Common Lisp と 人工知能プログラミング



小出 誠二

オントロノミー合同会社

目次

謝辞
添付フォントと制作ツール
序章 はじめに
なぜ Common Lisp か？
誰がこの本を読むのか？
なぜこの本が必要なのか？
どの Common Lisp 処理系を使うのか？
この本の構成
最近のAIブームとCommon Lisp
本書ホームページとGitHubページ
第1章 Lispの基本
1.1 リストを入力してみよう
Lispの対話型機能
Lispは関数型言語である
Lispの関数呼び出しは前置形式(prefix notation)である
リスト，アトム，そしてS式
シンボル
1.2 書式を評価する
quoteは評価を抑制する
1.3 carとcdrはリストを分解する
1.4 シンボルは値を持つことができる
setqの前の引数は評価されないが後ろの引数は評価される
1.5 consはリストを合成する
空リストはnilと同じである
1.6 便利なリスト処理関数
キーバリュー・ペア
1.7 数，整数，浮動小数点数，有理数
第2章 関数定義および述語と条件文
2.1 関数定義の書式
ラムダ式で無名関数を作る
2.2 述語
2.3 論理結合演算
2.4 条件文
if 特殊形式
cond マクロ
case, ecase, typecase
第3章 再帰プログラムを組む
3.1 リストの横方向に再帰処理する
再帰しながらその帰りに簡単な仕事をする場合がある
再帰から戻ったときにリストを構成する場合がある
3.2 リストの縦方向にも再帰処理する
一般的には，リストを対象とする再帰の枝は，横方向（巾方向）と縦方向（深さ方向）の二つである
第4章 変数束縛，動的変数，大域変数
4.1 ラムダ変数束縛

4.2 let 変数束縛
4.3 特殊変数と動的スコープ
シンボル vs. 変数
特殊変数と大域変数
4.4 defvar, defparameter, defconstant
大域変数は動的変数
第5章 Lisp の内幕
5.1 consセル
5.2 rplaca，rplacd，そして nconc
5.3 破壊的関数
5.4 シンボルとキーワード
属性リスト
パッケージ
内部シンボル，外部シンボル
キーワードシンボル
5.5 汎参照（汎変数）
defsetf
第6章 高階関数と MAP 関数
6.1 関数は第1級のクラス
6.2 ラムダ式再び
付加パラメータ
レストパラメータ
キーワードパラメータ
付加パラメータとレストパラメータの組み合わせ
レストパラメータとキーワードパラメータの併用
6.3 MAP 関数
mapcar で得られる nil をつぶしたいときは mapcan (mappend) を使う
6.4 その他の MAP 関数
some と every
MAP と REDUCE
第7章 関数閉包とエクステント
7.1 関数閉包
7.2 エクステントとは何か？
誰も触れない秘密の変数
第 8 章 マクロを書く
8.1 マクロ展開
8.2 バッククォート・テンプレート
let 構文を自作する
8.3 変数の衝突と once-only
変数の衝突
マクロ呼び出しパラメータに副作用がある時は once-only
8.4 Elizaルール
8.5 マクロにおけるラムダパラメータ
8.6 マクロを書くマクロ
第9章 コンパイラ
9.1 load と compile と compile-file
単独の関数をコンパイルする
ソースファイルをコンパイルする

9.2 eval-when で評価時期のコントロール
9.3 #. 形式でコンパイル時の値を固定する
第10章 末尾再帰，継続，コルーチン
10.1 末尾再帰
10.2 継続
10.3 コルーチン
第11章 Lisp で作る Scheme
11.1 末尾再帰なしのインタープリタ
最低限の関数
環境と変数
Scheme マクロ
11.2 末尾再帰のインタープリタ
11.3 call/cc によるインタープリタ
call/ccによる非局所的脱出
call/ccを用いたreader/writer
11.4 動的束縛の実装
動的束縛のみの実装
動的束縛をオプションとするScheme
第12章 配列，文字列，列
12.1 配列
12.2 ベクタ
12.3 配列の共有，フィルポインタ，可変配列
displaced-to
フィルポインタ
可変配列
12.4 配列の型階層と特殊配列
12.5 文字と文字列
エスケープ文字
12.6 列（シーケンス）
12.7 ハッシュ表
第13章 Lisp の型システム
13.1 データ型，型指定子
13.2 Lisp の型階層
型階層は束
網羅的和集合と網羅的分割
上下関係が計算できない？
Type Upgrading
13.3 新しい型の定義
論理結合の型指定子
メンバー指定の型指定子
数の範囲指定の型指定子
配列要素の型と長さを特殊化した型指定子
新規の型指定子
deftype
第14章 構造体
14.1 Common Lisp の Universal Time
グリニッチ標準時GMTと協定世界時UTC
14.2 date 構造体をオプションなしで定義する

14.3 defstructオプション
14.4 includeオプションによる型階層定義
第15章 CLOS とメタオブジェクト・プロトコル
15.1 オブジェクト指向とは何か？
15.2 総称関数とメソッド
多相性（polymorphism）
スロット特定子を使ってスロットアクセッサを定義
with-slots でスロットアクセス
初期化引数を使う(initarg)
スロット規定値を定義する(initform)
15.3 上位クラスと下位クラス
CLOS クラスは Common Lisp の型である
実現体はそのクラスとすべての上位クラスからスロットを継承する
下位クラスの情報は上位クラスに優先する
円と楕円，どちらが上位概念か？
上位クラスのメソッドの継承と下位クラスのメソッドの優先
15.4 メソッド修飾子
15.5 多重継承のクラス優先順位
15.6 メソッド結合の優先順位
15.7 文書検索，もう一つのオブジェクト指向的プログラミング
トークナイザは最初にトークンを切り出す
ノルマライザがある
文書アナライザがある
パイプラインを構成するのはプログラマーの仕事
コーパスがある
インデックス・ストア
プロジェクト・グーテンベルグ
15.8 CLOS と型システム
CLOSクラスは型である
構造体に対するメソッド定義
データ型に対するメソッド定義
15.9 複数引数におけるメソッドディスパッチ
15.10 CLOSプログラミングとMOP
CLOSプログラマ・インタフェース機能
メタクラスとは？
メタクラス・プログラミング
リフレクションとは？
standard-classのメタ巡回（circularity)と解釈可能クラス階層
standard-objectとstandard-classの奇妙な関係
第16章 Format の呪文
16.1 format 構文
16.2 文字列出力制御
大文字小文字変換
16.3 数出力制御
英語の複数形
16.4 繰返し構文
16.5 選択構文

16.6 幅揃え
16.7 プリティプリント
プリティプリントとは何か？
フォーマット指示子によるプリティプリント
プリティプリント論理ブロック
第17章 Loop の呪文
17.1 繰り返し
17.2 無条件実行節
17.3 蓄積節
リスト蓄積節
数蓄積節
17.4 条件実行節
17.5 コレクションについての for-as 節
リストについての for-as 節
ベクタについての for-as 節
ハッシュ表についての for-as 節
パッケージについての for-as 節
17.6 数字指定の for-as 節
17.7 多重のループ
17.8 変数の for-as 節
17.9 While と Until ループ
17.10 alwaysとthereisとnever
17.11 一時的変数の導入
17.12 分配束縛
第18章 エラー，警告，コンディションシステム
18.1 エラー
継続不可能なエラー
継続可能なエラー
18.2 警告
18.3 コンディションシステム
第19章 多値，ファイル，入出力ストリーム
19.1 多値
19.2 ファイル
物理パスネームと論理パスネーム
ロード時のディレクトリをアプリのホームにする
19.3 入出力ストリーム
定義済み入出力ストリーム
with-open-fileを使う
その他のストリーム
第20章 パッケージとモジュール
20.1 パッケージの定義方法と使い方
カレントパッケージと他パッケージの利用
(:export ...)には何を書くのか？
シャドウとシャドウイング・インポート
20.2 モジュールとは？
20.3 ASDF3とquicklisp
ファイルのモジュール化とシステム定義

quicklispとASDF
20.4 処理系依存の条件付き読み込み
付録1 Allegro Express のインストール
付1.1 ANSI Common Lisp with IDE のダウンロードとインストール
付1.1.1 Windows 32bit 版のインストール
付1.1.2 Ubuntu に Allegro Express をインストール
付1.1.3 Mac に Allegro express をインストール
付1.1.4 Windows/Wsl に Allegro express をインストール
Windows/Wsl のインストール
Wsl/Ubuntu に Allegro Express for Linux をインストール
付1.2 Modern Common Lisp の作成
付1.3 ACL with IDE の使い方
ACL プロジェクト
エディタと自動インデント
付1.4 ブレーク時の対処方法
付1.5 describeとinspect
付1.6 システムの終わり方
付録2 SBCL のインストール
付2.1 Ubuntu にパッケージとしてインストール
付2.2 最新バイナリーパッケージのインストール
付2.3 SBCL ライブラリ sb-aclrepl を要求する
付2.4 ライブラリマネージャ quicklisp を利用する
付2.5 開発環境 emacs+slime+sbcl
付2.6 ブレーク時の対処方法
付録3 マクロ文字とオブジェクト印字関数
付3.1 Homoiconic ということ
付3.2 オブジェクト印字関数のカスタマイズ
付3.3 マクロ文字によるread構文のカスタマイズ
付録４ Armed Bear Common Lisp (ABCL)のインストールとビルド
付4.1 Ubuntuにパッケージとしてインストール
付4.2 Apache ant によるソースコードからのビルド
付録５ GNU CLISPのインストール
付5.1 Ubuntuにパッケージとしてインストール
付録６ Paul Grahamの継続関数
付6.1 継続渡しマクロ
付6.2 継続をベースにしたマルチプロセスの実現
付6.3 継続をベースにした非決定的プログラミング
付録７ Test Harnesの使用法
テスト駆動開発について
付7.1 Allegro Common Lisp の Test Harness
参考文献（上）
第21章 人工知能プログラミング環境
21.1 PAIPとAIMAのプログラミング環境
21.2 人工知能用ユーティリティプログラム
リスト用関数
queue
reuse-cons
debug

delay と force
パターンマッチ
第22章 Elizaまたの名を人工無脳
22.1 Elizaとは何か？
22.2 ダウンロードと eliza1 の実行
22.3 eliza のコードウォークスルー
22.4 elizaプログラムの改良
22.5 ルールの追加
22.6 Eliza とは何であったか？
第23章 一般問題解決器
23.1 手段・目標解析，Means-Ends Analysis
23.2 状態，目標，作用子
23.3 GPS はどのように働くのか？(第１版）
23.4 世界をきっちりと記述しよう，宇宙ロボットによるORU交換
23.5 プログラムの改善（GPS 第2版）
単純な，あちらを立てればこちらが立たず問題
Sassman アノマリ
23.6 GPS とはなんであったか？
人間の問題解決行動
論理プログラミングからの視点
計画プログラムからの視点
第24章 推論と探索
24.1 前向き推論と後向き推論
前向き推論は事実から帰結に向かって推論する
後向き推論では仮説から事実に向かって推論する
24.2 AND/ORグラフと推論
24.3 米国飛行機旅行経路探索
深さ優先探索
幅優先探索
最良優先探索
A* アルゴリズム
24.4 推論にまつわる話題
第25章 フレームシステム
25.1 知識表現の枠組み
25.2 XRL システムのロードと実行
25.3 XRL のプログラムコード
25.4 フレームとはなんであったか？
第26章 ルールベースシステム
26.1 ルールベースシステムとは
26.2 MYCIN のロードと実行
Mycin コンテキスト
Mycin 確信度
Mycin キャッシング
Mycin パラメータ
Mycin ルール
Mycin 後向き推論
Mycin 対話インタフェース

26.3 エキスパートシステムとはなんであったか？
第27章 事例ベース推論
27.1 事例とは何か？
27.2 メモリ機構
MOPシステム
シンボルの属性リストを利用したmopの実現
抽象関係を判定する
新しい mop を生成する
Slotの束からmopを作る
mop をインストールするとはどういうことか？
mop 定義マクロ
属性とその値の継承
ロール・チェーン探索
パターン mop とは何か
27.3 カーネルシステムの組込みmopと局所手続き
グループとシーケンス
制約パターン
ケース
メモリ管理とユーティリティ
CBRプログラミング例
スロットマッチングによる分類
マッチング制御のための制約記述
27.4 判例ベース・システム Judge
27.5 Direct Memory Access Parser (DMAP)
27.6 事例ベース推論とは何であったか？
第28章 命題論理
28.1 命題論理とは何か
28.2 命題論理の演算
妥当性と世界モデル
28.3 命題論理式とその計算
命題論理式の統語論
28.4 命題論理式のプログラム
第29章 ユニフィケーション
29.1 ユニフィケーション（単一化）とは？
29.2 非破壊的単一化プログラム
29.3 破壊的単一化プログラム
29.4 複合物単一化プログラム
29.5 型を考慮した単一化プログラム
第30章 述語論理
30.1 述語論理とは何か
30.2 一階述語論理の実際
30.2.1 CLIF の Lisp 構文
30.2.2 AIMAのLispコード
30.2.3 SNARKのダウンロードとインストール
第31章 プラニング
31.1 状態空間プランニング
31.1.1 前向き探索

31.1.2 状態空間の論理記述
31.1.3 オペレータの定義
第32章 Lispで作るProlog
32.1 Prolog を実行する
32.2 prolog.lisp （インタープリタ版）のコードウォークスルー
32.3 Prolog プログラミング
32.4 Prolog の自動バックトラッキング
32.5 Prologc(コンパイラ版)のコードウォークスルー
32.6 Prolog の組み込み述語
終章 おわりに
付録 8 弁別木による知識ストア dtree
付8.1 弁別木による知識ストア dtree
付8.2 カテゴリ，個体，関係，値域・領域制約
付録 9 知識表現言語 krep
付9.1 弁別木による知識ストア dtree
付9.2 カテゴリ，個体，関係，値域・領域制約
参考文献（下）

Common Lispと人工知能プログラミング[全]

著者：小出誠二　©2017, 2018, 2019, 2020, 2021 2022, 2023, 2024, 2025

発行所：オントロノミー合同会社

NDC：007.64
NDC：007.13

ISBN：978-4-9909092-1-5 (ePub, mobi, azw3)

C3004 ¥3636E

謝辞

本書執筆では，上巻原稿完了前の段階で，内容のブラッシュアップと市場調査を目的に，３回連続の紹介講座を東京と横浜において開き，総勢60名の方々に参加をいただいた．参加者の中にはCommon Lispについてベテランの方々もいて，細かな誤りの修正から正確な記述への改善まで，記述内容を大いに向上することができた．講座参加者の全員に感謝する．中でも私がLisp考古学者とあだ名をつけた千葉将臣氏と，Schemeの動的束縛のプログラムを提供していただいた初谷久史氏には，その名を挙げて感謝の意を表する．

添付フォントと制作ツール

この電子書籍では，本体の日本語と英語には拡張IPAフォントを，コード部分にはRictyDiminishedフォントを用いている．いずれも現状のままで自由に配布できるフォントである．開発元の独立行政法人情報処理推進機構(IPA)と遊佐泰紀氏に感謝する．

この電子書籍の執筆にはSigilを，epubからmobiおよびazw3への変換にはcalibreを用いた．開発元であるコミュニティに感謝する．

序章 はじめに

> LISPは冗談に「計算機を誤用するための最も賢い方法」であるといわれてきた．この描写は大変な賛辞であると私は思う．この言葉によってLISPの持つ完全に自由な雰囲気が伝わってくるからである．ーーー LISPは，我々の最も才能のある人々が，以前は不可能であった試行を行う手助けをしてきたのである．
>
> [Dijkstra, 1972]（白井・片桐訳，1986）

