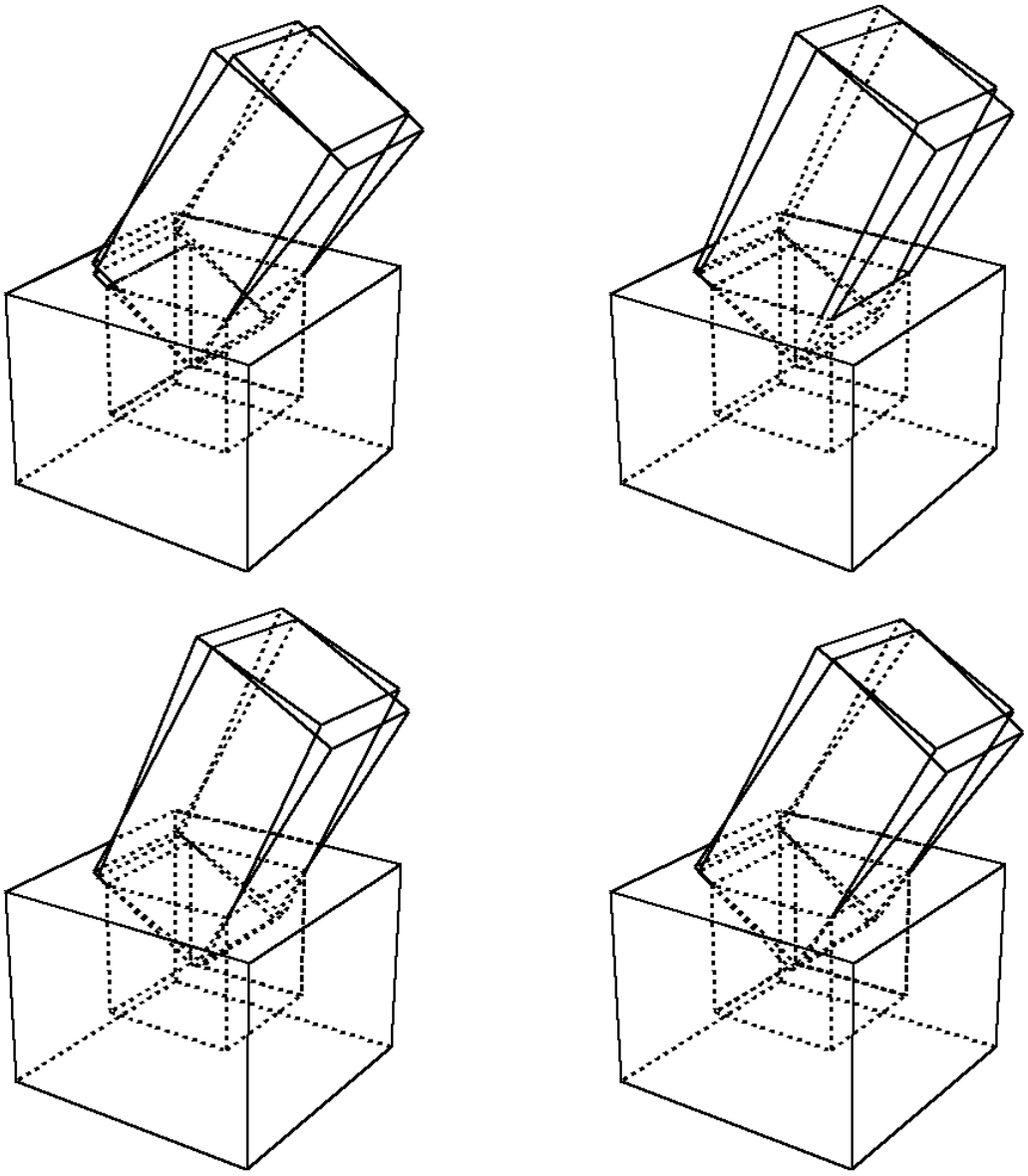

図 12: Possible motions of a peg in a hole

14.9 多角形の Voronoi Diagram

著者: *Philippe PIGNON,* **電総研ゲスト研究者**

このプログラムは，Common Lisp で書かれている。”A sweepline algorithm for Voronoi diagrams”, Proceedings of the 2nd Annual ACM symposium on computational geometry, 1986, 313-322. を手法として用い、多角形の場合への応用を行った。これは，サンプルプログラム付きの簡単な説明である。このプログラムは，ETL の Euslisp 環境で書かれているため，画像への出力もサポートしている。どの Common Lisp 上でも使用することはできるが，`utilities.l` で与えられている画像への関数を自分のディスプレイ環境へ合うように書き換える必要がある。この節の最後にその関数を示す。

目的: 多角形の集合の voronoi diagram の計算を行う。語彙を理解するために上記の文献を読んで、使用してください。ここでは、このプログラムに対する説明をしません。

入力: 多角形のリストと囲むための枠は，次のように定義する。

```
DATA= (
       (x11 y11 x12 y12 x13 y13 ...) first polygon,
                                     counterclocwise enumeration of vertices
       (x21 y21 x22 y22 x23 y23 ...) second polygon
              ...
       (xn1 yn1 xn2 yn2 xn3 yn3 ...) nth polygon

       (xf1 yf1 xf2 yf2 xf3 yf3 xf4 yf4) enclosing frame
      )
```

囲む枠は，**DATA** 内のどの位置にも配置することができる。また，内部と外部が矛盾しないように時計方向の順番でなければならない。多角形は交差の無い簡単な図形でなければならない。一直線あるいは平坦なエッジは受け付けない。独立した点あるいは線分も受け付けない。

出力: ***diagram***:2 重に接続されたエッジリストのリスト (utilities.l ファイルを参照) を返す。それぞれのエッジは，symbol であり，次に示すような field を含む property-list を持っている。

```
(start <pointer to a vertex>)
       (end <pointer to a vertex>)
       (pred <pointer to an edge>)
       (succ <pointer to an edge>)
       (left <pointer to a site>)
       (right <pointer to a site>)
       (type <:endpoint or :point-point or :segment-segment or :point-segment>)
       (outflag <t or nil>)
```

vertex は，symbol で”pos”field を含む property-list を持つ。この field は，cons*(x,y)* を含み，*vertex* の平面座標を示す。*pred* と *succ* の field は，decl 形式にしたがって反時計方向の前者と後者を与える (Shamos と Preparata の, Computational Geometry: An introduction, 1985, pp 15-17 を参照)。*site* も symbol であり，関連した情報を含む property-list を持つ。*site* は，元の入力データを記述しており，多角形の頂点である point あるいは多角形のエッジである segment を持つ。

type は，2 等分線の中点であり，それを分割する *site* の型より決定される。規約により，外側は start-end エッジの右側である。voronoi diagram は，2 等分線の内部と同様に外側を計算する。必要とする outflag を保つために outflag をソートする。

サンプル: サンプルプログラムを実行するためには，以下のようなステップを実施してください。

1. 自分の環境に以下のプログラムをコピーする。
   utilities.l　　幾何学ユーティリティ関数と eusx の画像出力関数
   polygonalvoronoi.l　プログラム本体
   testdata.l　　上記の書式によるデモデータ
2. もし，Euslisp を使用しないなら，命令にしたがって `utilities.l` を書き換え，”compatibility package”を修正する。。
3. 以下の 3 つのファイルをコンパイルしてロードするか、あるいはそのままロードする。
   utilities.l
   polygonalvoronoi.l
   testdata.l　　上記の書式によるデモデータを含んでいる。
4. (pv demoworld) でデモデータ上でプログラムが実行される。グローバル変数***diagram***には，voronoi diagram の 2 等分線が含まれている。

eusx(Xwindow インターフェースを持つ Euslisp) のもとでは，以下の命令で diagram の結果を画面上に表示することができる。

```
(make-display)          ;;Initializes the *display* window object
(dps demoworld *thick*) ;; Shows original data in thick lines
(dbs *diagram*)         ;; Shows the result
```

pv *data* [関数]

上記の書式で書かれた *data* から多角形の voronoi diagram を計算する。

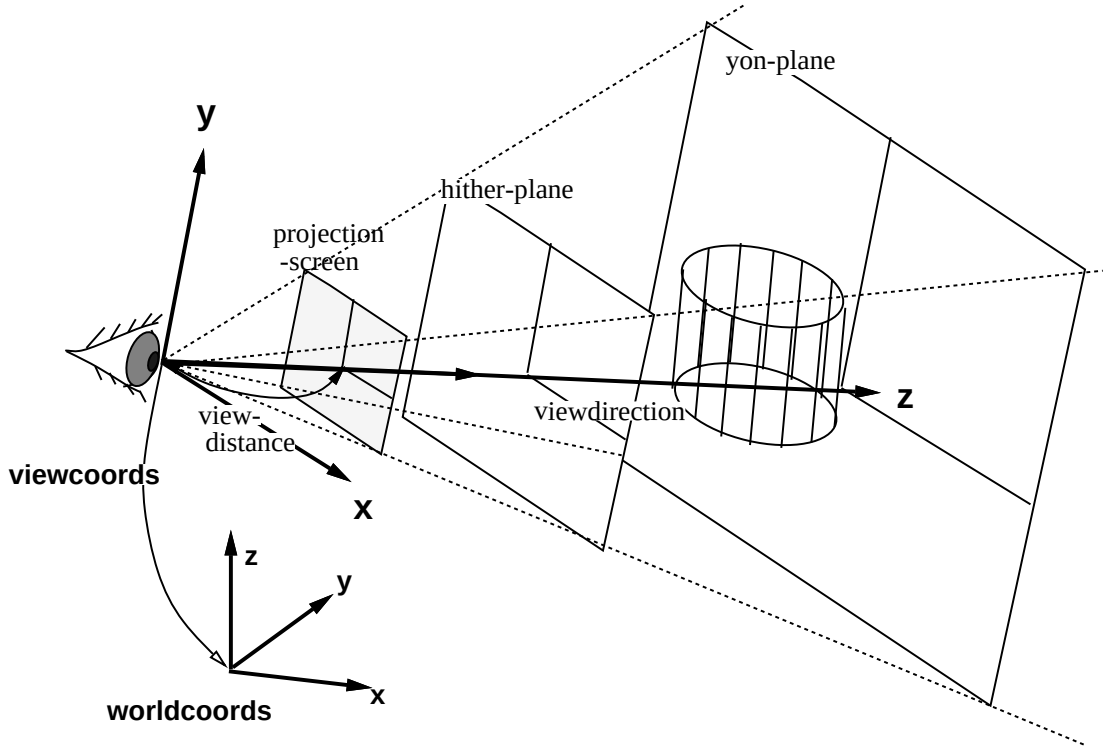

図 13: viewing 座標系と投影画面

15 視界とグラフィックス

15.1 視界 (viewing)

viewing オブジェクトは、viewing 座標系を処理する。この座標系の原点は仮想カメラの位置に置かれる。-*z* 軸方向がオブジェクトの視線方向で、xy 平面が投影画面である。**viewing** が **cascaded-coords** クラスを継承するので、**:translate** や**:rotate** や**:transform** のような座標変換メッセージを受け付ける。また、**cascaded-coords** から得られる他のオブジェクトを張り付けることができる。したがって、移動物体上のカメラシステムのシミュレーションができる。**viewing** の主な目的は、ワールド座標系で表現されるベクトルをカメラ座標系に変換することである。変換は、一般の座標変換に対して逆方向で与えられる。このローカル座標系内のベクトルはワールド座標系における表現に変換される。したがって、**viewing** は `viewcoords` スロットに逆変換された左手系変換を持つ。このスロットは、viewing 座標系として普通参照される。

viewing [クラス]

:super **cascaded-coords**
:slots (viewcoords)

viewing 変換を定義する。

:viewpoint [メソッド]

この **viewing** の原点のベクトル位置を返す。

:view-direction [メソッド]

viewing の原点から画面の中心までのベクトルを返す。これは、viewing 座標系の z 軸方向である。

:view-up [メソッド]

ワールド座標系におけるこの **viewing** の y 軸ベクトルを返す。y 軸は、**viewport** の上方である。

:view-right [メソッド]

ワールド座標系におけるこの **viewing** の x 軸ベクトルを返す。x 軸は、**viewport** の水平右方向である。

:look *from &optional (to #f(0 0 0))* [メソッド]

:look は、その目が *from* に位置されており、*to* の位置を見ているとして viewing 座標系を設定する。

:init *&key :target #f(0 0 0)* [メソッド]
:view-direction nil
:view-up #f(0.0 0.0 1.0))
:view-right nil
&allow-other-keys

viewing は、**cascaded-coords** を継承するので、*:pos* や*:rot* や*:euler* や*:rpy* などの*:init* のパラメータはすべて viewing 座標系の位置や姿勢を指定することに使用できる。しかしながら、viewing の*:init* は回転を決定する簡単な方法を持っている。もし、*:target* だけが与えられたとき、視線方向は視点から *target* 位置の方向に決定され、*:view-right* ベクトルはワールド座標系の xy 平面に平行な x 軸に決定される。*:view-direction* を*:target* の代わりに指定しても同じ様な効果を得られる。もし、*:view-up* または *:view-right* パラメータを*:target* あるいは*:view-direction* に加えて指定するならば、3 つの回転パラメータをすべて自分自身で決定することができる。

15.2 投影

parallel-projection と **perspective-projection** クラスは、投影変換を処理する。この変換は 4X4 の行列で表現される。すなわち、変換は 3 次元の同次座標系で与えられる。**projection** クラスは、両方のクラスの抽象クラスである。これらの投影クラスは、viewing クラスを継承しているので、2 つの座標変換（ワールド座標から viewing 座標系への変換と投影変換）を同時に実行することができる。3D ベクトルと `:project3` メッセージを投影オブジェクトに送ることにより、4 要素の実数ベクトル返す。**homo2normal** 関数は、この同次ベクトルを標準のベクトル表現に変換するために使用される。その結果は、標準デバイス座標系 (NDC) と呼ばれる座標系上に表現されるベクトルである。その中で、見えるベクトルはそれぞれの x,y,z 次元において-1 から 1 までの範囲で表される。ロボット世界の本当のカメラをシミュレートするために、**perspective-projection** は **parallel-projection** よりも多く使用される。**perspective-projection** は、定義されているパラメータが少し多い。`screenx` と `screeny` は、見える物体が投影される viewing 平面の上の window の大きさで、大きな画面と広い空間が投影される。`viewdistance` は、視点と view 平面との距離を定義しているが、視角にも関係する。`viewdistance` を大きくすると、view 平面の window に狭い範囲が投影される。`hither` と `yon` パラメータは、クリップする平面の前面と後面の距離を定義する。これら 2 つの平面の外側に位置するオブジェクトは、クリップから除外される。実際に、このクリップ処理は **viewport** オブジェクトによって実現されている。

projection [クラス]

:super **viewing**
:slots (screenx screeny hither yon projection-matrix)

4x4 行列であらわされる投影変換を定義する。

:projection *&optional pmat* [メソッド]

もし、*pmat* が与えられたならば、`projection-matrix` のスロットに設定する。**:projection** は、現在の 4x4 投影行列を返す。

:project *vec* [メソッド]

vec は、4 要素を持つ 3 次元同次ベクトルである。*vec* は、投影行列により変換される。そして、変換された結果である同次表現が返される。

:project3 *vec* [メソッド]

vec は、標準の 3D ベクトル。*vec* は、投影行列により同次化され変換される。そして、変換された結果である同次表現が返される。

:view *vec* [メソッド]

vec に viewing 変換と投影変換を連続的に適用する。そして、変換された結果である同次表現が返される。

:screen *xsize &optional (ysize xsize))* [メソッド]

viewing 画面の大きさを変える。大きくすると、広い view が得られる。

:hither *depth-to-front-clip-plane* [メソッド]

視点からクリップ前面までの距離を決定する。このクリップ前面よりも前にあるオブジェクトはクリップから除外される。

:yon *depth-to-back-clip-plane* [メソッド]

視点からクリップ後面までの距離を変える。このクリップ後面よりも後ろにあるオブジェクトはクリップから除外される。

:aspect *&oiptional ratio* [メソッド]

アスペクト比は、screen-y と screen-x との比である。もし、*ratio* が与えられたならば、アスペクト比は変えられ、screen-y は screen-x * *ratio* に設定される。**:aspect** は、現在のアスペクト比を返す。

:init *&key* *:hither* *100.0* [メソッド]
:yon *1000.0*
:aspect *1.0*
:screen *100.0*
:screen-x *screen*
:screen-y *(* screen-x aspect)*
&allow-other-keys

viewing と **projection** を初期化する。

parallel-viewing [クラス]

:super **projection**
:slots ()

平行投影を定義する。**hid**(陰線消去関数) は平行投影では扱うことが出来ない。

:make-projection [メソッド]

perspective-viewing [クラス]

:super **projection**
:slots (viewdistance)

透視投影変換を定義する。

:make-projection [メソッド]

:ray *u v* [メソッド]
視点から正規化画面の上にある *(u,v)* への単位方向ベクトルを返す。

:viewdistance *&optional vd* [メソッド]
`viewdistance` は、視点から画面迄の距離である。もし、*vd* が与えられたならば、`viewdistance` に設定される。`viewdistance` は、カメラの焦点距離と一致する。*vd* を大きくすれば、ズームアップされた view を得ることができる。**:viewdistance** は、現在の `viewdistance` を返す。

:view-angle *&optional ang* [メソッド]
画面の対角線を見込む角度が *ang* ラジアンであるように画面の大きさを設定する。20 度 (約 0.4 ラジアン) から 50 度 (約 0.9 ラジアン) までの角度が自然な透視 view を生成することができる。角度を大きくすると歪んだ view を生成する。そして、狭くすると直角 (平行)viewing のような平坦な view が生成される。**:view-angle** は、現在の視角あるいは新しい視角をラジアンで返す。

:zoom *&optional scale* [メソッド]
もし、*scale* が与えられたならば、画面は *scale* によって現在の大きさを相対的に変化させる（`viewdistance` は変化しない)。もし、*scale* に 0.5 を与えるならば、以前の view より 2 倍広い view を得られる。**:zoom** は、新しい視角をラジアンで返す。

:lookaround *alfa beta* [メソッド]
視点を移動し回転させる。回転中心は、視線の上で `hither` 平面と `yon` 平面の中間点に与えられる。viewing 座標系は、ワールド座標系の z 軸回りに *alfa* ラジアン回転し、ローカル座標系の x 軸回りに *beta* ラジアン回転される。**:lookaround** は、**viewing** の中心にあるオブジェクト回りに視線を動かすことができる。

:look-body *bodies* [メソッド]
視線、画面の大きさおよび hither/yon をすべての *bodies* に適合する viewport となるよう変える。視点は変化しない。視線は、すべての *bodies* の bounding box の中心を通る視線から選択される。

:init *&key (:viewdistance 100.0) &allow-other-keys* [メソッド]

15.3 Viewport

viewport クラスは、正規デバイス座標系 (NDC) の中の 3 次元 viewport のクリップを実行する。そして、デバイスに依存する座標系に結果を作る。**viewport** は、画面上の見える四角領域の境界表現である。**viewport** の物理的な大きさ（x 軸と y 軸方向のドット数）は、**:init** メッセージの中の *:width* と *:height* との引き数で与えられなければならない。*:xcenter* と *:ycenter* 引き数は、viewport の物理的な位置を決定する。画面の原点からのそれぞれの次元が絶対的に与えられているテクトロニクス 4014 のような基本的なディスプレイデバイスを使っているとき、これら 2 つのパラメータは、実際に画面の上にオブジェクトを描く位置を決定する。もし、位置が親 window から相対的に決まる Xwindow のような精巧なディスプレイデバイスを使っているなら、viewport を動かすために viewport のパラメータを変える必要はない。なぜなら、これらのパラメータは、実際のディスプレイ位置に依存しないからである。。

viewport クラスは、四角領域の左下を viewport の原点と仮定している。そして、y 軸は上方向に伸びてい

るとする。しかし、多くの window システムやディスプレイデバイスでは原点を左上とし、y 軸が下方向に伸びているとしている。この問題を回避するために、*:height* パラメータに負の値を与えればよい。

homo-viewport-clip *v1 v2* [関数]

v1 と *v2* は、4 要素を持つ同次ベクトルであって、3 次元空間の線分として表現される。その線分は、$x = -1, x = 1, y = -1, y = 1, z = 0, z = 1$ の境界でクリップされる。そして、2 つのベクトルのリストを返す。もし、その線分が **viewport** の外側に完全に置かれているならば、NIL を返す。

viewport [クラス]

:super **coordinates**
:slots

viewport 変換は、デバイスで指定される座標系に NDC（正規化デバイス座標系）を作る。**coordinates** クラスを継承しているため、**viewport** はサイズと投影画面の相対位置を定義している。

:xcenter *&optional xcenter* [メソッド]

この **viewport** の x 軸の中心を返す。もし、*xcenter* が与えられていれば、設定を行う。

:ycenter *&optional ycenter* [メソッド]

この **viewport** の y 軸の中心を返す。

:size *&optional size* [メソッド]

この **viewport** の x 軸と y 軸方向の大きさのリストを返す。

:width *&optional width* [メソッド]

この **viewport** の幅を *width* に設定する。

:height *&optional height* [メソッド]

この **viewport** の高さを *height* に設定する。

:screen-point-to-ndc *p* [メソッド]

p は、物理的画面の中の位置を表現する実数ベクトルである。*p* は、正規化デバイス座標系 (NDC) の中での表現に変換される。

:ndc-point-to-screen *p* [メソッド]

この **viewport** の NDC 表現である *p* を画面の物理的位置に変換する。

:ndc-line-to-screen *p1 p2 &optional (do-clip t)* [メソッド]

2 つの 3 次元ベクトル *p1* と *p2* は、NDC の中の線分を定義する。これらの 2 つの端点は、画面空間の表現に変換される。もし、*do-clip* が non-NIL なら、その線分はクリップされる。

:init *&key (:xcenter 100) (:ycenter 100) (:size 100) (width 100) (height 100)* [メソッド]

新しい **viewport** オブジェクトを作る。

15.4 Viewer

画面の上に描画するためには、4 つのオブジェクトが必要である。1 つは描かれたオブジェクト、2 つは viewing 座標系と投影で定義される **viewing**、3 つは NDC の中でのクリップ処理のための **viewport** と NDC から物理的画面座標系への変換、4 つは物理的ディスプレイデバイスの上に描画関数を実行する **viewsurface**。**viewer** オブジェクトは、**viewing** と **viewport** と **viewsurface** オブジェクトを持ち、座標系変換を連続的に制御す

る。15.5 節に記述される **draw** と **hid** 関数は **viewer** のインスタンスを使用する。

viewer [クラス]

:super	**object**
:slots	(eye :type viewint)
	(port :type viewport)
	(surface :type viewsurface)

viewing から viewport を経由して viewsurface へ移る Cascaded Coordinates の変換を定義する。

:viewing *&rest msg* [メソッド]
もし、*msg* が与えられたならば、*msg* は **viewing**(eye) オブジェクトに送られる。そうでなければ、**viewing**(eye) オブジェクトが返される。

:viewport *&rest msg* [メソッド]
もし、*msg* が与えられたならば、*msg* は **viewport**(port) オブジェクトに送られる。そうでなければ、**viewport**(port) オブジェクトが返される。

:viewsurface *&rest msg* [メソッド]
もし、*msg* が与えられたならば、*msg* は **viewsurface**(surface) オブジェクトに送られる。そうでなければ、**viewsurface**(surface) オブジェクトが返される。

:adjust-viewport [メソッド]
viewsurface の大きさが変えられたとき、**:adjust-viewport** は port に固有のメッセージを送ることにより **viewport** の変換を変える。

:resize *width height* [メソッド]
viewsurface に:resize メッセージを送り、**viewport** に:size メッセージを送ることにより viewsurface の大きさを変える。

:draw-line-ndc *p1 p2 &optional (do-clip t)* [メソッド]
NDC の中に定義される 2 つの端点 *p1,p2* を結ぶ線を描く。

:draw-polyline-ndc *polylines* [メソッド]
NDC の中に定義される端点を結ぶ多角形を描く。

:draw-star-ndc *center &optional (size 0.01)* [メソッド]
NDC の中に十字マークを描く。

:draw-box-ndc *low-left up-right* [メソッド]
NDC の中に四角形を描く。

:draw-arc-ndc *point width height angle1 angle2 &optional color* [メソッド]
NDC の中に円弧を描く。この **viewer** に結び付く **viewsurface** オブジェクトは、**:arc** メッセージを受けなければならない。

:draw-fill-arc-ndc *point width height angle1 angle2 &optional color* [メソッド]
NDC の中に塗り潰し円弧を描く。

:draw-string-ndc *position string &optional color* [メソッド]
NDC の中に定義される *position* に *string* を描く。

:draw-image-string-ndc *position string &optional color* [メソッド]

:draw-rectangle-ndc *position width height &optional color* [メソッド]

:draw-fill-rectangle-ndc *point width height &optional color* [メソッド]

:draw-line *p1 p2 &optional (do-clip t)* [メソッド]
ワールド座標系に定義される 2 つの端点 *p1,p2* を結ぶ線を描く。

:draw-star *position &optional (size 0.01)* [メソッド]
ワールド座標系の *position* 位置に十字マークを描く。

:draw-box *center &optional (size 0.01)* [メソッド]
ワールド座標系の *center* に四角形を描く。

:draw-arrow *p1 p2* [メソッド]
p1 から *p2* へ向けての矢印を描く。

:draw-edge *edge* [メソッド]

:draw-edge-image *edge-image* [メソッド]

:draw-faces *face-list &optional (normal-clip nil)* [メソッド]

:draw-body *body &optional (normal-clip nil)* [メソッド]

:draw-axis *coordinates &optional size* [メソッド]
coordinates で定義される軸を *size* の長さで描く。

:draw *&rest things* [メソッド]
3 次元の幾何学オブジェクトを描く。もし、オブジェクトが 3 次元ベクトルならば、その位置に小さな十字マークを描く。もし、3 次元ベクトルのリストであれば、多角形を描く。もし、*thing* が `:draw` メッセージを受けたならば、この **viewer** を引き数としてそのメソッドが呼び出される。もし、オブジェクトが `:drawners` メソッドを定義しているならば、**:draw** メッセージは`:drawners` の結果に送られる。`line`, `edge`, `polygon`, `face` および `body` オブジェクトは、**viewer** に定義されている**:draw-xxx**(xxx にそのオブジェクトのクラス名が入る) メソッドによって描かれる。

:erase *&rest things* [メソッド]
背景色で *things* を描く。

:init *&key :viewing :viewport :viewsurface* [メソッド]
viewing, viewport および *viewsurface* をこの **viewer** のスロット `eye`, `port` と `surface` に設定する。

view *&key (:size 500) (:width size) (:height size)* [関数]
(:x 100) (:y 100)
(:title "eusx")

(:border-width 3)
(:background 0)
(:viewpoint #f(300 200 100)) (:target #f(0 0 0))
(:viewdistance 5.0) (:hither 100.0) (:yon 10000.0)
(:screen 1.0) (:screen-x screen) (:screen-y screen)
(:xcenter 500) (:ycenter 400)

新しい **viewer** を作り、***viewer***リストに置く。

15.5 描画

draw *[viewer] &rest thing* [関数]

viewer に *thing* を描く。*thing* は、座標系、立体、面、エッジ、ベクトル、2 つのベクトルのリストのどれでも可能である。(progn (view) (draw (make-cube 10 20 30))) は、xwindow に立方体を描く。

draw-axis *[viewer] [size] &rest thing* [関数]

viewer の中に *thing* の座標系の軸を *size* の長さで描く。*thing* は、座標系から得られるどのオブジェクトでも可能である。

draw-arrow *p1 p2* [関数]

viewer に *p1* から *p2* に向かう矢印を描く。

hid *[viewer] &rest thing* [関数]

viewer に陰線処理された画像を描く。*thing* は、face または body が可能である。

hidd *[viewer] &rest thing* [関数]

陰線を点線で描くことを除いて、**hid** と同じである。

hid2 *body-list viewing* [関数]

edge-image オブジェクトで表現される陰線処理画像を生成する。その結果は***hid***に置かれる。

render *&key :bodies :faces (:viewer *viewer*) (:lights *light-sources*) (colormap *render-colormap*) (y 1.0)* [関数]

bodies と *faces* にレイトレーシングを行い、陰面消去した画像を生成する。**viewing, viewport** および **viewsurface は**、*viewer* から得られる。*lights* は、light-source（光源）オブジェクトのリストである。*colormap* は、Xwindow の **colormap** オブジェクトである。それぞれの *bodies* と *faces* は、割り当てられるカラー属性を持たなければならない。*colormap* に定義されているカラー LUT の名前を :color メッセージで送ることによりカラー属性を設定できる。一般にこの関数は、Xlib 環境下でのみ働く。demo/renderdemo.l のサンプルプログラムを見ること。

make-light-source *pos &optional (intensity 1.0)* [関数]

pos の位置に光源オブジェクトを作る。*intensity* は、デフォルトの光の強さを増す拡大比である。もっと正確に強さを決定するためには、光源の:intensity メソッドを使用する。

tektro *file &rest forms* [マクロ]

*tektro-port*ストリームのために *file* をオープンし、*forms* を評価する。これは、tektro 描画の出力を直接 *file* に書き込むために使用される。

kdraw *file &rest forms* [マクロ]

kdraw は、kdraw または idraw で読み込めるポストスクリプトファイルを生成するためのマクロ命令である。**kdraw** は、:output モードで *file* をオープンし、*viewer*を置き換えるための kdraw-viewsurface と viewport を作り、*forms* を評価する。それぞれの *forms* は、draw や hid のような描画関数のどれかを呼び出す。これらの *forms* からの描画メッセージは、kdraw-viewsurface に直接出力される。この出力は idraw や kdraw で認識できるポストスクリプト表現にメッセージを変換する。そして、*file* に蓄積する。idraw または kdraw が呼び出され *file* がオープンされたとき、EusViewer window に書いたものと同一の図形を見ることができる。その図形は、idraw の機能で変更することができる。そして、最終描画は epsfile 環境を用いることにより LaTeX ドキュメントに組み込むことができる。この機能は、”llib/kdraw.l”のファイルに記述されている。

pictdraw *file &rest forms* [マクロ]

pictdraw は、Macintosh の PICT フォーマットで画像ファイルを生成するためのマクロである。**pictdraw** は、*file* を`:output` モードでオープンし、pictdraw-viewsurface を作り、`*viewer*`の viewport に置き換え、*forms* を評価する。*forms* は、それぞれ `draw` あるいは `hid` のような描画関数のどれかを呼び出すものである。これらの書式からの描画メッセージは、`kdraw-viewsurface` に直接出力された後、PICT フォーマットへのメッセージに変換され, *file* へ蓄積される。この PICT フォーマットは、Macintosh の `macdraw` や `teachtext` で認識することができる。

hls2rgb *hue lightness saturation &optional (range 255)* [関数]