> ハッカーにとって，簡潔さより大事なことがある．自分のやりたいことがやれることだ．プログラミング言語の歴史をふりかえってみると，「正しくない」と考えられる行いをプログラマがするのを防ぐために驚くべき程の努力が払われて来た．これは危険なほどにおこがましい計画である．プログラマが必要とするであろうことを，どうやって言語設計者はあらかじめ知ることが出来ると言うのだ？ 言語設計者は，ユーザを自分のミスから守ってやらなきゃならないようなまぬけではなく，設計者が考えもしなかったようなことを実現できる天才と考えた方が良いと思う． 何をしようがまぬけは自分の足を撃つのだ．
>
> [Paul Graham, 2001]（川合訳, 2004）

なぜ Common Lisp か？

この書籍では， Common Lispを用いて人工知能プログラミングの基礎から簡単な応用までを学ぶ．なぜ人工知能プログラミングにCommon Lispかといえば，それが今ではGOFAI(Good Old Fashoned AI)と呼ばれる人工知能を学ぶのに最適な計算機言語だからである．そもそも人工知能という言葉が生まれたのは，1956年に米国ダートマス大学で行われた会議と言われている．そこで John McCarthy は IPLという言語で書かれた Logic Theoristというはじめての人工知能プログラムを見た．それをきっかけとして， McCarthyがIPLと似た人工知能用の言語を学生と一緒に開発したのがLispである．その後，米国では主にLispをベースに，多くの人工知能システムが開発され，コミュニティに広がっていったが，インターネットの発達につれてそれらのプログラムがアノニマスFTPによって公開されるようになった．そのあたりの事情は人工知能学会誌記事「人工知能用言語 Lisp の今と将来」（2009年，拙共著）に詳しいが，いずれにしてもその結果，出版された多くの人工知能の教科書で， Lispによるプログラムコードが掲載された．1980年は産業界における第2次AIブームを目前にしていたが，AIコミュニティが商業化直前のソフトウェアの著作権についてあまりうるさくなかった時代のことである．そういうわけで，今日でも人工知能(GOFAI)のプログラムコードを何か参照しようとすると，圧倒的にLispで書かれたものが多く，実際に動くGOFAI人工知能システムを作ろうとすれば， Lispを学ぶのが一番手っ取り早く，便利なのである．

Lispが人工知能プログラミングに向いている理由は，上記のような歴史的理由だけではない．工業製品としてのプログラムを産出するプログラミングにおいては，最初に要求仕様を確定し，次にシステム仕様を定めてからプログラミングを行うが，人工知能研究ではそれとは異なり，研究者が色々なアイデアをプログラムで実現しようと，試行錯誤的にプログラミングを行う．これを**探索的プログラミング**と言うが，そこで計算機言語に要求される基本的な性能は，研究者の思考を妨げないこと，やろうと思うことは計算機上でなん

でも実現できるような，柔軟でかつ研究者の思考をパワーアップするようなものであることである．松岡正剛氏は言う．「高級言語（プログラミング言語）にはいろいろのものがある．[...] これらの言語の特徴は大きくは，プログラマーの愚かな行為を防ぐような言語と，プログラマーがやりたいことを何でもできるようにする言語とに分かれる．」まさに慧眼である．愚かなプログラマーでも馬鹿なことをさせないようにする言語がJavaであり，賢いプログラマーに何でもできるようにする言語がLispなのである．

これまで最も包括的な人工知能プログラミングの教科書であったParadigms of Artificial Intelligence Programming (頭文字をとって俗にPAIP，パイプと呼ばれる)の中で，著者であるPeter Norvigは「何がLispを他の言語と違うようにしたのか？」として，Lispの特徴を次のように挙げた．

- 組込みのリスト処理
- 自動メモリ管理
- 動的型付け
- 第1級の関数
- 一様な構文
- 対話的環境
- 拡張性
- 歴史

リストはLisp生来のデータ構造であり，レンガを組み合わせて作られる建物のように，Lispシステムはセルとポインタからなるリストを基礎にできている．人工知能分野では，データはグラフ構造やツリー構造であることが多いが，これらの複雑なデータ構造もリストによって実現される．自動メモリ管理とはガーベジコレクションのことであるが，Lispは最初からガーベジコレクションありきで設計され，当初からプログラマーがメモリ管理という仕事にわずらわされることはなかった．また，今ではLispにも整数，浮動小数点数，配列，文字列，ハッシュなど，色々なデータ型が備えられているが，Lispでは型は変数につくのではなくデータにつく．そして変数にはどんな型のデータでもセットすることができる．Lispは関数型言語であるが，Lisp関数は関数に与える引数になり得る．関数を引数とする関数は高階関数と呼ばれるが，それによりプログラムを抽象化してコンパクトで理解しやすいプログラムコードにすることができる．Lispの構文はものすごく単純である．最低限必要とする構文はリストであり，その意味するところは最初の要素が関数でその他はその関数の引数になるということだけである．また，関数本体のコードもリスト構造をしており，Lispのデータもリスト構造なので，プログラムがプログラムを生成して実行するというコードが簡単に書ける．Lispのマクロ機能は他の言語には見られないほど高度である．Lispのプログラムは関数の集合であるが，そのプログラミングスタイルは，対話的環境の中で比較的短いコードを関数として定義し，それを実行・デバッグするというサイクルを早く回していくというものであった．要するにこれらの特徴すべてが，プログラマーに自由を保証し，計画なしの試行錯誤をプログラミングにおいて可能にしたのである．絵を書くときキャンバスにどのように絵を書くか計画的で学生時代に比較的よい成績を収める学生は，その後伸びないと聞くが，芸術における真の創造性は試行錯誤の中からしか生まれないとすれば，同様に，Lispはプログラミングにおいて**創造的プログラミング**を最大限可能にする言語なのである．

誰がこの本を読むのか？

この書籍は，実際にこれからシンボリックな人工知能のプログラミング方法を習得しようとする人を対象に書かれている．人工知能は言うに及ばず，計算機言語やLispによるプログラミングの経験を何ら前提とせず，Common Lispプログラミングの初歩から初めて，一人前のAI(GOFAI)プログラマーと言えるようになることを目標としている．ただし，次章以降の内容を読み進めるにあたって，どうしてもCommon Lispの処理系を手元に用意することが必要であり，PCやLinuxについてパッケージプログラムをダウンロードしてインストールしたり，ファイルを操作したりする程度のスキルは前提としている．Lispの発明者であるMcCarthyも言うように，Lispのプログラミングは2週間ほどで学べるが，既存のimperativeな計算機言語に慣れた人ほど，かえってそれが邪魔になって習得に時間がかかる傾向がある．CやC++にどっぷりつかった人は一度頭をまっさらにして本書を読み進めるよう注意されたい．

なぜこの本が必要なのか？

現在市販されていて容易に入手可能なLispによる人工知能プログラミングの教科書は，唯一Peter Norvigによる PAIP のみであるが，パッケージやCommon Lisp Object System (CLOS)などは使われていない．また，人工知能の教科書として最も有名なStuart J. Russell と Norvig による *Artificial Intelligence, A Modern Approach (AIMA)* には疑似コードがついており，AIMAホームページには疑似コードの主なものについてCommon Lisp版が掲載されているが，それも同様である．つまり，いずれもCommon Lispとは言っても *Common Lisp the Language* 第1版(*CLtL1*)にとどまっており，*Common Lisp the Language* 第2版(*CLtL2*)やANSI Common Lisp に準拠して今日AIプログラミングの教科書として右から左にすぐ通用するものは，英語まで含めても本書以外にはないという状況である．

どのCommon Lisp処理系を使うのか？

現在Common Lisp処理系として，複数の処理系が入手可能であるが，ここではAllegro Common Lisp(ACL)とSteel Bank Common Lisp(SBCL)を前提として話を進める．ACLは商用のCommon Lispであり，本格的なアプリ開発には商用版を購入する必要があるが，日本に販売代理店もあることから，大学・研究機関や企業で利用するには便利である．読者が本書籍の内容を学ぶためは，開発元のFranz社から無料のACL Expressをダウンロードして利用することができる．一方，SBCLはもともとフリーなオープンソースのCommon Lisp処理系であり，LinuxやPCに容易にインストールして利用することができる．どちらをインストールしたらよいか分からない，あるいはどちらでもよいという方には，ここではACL Expressをお勧めする．どちらかといえばACLは初心者にやさしく，SBCLはオープンソース・ソフトウェア(OSS)愛好者に適していると言えよう．この本では付録にてACLとSBCLの両方のインストール方法をガイドしているが，全体としては初心者を対象にACLに基づいて説明している．Emacs LispはCommon Lispではないので，本書籍のほとんどの部分で役に立たないことを注意しておく．

この本の構成

この本は上巻と下巻に分かれており，**この序章**も含めて Common Lispとそのプログラミングの説明を上巻に， Common Lispを用いたAIプログラミングの部分を下巻に収めた．目的別に入手しやすいように，上巻と下巻を別々に入手可能にしたが，本来は全体として一体のものであり，上巻で下巻を引用したり，逆に下巻から上巻を引用したりすることがあることを，あらかじめお断りしておく．

通常のプログラム言語では，言語機能や装備されているデータ型の説明から始まるのが通例であるが，本書では最低限の関数定義の書き方と再帰プログラミングから説明を始め， Common Lispの豊富な諸機能の説明は第12章以降に回した．その理由は， Common Lisp の言語機能のあれこれを知るよりも， Lispの本質を理解してもらうのが先決と考えるからである．Lispの Lispたる特徴を理解するのは大変であるが，それさえ掴まえてしまえば通常の言語機能などは言語仕様書などをたよりに自分で勉強することができる．Lispプログラミングを身につけるには，最初は S式と defun さえ分かればよいのである．

第1章は Lispの導入であり，付録の記述に従って Common Lispをインストールして立ち上げたところから始まる．この章ははじめて Lispにさわる人たちを対象としているので， Emacs Lispも含めて Lispに触ったことのある人は飛ばしてもかまわない．Lispの特徴である対話型の入出力と括弧の多い S 式に慣れてもらうのが目的である．

第2章では関数定義の書式を学んで， Common Lispに用意されているいくつかの述語と条件式の書き方を知り，ごく簡単なプログラミングに必要な最低限の知識を知ってもらう．実施例や問題の与え方は，順番に読んでいけば少しずつ無理なく進んでいけるように，配置してある．Common Lispのプログラミング方法は知っているという方は飛ばしてもらってもかまわない．

第3章では丸々1章分を費やして，再帰プログラムの書き方を学ぶ．再帰プログラミングこそが AIプログラミングに必須の技術であり， Lispに特徴的なところであるにもかかわらず，最近の Lispの教科書はこれについてほとんど記述していない．たとえば Peter Seibel の本はこれはこれで Common Lispの教科書としては出色のものではあるが，なんと再帰についての記述がない．なぜ再帰かといえば，それが人工知能実現に必須の技術だからなのであるが，この章の問題を順にきちんと追っていけば，いままで再帰プログラミングで挫折した人も必ずクリアできるはずである．また，今では Lisp処理系も Lispでプログラムされているが，例題を見て，問題を解いていくことで，個々の Lisp関数の Lispにおける実装も類推がつくようになるはずである．Lispを理解するためにも，再帰プログラミングは重要である．なお，この章における問題のほとんどは， *The Little LISPer* 第1版と WinstonによるLISPに記載のものである．

第4章は Lispの変数束縛についてである．再帰プログラミングと並んで変数束縛も，もう一つの Lispの本質である．昔の Lisp の変数束縛方式は現在なら動的束縛と呼ばれるものであり，そのために非常にわざとらしくプログラムすると， Lispをコンパイルしないで実行した場合とコンパイルして実行した場合で結果が異なるコードを書くことができて，これを FUNARG問題という名前で呼んでいた．Common Lispでは他のプログラミング言語と同様に静的束縛が基本となって，この FUNARG 問題が一掃された．これは大きな進歩であったが，動的変数も特殊変数と宣言することで作ることができて，これが非常に有効に働く場合もあ

り，両者をきちんと使い分けることがLispプログラミングでは大事である．この章では，これらの変数の異なりを理解し，特殊変数，大域変数なるものを学んで，正しく使えるようにする．ここまでがCommon Lisp初級プログラミングと言っていいかもしれない．

第5章では，それまで慎重に避けてきた副作用を伴う諸関数について学ぶ．第4章まではリストとはこういったものという紙面の字面上の理解でよかったが，ここではconsセルとポインタのモデルを学んで，そのモデル上で作用する関数の使い方を学ぶ． Common Lisp は人工知能言語のアセンブラ言語とでもいうような特徴もあり，変数の参照透明性を確保した関数プログラミングもできれば，計算性能を重視した副作用を含むプログラミングもできる．両方を学ぶことで，はじめて，バグのない効率的なコーディングができるようになる．

第6章は関数を引数に取る高階関数とMAP関数についてである．今日では多くの計算機言語がラムダ式を扱えるようになってきているが， Lispこそハスケル B. カリーとアロンゾ・チャーチによるラムダ計算に触発されて作られた関数型言語であり，プログラミングとCommon Lisp実装の至る所で関数型言語の利点が発揮されている．また，ここでは改めて複雑なCommon Lispのラムダパラメータの書き方について学ぶ．この章によって自分でも高階関数を書くことができるようになる．

第7章では理解の難しい変数のエクステントという概念を学ぶ．そのために関数閉包（クロージャ）とは何かを説明する．この章は第4章の続きと考えてもらってよいが，比較的理論的な話であるため，第4章とは分離してここに置いた．

第8章はいよいよマクロである．Ruby, Python, C#など多くの言語がLispの先進的なプログラミング機能を取り入れて発展してきたが，いまもってLispのみに残されている機能がLispのマクロである．これはCなどでいうマクロとは機能的に全く別物である．一部の人々からは嫌われている括弧のS式であるが，データとプログラムの表現を区別しないS式だからこそLispのマクロはLispのみが実装できる機能なのである．マクロは再帰プログラムを書けるようになった初級プログラマーが次にチャレンジしたいと思うものであり，第6章の高階関数とこの章のマクロを身につけて，立派な中級Lisperと言える．

第9章はコンパイラについて学ぶ．と言ってもコンパイル結果として生まれるコードには立ち入らず，コンパイル時と実行時が分離されているCommon Lispの特徴を理解して， **eval-when**を使えるようになるのが目的である．

第10章は末尾再帰と継続とコルーチンについて学ぶ．今では特にWeb方面でSmalltalkやRubyやJava言語によっても継続のテクニックが用いられているようだが，継続はGuy Steele, Jr.がLispによってActor理論を理解しようとするなかで偶然発見されたものであり，すべての計算機言語のアイデアをはぐくむLispのパワーを見せつける章である．

第11章ではCommon Lispを使って， SchemeのREPL (Read-Eval-Print-Loop)を実装する．これにより，関数とマクロの違いや静的変数束縛と大域変数など，それまでの章で説明してきたことがLispシステムの実装

を通じてより正確に理解されるであろう．ここではPAIPの範囲を超えてSchemeにおける動的変数束縛についても紹介している．プログラムコードのほんのわずかな実装の違いで静的にも動的にもなることを実感してほしい．

第12章では，配列，文字列，シーケンスの詳細な使い方について触れる．細かい話がどうしても多くなってしまうが，Common Lispらしいところではこれらのデータの型階層について述べており，最初は型階層に注目してざっと読むだけでもよいであろう．

第13章では，Common Lispのデータ型についてさらに学ぶ．Common Lispでは自分で特有のデータ型も定義できる．deftypeの使い方を説明するが，その部分は類書にはないものかも知れない．

第14章は構造体である．Common Lispの構造体は前章で述べたLispの型システムと統合されており，新しく構造体を作ってincludeを使えばCLOSと同様に型階層を形成することができ，しかもそれは組み込みの型と区別なく扱うことができる．

第15章はCLOSである．構造体の機能に加えて，汎化された関数概念としての総称関数とその特殊化されたものとしてのメソッドを備えたものが，Common LispのオブジェクトシステムCLOSである．構造体と同様にCLOSオブジェクトもCommon Lispの型システムに統合されている．この章ではCLOSの標準機能について学び，さらにANSI標準ではないが主要なCommon Lisp処理系には備えられているメタオブジェクト・プロトコル（MOP）について簡単に学ぶ．CLOSについてのよい教科書が英語でも手に入らない現在では，この章が価値あると判断される人も大勢いるであろう．

第16章はCommon Lisp内のもう一つの言語と呼ばれるほど，その高機能さと複雑さを有するFormat文についてである．ここではその仕様を網羅的に記述するというよりも，実用上の観点から諸機能の使い方について説明を試みた．一部からの要望により，プリティ・プリントの書き方についてもここで実用的に記載した．

第17章はLoop文について説明するが，本書を自己充足的にするために記載するものであり，Peter Seibelの22章を勉強した人には余分なところであろう．

第18章として，一部からの要望により，Common Lispのコンディションシステムについて解説する．古くからあったエラーや警告の機能は，今日ではより一般化されたコンディションシステムの一部としてCLOSによって構築されているが，正直に言ってコンディションシステムの使い方はいくら仕様書を読んでも分からない．なぜならばそれはCommon Lisp実装者向けに書いてあるからである．本章ではどう使うかという観点から仕様書にはない内容を盛り込んだ．

第19章として，ここまでまとまった説明をしてこなかった，多値，ストリーム，ファイルの取り扱いなどについて述べる．

第20章で最後にパッケージやモジュール機能の使い方を学ぶ．ASDF3についても簡単に触れる．Quicklispが普及してきているので，それに依存して使う分にはASDFを学ぶ必要性は薄れてきてはいるが，SBCLで自前のシステムをQuicklispで扱ったり配布しようとするとASDFを使うことになる．この部分もなかなか類書に記載がないところである．

ここまでがCommon Lispの部分であるが，その後ろに付録としてAllegro Common Lisp free Express とSBCL のダウンロードとインストール方法を付けた．手持ちのLisp処理系がない場合にはこちらの付録を参考にして，ACL あるいは SBCL どちらかの処理系をインストールすることになる．また，よりアドバンストな話になるが，reflectiveなLispを開発したBrian Smithの提示したLispの概念モデルを紹介して，これまでどのCommon Lisp教科書でも触れられていなかった，マクロ文字とreadtableのカスタマイズ方法についても紹介する．

最近になって ubuntu における ABCL や CLISP のインストールが簡単だということが判明したので，これらについても付４と５に追加情報として記載した．興味のある方は参考にされたい．

Common Lisp にはSchemeのような継続の機能は用意されていないが，第10章で述べるように，末尾再帰関数をコンパイルすることで Common Lisp でも継続を実装することができる．ポール・グレアムはマクロとLisp関数を一体化させてユーザの目からは継続を見えなくした機能を開発した．ポール・グレアムによる**継続関数**の実装について付６に記載した．第10章を学習した後，本格的に継続プログラミングをしたいという方は是非この付６に進んでいただきたい．

上巻を読み進める方法として，二通りの道がある．一つは付録にしたがってACLかSBCLをインストールしたあと，章の順番に従って読み進める方法である．この方法は言ってみればアイガー北壁に登るようなもので，一般的な言語仕様に類するものは置いておいて，他言語にはないCommon Lispの理解に進む方法であり，第11章の「Lispで作るScheme」がその頂点となる．途中まで道のりは険しいけれど，他言語と異なるCommon Lispの特徴を早く掴みたいというベテランハッカーにはお勧めである．

もう一つの方法は，付録にしたがってACLかSBCLをインストールしたあと，第1章から第6章の高階関数まで進み，次はCommon Lisp特有の難しいところを飛ばして第12章から20章まで進むという方法である．こちらの読み方は，通常のプログラム言語の勉強方法に近い．Lisp に慣れてから，残した章を読むというのも，早くCommon Lispのプログラミングをしたいという人にとってはいい方法であろう．こちらは初心者向きであるが，アイガー北壁に登りかけたけど，大変そうだから後回しにするという方法も正しい方法である．ただし，下巻の人工知能のプログラムでは，継続などのCommon Lisp特有のプログラミング手法を利用するものもあることを注意しておく．

第21章から32章までが下巻の人工知能プログラミングの部分である．その全体の構成はほぼ歴史的順序に並んでいる．全体としては *Paradigms of Artificial Intelligence Programming* (PAIP)を底本にしつつも，

PAIPにはないフレームシステム(第25章)や事例ベース推論(第27章)はSchank一派のものをベースにしている．

第21章は人工知能プログラミング環境と称して， PAIP と *Artificial Intelligence: Modern Approach*(AIMA) の Lisp環境を紹介し，次章以降で利用する各種ユーティリティ諸関数を説明する．ただし，以下の各章の人工知能プログラムでは， 各章の独立性を高めるために，PAIPやAIMAの諸関数のうち，必須のものだけを取り出して置いていることを注意しておく．

第22章は人工知能ではない俗に人工無脳と言われる Elizaプログラムである．Web上のボットや Siriのような対話機能を実装したいと思っている人には，そのプログラムコードは大いに参考になるであろう．

第23章は初期の人工知能システムである一般問題解決器（GPS）の紹介である．ここでは GPSの応用例として著者自身になる宇宙ロボットにおける計画実行システムを紹介している．

第24章は探索問題である．筆者自身も人工知能に取り組みだした当初思ったことであるが， AI＝探索？と思われるほど，初期のAIプログラムの基本技術は要するに探索であった．ここではウィンストンによる動物のエキスパートシステムを例題に，前向き推論，後向き推論を学び，PAIPオリジナルプログラムを少し変形して，深さ優先探索，幅優先探索，A*アルゴリズムなど，探索の基本について学ぶ．

第25章でフレームシステムを紹介する．過去にフレームシステムとして多くのシステムが開発されてきたが，ここでは Charniak, Riesbeck, McDermott によるXRLを紹介する．

第26章はルールベースシステムである．ルールベースシステムも過去多くのシステムが開発されてきたが，ここでは確信度を導入した典型的なルールベースシステム MYCIN を紹介する．

第27章では Riesbeck と Schank による *Inside Case-based Reasoning* の記述に基づいて事例ベース推論について説明する．現在の Common Lispで動く事例ベース推論システムに関する教科書はここで紹介するものだけであろう．

当初章立て計画時には，述語論理で一章を持つとして，命題論理はその中に含めるつもりであった．しかし論理変項がない命題論理においても充足性の問題が議論されることから命題論理の説明のところで議論した方がよいと考え，命題論理だけで一章立てることとした．**第28章**では命題論理について説明したのち，RussellとNorvigによるArtificial Intelligence -- A Modern Approach (AIMA)のコードを参考に，命題論理のプログラムを紹介する．

第29章では述語論理などこれ以降の章にて基本となる単一化（ユニフィケーション）について一章を起こして説明する．

第30章では一階述語論理について説明する．述語論理プログラミングというのは人工知能の歴史の中でも比較的新しい分野である．そのためか，PAIPにはちゃんとした述語論理のプログラム的記述がなく，いき

なりPrologに走っている．この章では AIMAのプログラムからその内容を借りて述語論理の説明をする．述語論理となると，PAIPには述語論理のコードがなく、AIMAにも不十分さが見られるところであるが、極力努力して記述したい．

第31章はプラニングである．PAIPにはプラニングの章はないし， AIMAでは章立てて説明はあるが，掲載された疑似コードに相当する実コードはどこにもない．ここではプラニングに関する優れた教科書 Ghallab, Nau, Traverso の*Automated Planning Theory and Practice*の定式化に沿った筆者のプログラミング例を紹介する．

第32章は Lispで作る Prologである．この章の内容はほぼ PAIPのそれと同じである．

最後に**終章**において，本書籍と人工知能プログラミングについて振り返って概観し，残されているもの，現在望まれている事項やレベルとの差異を明らかにして，将来への課題とそれに対する個人的な見通しと希望も述べることにする．

PAIPでは14.10節で「表現力の問題の解決法」と題して，1) 高階述語に対処する方法，2) フレームに基づいた構文の導入，3) 多重世界のサポート，としてProlog の拡張を実施している．実際に行われているのは，弁別木によるデータベース実装と，局所的手続きをPrologに追加することによる前向き推論機能の導入である．これらは，それ自身で学ぶ価値があるし，扱われている「表現量の問題」は Semantic Web やオントロジーの問題に直結するものであることから，ここでその内容を**付９知識表現言語 krep** として紹介することとした．将来，Semantic Web 用言語に進むうえで大いに助けとなるであろう．

最近のAIブームとCommon Lisp

2015年に本書籍の出版を思い立って執筆を開始したが，ほぼ同時に，日本でも深層学習（ディープラーニング）による第三次AIブームが起こり始めた．このブーム到来の最大のきっかけとなったのはGoogleがディープラーニングによって大量の画像の中から猫を自動的に認識したことにあるが，執筆中にも益々この動きが熱くなって，官民あげてのAIブームとなった．このときはAIといえばそれはすべて深層学習のことであった．ブームはいつかしぼむからブームと呼ばれるのであるが，その深層学習によるAIブームがおさまらないうちに，今度は2022年11月に米国のOpenAIがChatGPTを無料で公開したのをきっかけに，大規模言語モデル (LLM) に基づくAIが大流行となり，今やAIといえばChatGPTやGeminiをどう使うかというという話になった．そしてとうとう2024年のノーベル物理学賞にはプリンストン大学のジョン・ホップフィールド教授とトロント大学のジェフリー・ヒントン教授が選ばれ，ノーベル化学賞にはGoogle DeepMind社のデミス・ハサビス氏とジョンジャンパー氏が選ばれた．つまり，ニューラルネット研究者がノーベル物理学賞とノーベル化学賞をとったのである．AIエージェントでなくて人間でよかったと愚痴とも冗談とも言えないような感想を述べざるをえない．

AIを実現技術として大きく分ければシンボリックAIとニューラルネットAIに分かれる．シンボリックAIというのは本書でも扱う Lisp とか Prolog といったAIプログラム言語をつかって探索とか推論といった機能を

実現するもので，今ではGood Old fashioned AI(GOFAI)と多少の揶揄を込めて呼ばれる．一方ニューラルネットAIは人の神経細胞の結合や働きを計算機上で実現して，AIに迫ろうとするもので，当然のことながらGOFAIのほうが少ない計算機資源で目的を実現できるが，すべてを人手でプログラムしなければならない．それにたいしてニューラルネットAIではプログラムするのではなく，データから機械に学習させてその結果として目的を実現する．だからまずは学習のための膨大なデータと膨大な計算資源が必要になる．60～80年代では計算機資源不足のためにニューラルネットAIは使い物にならなかったが，近年ではそれがニューラルネットを層状に何層も積み重ねたディープラーニングが使えるようになり，現在に至っている．

大規模言語モデルというのは，ウェブなどの言語情報を深層学習で学習したもののことであるが，昔は事実上無限と思われていたウェブをAIが学習しつくして，AIの学習データが足らなくなるということが真剣に心配されるようになってきた．４年間で五千億ドル（約78兆円）を投資するというスターゲート・プロジェクトもその最初の15兆円の投資はAIのためのデータセンターの建設のためなのである．今やAI研究開発は一研究者や一国家の手を離れた壮大なビッグプロジェクトなのである．ひるがえって，本書で扱うGOFAIは中身をブラックボックスとせず，手作りのAIを求める人にむけた，貧者のためのAIというべきものかも知れない．

本書では第1次AIブームの**探索**，第2次AIブームの**知識**については扱っているが，ディープラーニングについては何も教えない．ベイズ理論などの確率的機械学習についても何も教えない．その理由は，筆者がそれらの新しいAI技術については執筆に足るものを何も持ち合わせていないということや，本書の底本となるPAIPにも記述がないということがある．しかしそれはCommon Lispがこれらの新しい技術に向いていないということを意味してはいない．現在ディープラーニングでは大量データの処理速度を求めてGPUを用いた専用ツールがもてはやされる一方で，インタープリタしかなくて計算速度に問題があるPythonベースのツールが提供されるというちぐはぐな状況が生まれている．ディープラーニングにLispが向いていないという証拠はどこにもなく，逆にLispベースのディープラーニング・ツールがPythonベースのものより早いという結果も生まれている．要するに，現在ディープラーニングとCommon Lisp両方に精通した人材がいなくて，まだCommon Lisp+ディープラーニングの技術が蓄積されていないというだけのことである．この極端な状況は早晩改善されるであろうし，本書に取り組んだ読者の中からそういう研究者が出てくることを大いに期待している．ディープラーニングでシンボルグラウンディング問題に取り組むならば，その特徴からLispで取り組むのが最適であるというのは，分かる人には即座に了解されるところである．

本書ホームページとGitHubページ

電子書籍が紙の書籍よりも優れている点には，コードをタイプ打ちすることなく，簡単にコピー&ペーストできるということがある．とは言え，本書をKindleのようなデバイスで読みつつ，パソコンに向かっている読者もいるであろう．筆者は初心者が計算機言語を学ぶにはコピペよりも手打ちしたほうがよいと信じているが，それでも下巻の人工知能プログラミングではさすがにソースコードが提供された方が何かと便利である．また上巻でも筆者の意には反するが，ソースコードを置いてほしいという読者からの要望が出てくる

ことは目に見えている．そのため，各章に記載されるコードを置いたGitHubページを用意した．希望者は自由にダウンロードして利用されたい．

- 上巻および下巻のGitHubページ – http://github.com/cl-aip

また，本書について更新情報などを提供するホームページは，以下のとおりである．PayPalで本書を購入するなどしてこちらがメールアドレスを把握している読者には，メールアドレスによるログイン可能とする．こちらのページは出版元であるオントロノミー合同会社が管理する．書籍をPayPalで購入したが本ホームページにアクセスできないという方は，koide於ontononomy.co.jpまで連絡されたい．なお，本の購入サイトは http://ontolonomy.co.jp/books/ である．

- 本書ホームページ – http://cl-aip.website/

また，本書を中心としたユーザ交流の場としてのホームページも用意したが， 5年間の間何の活用のアイデアも生まれなかったので現在は停止している．~~ユーザにおかれては，是非このホームページの使い道まで含めて，活用方法を探っていただきたい．こちらについて，質問，提案などなんでも，info@cl-aip.orgまで寄せられたい．~~

- Common Lisp と 人工知能プログラミングのコミュニティサイト – ~~http://www.cl-aip.org/wiki/~~

第1章 Lispの基本

> LISPは主に記号処理や人工知能研究に用いられてきた．これは一つには記号処理がしやすくなるからであり，一つにはAIのプログラマが，離陸前に飛行計画を書類で提出したがらない操縦士の様に，ちゃんとしつけられておらず怠け者になりがちだからである ― LISPの対話的な構造によって計画を立てずともうまくやれるからである．
>
> [Charniak/Riesbeck/McDermott, 1980]
>
> （人工知能プログラミング，白井・片桐訳，1986）