HLS(Hue, Lightness, Saturation) で表現される色を、RGB 表現に変換する。HLS は、しばしば HSL として参照される。*hue* は、rainbow circle(0 から 360) の色で表現される。0 が赤で 45 が黄で 120 が緑で 240 が青で 270 が紫そして 360 が再び赤となる。*lightness* は、0.0 から 0.1 の値を持ち、黒から白までの明るさを表現する。*lightness* が 0 のときは、*hue* や *saturation* にかかわらず黒となる。そして、*lightness* が 1 のときは、白となる。*saturation* は、0.0 から 1.0 までの値を持ち、色の強さを表現する。*saturation* の値が大きいと鮮明な色調を生成し、小さい値は弱く濁った色調を生成する。*range* は、RGB 値の限界を示す。もし、それぞれの色に 8 ビットの値が割り当てられているカラーディスプレイを使っているならば、*range* は 255 とすべきである。もし、RGB に 16 ビットの整数が仮想的に割り当てられている Xwindow を使っているならば、*range* は 65536 とすべきである。HSV と HLS との違いに注意すること。HLS では、鮮明な (rainbow) 色は `lightness=0.5` で定義されている。

rgb2hls *red green blue &optional (range 255)* [関数]

RGB の色表現を、HLS 表現に変換する。

15.6 アニメーション

EusLisp のアニメーションは、グラフィックアクセラレータを持たない普通のワークステーション上での擬似リアルタイムグラフィックス機能を備えている。その基本的な考え方は、長い時間かかって生成された 1 連の画像を高速に再表示することである。画像は 2 つの方法で保存される。1 つは、完全なピクセル画像を持つたくさんの Xwindow pixmap を保存する。もう 1 つは、陰線処理で得られる線分データを保存する。前者は、高速で陰面処理された画像を表示するための方法であるが、長いアニメーションではたくさんのメモリーを X server に要求するため適さない。後者は、メモリーが少なくて済み、データをディスクに蓄積するのに適する。しかし、線分の数が増加したならば、性能を悪化させる。

他の方法として、描かれるオブジェクトの構成を得て、***viewer***に描画を生成する関数をユーザーが作ることもできる。**pixmap-animation** は、*count* 引数で指定された数と同じ回数この関数を呼び出す。それぞれの呼び出し後、Xwindow と仮定される `viewsurface` の内容は、新しく作られた Xwindow pixmap にコピーされる。これらの pixmap は、**playback-pixmap** で再表示される。同様に、**hid-lines-animation** は **hid** の結果から見える線分を抜き出し、リストに蓄積する。そのリストは、**playback-hid-lines** によって再表示される。

以下に示す関数は、"llib/animation.l" に定義されており、"llib/animdemo.l" の中には ETA3 マニピュレータのモデルに関して **hid-lines-animation** を用いたアニメーションのサンプルプログラムを含んでいる。

pixmap-animation *count &rest forms* [マクロ]

forms は、*count* 回評価される。それぞれの評価後、`*viewsurface*`の内容は新しい pixmap にコピーされる。*count* 枚の **pixmap** のリストが、返される。

playback-pixmaps *pixmaps &optional (surf *viewsurface*)* [関数]

pixmaps リストのなかの **pixmap** はそれぞれ、*surf* に連続的にコピーされる。

hid-lines-animation *count &rest forms* [マクロ]

hid への呼び出しを含む *forms* が *count* 回評価される。それぞれの評価後、***hid***が持つ **hid** の結果は検索され、見える線分は 2 点一組のリストの形で集められる。*count* 長さのリストが返される。

playback-hid-lines *lines &optional (view *viewer*)* [関数]

lines は、2 点一組のリストである。*view* の上に線分を連続的に描く。他の pixmap を割り当てるときにフリッカーフリーアニメーションを生成するために 2 重バッファ技法が使用される。

list-visible-segments *hid-result* [関数]

hid-result の edge 画像のリストから見える線分を集める。

16 Xwindow インターフェース

Euslisp 上の Xwindow インターフェースは、'eusx' という名前で Euslisp が呼び出されたとき、実行可能となる。[3]eusx は、起動時に環境変数"DISPLAY"を参照して Xserver に接続を試みるため、自分の Xsever に環境変数"DISPLAY"が正しく設定されていなければならない。

Euslisp には、次の 3 つのレベルの Xwindow インターフェースが定義されている。(1) Xlib 関数, (2) Xlib クラスと (3) XToolKit クラスである。この節と次の XToolKit の節に記述されている全ての Xwindow 関数は、"X"パッケージの中に含まれている。それらの関数名は、元の Xlib 関数名から最初の"X"を省いき、全ての文字を大文字に変更したものとなっている。例えば XdefaultGC は X:DEFAULTGC と名付けられていて、X:XDEFAULTGC ではない。

Xlib 関数は、Xwindow システムへの低レベルインターフェースで、foreign 関数として定義されている。これらの Xlib 関数は、パラメータの型チェックあるいはパラメータの数のチェックを実行していないため、十分注意して使用しなければならない。例えば、すべての Xlib の呼び出しにおいて Xserver との接続を確認するために x:*display*を引き数として要求する。もし、指定し忘れたならば、Xlib はエラーを通知し、そのプロセスは死ぬ。このような不便を避け、オブジェクト指向のインターフェースを作るために、2 番めのレベルのインターフェースである Xlib クラスが備わっている。この節では、この 2 番めのレベルのインターフェースに焦点を当てる。XToolKit と呼ばれるもっと高レベルの Xwindow ライブラリは、次節で説明されている。

この節に記述されているクラスは、以下の継承の階層を持っている。

```
propertied-object
   viewsurface
      x:xobject
         x:gcontext
         x:xdrawable
             x:xpixmap
             x:xwindow
   colormap
```

16.1 Xlib のグローバル変数とその他関数

x:*display* [変数]
X の display ID（整数）

x:*root* [変数]
デフォルトの root window オブジェクト

x:*screen* [変数]
デフォルトの screen ID（整数）

x:*visual* [変数]
デフォルトの visual ID（整数）

x:*blackpixel* [変数]
黒色の pixel 値 = 1

x:*whitepixel* [変数]

[3]eusx は、eus へのシンボリックリンクである。

白色の pixel 値 = 0

x:*fg-pixel* [変数]
window 作成時に参照されるデフォルトの文字色の pixel 値、ふつう`*blackpixel*`。

x:*bg-pixel* [変数]
window 作成時に参照される背景色の pixel 値、ふつう`*whitepixel`。

x:*color-map* [変数]
システムのデフォルトカラーマップ

x:*defaultGC* [変数]
pixmap 生成時に参照されるデフォルト gcontext

x:*whitegc* [変数]
文字色が白色の gcontext

x:*blackgc* [変数]
文字色が黒色の gcontext

gray-pixmap [変数]
`(make-gray-pixmap 0.5)` の結果。

gray25-pixmap [変数]
1/4 のピクセルが`*fg-pixel*`であり、3/4 が`*bg-pixel*`である 16x16 の pixmap。

gray50-pixmap [変数]
1/2 のピクセルが`*fg-pixel*`である 16x16 の pixmap。

gray75-pixmap [変数]
3/4 のピクセルが黒色である 16x16 の pixmap。

gray25-gc [変数]
`*gray25-pixmap*`から作る 25%のグレー GC。

gray50-gc [変数]
`*gray50-pixmap*`から作る 50%のグレー GC。

gray75-gc [変数]
`*gray75-pixmap*`から作る 75%のグレー GC。

gray [変数]
`"#b0b0b0"`

bisque1 [変数]
`"#ffe4c4"`

bisque2 [変数]
`"#eed5b7"`

bisque3 [変数]
`"#cdb79e"`

lightblue2 [変数]

"#b2dfee"

lightpink1 [変数]

"#ffaeb9"

maroon [変数]

"#b03060"

max-intensity [変数]

65535

font-cour8 [変数]

(font-id "*-courier-medium-r-*-8-*")

font-cour10 [変数]

(font-id "*-courier-medium-r-*-10-*")

font-cour12 [変数]

(font-id "*-courier-medium-r-*-12-*")

font-cour14 [変数]

(font-id "*-courier-medium-r-*-14-*")

font-cour18 [変数]

(font-id "*-courier-medium-r-*-18-*")

font-courb12 [変数]

(font-id "*-courier-bold-r-*-12-*")

font-courb14 [変数]

(font-id "*-courier-bold-r-*-14-*")

font-courb18 [変数]

(font-id "*-courier-bold-r-*-18-*")

font-helvetica-12 [変数]

(font-id "*-Helvetica-Medium-R-Normal-*-12-*")

font-lucidasans-bold-12 [変数]

(font-id "lucidasans-bold-12")

font-lucidasans-bold-14 [変数]

(font-id "lucidasans-bold-14")

font-helvetica-bold-12 [変数]

(font-id "*-Helvetica-Bold-R-Normal-*-12-")

font-a14 [変数]

(font-id "*-fixed-medium-r-normal-*-14-*")

x:*xwindows* [変数]

Euslisp による子 window を含んだ全ての window のリストを保持する。

x:*xwindow-hash-tab* [変数]

drawable ID から xwindow オブジェクトを探すためのハッシュテーブル。`x:nextevent` で得られるイベント構造は window ID であるため、`x:window-main-loop` はこのテーブルを使用して一致する xwindow オブジェクトを知るために `x:event-window` を呼び出す。

xflush [関数]

Xlib のコマンドバッファに保有するコマンドをすべて Xserver に送る。Xlib バッファが Xserver に出力するため、Xserver に発行されたコマンドは、すぐに実行されない。これは、ネットワークの渋滞およびプロセスの切替え頻度を減少させるために必要である。コマンドの効果を見るためにコマンドバッファを掃き出す方法として、**xflush** を使用するかあるいは**:flush** メッセージを xwindow オブジェクトに送る。

16.2 Xwindow

Xobject [クラス]

:super **geometry:viewsurface**
:slots

すべての Xwindow に関連するクラスの共通のスーパークラスである。現在、スロットもメソッドも定義されていない。

Xdrawable [クラス]

```
:super   Xobject
:slots   (drawable      ; drawable ID
          gcon          ; this drawable's default graphic context object
          bg-color      ; background color
          width height  ; horizontal and vertical dimensions in dots
```

Xdrawable は、線分や文字列のようなグラフィックオブジェクトを描くための四角領域を定義する。**Xdrawable** は、xwindow や xpixmap のための共通メソッドを定義するための抽象クラスであり、このクラスのインスタンスは何の効果も持っていない。

:init *id* [メソッド]

id は、この drawable の ID として `drawable` スロットに設定される。新しい GC(graphic context) が生成され、この drawable オブジェクトのデフォルト GC として `gcon` に設定される。

:drawable [メソッド]

drawable ID を返す。

:flush [メソッド]

Xlib のバッファに保有されるコマンドを掃き出す。

:geometry [メソッド]

7 つの幾何学属性のリストを返す。そのリストは、root-window-id, x 座標, y 座標, 幅, 高さ, 枠線の幅, visual の深さである。

:height [メソッド]

この **Xdrawable** の高さ（y 軸方向のドット数）を返す。

:width [メソッド]

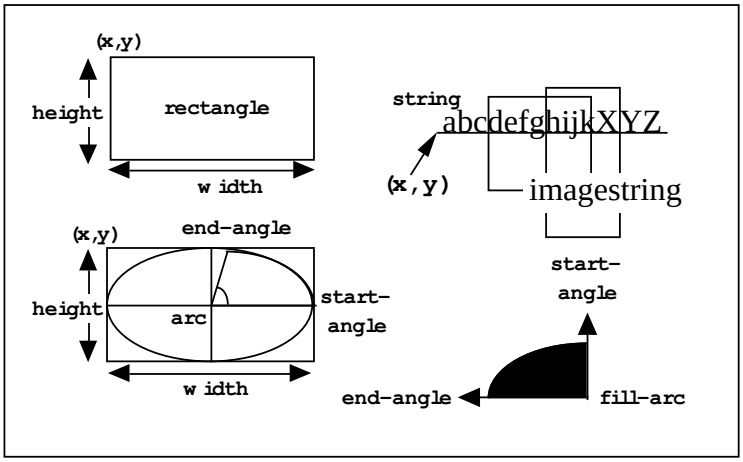

図 14: 描画の基本

この **Xdrawable** の幅（x 軸方向のドット数）を返す。

:gc *&rest newgc* [メソッド]

もし、*newgc* が与えられない場合、現在の GC オブジェクトを返す。もし、*newgc* が **gcontext** のインスタンスなら、この **Xdrawable** の gc に設定する。そうでなければ、*newgc* はメッセージとみなされ、現在の gc に送られる。

:pos [メソッド]

この **Xdrawable** の位置を示す整数ベクトルを返す。位置は親 window の相対位置としていつも定義され、window マネージャの仲介のもとにルート window の直接の子 window として生成された window は、ルート window の本当の位置に関わらず、環境のタイトル window の固定座標を返す。

:x [メソッド]

この **Xdrawable** の親 window との相対的な現在の x 座標を返す。

:y [メソッド]

この **Xdrawable** の親 window との相対的な現在の y 座標を返す。

:copy-from *drw* [メソッド]

drw は、他の **drawable** オブジェクト (Xwindow または pixmap) である。*drw* の内容がこの **Xdrawable** にコピーされる。

:point *x y &optional (gc gccon)* [メソッド]

(x, y) の位置にオプションの *gc* で点を描く。

:line *x1 y1 x2 y2 &optional (gc gcon)* [メソッド]

(x1,y1) から *(x2,y2)* へオプションの *gc* を用いて線分を描く。*x1, y1, x2, y2* は整数でなければならない。

:rectangle *x y width height &optional (gc gcon)* [メソッド]

(x,y) を中心として *width* の幅と *height* の高さを持つ四角形を描く。

:arc *x y width height angle1 angle2 &optional (gc gcon)* [メソッド]

(x,y) を中心として *width* の幅と *height* の高さを持つ四角形に内接する楕円の円弧を描く。*angle1* が始まりの角度を示し、*angle2* が終わりの角度を示す。これらの角度の単位はラジアンである。

:fill-rectangle *x y width height &optional (gc gcon)* [メソッド]

四角領域を埋める。

:fill-arc *x y width height angle1 angle2 &optional (gc gcon)* [メソッド]
円弧の中を埋める。

:string *x y str &optional (gc gcon)* [メソッド]
(x,y) の位置から文字列 *str* を表示する。背景は、書かない。

:image-string *x y str &optional (gc gcon)* [メソッド]
文字列 *str* を表示する。文字列は、背景色で満たされる。

:getimage *&key :x :y :width :height (:mask #ffffffff) (:format 2)* [メソッド]
server から ximage を取り、ピクセルデータを文字列として返す。server から送られるピクセルデータは、一端 Xlib の ximage 構造に蓄積される。その後、行毎に文字列にコピーされる。ximage 構造は、自動的に破壊される。`:getimage` により得られた画像文字列は、`pixel-image` を作るために使用できる。また、**??**節に書かれているように pbm フォーマットのファイルに書き込むことができる。

:putimage *image &key :src-x :src-y :dst-x :dst-y :width :height ((:gc g) gc)* [メソッド]
この **Xdrawable** の指定された位置に *image* を表示する。*image* は、文字列あるいは ximage 構造へのアドレスポインターである。

:draw-line *from to* [メソッド]
:line メソッドと同じである。他の **viewsurface** クラスとの互換性を提供できる。

:line-width *&optional dots* [メソッド]
この **Xdrawable** のデフォルト GC の線の幅を設定する。`:gc :line-width` メッセージの使用を推薦する。

:line-style *&optional dash* [メソッド]
この **Xdrawable** のデフォルト GC の線スタイルを設定する。`:gc :line-style` の使用が好まれる。

:color *&optional c* [メソッド]
この **Xdrawable** の色を設定する。

:clear [メソッド]
全画面を消去する。この関数は、`:clear-area` を呼び出す。

:clear-area *&key :x :y :width :height :gc* [メソッド]
:fill-rectangle メソッドを用いて四角領域を消去する。

Xpixmap [クラス]

:super **Xdrawable**
:slots

pixmap は、画像バッファあるいは背景のパターンとしてしばしば用いられる drawable である。xwindow と異なり、xwindow にコピーされるまで pixmap 自身を見ることはできないし、pixmap はどんなイベントも発生しない。

:init *id* [メソッド]
この pixmap を初期化する。

:create *&key (:width 500) (:height 500) (:depth 1) (:gc *defaultgc*)* [メソッド]
デフォルト GC として*:gc* を持つ、*width*×*height* の pixmap を作成する。

:create-from-bitmap-file *fname* [メソッド]

file で指定される bitmap ファイルから pixmap を作る。

:write-to-bitmap-file *fname* [メソッド]

この pixmap の内容を *fname* で指定される bitmap ファイルに書き込む。このファイルは、**:create-from-bitmap-file** メソッドで pixmap を作り、読み戻すことができる。

:destroy [メソッド]

この pixmap を破壊し、X resource を開放する。

Xwindow [クラス]

:super	**Xdrawable**
:slots	(parent subwindows backing-store)

Xwindow は、画面の見える四角領域を定義する。これは、**text-window** やグラフィックオブジェクトを描くための **canvas** だけでなく、window というよりはむしろグラフィックオブジェクトのようなたくさんの **panel-item** や **scroll-bars** からも継承される。

:create *&key ((:parent *root*)* [メソッド]
(:x 0) (:y 0) (:size 256) (:width size) (:height size) (:border-width 2)
(:save-under nil) (:backing-store :always) (:backing-pixmap nil)
*(:border *fg-pixel*) (:background *bg-pixel*)*
(:map T) (:gravity :northwest)
(:title "WINDOW") (:name title)
(:font)
:event-mask (:key :button :enterLeave :configure :motion)

xwindow 生成し、初期化する。*:parent* が与えられたとき、この window は*:parent* の子 window として生成され、*:parent* の `subwindows` リストに蓄積される。*:x, :y, :size, :width, :height* と *:border-width* は、この window の位置と寸法を決定する。*:save-under* と *:backing-store* は、window が再マップされたときに生じる Xserver の行動を制御する。*:backing-store* は`:notUseful`, `:WhenMapped`, `:Always` のどれかであるが、*:save-under* は T あるいは NIL である。*:backing-pixmap* が T のとき、この window と同じサイズの pixmap が Euslisp により生成され、もし Xserver が `backing-store` の容量を持っていない場合は、backing-store として蓄積される。*:border* と *:background* は、`border_pixel` と `background_pixel` 属性をそれぞれ指定する。もし、**panel** の中の panel-button としてたくさんの小さな window を作成するような場合で、この window が生成後にすぐ表示されるべきでないならば、*:map* は NIL にセットされなければならない。*:title* は、window のタイトルバーに現れる window のタイトルである。*:name* は、この window の plist に蓄積される window の名前であり、プリンタにより表示される名前である。この window への X のイベントは、*:event-mask* によって決定される。それはビットで構成される event-mask の整数表現あるいは次の symbol のリストである。`:key`, `:button`, `:enterLeave`, `:motion` と `:configure`。もし、もっと正確な制御が必要ならば、次の symbol をそれぞれのイベントに指定することができる。`:keyPress`, `:keyRelease`, `:ButtonPress`, `:ButtonRelease`, `:EnterWindow`, `:LeaveWindow`, `:PointerMotion`, `:PointerMotionHint`, `:ButtonMotion`, `:KeyMapState`, `:Exposure`, `:VisibilityChange`, `:StructureNotify`, `:ResezeRedirect`, `:SubstructureNotify`, `:SubstructureRedirect`, `:FocusChange`, `:PropertyChange`, `:ColormapChange` と `:OwnerGrabButton`。`:key` は、`:keyPress` と `:KeyRelease` の両方が指定でき、`:button` は、`:ButtonPress` と `:ButtonRelease` の両方が指定できる。サーバからイベントが送られてきたとき、**window-main-loop** は、そのイベント構造を解析し、`:KeyPress`, `:KeyRelease`, `:buttonPress`, `:ButtonRelease`, `:EnterNotify`, `:LeaveNotify`, `:MotionNotify`, `:ConfigureNotify` メッセージをそのイベントが発

生した window に送る。

:map [メソッド]
この **Xwindow** とその子 window を全て表示する。

:unmap [メソッド]
この **Xwindow** をとその子 window を全て非表示にする。

:selectinput *event-mask* [メソッド]
event-mask は、整数かあるいはイベントマスク symbol のリストである。ビットが 1 となっているかあるいは *event-mask* リストに列挙されているイベントは、それぞれこの window に通知される。

:destroy [メソッド]
この **Xwindow** を破壊し、X resource を開放する。window オブジェクトは、ガーベージコレクトされないため、`*xwindow*`や`*xwindow-hash-tab*`の中の一致するエントリも削除される。全ての子 window も`:destroy` を送ることにより削除する。この window は、親 window の子 window のリストから削除される。`drawable`ID は、NIL に設定される。

:parent [メソッド]
親 window オブジェクトを返す。

:subwindows [メソッド]
全ての子 window のリストを返す。子 window は、もっとも最近作られたものがリストの最初である。この window の直接の子 window だけがリストされ、子 window の子 window はリストされない。

:associate *child* [メソッド]
この window の子 window として *child* window を登録する。

:dissociate *child* [メソッド]
子 window のリストから *child* window を削除する。

:title *title* [メソッド]
この window のタイトル名を変更する。そのタイトル名は Xserver に送られるため、一旦蓄積され、window manager によって表示される。

:attributes [メソッド]
この window の属性を表現する整数ベクトルを返す。

:visual [メソッド]
この **Xwindow** の visual resource ID を返す。

:screen [メソッド]
この **Xwindow** の screen resource ID を返す。

:root [メソッド]
root window ID を返す。

:location [メソッド]
この window の x と y 座標を記述する 2 次元の整数ベクトルを返す。

:depth [メソッド]
この window の深さ (カラープレーンの数) を返す。

:size [メソッド]

この window のサイズ (高さと幅) を返す。

:colormap [メソッド]
この window の colormap resource ID を返す。

:move *newx newy* [メソッド]
この window の位置を *(newx,newy)* に変更する。位置は、親 window の相対位置で与えられる。

:resize *width height* [メソッド]
この window のサイズを変更する。たぶん、大きさのパラメータがクライアント側の Xlib に中にキャッシュされるため、**:resize** の直後に:geometry メッセージを送ると誤った（古い）結果を返す。

:raise [メソッド]
この window を前面に持ってくる。

:lower [メソッド]
この window を後ろに置く。

:background *pixel* [メソッド]
背景のピクセル値（カラーマップのインデックス）を *pixel* に変更する。*pixel* 値は、bg-color スロットにも保存される。:clear 処理は、現在の背景を指定された *pixel* で埋める。

:background-pixmap *pixmap* [メソッド]
背景の pixmap 値を *pixmap* に変更する。

:border *pixel* [メソッド]
この window の枠線の色を *pixel* に設定する。

:set-colormap *cmap* [メソッド]
colormap を設定する。

:clear [メソッド]
この **Xwindow** 内を全て消去する。

:clear-area *&key :x :y :width :height* [メソッド]
この **Xwindow** の指定された四角領域を消去する。

make-xwindow *&rest args* [関数]
引き数 *args* で示される **Xwindow** を作る。

init-xwindow *&optional (display (getenv "DISPLAY"))* [関数]
eusx が起動するときに最初に呼び出される関数である。**init-xwindow** は、*display* で指定される Xserver に接続し、16.1 節に記述されているデフォルト変数を初期化する。**init-xwindow** は、デフォルトフォントをロードし、グローバル変数に設定する。例えば、font-courb12, lucidasans-bold-12 など。このフォントロードは、起動時に時間遅れを引き起こす。フォントロードの数を減らしたり、ワイルドカード文字"*"を使用せずに正確なフォント名を指定することにより、その遅れを短くできる。

16.3 Graphic Context

gcontext [クラス]

:super **Xobject**

:slots (gcid GCValues)

graphic context(GC) を定義する。Euslisp の中で、全ての window はデフォルト GC を持っている。

:create *&key (:drawable defaultRootWindow)* [メソッド]
*(:foreground *fg-pixel*) (:background *bg-pixel*)*
:function :plane-mask
:line-width :line-style :cap-style :join-style
:font :dash

与えられた属性で GC を作成する。*drawable* は、画面と画面の深さを知るために Xserver により使用される。結果の GC は、同じ画面上で作成される限り、どの drawable でも使用できる。

:gc [メソッド]
X の GC ID を返す。

:free [メソッド]
この **GC** を開放する。

:copy [メソッド]
この **GC** のコピーを作る。

:foreground *&optional color* [メソッド]
もし、*color* が与えられたならば、文字色に *color* を設定する。*color* はピクセル値である。

:background *&optional color* [メソッド]
もし、*color* が与えられたならば、背景色に *color* を設定する。*color* はピクセル値である。

:foreback *fore back* [メソッド]
一度に文字色と背景色を設定する。

:planemask *&optional plane-mask* [メソッド]
plane-mask を設定する。

:function x [メソッド]
描画機能を設定する。xは、以下に示す数字かキーワードの内の 1 つである。`0=Clear, 1=And, 2=AndReverse, 3=Copy, 4=AndInverted, 5=NoOp, 6=Xor, 7=Or, 8=Nor, 9=Equiv, 10=Invert, 11=XorReverse, 12=CopyInverted, 13=OrInverted, 14=Nand, 15=Set, :clear, :and, :andReverse, :copy, :andInverted, :NoOp, :Xor, :Or, :Nor, :Equiv, :Invert, :XorReverse, :CopyInverted, :OrInverted, :Nand, :Set`

:font x [メソッド]
この GC のフォント属性を設定する。xは、フォント名あるいはフォント ID である。もし、xがフォント名（文字列）であったならば、**:font** はフォント ID を決めるために **x:LoadQueryFont** を呼び出す。もし、見つからなかった場合、`"no such font ..."`が警告される。もし、xが NIL（与えられなかった）ならば、この GC の現在のフォント ID が返される。

:line-width x [メソッド]
線幅をピクセル数 x で設定する。

:line-style x [メソッド]
線スタイル (実線、点線など) を設定する。

:dash *&rest x* [メソッド]

x のそれぞれの要素は、整数である。**:dash** は、線スタイルの点線パターンを設定する。

:tile *pixmap* [メソッド]

pixmap をこの GC のタイルパターンに設定する。

:stipple *pixmap* [メソッド]

pixmap をこの GC の点画に設定する。

:get-attribute *attr* [メソッド]

属性値を得る。*attr*は、`:function`, `:plane-mask`, `:foreground`, `:background`, `:line-width`, `:line-style` `:cap-style`, `:join-style`, `:fill-style`, `:fill-rule`, `:font` の内の 1 つである。属性値を表す整数が返される。

:change-attributes *&key :function :plane-mask :foreground :background :line-width :line-style :cap-style :join-style :font :dash* [メソッド]

属性値を変更する。同時に複数の属性値を変更できる。

font-id *fontname* [関数]

もし、*fontname* が整数であるなら、フォント ID とみなしてその値を返す。もし、*fontname* が文字列であるなら、**x:LoadQueryFont** を使用してフォント構造を得て、そのフォント ID を返す。*fontname* は、正確な名前の略語でも良い。例えば、24 ポイントのクーリエフォントとして`"*-courier-24-*"`を指定できる。もし、フォントが見つからなければ、`can't load font` の警告を出力する。

textdots *str font-id* [関数]

str 文字列の ascent descent width をドット単位に示す 3 つの整数のリストを返す。

16.4 色とカラーマップ

colormap [クラス]

:super	**object**
:slots	(cmapid planes pixels LUT-list)

Xwindow のカラーマップおよびアプリケーション指向のカラールックアップテーブルを定義する。カラーは RGB 値で表現され、その範囲は 0〜65535 である。カラーマップのカラーセルは、8 ビットの擬似カラーディスプレイの上の 0〜255 の範囲の値に設定される。

ここで、8 ビットの擬似カラーディスプレイの機能があり、256 色を選択することができると仮定する。基本的にカラーマップを使用する 2 つの方法がある。1 つは、システムデフォルトのカラーマップを共有する方法で、もう 1 つはプロセスに独自のカラーマップを作成する方法である。もし、システムのデフォルトカラーマップを使用する場合、マップのすべてのカラーセルを使いきらないように注意しなければならない。なぜなら、マップは多くののプロセスから共有されているからである。もし、独自のカラーマップを使用する場合、他のプロセスを気にすることなく 256 色すべてを使用することができる。しかし、そのマップは明確に独自の window に設定しなければならない。マウスのポインターが window 内のどこかに動かされたとき、カラーマップは window manager により活性化される。

システムデフォルトのカラーマップは eusx が実行される最初に **x:colormap** のクラスのインスタンスとして、**x:*color-map***に設定されている。もし、独自のカラーマップを使用するとき、**x:colormap** のインスタ

ンスを作る。これらのインスタンスは、x server で定義された colormap オブジェクトと一致しており、それぞれのインスタンスの `cmapid` に示されている。

システムデフォルトのカラーマップを使用するとき、他のプロセスと共有するカラーを読み込み専用 (*read-only*) に、Euslisp の独自カラーを読み書き可能 (*read-write*) に定義することができる。”読み込み専用”は、カラーセルに割り当てられる任意のカラーに定義することができる。しかし、割り当てた後変更することができない。もう一方で、”読み書き可能”カラーは定義した後でさえ、変更することができる。共有カラーは、他のプロセスが変更を予期していないため”読み込み専用”である。この”読み込み専用”と”読み書き可能”の属性は、それぞれのカラーに付けられる。(しばしば、カラーセルとして参照される)

colormap オブジェクトは、color ID から RGB の物理的な表現への変換を定義する。しかしながら、これらの論理的な color ID は、任意に選択することができない。特に、システムのデフォルトのカラーマップを使用しているとき、使用できない。color ID (しばしば'pixel' として参照される) は、カラーマップの特別な色のインデックスである。Xlib は、割り当ての要求があると、共有カラーのために空いたインデックスの１つを選択する。したがって、たくさんのグレー階調のカラーを連続的に割り当てることを保証することあるいは最初 (0 番目) のインデックスから始めることはできない。

アプリケーションの観点から、もっと論理的なカラー名が必要とされる。例えば、グレー階調の数は明るさをインデックスとして参照すべきである。レイトレーシングプログラムは、HLS で定義される違った明るさのカラーのグループのために連続的なインデックスの割り当てを要求する。

この問題に対処するために、Euslisp のカラーマップはルックアップテーブル (LUT) と呼ばれる別の変換テーブルを提供している。論理的なカラーグループのために、LUT を定義でき、symbol 名を付けることができる。１つ以上の LUT をカラーマップとして定義できる。LUT は、Xserver が認識できるように、アプリケーションが指定した論理カラーのインデックスを物理ピクセル値に変換するために整数ベクトルである。

:id [メソッド]

`cmapid` を返す。

:query *pix* [メソッド]

指定されたピクセル数の RGB 値を返す。

:alloc *pix r g b* [メソッド]

このメソッドは、`:store nil r g b` と同一である。新しいカラーセルがこのカラーマップに配置され、指定された RGB 値に割り当てられる。

:store *pix r g b* [メソッド]

pix 番目のカラーセルの RGB 値を設定する。

:store *pix color-name* [メソッド]

:store は、カラーマップに色を設定する低レベルメソッドである。１つの書式として、RGB 値を 0〜65535 で指定する方法である。他の書式として、”red” や”navy-blue”のようなカラー名で指定する。もし、*color-name* がなければ、NIL を返す。ピクセルはカラーマップのインデックスあるいは NIL である。もし整数なら、カラーセルは読み書き可能でなければならない。もし NIL なら、共有の読み込み専用カラーセルが割り当てられている。**:store** は、カラーマップ内のカラーセルのインデックスを返す。

:store-hls *pix hue lightness saturation* [メソッド]

HLS(Hue, Lightness and Saturation) で指定される色をカラーマップの *pix* 番目に蓄積する。もし、*pix* が NIL なら、共有の読み込み専用のカラーセルが割り当てられる。**:store-hls** は、カラーセルに割り当てられるインデックスを返す。

:destroy [メソッド]

この **colormap** を破壊し、リソースを空にする。

:pixel *LUT-name id* [メソッド]

LUT の中から *id* 番目を調べて、ピクセル値を返す。*LUT-name* は、`:define-LUT` で定義されたルックアップテーブルの名前である。

:allocate-private-colors *num* [メソッド]

独自のカラーマップに *num* 個のカラーセルを割り当てる。

:allocate-colors *rgb-list [private]* [メソッド]

rgb-list のそれぞれの要素は、red,green,blue のリストである。カラーセルは、それぞれの RGB 値が割り当てられ、ピクセル値を要素とする整数ベクトルを返す。

:define-LUT *LUT-name rgb-list [private]* [メソッド]

rgb-list に記述されたカラーが割り当てられ、LUT が *LUT-name* のシンボリック名で登録される。独自のカラーセルを定義するためには、*private* を T に設定すること。

:define-gray-scale-LUT *LUT-name levels [private]* [メソッド]

線形のグレースケールカラーで表現される *levels* 階調のカラーセルを割り当て、LUT を返す。例えば、`(send x:*color-map* :define-gray-scale-LUT 'gray8 8)` は、システムのデフォルトカラーマップの中に８つのグレーカラーを配置し、`#i(29 30 31 48 49 50 51 0)` のような整数ベクトルを返す。物理ピクセル値は、`:pixel` メッセージを送ることにより得られる。例えば、`(send x:*color-map* :pixel 'gray8 2)` は、31 を返す。

:define-rgb-LUT *LUT-name red green blue [private]* [メソッド]

RGB 表現を縮小した LUT を定義する。例えば、もし、*red=green=blue*=2 なら、カラーセルには $2^{2+2+2} = 2^6 = 64$ が割り当てられる。

:define-hls-LUT *LUT-name count hue low-brightness high-brightness saturation [private]* [メソッド]

HLS で使用する *count* 数のカラーを配置する。*hue* (0..360),*saturation* (0..1),*low-brightness* と *high-brightness* の明るさの差をカラーマップに蓄積される。*LUT-name* で名付けられる LUT も作られる。

:define-rainbow-LUT *LUT-name count* [メソッド]
(hue-start 0) (hue-end 360)
(brightness 0.5)
(saturation 1.0) (private nil)

HLS モデルを用いて *count* の色を配置する。*brightness* (0..1) と *saturation* (0..1) と, *hue-start* と *hue-end* 間の異なった hue を持つ色をカラーマップに蓄積する。*LUT-name* を名付けられた LUT も生成される。

:LUT-list [メソッド]

このカラーマップに定義されている全ての LUT のリストを返す。リストのそれぞれのエントリは、LUT 名と整数ベクトルの組である。

:LUT-names [メソッド]

このカラーマップの全ての LUT の名前のリストを返す。

:LUT *name* [メソッド]

name で指定される整数ベクトル (LUT) を返す。

:size *LUT* [メソッド]

LUT の長さを返す。

:planes [メソッド]

このカラーマップのプレーンを返す。

:set-window *xwin* [メソッド]
このカラーマップを *xwin* の window と関連付ける。このカラーマップは、*xwin* にカーソルが入ったとき活性化される。

:free *pixel — LUT* [メソッド]
pixel の場所にあるカラーセルを開放するかあるいは *LUT* のすべてを開放する。

:init *[cmapid]* [メソッド]
このカラーマップを cmapid で初期化する。登録された LUT はすべて捨てられる。

:create *&key (planes 0) (colors 1) (visual *visual*) (contiguous i l)* [メソッド]
新しいカラーマップを作成する。

XColor [クラス]

:super	**cstruct**
:slots	((pixel :integer)
	(red :short)
	(green :short)
	(blue :short)
	(flags :byte)
	(pad :byte))

RGB モデルのカラーを定義する。それぞれのスロットに値を割り当てるには、**setf** を用いる。RGB 値は、符合拡張され、最大値は −1 と表現される。

:red [メソッド]
この **XColor** の赤色の値を返す。

:blue [メソッド]
この **XColor** の青色の値を返す。

:green [メソッド]
この **XColor** の緑色の値を返す。

:color [メソッド]
この **XColor** の RGB 値のリストを返す。

:init *pix R G B &optional (f 7)* [メソッド]
XColor を初期化する。

find-visual *type depth &optional (screen 0)* [関数]
type と *depth* で指定される visual-ID を見つける。*type* は、`:StaticGray`, `:GrayScale`, `:StaticColor`, `:pseudoColor`, `:TrueColor` あるいは`:DirectColor` のどれかである。ふつう、*depth* は 1, 8 または 24 である。

17 XToolKit

XToolKit は、ボタン、プルダウンメニュ、テキスト window などの GUI 要素を使用して GUI (Graphical User Interface) を作成するのを容易にするための高レベル Xwindow インターフェースである。Xlib クラスとの大きな違いは、XToolKit が Xserver から送られる Xevent と一致するユーザーが定義した対話ルーチンを呼び出し、それらの対話指向 window パーツと一致した外観を提供することである。XToolKit に含まれるクラスは、以下の継承構造を持っている。

```
xwindow
     panel
          menubar-panel
          menu-panel
          filepanel
          textviewpanel
          confirmpanel
     panel-item
          button-item
               menu-button-item
               bitmap-button-item
               menu-item
          text-item
          slider-item
          choice-item
          joystick-item
     canvas
     textwindow
          buffertextwindow
               scrolltextwindow
          textedit
     scroll-bar
          horizontal-scroll-bar
```

以下に示す xwindow クラスは XToolKit の 5 つの基本クラスである。`panel`, `panel-item`, `canvas`, `textWindow` と `scroll-bar`。`menubar-panel` と `menu-panel` は、`panel` の下に定義される。新しいアプリケーション window を作り、イベントの上でそれを実行させるための基本的な方策は、以下の通りである。

1. **アプリケーションクラスの定義** アプリケーションクラス window は、XToolKit の要素を置く能力を持つ **panel** のサブクラスとして定義されなければならない。

2. **イベントハンドラの定義** アプリケーションクラスにおいて、ボタンが押されたり、メニューアイテムが選択されたりしたときに呼び出されるイベントハンドラを定義する。イベントハンドラは、panel-item の指定された引数を持つメソッドとして定義すべきである。

3. **サブパネルの定義** もし、`menubar-panel` を使用するなら、アプリケーション window の一番上におかれる。したがって、`:create-menubar` によって最初に作成されなければならない。同様に、`menu-panel` は、その `menu-panel` と関連する `menu-button-item` より前に定義する必要がある。

4. **パネルアイテムの作成** `button-item`, `text-item`, `slider-item` などのようなパネルアイテムは、(`send-super :create-item` *`class label object method`*) によって作成することができる。上記で定義されたイベ

ントハンドラは、それぞれのパネルアイテムと接続される。これらの初期化手続きは、アプリケーション window クラスの :create メソッドの中で定義すべきである。必要なときはいつでもイベント送信を停止するための quit ボタンを定義することを忘れないこと。どんな textWindow と canvas も、:locate-item メソッドを経由してアプリケーション window の中に置くことができる。

5. **window 全体の作成** :create メッセージをアプリケーションクラスに送ることで、window に XToolKit の要素を正しく置いたアプリケーション window を作成する。

6. **イベント送信の実行** Xserver からイベントを受け、一致する window に配るためには、window-main-loop を実行すること。Solaris2 上では、イベントを配るための異なったスレッドである window-main-thread で実行する。window-main-thread では、最上位レベルの対話が活きている。window-main-thread を２回以上実行してはならない。

17.1 X イベント

現在の実行において、イベント構造は固定イベントバッファ（25 要素の整数ベクトル）から受け、同じバッファが全てのイベントに関して再使用される。イベント構造は、同時に 2 つ以上のイベントを参照する必要があるとき、コピーされなければならない。

window-main-loop は、Xserver から送られる全てのイベントを捕獲し、イベントが発生した window に配るための関数である。

event [変数]
もっとも最近のイベント構造を持つ 25 要素の整数ベクトル

next-event [関数]
event の中にイベント構造を蓄積し、もし１つでも未決定のイベントがあればそれを返し、なければ NIL を返す。

event-type *event* [関数]
event のイベント型を表現するキーワード symbol 返す。イベント型キーワードは、:KeyPress (2), :KeyRelease (3), :ButtonPress (4), :ButtonRelease (5), :MotionNotify (6), :EnterNotify (7), :LeaveNotify (8), :FocusIn (9), :FocusOut (0), :KeymapNotify (1), :Expose (12), :GraphicsExpose (13), :NoExpose (14), :VisibilityNotify (15), :CreateNotify (16), :DestroyNotify (17), :UnmapNotify (18), :MapNotify (19), :MapRequest (20), :ConfigureNotify (22), :ConfigureRequest (23), :GravityNotify (24), :ResizeRequest (25), :CirculateNotify (26), :CirculateRequest (27), :PropertyNotify (28), :SelectionClear (29), :SelectionRequest (30), :SelectionNotify (31), :ColormapNotify (32), :ClientMessage (33), :MappingNotify (34), :LASTEvent (35) である。

event-window *event* [関数]
event が発生した window オブジェクトを返す。

event-x *event* [関数]
event からそのイベントが発生した x 座標を抜き出す。(すなわち、window 内におけるマウスポインタの横方向の相対的な位置)

event-y *event* [関数]
event からそのイベントが発生した y 座標を抜き出す。(すなわち、window 内におけるマウスポインタの縦方向の相対的な位置)

event-width *event* [関数]

:configureNotify イベントに幅パラメータを表現する *event* の８つの要素を返す。

event-height *event* [関数]

:configureNotify イベントに高さパラメータを表現する *event* の 8 つの要素を返す。

event-state *event* [関数]

キーの状態で変更されたマウスボタンを表現するキーワードのリストを返す。キーワードは、:shift, :control, :meta, :left, :middle と:right である。例えば、もしシフトキーが押されている状態で左のボタンが押されたならば、(:shift :left) が返される。

display-events [関数]

x:nextevent によって捕獲された全ての xwindow イベントを表示する。Control-C は、この関数を停止させる唯一の方法である。

window-main-loop *&rest forms* [マクロ]

Xevent を受け、イベントが発生した window オブジェクトにそれを配る。イベントの型に沿って、:KeyPress, :KeyRelease, :ButtonPress, :ButtonRelease, :MotionNotify, :EnterNotify, :LeaveNoti や :ConfigureNotify と名付けられた window クラスのメソッドが *event* を引数として呼び出される。もし、*forms* が与えられたならば、到着したイベントがチェックされるとき毎にそれらを評価する。

window-main-thread [関数]

スレッドであることを除いて **window-main-loop** と同じことをする。**window-main-thread** は、Solaris2 でのみ実現されている。**window-main-thread** は、read-eval-print が入力されないエラーハンドラをインストールしている。エラー情報を表示した後、そのイベントは処理を続ける。

17.2 パネル

panel [クラス]

:super	**xwindow**
:slots	(pos items fontid
	rows columns ;total number of rows and columns
	next-x next-y
	item-width item-height)

panel は、パネルアイテムや他の panel を含んだどんな xwindow も置くことができる xwindow である。**panel** オブジェクトは、panel の中で生成されたパネルアイテムへのデフォルトフォントを供給する。アプリケーション window は、**panel** のサブクラスとして定義去れなければならない。

:create *&rest args &key ((:item-height iheight) 30) ((:item-width iwidth) 50) (:font font-lucidasans-bold-12) ((:background color) *bisque1*) &allow-other-keys)* [メソッド]

panel を生成し、初期化する。スーパークラスの:create が呼び出されるため、**xwindow** に対する全ての生成用パラメータ（*:width, :height, :border-width* など）が許される。*:item-height* と *:item-width* は、最小の高さと幅をそれぞれのパネルアイテムに与える。

:items [メソッド]

関連するアイテムを全てリストで返す。

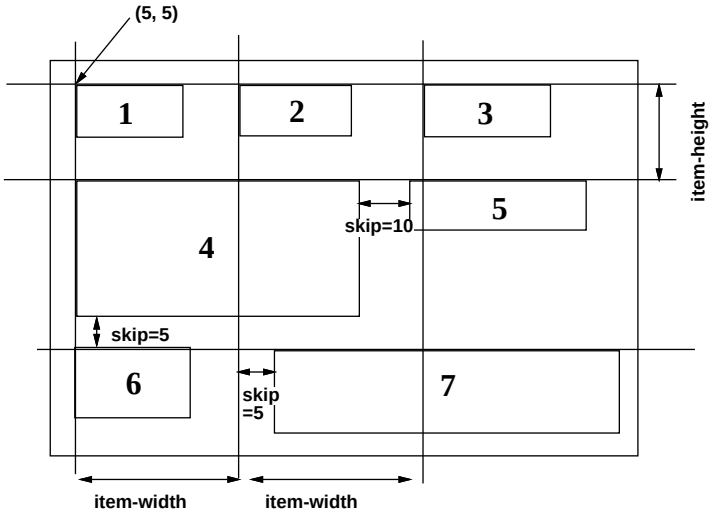

図 15: panel のアイテムレイアウト

:locate-item *item &optional x y* [メソッド]

item は、xwindow オブジェクト（ふつうはパネルアイテム）である。もし *x* と *y* が与えられたならば、アイテムはそこに置かれる。そうでなければ、*item* は、もっとも最近に置かれたアイテムに隣接するように置かれる。アイテムは、図 15 のように上から下に向かって、また左から右に向かって置かれていく。**:locate-item** は、また `items` や `subwindows` リストに *item* を追加し、`:map` を送ることにより見えるようにする。

:create-item *klass label receiver method &rest args* [メソッド]
&key ((:font fontid)
&allow-other-keys)

klass で指定されるパネルアイテムのクラス (すなわち、`button-item`, `menu-button-item`, `slider-item`, `joystick-item` など), のインスタンスを作り、`:locate-item` を用いて panel にアイテムを置く。*args* は、*klass* の `:create` メソッドに送られる。*label* は、パネルアイテムの中に書かれる識別文字列である。*receiver* と *method* は、一致するイベントを呼び出すイベントハンドラを指定する。

:delete-items [メソッド]

パネルアイテムを全て削除する。

:create-menubar *&rest args* [メソッド]
&key (:font fontid)
&allow-other-keys

menubar-panel を作成し、panel の最上部に置く。

以下に示すメソッドは、イベントがイベントハンドラのない panel に送られたとき、”subclass’s responsibility” 警告メッセージを避けるために提供されている。ユーザーのアプリケーションでは、これらのメソッドを上書きしなければならない。

:quit *&rest a* [メソッド]

window-main-loop に `:quit` メッセージを送り、イベント処理を停止する。

:KeyPress *event* [メソッド]

NIL を返す。

:KeyRelease *event* [メソッド]
NIL を返す。

:ButtonPress *event* [メソッド]
NIL を返す。

:ButtonRelease *event* [メソッド]
NIL を返す。

:MotionNotify *event* [メソッド]
NIL を返す。

:EnterNotify *event* [メソッド]
NIL を返す。

:LeaveNotify *event* [メソッド]
NIL を返す。

17.2.1 サブパネル (メニューとメニューバー)

menu-panel [クラス]

:super	**panel**
:slots	(items item-dots item-height
	charwidth charheight
	height-offset
	highlight-item
	color-pixels
	active-color)

menu-panel は、`panel-button` と `menu-item` のみを含むことができるパネルの一種である。`panel` と異なり、`menu-panel` はふつう見えないし、`menu-panel` と関連した `button-item` が押された時に表示される。もし、menu-panel がいつも見えるように作られたならば、ピンを刺したメニューとなる。マウスイベントに対する menu-item の応答は、アイテムの外のどこかで押されたマウスボタンのようにふつうの menu-button と全く異なっている。`menu-panel` を使用するためには、最初に作成し、その中に button-item を置く。それから、`menu-button-item` が panel の中あるいは menubar の中に*:menu* の引数として `menu-panel` と一緒に作成される。

:create *&rest args &key (:items) (:border-width 0) (:font font-courb12) (:width 100) (:height-offset 15) (:color *bisque1*) (:active *bisque2*) &allow-other-keys* [メソッド]
menu-panel window を作成する。その window の大きさは、新しい menu-item が追加される時に拡張される。

:add-item *label name &optional (receiver self) &rest mesg* [メソッド]
この **menu-panel** window の中に menu アイテムを追加し、対応する行動を張り付ける。マウスボタンがアイテムの上で外されたとき、*receiver* オブジェクトは *mesg* を受け取る。

menubar-panel [クラス]

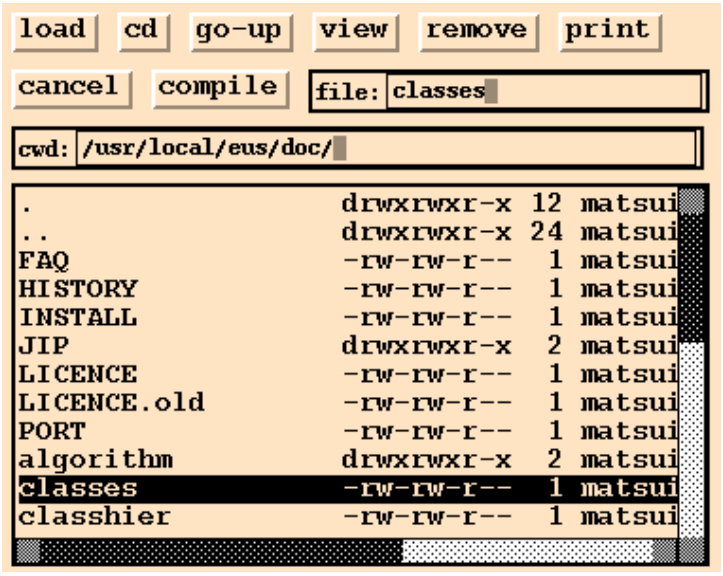

図 16: ファイルパネル window

:super **panel**
:slots

`menubar-panel` は、親 panel の最上部にいつも置かれるサブパネルである。メニューバーに置かれるパネルアイテムは、`menu-button-item` でなければならない。menubar-panel は、panel の `:create-menubar` メソッドにより生成される。

17.2.2 ファイルパネル

FilePanel は、ファイルやディレクトリを対話的に処理するアプリケーション window である。`cd` や `go-up` ボタンを使用することにより、どんなディレクトリも見に行くことができるし、以下の `ScrollTextWindow` の中にディレクトリ内に含まれるファイルを表示する。テキストファイルは、異なった window(TextViewPanel) の中に表示することができる。ファイルは、また印刷することができ、削除することができ、ボタンをクリックすることにより簡単にコンパイルすることができる。ファイルを印刷するとき、`a2ps` *`file`* `|` `lpr` コマンドが fork されたプロセスとして実行される。

17.2.3 テキスト表示パネル

TextViewPanel は、テキストファイルを表示するためのアプリケーション window クラスである (図 17)。プログラムテキストは、もっとも簡単なアプリケーション window の 1 つがどのように記述されているかを見れる。`:create` メソッドにおいて、quit ボタンと find ボタンとファイルの中を捜すための文字列を供給するための text-item を作成する。view-window は、縦と横にスクロールバーを持ちファイルを表示するための ScrollTextWindow である。TextViewPanel は、window マネージャーにより一番外側のタイトル window の大きさを変えたとき view-window の大きさを変えるために `:ConfigureNotify` イベントを捕獲する。

```
(defclass TextViewPanel :super panel
        :slots (quit-button find-button find-text view-window))
```

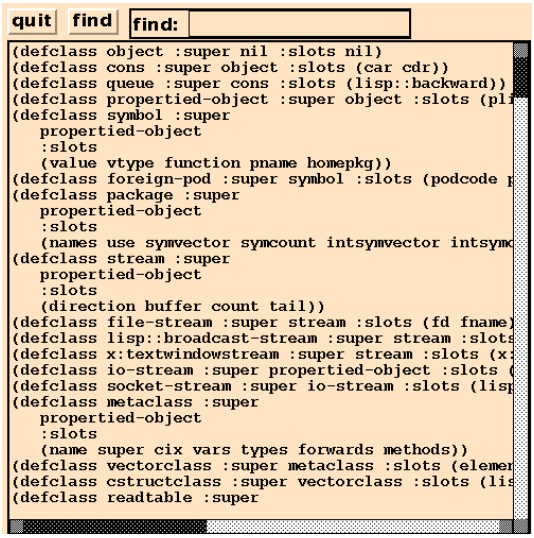

図 17: テキスト表示パネル window

```
(defmethod TextViewPanel
 (:create (file &rest args &key (width 400) &allow-other-keys)
    (send-super* :create :width width args)
    (setq quit-button
          (send self :create-item panel-button "quit" self :quit))
    (setq find-button
          (send self :create-item panel-button "find" self :find))
    (setq find-text
          (send self :create-item text-item "find: " self :find))
    (setq view-window
            (send self :locate-item
                (instance ScrollTextWindow :create
                   :width (setq width (- (send self :width) 10))
                   :height (- (setq height (send self :height)) 38)
                   :scroll-bar t :horizontal-scroll-bar t
                   :map nil      :parent self)))
    (send view-window :read-file file))
 (:quit (event)  (send self :destroy))
 (:find (event)
    (let ((findstr (send find-text :value)) (found)
          (nlines (send view-window :nlines)))
        (do ((i 0 (1+ i)))
            ((or (>= i nlines) found))
           (if (substringp findstr (send view-window :line i)) (setq found i)))
        (when found
           (send view-window :display-selection found)
           (send view-window :locate found))))
 (:resize (w h)
    (setq width w height h)
    (send view-window :resize (- w 10) (- h 38)))
```

```
(:configureNotify (event)
  (let ((newwidth (send self :width))
        (newheight (send self :height)))
       (when (or (/= newwidth width) (/= newheight height))
         (send self :resize newwidth newheight)))  ) )
```

17.3 パネルアイテム

panel-item [クラス]

:super **xwindow**
:slots (pos notify-object notify-method
fontid label labeldots)

panel-item は、パネルアイテム window のすべての種類において、アイテムが指定するイベントが発生したとき `notify-object` の `notify-method` を呼び出すための抽象クラスである。

:notify *&rest args* [メソッド]

`notify-object` の `notify-method` を呼び出す。イベント応答や `notify-method` へ送るための引き数がアイテムにより区別される。

button-item ボタンは、同じ button-item の押し、外し時に応答。引き数は button-item オブジェクトである。

menu-button-item メニューアイテムの選択時に応答。引き数は、menu-button-item オブジェクトである。

choice-item 新しい選択ボタンの選択時に応答。引き数は、choice-item オブジェクトとその選択番号である。

text-item 改行あるいはリターンの入力時に応答。引き数は、text-item オブジェクトと入力行（文字列）である。

slider-item スライダーノブは、つかみと移動時に応答。引き数は、slider-item オブジェクトと新たな値である。

joystick-item ジョイスティックは、つかみと移動時に応答。引き数は slider-item オブジェクトと新しい x と y の値である。

:create *name reciever method &rest args* [メソッド]
&key ((:width w) 100) ((:height h) 100) (:font font-courb12)
&allow-other-keys

パネルアイテムを作成する。パネルアイテムは、抽象クラスである。このメソッドは、サブクラスによって `send-super` を通してのみ呼び出すべきである。

button-item [クラス]

:super **panel-item**
:slots

button-item は、簡単なパネルアイテムである。button-item は、四角ボックスとその中のラベル文字列を持っている。クリックされたとき、button-item は panel-item オブジェクトを唯一の引き数として `notify-object` の `notify-method` を呼び出す。

:draw-label *&optional (state :top) (color bg-color) (border 2) (offset)* [メソッド]

button-item のラベルを書く。

:create *label revciever method &rest args* [メソッド]
&key :width :height (:font (send parent :gc :font))
(:background (send parent :gc :background))
(:border-width 0)
(:state :top)
&allow-other-keys

button-item を作成する。もし、ボタンの幅と高さが与えられないなら、サイズは与えられたフォントを用いて書かれたラベル文字列に合わせて自動的に設定される。*:border-width* はデフォルトで 0 であるが、擬似 3 次元表現でボタンを浮き出しにする。背景やフォントは親 window(すなわち、panel) で定義されているものをデフォルトとする。

:ButtonPress *event* [メソッド]

もし、ボタンであれば、背景色をグレーにする。

:ButtonRelease *event* [メソッド]

event の背景色を標準に変更する。

menu-button-item [クラス]

:super	**button-item**
:slots	(items item-dots item-labels charwidth charheight menu-window window-pos high-light)

プルダウンメニューを定義する。**menu-button-item** は、`button-item` のようであるが、**menu-button-item** は、ボタンの下の関連する `menu-panel` が押されたとき、`notify-object` にすぐにメッセージを送る代わりに、活性化させる。メニューアイテムの 1 つの上でマウスボタンが外されたときに、本当のメッセージが送られる。

:create *label reciever method* [メソッド]
&rest args
&key (:menu nil) (:items) (:state :flat)
&allow-other-keys

プルダウンメニューを作成する。*receiver* と *method* 引き数は、影響を与えない。

:ButtonPress *event* [メソッド]

プルダウンメニューのボタンを反転させ、ボタンの下に関連する menu-panel をさらす。

:ButtonRelease *event* [メソッド]

このボタンの下の `menu-panel` を消し、このボタンを元に戻す。

bitmap-button-item [クラス]

:super	**button-item**
:slots	(pixmap-id bitmap-width bitmap-height)

bitmap-button-item の関数は、`button-item` に似ているが、表現が異なっている。`button-item` の場合にボタンの上に簡単なラベル文字列を描く代わりに、`bitmap-button-item` では、ボタンが作られたときに bitmap ファイルからロードされる pixmap を描く。

:create *bitmap-file reciever method &rest args* [メソッド]
&key :width :height
&allow-other-keys)

bitmap-button-item を作成する。最初の引き数 *bitmap-file* は、`button-item` の *label* 引き数を置き換えたものである。

:draw-label *&optional (state :flat) (color bg-color) (border 2)* [メソッド]
ボタンの上に bitmap か pixmap を描く。

:create-bitmap-from-file *fname* [メソッド]
fname という名の bitmap ファイルから pixmap を作り、`pixmap-id` にその ID を入れる。

choice-item [クラス]

:super	**button-item**
:slots	(choice-list active-choice transient-choice choice-dots choice-pos button-size)

choice-item は、丸い選択ボタンの集合である。１つの選択はいつも活性化しており、同時に１つの選択だけが活性化することができる。**choice-item** は、ラジオボタンと同様な機能を提供する。

:create *label reciever method &rest args* [メソッド]
&key (:choices '("0" "1")) (:initial-choice 0)
(:font (send parent :gc :font))
(:button-size 13)
(:border-width 0)

choice-item-button を作成する。それぞれの選択ボタンは*:button-size* の半径を持つ円である。新しい選択ボタンが選択されたとき、`notify-object` の `notify-method` が choice-item オブジェクトと選択された選択ボタンの番号と一緒に呼び出される。

:value *&optional (new-choice) (invocation)* [メソッド]
もし、*new-choice* が与えられたならば、現在の活性化選択ボタンとして設定し、対応する円が黒色になる。もし *invocation* も指定されているなら、`notify-object` の `notify-method` が呼び出される。`:value` は、現在の（あるいは新しい）選択ボタンの番号を返す。

:draw-active-button *&optional (old-choice active-choice) (new-choice active-choice)* [メソッド]
ボタンを活性化として書く。

:ButtonPress *event* [メソッド]
もし、選択ボタンのどこかでマウスボタンが押されているなら、その番号が `transient-choice` に記録される。マウスボタンが外されるまでそれ以上の行動は、起こさない。

:ButtonRelease *event* [メソッド]
もし、既に押されていたところと同じボタンの上でマウスボタンが外されたなら、`active-choice` が更新され、`notify-object` の `notify-method` が呼び出される。

slider-item [クラス]

:super	**panel-item**
:slots	(min-value max-value value minlabel maxlabel valueformat

bar-x bar-y bar-width bar-height valuedots label-base
nob-x nob-moving
charwidth)

`choice-item` が離散的な値の選択に使用されるのに対し、**slider-item** は `min-value` と `max-value` の間の範囲の連続的な値に対して使用される。それぞれ値が変化した瞬間、slider-item オブジェクトと新しい値が引き数として一緒に `notify-object` の `notify-method` が呼び出される。

:create *label reciever method &rest args* [メソッド]
&key (:min 0.0) (:max 1.0) (:parent)
(:min-label "") (:max-label "") (:value-format " 4,2f")
(:font font-courb12) (:span 100) (:border-width 0) (:initial-value min)

slider-item を作成する。スライドのノブは、バーの上に小さな黒の四角として表示される。左端が *:min* 値を表現し、右端が*:max* 値を表現する。バーの長さは、*:span* ドットに引き伸ばす。現在の値は、*:value-format* で slider-item の右に表示する。

:value *&optional newval invocation* [メソッド]
もし、*newval* が与えられたなら、現在の値として設定され、ノブは対応する位置にスライドする。もし、*invocation* も non NIL に指定されていたなら、`notify-object` の `notify-method` が呼び出される。`:value` は、現在の（新しい）値を返す。

joystick-item [クラス]

:super **panel-item**
:slots (stick-size min-x min-y max-x max-y
center-x center-y stick-x stick-y
value-x value-y
stick-return stick-grabbed
fraction-x fraction-y)

joystick-item は、2 次元の slider-item としてみなすことができる。2 つの連続値はクモの巣のような同心円図の上を動く黒い円によって指定することができる (図 18)。

:create *name reciever method &rest args* [メソッド]
&key (:stick-size 5) (:return nil)
(:min-x -1.0) (:max-x 1.0)
(:min-y -1.0) (:max-y 1.0)
&allow-other-keys)

:stick-size は、スティックの黒い円の半径である。同心円図の円の大きさは、**joystick-item** window の幅と高さに合うように決定される。もし、*:return* が non NIL であるなら、ジョイスティックは、マウスボタンが外された時の原点に帰る。そうでないなら、ジョイスティックは、外された位置に残る。

:value *&optional (newx) (newy) (invocation)* [メソッド]
もし、*newx* と *newy* が与えられたなら、現在の位置として設定され、ジョイスティックは同心円図の対応する位置に移動する。もし、*invocation* も non NIL に指定されたなら、`notify-object` の `notify-method` が、**joystick-item** オブジェクトと x,y 値を引き数として一緒に呼び出される。`:value` は、現在の（新しい）値のリストを返す。