1.1 リストを入力してみよう

ACLあるいはSBCLを立ち上げて，最初に次のように見えるLisp対話環境をLisp Listenerとかトップレベルと呼ぶ．`cl-user(1)`あるいは`CL-USER(1)`という入力プロンプトが見えていて，ユーザからの入力待ちになっているはずである．まずはそこに次のようにタイプ入力してみよう．

```
cl-user(1): (+ 3 4)
7
cl-user(2): (+ 3 (* 2 4))
11
```

ここで`cl-user(1)`とか`cl-user(2)`というのは，Lispシステム側から出力された入力用プロンプトであり，`cl-user`は現在カレントパッケージ（第20章で詳述する）がcommon-lisp-userであることを示している．ACLのmlisp (modern lisp)を使えば，この通りであるが，ACLのalisp (ANSI lisp)やSBCLであれば，プロンプトは大文字の`CL-USER`であろうし，ACLのIDE付きのバージョンであれば，`cg-user`や`CG-USER`であろう．

Lispの対話型機能

今は対話型計算機言語であるPythonやRubyが普及しているので，改めてLispの対話機能と言ってもめずらしくもないかもしれないが，もともとPythonやRubyもLispの影響を強く受けて開発されたものである．Lispは最初から対話型を基本とするプログラム言語として生まれ，当時のFORTLANのバッチ・コンパイル型とは異なるプログラム開発の性格を有していた．昔はこのプログラミング・スタイルの違いも対話型計算機言語に慣れていないFORTRANやCのプログラマーには戸惑いの原因となっていたかもしれない．Lispが人工知能研究に用いられた理由の一つは，あらかじめ仕様を決められず，アイデアを試行錯誤的にプログラムとして実現する探索的プログラミングにLispの対話的機能が向いていたということがある．

Lispは関数型言語である

これも前述と同様に，今でこそLispのほかにHaskellやF#も関数型言語なので，改めてLispは関数型言語と強くいう必要もないかもしれないが，ラムダ式の存在から分かるように，Lispこそ元祖関数型言語であった．すなわち，プログラムは関数呼び出しを基本として作られ，入れ子構造の関数呼び出し形式によって，関数呼び出しで返される値が，次の関数呼び出しの引数として与えられる．たとえば，次のとおりである．

```
cl-user(2): (+ 3 (* 2 2))
7
```

Lispの計算は括弧の内側から外側に向かって行われるのだ.

Lispでは，プログラムを作ることすなわち関数を定義することである．上巻を終えたころにはそのプログラミング・スタイルも習得されていることであろう.

Lispの関数呼び出しは前置形式(prefix notation)である

上記入力例から分かるように，Lispでは加減乗除の演算さえも関数実行であり，しかもその呼び出し形式も他の言語とは随分と異なっている．すなわち，いついかなる場合も，関数実行の場合には丸括弧でくくられたリストの1番目が関数の役割を果たし，その他はその関数を実行する際の実引数である．算術演算さえ関数呼び出しであるため，その引数を括弧で括らないといけない．だから，演算子の優先順位を気にしなければならないということがない.

前置形式でなければ

《関数名》(《引数1》 《引数2》 ...)

となるところが **Lispでは**

(《関数名》 《引数1》 《引数2》 ...)

となる．また，コマンドも関数呼び出しである．たとえば，Pythonでは「`exit()`」や「`quit()`」であるのが，Lispでは「`(exit)`」や「`(quit)`」となる.

リスト，アトム，そしてS式

丸括弧でくくられた字句形式を**リスト**と言う．一方，括弧のない字句形式を**アトム**と言う．Lispではリストやアトムは括弧とスペースや改行で区切られる．上記例では `+`，`3`，`4`などがアトムであり，`(+ 3 4)`はリストである．リストとアトムのことをまとめて **S式**(symbolic expression)と言う．リスト以外の字句の並びはアトムである．当然のことながら，リストのリストもリストである．またこれも当然のことながら，開き括弧の数と閉じ括弧の数は整合的に対応しなければならない.

問題 1.1 次のものは，アトム，リスト，誤った表記のいずれかである．区別せよ.

1. `atom`
2. `(this is an atom)`
3. `((a b) (c d))`
4. `3`
5. `(3)`
6. `(list 3)`
7. `)(`
8. `((()))`
9. `(() ())`
10. `(())`

11. ())(
12. ((abc

シンボル

正しい表記であることを前提にして，数アトム以外のアトムはシンボルと呼ばれる(と今のところは言っておこう)．Lispのシンボルは通常の計算機言語では関数名や変数名に相当するもののように見えるかもしれないが，他の言語とは異なってその文字構成にほとんど制限がない．字句のならびがあったとき見た目で明らかに数以外のものはシンボルと思ってもらってよい．以下のいずれもリスプシンボルである．

- symbol
- 0symbol
- *symbol*
- +symbol+
- $/=
- +$
- b^2-4*a*c
- 1+
- 1-

以下のものはシンボルではなく，数である．

- +1
- -1
- 2/3

1.2 書式を評価する

S式を Lispに与えて解釈させ，その値を得ることを**評価**(evaluation)という．S式が評価されるとき，そのS式は**書式**(form)と呼ばれる．「評価する」とは書式の値を計算することである．では書式を評価したときに得られる値は何かと言えば，数アトムの場合はその値であるけれども，一般にシンボルには最初は値が設定されておらず(unbound)，一部のシンボルのみが Common Lisp処理系組み込みの値を持っている．具体的にはたとえば，次のようになっている．

```
cl-user(3): 3
3
cl-user(4): -3
-3
cl-user(5): 3.14
3.14
cl-user(6): pi
3.141592653589793d0
cl-user(7): t
t
cl-user(8): nil
nil
cl-user(9): most-positive-fixnum
1152921504606846975
cl-user(10): most-negative-fixnum
-1152921504606846976
cl-user(11):
```

- 数アトムの値はその値である．
- pi の値は倍精度浮動小数点の3.141592653589793である．
- t の値は t である．
- nil の値は nil である．
- most-positive-fixnum の値は1152921504606846975である．
- most-negative-fixnum の値は−1152921504606846976である．

ここで，piや most-positive-fixnum，most-negative-fixnumの値は32bitマシンか64bitマシンかで異なるし，Common Lisp処理系によっても異なる可能性もある．

浮動小数点演算では，数値の1に対して加算してそれより小さい数では区別できなくなる最大の数がある．これを**イプシロン**と呼ぶがCommon Lispではこれが定義してある．たとえば次のようである．

```
cl-user(12): short-float-epsilon
5.960465e-8
cl-user(13): single-float-epsilon
5.960465e-8
cl-user(14): double-float-epsilon
1.1102230246251568d-16
cl-user(15): long-float-epsilon
1.1102230246251568d-16
cl-user(16): (not (= (float 1 short-float-epsilon)
                     (+ (float 1 short-float-epsilon)
                        short-float-epsilon)))
t
cl-user(17): (not (= (float 1 short-float-epsilon)
                     (+ (float 1 short-float-epsilon)
                        (* 0.5 short-float-epsilon))))
nil
```

書式が関数形式の場合には，それを評価すると関数の実行結果が得られる．

```
cl-user(18): (max 2 3 4)
4
cl-user(19): (min 2 3 4)
2
cl-user(20): (/ (+ 3 1) (- 3 1))
2
cl-user(21): (sin pi)
1.2246467991473532d-16
cl-user(22): (cos pi)
-1.0d0
```

問題1.2 次のS式を評価したときの値を予測せよ．

1. (* (+ 3 1) (- 3 1))
2. (* (max 3 4 5) (min 3 4 5))
3. (min (max 3 1 4) (max 2 7 1))

quoteは評価を抑制する

Lispの対話機能でLisp Listenerに書式を与えると，それはその場で自動的に一回評価されて結果が返されるが，次のようにクォートして評価を抑制することができる．

```
cl-user(16): '(cos pi)
(cos pi)
cl-user(17): (quote (cos pi))
(cos pi)
cl-user(18): '(this is quoted)
(this is quoted)
cl-user(19): (quote (quoted))
(quoted)
cl-user(20): 'quote
quote
cl-user(21): (quote quote)
quote
cl-user(22): '3.14
3.14
cl-user(23): 'pi
pi
```

ここでシンボルやリストの直前に置いたシングルクォート記号はLispにとっては書式をquoteで括ることと全く同一で，いわゆる人にとって見やすくするためのシンタックス・シュガーである．quote書式を評価して返される値はそのクォートされたS式そのものである．

```
(quote 《書式》) ⇒ 《書式》
```

1.3 carとcdrはリストを分解する

リストはLispの基本構造であるが，関数carと cdrまたはfirstとrestは，リストをその要素の1番目と2番目以降のリストに分ける．

```
cl-user(24): (first '(lisp is a language for AI))
lisp
cl-user(25): (car '(lisp is a language for AI))
lisp
cl-user(26): (rest '(lisp is a language for AI))
(is a language for AI)
cl-user(27): (cdr '(lisp is a language for AI))
(is a language for AI)
cl-user(28): (first '((head) and tail))
(head)
cl-user(29): (rest '((head) and tail))
(and tail)
```

carとfirst，cdrとrestに何の違いもない．carとcdrは古くからある言葉であり，古いLisperはこちらを平気で使う．一時期，carとcdrはやめてfirstとrestにしようと考える人たちがいたが，結局そんなことはほとんど意味がなく，carとcdrは今も使われている．firstだけでなく，second，third，... とtenthまである．一方，cxxxxrと4個までxの部位にaかdを書くことができて，たとえば，caaddrとは(car (car (cdr (cdr ...))))の意味である．

問題 1.3 次の書式の値はどうなるか予測せよ.

1. `(first '(a b) (c d))`
2. `(first (rest '(a b) (c d)))`
3. `(rest (first '(a b) (c d)))`
4. `(rest (first (rest '(a b) (c d))))`
5. `(first (rest (first '(a b) (c d))))`

問題 1.4 次の書式を評価して，なぜエラーとなるのかを説明せよ．またエラーを起こさずにbを取り出すにはどうすればよいか答えよ.

1. `(car (cdr (a b c)))`

エラーが起きると，IDEなどのウィンドウ環境で実行中のときはブレークウィンドウが開くので，その場合には「Cancel」などでそのウィンドウを閉じればよい．IDEではない環境ではリスタートアクションを聞かれる．その場合にはACLでは"Return to Top Level"を選択して戻ればよい．その具体的な指示の仕方は処理系によって少しづつ異なる．付1.4(ACL)や付2.5(SBCL)を参照されたい.

次の二つの問題は，Lispの括弧を読むことと，取り扱いに慣れるためのものである．初心者は必ずチャレンジしてもらいたい.

問題 1.5 次の書式を評価した結果を予測せよ.

1. `(car '(cdr (a b c)))`
2. `(car (cdr (car (cdr '((a b) (c d) (e f))))))`
3. `(car (car (cdr (cdr '((a b) (c d) (e f))))))`
4. `(car (car (cdr '(cdr ((a b) (c d) (e f))))))`
5. `(car (cdr '(cdr (cdr ((a b) (c d) (e f))))))`
6. `(car '(car (cdr (cdr ((a b) (c d) (e f))))))`
7. `'(car (car (cdr (cdr ((a b) (c d) (e f))))))`

問題 1.6 次のリストから pearを取り出すような carと cdrのつらなりの書式を書け.

1. `(apple orange pear grapefruit)`
2. `((apple orange) (pear grapefruit))`
3. `(((apple) (orange) (pear) (grapefruit)))`
4. `(apple (orange) ((pear)) (((grapefruit))))`
5. `((((apple))) ((orange)) (pear) grapefruit)`
6. `((((apple) orange) pear) grapefruit)`

1.4 シンボルは値を持つことができる

数アトムの値はその数そのものであった．シンボルアトムの値については，これまでの例では，`t` と `nil` と`pi` と`most-positive-fixnum`, `most-negative-fixnum`が出てきて，tの値はt，nilの値はnil，piの値は円周率πらしき値， `most-positive-fixnum`と`most-negative-fixnum` の値はとても大きな正負の整数であった．これらのシステムに組み込みのシンボルには，すでに決まった値が与えられているが，**それ以外のシンボルアトムにはプログラマーが自由に値をセットすることができる**.

次の例を見てほしい.

```
cl-user(30): foo
Error: Attempt to take the value of the unbound variable foo'.
  [condition type: unbound-variable]

Restart actions (select using :continue):
 0: Try evaluating foo again.
 1: Set the symbol-value of foo and use its value.
 2: Use a value without setting foo.
 3: Return to Top Level (an "abort" restart).
 4: Abort entirely from this (lisp) process.
[1] cl-user(31): :continue 3
cl-user(32): (setq foo 'bar)
bar
cl-user(33): foo
bar
cl-user(34):
```

最初アトム fooには値がセットされていないので，それを評価しようとしたときにエラーとなった．エラーの場合には（処理系ごとに多少の違いはあるが）対応を求められ，それに対して指示をする．上記例はACL mlispの場合であるが，「:continue 3」とタイプ入力してトップレベルに戻した経緯が上記例である．SBCLの場合には付2.5を参照されたい．次に，setqを用いてfooの値に barをセットしたあとでは，fooを評価して，その値である barが得られた．

setqの前の引数は評価されないが後ろの引数は評価される

setq はトップレベルではシンボルに値をセットするのに用いられるが，実はこれは関数ではない．Lispの処理系の中に組み込みで処理方法が記述されている書式が**特殊形式**と呼ばれるものであり，前出の quoteも特殊形式であるが，この setqも特殊形式であり，setq書式の前の引数は評価されないが，後ろの引数は評価される．(setq foo 'bar)において barにクォートがついているのは，fooの値に barそのものを与えたいからである．さもないと barが評価されてシステムはその値を fooの値にしようとする．

Lispを学びだして最初のとまどいがクォートを付けるべきなのかそれとも付けてはいけないのか，評価されるのかされないのか，ではなかろうか．ついついクォートを忘れてしまうというのは初心者に有りがちな誤りである．次の例を示す．

```
cl-user(34): (setq list '(a b c))
(a b c)
cl-user(35): list
(a b c)
cl-user(36): 'list
list
cl-user(37): (car list)
a
cl-user(38): (cdr list)
(b c)
```

今度はシンボルの値としてリストをセットした．

- シンボルアトムにはその値として，数でも，シンボルでも，リストでもセットできる．

- setq 書式の返す値はセットされた値である．
- setq の結果，setq 以前と以後では変化がある．このようなものを**副作用**(side effect)と言う．

Lispは関数型言語とは言うものの，完全な関数型ではない．（実際には入出力などは副作用にあたり，理論的に完全な関数型言語というのは現実にはありえない．）Lispはこの他にも様々な副作用を有しているが，極力副作用を無くして関数型言語の特長である変数の参照透明性を確保することもできれば，極端に副作用ゴリゴリのプログラミングをすることもできる．すべてはプログラマーのプログラミング・スタイル次第であるが，この書籍ではCommon Lispとしてまっとうな，中庸なプログラミング・スタイルを心がけるようにしている．