以下に上に記述されている panel-item を使った短いプログラムを示し、図 18 がパネルの中にどのように表示されるかを示したものである。

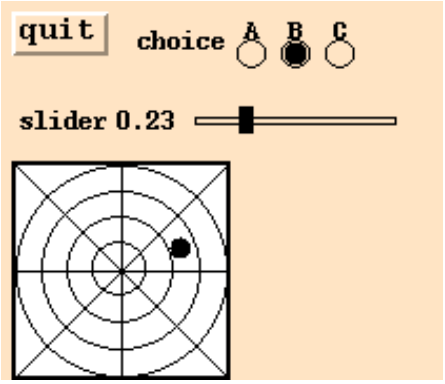

図 18: panel の中に作成された panel-item

```
(in-package "X")
(defclass testPanel :super panel
        :slots (quit joy choi sli))
(defmethod testPanel
 (:create (&rest args)
    (send-super* :create :width 210 :height 180
                 :font font-courb12 args)
    (send-super :create-item button-item "quit" self :quit :font font-courb14)
    (send-super :create-item choice-item "choice" self :choice
                :choices '(" A " " B " " C ")
                :font font-courb12)
    (send-super :create-item slider-item "slider" self :slider
                :span 90)
    (send-super :create-item joystick-item "joy" self :joy)
    self)
 (:choice (obj c) (format t "choice: ~S ~d~%" obj c))
 (:slider (obj val) (format t "slider: ~S ~s~%" obj val))
 (:joy (obj x y) (format t "joy: ~S ~s ~s~%" obj x y)) )
(instance testPanel :create)
(window-main-thread)
```

text-item [クラス]

:super **panel-item**
:slots (textwin)

text-item は、ファイル名のような短いテキストを表示したり入力したりするために使用する。**text-item** は、ラベル文字列とその右側に小さなテキスト window を持っている。テキスト window 内にポインタが置かれたとき、キー入力が可能となり、入力された文字がバッファに入れられる。テキスト window 内の行修正が可能である。`control-F` と `control-B` は前後に 1 文字動かし、`del` はカーソルの左の 1 文字を削除し、`control-D` はカーソル位置の文字を削除し、カーソル位置にはどんなグラフィック文字も挿入できる。マウスボタンをクリックすれば、クリックされた文字にカーソルを移動させる。enter（改行）キーを打つことにより、バッファされたテキストが `notify-object` の `notify-method` に送られる。

:create *label revciever method &rest args* [メソッド]
&key (:font font-courb12) (:columns 20) (:initial-value) (:border-width 0)
&allow-other-keys

text-item を作成する。テキスト window の行バッファには、長さの制限が無いけれども、見える部分は *columns* 文字に制限されている。

:getstring [メソッド]
キーバッファ内の文字列を返す。

17.4 キャンバス

canvas [クラス]

:super	**xwindow**
:slots	(topleft bottomright)

canvas は、図や画像を入れるための Xwindow である。現在、領域選択機能のみ実現されている。ButtonPress イベントにより、**canvas** は押された位置を左上の端とし、現在の位置を右下の端とする四角を描き始める。ButtonRelease により、`notify-object` の `notify-method` が送られる。canvas 内に図や画像を描くためには Xdrawable のメソッドが使用される。

17.5 テキスト window

`TextWindow` と `BufferTextWidnow` と `ScrollTextWindow` の 3 つのテキスト window がある。

textWindow [クラス]

:super	**xwindow**	
:slots	(fontid	
	charwidth charheight charascent dots	
	win-row-max win-col-max	
	win-row win-col	;physical current position in window
	x y	
	charbuf	; for charcode conversion
	keybuf keycount	;for key input
	echo	
	show-cursor cursor-on	;boolean
	kill delete	;control character
	notify-object notify-method	
	)	

メッセージを表示するために使用可能な仮想端末を実現する。表示される内容は、バッファされないし、**TextWindow** に既に表示された文字や行を引き出す方法はない。基本的に、**TextWindow** はカーソル移動、行削除、領域削除、表示テキストのスクロール、文字列挿入などを持つダンプ端末と似た能力を持っている。また、テキストカーソルはマウスポインタで指示された位置に移動することができる。

:init *id* [メソッド]

id 番目のテキスト window を初期化する。

:create *&rest args* [メソッド]

&key :width :height (:font font-courb14) :rows :columns
(:show-cursor nil) (:notify-object nil) (:notify-method nil)
&allow-other-keys

text-window を作成する。window の大きさは、*:width* と *:height* かあるいは *:rows* と *:columns* で指定されたものとなる。*:notify-object* の *:notify-method* は、改行文字が入力されたときに呼び出される。

:cursor *flag* [メソッド]

flag は、`:on`, `:off`, `:toggle` のどれかが可能である。テキストカーソルは、`win-row` と `win-col` の位置である。もし、*flag* が `:on` であれば、テキストカーソルは表示され、*flag* が `:off` ならば、消される。また、*flag* が `toggle` ならば、反対になる。このメソッドは、カーソルの位置の文字を更新するときはいつでも、呼び出されなければならない。

:clear [メソッド]

テキスト window 内を消去する。

:clear-eol *&optional (r win-row) (c win-col) (csr :on)* [メソッド]

r と *c* で指定される位置の文字以降の残りの行を消去する。カーソル位置の文字も含む。

:clear-lines *lines &optional (r win-row)* [メソッド]

r 番目の行以降の複数行を消去する。

:clear-eos *&optional (r win-row) (c win-col)* [メソッド]

r と *c* で指定される位置から画面の最後までの領域を消去する。

:win-row-max [メソッド]

この window に表示可能な最大行数を返す。

:win-col-max [メソッド]

この window に表示可能な最大列数を返す。

:xy *&optional (r win-row) (c win-col)* [メソッド]

r と *c* で指定される位置の文字のピクセル座標を計算する。

:goto *r c &optional (cursor :on)* [メソッド]

r 番目の行の *c* 番目の列にカーソルを移動する。

:goback *&optional (csr :on)* [メソッド]

カーソルを 1 文字戻す。

:advance *&optional (n 1)* [メソッド]

n 文字だけカーソルを進める。

:scroll *&optional (n 1)* [メソッド]

n 行だけ縦方向にテキスト window をスクロールする。

:horizontal-scroll *&optional (n 1)* [メソッド]

n 列だけ横方向にテキストをスクロールする。

:newline [メソッド]

次の行の最初にカーソルを移動する。

:putch *ch* [メソッド]
カーソル位置に文字 *ch* を挿入する。行の残りは、１文字前方に進められる。

:putstring *str &optional (e (length str))* [メソッド]
カーソル位置に *str* を置く。

:event-row *event* [メソッド]

:event-col *event* [メソッド]
event における、テキストカーソルの位置をを返す。

:KeyPress *event* [メソッド]
カーソル位置に入力された文字を挿入する。もし、文字が改行であったなら、`notify-object` に通知する。

TextWindowStream [クラス]

:super **stream**
:slots (textwin)

`TextWindowStream` は、TextWdinow に接続された出力ストリームである。`print, format, write-byte` などによってこのストリームに出力される文字や文字列が TextWdindow に表示される。通常のファイルストリームとしては、出力データはバッファされる。

:flush [メソッド]
バッファされたテキスト文字列を掃き出し、TextWindow に送る。`finish-output` やこのストリームへの改行文字の書き込みは、このメソッドを自動的に呼び出す。

make-text-window-stream *xwin* [関数]
text-window-stream を作り、そのストリームオブジェクトを返す。

BufferTextWindow [クラス]

:super **TextWindow**
:slots (linebuf expbuf max-line-length row col)

TextWindow の内容を表現する行バッファを保持する。`linebuf` は、行のベクトルである。`exbuf` は、タブ拡張されたテキストを持つ。window に表示可能な行のみが保持される。**BufferTextWindow** は、数行のテキストを持つ簡単なテキストエディタとして使用することができる。`text-item` は、表示可能な行バッファとして **BufferTextWindow** を使う。

:line *n* [メソッド]
n 番目の行の内容を文字列として返す。

:nlines [メソッド]
`linebuf` の行数を返す。

:all-lines [メソッド]
文字列のベクトルである `linebuf` を返す。

:refresh-line *&optional (r win-row) (c win-col)* [メソッド]

r 番目の行の *c* 番目の列以降を再書き込みする。

:refresh *&optional (start 0)* [メソッド]
start 番目の行以降の行を再書き込みする。

:insert-string *string* [メソッド]
カーソル位置に *string* を挿入する。

:insert *ch* [メソッド]
カーソル位置に文字を挿入する。

:delete *n* [メソッド]
カーソル以降の *n* 文字を削除する。

expand-tab *src &optional (offset 0)* [関数]
src は、タブを含んだ可能性のある文字列である。これらのタブは、タブの停止位置が 8 の倍数であると仮定して空白文字に置き換えられる。

ScrollTextWindow [クラス]

:super	**BufferTextWindow**	
:slots	(top-row top-col	;display-starting position
	scroll-bar-window	
	horizontal-scroll-bar-window	
	selected-line)	

ScrollTextWindow は、行数制限がなく、縦と横にスクロールバーを持った `BufferTextWindow` を定義する。**ScrollTextWindow** は、スクロールバーを伴って window の大きさを変更するため、あるいはテキストを再表示するために`:configureNotify` イベントを扱うことができる。クリックすることによって、行を選択することができる。

:create *&rest args &key (scroll-bar nil) (horizontal-scroll-bar nil) &allow-other-keys* [メソッド]
スクロールバーが必要なとき、それぞれのキーワードの引き数に T を指定する。

:locate *n* [メソッド]
window の上部から *n* 番目の行にバッファされたテキストを表示する。

:display-selection *selection* [メソッド]
selection は、選択された行の位置を表現する。選択された行がすべて高輝度で表示される。

:selection [メソッド]
選択された行（文字列）を返す。

:read-file *fname* [メソッド]
fname で指定されるテキストファイルを `linebuf` に読み込み、タブを拡張し、window に表示する。カーソルは、画面の最初に置かれる。

:display-string *strings* [メソッド]
strings は、行（文字列）の列である。*strings* は、`linebuf` にコピーされ、window に表示される。

:scroll *n* [メソッド]
n 行、縦にスクロールする。

:horizontal-scroll *n* [メソッド]

n 列、横にスクロールする。

:ButtonRelease *event* [メソッド]

マウスポインタが置かれている行が選択される。もし、window が作成されたときに `notification` が指定されているならば、`notify-object` の `notify-method` が呼び出される。

:resize *w h* [メソッド]

window の大きさを変更し、新しいサイズに合うように内容を再表示する。もし、スクロールバーがあれば、同じメッセージが送られる。

第III部
irteus 拡張

18　ロボットモデリング

18.1　ロボットのデータ構造とモデリング

18.1.1　ロボットのデータ構造と順運動学

ロボットの構造はリンクと関節から構成されていると考えることが出来るが，ロボットを関節とリンクに分割する方法として

- (a) 切り離されるリンクの側に関節を含める
- (b) 胴体，あるいは胴体に近いほうに関節を含める

が考えられる．コンピュータのデータ構造を考慮し，(a) が利用されている．その理由は胴体以外のすべてのリンクにおいて，必ず関節を一つ含んだ構造となり，すべてのリンクを同一のアルゴリズムで扱うことが出きるためである．

この様に分割されたリンクを計算機上で表現するためにはツリー構造を利用することが出来る．一般的にはツリー構造を作るときに二分木にすることでデータ構造を簡略化することが多い．

ロボットのリンクにおける同次変換行列の求め方としては，関節回転座標系上に原点をもつ Σ_j を設定し，親リンク座標系からみた回転軸ベクトルが a_j, Σ_j の原点が b_j であり，回転の関節角度を q_j とする．

このとき Σ_j の親リンク相対の同次変換行列は

$$ {}^iT_j = \begin{bmatrix} e^{\hat{a}_j q_j} & b_j \\ 0\,0\,0 & 1 \end{bmatrix} $$

と書くことが出来る．

ここで，$e^{\hat{a}_j q_j}$ は，一定速度の角速度ベクトルによって生ずる回転行列を与える以下の Rodrigues の式を用いている．これを回転軸 a 周りに $wt[rad]$ だけ回転する回転行列を与えるものとして利用している．

$$ e^{\hat{\omega} t} = E + \hat{a} sin(wt) + \hat{a}^2(1 - cos(wt)) $$

親リンクの位置姿勢 p_i, R_i が既知だとすると，Σ_i の同次変換行列を

$$ T_i = \begin{bmatrix} R_i & p_i \\ 0\,0\,0 & 1 \end{bmatrix} $$

と作ることができ，ここから

$$ T_j = T_i \ {}^iT_j $$

として計算できる．これをロボットのルートリンクから初めてすべてのリンクに順次適用することでロボットの全身の関節角度情報から姿勢情報を算出することができ，これを順運動学と呼ぶ．

18.1.2　EusLisp による幾何情報のモデリング

Euslisp の幾何モデリングでは，基本モデル (body) の生成，body の合成関数，複合モデル (bodyset) の生成と 3 つの段階がある．

これまでに以下のような基本モデルの生成，合成が可能な事を見てきている．

```
(setq c1 (make-cube 100 100 100))
(send c1 :locate #f(0 0 50))
(send c1 :rotate (deg2rad 30) :x)
(send c1 :set-color :yellow)
(objects (list c1))

(setq c2 (make-cylinder 50 100))
(send c2 :move-to
      (make-coords
       :pos #f(20 30 40)
       :rpy (float-vector 0 0 (deg2rad 90)))
      :world)
(send c2 :set-color :green)
(objects (list c1 c2))

(setq c3 (body+ c1 c2))
(setq c4 (body- c1 c2))
(setq c5 (body* c1 c2))
```

bodyset は irteus で導入された複合モデルであり，body で扱えない複数の物体や複数の色を扱うためのものである．

```
(setq c1 (make-cube 100 100 100))
(send c1 :set-color :red)
(setq c2 (make-cylinder 30 100))
(send c2 :set-color :green)
(send c1 :assoc c2)           ;;; これを忘れいように
(setq b1 (instance bodyset :init
                   (make-cascoords)
                   :bodies (list c1 c2)))
(objects (list b1))
```

18.1.3 幾何情報の親子関係を利用したサンプルプログラム

```
(setq c1 (make-cube 100 100 100))
(setq c2 (make-cube 50 50 50))
(send c1 :set-color :red)
(send c2 :set-color :blue)
(send c2 :locate #f(300 0 0))
(send c1 :assoc c2)
(objects (list c1 c2))
(do-until-key
 (send c1 :rotate (deg2rad 5) :z)
 (send *irtviewer* :draw-objects)
 (x::window-main-one) ;; process window event
 )
```

18.1.4 bodyset-link と joint を用いたロボット（多リンク系）のモデリング

irteus ではロボットリンクを記述するクラスとして bodyset-link(irtmodel.l) というクラスが用意されている．これは機構情報と幾何情報をもち，一般的な木構造でロボットの構造が表現されている．また，joint クラスを用いて関節情報を扱っている．

```
(defclass bodyset-link
  :super bodyset
  :slots (joint parent-link child-links analysis-level default-coords
                weight acentroid inertia-tensor
                angular-velocity angular-acceleration
                spacial-velocity spacial-acceleration
                momentum-velocity angular-momentum-velocity
                momentum angular-momentum
                force moment ext-force ext-moment))
```

ジョイント（関節）のモデリングは joint クラス (irtmodel.l) を用いる．joint クラスは基底クラスであり，実際には rotational-joint, linear-joint 等を利用する．joint の子クラスで作られた関節は，:joint-angle メソッドで関節角度を指定することが出来る．

```
(defclass joint
  :super propertied-object
  :slots (parent-link child-link joint-angle min-angle max-angle
default-coords))
(defmethod joint
  (:init (&key name
               ((:child-link clink)) ((:parent-link plink))
               (min -90) (max 90) &allow-other-keys)
         (send self :name name)
         (setq parent-link plink child-link clink
               min-angle min max-angle max)
         self))

(defclass rotational-joint
  :super joint
  :slots (axis))
(defmethod rotational-joint
  (:init (&rest args &key ((:axis ax) :z) &allow-other-keys)
         (setq axis ax joint-angle 0.0)
         (send-super* :init args)
         self)
  (:joint-angle
   (&optional v)
   (when v
        (setq relang (- v joint-angle) joint-angle v)
        (send child-link :rotate (deg2rad relang) axis)))
     joint-angle))
```

ここでは，joint, parent-link, child-links, defualt-coords を利用する．
簡単な 1 関節ロボットの例としてサーボモジュールを作ってみると

```
(defun make-servo nil
  (let (b1 b2)
    (setq b1 (make-cube 35 20 46))
    (send b1 :locate #f(9.5 0 0))
    (setq b2 (make-cylinder 3 60))
    (send b2 :locate #f(0 0 -30))
    (setq b1 (body+ b1 b2))
    (send b1 :set-color :gray20)
    b1))

(defun make-hinji nil
  (let ((b2 (make-cube 22 16 58))
        (b1 (make-cube 26 20 54)))
    (send b2 :locate #f(-4 0 0))
    (setq b2 (body- b2 b1))
    (send b1 :set-color :gray80)
    b2))

(setq h1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-hinji))))
(setq s1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-servo))))
(setq j1 (instance rotational-joint :init :parent-link h1 :child-link s1 :axis :z))
;; instance cascaded coords
(setq r (instance cascaded-link :init))
(send r :assoc h1)
(send h1 :assoc s1)
(setq (r . links) (list h1 s1))
(setq (r . joint-list) (list j1))
(send r :init-ending)
```

となる．

ここでは，h1，s1 という bodyset-link と，j1 という rotational-joint を作成し，ここから cascaded-link

という，連結リンクからなるモデルを生成している．cascaded-link は cascaded-coords の子クラスであるため，r（cascaded-link），h1，s1 の座標系の親子関係を:assoc を利用して設定している．

（r . links）という記法は r というオブジェクトのスロット変数（メンバ変数）である links にアクセスしている．ここでは，links および joint-list に適切な値をセットし，(send r :init-ending) として必要な初期設定を行っている．

これで r という 2 つのリンクと 1 つの関節情報を含んだ 1 つのオブジェクトを生成できる．これで例えば (objects (list h1 s1)) ではなく，(objects (list r)) としてロボットをビューワに表示できる．また，(send r :locate #f(100 0 0)) などの利用も可能になっている．

cascaded-link クラスのメソッドの利用例としては以下ある．:joint-list，:links といった関節リストやリンクリストへのアクセサに加え，関節角度ベクトルへのアクセスを提供する:angle-vector メソッドが重要である．これを引数なしで呼び出せば現在の関節角度が得られ，これに関節角度ベクトルを引数に与えて呼び出せば，その引数が示す関節角度ベクトルをロボットモデルに反映させることができる．

```
$ (objects (list r))
(#<servo-model #X628abb0  0.0 0.0 0.0 / 0.0 0.0 0.0>)
;; useful cascaded-link methods
$ (send r :joint-list)
(#<rotational-joint #X6062990 :joint101067152>)
$ (send r :links)
(#<bodyset-link #X62ccb10 :bodyset103598864  0.0 0.0 0.0 / 0.0 0.0 0.0>
 #<bodyset-link #X6305830 :bodyset103831600  0.0 0.0 0.0 / 0.524 0.0 0.0>)
$ (send r :angle-vector)
#f(0.0)
$ (send r :angle-vector (float-vector 30))
#f(30.0)
```

18.1.5 cascaded-link を用いたロボット（多リンク系）のモデリング

一方で多リンク系のモデリング用のクラスとして cascaded-link というクラスがある．これには，links, joint-list というスロット変数があり，ここに bodyset-link, ならびに joint のインスタンスのリストをバインドして利用する．以下は，cascaded-link の子クラスを定義しここでロボットモデリングに関する初期化処理を行うという書き方の例である．

```
(defclass cascaded-link
  :super cascaded-coords
  :slots (links joint-list bodies collision-avoidance-links))

(defmethod cascaded-link
  (:init (&rest args &key name &allow-other-keys)
         (send-super-lexpr :init args)
         self)
  (:init-ending
   ()
   (setq bodies (flatten (send-all links :bodies)))
   (dolist (j joint-list)
     (send (send j :child-link) :add-joint j)
     (send (send j :child-link) :add-parent-link (send j :parent-link))
     (send (send j :parent-link) :add-child-links (send j :child-link)))
   (send self :update-descendants))
)


(defclass servo-model
  :super cascaded-link
  :slots (h1 s1 j1))
(defmethod servo-model
  (:init ()
   (let ()
     (send-super :init)
     (setq h1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-hinji))))
     (setq s1 (instance bodyset-link :init (make-cascoords) :bodies (list (make-servo))))
```

```
    (setq j1 (instance rotational-joint :init :parent-link h1 :child-link s1 :axis :z))

    ;; instance cascaded coords
    (setq links (list h1 s1))
    (setq joint-list (list j1))
    ;;
    (send self :assoc h1)
    (send h1 :assoc s1)
    ;;
    (send self :init-ending)
    self))
  ;;
  ;; (send r :j1 :joint-angle 30)
  (:j1 (&rest args) (forward-message-to j1 args))
  )

(setq r (instance servo-model :init))
```

このようなクラスを定義して (setq r (instance servo-model :init)) としても同じようにロボットモデルのインスタンスを作成することができ，先ほどのメソッドを利用できる．クラス定義するメリットは (:j1 (&rest args) (forward-message-to j1 args)) というメソッド定義により，関節モデルのインスタンスへのアクセサを提供することができる．これにより，特定の関節だけの値を知りたいとき，あるいは値をセットしたい時には (send r :j1 :joint-angle) や (send r :j1 :joint-angle 30) という指示が可能になっている．このロボットを動かす場合は前例と同じように

```
(objects (list r))
(dotimes (i 300)
  (send r :angle-vector (float-vector (* 90 (sin (/ i 100.0)))))
  (send *irtviewer* :draw-objects))
```

などとするとよい．

```
(setq i 0)
(do-until-key
  (send r :angle-vector (float-vector (* 90 (sin (/ i 100.0)))))
  (send *irtviewer* :draw-objects)
  (incf i))
```

とすると，次にキーボードを押下するまでプログラムは動きつづける．

さらに，少し拡張してこれを用いて３リンク２ジョイントのロボットをモデリングした例が以下になる．:joint-angle というメソッドに#f(0 0) というベクトルを引数に与えることで全ての関節角度列を指定することが出来る．

```
(defclass hinji-servo-robot
  :super cascaded-link)
(defmethod hinji-servo-robot
  (:init
   ()
   (let (h1 s1 h2 s2 l1 l2 l3)
     (send-super :init)
     (setq h1 (make-hinji))
     (setq s1 (make-servo))
     (setq h2 (make-hinji))
     (setq s2 (make-servo))
     (send h2 :locate #f(42 0 0))
     (send s1 :assoc h2)
     (setq l1 (instance bodyset-link :init (make-cascoords) :bodies (list h1)))
     (setq l2 (instance bodyset-link :init (make-cascoords) :bodies (list s1 h2)))
     (setq l3 (instance bodyset-link :init (make-cascoords) :bodies (list s2)))
     (send l3 :locate #f(42 0 0))

     (send self :assoc l1)
```

```
    (send l1 :assoc l2)
    (send l2 :assoc l3)

    (setq joint-list
          (list
           (instance rotational-joint
                     :init :parent-link l1 :child-link l2
                     :axis :z)
           (instance rotational-joint
                     :init :parent-link l2 :child-link l3
                     :axis :z)))
    (setq links (list l1 l2 l3))
    (send self :init-ending)
    )))
(setq r (instance hinji-servo-robot :init))
(objects (list r))

(dotimes (i 10000)
  (send r :angle-vector (float-vector (* 90 (sin (/ i 500.0))) (* 90 (sin (/ i 1000.0)))))
  (send *irtviewer* :draw-objects))
```

18.2 ロボットの動作生成

18.2.1 逆運動学

逆運動学においては, エンドエフェクタの位置・姿勢 ${}^0_n\boldsymbol{H}$ からマニピュレータの関節角度ベクトル $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n)^T$ を求める.

ここで, エンドエフェクタの位置・姿勢 $\boldsymbol{r}$ は関節角度ベクトルを用いて

$$\boldsymbol{r} = \boldsymbol{f}(\boldsymbol{\theta}) \tag{1}$$

とかける. Equation 18.2.1 は Equation 18.2.1 のように記述し, 関節角度ベクトルを求める.

$$\boldsymbol{\theta} = \boldsymbol{f}^{-1}(\boldsymbol{r}) \tag{2}$$

における f^{-1} は一般に非線形な関数となる. そこでを時刻 t に関して微分することで, 線形な式

$$\dot{\boldsymbol{r}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{3}$$

$$= \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \tag{4}$$

を得る. ここで, $\boldsymbol{J}(\boldsymbol{\theta})$ は $m \times n$ のヤコビ行列である. m はベクトル $\boldsymbol{r}$ の次元, n はベクトル $\boldsymbol{\theta}$ の次元である. $\dot{\boldsymbol{r}}$ は速度・角速度ベクトルである.

ヤコビ行列が正則であるとき逆行列 $\boldsymbol{J}(\boldsymbol{\theta})^{-1}$ を用いて以下のようにしてこの線型方程式の解を得ることができる.

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}(\boldsymbol{\theta})^{-1}\dot{\boldsymbol{r}} \tag{5}$$

しかし, 一般にヤコビ行列は正則でないので, ヤコビ行列の疑似逆行列 $\boldsymbol{J}^{\#}(\boldsymbol{\theta})$ が用いられる (Equation 6).

$$\boldsymbol{A}^{\#} = \begin{cases} \boldsymbol{A}^{-1} \ (m = n = rank\boldsymbol{A}) \\ \boldsymbol{A}^T \ (\boldsymbol{A}\boldsymbol{A}^T)^{-1} (n > m = rank\boldsymbol{A}) \\ (\boldsymbol{A}^T\boldsymbol{A})^{-1}\boldsymbol{A}^T \ (m > n = rank\boldsymbol{A}) \end{cases} \tag{6}$$

Equation 4 は, $m > n$ のときは Equation 7 を, $n >= m$ のときは Equation 9 を, 最小化する最小二乗解を求める問題と捉え, 解を得る.

$$\min_{\dot{\boldsymbol{\theta}}} \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)^T \left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right) \tag{7}$$

$$\begin{aligned} \min_{\dot{\boldsymbol{\theta}}} \quad & \dot{\boldsymbol{\theta}}^T\dot{\boldsymbol{\theta}} \\ s.t. \quad & \dot{\boldsymbol{r}} = \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}} \end{aligned} \tag{8}$$

関節角速度は次のように求まる.

$$\dot{\boldsymbol{\theta}} = \boldsymbol{J}^{\#}(\boldsymbol{\theta})\dot{\boldsymbol{r}} + \left(\boldsymbol{E}_n - \boldsymbol{J}^{\#}(\boldsymbol{\theta})\boldsymbol{J}(\boldsymbol{\theta})\right)\boldsymbol{z} \tag{9}$$

しかしながら, Equation 9 に従って解を求めると, ヤコビ行列 $\boldsymbol{J}(\boldsymbol{\theta})$ がフルランクでなくなる特異点に近づくと, $\left|\dot{\boldsymbol{\theta}}\right|$ が大きくなり不安定な振舞いが生じる. そこで, Nakamura et al. の SR-Inverse[4] を用いることで, この特異点を回避する.

本研究ではヤコビ行列の疑似逆行列 $\boldsymbol{J}^{\#}(\boldsymbol{\theta})$ の代わりに, Equation 10 に示す $\boldsymbol{J}^{*}(\boldsymbol{\theta})$ を用いる.

$$\boldsymbol{J}^{*}(\boldsymbol{\theta}) = \boldsymbol{J}^T\left(\boldsymbol{J}\boldsymbol{J}^T + \epsilon\boldsymbol{E}_m\right)^{-1} \tag{10}$$

これは, Equation 7 の代わりに, Equation 11 を最小化する最適化問題を解くことにより得られたものである.

$$\min_{\dot{\boldsymbol{\theta}}}\{\dot{\boldsymbol{\theta}}^T\dot{\boldsymbol{\theta}} + \epsilon\left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)^T\left(\dot{\boldsymbol{r}} - \boldsymbol{J}(\boldsymbol{\theta})\dot{\boldsymbol{\theta}}\right)\} \tag{11}$$

ヤコビ行列 $\boldsymbol{J}(\boldsymbol{\theta})$ が特異点に近づいているかの指標には可操作度 $\kappa(\boldsymbol{\theta})$[5] が用いられる (Equation 12).

$$\kappa(\boldsymbol{\theta}) = \sqrt{\boldsymbol{J}(\boldsymbol{\theta})\boldsymbol{J}^T(\boldsymbol{\theta})} \tag{12}$$

微分運動学方程式におけるタスク空間次元の選択行列[6] は見通しの良い定式化のために省略するが, 以降で導出する全ての式において適用可能であることをあらかじめことわっておく.

18.2.2 基礎ヤコビ行列

一次元対偶を関節に持つマニピュレータのヤコビアンは基礎ヤコビ行列[7] により計算することが可能である. 基礎ヤコビ行列の第 j 関節に対応するヤコビアンの列ベクトル $\boldsymbol{J}_j$ は

$$\boldsymbol{J}_j = \begin{cases} \begin{bmatrix} \boldsymbol{a}_j \\ \boldsymbol{0} \end{bmatrix} & \text{if linear joint} \\ \begin{bmatrix} \boldsymbol{a}_j \times (\boldsymbol{p}_{end} - \boldsymbol{p}_j) \\ \boldsymbol{a}_j \end{bmatrix} & \text{if rotational joint} \end{cases} \tag{13}$$

と表される. $\boldsymbol{a}_j$・$\boldsymbol{p}_j$ はそれぞれ第 j 関節の関節軸単位ベクトル・位置ベクトルであり, $\boldsymbol{p}_{end}$ はヤコビアンで運動を制御するエンドエフェクタの位置ベクトルである. 上記では 1 自由度対偶の回転関節・直動関節について導出したが，その他の関節でもこれらの列ベクトルを連結した行列としてヤコビアンを定義可能である．非全方位台車の運動を表す 2 自由度関節は前後退の直動関節と旋回のための回転関節から構成できる．全方位台車の運動を表す 3 自由度関節は並進 2 自由度の直動関節と旋回のための回転関節から構成できる．球関節は姿勢を姿勢行列で，姿勢変化を等価角軸変換によるものとすると，3 つの回転関節をつなぎ合わせたものとみなせる.

[4] Inverse kinematic solutions with singularity robustness for robot manipulator control: Y.Nakamura and H. Hanafusa, Journal of Dynamic Systems, Measurement, and Control, vol. 108, pp 163-171, 1986

[5] ロボットアームの可操作度, 吉川恒夫, 日本ロボット学会誌, vol. 2, no. 1, pp. 63-67, 1984.

[6] Hybrid Position/Force Control: A Correct Formuration, William D. Fisher, M. Shahid Mujtaba, The Internationaltional Journal of Robotics Research, vol. 11, no. 4, pp. 299-311, 1992.

[7] A unified approach for motion and force control of robot manipulators: The operational space formulation, O. Khatib, IEEE Journal of Robotics and Automation, vol. 3, no. 1, pp. 43-53, 1987.

18.2.3 関節角度限界回避を含む逆運動学

ロボットマニピュレータの軌道生成において, 関節角度限界を考慮することはロボットによる実機実験の際に重要となる. 本節では, 文献[8] [9] の式および文章を引用しつつ, 関節角度限界の回避を含む逆運動学について説明する.

重み付きノルムを以下のように定義する.

$$\left|\dot{\boldsymbol{\theta}}\right|_{\boldsymbol{W}} = \sqrt{\dot{\boldsymbol{\theta}}^T \boldsymbol{W} \dot{\boldsymbol{\theta}}} \tag{14}$$

ここで, $\boldsymbol{W}$ は $\boldsymbol{W} \in \boldsymbol{R}^{n \times n}$ であり, 対象で全ての要素が正である重み係数行列である. この $\boldsymbol{W}$ を用いて, $\boldsymbol{J}_{\boldsymbol{W}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{W}}$ を以下のように定義する.

$$\boldsymbol{J}_{\boldsymbol{W}} = \boldsymbol{J}\boldsymbol{W}^{-\frac{1}{2}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{W}} = \boldsymbol{W}^{\frac{1}{2}} \dot{\boldsymbol{\theta}} \tag{15}$$

この $\boldsymbol{J}_{\boldsymbol{W}}, \dot{\boldsymbol{\theta}}_{\boldsymbol{W}}$ を用いて, 以下の式を得る.

$$\dot{\boldsymbol{r}} = \boldsymbol{J}_{\boldsymbol{W}} \dot{\boldsymbol{\theta}}_{\boldsymbol{W}} \tag{16}$$

$$\left|\dot{\boldsymbol{\theta}}\right|_{\boldsymbol{W}} = \sqrt{\dot{\boldsymbol{\theta}}_{\boldsymbol{W}}^T \dot{\boldsymbol{\theta}}_{\boldsymbol{W}}} \tag{17}$$

これによって線型方程式の解は[9]から以下のように記述できる.

$$\dot{\boldsymbol{\theta}}_{\boldsymbol{W}} = \boldsymbol{W}^{-1} \boldsymbol{J}^T \left(\boldsymbol{J} \boldsymbol{W}^{-1} \boldsymbol{J}^T \right)^{-1} \dot{\boldsymbol{r}} \tag{18}$$

また、現在の関節角度 θ が関節角度限界 $\theta_{i,\max}, \theta_{i,\min}$ に対してどの程度余裕があるかを評価するための関数 $H(\boldsymbol{\theta})$ は以下のようになる[10]).

$$H(\boldsymbol{\theta}) = \sum_{i=1}^{n} \frac{1}{4} \frac{(\theta_{i,\max} - \theta_{i,\min})^2}{(\theta_{i,\max} - \theta_i)(\theta_i - \theta_{i,\min})} \tag{19}$$

次に Equation 20 に示すような $n \times n$ の重み係数行列 $\boldsymbol{W}$ を考える.

$$\boldsymbol{W} = \begin{bmatrix} w_1 & 0 & 0 & \cdots & 0 \\ 0 & w_2 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \ddots & \cdots \\ 0 & 0 & 0 & \cdots & w_n \end{bmatrix} \tag{20}$$

ここで w_i は

$$w_i = 1 + \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| \tag{21}$$

である.

さらに Equation 19 から次の式を得る.

$$\frac{\partial H(\boldsymbol{\theta})}{\partial \theta_i} = \frac{(\theta_{i,\max} - \theta_{i,\min})^2 (2\theta_i - \theta_{i,\max} - \theta_{i,\min})}{4(\theta_{i,\max} - \theta_i)^2 (\theta_i - \theta_{i,\min})^2} \tag{22}$$

[8] Exploiting Task Intervals for Whole Body Robot Control, Michael Gienger and Herbert Jansen and Christian Goeric In Proceedings of the 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06), pp. 2484 - 2490, 2006

[9] A weighted least-norm solution based scheme for avoiding jointlimits for redundant joint manipulators, Tan Fung Chan and Dubey R.V., Robotics and Automation, IEEE Transactions on, pp. 286-292,1995

[10] Efficient gradient projection optimization for manipulators withmultiple degrees of redundancy, Zghal H., Dubey R.V., Euler J.A., 1990 IEEE International Conference on Robotics and Automation, pp. 1006-1011, 1990.

関節角度限界から遠ざかる向きに関節角度が動いている場合には重み係数行列を変化させる必要はないので,w_i を以下のように定義しなおす.

$$w_i = \begin{cases} 1 + \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| & if \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| \geq 0 \\ 1 & if \left| \frac{\partial \boldsymbol{H}(\boldsymbol{\theta})}{\partial \theta_i} \right| < 0 \end{cases} \tag{23}$$

この w_i および $\boldsymbol{W}$ を用いることで関節角度限界回避を含む逆運動学を解くことができる.

18.2.4 衝突回避を含む逆運動学

ロボットの動作中での自己衝突や環境モデルとの衝突は幾何形状モデルが存在すれば計算することが可能である. ここではSugiura et al. により提案されている効率的な衝突回避計算[11] [12] を応用した動作生成法を示す. 実際の実装はSugiura et al. の手法に加え, タスク作業空間のNullSpaceの利用を係数により制御できるようにした点や擬似逆行列ではなくSR-Inverseを用いて特異点にロバストにしている点などが追加されている.

18.2.5 衝突回避のための関節角速度計算法

逆運動学計算における目標タスクと衝突回避の統合はリンク間最短距離を用いたblending係数により行われる. これにより, 衝突回避の必要のないときは目標タスクを厳密に満し衝突回避の必要性があらわれたときに目標タスクをあきらめて衝突回避の行われる関節角速度計算を行うことが可能になる. 最終的な関節角速度の関係式はEquation 24 で得られる. 以下では ca の添字は衝突回避計算のための成分を表し, $task$ の部分は衝突回避計算以外のタスク目標を表すことにする.

$$\dot{\boldsymbol{\theta}} = f(d)\dot{\boldsymbol{\theta}}_{ca} + (1 - f(d))\,\dot{\boldsymbol{\theta}}_{task} \tag{24}$$

blending係数 $f(d)$ は, リンク間距離 d と閾値 d_a・d_b の関数として計算される (Equation 25).

$$f(d) = \begin{cases} (d - d_a)/(d_b - d_a) & if d < d_a \\ 0 & otherwise \end{cases} \tag{25}$$

d_a は衝突回避計算を行い始める値 (yellow zone[12]) であり, d_b は目標タスクを阻害しても衝突回避を行う閾値 (orange zone[12]) である.

衝突計算をする2リンク間の最短距離・最近傍点が計算できた場合の衝突を回避するための動作戦略は2リンク間に作用する仮想的な反力ポテンシャルから導出される.

2リンク間の最近傍点同士をつなぐベクトル $\boldsymbol{p}$ を用いた2リンク間反力から導出される速度計算をEquation 26 に記す.

$$\boldsymbol{\delta x} = \begin{cases} 0 & if\,|\boldsymbol{p}| > d_a \\ (d_a/\,|\boldsymbol{p}| - 1)\boldsymbol{p} & else \end{cases} \tag{26}$$

これを用いた関節角速度計算はEquation 27 となる.

$$\dot{\boldsymbol{\theta}}_{ca} = \boldsymbol{J}_{ca}^T k_{joint} \boldsymbol{\delta x} + (\boldsymbol{E}_n - \boldsymbol{J}_{task}^* \boldsymbol{J}_{task}) \boldsymbol{J}_{ca}^T k_{null} \boldsymbol{\delta x} \tag{27}$$

[11] Real-Time Self Collision Avoidance for Humanoids by means of Nullspace Criteria and Task Intervals, H. Sugiura, M. Gienger, H. Janssen, C. Goerick, Proceedings of the 2006 IEEE-RAS International Conference on Humanoid Robots, pp. 575-580, 2006.

[12] Real-time collision avoidance with whole body motion control for humanoid robots, Hisashi Sugiura, Michael Gienger, Herbert Janssen, Christian Goerick, In Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07), pp. 2053 - 2068, 2007

k_{joint}・k_{null} はそれぞれ反力ポテンシャルを目標タスクの NullSpace に分配するかそうでないかを制御する係数である.

18.2.6 衝突回避計算例

以下ではロボットモデル・環境モデルを用いた衝突回避例を示す. 本研究では, ロボットのリンク同士, またはリンクと物体の衝突判定には, 衝突判定ライブラリ PQP(A Proximity Query Package) [13]を用いた.

Fig.19 では $d_a = 200[mm]$,$d_b = 0.1 * d_a = 20[mm]$, $k_{joint} = k_{null} = 1.0$ と設定した.

この衝突判定計算では, 衝突判定をリンクの設定を

1. リンクのリスト n_{ca} を登録
2. 登録されたリンクのリストから全リンクのペア ${}_{n_{ca}}C_2$ を計算
3. 隣接するリンクのペア, 常に交わりを持つリンクのペアなどを除外

のように行うという工夫を行っている.

Fig.19 例では衝突判定をするリンクを「前腕リンク」「上腕リンク」「体幹リンク」「ベースリンク」の 4 つとして登録した. この場合, ${}_4C_2$ 通りのリンクのペア数から隣接するリンクが除外され, 全リンクペアは「前腕リンク-体幹リンク」「前腕リンク-ベースリンク」「上腕リンク-ベースリンク」の 3 通りとなる.

Fig.19 の 3 本の線 (赤 1 本, 緑 2 本) が衝突形状モデル間での最近傍点同士をつないだ最短距離ベクトルである. 全リンクペアのうち赤い線が最も距離が近いペアであり, このリンクペアより衝突回避のための逆運動学計算を行っている.

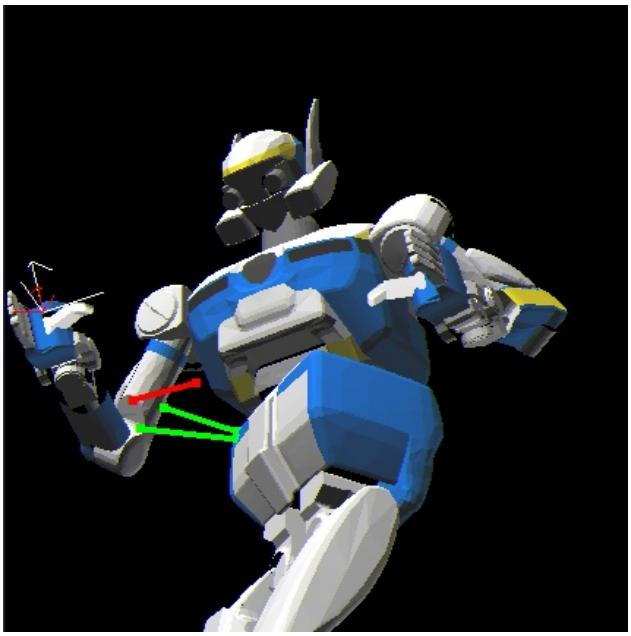

図 19: Example of Collision Avoidance

[13] Fast distance queries with rectangular swept sphere volumes, Larsen E., Gottschalk S., Lin M.C., Manocha D, Proceedings of The 2000 IEEE International Conference on Robotics and Automation, pp. 3719-3726, 2000.

18.2.7 非ブロック対角ヤコビアンによる全身協調動作生成

ヒューマノイドは枝分かれのある複雑な構造を持ち, 複数のマニピュレータで協調して動作を行う必要がある (Fig.20).

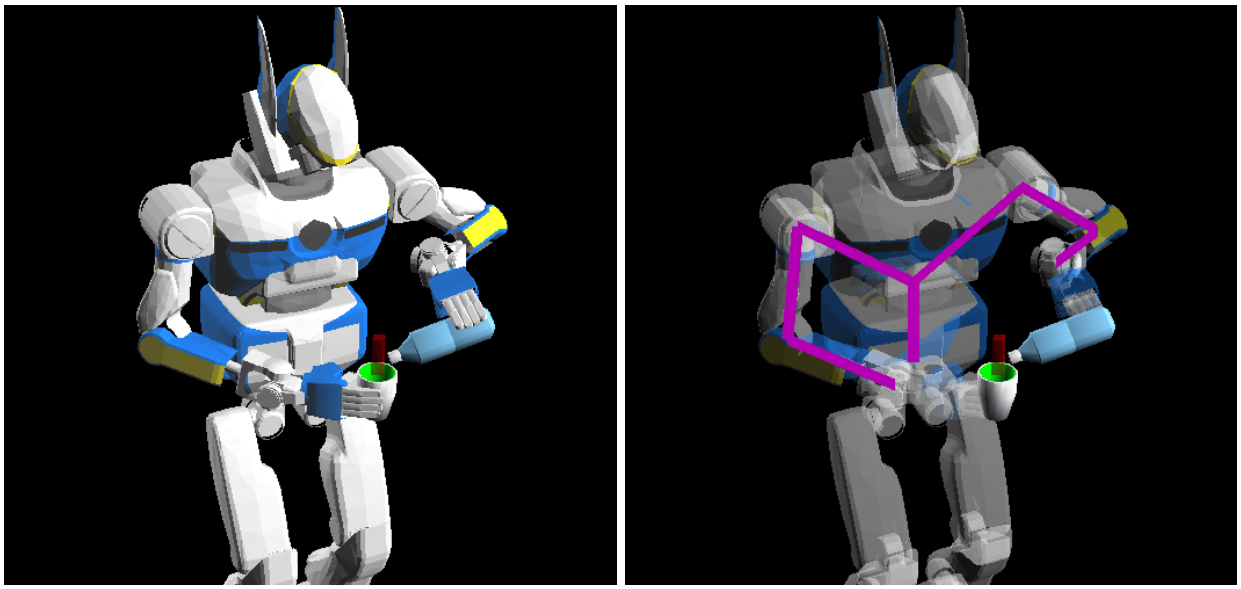

図 20: Duplicate Link Sequence

複数マニピュレータの動作例として,

- リンク間に重複がない場合
  それぞれのマニピュレータについて Equation 5 式を用いて関節角速度を求める. もしくは, 複数の式を連立した方程式 (ヤコビアンはブロック対角行列となる) を用いて関節角速度を求めても良い.
- リンク間に重複がある場合
  リンク間に重複がある場合は, リンク間の重複を考慮したヤコビアンを考える必要がある. 例えば, 双腕動作を行う場合, 左腕のマニピュレータのリンク系列と右腕のマニピュレータのリンク系列とで, 体幹部リンク系列が重複し, その部位は左右で協調して関節角速度を求める必要がある.

次節ではリンク間に重複がある場合の非ブロック対角なヤコビアンの計算法およびそれを用いた関節角速度計算法を述べる (前者の重複がない場合も以下の計算方法により後者の一部として計算可能である).

18.2.8 リンク間重複があるヤコビアン計算と関節角度計算

微分運動学方程式を求める際の条件を以下に示す.

- マニピュレータの本数　L 本
- 全関節数　N 個
- マニピュレータの先端速度・角速度ベクトル　$[\boldsymbol{\xi}_0^T, ..., \boldsymbol{\xi}_{L-1}^T]^T$
- 各関節角速度ベクトル　$[\dot{\boldsymbol{\theta}}_{\mathbf{0}}{}^T, ..., \dot{\boldsymbol{\theta}}_{\boldsymbol{L-1}}{}^T]^T$
- 関節の添字和集合　$S = \{0, \ldots, N-1\}$
  ただし, マニピュレータ i の添字集合 S_i を用いて S は $S = S_0 \cup \ldots \cup S_{L-1}$ と表せる.

- S に基づく関節速度ベクトル　$[\dot{\theta}_0, ..., \dot{\theta}_{N-1}]^T$

とする.

運動学関係式は Equation 28 のようになる.

$$\begin{bmatrix} \boldsymbol{\xi}_0 \\ \vdots \\ \boldsymbol{\xi}_{L-1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_{0,0} & \cdots & \boldsymbol{J}_{0,N-1} \\ \vdots & \boldsymbol{J}_{i,j} & \vdots \\ \boldsymbol{J}_{L-1,0} & \cdots & \boldsymbol{J}_{L-1,N-1} \end{bmatrix} \begin{bmatrix} \dot{\theta}_0 \\ \vdots \\ \dot{\theta}_{N-1} \end{bmatrix} \tag{28}$$

小行列 $\boldsymbol{J}_{i,j}$ は以下のように求まる.

$$\boldsymbol{J}_{i,j} = \begin{cases} \boldsymbol{J}_j & \text{if } j\text{-th joint} \in i\text{-th link array} \quad (29) \\ \boldsymbol{0} & \text{otherwise} \quad (30) \end{cases}$$

ここで，$\boldsymbol{J}_j$ は Equation 13 のもの.

Equation 28 を単一のマニピュレータの逆運動学解法と同様に SR-Inverse を用いて関節角速度を求めることができる.

ここでの非ブロック対角ヤコビアンの計算法は, アーム・多指ハンドの動作生成[14]において登場する運動学関係式から求まるヤコビアンを導出することが可能である.

18.2.9　ベースリンク仮想ジョイントを用いた全身逆運動学法

一般に関節数が N であるのロボットの運動を表現するためにはベースリンクの位置姿勢と関節角自由度を合わせた $N+6$ 個の変数が必要である．ベースリンクとなる位置姿勢の変数を用いたロボットの運動の定式化は宇宙ロボット[15]だけでなく, 環境に固定されないヒューマノイドロボット[16]の場合にも重要である.

ここでは腕・脚といったマニピュレータにベースリンクに 3 自由度の直動関節と 3 自由度の回転関節が仮想的に付随したマニピュレータ構成を考える (Fig.21). 上記の仮想的な 6 自由度関節を本研究ではベースリンク仮想ジョイントと名づける. ベースリンク仮想ジョイントを用いることによりヒューマノイドの腰が動き全身関節が駆動され, 運動学, ひいては動力学的な解空間が拡充されることが期待できる.

18.2.10　ベースリンク仮想ジョイントヤコビアン

ベースリンク仮想ジョイントのヤコビアンは基礎ヤコビ行列の計算 (Equation 13) を利用し，絶対座標系 x，y，z 軸の直動関節と絶対座標系 x，y，z 軸回りの回転関節をそれぞれ連結した 6×6 行列である．ちなみに, 並進・回転成分のルートリンク仮想ジョイントのヤコビアンは以下のように書き下すこともできる.

$$\boldsymbol{J}_{B,l} = \begin{bmatrix} \boldsymbol{E}_3 & -\hat{\boldsymbol{p}}_{B\to l} \\ \boldsymbol{0} & \boldsymbol{E}_3 \end{bmatrix} \tag{31}$$

$\boldsymbol{p}_{B\to l}$ はベースリンク位置から添字 l で表現する位置までの差分ベクトルである.

[14] アーム・多指ハンド機構による把握と操り, 永井 清, 吉川 恒夫, 日本ロボット学会誌, vol. 13, no. 7, pp. 994-1005, 1995.
[15] 一般化ヤコビ行列を用いた宇宙用ロボットマニピュレータの分解速度制御, 梅谷 陽二, 吉田 和哉, 日本ロボット学会誌, vol. 4, no. 7, pp. 63-73, 1989.
[16] Control of Free-Floating Humanoid Robots Through Task Prioritization, Luis Sentis and Oussama Khatib, Proceedings of The 2005 IEEE International Conference on Robotics and Automation, pp. 1718-1723, 2005

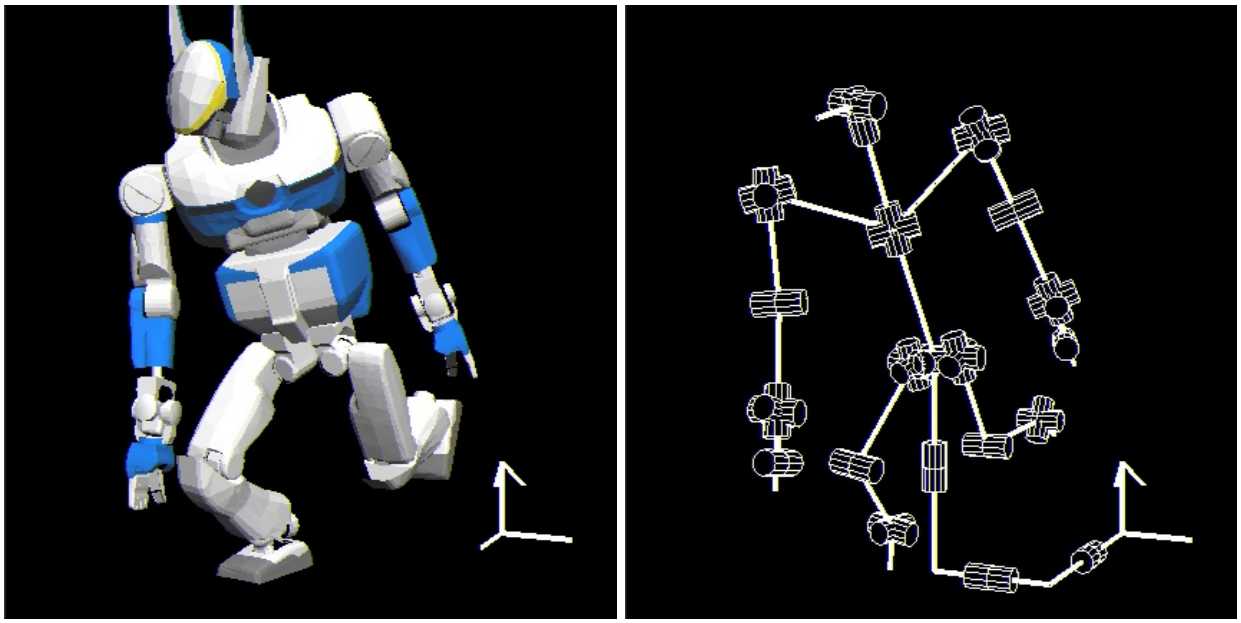

図 21: Concept of the Virtual Joint of the Base Link
(Left figure) Overview of the Robot Model
(Right figure) Skeleton Figure of Robot Model with the Virtual Joint

18.2.11 マスプロパティ計算

複数の質量・重心・慣性行列を統合し単一の質量・重心・慣性行列の組 $[m_{new}, \boldsymbol{c}_{new}, \boldsymbol{I}_{new}]$ を計算する演算関数を次のように定義する．

$$[m_{new}, \boldsymbol{c}_{new}, \boldsymbol{I}_{new}] = AddMassProperty([m_1, \boldsymbol{c}_1, \boldsymbol{I}_1], \ldots, [m_K, \boldsymbol{c}_K, \boldsymbol{I}_K]) \tag{32}$$

これは次のような演算である．

$$m_{new} = \sum_{j=1}^{K} m_j \tag{33}$$

$$\boldsymbol{c}_{new} = \frac{1}{m_{new}} \sum_{j=1}^{K} m_j \boldsymbol{c}_j \tag{34}$$

$$\boldsymbol{I}_{new} = \sum_{j=1}^{K} (\boldsymbol{I}_j + m_j \boldsymbol{D}(\boldsymbol{c}_j - \boldsymbol{c}_{new})) \tag{35}$$

ここで，$\boldsymbol{D}(\boldsymbol{r}) = \hat{\boldsymbol{r}}^T \hat{\boldsymbol{r}}$ とする．

18.2.12 運動量・角運動量ヤコビアン

シリアルリンクマニピュレータを対象とし，運動量・角運動量ヤコビアンを導出する．運動量・原点まわりの角運動量を各関節変数で表現し，その偏微分でヤコビアンの行を計算する．

第 j 関節の運動変数を θ_j とする．まず，回転・並進の 1 自由度関節を考える．

$$\boldsymbol{P}_j = \begin{cases} \boldsymbol{a}_j \dot{\theta}_j \times (\tilde{\boldsymbol{c}}_j - \boldsymbol{p}_j) \tilde{m}_j & \text{if rotational joint} \\ \boldsymbol{a}_j \dot{\theta}_j \tilde{m}_j & \text{if linear joint} \end{cases} \tag{36}$$

$$\boldsymbol{L}_j = \begin{cases} \tilde{\boldsymbol{c}}_j \boldsymbol{P}_j + \tilde{\boldsymbol{I}}_j \boldsymbol{a}_j \dot{\theta}_j & \text{if rotational joint} \\ \boldsymbol{0} & \text{if linear joint} \end{cases} \tag{37}$$

ここで，$[\tilde{m}_j, \tilde{\boldsymbol{c}}_j, \tilde{\boldsymbol{I}}_j]$ は AddMassProperty 関数に第 j 関節の子リンクより末端側のリンクのマスプロパティを与えたものであり，実際には再帰計算により計算する[17]．これらを $\dot{\theta}_j$ で割ることによりヤコビアンの各列ベクトルを得る．

$$\boldsymbol{m}_j = \begin{cases} \boldsymbol{a}_j \times (\tilde{\boldsymbol{c}}_j - \boldsymbol{p}_j)\tilde{m}_j & \text{if rotational joint} \\ \boldsymbol{a}_j \tilde{m}_j & \text{if linear joint} \end{cases} \tag{38}$$

$$\boldsymbol{h}_j = \begin{cases} \tilde{\boldsymbol{c}}_j \times \boldsymbol{m}_j + \tilde{\boldsymbol{I}}_j \boldsymbol{a}_j & \text{if rotational joint} \\ \boldsymbol{0} & \text{if linear joint} \end{cases} \tag{39}$$

これより慣性行列は次のように計算できる．

$$\boldsymbol{M}_{\dot{\boldsymbol{\theta}}} = [\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N] \tag{40}$$

$$\boldsymbol{H}_{\dot{\boldsymbol{\theta}}} = [\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N] - \hat{\boldsymbol{p}}_G \boldsymbol{M}_{\dot{\boldsymbol{\theta}}} \tag{41}$$

ここでは，全関節数を N とした．また，ベースリンクは直動関節 x，y，z 軸，回転関節 x，y，z 軸をもつと考え整理し，次のようになる．

$$\begin{bmatrix} \boldsymbol{M}_B \\ \boldsymbol{H}_B \end{bmatrix} = \begin{bmatrix} M_r \boldsymbol{E}_3 & -M_r \hat{\boldsymbol{p}}_{B\to G} \\ \boldsymbol{0} & \tilde{\boldsymbol{I}} \end{bmatrix} \tag{42}$$

これを用いて重心まわりの角運動量・運動量は次のようになる．

$$\begin{bmatrix} \boldsymbol{P} \\ \boldsymbol{L} \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}_B & \boldsymbol{M}_{\dot{\boldsymbol{\theta}}} \\ \boldsymbol{H}_B & \boldsymbol{H}_{\dot{\boldsymbol{\theta}}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\xi}_B \\ \dot{\boldsymbol{\theta}} \end{bmatrix} \tag{43}$$

ここでヒューマノイドの全質量 M_r，重心位置 $\boldsymbol{p}_G$，慣性テンソル $\tilde{\boldsymbol{I}}$ は次のように全リンクのマスプロパティ演算より求める．

$$[M_r, \boldsymbol{p}_G, \tilde{\boldsymbol{I}}] = AddMassProperty([m_1, \boldsymbol{c}_1, \boldsymbol{I}_1], \ldots, [m_N, \boldsymbol{c}_N, \boldsymbol{I}_N]) \tag{44}$$

18.2.13 重心ヤコビアン

重心ヤコビアンは重心速度と関節角速度の間のヤコビアンである．本論文ではベースリンク仮想ジョイントを用いるため，ベースリンクに 6 自由度関節がついたと考えベースリンク速度角速度・関節角速度の重心速度に対するヤコビアンを重心ヤコビアンとして用いる．具体的には，ベースリンク成分 $\boldsymbol{M}_B$ と使用関節について抜き出した成分 $\boldsymbol{M}'_{\dot{\boldsymbol{\theta}}}$ による運動量ヤコビアンを全質量で割ることで重心ヤコビアンを計算する．

$$\boldsymbol{J}_G = \frac{1}{M_r} \begin{bmatrix} \boldsymbol{M}_B & \boldsymbol{M}'_{\dot{\boldsymbol{\theta}}} \end{bmatrix} \tag{45}$$

18.3 ロボットの動作生成プログラミング

18.3.1 三軸関節ロボットを使ったヤコビアン，逆運動学の例

3 軸関節をもつロボットを定義し，逆運動学やヤコビアンの計算例を紹介する．
ロボットの定義は以下の用になる．

[17] Resolved Momentum Control:Humanoid Motion Planning based on the Linear and Angular Momentum, S.Kajita, F.Kanehiro, K.Kaneko, K.Fujiwara, K.Harada, K.Yokoi,H.Hirukawa, In Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'03), pp. 1644-1650, 2003

```
(defclass 3dof-robot
  :super cascaded-link
  :slots (end-coords l1 l2 l3 l4 j1 j2 j3))
(defmethod 3dof-robot
  (:init ()
   (let (b)
     (send-super :init)

     (setq b (make-cube 10 10 20))
     (send b :locate #f(0 0 10))
     (send b :set-color :red)
     (setq l4 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l4))
     (setq end-coords (make-cascoords :pos #f(0 0 20)))
     (send l4 :assoc end-coords)
     (send l4 :locate #f(0 0 100))
     ;;
     (setq b (make-cube 10 10 100))
     (send b :locate #f(0 0 50))
     (send b :set-color :green)
     (setq l3 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l3))
     (send l3 :assoc l4)
     (send l3 :locate #f(0 0 100))
     ;;
     (setq b (make-cube 10 10 100))
     (send b :locate #f(0 0 50))
     (send b :set-color :blue)
     (setq l2 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l2))
     (send l2 :assoc l3)
     (send l2 :locate #f(0 0 20))
     ;;
     (setq b (body+ (make-cube 10 10 20 :pos #f(0 0 10)) (make-cube 300 300 2)))
     (send b :set-color :white)
     (setq l1 (instance bodyset-link :init (make-cascoords) :bodies (list b) :name 'l2))
     (send l1 :assoc l2)
     ;;
     (setq j1 (instance rotational-joint :init :name 'j1
                :parent-link l1 :child-link l2 :axis :y :min -100 :max 100)
           j2 (instance rotational-joint :init  :name 'j2
                :parent-link l2 :child-link l3 :axis :y :min -100 :max 100)
           j3 (instance rotational-joint :init  :name 'j3
                :parent-link l3 :child-link l4 :axis :y :min -100 :max 100))
     ;;
     (setq links (list l1 l2 l3 l4))
     (setq joint-list (list j1 j2 j3))
     ;;
     (send self :init-ending)
     self))
  (:end-coords (&rest args) (forward-message-to end-coords args))
  )
```

ここではロボットの手先の座標を end-coords というスロット変数に格納し，さらにこれにアクセスするためのメソッドを用意してある．

これまでと同様，

```
(setq r (instance 3dof-robot :init))
(objects (list r))
(send r :angle-vector #f(30 30 30))
```

としてロボットモデルの生成，表示，関節角度の指定が可能である．さらに，

```
(send (send r :end-coords) :draw-on :flush t)
```

とすると，ロボットの end-coords(端点座標系）の表示が可能であるが，マウスイベントが発生すると消えてしまう．恒久的に表示するためには

```
(objects (list r (send r :end-coords)))
```

とするとよい.
次に，ヤコビアン，逆運動学の例を示す．まず基本になるのが,

```
(send r :link-list (send r :end-coords :parent))
```

として得られるリンクのリストである．これはロボットのルート（胴体）から引数となるリンクまでのたどれるリンクを返す.