1.5 consはリストを合成する

carはリストの1番目の要素を取り出し，cdrはその残りの部分をリストとして返したが，丁度その逆を行うのが cons である．次の例を見られたい．

```
cl-user(39): (cons 'lisp '(is a language))
(lisp is a language)
cl-user(40): (setq list '(lisp is a language))
(lisp is a language)
cl-user(41): (cons (first list) (rest list))
(lisp is a language)
cl-user(42):
```

cons のほかにもリストを作る関数として list と append があるが，これら三者の微妙な違いに習熟しよう．以下の例を見られたい．

```
cl-user(42): (list 'lisp 'is 'a 'language)
(lisp is a language)
cl-user(43): (setq list (list 'lisp 'is 'a 'language))
(lisp is a language)
cl-user(44): list
(lisp is a language)
cl-user(45): (append list '(for AI))
(lisp is a language for AI)
cl-user(46): (append '(1 2) '(3) '() '() '(4) '())
(1 2 3 4)
```

- append は任意個のリストをとって，それらをつなぐ．
- list は任意個の引数を括弧でくくる．
- cons は 2 個の引数をとり，最初の引数を次の引数であるリストの先頭にいれたリストを作る．

Common Lispでは関数名と同じ名前のシンボルを変数にも使うことができる．実施例43では最初の `list` が変数として使われるシンボルで，2番目が関数としてのシンボルになっている．Schemeでは変数の名前空間と関数の名前空間が同一になっているが，Common Lispでは別々の空間になっているため，このようなことができて，同じシンボルを同時に変数にも関数にも使用することが可能である．ちなみに，シンボルからそこにセットされている関数を取り出すには symbol-function，値を取り出すには symbol-valueを使う．

問題 1.7 以下の問題は連続した実行と考えて，副作用を考慮して次の書式を順に予測せよ．

1. `(setq tools (list 'hammer 'screwdriver))`
2. `(cons 'pliers tools)`
3. `tools`
4. `(setq tools (cons 'pliers tools))`
5. `tools`
6. `(append '(saw wrench) tools)`
7. `tools`
8. `(setq tools (append '(saw wrench) tools))`
9. `tools`

空リストはnilと同じである

次の実施例を見られたい.

```
cl-user(47): '()
nil
cl-user(48): ()
nil
cl-user(49): (setq list '(lisp is a language))
(lisp is a language)
cl-user(50): (rest (rest (rest list)))
(language)
cl-user(51): (rest (rest (rest (rest list))))
nil
```

実施例47ではクォートつきで空リストを与えて，その評価結果として nil を得た．クォートは評価を抑制するから，返される値は「( )」のはずである．実は多くのLispで空リストとnilは全く同一である．ただし，SchemeやClojurなどでは異なっている．Common Lisp策定のときに空リストとnilを区別すべきかどうか喧々諤々の議論が行われ，結局のところ従来通りの区別なしということになった経緯がある．実施例48では nil が与えられたと同様であり， `nil` の値は nil であることから，nil が得られた．空リストが nil であるということは，最後の実施例51からも分かる.

nil を普通のアトムとして明示的に扱おうとすると何かと不都合なことが多い．次の例はどうだろうか.

```
cl-user(52): (car '(nil))
nil
cl-user(53): (car '())
nil
cl-user(54): (car '(( )))
nil
```

リストの中の nil は空リストと区別がつかない．コードを書くときには注意が必要である.

問題 1.8 次の書式の評価結果を予測せよ.

1. `(append '(a b c) '())`
2. `(list '(a b c) '())`
3. `(cons '(a b c) '())`

1.6 便利なリスト処理関数

リスト処理するのに，car，cdr，cons，append，list は頻繁に用いられる関数であるが，ここでその他のリスト関連の関数について述べる．

- last はリストの最後の要素を含む 1 要素のリストを返す．
- butlast はリストから last を除いたものを返す．

```
cl-user(55): (last '(a b c))
(c)
cl-user(56): (last '((a b) (c d)))
((c d))
cl-user(57): (butlast '(a b c))
(a b)
```

- nth は（0から数えて) n番目の要素を返す．
- nthcdr は（0から数えて) n番目の要素に始まるリストを返す．
- length はリストの要素数を返す．
- reverse はリストの要素の並びを逆順にして返す．

```
cl-user(58): (nth 2 '(a b c d))
c
cl-user(59): (nthcdr 2 '(a b c d))
(c d)
cl-user(60): (length '(a b c d))
4
cl-user(61): (length '((a b)))
1
cl-user(62): (reverse '(a b c d))
(d c b a)
cl-user(63): (reverse '((a b) (c d)))
((c d) (a b))
```

問題 1.9 次の書式の評価結果を予測せよ．

1. `(length '(plato socrates aristotle))`
2. `(length '((plato) (socrates) (aristotle)))`
3. `(length '((plato socrates aristotle)))`
4. `(reverse '(plato socrates aristotle))`
5. `(reverse '((plato) (socrates) (aristotle)))`
6. `(reverse '((plato socrates aristotle)))`

キーバリュー・ペア

簡単なキーバリュー・ペアを実装する仕組みとしてLispの連想リスト(association list)を用いることができる．以下は県庁所在地のデータベースである．

```
cl-user(3): (setq *db*
'((北海道 札幌市)
  (青森県 青森市)
  (岩手県 盛岡市)
  (宮城県 仙台市)
  (秋田県 秋田市)
  (山形県 山形市)
  (福島県 福島市)
  (茨城県 水戸市)
```

```
  (栃木県 宇都宮市)
  (群馬県 前橋市)
  (埼玉県 さいたま市)
  (千葉県 千葉市)
  (東京都 新宿区)
  (神奈川県 横浜市)
  (新潟県 新潟市)
  (富山県 富山市)
  (石川県 金沢市)
  (福井県 福井市)
  (山梨県 甲府市)
  (長野県 長野市)
  (岐阜県 岐阜市)
  (静岡県 静岡市)
  (愛知県 名古屋市)
  (三重県 津市)
  (滋賀県 大津市)
  (京都府 京都市)
  (大阪府 大阪市)
  (兵庫県 神戸市)
  (奈良県 奈良市)
  (和歌山県 和歌山市)
  (鳥取県 鳥取市)
  (島根県 松江市)
  (岡山県 岡山市)
  (広島県 広島市)
  (山口県 山口市)
  (徳島県 徳島市)
  (香川県 高松市)
  (愛媛県 松山市)
  (高知県 高知市)
  (福岡県 福岡市)
  (佐賀県 佐賀市)
  (長崎県 長崎市)
  (熊本県 熊本市)
  (大分県 大分市)
  (宮崎県 宮崎市)
  (鹿児島県 鹿児島市)
  (沖縄県 那覇市)
  ))
((岩手県 盛岡市)(宮城県 仙台市)(秋田県 秋田市)...)
cl-user(4): (assoc '神奈川県 *db*)
(神奈川県 横浜市)
cl-user(5): (cadr (assoc '神奈川県 *db*))
横浜市
```

連想リストでは，各要素は次のような構造をしている．

```
(《key》 《value1》 《value2》 . . . )
```

関数assoc は連想リストの《key》部を先頭から順にサーチして，それに eql の意味で等しい要素が見つかればそれを返す．したがって，上記で文字列を用いずにシンボルにしたことには意味がある． eql での比較では文字列は使えないからである．（詳細は2.2節で述べる．）

```
cl-user(7): (eql "横浜市" "横浜市")
nil
```

```
cl-user(8): (eql '横浜市 '横浜市)
t
```

もし文字列比較にしたいのなら，次のようにテスト関数として明示的に string=を指定する．loop については第17章，バッククォートについては8.2節で説明している．関数 string は，引数が文字列なら文字列を返し，シンボルならそれを文字列にしたものを返す．

```
cl-user(9): (setq *db2*
                  (loop for (key val) in *db*
                    collect `(,(string key) ,(string val))))
(("岩手県" "盛岡市") ("宮城県" "仙台市") ("秋田県" "秋田市") ...)
cl-user(10): (assoc "神奈川県" *db2* :test #'string=)
("神奈川県" "横浜市")
```

assoc は条件に合致した要素が複数あっても，最初に見つかった要素のみを返す．したがって，新しいキーバリュー・ペアを先頭に cons すると，古いペアは隠ぺいされる．しかし新しいペアを取り除けば，再び古いペアが検索されるようになる．第11章ではこの連想リストによりSchemeの環境が提供される．

1.7 数，整数，浮動小数点数，有理数

Common Lispにおいても整数や浮動小数点数など，Lisp以外の計算機言語と同様な数の型が用意されているが，ほかにも有理数や複素数などがある．ここでは Common Lispに特徴的な有理数について紹介する．

岩波書店の数学辞典によれば，有理数とは次のようなものである．

> 「整数の対 (a, b)において $b \neq 0$であるとし，このような対 (a, b) の全体を Pとする． Pに属する対の間の同値関係 $(a, b) \simeq (c, d)$を $ad = bc$によって定義する．これによって Pを類別し，その類を有理数という．」
>
> ［岩波数学辞典，第3版］

Common Lispでは有理数の体系をこの定義に忠実に実装しているので，割り算で割り切れない場合にはその結果は有理数となる．以下の実施例を見られたい．

```
cl-user(60): (/ 1 3)
1/3
cl-user(61): 1/3
1/3
cl-user(62): (+ 1/3 1/3 1/3)
1
cl-user(63): (/ 22 7)
22/7
cl-user(64): (round (/ 22 7))
3
1/7
```

浮動小数点を使わないかぎり，Lispでは計算での桁落ちを心配する必要はない．

第2章 関数定義および述語と条件文

第1章で述べたように，Lispでプログラムするということはすなわち関数定義をするということである．本章では関数定義の書式と書き方について学ぶ．述語とはその返す値が真あるいは偽を表すと解釈される関数のことである．本章ではこの述語と条件文の書き方についても学ぶ．

2.1 関数定義の書式

数を2乗する関数はCommon Lispにはない．それを自分で定義するには，次のようなdefunの書式で書いて評価すればよい．

```
(defun square (n)
  "returns the square of number 《n》."
  (* n n))
```

あるいは，リストを引数としてその先頭の要素が最後の要素となるようなリストを返す関数left-rotateは次のように書けばよい．

```
(defun left-rotate (list)
  "moves the first element to the end of the 《list》."
  (append (rest list)
          (list (first list))))
```

ここで，squareやleft-rotateは新しく定義する関数の名前であり，それに続く(n)や(list)が仮引数リスト(関数パラメータリスト)で，その中のnやlistが仮引数(関数パラメータ)であり，その次の文字列はオプションであってもなくてもよい文字列で，通常は関数の説明となるものをドキュメンテーションとする．それ以下が関数定義の本体である．square は見ての通りだが，left-rotate の意図するところは，わたってくる引数 list についてその restを取り出して，それと引数 listの第1要素をリストしたものを appendでつなぐ．これでまさに先頭の要素が最後の要素となるようなリストが得られることが分かる．

問題 2.1 この squareと left-rotate 定義を自分のLisp環境に定義し，実際に計算を試してみよう．

問題 2.2 数の3乗を返す関数cubeを定義せよ．

問題 2.3 Lisp組み込みの関数 lastはリストの最後の要素を含む長さ 1 のリストを返す．そうではなくリストの最後の要素を返す関数last1を定義せよ．

問題 2.4 最後の要素がリストの最初の要素となるようなリストを返す関数right-rotateを定義せよ．(hint:最後の要素を除いたリストが必要になるが，それに必要な関数はすでにこの書籍で既出である．さて，それはなんだろうか？)

正式な関数定義の書式を以下に示す．

```
(defun 《関数名》 《ラムダリスト》
  [{《宣言》}* | 《ドキュメンテーション》 ]
```

```
  {《書式》}* )
```

ここで，本書籍で使う**BNF記法**について説明する． [《A》]は《A》があってもなくてもよいことを表し， [《A》|《B》]の縦棒は《A》か《B》かの選択を表し， {《A》}*は《A》の0個以上の並びを意味している．同様に， {《A》}+は《A》の1個以上の並びを意味している．

上記の定義式ではドキュメンテーションを書いたら宣言文を書けないかのように読めるが，実はそんなことはなく両方を書くことができる．そのほかにも中級・上級用にはラムダリストの書き方などいくつか詳細に記述すべき事項があるが，ここではこれまでに止め，それ以上のことは必要な時点でおいおい説明することにする．特に，ラムダリストについては6.2節で詳述している．

これまで出現した例題では関数本体中に1個の書式しかなかったが，関数本体には複数の書式を並べて書ける．その場合，関数実行されたときその関数が返す値は最後の書式の値である．つまり最後以外の書式は副作用のためにあると言ってもよい．これを**暗黙のPROGN**と言う．progn とは次のような書式をしているものだが，progn が返す値は，並んでいる書式の最後の値であり，関数本体は明示的にはなくてもこのprogn で括られていることと同等という意味である．

```
(progn
  {《書式》}* )
```

ラムダ式で無名関数を作る

defunを使った関数定義では関数定義の本体が関数名の symbol-functionの値としてセットされる．たとえば，上記 left-rotateでは，実は次のような書式が symbol-functionの値にセットされる．

```
(lambda (list)
  (block left-rotate
    (append (rest list)
            (list (first list)))))
```

Lisp処理系が書式を評価するとき，リスト形式でかつ最初の要素がシンボルでそれに関数定義があれば，定義されている上記のようなラムダ式をとりだして，順次実行する．ところが，関数名の代わりに直接ラムダ式を書くこともできて，そのような関数は無名関数と呼ばれる．以下の実行例を見られたい．

```
cl-user(3): ((lambda (list)
               (append (rest list)
                       (list (first list))))
             '(a b c d))
(b c d a)
```

いちいち defunで関数定義をするほどのこともなく，その場でちょっと一時的な関数がほしいときに，このラムダ式による無名関数がよく用いられる．第6章のMAP関数では頻繁に無名関数が用いられる．ラムダ式については6.2節でさらに詳細に述べる．

2.2 述語

関数のうち，とくにBooleanに相当する真あるいは偽を返す関数を述語と呼ぶ．先に組み込みのシンボルtとnilを見たが，Lispでは偽を表すには常にnilが用いられ，真を表すにはnil以外のリスプオブジェクトが用いられる．**nil以外であればなんでも真と解釈される**のであるが，真であることを特に表現するにはtが用いられる．

Lispではあらかじめ様々な述語が用意されている．

- atomはアトムかどうかをテストする

```
cl-user(4): (atom 'foo)
t
cl-user(5): (atom '(this is a list))
nil
cl-user(6): (atom 'pi)
t
cl-user(7): (atom pi)
t
cl-user(8): (atom t)
t
cl-user(9): (atom 1234)
t
cl-user(10): (atom -3.14)
t
cl-user(11): (atom 1/3)
t
```

- listpはリストかどうかをテストする

```
cl-user(12): (listp 'foo)
nil
cl-user(13): (listp '(this is a list))
t
```

- symbolpはシンボルかどうかをテストする

```
cl-user(14): (symbolp 'foo)
t
cl-user(15): (symbolp '(this is not a symbol))
nil
cl-user(16): (symbolp 'pi)
t
cl-user(17): (symbolp pi)
nil
cl-user(18): (symbolp 1234)
nil
```

- numberpは数かどうかをテストする

```
cl-user(19): (numberp 'foo)
nil
cl-user(20): (numberp 5)
t
cl-user(21): (setq foo 5)
5
cl-user(22): (numberp foo)
t
```

```
cl-user(23): (numberp 'pi)
nil
cl-user(24): (numberp pi)
t
cl-user(25): (numberp 1234)
t
```

Lispでアトムなのかリストなのか曖昧なものが nil である．nil は偽として用いられるが同時に空リストでもある．はたしてそれは偽のためのアトムなのだろうかそれともやはりリストなのだろうか．

```
cl-user(26): (atom 'nil)
t
cl-user(27): (atom '())
t
cl-user(28): (listp 'nil)
t
cl-user(29): (listp '())
t
```