:calc-jacobian-from-link-list メソッドはリンクのリストを引数にとり，この各リンクに存在するジョイント（関節）に対応するヤコビアンを計算することができる．また，:move-target キーワード引数でエンドエフェクタの座標系を指定してる．その他のキーワード引数については後述する.

```
(dotimes (i 100)
  (setq j (send r :calc-jacobian-from-link-list
                (send r :link-list (send r :end-coords :parent))
                :move-target (send r :end-coords)
                :rotation-axis t
                :translation-axis t))
  (setq j# (sr-inverse j))
  (setq da (transform j# #f(1 0 0 0 0 0)))
  ;;(setq da (transform j# #f(0 0 0 0 -1 0)))
  (send r :angle-vector (v+ (send r :angle-vector) da))
  (send *irtviewer* :draw-objects)
  )
```

ここではリンクの長さ（ジョイントの数）は 3 個なので 6 行 3 列のヤコビアン (j) が計算される．これの逆行列 (j#) を作り，位置姿勢の 6 自由度の目標速度・角速度 (#f(1 0 0 0 0 0)) を与えると，それに対応する関節速度 (da) が計算でき，これを現在の関節角度に足している ((v+ (send r :angle-vector) da)).

次に，ロボットの端点作業の位置は合わせるが姿勢は拘束せず任意のままでよい，という場合の例を示す．ここでは，:calc-jacobian-from-link-list のオプショナル引数として:rotation-axis, :translation-axis があり，それぞれ位置，姿勢での拘束条件を示す．t は三軸拘束，nil は拘束なし，その他に:x, :y, :z を指定することができる.

```
(setq translation-axis t)
(setq rotation-axis nil)
(dotimes (i 2000)
  (setq j (send r :calc-jacobian-from-link-list
                (send r :link-list (send r :end-coords :parent))
                :move-target (send r :end-coords)
                :rotation-axis rotation-axis
                :translation-axis translation-axis))
  (setq j# (sr-inverse j))
  (setq c (make-cascoords :pos (float-vector (* 100 (sin (/ i 500.0))) 0 200)))
  (setq dif-pos (send (send r :end-coords) :difference-position c))
  (setq da (transform j# dif-pos))
  (send r :angle-vector (v+ (send r :angle-vector) da))
  (send *irtviewer* :draw-objects :flush nil)
  (send c :draw-on :flush t)
  )
```

ここでは位置の三軸のみを拘束した 3 行 3 列のヤコビアンを計算し，これの逆行列からロボットの関節に速度を与えている．さらに，ここでは

```
(send *irtviewer* :draw-objects :flush nil)
```

として*irtviewer*に画面を描画しているが，実際にディスプレイに表示するフラッシュ処理は行わず，その次の行の

```
(send c :draw-on :flush t)
```

で目標座標は表示し，かつフラッシュ処理を行っている.
上記の計算をまとめた逆運動学メソッドが:inverse-kinematics である．第一引数に目標座標系を指定し，ヤコビアン計算のときと同様にキーワード引数で:move-target, :translation-axis, :rotation-axis を指定する．また，:debug-view キーワード引数に t を与えると計算中の様子をテキスト並びに視覚的に提示してくれる.

```
(setq c (make-cascoords :pos #f(100 0 0) :rpy (float-vector 0 pi 0)))
(send r :inverse-kinematics  c
      :link-list (send r :link-list (send r :end-coords :parent))
      :move-target (send r :end-coords)
      :translation-axis t
      :rotation-axis t
      :debug-view t)
```

逆運運動学が失敗する場合のサンプルとして以下のプログラムを見てみよう.

```
(dotimes (i 400)
  (setq c (make-cascoords
            :pos (float-vector (+ 100 (* 80 (sin (/ i 100.0)))) 0 0)
            :rpy (float-vector 0 pi 0)))
  (send r :inverse-kinematics  c
        :link-list (send r :link-list (send r :end-coords :parent))
        :move-target (send r :end-coords) :translation-axis t  :rotation-axis t)
  (x::window-main-one)
  (send *irtviewer* :draw-objects :flush nil)
  (send c :draw-on :flush t)
  )
```

このプログラムを実行すると以下のようなエラーが出てくる.

```
;; inverse-kinematics failed.
;; dif-pos : #f(11.7826 0.0 0.008449)/(11.7826/1)
;; dif-rot : #f(0.0 2.686130e-05 0.0)/(2.686130e-05/0.017453)
;;  coords : #<coordinates #X4bcccb0  0.0 0.0 0.0 / 0.0 0.0 0.0>
;;  angles : (14.9993 150 15.0006)
;;    args : ((#<cascaded-coords #X4b668a0  39.982 0.0 0.0 / 3.142 1.225e-16 3.14
2>) :link-list (#<bodyset-link #X4cf8e60 l2  0.0 0.0 20.0 / 0.0 0.262 0.0> #<body
set-link #X4cc8008 l3  25.866 0.0 116.597 / 3.142 0.262 3.142> #<bodyset-link #X4
c7a0d0 l4  51.764 0.0 20.009 / 3.142 2.686e-05 3.142>) :move-target;; #<cascaded-
coords #X4c93640  51.764 0.0 0.009 / 3.142 2.686e-05 3.142> :translation-axis t :
rotation-axis t)
```

これは，関節の駆動範囲の制限から目標位置に手先が届かない状況である．このような場面では，例えば，手先の位置さえ目標位置に届けばよく姿勢を無視してよい場合:rotation-axis nil と指定することができる.

また，:thre や:rthre を使うことで逆運動学計算の終了条件である位置姿勢の誤差を指定することができる．正確な計算が求められていない状況ではこの値をデフォルトの 1, (deg2rad 1) より大きい値を利用するのもよい.

また, 逆運動学の計算に失敗した場合, デフォルトでは逆運動学計算を始める前の姿勢まで戻るが, :revert-if-fail というキーワード引数を nil と指定すると，指定されたの回数の計算を繰り替えしたあと，その姿勢のまま関数から抜けてくる．指定の回数もまた，:stop というキーワード引数で指定することができる.

```
(setq c (make-cascoords :pos #f(300 0 0) :rpy (float-vector 0 pi 0)))
(send r :inverse-kinematics  c
      :link-list (send r :link-list (send r :end-coords :parent))
      :move-target (send r :end-coords)
      :translation-axis t
      :rotation-axis nil
      :revert-if-fail nil)
```

18.3.2 irteus のサンプルプログラムにおける例

cascaded-coords クラスでは

- (:link-list (to &optional form))
- (:calc-jacobian-from-link-list (link-list &key move-target (rotation-axis nil)))

というメソッドが用意されている．

前者はリンクを引数として，ルートリンクからこのリンクまでの経路を計算し，リンクのリストとして返す．後者はこのリンクのリストを引数とし，move-target 座標系をに対するヤコビアンを計算する．

concatenate result-type a b は a b を連結し result-type 型に変換し返し，scale a b はベクトル b の全ての要素をスカラー a 倍し，matrix-log は行列対数関数を計算する．

```
(if (not (boundp '*irtviewer*)) (make-irtviewer))

(load "irteus/demo/sample-arm-model.l")
(setq *sarm* (instance sarmclass :init))
(send *sarm* :reset-pose)
(setq *target* (make-coords :pos #f(350 200 400)))
(objects (list *sarm* *target*))

(do-until-key
  ;; step 3
  (setq c (send *sarm* :end-coords))
  (send c :draw-on :flush t)
  ;; step 4
  ;; step 4
  (setq dp (scale 0.001 (v- (send *target* :worldpos) (send c :worldpos))) ;; mm->m
        dw (matrix-log (m* (transpose (send c :worldrot)) (send *target* :worldrot))))
  (format t "dp = ~7,3f ~7,3f ~7,3f, dw = ~7,3f ~7,3f ~7,3f~%"
          (elt dp 0) (elt dp 1) (elt dp 2)
          (elt dw 0) (elt dw 1) (elt dw 2))
  ;; step 5
  (when (< (+ (norm dp) (norm dw)) 0.01) (return))
  ;; step 6
  (setq ll (send *sarm* :link-list (send *sarm* :end-coords :parent)))
  (setq j (send *sarm* :calc-jacobian-from-link-list
                ll :move-target (send *sarm* :end-coords)
                :trnaslation-axis t :rotation-axis t))
  (setq q (scale 1.0 (transform (pseudo-inverse j) (concatenate float-vector dp dw))))
  ;; step 7
  (dotimes (i (length ll))
    (send (send (elt ll i) :joint) :joint-angle (elt q i) :relative t))
  ;; draw
  (send *irtviewer* :draw-objects)
  (x::window-main-one))
```

実際のプログラミングでは:inverse-kinematics というメソッドが用意されており，ここでは特異点や関節リミットの回避，あるいは自己衝突回避等の機能が追加されている．

18.3.3 実際のロボットモデル

実際のロボットや環境を利用した実践的なサンプルプログラムを見てみよう．

まず，最初はロボットや環境のモデルファイルを読み込む．これらのファイルは$EUSDIR/models に，これらのファイルをロードしインスタンスを生成するプログラムは以下のように書くことができる．(room73b2) や (h7) はこれらのファイル内で定義されている関数である．ロボットのモデル (robot-model) は irtrobot.l ファイルで定義されており，cascaded-link クラスの子クラスになっている．ロボットとは larm,rarm,lleg,rleg,head のリンクのツリーからなるものとして定義されており，(send *robot* :larm) や (send *robot* :head) としてロボットのリム (limb) にアクセスでき，右手の逆運動学，左手の逆運動学等という利用方法が可能になっている．

```
(load "models/room73b2-scene.l")
(load "models/h7-robot.l")
(setq *room* (room73b2))
(setq *robot* (h7))
(objects (list *robot* *room*))
```

ロボットには:reset-pose というメソッドがありこれで初期姿勢をとることができる．

```
(send *robot* :reset-pose)
```

次に，ロボットを部屋の中で移動させたい．部屋内の代表的な座標は (send *room* :spots) で取得できる．この中から目的の座標を得る場合はその座標の名前を引数として:spot メソッドを呼び出す．ちなみに，このメソッドの定義は prog/jskeus/irteus/irtscene.l にあり

```
(defmethod scene-model
  (:spots
   (&optional name)
   (append
    (mapcan
     #'(lambda(x)(if (derivedp x scene-model) (send x :spots name) nil))
     objs)
    (mapcan #'(lambda (o)
(if (and (eq (class o) cascaded-coords)
 (or (null name) (string= name (send o :name))))
    (list o)))
    objs)))
  (:spot
   (name)
   (let ((r (send self :spots name)))
     (case (length r)
       (0 (warning-message 1 "could not found spot(~A)" name) nil)
       (1 (car r))
       (t (warning-message 1 "found multiple spot ~A for given name(~A)" r name) (car r)))))
  )
```

となっている．
ロボットもまた coordinates クラスの子クラスなので:move-to メソッドを利用できる．また，このロボットの原点は腰にあるので足が地面につくように:locate メソッドを使って移動する．

```
(send *robot* :move-to (send *room* :spot "cook-spot") :world)
(send *robot* :locate #f(0 0 550))
```

現状では*irtviewer*の画面上でロボットが小さくなっているので，以下のメソッド利用し，ロボットが画面いっぱいになるように調整する．

```
(send *irtviewer* :look-all
      (geo::make-bounding-box
       (flatten (send-all (send *robot* :bodies) :vertices))))
```

次に環境中の物体を選択する．ここでは:object メソッドを利用する．これは，:spots，:spot と同様の振る舞いをするため，どのような物体があるかは，(send-all (send *room* :objects) :name) として知ることができる．room73b2-kettle の他に room73b2-mug-cup や room73b2-knife 等を利用するとよい．

```
(setq *kettle* (send *room* :object "room73b2-kettle"))
```

環境モデルの初期化直後は物体は部屋に assoc されているため，以下の用に親子関係を解消しておく．こうしないと物体を把持するなどの場合に問題が生じる．

```
(if (send *kettle* :parent) (send (send *kettle* :parent) :dissoc *kettle*))
```

ロボットの視線を対象物に向けるためのメソッドとして以下のようなものがある．

```
(send *robot* :head :look-at (send *kettle* :worldpos))
```

対象物体には，その物体を把持するための利用したらよい座標系が:handle メソッドとして記述されている場合がある．このメソッドはリストを返すため以下の様に (car (send *kettle* :handle)) としてその座標系を知ることができる．この座標がどこにあるか確認するためには (send (car (send *kettle* :handle)) :draw-on :flush とするとよい．
したがってこの物体手を伸ばすためには

```
(send *robot* :larm :inverse-kinematics
      (car (send *kettle* :handle))
      :link-list (send *robot* :link-list (send *robot* :larm :end-coords :parent))
      :move-target (send *robot* :larm :end-coords)
      :rotation-axis :z
      :debug-view t)
```

となる.
ここで，ロボットの手先と対象物体の座標系を連結し，

```
(send *robot* :larm :end-coords :assoc *kettle*)
```

以下の様にして世界座標系で 100[mm] 持ち上げることができる.

```
(send *robot* :larm :move-end-pos #f(0 0 100) :world
        :debug-view t :look-at-target t)
```

`:look-at-target` は移動中に首の向きを常に対象を見つづけるようにするという指令である.

18.4 ロボットモデル

ロボットの身体はリンクとジョイントから構成されるが、それぞれ `bodyset-link` と `joint` クラスを利用しモデル絵を作成する。ロボットの身体はこれらの要素を含んだ `cascaded-link` という，連結リンクとしてモデルを生成する.

実際には `joint` は抽象クラスであり `rotational-joint`,`linear-joint`, `wheel-joint`,`omniwheel-joint`, `sphere-joint` を選択肢、また四肢を持つロボットの場合は `cascaded-link` ではなく `robot-model` クラスを利用する。

joint [クラス]

:super **propertied-object**
:slots parent-link child-link joint-angle min-angle max-angle default-coords joint-velocity joi

:init *&key (name (intern (format nil joint A (system:address self)) KEYWORD))* [method]
((:child-link clink))
((:parent-link plink))
(min -90)
(max 90)
((:max-joint-velocity mjv))
((:max-joint-torque mjt))
((:joint-min-max-table mm-table))
((:joint-min-max-target mm-target))
&allow-other-keys

abstract class of joint, users need to use rotational-joint, linear-joint, sphere-joint, 6dof-joint, wheel-joint or omniwheel-joint.
use :parent-link/:child-link for specifying links that this joint connect to and :min/:min for range of joint angle in degree.

:min-angle *&optional v* [メソッド]

If v is set, it updates min-angle of this instance. :min-angle returns minimal angle of this joint in degree.

:max-angle *&optional v* [メソッド]

If v is set, it updates max-angle of this instance. :max-angle returns maximum angle of this joint in degree.

:parent-link *&rest args* [メソッド]

Returns parent link of this joint. if any arguments is set, it is passed to the parent-link.

:child-link *&rest args* [メソッド]

Returns child link of this joint. if any arguments is set, it is passed to the child-link.

:calc-dav-gain *dav i periodic-time* [メソッド]

:joint-dof [メソッド]

:speed-to-angle *&rest args* [メソッド]

:angle-to-speed *&rest args* [メソッド]

:calc-jacobian *&rest args* [メソッド]

:joint-velocity *&optional jv* [メソッド]

:joint-acceleration *&optional ja* [メソッド]

:joint-torque *&optional jt* [メソッド]

:max-joint-velocity *&optional mjv* [メソッド]

:max-joint-torque *&optional mjt* [メソッド]

:joint-min-max-table *&optional mm-table* [メソッド]

:joint-min-max-target *&optional mm-target* [メソッド]

:joint-min-max-table-angle-interpolate *target-angle min-or-max* [メソッド]

:joint-min-max-table-min-angle *&optional (target-angle (send joint-min-max-target :joint-angle))* [メソッド]

:joint-min-max-table-max-angle *&optional (target-angle (send joint-min-max-target :joint-angle))* [メソッド]

rotational-joint [クラス]

:super **joint**
:slots axis

:init *&rest args &key ((:axis ax) :z)* [method]
((:max-joint-velocity mjv) 5)
((:max-joint-torque mjt) 100)
&allow-other-keys

create instance of rotational-joint. :axis is either (:x, :y, :z) or vector. :min-angle and :max-angle takes in radius, but velocity and torque are given in SI units.

:joint-angle *&optional v &key relative* [method]
&allow-other-keys

Return joint-angle if v is not set, if v is given, set joint angle. v is rotational value in degree.

:joint-dof [メソッド]

Returns DOF of rotational joint, 1.

:calc-angle-speed-gain *dav i periodic-time* [メソッド]

:speed-to-angle *v* [メソッド]

:angle-to-speed *v* [メソッド]

:calc-jacobian *&rest args* [メソッド]

linear-joint [クラス]

:super **joint**
:slots axis

:init *&rest args &key ((:axis ax) :z)* [method]
((:max-joint-velocity mjv) (/ pi 4))
((:max-joint-torque mjt) 100)
&allow-other-keys

Create instance of linear-joint. :axis is either (:x, :y, :z) or vector. :min-angle and :max-angle takes in [mm], but velocity and torque are given in SI units.

:joint-angle *&optional v &key relative* [method]
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is linear value in [mm].

:joint-dof [メソッド]

Returns DOF of linear joint, 1.

:calc-angle-speed-gain *dav i periodic-time* [メソッド]

:speed-to-angle *v* [メソッド]

:angle-to-speed *v* [メソッド]

:calc-jacobian *&rest args* [メソッド]

wheel-joint [クラス]

:super **joint**
:slots axis

:init *&rest args &key (min (float-vector ∗-inf∗∗-inf∗))* [method]
(max (float-vector ∗inf∗∗inf∗))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ pi 4)))

((:max-joint-torque mjt) (float-vector 100 100))
&allow-other-keys

Create instance of wheel-joint.

:joint-angle *&optional v &key relative* [method]
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is (float-vector translation-x[mm] rotation-z[deg])

:joint-dof [メソッド]

Returns DOF of linear joint, 2.

:calc-angle-speed-gain *dav i periodic-time* [メソッド]

:speed-to-angle *dv* [メソッド]

:angle-to-speed *dv* [メソッド]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [メソッド]

omniwheel-joint [クラス]

:super **joint**
:slots axis

:init *&rest args &key (min (float-vector ∗-inf∗∗-inf∗∗-inf∗))* [method]
(max (float-vector ∗inf∗∗inf∗∗inf∗))
((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ 0.08 0.05) (/ pi 4)))
((:max-joint-torque mjt) (float-vector 100 100 100))
&allow-other-keys

create instance of omniwheel-joint.

:joint-angle *&optional v &key relative* [method]
&allow-other-keys

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is (float-vector translation-x[mm] translation-y[mm] rotation-z[deg])

:joint-dof [メソッド]

Returns DOF of linear joint, 3.

:calc-angle-speed-gain *dav i periodic-time* [メソッド]

:speed-to-angle *dv* [メソッド]

:angle-to-speed *dv* [メソッド]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [メソッド]

sphere-joint [クラス]

:super **joint**
:slots axis

:init *&rest args &key (min (float-vector *-inf**-inf**-inf*)) (max (float-vector *inf**inf**inf*)) ((:max-joint-velocity mjv) (float-vector (/ pi 4) (/ pi 4) (/ pi 4))) ((:max-joint-torque mjt) (float-vector 100 100 100)) &allow-other-keys* [method]

Create instance of sphere-joint. min/max are defiend as a region of angular velocity in degree.

:joint-angle *&optional v &key relative &allow-other-keys* [method]

return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector by axis-angle representation, i.e (scale rotation-angle-from-default-coords[deg] axis-unit-vector)

:joint-angle-rpy *&optional v &key relative* [メソッド]

Return joint-angle if v is not set, if v is given, set joint-angle vector by RPY representation, i.e. (float-vector yaw[deg] roll[deg] pitch[deg])

:joint-dof [メソッド]

Returns DOF of linear joint, 3.

:joint-euler-angle *&key (axis-order '(:z :y :x)) ((:child-rot m) (send child-link :rot))* [method]

Return joint-angle if v is not set, if v is given, set joint-angle vector by euler representation.

:calc-angle-speed-gain *dav i periodic-time* [メソッド]

:speed-to-angle *dv* [メソッド]

:angle-to-speed *dv* [メソッド]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [メソッド]

6dof-joint [クラス]

:super **joint**
:slots axis

:init *&rest args &key (min (float-vector *-inf**-inf**-inf**-inf**-inf**-inf*)) (max (float-vector *inf**inf**inf**inf**inf**inf*)) ((:max-joint-velocity mjv) (float-vector (/ 0.08 0.05) (/ 0.08 0.05) (/ 0.08 0.05) (/ pi 4) (/ p ((:max-joint-mjt mjt) (float-vector 100 100 100 100 100 100)) (absolute-p nil) &allow-other-keys* [method]

Create instance of 6dof-joint.

:joint-angle *&optional v &key relative* [method]
&allow-other-keys

Return joint-angle if v is not set, if v is given, set joint angle vector, which is 6D vector of 3D translation[mm] and 3D rotation[deg], i.e. (find-if #'(lambda (x) (eq (send (car x) :name) 'sphere-joint)) (documentation :joint-angle))

:joint-angle-rpy *&optional v &key relative* [メソッド]

Return joint-angle if v is not set, if v is given, set joint angle. v is joint-angle vector, which is 6D vector of 3D translation[mm] and 3D rotation[deg], for rotation, please see (find-if #'(lambda (x) (eq (send (car x) :name) 'sphere-joint)) (documentation :joint-angle-rpy))

:joint-dof [メソッド]

Returns DOF of linear joint, 6.

:calc-angle-speed-gain *dav i periodic-time* [メソッド]

:speed-to-angle *dv* [メソッド]

:angle-to-speed *dv* [メソッド]

:calc-jacobian *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [メソッド]

bodyset-link [クラス]

:super **bodyset**
:slots joint parent-link child-links analysis-level default-coords weight acentroid inertia-tenso

:init *coords &rest args &key ((:analysis-level level) :body)* [method]
((:weight w) 1)
((:centroid c) #f(0.0 0.0 0.0))
((:inertia-tensor i) (unit-matrix 3))
&allow-other-keys

Create instance of bodyset-link.

:worldcoords *&optional (level analysis-level)* [メソッド]

Returns a coordinates object which represents this coord in the world by concatenating all the cascoords from the root to this coords.

:analysis-level *&optional v* [メソッド]

Change analysis level :coords only changes kinematics level and :body changes geometry too.

:weight *&optional w* [メソッド]

Returns a weight of the link. If w is given, set weight.

:centroid *&optional c* [メソッド]

Returns a centroid of the link. If c is given, set new centroid.

:inertia-tensor *&optional i* [メソッド]

Returns a inertia tensor of the link. If c is given, set new intertia tensor.

:joint *&rest args* [メソッド]
Returns a joint associated with this link. If args is given, args are forward to the joint.

:add-joint *j* [メソッド]
Set j as joint of this link

:del-joint [メソッド]
Remove current joint of this link

:parent-link [メソッド]
Returns parent link

:child-links [メソッド]
Returns child links

:add-child-links *l* [メソッド]
Add l to child links

:add-parent-link *l* [メソッド]
Set l as parent link

:del-child-link *l* [メソッド]
Delete l from child links

:del-parent-link [メソッド]
Delete parent link

:default-coords *&optional c* [メソッド]

cascaded-link [クラス]

:super **cascaded-coords**
:slots links joint-list bodies collision-avoidance-links end-coords-list

:init *&rest args &key name &allow-other-keys* [method]

Create cascaded-link.

:init-ending [メソッド]
This method is to called finalize the instantiation of the cascaded-link. This update bodies and child-link and parent link from joint-list

:links *&rest args* [メソッド]
Returns links, or args is passed to links

:joint-list *&rest args* [メソッド]
Returns joint list, or args is passed to joints

:link *name* [メソッド]
Return a link with given name.

:joint *name* [メソッド]

Return a joint with given name.

:end-coords *name* [メソッド]

Returns end-coords with given name

:bodies *&rest args* [メソッド]

Return bodies of this object. If args is given it passed to all bodies

:faces [メソッド]

Return faces of this object.

:angle-vector *&optional vec (angle-vector (instantiate float-vector (calc-target-joint-dimension joint-list)))* [メソッド]

Returns angle-vector of this object, if vec is given, it updates angles of all joint. If given angle-vector violate min/max range, the value is modified.

:link-list *to &optional from* [メソッド]

Find link list from to link to from link.

:plot-joint-min-max-table *joint0 joint1* [メソッド]

Plot joint min max table on Euslisp window.

:calc-jacobian-from-link-list *link-list &rest args &key move-target* [method]
(transform-coords move-target)
(rotation-axis (cond ((atom move-target) nil) (t (make-list
(translation-axis (cond ((atom move-target) t) (t (make-lis
(col-offset 0)
(dim (send self :calc-target-axis-dimension rotation-axis tra
(fik-len (send self :calc-target-joint-dimension link-list))
fik
(tmp-v0 (instantiate float-vector 0))
(tmp-v1 (instantiate float-vector 1))
(tmp-v2 (instantiate float-vector 2))
(tmp-v3 (instantiate float-vector 3))
(tmp-v3a (instantiate float-vector 3))
(tmp-v3b (instantiate float-vector 3))
(tmp-m33 (make-matrix 3 3))
&allow-other-keys

Calculate jacobian matrix from link-list and move-target. Unit system is [m] or [rad], not [mm] or [deg].

:inverse-kinematics-loop *dif-pos dif-rot &rest args &key (stop 1)* [method]
(loop 0)
link-list
move-target
(rotation-axis (cond ((atom move-target) t) (t (make-list
(translation-axis (cond ((atom move-target) t) (t (make-
(thre (cond ((atom move-target) 1) (t (make-list (length
(rthre (cond ((atom move-target) (deg2rad 1)) (t (make-

(dif-pos-ratio 1.0)
(dif-rot-ratio 1.0)
union-link-list
target-coords
(jacobi)
(additional-check)
(additional-jacobi)
(additional-vel)
(centroid-thre 1.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(cog-gain 1.0)
(min-loop (/ stop 10))
debug-view
ik-args
&allow-other-keys

:inverse-kinematics-loop is one loop calculation for :inverse-kinematics.
In this method, joint position difference satisfying workspace difference (dif-pos, dif-rot) are calculated and euslisp model joint angles are updated.
Optional arguments:
:additional-check
This argument is to add optional best-effort convergence conditions.
:additional-check should be function or lambda.
best-effort =>In :inverse-kinematics-loop, 'success' is overwritten by '(and success additional-check)'
In :inverse-kinematics, 'success is not overwritten.
So, :inverse-kinematics-loop wait until ':additional-check' becomes 't' as possible,
but ':additional-check' is neglected in the final :inverse-kinematics return.
:min-loop
Minimam loop count (nil by default).
If integer is specified, :inverse-kinematics-loop does returns :ik-continues and continueing solving IK.
If min-loop is nil, do not consider loop counting for IK convergence.

:inverse-kinematics *target-coords &rest args &key* *(stop 50)* [method]
(link-list)
(move-target)
(debug-view)
(warnp t)
(revert-if-fail t)
(rotation-axis (cond ((atom move-target) t) (t (make-list (lengt
(translation-axis (cond ((atom move-target) t) (t (make-list (le
(joint-args)
(thre (cond ((atom move-target) 1) (t (make-list (length move-t
(rthre (cond ((atom move-target) (deg2rad 1)) (t (make-list (le
(union-link-list)

(centroid-thre 1.0)
(target-centroid-pos)
(centroid-offset-func)
(cog-translation-axis :z)
(cog-null-space nil)
(dump-command :fail-only)
(periodic-time 0.5)
(check-collision)
(additional-jacobi)
(additional-vel)
&allow-other-keys

Move move-target to target-coords.
dump-command should be t, nil, or :fail-only.
t : dump log both in success and fail.
:fail-only : dump log only in fail.
nil : do not dump log.

:inverse-kinematics-for-closed-loop-forward-kinematics *target-coords &rest args &key (move-target)* [method]
(link-list)
(move-joints-hook)
(additional-weight-list)
(constrained-joint-list)
(constrained-joint-angl
&allow-other-keys

Solve inverse-kinematics for closed loop forward kinematics.
Move move-target to target-coords with link-list.
link-list loop should be close when move-target reachs target-coords.
constrained-joint-list is list of joints specified given joint angles in closed loop.
constrained-joint-angle-list is list of joint-angle for constrained-joint-list.

:calc-jacobian-for-interlocking-joints *link-list &key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [メソッド]

Calculate jacobian to keep interlocking joint velocity same.
dtheta_0 = dtheta_1 =>[... 0 1 0 ... 0 -1 0 ....][...dtheta_0...dtheta_1...]

:calc-vel-for-interlocking-joints *link-list &key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [メソッド]

Calculate 0 velocity for keeping interlocking joint at the same joint angle.

:set-midpoint-for-interlocking-joints *&key (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [メソッド]

Set interlocking joints at mid point of each joint angle.

:interlocking-joint-pairs [メソッド]

Interlocking joint pairs.
pairs are (list (cons joint0 joint1) ...)
If users want to use interlocking joints, please overwrite this method.

:check-interlocking-joint-angle-validity *&key (angle-thre 0.001) (interlocking-joint-pairs (send self :interlocking-joint-pairs))* [method]

Check if all interlocking joint pairs are same values.

:update-descendants *&rest args* [メソッド]

:find-link-route *to &optional from* [メソッド]

:make-joint-min-max-table *l0 l1 joint0 joint1 &key (fat 0) (fat2 nil) (debug nil) (margin 0.0) (overwrite-collision-model nil)* [メソッド]

:make-min-max-table-using-collision-check *l0 l1 joint0 joint1 joint-range0 joint-range1 min-joint0 min-joint1 fat fat2 debug margin* [メソッド]

:plot-joint-min-max-table-common *joint0 joint1* [メソッド]

:calc-target-axis-dimension *rotation-axis translation-axis* [メソッド]

:calc-union-link-list *link-list* [メソッド]

:calc-target-joint-dimension *link-list* [メソッド]

:calc-inverse-jacobian *jacobi &rest args &key ((:manipulability-limit ml) 0.1) ((:manipulability-gain mg) 0.001) weight debug-view ret wmat tmat umat umat2 mat-tmp mat-tmp-rc tmp-mrr tmp-mrr2 &allow-other-keys* [メソッド]

:calc-gradh-from-link-list *link-list &optional (res (instantiate float-vector (length link-list)))* [メソッド]

:calc-joint-angle-speed *union-vel &rest args &key angle-speed (angle-speed-blending 0.5) jacobi jacobi# null-space i-j#j debug-view weight wmat tmp-len tmp-len2 fik-len &allow-other-keys* [メソッド]

:calc-joint-angle-speed-gain *union-link-list dav periodic-time* [メソッド]

:collision-avoidance-links *&optional l* [メソッド]

:collision-avoidance-link-pair-from-link-list *link-lists &key obstacles ((:collision-avoidance-links collision-links) collision-avoidance-links) debug* [メソッド]

:collision-avoidance-calc-distance *&rest args &key union-link-list (warnp t) ((:collision-avoidance-link-pair pair-list)) &allow-other-keys* [メソッド]

:collision-avoidance-args *pair link-list* [メソッド]

:collision-avoidance *&rest args &key avoid-collision-distance avoid-collision-joint-gain avoid-collision-null-gain ((:collision-avoidance-link-pair pair-list)) (union-link-list) (link-list) (weight) (fik-len (send self :calc-target-joint-dimension union-link-list)) debug-view &allow-other-keys* [メソッド]

:move-joints *union-vel &rest args &key union-link-list (periodic-time 0.05) (joint-args) (debug-view nil) (move-joints-hook) &allow-other-keys* [メソッド]

:find-joint-angle-limit-weight-old-from-union-link-list *union-link-list* [メソッド]

:reset-joint-angle-limit-weight-old *union-link-list* [メソッド]

:calc-weight-from-joint-limit *avoid-weight-gain fik-len link-list union-link-list debug-view weight tmp-weight tmp-len* [メソッド]

:calc-inverse-kinematics-weight-from-link-list *link-list &key (avoid-weight-gain 1.0) (union-link-list (send self :calc-union-link-list link-list)) (fik-len (send self :calc-target-joint-dimension union-link-list)) (weight (fill (instantiate float-vector fik-len) 1)) (additional-weight-list) (debug-view) (tmp-weight (instantiate float-vector fik-len)) (tmp-len (instantiate float-vector fik-len))* [メソッド]

:calc-nspace-from-joint-limit *avoid-nspace-gain union-link-list weight debug-view tmp-nspace* [メソッド]

:calc-inverse-kinematics-nspace-from-link-list *link-list &key (avoid-nspace-gain 0.01) (union-link-list (send self :calc-union-link-list link-list)) (fik-len (send self :calc-target-joint-dimension union-link-list)) (null-space) (debug-view) (additional-nspace-list) (cog-gain 0.0) (target-centroid-pos) (centroid-offset-func) (cog-translation-axis :z) (cog-null-space nil) (weight (fill (instantiate float-vector fik-len) 1.0)) (update-mass-properties t) (tmp-nspace (instantiate float-vector fik-len))* [メソッド]

:move-joints-avoidance *union-vel &rest args &key union-link-list link-list (fik-len (send self :calc-target-joint-dimension union-link-list)) (weight (fill (instantiate float-vector fik-len) 1)) (null-space) (avoid-nspace-gain 0.01) (avoid-weight-gain 1.0) (avoid-collision-distance 200) (avoid-collision-null-gain 1.0) (avoid-collision-joint-gain 1.0) ((:collision-avoidance-link-pair pair-list) (send self :collision-avoidance-link-pair-from-link-list link-list :obstacles (cadr (memq :obstacles args)) :debug (cadr (memq :debug-view args)))) (cog-gain 0.0) (target-centroid-pos) (centroid-offset-func) (cog-translation-axis :z) (cog-null-space nil) (additional-weight-list) (additional-nspace-list) (tmp-len (instantiate float-vector fik-len)) (tmp-len2 (instantiate float-vector fik-len)) (tmp-weight (instantiate float-vector fik-len)) (tmp-nspace (instantiate float-vector fik-len)) (tmp-mcc (make-matrix fik-len fik-len)) (tmp-mcc2 (make-matrix fik-len fik-len)) (debug-view) (jacobi) &allow-other-keys* [メソッド]

:inverse-kinematics-args *&rest args &key union-link-list rotation-axis translation-axis additional-jacobi-dimension &allow-other-keys* [メソッド]

:draw-collision-debug-view [メソッド]

:ik-convergence-check *success dif-pos dif-rot rotation-axis translation-axis thre rthre centroid-thre target-centroid-pos centroid-offset-func cog-translation-axis &key (update-mass-properties t)* [メソッド]

:calc-vel-from-pos *dif-pos translation-axis &rest args &key (p-limit 100.0) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-v3 (instantiate float-vector 3)) &allow-other-keys* [メソッド]

:calc-vel-from-rot *dif-rot rotation-axis &rest args &key (r-limit 0.5) (tmp-v0 (instantiate float-vector 0)) (tmp-v1 (instantiate float-vector 1)) (tmp-v2 (instantiate float-vector 2)) (tmp-v3 (instantiate float-vector 3)) &allow-other-keys* [メソッド]

:collision-check-pairs *&key ((:links ls) (cons (car links) (all-child-links (car links))))* [メソッド]

:self-collision-check *&key (mode :all) (pairs (send self :collision-check-pairs)) (collision-func 'pqp-collision-check)* [メソッド]

:calc-grasp-matrix *contact-points &optional (ret (make-matrix 6 (*6 (length contact-points))))* [メソッド]

eusmodel-validity-check *robot* [関数]

Check if the robot model is validate

calc-jacobian-default-rotate-vector *paxis world-default-coords child-reverse transform-coords tmp-v3 tmp-m33* [関数]

calc-jacobian-rotational *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [関数]

calc-jacobian-linear *fik row column joint paxis child-link world-default-coords child-reverse move-target transform-coords rotation-axis translation-axis tmp-v0 tmp-v1 tmp-v2 tmp-v3 tmp-v3a tmp-v3b tmp-m33* [関数]

calc-angle-speed-gain-scalar *j dav i periodic-time* [関数]

calc-angle-speed-gain-vector *j dav i periodic-time* [関数]

all-child-links *s &optional (pred #'identity)* [関数]

calc-dif-with-axis *dif axis &optional tmp-v0 tmp-v1 tmp-v2* [関数]

calc-target-joint-dimension *joint-list* [関数]

calc-joint-angle-min-max-for-limit-calculation *j kk jamm* [関数]

joint-angle-limit-weight *j-l &optional (res (instantiate float-vector (calc-target-joint-dimension j-l)))* [関数]

joint-angle-limit-nspace *j-l &optional (res (instantiate float-vector (calc-target-joint-dimension j-l)))* [関数]

calc-jacobian-from-link-list-including-robot-and-obj-virtual-joint *link-list move-target obj-move-target robot &key (rotation-axis '(t t)) (translation-axis '(t t)) (fik (make-matrix (send robot :calc-target-axis-dimension rotation-axis translation-axis) (send robot :calc-target-joint-dimension link-list)))* [関数]

append-obj-virtual-joint *link-list target-coords &key (joint-class 6dof-joint) (joint-args) (vplink) (vplink-coords) (vclink-coords)* [関数]

append-multiple-obj-virtual-joint *link-list target-coords &key (joint-class '(6dof-joint)) (joint-args '(nil)) (vplink) (vplink-coords) (vclink-coords)* [関数]

eusmodel-validity-check-one *robot* [関数]

bodyset [クラス]

:super **cascaded-coords**

:slots (geometry::bodies :type cons)

:init *coords &rest args &key (name (intern (format nil bodyset A (system:address self)) KEYWORD)) ((:bodies geometry::bs)) &allow-other-keys* [method]

Create bodyset object

:bodies *&rest args* [メソッド]

:faces [メソッド]

:worldcoords [メソッド]

:draw-on *&rest args* [メソッド]

midcoords *geometry::p geometry::c1 geometry::c2* [関数]

Returns mid (or p) coordinates of given two cooridnates c1 and c2

orient-coords-to-axis *geometry::target-coords geometry::v &optional (geometry::axis :z)* [関数]

orient 'axis' in 'target-coords' to the direction specified by 'v' destructively.
'v' must be non-zero vector.

geometry::face-to-triangle-aux *geometry::f* [関数]

triangulate the face.

geometry::face-to-triangle *geometry::f* [関数]

convert face to set of triangles.

geometry::face-to-tessel-triangle *geometry::f geometry::num* [関数]

return polygon if triangable, return nil if it is not.

body-to-faces *geometry::abody* [関数]

return triangled faces of given body

make-sphere *geometry::r &rest args* [関数]
make sphere of given r

make-ring *geometry::ring-radius geometry::pipe-radius &rest args &key (geometry::segments 16)* [関数]
make ring of given ring and pipe radius

x-of-cube *geometry::cub* [関数]
return x of cube.

y-of-cube *geometry::cub* [関数]
return y of cube.

z-of-cube *geometry::cub* [関数]
return z of cube.

height-of-cylinder *geometry::cyl* [関数]
return height of cylinder.

radius-of-cylinder *geometry::cyl* [関数]
return radius of cylinder.

radius-of-sphere *geometry::sp* [関数]
return radius of shape.

geometry::make-faceset-from-vertices *geometry::vs* [関数]
create faceset from vertices.

matrix-to-euler-angle *geometry::m geometry::axis-order* [関数]
return euler angle from matrix.

geometry::quaternion-from-two-vectors *geometry::a geometry::b* [関数]
Comupute quaternion which rotate vector a into b.

transform-coords *geometry::c1 geometry::c2 &optional (geometry::c3 (let ((geometry::dim (send geometry::c1 :dimension))) (instance coordinates :newcoords (unit-matrix geometry::dim) (instantiate float-vector geometry::dim))))* [関数]
geometry::face-to-triangle-rest-polygon *geometry::f geometry::num geometry::edgs* [関数]
geometry::face-to-triangle-make-simple *geometry::f* [関数]
body-to-triangles *geometry::abody &optional (geometry::limit 50)* [関数]
geometry::triangle-to-triangle *geometry::aface &optional (geometry::limit 50)* [関数]

robot-model [クラス]

:super **cascaded-link**
:slots larm-end-coords rarm-end-coords lleg-end-coords rleg-end-coords head-end-coords tor

:camera *sensor-name* [メソッド]
Returns camera with given name

:force-sensor *sensor-name* [メソッド]
Returns force sensor with given name

:imu-sensor *sensor-name* [メソッド]
Returns imu sensor of given name

:force-sensors [メソッド]

Returns force sensors.

:imu-sensors [メソッド]

Returns imu sensors.

:cameras [メソッド]

Returns camera sensors.

:look-at-hand *l/r* [メソッド]

look at hand position, l/r supports :rarm, :larm, :arms, and '(:rarm :larm)

:inverse-kinematics *target-coords &rest args &key look-at-target* [method]
(move-target)
(link-list (if (atom move-target) (send self :link-list (send move-
&allow-other-keys

solve inverse kinematics, move move-target to target-coords
look-at-target suppots t, nil, float-vector, coords, list of float-vector, list of coords
link-list is set by default based on move-target ->root link link-list

:inverse-kinematics-loop *dif-pos dif-rot &rest args &key target-coords* [method]
debug-view
look-at-target
(move-target)
(link-list (if (atom move-target) (send self :link-list (send
&allow-other-keys

move move-target using dif-pos and dif-rot,
look-at-target suppots t, nil, float-vector, coords, list of float-vector, list of coords
link-list is set by default based on move-target ->root link link-list

:look-at-target *look-at-target &key (target-coords)* [メソッド]

move robot head to look at targets, look-at-target support t/nil float-vector coordinates, center of list of float-vector or list of coordinates

:init-pose [メソッド]

Set robot to initial posture.

:torque-vector *&key (force-list)* [method]
(moment-list)
(target-coords)
(debug-view nil)
(calc-statics-p t)
(dt 0.005)
(av (send self :angle-vector))
(root-coords (send (car (send self :links)) :copy-worldcoords))
(calc-torque-buffer-args (send self :calc-torque-buffer-args))

Returns torque vector

:calc-force-from-joint-torque *limb all-torque &key (move-target (send self limb :end-coords))* [method]
(use-torso)

Calculates end-effector force and moment from joint torques.

:fullbody-inverse-kinematics *target-coords &rest args &key (move-target) (link-list) (min (float-vector -500 -500 -500 -20 -20 -10)) (max (float-vector 500 500 25 20 20 10)) (root-link-virtual-joint-weight #f(0.1 0.1 0.1 0.1 0.5 (target-centroid-pos (apply #'midpoint 0.5 (send self (cog-gain 1.0) (cog-translation-axis :z) (centroid-offset-func nil) (centroid-thre 5.0) (additional-weight-list) (joint-args nil) (cog-null-space t) &allow-other-keys* [method]

fullbody inverse kinematics for legged robot.
necessary args : target-coords, move-target, and link-list must include legs' (or leg's) parameters
ex. (send *robot*:fullbody-inverse-kinematics (list rarm-tc rleg-tc lleg-tc) :move-target (list rarm-mt rleg-mt lleg-mt) :link-list (list rarm-ll rleg-ll lleg-ll))

:print-vector-for-robot-limb *vec* [メソッド]

Print angle vector with limb alingment and limb indent.
For example, if robot is rarm, larm, and torso, print result is:
#f(
rarm-j0 ... rarm-jN
larm-j0 ... larm-jN
torso-j0 ... torso-jN
)

:calc-zmp-from-forces-moments *forces moments &key (wrt :world) (limbs '(:rleg :lleg)) (force-sensors (mapcar #'(lambda (l) (send self :force-sen (cop-coords (mapcar #'(lambda (l) (send self l :end-coords (fz-thre 0.001) (limb-cop-fz-list (mapcar #'(lambda (fs f m cc) (let ((fsp* [method]

Calculate zmp[mm] from sensor local forces and moments
If force_z is large, zmp can be defined and returns 3D zmp.
Otherwise, zmp cannot be defined and returns nil.

:foot-midcoords *&optional (mid 0.5)* [メソッド]

Calculate midcoords of :rleg and :lleg end-coords.
In the following codes, leged robot is assumed.

:fix-leg-to-coords *fix-coords &optional (l/r :both) &key (mid 0.5) &allow-other-keys* [method]

Fix robot's legs to a coords
In the following codes, leged robot is assumed.

:move-centroid-on-foot *leg fix-limbs &rest args &key (thre (mapcar #'(lambda (x) (if (memq x '(:rleg :lleg)) 1 5)) (rthre (mapcar #'(lambda (x) (deg2rad (if (memq x '(:rleg :(mid 0.5) (target-centroid-pos (if (eq leg :both) (apply #'midpoint mid (fix-limbs-target-coords (mapcar #'(lambda (x) (send self x :(root-link-virtual-joint-weight #f(0.1 0.1 0.0 0.0 0.0 0.5)) &allow-other-keys*

Move robot COG to change centroid-on-foot location,
leg : legs for target of robot's centroid, which should be :both, :rleg, and :lleg.
fix-limbs : limb names which are fixed in this IK.

:calc-walk-pattern-from-footstep-list *footstep-list &key (default-step-height 50) (dt 0.1) (default-step-time 1.0) (solve-angle-vector-args) (debug-view nil) ((:all-limbs al) '(:rleg :lleg)) ((:default-zmp-offsets dzo) (mapcan #'(lambda (x) (list (init-pose-function #'(lambda nil (send self :move-centr (start-with-double-support t) (end-with-double-support t) (ik-thre 1) (ik-rthre (deg2rad 1)) (calc-zmp t)* [method]

Calculate walking pattern from foot step list and return pattern list as a list of angle-vector, root-coords, time, and so on.

:gen-footstep-parameter *&key (ratio 1.0)* [メソッド]

Generate footstep parameter

:go-pos-params->footstep-list *xx yy th &key ((:footstep-parameter prm) (send self :footstep-parameter))* [metho *((:default-half-offset defp) (cadr (memq :default-half-offset prm))) ((:forward-offset-length xx-max) (cadr (memq :forward-offset-length ((:outside-offset-length yy-max) (cadr (memq :outside-offset-length ((:rotate-rad th-max) (abs (rad2deg (cadr (memq :rotate-rad prm)) (gen-go-pos-step-node-func #'(lambda (mc leg leg-translate-pos) (le*

Calculate foot step list from goal x position [mm], goal y position [mm], and goal yaw orientation [deg].

:support-polygons [メソッド]

Return support polygons.

:support-polygon *name* [メソッド]

Return support polygon with given name.

:make-default-linear-link-joint-between-attach-coords *attach-coords-0 attach-coords-1 end-coords-name linear-joint-name* [メソッド]

Make default linear arctuator module such as muscle and cylinder and append lins and joint-list.
Module includes parent-link =>(j0) =>l0 =>(j1) =>l1 (linear actuator) =>(j2) =>l2 =>end-coords.
attach-coords-0 is root side coords which linear actulator is attached to.

attach-coords-1 is end side coords which linear actulator is attached to.
end-coords-name is the name of end-coords.
linear-joint-name is the name of linear actuator.

:init-ending [メソッド]

:limb *limb method &rest args* [メソッド]

:inverse-kinematics-loop-for-look-at *limb &rest args* [メソッド]

:gripper *limb &rest args* [メソッド]

:get-sensor-method *sensor-type sensor-name* [メソッド]

:get-sensors-method-by-limb *sensors-type limb* [メソッド]

:larm *&rest args* [メソッド]

:rarm *&rest args* [メソッド]

:lleg *&rest args* [メソッド]

:rleg *&rest args* [メソッド]

:head *&rest args* [メソッド]

:torso *&rest args* [メソッド]

:arms *&rest args* [メソッド]

:legs *&rest args* [メソッド]

:joint-angle-limit-nspace-for-6dof *&key (avoid-nspace-gain 0.01) (limbs '(:rleg :lleg))* [メソッド]

:joint-order *limb &optional jname-list* [メソッド]

:draw-gg-debug-view *end-coords-list contact-state rz cog pz czmp dt* [メソッド]

:footstep-parameter [メソッド]

:make-support-polygons [メソッド]

:make-sole-polygon *name* [メソッド]
make-default-robot-link *len radius axis name &optional extbody* [関数]

18.5 センサモデル

sensor-model [クラス]

:super **body**

:slots data profile

:profile *&optional p* [メソッド]

:signal *rawinfo* [メソッド]

:simulate *model* [メソッド]

:read [メソッド]

:draw-sensor *v* [メソッド]

:init *shape &key name &allow-other-keys* [メソッド]

bumper-model [クラス]

:super **sensor-model**
:slots bumper-threshold

:init *b &rest args &key ((:bumper-threshold bt) 20) name* [method]

Create bumper model, b is the shape of an object and bt is the threshold in distance[mm].

:simulate *objs* [メソッド]

Simulate bumper, with given objects, return 1 if the sensor detects an object and 0 if not.

:draw *vwer* [メソッド]

:draw-sensor *vwer* [メソッド]

camera-model [クラス]

:super **sensor-model**
:slots (vwing :forward (:projection :newprojection :view :viewpoint :view-direction :viewdist

:init *b &rest args &key ((:width pw) 320) ((:height ph) 240) (view-up #f(0.0 1.0 0.0)) (viewdistance 5.0) (hither 100.0) (yon 10000.0) &allow-other-keys* [method]

Create camera model. b is the shape of an object

:width [メソッド]

Returns width of the camera in pixel.

:height [メソッド]

Returns height of the camera in pixel.

:fovy [メソッド]

Returns field of view in degree

:cx [メソッド]

Returns center x.

:cy [メソッド]

Returns center y.

:fx [メソッド]

Returns focal length of x.

:fy [メソッド]

Returns focal length of y.

:screen-point *pos* [メソッド]

Returns point in screen corresponds to the given pos.

:3d-point *x y d* [メソッド]

Returns 3d position

:ray *x y* [メソッド]

Returns ray vector of given x and y.

:viewing *&rest args* [メソッド]

:draw-on *&rest args &key ((:viewer vwer) ∗viewer∗) &allow-other-keys* [メソッド]

:draw-sensor *vwer &key flush (width 1) (color (float-vector 1 1 1))* [メソッド]

:draw-objects *vwr objs* [メソッド]

:get-image *vwr &key (points) (colors)* [メソッド]

make-camera-from-param *&key pwidth pheight fx fy cx cy (tx 0) (ty 0) parent-coords name* [function]

Create camera object from given parameters.

18.6 環境モデル

scene-model [クラス]

:super **cascaded-coords**
:slots name objs

:init *&rest args &key ((:name n) scene) ((:objects o))* [method]

Create scene model

:objects [メソッド]

Returns objects in the scene.

:find-object *name* [メソッド]

Returns objects with given name.

:spots *&optional name* [メソッド]

Returns spots in the scene. If name is given returns spot of given name.

:object *name* [メソッド]

Returns object of given name.

:spot *name* [メソッド]

Returns scene of given name.

:bodies [メソッド]

18.7 動力学計算・歩行動作生成

riccati-equation [クラス]

:super **propertied-object**
:slots a b c p q r k a-bkt r+btpb-1

:init *aa bb cc qq rr* [メソッド]

:solve [メソッド]

preview-controller [クラス]

:super **riccati-equation**
:slots xk uk delay f1-n y1-n dim queue-index initialize-queue-p additional-data-queue finishe

:init *dt &key (q) (r)* [method]

((:delay d))
((:dim tmp-dim))
(init-xk (float-vector 0 0 0))
((:a _a))
((:b _b))
((:c _c))
(state-dim (array-dimension _a 0))
((:initialize-queue-p iqp))

Initialize preview-controller.
Q is weighting of output error and R is weighting of input.
dt is sampling time [s].
delay is preview time [s].
init-xk is initial state value.
A, B, C are state eq matrices.
If initialize-queue-p is t, fill all queue by the first input at the begenning, otherwise, do not fill queue at the first.

:update-xk *p &optional (add-data)* [メソッド]

Update xk by inputting reference output.
Return value : nil (initializing) =>return values (middle) =>nil (finished)
If p is nil, automatically the last value in queue is used as input and preview controller starts finishing.

:finishedp [メソッド]

Finished or not.

:last-reference-output-vector [メソッド]

Last value of reference output queue vector (y_k+N_ref).
Last value is latest future value.

:current-reference-output-vector [メソッド]

First value of reference output queue vector (y_k_ref).
First value is oldest future value and it can be used as current reference value.

:current-state-vector [メソッド]

Current state value (xk).

:current-output-vector [メソッド]

Current output value (yk).

:current-additional-data [メソッド]

Current additional data value.
First value of additional-data-queue.

:pass-preview-controller *reference-output-vector-list* [メソッド]

Get preview controller results from reference-output-vector-list and returns list.

:calc-f [メソッド]

:calc-u [メソッド]

:calc-xk [メソッド]

extended-preview-controller [クラス]

:super **preview-controller**
:slots orga orgb orgc xk∗

:init *dt &key (q) (r) ((:delay d)) ((:dim tmp-dim)) (init-xk (float-vector 0 0 0)) ((:a _orga)) ((:b _orgb)) ((:c _orgc)) (state-dim (array-dimension _orga 0)) ((:initialize-queue-p iqp))* [method]

Initialize preview-controller in extended system (error system).
Q is weighting of output error and R is weighting of input.
dt is sampling time [s].
delay is preview time [s].
init-xk is initial state value.
A, B, C are state eq matrices for original system and slot variables A,B,C are used for error system matrices.
If initialize-queue-p is t, fill all queue by the first input at the begenning, otherwise, do not fill queue at the first.

:current-output-vector [メソッド]

Current additional data value.
First value of additional-data-queue.

:calc-f [メソッド]

:calc-u [メソッド]

:calc-xk [メソッド]

preview-control-cart-table-cog-trajectory-generator [クラス]

:super **propertied-object**
:slots pc cog-z zmp-z

:init *dt _zc &key (q 1) (r 1.000000e-06) ((:delay d) 1.6) (init-xk (float-vector 0 0 0)) ((:a _a) (make-matrix 3 3 (list (list 1 dt (∗0.5 dt dt)) (list 0 1 dt) (list 0 0 1)))) ((:b _b) (make-matrix 3 1 (list (list (∗(/ 1.0 6.0) dt dt dt)) (list (∗0.5 dt dt)) (list dt)))) ((:c _c) (make-matrix 1 3 (list (list 1.0 0.0 (- (/ _zc (elt ∗g-vec∗2)))))))* [method]

((:initialize-queue-p iqp))
(preview-controller-class extended-preview-controller)

COG (xy) trajectory generator using preview-control convert reference ZMP from reference COG.
dt ->sampling time[s], _zc is height of COG [mm].
preview-controller-class is preview controller class (extended-preview-controller by default).
For other arguments, please see preview-controller and extended-preview-controller :init documentation.

:refcog [メソッド]

Reference COG [mm].

:cart-zmp [メソッド]

Cart-table system ZMP[mm] as an output variable.

:last-refzmp [メソッド]

Reference zmp at the last of queue.

:current-refzmp [メソッド]

Current reference zmp at the first of queue.

:update-xk *p &optional (add-data)* [メソッド]

Update xk and returns zmp and cog values.
For arguments, please see preview-controller and extended-preview-controller :update-xk.

:finishedp [メソッド]

Finished or not.

:current-additional-data [メソッド]

Current additional data value.

:pass-preview-controller *reference-output-vector-list* [メソッド]

Get preview controller results from reference-output-vector-list and returns list.

:cog-z [メソッド]

COG Z [mm].

:update-cog-z *zc* [メソッド]

gait-generator [クラス]

:super **propertied-object**
:slots robot dt footstep-node-list support-leg-list support-leg-coords-list swing-leg-dst-coords

:init *rb _dt* [メソッド]

:get-footstep-limbs *fs* [メソッド]

:get-counter-footstep-limbs *fs* [メソッド]

:get-limbs-zmp-list *limb-coords limb-names* [メソッド]

:get-limbs-zmp *limb-coords limb-names* [メソッド]

:get-swing-limbs *limbs* [メソッド]

:initialize-gait-parameter *fsl time cog &key ((:default-step-height dsh) 50) ((:default-double-support-ratio ddsr) 0.2) (delay 1.6) ((:all-limbs al) '(:rleg :lleg)) ((:default-zmp-offsets dzo) (mapcan #'(lambda (x) (list x (float-vector 0 0 0))) al)) (q 1.0) (r 1.000000e-06) (thre 1) (rthre (deg2rad 1)) (start-with-double-support t) ((:end-with-double-support ewds) t)* [メソッド]

:finalize-gait-parameter [メソッド]

:make-gait-parameter [メソッド]

:calc-current-swing-leg-coords *ratio src dst &key (type :shuffling) (step-height default-step-height)* [メソッド]

:calc-ratio-from-double-support-ratio [メソッド]

:calc-current-refzmp *prev cur next* [メソッド]

:calc-one-tick-gait-parameter *type* [メソッド]

:proc-one-tick *&key (type :shuffling) (solve-angle-vector :solve-av-by-move-centroid-on-foot) (solve-angle-vector-args) (debug nil)* [メソッド]

:update-current-gait-parameter [メソッド]

:solve-angle-vector *support-leg support-leg-coords swing-leg-coords cog &key (solve-angle-vector :solve-av-by-move-centroid-on-foot) (solve-angle-vector-args)* [メソッド]

:solve-av-by-move-centroid-on-foot *support-leg support-leg-coords swing-leg-coords cog robot &rest args &key (cog-gain 3.5) (stop 100) (additional-nspace-list) &allow-other-keys* [メソッド]

:cycloid-midpoint *ratio start goal height &key (top-ratio 0.5)* [メソッド]

:cycloid-midcoords *ratio start goal height &key (top-ratio 0.5)* [メソッド]

calc-inertia-matrix-rotational *mat row column paxis m-til c-til i-til axis-for-angular child-link world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [関数]

calc-inertia-matrix-linear *mat row column paxis m-til c-til i-til axis-for-angular child-link world-default-coords translation-axis rotation-axis tmp-v0 tmp-v1 tmp-v2 tmp-va tmp-vb tmp-vc tmp-vd tmp-m* [関数]

19 ロボットビューワ

x::irtviewer [クラス]

:super **x:panel**

:slots x::viewer x::objects x::draw-things x::previous-cursor-pos x::left-right-angle x::up-down

:create *&rest args &key (x::title IRT viewer) (x::view-name (gensym title)) (x::hither 200.0) (x::yon 50000.0) (x::width 500) (x::height 500) ((:draw-origin do) 150) ((:draw-floor x::df) nil) &allow-other-keys* [メソッド]

:viewer *&rest args* [メソッド]

:redraw [メソッド]

:expose *x:event* [メソッド]

:resize *x::newwidth x::newheight* [メソッド]

:configurenotify *x:event* [メソッド]

:viewtarget *&optional x::p* [メソッド]

:viewpoint *&optional x::p* [メソッド]

:look1 *&optional (x::vt x::viewtarget) (x::lra x::left-right-angle) (x::uda x::up-down-angle)* [メソッド]

:look-all *&optional x::bbox* [メソッド]

:move-viewing-around-viewtarget *x:event x::x x::y x::dx x::dy x::vwr* [メソッド]

:set-cursor-pos-event *x:event* [メソッド]

:move-coords-event *x:event* [メソッド]

:draw-event *x:event* [メソッド]

:draw-objects *&rest args* [メソッド]

:objects *&rest args* [メソッド]

:select-drawmode *x::mode* [メソッド]

:flush [メソッド]

:change-background *x::col* [メソッド]

viewer-dummy [クラス]

:super **propertied-object**
:slots nil

:nomethod *&rest args* [メソッド]

irtviewer-dummy [クラス]

:super **propertied-object**
:slots objects draw-things

:objects *&rest args* [メソッド]

:nomethod *&rest args* [メソッド]

make-irtviewer *&rest args* [関数]

Create irtviewer

:view-name title
:hither near cropping plane
:yon far cropping plane
:width width of the window
:height height of the window
:draw-origin size of origin arrow, use nil to disable it
:draw-floor use t to view floor

x::make-lr-ud-coords *x::lra x::uda* [関数]
x::draw-things *x::objs* [関数]
objects *&optional (objs t) vw* [関数]
make-irtviewer-dummy *&rest args* [関数]

20 干渉計算

干渉計算には２組の幾何モデルが交差するかを判定する物である．irteus ではノースカロライナ大学の Lin 氏らのグループにより開発された PQP を他言語インターフェースを介して利用できるようにしてある．(他の干渉計算ソフトウェアパッケージについては http://gamma.cs.unc.edu/research/collision/に詳しい．) PQP は (1) ２つのモデルが交差するかを判定する衝突検出，(2) ２つのモデル間の最初距離を算出する距離計算，(3) ２つのモデルがある距離以下であるかを判定する近接検証，等の３つ機能を提供する．

PQP ソフトウェアパッケージの使い方は irteus/PQP/README.txt に書いてあり，irteus/PQP/src/PQP.h を読むことで理解できるようになっている．

20.1 irteus から PQP の呼び出し

irteus で PQP を使うためのファイルは CPQP.C, euspqp.c, pqp.l からなる． ２つの幾何モデルが衝突してしるか否かを判定するためには，

```
(defun pqp-collision-check (model1 model2
       &optional (flag PQP_FIRST_CONTACT) &key (fat 0) (fat2 nil))
  (let ((m1 (get model1 :pqpmodel))  (m2 (get model2 :pqpmodel))
        (r1 (send model1 :worldrot)) (t1 (send model1 :worldpos))
        (r2 (send model2 :worldrot)) (t2 (send model2 :worldpos)))
    (if (null fat2) (setq fat2 fat))
    (if (null m1) (setq m1 (send model1 :make-pqpmodel :fat fat)))
    (if (null m2) (setq m2 (send model2 :make-pqpmodel :fat fat2)))
    (pqpcollide r1 t1 m1 r2 t2 m2 flag)))
```