空リストであることをはっきりさせるためには `'()` を用い，そうでない場合には `nil` を用いる人がいる．よい習慣だと思う．

- 空リストかどうかを調べる方法には null と endp の二つがある

```
cl-user(31): (null '(this is not empty))
nil
cl-user(32): (endp '(this is not empty))
nil
cl-user(33): (null '())
t
cl-user(34): (endp '())
t
cl-user(35): (null 'this-is-not-a-list)
nil
cl-user(36): (endp 'this-is-not-a-list)
Error: Attempt to take the cdr of this-is-not-a-list which is not listp.
  [condition type: type-error]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(37): :continue 0
cl-user(38):
```

述語は最後が p の場合がほとんどであるが（もちろん“predicate”の p に違いない），なぜか atom と null はそうではない．ときどき nullp と間違って書いてしまう．アトムが与えられても null はエラーにならないが，endp ではエラーとなる．扱うのが必ずリストであり，アトムはこないはずであると分かっている場合には，積極的に endp を使う人がいる．よい習慣である．

- 等しいという意味にもいろいろある

```
cl-user(40): (eq 'foo 'foo)
t
```

```
cl-user(41): (setq bar 'foo bii 'foo)
foo
cl-user(42): (eq bar bii)
t
cl-user(43): (setq foo '(this is a list))
(this is a list)
cl-user(44): (setq bar foo)
(this is a list)
cl-user(45): (eq foo bar)
t
cl-user(46): (eq foo '(this is a list))
nil
cl-user(47): (eql foo '(this is a list))
nil
cl-user(48): (equal foo '(this is a list))
t
```

正確な理解は第5章で得られるが，簡単に言うと eq は Lisp のポインタとして同一のものであるという意味で，eql は対象が組み込みのリスプオブジェクト（文字とか構造体とかCLOSオブジェクトなど）として等しいかどうか，equal は対象が特にリストで見かけ上等しいかどうかを調べるために用いられる．上記実施例の45と46，48の違いに留意されたい．なお，eq で真であれば eql でも equal でも必ず真であり，eql で真であれば，equal でも必ず真である．このほかに，数についても数値としての等しさを調べるための = や，文字列では文字列としての等しさを調べるための string= ，文字列でアルファベットの大文字小文字を区別しない string-equal などもある．

- リストの要素にあるかどうか，メンバーシップをしらべるには member と find がある

```
cl-user(49): (setq tools
                   '(saw wrench pliers hammer screwdriver))
(saw wrench pliers hammer screwdriver)
cl-user(50): (member 'hammer tools)
(hammer screwdriver)
cl-user(51): (find 'hammer tools)
hammer
```

member はもしメンバーがあればそのメンバーを最初の要素とするリストを返す．find はもしメンバーが見つかれば，そのメンバーを返す．実は find はリストというよりも，もう少し広い型である列（シーケンス）に対する操作であり，配列や文字列に対しても有効であるが，member はリストのみを対象とする．配列については第12章を参照されたい．

問題 2.5 リストを引数としてそれが回文（「たけやぶやけた」のようにひっくり返しても同じになるもの）かどうかを調べる次のような関数 paridrome-p を定義せよ．リストの並びを逆順にするには reverse を用いると便利である．

```
(parindrome-p '(a b c b a)) ⇒ t
```

2.3 論理結合演算

複数の論理式を結合するのが and や or の演算子であるが，Lisp においても述語による真偽値を結合する and や or や，真偽値を反転する not がある．

- 関数 not は引数の真偽値を反転する

```
cl-user(52): (not t)
nil
cl-user(53): (not nil)
t
cl-user(54): (not 'something)
nil
cl-user(55): (not (not 'something))
t
```

nil 以外のオブジェクトは真偽値としては真と解釈されることに注意されたい．それでは都合が悪く，nil のときには nil を得たいが，nil 以外のオブジェクトに対しては t を得たいときのイディオムとして (not (not 《何か》))というパターンがよく用いられる．

- マクロ and と or の評価は予想通り

```
cl-user(56): (and t t t)
t
cl-user(57): (and t t nil)
nil
cl-user(58): (or nil nil nil)
nil
cl-user(59): (or nil nil t)
t
cl-user(60): (setq foo 'baz)
baz
cl-user(61): foo
baz
cl-user(62): (and nil (setq foo 'bar))
nil
cl-user(63): foo
baz
cl-user(64): (and 1 2 'foo)
foo
cl-user(65): (or t (setq foo 'bar))
t
cl-user(66): foo
baz
cl-user(67): (or nil nil 'foo)
foo
```

ここで大事なことは，and では引数の並びを順に評価していって，途中で nil が得られたら，それ以降の引数は評価されない，ということである．最後まで到達したらそのときに and 構文の返す値は最後の要素の値である．対照的に or では引数の並びを順に評価していって，途中で nil 以外の値が得られたらその値を返す．すべてが nil の場合には最後まで到達して，そのときには nil を返す．マクロについては第8章で詳しく述べる．

問題 2.6 任意のS式を引数として，それが要素数 1 のリストであれば真を返し，さもなければ偽を返す述語 length=1 を定義せよ．nil ではない確かにリストであることを調べるには述語 consp が便利である．

問題 2.7 任意のリストとS式を引数として，リストの第 1 要素が与えられたS式であるかどうかを調べる述語 starts-with を定義せよ．nil ではない確かにリストであることを調べるには述語 consp が便利である．

```
(starts-with '(a b c) 'a) ⇒ t
```

2.4 条件文

if 特殊形式

条件に対応して処理内容を変更するための条件文は，プログラミングに必須のものである．これなくしてはその手続きは一見どんなに複雑に見えても，その制御はただの一直線状のものとなる．Common Lispの条件文には if 形式もあるが，cond 形式もよく使われる．if 形式では分岐は二つだけだから，複雑な処理になると if の入れ子構造になってしまうが，cond 形式では任意個の分岐が許される．

if 条件文の書式は以下のとおりである．

```
(if 《test部》 《then部》 [《else部》])
```

最初に《test部》が評価され，結果が真であれば《then部》のみが評価され，《else部》は評価されない．《test部》の結果が偽であれば，《then部》は評価されず，《else部》がある場合にはそれが評価される．Common Lispの **if 構文には「else」という字句はない**こと，《then部》も《else部》もまとまった一つの書式でなければならないことに留意されたい．

```
cl-user(68): (setq tools
                   '(saw wrench pliers hammer screwdriver))
(saw wrench pliers hammer screwdriver)
cl-user(69): (if (member 'hammer tools)
                 'THERE-IS-A-HAMMER
               'NO-HAMMER)
THERE-IS-A-HAMMER
cl-user(70): (if (member 'apple tools)
                 'THERE-IS-AN-APPLE
               'NO-APPLE)
NO-APPLE
```

if 書式では，《then部》でも《else部》でも，一つの書式しか許されないが，《else部》はないという場合には，**when 書式を使えば複数の《then部》の並びを置くことができる**．

```
(when 《test部》 {《then部》}*)
```

対照的に《test部》が偽のときのみ《else部》を実行したい場合には，unless を用いればよい．

```
(unless 《test部》 {《else部》}*)
```

以下を参考にされたい.

```
cl-user(71): (setq tools
                   '(saw wrench pliers hammer screwdriver))
(saw wrench pliers hammer screwdriver)
cl-user(72): (when (member 'pliers tools)
               (setq tools (remove 'pliers tools))
               (setq tools (cons 'pincers tools)))
(pincers saw wrench hammer screwdriver)
cl-user(73): tools
(pincers saw wrench hammer screwdriver)
cl-user(74): (unless (member 'pliers tools)
               (setq tools (cons 'pliers tools)))
(pliers pincers saw wrench hammer screwdriver)
cl-user(75): tools
(pliers pincers saw wrench hammer screwdriver)
```

cond マクロ

cond は最も広範囲に条件を適応可能であり，その書式は if や when よりも少々複雑であるが，慣れればしばしば用いるようになる.

(cond { (《テスト》 {《帰結》}*) }*)

cond の次には (《テスト》 {《帰結》}*) の並びが続くが，この一つの形式 (《テスト》 {《帰結》}*) は cond 節と呼ばれる．処理系はこの cond 節の並びを順に上から調べ，《テスト》が真を返すと，それ以降の cond 節は調べずに《帰結》の並びを順に評価する．その最後の《帰結》の値が cond 条件文全体が返す値となる．どの cond 節も成立しない場合は nil が返される．以下の実行例を参照されたい.

```
cl-user(80): (setq thing 'sphere r 1)
1
cl-user(81): (cond ((eq thing 'circle) (* pi r r))
                   ((eq thing 'sphere) (* 4 pi r r)))
12.566370614359172d0
cl-user(82): (setq thing 'circle r 1)
1
cl-user(83): (cond ((eq thing 'circle) (* pi r r))
                   ((eq thing 'sphere) (* 4 pi r r)))
3.141592653589793d0
```

cond 条件文では最後のテスト部をただの t として，それまでの cond 節をすべて抜けてきても，最後に必ず評価されるべき書式を置くというイディオムがしばしば使われる.

```
cl-user(84): (setq tools
                   '(saw wrench pliers hammer screwdriver))
(saw wrench pliers hammer screwdriver)
cl-user(85): (cond ((member 'hammer tools)
                    'THERE-IS-A-HAMMER)
                   (t 'NO-HAMMER))
THERE-IS-A-HAMMER
cl-user(86): (cond ((member 'apple tools)
                    'THERE-IS-AN-APPLE)
                   (t 'NO-APPLE))
NO-APPLE
```

case, ecase, typecase

単にシンボル比較をしたいだけのときには，もっと簡便な case 文を使うことができる．

```
(case 《書式》
  {(《結果》 {《帰結》}* )}*
 [(otherwise {《帰結》}* )] )
```

最初に《書式》が評価され，その値が順に《結果》のシンボルと等しいかどうか調べられる．あとは cond 条件文と同様であるが，最後にオプションとして otherwise 節をおいてもよい．実は otherwise の代わりに t でもよいのであるが，時々《書式》の結果が t であるかどうかを調べたい時があり，そのためにも普段から otherwise を使っていたほうがよいであろう．いままで《書式》の値が otherwise であるかどうかを調べたいという場面に出会ったことはない．

```
cl-user(87): (setq thing 'sphere r 1)
1
cl-user(88): (case thing
               (circle (* pi r r))
               (sphere (* 4 pi r r)))
12.566370614359172d0
```

《書式》の結果が本当に t のときのみに《帰結》を実行させるには次のようなコードにする．

```
cl-user(89): (setq foo t)
t
cl-user(90): (case foo
              ((nil) (format t ";NIL given."))
              ((t)   (format t ";T given."))
              (otherwise
               (format t ";Anything else than NIL and T.")))
;T given.
nil
```

実は《結果》はアトムではなくてリストであってもよい．その場合には《書式》の評価結果がリスト中の並びのどれかと合致すればよい．上記のコードはそれを積極的に利用したものである．

《書式》の評価結果はあれこれに限られているはずだという場合には，case ではなく ecase を用いる．たとえば，次のようである．ecase では otherwise 節は書かれない．そして，もし書式の評価結果がどれにも合致しなければ，エラーとなる．

```
cl-user(1): (setq weekday 'Sun.)
Sun.
cl-user(2): (ecase weekday
              ((Mon. Monday monday)
               '(he ate through one apple.))
              ((Tue. Tuesday tuesday)
               '(he ate through two pears.))
              ((Wed. Wednesday wednesday)
               '(he ate through three plums.))
              ((Thu. Thursday thursday)
               '(he ate through four strawberries.))
```

```
                ((Fri. Friday friday)
                 '(he ate through five oranges.))
                ((Sat. Saturday)
                 '(he ate through one piece of chocolate cake\,
                   one ice-cream\, one pickle\, one slice of
                   Swiss cheese\, one slice of salami\, one
                   lollipop\, one piece of cherry pie\, one
                   sausage\, one cupcake\, and one slice of
                   watermelon.))
                ((Sun. Sunday sunday)
                 '(he has become a beautiful butterfly!)))
(he has become a beautiful butterfly!)
```

caseやecaseでは《キー書式》の評価結果と《キーバリュー》が比較されたが，typecaseでは《キー書式》評価結果の型が調べられる．Lispの型については第12章と第13章で詳述する．

```
cl-user(3): (setq foo 'a-symbol)
a-symbol
cl-user(4): (typecase foo
              (number '(this is a number.))
              (string '(this is a string.))
              (symbol '(this is a symbol.))
              (array '(this is an array.))
              (otherwise '(anything else.)))
(this is a symbol.)
```

第3章 再帰プログラムを組む

> Lisp には「再帰の心」が浸みわたっている.
>
> [Hofstadter, 1985]（メタマジック・ゲーム，竹内・斉藤・片桐訳）

> 再帰というのは理解が難しく，学ぶ必要のある技術です．それでも一度身につけ，頭の中に入れたら，それは最も強力なプログラミングの道具の一つとなります．だから目を背けて済ますというわけにはいかないのです.
>
> [Guy Steel Jr., 2009]（Coders at Work，青木訳）

> [再帰]を原因として人は普通おかしくならない一方で，機械にとっての再帰は元凶の一つである．その根底には，再帰が記号系一般において避けようのないものであることがある.
>
> [田中久美子, 2022]（記号と再帰新装版，東京大学出版会）

本章では初心者にとって最大の難関である，再帰プログラミングについて学び，その技術を習得する．昔のFORTRANやBASICでは再帰プログラミングができなかった．今ではJavaでも再帰は書けるが，それほど多用されているようには見えない．何故 Lisp では再帰かと言えば，多くの分野で対象物に再帰構造が見られるが，特に人工知能分野ではそれが顕著だからである．再帰構造を有する問題をやっつけるには再帰プログラミングができなければならない.

3.1 リストの横方向に再帰処理する

多くの分野で対象物をツリー構造で捉えることができ，ツリー構造はリストの入れ子構造で自然に表現できる．そしてリストの入れ子構造を処理するのに，しばしば再帰処理が有効である．あるリストが与えられたとき，そのリストの中のリスト構造には手を触れずに，まずは最上層でのリスト要素だけを問題にして，それの再帰処理に慣れてみよう.