を呼び出せば良い．r1,r1,r2,t1 はそれぞれの物体の並進ベクトル，回転行列となり，(get model1 :pqpmodel) で PQP の幾何モデルへのポインタを参照する．このポインタは:make-pqpmodel メソッドの中で以下のよう計算される．

```
(defmethod cascaded-coords
  (:make-pqpmodel
   (&key (fat 0))
   (let ((m (pqpmakemodel))
         vs v1 v2 v3 (id 0))
     (setf (get self :pqpmodel) m)
     (pqpbeginmodel m)
     (dolist (f (send self :faces))
       (dolist (poly (face-to-triangle-aux f))
         (setq vs (send poly :vertices)
               v1 (send self :inverse-transform-vector (first vs))
               v2 (send self :inverse-transform-vector (second vs))
               v3 (send self :inverse-transform-vector (third vs)))
```

```
        (when (not (= fat 0))
          (setq v1 (v+ v1 (scale fat (normalize-vector v1)))
                v2 (v+ v2 (scale fat (normalize-vector v2)))
                v3 (v+ v3 (scale fat (normalize-vector v3)))))
        (pqpaddtri m v1 v2 v3 id)
        (incf id)))
    (pqpendmodel m)
    m)))
```

ここでは，まず (pqpmakemodel) が呼び出されている．pqpmakemodel の中では，euqpqp.c で定義されている，

```
pointer PQPMAKEMODEL(register context *ctx, int n, register pointer *argv)
{
    int addr = PQP_MakeModel();
    return makeint(addr);
}
```

が呼び出されており，これは，CPQP.C の

```
PQP\_Model *PQP_MakeModel()
{
    return new PQP_Model();
}
```

が呼ばれている．PQP_Model() は PQP.h で定義されているものであり，この様にして euslisp 内の関数が実際の PQP ライブラリの関数に渡されている以降，(pqpbeginmodel m) で PQP の幾何モデルのインスタンスを作成し，(pqpaddtri m v1 v2 v3 id) として面情報を登録している．

20.2 ロボット動作と干渉計算

ハンドで物体をつかむ，という動作の静的なシミュレーションを行う場合に手（指）のリンクと対象物体の干渉を調べ，これが起こるところで物体をつかむ動作を停止させるということが出来る．

```
(objects (list *sarm* *target*))

(send *sarm* :solve-ik *target* :debug-view t)
(while (> a 0)
  (if (pqp-collision-check-objects
       (list (send *sarm* :joint-fr :child-link)
             (send *sarm* :joint-fl :child-link))
       (list *target*))
      (return))
  (decf a 0.1)
  (send *irtviewer* :draw-objects)
  (send *sarm* :move-fingers a))
(send *sarm* :end-coords :assoc *target*)

(dotimes (i 100)
  (send *sarm* :joint0 :joint-angle 1 :relative t)
  (send *irtviewer* :draw-objects))
(send *sarm* :end-coords :dissoc *target*)
(dotimes (i 100)
  (send *sarm* :joint0 :joint-angle -1 :relative t)
  (send *irtviewer* :draw-objects))
```

同様の機能が，”irteus/demo/sample-arm-model.l”ファイルの:open-hand, :close-hand というメソッドで提供されている．

pqp-collision-check-objects *geometry::obj1 geometry::obj2 &key (geometry::fat 0.2)* [関数]

return nil or t

pqp-collision-check *geometry::model1 geometry::model2 &optional (geometry::flag geometry::pqp_first_contact) &key (geometry::fat 0) (geometry::fat2 nil)* [関数]

pqp-collision-distance *geometry::model1 geometry::model2 &key (geometry::fat 0) (geometry::fat2 nil) (geometry::qsize 2)* [関数]

21 BVHデータ

bvh-link [クラス]

:super **bodyset-link**
:slots type offset channels neutral

:init *name typ offst chs parent children* [メソッド]
create link for bvh model

:type [メソッド]

:offset [メソッド]

:channels [メソッド]

bvh-sphere-joint [クラス]

:super **sphere-joint**
:slots axis-order bvh-offset-rotation

:init *&rest args &key (order (list :z :x :y)) ((:bvh-offset-rotation bvh-rotation) (unit-matrix 3)) &allow-other-keys* [method]
create joint for bvh model

:joint-angle-bvh *&optional v* [メソッド]

:joint-angle-bvh-offset *&optional v* [メソッド]

:joint-angle-bvh-impl *v bvh-offset* [メソッド]

:axis-order [メソッド]

:bvh-offset-rotation [メソッド]

bvh-6dof-joint [クラス]

:super **6dof-joint**
:slots scale axis-order bvh-offset-rotation

:init *&rest args &key (order (list :x :y :z :z :x :y)) ((:scale scl)) ((:bvh-offset-rotation bvh-rotation) (unit-matrix 3)) &allow-other-keys* [メソッド]

:joint-angle-bvh *&optional v* [メソッド]

:joint-angle-bvh-offset *&optional v* [メソッド]

:joint-angle-bvh-impl *v bvh-offset* [メソッド]

:axis-order [メソッド]

:bvh-offset-rotation [メソッド]

bvh-robot-model [クラス]

:super **robot-model**
:slots nil

:init *&rest args &key tree coords ((:scale scl))* [method]

create robot model for bvh model

:make-bvh-link *tree &key parent ((:scale scl))* [メソッド]

:angle-vector *&optional vec (angle-vector (instantiate float-vector (calc-target-joint-dimension joint-list)))* [メソッド]

:dump-joints *links &key (depth 0) (strm *standard-output*)* [メソッド]

:dump-hierarchy *&optional (strm *standard-output*)* [メソッド]

:dump-motion *&optional (strm *standard-output*)* [メソッド]

:copy-state-to *robot* [メソッド]

:fix-joint-angle *i limb joint-name joint-order a* [メソッド]

:fix-joint-order *jo limb* [メソッド]

:bvh-offset-rotate *name* [メソッド]

:init-end-coords [メソッド]

:init-root-link [メソッド]

motion-capture-data [クラス]

:super **propertied-object**
:slots frame model animation

:init *fname &key (coords (make-coords)) ((:scale scl))* [メソッド]

:model *&rest args* [メソッド]

:animation *&rest args* [メソッド]

:frame *&optional f* [メソッド]

:frame-length [メソッド]

:animate *&rest args &key (start 0) (step 1) (end (send self :frame-length)) (interval 20) &allow-other-keys* [メソッド]

rikiya-bvh-robot-model [クラス]

:super **bvh-robot-model**
:slots nil

:init *&rest args* [メソッド]

tum-bvh-robot-model [クラス]

:super **bvh-robot-model**
:slots nil

:init *&rest args* [メソッド]

cmu-bvh-robot-model [クラス]

:super **bvh-robot-model**
:slots nil

:init *&rest args* [メソッド]

read-bvh *fname &key scale* [関数]

read bvh file

bvh2eus *fname &rest args* [関数]

read bvh file and anmiate robot model in the viewer

load-mcd *fname &key (scale) (coords) (bvh-robot-model-class bvh-robot-model)* [function]

load motion capture data

parse-bvh-sexp *src &key ((:scale scl))* [関数]

make-bvh-robot-model *bvh-data &rest args* [関数]

22 Collada データ

collada::eusmodel-description *collada::model* [関数]

convert a 'model' to eusmodel-description

collada::eusmodel-link-specs *collada::links* [関数]

convert 'links' to <link-specs>

collada::eusmodel-joint-specs *collada::joints* [関数]

convert 'joints' to <joint-specs>

collada::eusmodel-description->collada *name collada::description &key (scale 0.001)* [関数]

convert eusmodel-descrption to collada sxml

collada::matrix->collada-rotate-vector *collada::mat* [関数]

convert a rotation matrix to axis-angle.

convert-irtmodel-to-collada *collada::model-file &optional (collada::output-full-dir (send (truename ./) :namestring)) (collada::model-name) (collada::exit-p t)* [関数]

convert irtmodel to collada model. (convert-irtmodel-to-collada irtmodel-file-path &optional (output-full-dir (send (truename "./") :namestring)) (model-name))

collada::symbol->string *collada::sym* [関数]
collada::->string *collada::val* [関数]
collada::string-append *&rest args* [関数]
collada::make-attr *collada::l collada::ac* [関数]
collada::make-xml *collada::x collada::bef collada::aft* [関数]
collada::sxml->xml *collada::sxml* [関数]
collada::xml-output-to-string-stream *collada::ss collada::l* [関数]
collada::cat-normal *collada::l collada::s* [関数]
collada::cat-clark *collada::l collada::s collada::i* [関数]
collada::verificate-unique-strings *names* [関数]
collada::eusmodel-link-spec *collada::link* [関数]
collada::eusmodel-mesh-spec *collada::link* [関数]
collada::eusmodel-joint-spec *collada::_joint* [関数]
collada::eusmodel-limit-spec *collada::_joint* [関数]
collada::eusmodel-endcoords-specs *collada::model* [関数]
collada::eusmodel-link-description *collada::description* [関数]
collada::eusmodel-joint-description *collada::description* [関数]
collada::eusmodel-endcoords-description *collada::description* [関数]
collada::setup-collada-filesystem *collada::obj-name collada::base-dir* [関数]
collada::range2 *collada::n* [関数]
eus2collada *collada::obj collada::full-root-dir &key (scale 0.001) (collada::file-suffix .dae)* [関数]
collada::collada-node-id *collada::link-descrption* [関数]
collada::collada-node-name *collada::link-descrption* [関数]
collada::links-description->collada-library-materials *collada::links-desc* [関数]
collada::link-description->collada-materials *collada::link-desc* [関数]
collada::mesh-description->collada-material *collada::mat collada::effect* [関数]
collada::links-description->collada-library-effects *collada::links-desc* [関数]
collada::link-description->collada-effects *collada::link-desc* [関数]
collada::mesh-description->collada-effect *collada::mesh id* [関数]
collada::matrix->collada-string *collada::mat* [関数]
collada::find-parent-liks-from-link-description *collada::target-link collada::desc* [関数]
collada::eusmodel-description->collada-node-transformations *collada::target-link collada::desc* [関数]
collada::eusmodel-description->collada-node *collada::target-link collada::desc* [関数]
collada::eusmodel-description->collada-library-visual-scenes *name collada::desc* [関数]
collada::mesh-description->instance-material *collada::s* [関数]
collada::link-description->collada-bind-material *collada::link* [関数]

collada::eusmodel-description->collada-library-kinematics-scenes *name collada::desc* [関数]
collada::eusmodel-description->collada-library-kinematics-models *name collada::desc* [関数]
collada::eusmodel-description->collada-kinematics-model *name collada::desc* [関数]
collada::eusmodel-description->collada-library-physics-scenes *name collada::desc* [関数]
collada::eusmodel-description->collada-library-physics-models *name collada::desc* [関数]
collada::find-root-link-from-joints-description *collada::joints-description* [関数]
collada::find-link-from-links-description *name collada::links-description* [関数]
collada::eusmodel-description->collada-links *collada::description* [関数]
collada::find-joint-from-link-description *collada::target collada::joints* [関数]
collada::find-child-link-descriptions *collada::parent collada::links collada::joints* [関数]
collada::eusmodel-description->collada-library-articulated-systems *collada::desc name* [関数]
collada::eusmodel-endcoords-description->openrave-manipulator *collada::end-coords collada::description* [関数]
collada::eusmodel-description->collada-links-tree *collada::target collada::links collada::joints* [関数]
collada::joints-description->collada-instance-joints *collada::joints-desc* [関数]
collada::joint-description->collada-instance-joint *collada::joint-desc* [関数]
collada::eusmodel-description->collada-library-joints *collada::description* [関数]
collada::joints-description->collada-joints *collada::joints-description* [関数]
collada::collada-joint-id *collada::joint-description* [関数]
collada::joint-description->collada-joint *collada::joint-description* [関数]
collada::linear-joint-description->collada-joint *collada::joint-description* [関数]
collada::rotational-joint-description->collada-joint *collada::joint-description* [関数]
collada::eusmodel-description->collada-scene *collada::description* [関数]
collada::eusmodel-description->collada-library-geometries *collada::description* [関数]
collada::collada-geometry-id-base *collada::link-descrption* [関数]
collada::collada-geometry-name-base *collada::link-descrption* [関数]
collada::links-description->collada-geometries *collada::links-description* [関数]
collada::mesh-object->collada-mesh *collada::mesh id* [関数]
collada::link-description->collada-geometry *collada::link-description* [関数]
collada::mesh->collada-indices *collada::mesh* [関数]
collada::mesh-vertices->collada-string *collada::mesh* [関数]
collada::mesh-normals->collada-string *collada::mesh* [関数]
collada::float-vector->collada-string *collada::v* [関数]
collada::enum-integer-list *collada::n* [関数]
collada::search-minimum-rotation-matrix *collada::mat* [関数]
collada::estimate-class-name *collada::model-file* [関数]
collada::remove-directory-name *fname* [関数]

23 ポイントクラウドデータ

pointcloud [クラス]

:super **cascaded-coords**
:slots parray carray narray curvature pcolor psize awidth asize box height width view-coord

:init *&rest args &key* *((:points mat))* [method]
((:colors cary))
((:normals nary))
((:curvatures curv))

((:height ht))
((:width wd))
(point-color (float-vector 0 1 0))
(point-size 2.0)
(fill)
(arrow-width 2.0)
(arrow-size 0.0)

Create point cloud object

:size-change *&optional wd ht* [メソッド]

change width and height, this method does not change points data

:points *&optional pts wd ht* [メソッド]

replace points, pts should be list of points or n*times* matrix

:colors *&optional cls* [メソッド]

replace colors, cls should be list of points or n*times* matrix

:normals *&optional nmls* [メソッド]

replace normals by, nmls should be list of points or n*times*3 matrix

:point-list *&optional (remove-nan)* [メソッド]

return list of points

:color-list [メソッド]

return list of colors

:normal-list [メソッド]

return list of normals

:centroid [メソッド]

retrun centroid of this point cloud

:append *point-list &key (create t)* [メソッド]

append point cloud list to this point cloud.

if :create is true, return appended point cloud and original point cloud does not change.

:filter *&rest args &key create* [method]
&allow-other-keys

this method can take the same keywords with :filter-with-indices method.

if :create is true, return filtered point cloud and original point cloud does not change.

:filter-with-indices *idx-lst &key (create)* [method]
(negative)

filter point cloud with list of index (points which are indicated by indices will remain).

if :create is true, return filtered point cloud and original point cloud does not change.

if :negative is true, points which are indicated by indices will be removed.

:filtered-indices *&key key* [method]
ckey
nkey
pckey
pnkey
pcnkey
negative
&allow-other-keys

create list of index where filter function retrun true.

key, ckey, nkey are filter function for points, colors, normals. They are expected to take one argument and return t or nil.

pckey, pnkey are filter function for points and colors, points and normals. They are expected to take two arguments and return t or nil.

pcnkey is filter function for points, colors and normals. It is expected to take three arguments and return t or nil.

:step *step &key (fixsize)* [method]
(create)

downsample points with step

:copy-from *pc* [メソッド]

update object by pc

:transform-points *coords &key (create)* [メソッド]

transform points and normals with coords.

if :create is true, return transformed point cloud and original point cloud does not change.

:convert-to-world *&key (create)* [メソッド]

transform points and normals with self coords. points data should be the same as displayed

:reset-box [メソッド]

:box [メソッド]

:vertices [メソッド]

:size [メソッド]

:width [メソッド]

:height [メソッド]

:view-coords *&optional vc* [メソッド]

:curvatures *&optional curv* [メソッド]

:curvature-list [メソッド]

:set-color *col &optional (_transparent)* [メソッド]

:point-color *&optional pc* [メソッド]

:point-size *&optional ps* [メソッド]

:axis-length *&optional al* [メソッド]

:axis-width *&optional aw* [メソッド]

:clear-color [メソッド]

:clear-normal [メソッド]

:nfilter *&rest args* [メソッド]

:viewangle-inlier *&key (min-z 0.0) (hangle 44.0) (vangle 35.0)* [メソッド]

:image-position-inlier *&key (ipkey) (height 144) (width 176) (cy (/ (float (- height 1)) 2)) (cx (/ (float (- width 1)) 2)) negative* [メソッド]

:image-circle-filter *dist &key (height 144) (width 176) (cy (/ (float (- height 1)) 2)) (cx (/ (float (- width 1)) 2)) create negative* [メソッド]

:step-inlier *step offx offy* [メソッド]

:generate-color-histogram-hs *&key (h-step 9) (s-step 7) (hlimits (cons 360.0 0.0)) (vlimits (cons 1.0 0.15)) (slimits (cons 1.0 0.25)) (rotate-hue) (color-scale 255.0) (sizelimits 1)* [メソッド]

:set-offset *cds &key (create)* [メソッド]

:drawnormalmode *&optional mode* [メソッド]

:transparent *&optional trs* [メソッド]

:draw *vwer* [メソッド]

make-random-pointcloud *&key (num 1000) (with-color) (with-normal) (scale 100.0)* [関数]

24 グラフ表現

node [クラス]

:super **propertied-object**

:slots arc-list

:init *n* [メソッド]

:arc-list [メソッド]

:successors [メソッド]

:add-arc *a* [メソッド]

:remove-arc *a* [メソッド]

:remove-all-arcs [メソッド]

:unlink *n* [メソッド]

:add-neighbor *n &optional a* [メソッド]

:neighbors *&optional args* [メソッド]

arc [クラス]

:super **propertied-object**
:slots from to

:init *from_ to_* [メソッド]

:from [メソッド]

:to [メソッド]

:prin1 *&optional (strm t) &rest msgs* [メソッド]

directed-graph [クラス]

:super **propertied-object**
:slots nodes

:init [メソッド]

:successors *node &rest args* [メソッド]

:node *name* [メソッド]

:nodes *&optional arg* [メソッド]

:add-node *n* [メソッド]

:remove-node *n* [メソッド]

:clear-nodes [メソッド]

:add-arc-from-to *from to* [メソッド]

:remove-arc *arc* [メソッド]

:adjacency-matrix [メソッド]

:adjacency-list [メソッド]

:write-to-dot *fname &optional result-path (title output)* [メソッド]

:write-to-pdf *fname &optional result-path (title (string-right-trim .pdf fname))* [メソッド]

costed-arc [クラス]

:super **arc**
:slots cost

:init *from to c* [メソッド]

:cost [メソッド]

costed-graph [クラス]

:super **directed-graph**
:slots nil

:add-arc *from to cost &key (both nil)* [メソッド]

:add-arc-from-to *from to cost &key (both nil)* [メソッド]

:path-cost *from arc to* [メソッド]

graph [クラス]

:super **costed-graph**
:slots start-state goal-state

:goal-test *gs* [メソッド]

:path-cost *from arc to* [メソッド]

:start-state *&optional arg* [メソッド]

:goal-state *&optional arg* [メソッド]

:add-arc *from to &key (both nil)* [メソッド]

:add-arc-from-to *from to &key (both nil)* [メソッド]

arced-node [クラス]

:super **node**
:slots nil

:init *&key name* [メソッド]

:find-action *n* [メソッド]

:neighbor-action-alist [メソッド]

solver-node [クラス]

:super **propertied-object**
:slots state cost parent action memorized-path

:init *st &key ((:cost c) 0) ((:parent p) nil) ((:action a) nil)* [メソッド]

:path *&optional (prev nil)* [メソッド]

:expand *prblm &rest args* [メソッド]

:state *&optional arg* [メソッド]

:cost *&optional arg* [メソッド]

:parent *&optional arg* [メソッド]

:action *&optional arg* [メソッド]

solver [クラス]

:super **propertied-object**
:slots nil

:init [メソッド]

:solve *prblm* [メソッド]

:solve-by-name *prblm s g &key (verbose nil)* [メソッド]

graph-search-solver [クラス]

:super **solver**
:slots open-list close-list

:solve-init *prblm* [メソッド]

:find-node-in-close-list *n* [メソッド]

:solve *prblm &key (verbose nil)* [メソッド]

:add-to-open-list *obj/list* [メソッド]

:null-open-list? [メソッド]

:clear-open-list [メソッド]

:add-list-to-open-list *lst* [メソッド]

:add-object-to-open-list *lst* [メソッド]

:pop-from-open-list *&key (debug)* [メソッド]

:open-list *&optional arg* [メソッド]

:close-list *&optional arg* [メソッド]

breadth-first-graph-search-solver [クラス]

:super **graph-search-solver**
:slots nil

:init [メソッド]

:clear-open-list [メソッド]

:add-list-to-open-list *lst* [メソッド]

:add-object-to-open-list *obj* [メソッド]

:pop-from-open-list *&key (debug)* [メソッド]

depth-first-graph-search-solver [クラス]

:super **graph-search-solver**
:slots nil

:init [メソッド]

:clear-open-list [メソッド]

:add-list-to-open-list *lst* [メソッド]

:add-object-to-open-list *obj* [メソッド]

:pop-from-open-list *&key (debug)* [メソッド]

best-first-graph-search-solver [クラス]

:super **graph-search-solver**
:slots aproblem

:init *p* [メソッド]

:clear-open-list [メソッド]

:add-list-to-open-list *lst* [メソッド]

:add-object-to-open-list *obj* [メソッド]

:pop-from-open-list *&key (debug nil)* [メソッド]

:fn *n p* [メソッド]

a*-graph-search-solver [クラス]

:super **best-first-graph-search-solver**
:slots nil

:init *p* [メソッド]

:fn *n p &key (debug nil)* [メソッド]

:gn *n p* [メソッド]

:hn *n p* [メソッド]

25 irteus拡張

25.1 GL/X表示

gl::glvertices [クラス]

:super **cascaded-coords**
:slots gl::mesh-list gl::filename gl::bbox

:set-color *gl::color &optional (gl::transparent)* [メソッド]
set color as float vector of 3 elements, and transparent as float, all values are betwenn 0 to 1

:actual-vertices [メソッド]
return list of vertices(float-vector), it returns all vertices of this object

:calc-bounding-box [メソッド]
calculate and set bounding box of this object

:vertices [メソッド]
return list of vertices(float-vector), it returns vertices of bounding box of this object

:reset-offset-from-parent [メソッド]

move vertices in this object using self transformation, this method change values of vertices. coordinates's method such as :transform just change view of this object

:expand-vertices [メソッド]
expand vertices number as same number of indices, it enable to set individual normal to every vertices

:use-flat-shader [メソッド]
use flat shader mode, use opengl function of glShadeModel(GL_FLAT)

:use-smooth-shader [メソッド]
use smooth shader mode, use opengl function of glShadeModel(GL_SMOOTH) default

:calc-normals *&optional (gl::force nil) (gl::expand t) (gl::flat t)* [メソッド]
normal calculation
if force option is true, clear current normalset.
if exapnd option is ture, do :expand-vertices.
if flat option is true, use-flat-shader

:mirror-axis *&key (gl::create t) (gl::invert-faces t) (gl::axis :y)* [method]
creating mirror vertices respect to :axis

:convert-to-faces *&rest args &key (gl::wrt :local) &allow-other-keys* [method]
create list of faces using vertices of this object

:convert-to-faceset *&rest args* [メソッド]
create faceset using vertices of this object

:set-offset *gl::cds &key (gl::create)* [メソッド]
move vertices in this object using given coordinates, this method change values of vertices. coordinates's method such as :transform just change view of this object

:convert-to-world *&key (gl::create)* [メソッド]
move vertices in this object using self's coordinates. vertices data should be moved as the same as displayed

:glvertices *&optional (name) (gl::test #'string=)* [メソッド]
create individual glvertices objects from mesh-list. if name is given, search mesh has the same name

:append-glvertices *gl::glv-lst* [メソッド]
append list of glvertices to this object

:init *gl::mlst &rest args &key ((:filename gl::fn)) &allow-other-keys* [メソッド]

:filename *&optional gl::nm* [メソッド]

:get-meshinfo *gl::key &optional (pos -1)* [メソッド]

:set-meshinfo *gl::key gl::info &optional (pos -1)* [メソッド]

:get-material *&optional (pos -1)* [メソッド]

:set-material *gl::mat &optional (pos -1)* [メソッド]

:expand-vertices-info *gl::minfo* [メソッド]

:faces [メソッド]

:draw-on *&key ((:viewer gl::vwer) *viewer*)* [メソッド]

:draw *gl::vwr &rest args* [メソッド]

:collision-check-objects *&rest args* [メソッド]

:box [メソッド]

gl::glbody [クラス]

:super **body**
:slots gl::aglvertices

:glvertices *&rest args* [メソッド]

:draw *gl::vwr* [メソッド]

:set-color *&rest args* [メソッド]

gl::find-color *gl::color* [関数]

returns color vector of given color name, the name is defiend in https://github.com/euslisp/jskeus/blob/master/ir

gl::transparent *gl::abody gl::param* [関数]

Set abody to transparent with param

gl::make-glvertices-from-faceset *gl::fs &key (gl::material)* [関数]

returns glvertices instance
fs is geomatry::faceset

gl::make-glvertices-from-faces *gl::flst &key (gl::material)* [関数]

returns glvertices instance
flst is list of geomatry::face

gl::write-wrl-from-glvertices *fname gl::glv &rest args* [関数]

write .wrl file from instance of glvertices

gl::set-stereo-gl-attribute [関数]

gl::reset-gl-attribute [関数]

gl::delete-displaylist-id *gl::dllst* [関数]

gl::draw-globjects *gl::vwr gl::draw-things &key (gl::clear t) (gl::flush t) (gl::draw-origin 150) (gl::draw-floor nil)* [関数]

gl::draw-glbody *gl::vwr gl::abody* [関数]

gl::_dump-wrl-shape *gl::strm gl::mesh &key ((:scale gl::scl) 1.0) (gl::use_ambient nil) (gl::use_normal nil) (gl::use_texture nil) &allow-other-keys* [関数]

x::tabbed-panel [クラス]

:super **x:panel**
:slots x::tabbed-buttons x::tabbed-panels x::selected-tabbed-panel

:create *&rest args* [メソッド]

:add-tabbed-panel *name* [メソッド]

:change-tabbed-panel *x::obj* [メソッド]

:tabbed-button *name &rest args* [メソッド]

:tabbed-panel *name &rest args* [メソッド]

:resize *x::w h* [メソッド]

x::panel-tab-button-item [クラス]

:super **x:button-item**
:slots nil

:draw-label *&optional (x::state :up) (x::offset 0)* [メソッド]
x::window-main-one *&optional fd* [関数]
x::event-far *x::e* [関数]
x::event-near *x::e* [関数]

25.2 ユーティリティ関数

mtimer [クラス]

:super **object**
:slots buf

:init [メソッド]

Initialize timer object.

:start [メソッド]

Start timer.

:stop [メソッド]

Stop timer and returns elapsed time in seconds.

permutation *lst n* [関数]

Returns permutation of given list

combination *lst n* [関数]

Returns combination of given list

find-extreams *datum &key (key #'identity)* [function]

(identity #'=)
(bigger #'>)

Returns the elements of datum which maximizes key function

eus-server *&optional (port 6666) &key (host (unix:gethostname))* [関数]

Create euslisp interpreter server, data sent to socket is evaluated as lisp expression

connect-server-until-success *host port &key (max-port (+ port 20)) (return-with-port nil)* [function]

Connect euslisp interpreter server until success

format-array *arr &optional (header) (in 7) (fl 3) (strm *error-output*) (use-line-break t)* [関数]

print formatted array

his2rgb *h &optional (i 1.0) (s 1.0) ret* [関数]

convert his to rgb (0 <= h <= 360, 0.0 <= i <= 1.0, 0.0 <= s <= 1.0)

hvs2rgb *h &optional (i 1.0) (s 1.0) ret* [関数]

convert hvs to rgb (0 <= h <= 360, 0.0 <= i <= 1.0, 0.0 <= s <= 1.0)

rgb2his *r &optional g b ret* [関数]

convert rgb to his (0 <= r,g,b <= 255)

rgb2hvs *r &optional g b ret* [関数]

convert rt to hvs (0 <= r,g,b <= 255)

color-category10 *i* [関数]

Choose good color from 10 colors

color-category20 *i* [関数]

Choose good color from 20 colors

make-robot-model-from-name *name &rest args* [関数]

make a robot model from string: (make-robot-model "pr2")

forward-message-to *to args* [関数]
forward-message-to-all *to args* [関数]
mapjoin *expr seq1 seq2* [関数]
need-thread *n &optional (lsize (*512 1024)) (csize lsize)* [関数]
piped-fork-returns-list *cmd &optional args* [関数]

25.3 数学関数

inverse-matrix *mat* [関数]

returns inverse matrix of mat

diagonal *v* [関数]

make diagonal matrix from given vecgtor, diagonal #f(1 2) ->#2f((1 0)(0 2))

minor-matrix *m ic jc* [関数]

return a matrix removing ic row and jc col elements from m

atan2 *y x* [関数]
returns atan2 of y and x (atan (/ y x))

outer-product-matrix *v &optional (ret (unit-matrix 3))* [関数]
returns outer product matrix of given v

```
matrix(a) v = a *v

0 -w2 w1

w2 0 -w0

-w1 w0 0
```

matrix2quaternion *m* [関数]
returns quaternion of given matrix

quaternion2matrix *q* [関数]
returns matrix of given quaternion

matrix-log *m* [関数]
returns matrix log of given m, it returns [-pi, pi]

matrix-exponent *omega &optional (p 1.0)* [関数]
returns exponent of given omega

midrot *p r1 r2* [関数]
returns mid (or p) rotation matrix of given two matrix r1 and r1

pseudo-inverse *mat &optional weight-vector ret wmat mat-tmp* [関数]
returns pseudo inverse of given mat

sr-inverse *mat &optional (k 1.0) weight-vector ret wmat tmat umat umat2 mat-tmp mat-tmp-rc mat-tmp-rr mat-tmp-rr2* [関数]
returns sr-inverse of given mat

manipulability *jacobi &optional tmp-mrr tmp-mcr* [関数]
return manipulability of given matrix

random-gauss *&optional (m 0) (s 1)* [関数]
make random gauss, m:mean s:standard-deviation

gaussian-random *dim &optional (m 0) (s 1)* [関数]
make random gauss vector, replacement for quasi-random defined in matlib.c

normalize-vector *v &optional r (eps 1.000000e-20)* [関数]
calculate normalize-vector #f(0 0 0)->#f(0 0 0).

pseudo-inverse-org *m &optional ret winv mat-tmp-cr* [関数]
sr-inverse-org *mat &optional (k 1) me mat-tmp-cr mat-tmp-rr* [関数]

eigen-decompose *m* [関数]

lms *point-list* [関数]

lms-estimate *res point-* [関数]

lms-error *result point-list* [関数]

lmeds *point-list &key (num 5) (err-rate 0.3) (iteration) (ransac-threshold) (lms-func #'lms) (lmeds-error-func #'lmeds-error) (lms-estimate-func #'lms-estimate)* [関数]

lmeds-error *result point-list &key (lms-estimate-func #'lms-estimate)* [関数]

lmeds-error-mat *result mat &key (lms-estimate-func #'lms-estimate)* [関数]

25.4 画像関数

read-image-file *fname* [関数]

read image of given fname. It returns instance of **grayscale-image** or **color-image24**.

write-image-file *fname image::img* [関数]

write img to given fname

read-png-file *fname* [関数]

write-png-file *fname image::img* [関数]