我々の member の定義では，リストの横方向に再帰が行われる.

```
(defun our-member (item list)
  "This is almost the same function as common lisp's member."
  (cond ((endp list) nil)                  ; empty list
        ((equal item (car list)) list)     ; matched to the first
        (t (our-member item (cdr list))))); else next
```

ここでセミコロンはそれ以降行末までをコメントとする．最初に `endp` で，もし `list` が空リストであれば，nil を返し，もし先頭要素がマッチすれば，その `list` をそのまま返し，さもなければ，`cdr` について再帰する．試しにこのコード実行をトレースして見てみよう.

```
cl-user(2): (setq tools
                  '(saw wrench pliers hammer screwdriver))
(saw wrench pliers hammer screwdriver)
cl-user(3): (our-member 'hammer tools)
(hammer screwdriver)
cl-user(4): (trace our-member)
(our-member)
cl-user(5): (our-member 'hammer tools)
```

```
 0[2]: (our-member hammer (saw wrench pliers hammer screwdriver))
   1[2]: (our-member hammer (wrench pliers hammer screwdriver))
     2[2]: (our-member hammer (pliers hammer screwdriver))
       3[2]: (our-member hammer (hammer screwdriver))
       3[2]: returned (hammer screwdriver)
     2[2]: returned (hammer screwdriver)
   1[2]: returned (hammer screwdriver)
 0[2]: returned (hammer screwdriver)
(hammer screwdriver)
cl-user(6): (untrace)
nil
```

再帰に慣れない人のとまどいは，再帰した時にそれまでのデータはどこに行ってしまうのだろうかという漠然とした不安ではなかろうか．Javaも含めて現在のほとんどのプログラムでは，関数呼び出しでは制御上の戻り番地と一緒に与えられた引数がスタックに積まれて実行される．だから引数に関するかぎりは，それはスタックに積まれているから，今は現在の値に隠れて見えないけど，関数呼び出しから戻れば用のなくなった情報がスタックから除かれて，またそのときの値が見えるから大丈夫ということである．1回前の自分と今の自分は同じようでも異なる環境にあって，戻ったときにはその環境は再現されている．だから今の自分は昔の自分のことを心配しなくてよいし，昔の自分に戻ったときは先の自分のことは完全に忘れている．

再帰プログラムはたとえば典型的には次のようなパターンをしている．

```
(defun 《関数名》 (《引数並び》)
  "A typical pattern of recursive function."
  (cond (《再帰停止条件》 《停止したときの仕事》)
        (《簡単な場合》 《簡単なときにやる仕事》)
        (t 《一般的な場合だからそれを１ステップ簡単化して再帰する》)))
```

この場合，並べる順序も大事である．停止条件は必ず最初であり，それは最も特殊な場合や極限状態（この場合は引数に最初から空リストが与えられたとき）に相当する．簡単な場合では（この場合は望みの要素が最初にあった場合），簡単に答えが得られるが，さもなければそれは一般的な場合であり，そのときにはそれを一段階ブレークダウンして再帰する．この場合には，最初がマッチしなかったら，その次に残りのリストについて調べればよいということである．大事なことは，この1段階のブレークダウンが，それより前にコードとして存在する簡単な場合と停止条件につながっていることである．

さて，これからいくつか関数を再帰で作ってもらうが，再帰プログラムを書くにはコツがある．これからそれを伝授しよう．

1. まず最初に，すでにその関数はできているものと強く考える．
2. 次に，一般的な場合を考えて，それを一段だけ簡単な場合に，場合分けしてブレークダウンする．リストを横方向に動くときは，(car list)と(cdr list)がそれである．
3. それぞれに対してプログラムする．このとき必要ならその関数を用いてさらに再帰する．リストの横方向のみについて調べればよいときには，(car list)に対しては何もしないが，(cdr list)については調べなければならない．このとき自分はすでに立派に働くその関数を持っていると思って再帰させる．our-member の場合には，自分自身に(cdr list)を任せてしまえばよい．だって，すでにそういう働きをする our-member があるのだから．

4. 最後に，ブレークダウンの極限を考えて，停止条件を置く．この例では (endp list) で調べて，そのときは nil を返す．最後に到達したということは，それまでに見つからなかったということだから，それでよい．

問題 3.1 lat というのは list of atom のことである．lat の要素に一つでも数があれば真を返すような関数 is-there-num を定義せよ．(hint: our-member では要素が item と等しかったらだったでしょ．今度は数だったらだから・・・)

```
> (is-there-num '(a b five c))
nil
> (is-there-num '(a b 5 c))
(5 c)
```

問題 3.2 リストを引数としてその要素すべてがアトムなら t をさもなければ nil を返す関数 is-lat を定義せよ．(hint: our-member では要素が item と等しかったらすぐ戻ったでしょ．だけど今度は最後まで行ったら t だけど，・・・)

問題 3.3 lnum というのは list of number のことである．リストを引数としてその要素すべてが数字なら t をさもなければ nil を返す関数 is-lnum を定義せよ．

再帰しながらその帰りに簡単な仕事をする場合がある

これまでの例では，戻るときには nil あるいは何かの値を持ってトップレベルまで一気に戻った．次の例は戻るときに少しづつ仕事をして戻る例である．

```
(defun count-elements (list)
  (cond ((endp list) 0)
        (t (1+ (count-elements (cdr list))))))
```

この例では，リストを横方向になめながら (cdr を使う)，最後に 0 を返し，再帰で返された値に 1 を加えてまた返すことで，最終的にそのリストの要素数が得られる．

```
cl-user(7): (defun count-elements (list)
  (cond ((endp list) 0)
        (t (1+ (count-elements (cdr list))))))
count-elements
cl-user(8): (trace count-elements)
(count-elements)
cl-user(9): (count-elements '(one two three four five))
 0[2]: (count-elements (one two three four five))
   1[2]: (count-elements (two three four five))
     2[2]: (count-elements (three four five))
       3[2]: (count-elements (four five))
         4[2]: (count-elements (five))
           5[2]: (count-elements nil)
           5[2]: returned 0
         4[2]: returned 1
       3[2]: returned 2
     2[2]: returned 3
   1[2]: returned 4
 0[2]: returned 5
```

```
5
cl-user(10): (untrace)
nil
```

再帰から戻ったときにリストを構成する場合がある

Common Lispの copy-list は与えられた引数のリストをコピーする．それは元のリストに equal ではあるが eq ではない．

```
cl-user(11): (setq foo '(a (b c) (d)))
(a (b c) (d))
cl-user(12): (eq foo (copy-list foo))
nil
cl-user(13): (equal foo (copy-list foo))
t
```

我々の copy-list も全く同様に働く．

```
(defun our-copy-list (list)
  (cond ((endp list) nil)
        (t (cons (car list)
                 (our-copy-list (cdr list))))))
```

```
cl-user(17): (setq foo '(a (b c) (d)))
(a (b c) (d))
cl-user(18): (trace our-copy-list)
(our-copy-list)
cl-user(19): (eq foo (our-copy-list foo))
 0[2]: (our-copy-list (a (b c) (d)))
   1[2]: (our-copy-list ((b c) (d)))
     2[2]: (our-copy-list ((d)))
       3[2]: (our-copy-list nil)
       3[2]: returned nil
     2[2]: returned ((d))
   1[2]: returned ((b c) (d))
 0[2]: returned (a (b c) (d))
nil
cl-user(20): (untrace)
nil
```

ここで，終端までいってから戻るときに，`(cons '(d) nil) => ((d))` であることに留意されたい．

reverseは与えられたリストの逆順のリストを作って返す関数である．我々のreverseを定義してみよう．

```
cl-user(2): (defun our-reverse (list)
              (cond ((null list) '())
                    (t (append (our-reverse (cdr list))
                               (list (car list))))))
our-reverse
cl-user(3): (our-reverse '(a (b c) d (e f)))
((e f) d (b c) a)
```

our-copyもour-reverseもリスト内のリストについては何もしないことに留意されたい．

```
cl-user(4): (trace our-reverse)
(our-reverse)
```

```
cl-user(5): (our-reverse '(a (b c) d (e f)))
 0[5]: (our-reverse (a (b c) d (e f)))
   1[5]: (our-reverse ((b c) d (e f)))
     2[5]: (our-reverse (d (e f)))
       3[5]: (our-reverse ((e f)))
         4[5]: (our-reverse nil)
         4[5]: returned nil
       3[5]: returned ((e f))
     2[5]: returned ((e f) d)
   1[5]: returned ((e f) d (b c))
 0[5]: returned ((e f) d (b c) a)
((e f) d (b c) a)
cl-user(6): (untrace our-reverse)
(our-reverse)
```

問題 3.4 要素がすべて数であるようなリストをもらって，各要素の2乗を返す関数 square-list を定義せよ.

```
> (square-list '(1 2 3 4))
(1 4 9 16)
```

今まではリストを作って返すといっても，その構造は変わらなかった．元の構造と変わったリストを返す場合には，少しだけ難しくなる.

前述の square-list と一見似ているが要素がすべて数であるようなリストをもらって，各要素の平方根のリストを返す関数 sqrt-list を考える．Common Lispでは複素数の体系も用意されているので，実は負数の sqrt もエラーとならずに計算できるが，今ここでは，負数であればこれを計算しないで，リストの長さが短くなるようなプログラムを考える．それはこのようなものになる.

```
cl-user(3): (defun sqrt-list (l)
              (cond ((null l) '())
                    ((< (car l) 0)
                     (sqrt-list (cdr l)))
                    ((= (car l) 0)
                     (cons 0 (sqrt-list (cdr l))))
                    (t (cons (sqrt (car l))
                             (sqrt-list (cdr l))))))
sqrt-list
cl-user(4): (sqrt-list '(-9 0 4 -25))
(0 2.0)
```

おやおや，4の平方根は2ではなくて2.0らしい．2と2.0は数の体系では等しいが，Lisp のオブジェクトとしては明らかに違うものである.

```
cl-user(5): (sqrt 4)
2.0
cl-user(6): (= 2 *)
t
cg-user(7): (equal 2 (sqrt 4))
nil
```

ここで，トップレベルでのREPL（READ-EVAL-PRINT-LOOP）環境において星一つ「*」は直前で返された結果を指示することを注意しておく.

数を扱うときには，それを数学的に見ているのか，Lisp のオブジェクトとして扱っているのかを区別しなければならない．ちなみに，XMLデータスキーマやRDFでは，字句空間あるいは語彙レベルと，値空間を区別する．字句空間では整数の2と浮動小数点数2.0はもちろん異なるが，値空間では等価である．Common Lispにおいては，Lisp 環境において2と2.0は片や整数，片や浮動点数と異なる実装だが，値空間としては等価でなければならない．そのために関数 = がある．

問題 3.5 item と lat を引数とし，リスト lat の中で最も左にある item を取り除く関数 rember を定義せよ．

```
> (rember 'apple '(saw hammer apple wrench))
(saw hammer wrench)
```

問題 3.6 lat を要素とするリスト lats を引数とし，その要素である各 lat の先頭の要素を集めたリストを返す関数 collect-firsts を定義せよ．

問題 3.7 key と item と lat を引数とし，リスト lat の中の key の左側に item を挿入する insert-left を定義せよ．

問題 3.8 key と item と lat を引数とし，リスト lat の中の key の右側に item を挿入する insert-right を定義せよ．

```
> (insert-right 'is 'very
                '(intelligent agent is nice))
(intelligent agent is very nice)
```

3.2 リストの縦方向にも再帰処理する

一般的には，リストを対象とする再帰の枝は，横方向（巾方向）と縦方向（深さ方向）の二つである

lat では要素はすべてアトムで，深さ方向のリストがなかった．一般にツリー構造ではリストの中にもリストが出現する．さて Common Lispの copy-tree は深さ方向も含めて，リストをコピーする．今はそんな機能を何に使うのか分からないかもしれないが，副作用があっても大丈夫なように，同型のリストをコピーして使いたいというときもある．我々の copy-tree は，次のように定義される．

```
(defun our-copy-tree (list)
  (cond ((endp list) nil)
        ((atom (car list))
         (cons (car list)
               (our-copy-tree (cdr list))))
        (t (cons (our-copy-tree (car list))
                 (our-copy-tree (cdr list))))))
```

our-copy-list では一般的な場合に (car list)については再帰しなかったが，今回の our-copy-tree では再帰している．ただし，(car list)がアトムの場合には再帰する必要がなく，その場合は(cdr list)についてのみ再帰している．トレースすると，横方向と縦方向に再帰されていくのがよく分かる．

```
cl-user(1): (defun our-copy-tree (list)
  (cond ((endp list) nil)
        ((atom (car list))
         (cons (car list)
               (our-copy-tree (cdr list))))
        (t (cons (our-copy-tree (car list))
                 (our-copy-tree (cdr list))))))
our-copy-tree
cl-user(2): (setq sentence
'(S (N time)
   (VP (V flies)
       (PP (P like)
           (NP (Det an)
               (N arrow))))))
(S (N time) (VP (V flies) (PP (P like) (NP (Det an) (N arrow)))))
cl-user(3): (trace our-copy-tree)
(our-copy-tree)
cl-user(4): (our-copy-tree sentence)
 0[2]: (our-copy-tree (S (N time)
                         (VP (V flies) (PP (P like) (NP # #)))))
   1[2]: (our-copy-tree ((N time)
                          (VP (V flies) (PP (P like) (NP # #)))))
     2[2]: (our-copy-tree (N time))
       3[2]: (our-copy-tree (time))
         4[2]: (our-copy-tree nil)
         4[2]: returned nil
       3[2]: returned (time)
     2[2]: returned (N time)
     2[2]: (our-copy-tree ((VP (V flies) (PP (P like) (NP # #)))))
       3[2]: (our-copy-tree (VP (V flies)
                                (PP (P like) (NP (Det an) (N arrow)))))
         4[2]: (our-copy-tree ((V flies)
                               (PP (P like) (NP (Det an) (N arrow)))))
           5[2]: (our-copy-tree (V flies))
             6[2]: (our-copy-tree (flies))
               7[2]: (our-copy-tree nil)
               7[2]: returned nil
             6[2]: returned (flies)
           5[2]: returned (V flies)
           5[2]: (our-copy-tree ((PP (P like) (NP (Det an) (N arrow)))))
             6[2]: (our-copy-tree (PP (P like) (NP (Det an) (N arrow))))
               7[2]: (our-copy-tree ((P like) (NP (Det an) (N arrow))))
                 8[2]: (our-copy-tree (P like))
                   9[2]: (our-copy-tree (like))
                     10[2]: (our-copy-tree nil)
                     10[2]: returned nil
                   9[2]: returned (like)
                 8[2]: returned (P like)
                 8[2]: (our-copy-tree ((NP (Det an) (N arrow))))
                   9[2]: (our-copy-tree (NP (Det an) (N arrow)))
                     10[2]: (our-copy-tree ((Det an) (N arrow)))
                       11[2]: (our-copy-tree (Det an))
                         12[2]: (our-copy-tree (an))
                           13[2]: (our-copy-tree nil)
                           13[2]: returned nil
                         12[2]: returned (an)
                       11[2]: returned (Det an)
                       11[2]: (our-copy-tree ((N arrow)))
                         12[2]: (our-copy-tree (N arrow))
```

```
                        13[2]: (our-copy-tree (arrow))
                          14[2]: (our-copy-tree nil)
                          14[2]: returned nil
                        13[2]: returned (arrow)
                      12[2]: returned (N arrow)
                      12[2]: (our-copy-tree nil)
                      12[2]: returned nil
                    11[2]: returned ((N arrow))
                  10[2]: returned ((Det an) (N arrow))
                9[2]: returned (NP (Det an) (N arrow))
                9[2]: (our-copy-tree nil)
                9[2]: returned nil
              8[2]: returned ((NP (Det an) (N arrow)))
            7[2]: returned ((P like) (NP (Det an) (N arrow)))
          6[2]: returned (PP (P like) (NP (Det an) (N arrow)))
          6[2]: (our-copy-tree nil)
          6[2]: returned nil
        5[2]: returned ((PP (P like) (NP (Det an) (N arrow))))
      4[2]: returned ((V flies) (PP (P like) (NP (Det an) (N arrow))))
    3[2]: returned
            (VP (V flies) (PP (P like) (NP (Det an) (N arrow))))
    3[2]: (our-copy-tree nil)
    3[2]: returned nil
  2[2]: returned ((VP (V flies) (PP (P like) (NP # #))))
 1[2]: returned ((N time) (VP (V flies) (PP (P like) (NP # #))))
0[2]: returned (S (N time) (VP (V flies) (PP (P like) (NP # #))))
(S (N time) (VP (V flies) (PP (P like) (NP (Det an) (N arrow)))))
```

さて，最後に問題をいくつか出す．もし再帰プログラミングを自分のものとしたければ，最後までチャレンジしていただきたい．特に squash はあとから手を変え品を変えて同じ問題を解いてもらいたいので，少し難しいが是非チャレンジしてほしい．

問題 3.9 リストを引数とし，その中のアトムの数を数える count-atoms を定義せよ．(hint: リストはツリー構造であるとして，リストの縦方向にも再帰する．)

問題 3.10 リストを引数とし，その中のアトムがすべて数なら t を返す all-numberp を定義せよ．

問題 3.11 our-reverseはリストの中のリストについては手を触れなかった．リストの中のリストについても並びを逆順にするreverse-treeを定義せよ．

問題 3.12 リストを引数とし，その中のシンボル not をすべて取り除く関数 remove-not を定義せよ．

問題 3.13 リストを引数とし，その中のアトムだけを取り出してならべたリストを返す関数 squash を定義せよ．

```
> (squash '(a (a (a (a b))) (((a b) b) b) b))
(a a a a b a b b b b)
```

第4章 変数束縛，動的変数，大域変数

4.1 ラムダ変数束縛

imperativeな計算機言語では，通常，変数を名前の付いた箱というように教えて，その箱に何か値になるものを入れることを，代入と呼んでいる．この書籍でも今までシンボルに値を結び付けることを「セットする」と呼んでいた．一方，変数束縛という言葉は，Lispの理論的基礎となったラムダ計算から来ていて，関数引数である変数にその値として何かを結び付けることである．あるLisp関数があったとき，そのすべての引数すなわち関数パラメータに値が結びついて一体のものとなり，計算可能になった状態を考えてほしい．変数束縛により関数は計算可能になるのだ．

先にleft-rotateを無名関数にした例でつぎのような書式を提示した．

```
(lambda (list)
  (append (rest list)
          (list (first list))))
```

しかし，この状態では関数パラメータが変数束縛されていないために，まだ評価できない．次のように，変数束縛されてはじめて書式として完成される．

```
((lambda (list)
   (append (rest list)
           (list (first list))))
 '(a b c d))
```

実際，Lisp処理系はこの書式を計算することができる．ここで用いられる関数引数を**ラムダ変数** (lambda variable) と呼ぶ．

次の例は利率と元本から新しい金額を計算する例であるが，このようにすべてのラムダ変数が束縛されている場合にはほとんど問題が生じない

```
cl-user(21): (defun interest (bal rat)
               (* bal rat))
interest
cl-user(22): (defun newbalance (balance rate)
               (+ balance (interest balance rate)))
newbalance
cl-user(23): (newbalance 100.0 0.1)
110.0
```

しかし次に示すように，利率ratあるいはrateが関数引数でない場合には，その値がどこかに定まっていて値が得られない限り計算可能ではない．そのような変数は**自由変数**と呼ばれる．

```
cl-user(2): (defun interest (bal)
              (* bal rate))
interest
cl-user(3): (defun newbalance (balance)
               (+ balance (interest balance)))
```

```
newbalance
cl-user(4): (newbalance 100.0)
Error: Attempt to take the value of the unbound variable `rate'.
[condition type: unbound-variable]
cl-user(5): (setq rate 0.1)
0.1
cl-user(6): (newbalance 100.0)
110.0
```

上記例では，interestで使われるrateは自由変数となっている．最初のnewbalanceの計算ではエラーとなったが，**トップレベルにおいてシンボルrateに値をセットして計算可能**になった．トップレベルにおいて関数の自由変数と同じシンボルの値はどこからでも利用できるのだ．

それではLispシステムを立ち上げ直して，次のような例ではどうだろうか．

```
cl-user(4): (defun interest (bal)
              (* bal rate))
interest
cl-user(5): (defun newbalance (balance rate)
              (+ balance (interest balance)))
newbalance
cl-user(6): (newbalance 100.0 0.1)
Error: Attempt to take the value of the unbound variable `rate'.
[condition type: unbound-variable]
cl-user(7): (defun newbalance (balance rate)
              (declare (special rate))
              (+ balance (interest balance)))
newbalance
cl-user(8): (newbalance 100.0 0.1)
110.0
```

関数 interest では相変わらず rate は自由変数である．一方，関数 newbalance で rate はラムダ変数になっており，問題はその rate を関数 interest で使うことができるかどうかである．実施例6の実行時には interest の rate は未束縛エラーとなった．関数 interest の中から newbalance のラムダパラメータが見えなかったのである．ところが，実施例7の newbalance 再定義においてラムダ変数である rate を**特殊変数**であるとdeclare表記で宣言した．すると実施例8に示すように計算可能となった．

変数束縛には二種類の方法がある．それが**静的束縛**と**動的束縛**である．静的束縛では束縛条件はあくまでもコードテキストの字面上だけで決まる．だから関数interestの自由変数rateについての字面上の情報はなにもなく，もしトップレベルにおいても値がセットされていなければ，未束縛となるしかない．一方，動的束縛の束縛条件は実行時にスタックのように積まれていく束縛環境によって決まる．だから関数newbalanceの再定義においてrateを特殊変数としたことでそれが動的な束縛環境となり，静的には見えていなくても実行時には見えるようになった．

上記例を静的束縛で解決する方法はある．とにかく字面上で静的な環境としてrateが見えればよいのである．

```
cl-user(2): (defun newbalance (balance rate)
              (unless (fboundp 'interest)
```

```
                (defun interest (bal)
                  (* bal rate)))
              (+ balance (interest balance)))
newbalance
cl-user(3): (newbalance 100.0 0.1)
110.0
```

こんなきたないコードは書きたくはないが，この例ではinterestの定義において自由変数であるrateは，字面上でnewbalanceのラムダ変数であるrateであると分かる．この場合には，静的変数として自由変数があっても計算可能になる．fboundpは関数定義の有無を調べるときに用いられる述語である．

4.2 let 変数束縛

プログラム中で局所変数が必要なとき，次のような let 構文を用いて，局所変数を導入する．

```
(let ({《変数》|(《変数》 [《初期書式》])}*)
  {declaration}*
  {form}* )
```

let の次の括弧には変数リストを書く．変数リストの要素は，変数名かまたは括弧で囲われた変数名と初期値を与えるための書式のリストである．この《初期書式》は評価され，その値が《変数》に束縛される．《初期書式》がないあるいは変数リストの要素がただの変数名だけの場合は，自動的にその値として nil が与えられる．次の例を見てみよう．

```
cl-user(1): (defun foo ()
              (let ((x 1))
                (print x)
                (let ((x 0))
                  (print x))
                (print x)))
foo
cl-user(2): (foo)

1
0
1
1
```

ここで，最後に表示される1は関数 foo の返す値であるが，その前の 1, 0, 1 は関数 foo 内部の print 文で印刷されたものである．最初の 1 は最初の let 書式にある変数 x が印刷されたものであり，その値は let 書式で与えられた 1 である．二番目の 0 は二番目の let 書式の変数 x が印刷されたものであり，その値は 0 である．三番目の 1 は foo の最後の print 文で印刷されたものであり，その値は最初の let 書式で与えられた 1 である．

let 構文で導入される変数を **let変数** と呼ぶことにしよう．上記例が教えるものは，let 変数の有効範囲はその let 書式の中だけということであり，これを「**let 変数のスコープはその let 構文の中**である」と言う．そして let 構文の中で入れ子にある let 構文によって，同じ変数名が導入された場合には外側の値は内側の

let 構文によって**隠ぺい**される．内側の let 構文より外では再び外側の let 構文のスコープとなり，変数の値もそのスコープにおけるものに戻る．

逆に，いくら let 構文の入れ子が多重にあっても，外側で導入された let 変数の値は内側で隠ぺいされない限り多重の入れ子を経由して外へ見ていけば分かる．次の例を参照されたい．

```
cl-user(3): (defun bar (x)
              (let ((y 2))
                (let ((z 3))
                  (list x y z))))
bar
cl-user(4): (bar 1)
(1 2 3)
```

ここでは最も内側で引用される x や y の値はその外側の let における値であることを見られたい．

一つの let 構文において複数の let 変数を置くことができるが，その場合複数の束縛は並列に行われる．すなわち，前の束縛が後の初期値の計算に影響することはなく，（初期値評価時の副作用が悪さをしないかぎり）どういう順序に並べても，束縛値は常に同じになる．次の例を見られたい．

```
cl-user(8): (setq x 'outside)
outside
cl-user(9): (let ((x 'inside)
                  (y x))
              (list x y))
(inside outside)
cl-user(10): (let ((y x)
                   (x 'inside))
               (list x y))
(inside outside)
```

しかし構文の見かけは同じでも let* では変数束縛は並びの順番に行われて，前の変数束縛が後の変数束縛に影響する．次の例を見られたい．

```
cl-user(11): (let* ((x 'inside)
                    (y x))
               (list x y))
(inside inside)
```

4.3 特殊変数と動的スコープ

上記のlet変数は静的束縛であり変数は静的変数である．すなわち変数の値の参照は字面上で決定される．しかしラムダ変数と同様に， let 変数にも静的変数と動的変数がある．special と宣言すればlet変数も動的変数になる．

```
cl-user(2): (defun interest (bal)
              (* bal rate))
interest
cl-user(3): (defun newbalance (balance)
              (let ((rate 0.1))
                (+ balance (interest balance))))
```

```
newbalance
cl-user(4): (newbalance 100.0)
Error: Attempt to take the value of the unbound variable `rate'.
[condition type: unbound-variable]
cl-user(5): (defun newbalance (balance)
              (let ((rate 0.1))
                (declare (special rate))
                (+ balance (interest balance))))
newbalance
cl-user(6): (newbalance 100.0)
110.0
```

スペシャル宣言されないlet変数は静的変数(レキシカル変数)である．関数interest内部のrateは自由変数であるが，関数newbalanceにおいてlet変数としてrateが導入されても，字面上では関数interest中からそれを見ることができない．実施例5においてnewbalanceを再定義したが，そのときにはlet変数であるrateを特殊変数と宣言している．するとラムダ変数のときと同様に，実行時にlet変数の束縛情報が動的束縛環境にスタックのように積まれて，関数interestがそれを参照することができるようになる．

動的変数であっても，静的変数と同様に，**変数の隠ぺい**はある．次の例を見られたい．

```
cl-user(2): (defun foo ()
              (let ((x 1))
                (declare (special x))
                (print x)
                (let ((x 0))
                  (declare (special x))
                  (print x))
                (print x)))
foo
cl-user(3): (foo)

1
0
1
1
```

この例では，最初のlet変数xも，その次のlet変数xも特殊変数と宣言されているが，変数の隠ぺいがあって，内側のlet変数xによって外側のlet変数xは隠ぺいされた．ただし，この隠ぺいは静的変数とは異なり，関数実行時に動的にスタックに積まれることにより，常に最上位のものしか見えないということで起こっていることを注意しておく．

Common Lispには「実際には2つの種類の変数がある」(*CLtL2*, 5.1.2 変数)とされている．それがレキシカル変数（または静的変数）と特殊変数（または動的変数）である．上記のように，変数の値を考える時に，プログラムテキストの字面上で見ていけばよい方式を**レキシカルスコープ**とか**静的スコープ**といい，そういう変数をレキシカル変数とか静的変数とかと言う．静的変数では実行時に何が束縛されるかということは考えずに，プログラムテキスト上だけで考えればよい．ところが古いリスプの束縛方式では実行時にその変数には何が束縛されているかを考えなくてはいけない方式であった．これを**動的スコープ**と言い，そのような

変数を**動的変数**と言う．let 変数でもラムダ変数でも，それをスペシャル変数と宣言すれば，動的変数となる．

今では emacs を除いてどのプログラム言語でも変数は静的変数であるのが普通であるが，動的スコープが有効に使える場面というのがある．以下の実施例はプログラムパターンとしては悪い例であるが，動的変数の特殊性をよく表している．ただし，ACLでは26行目の再定義は必要ないが，SBCLでは必要であった．

```
cl-user(21): (defun interest (balance)
               (* balance x))
interest
cl-user(22): (defun newbalance (x)
               (+ x (interest x)))
newbalance
cl-user(23): (setq x 0.1)
0.1
cl-user(24): (newbalance 100.0)
110.0
cl-user(25): (proclaim '(special x))
t
cl-user(26): (defun newbalance (x)
               (+ x (interest x)))
newbalance
cl-user(27): (newbalance 100.0)
10100.0
```

さて何が起こったのだろうか．21行目で差引残高に利率 x を掛けて利息を出す関数を定義した．この関数では利率 x の値を定める手段がない．22行目で元金に利息を加えて新しい差引残高をする関数を定義した．この関数では元金 x は関数引数でいわばラムダ変数である．23行目で関数の外側で利率 x を 1 割として，24行目で 100.0 に対する新しい残高を計算して 110.0 となった．このとき利率 x の値はその外側で定義されている 0.1 である．ところが，25行目でproclaimを使って大域的に x は特殊変数と宣言する．すると今度は 10100.0 になった．これは27行目で newbalance が実行されたとき，関数引数である元金 x には 100.0 が束縛されるが，変数 x は大域的に特殊変数とされたため，外側にある利率 x と区別がなくなり，newbalance 内部の interest を実行時にその関数本体中の利率のつもりだった x が100.0 となり，その結果関数 interest の返す値は 10000.0 となってしまうわけである．一般に特殊変数を使うには注意が必要であり，普通は特別な目的のために使用する．

シンボル vs. 変数

Lispシンボルと変数は異なる概念である．第1章および第2章で述べたように，シンボルは Lisp の基本の構成要素(オブジェクト)であり，第5章で述べるように値や関数定義を持つことができるが，**「シンボルすなわち変数」ではない**．Javaにはフィールド変数，ローカル変数，パラメータの3種類の変数があって，変数を含めて識別に用いるトークンを識別子と呼んでいる．一方，Lispでは「シンボルは Common Lispプログラムの変数の名前として用いられる」(*CLtL2*, 5.1.2 変数)が，シンボル＝変数ではない．システムを立ち上げ直して実行した以下の例を参照されたい．

```
cl-user(3): (setq x 1)
1
cl-user(4): (describe 'x)
x is a new symbol.
  Its value is 1
  It is internal in the common-lisp-user package.
cl-user(5): (defun foo ()
              (let ((x 2))
                (print x)
                (describe 'x)))
foo
cl-user(6): (foo)

2
x is a new symbol.
  Its value is 1
  It is internal in the common-lisp-user package.
```

この実施例で，関数 foo の内部の let変数である x は，トップレベルで与えたシンボルとは異なるものであるということがよく分かる．describeされる x はトップレベルのシンボルであり，let変数の x ではない．それでは let変数というのは定義的にトップレベルのシンボルではないと言えるのかというと，そうでもない．特殊変数である let変数は一見トップレベルのシンボルであるかのように振る舞うのだ．

```
cl-user(7): (defun foo ()
              (let ((x 2))
                (declare (special x))
                (print x)
                (describe 'x)))
foo
cl-user(8): (foo)

2
x is a new symbol.
  Its value is 2
  It is internal in the common-lisp-user package.
cl-user(9): x
1
cl-user(10): (compile 'foo)
foo
nil
nil
cl-user(11): (foo)

2
x is a new symbol.
  Its value is 2
  It is internal in the common-lisp-user package.
cl-user(12): x
1
```

この例では関数 foo の中の let変数を特殊変数とした．するとdescribeされる x は特殊変数である x であり，シンボルのようにdescribeされる．ただし，実行終了後はスタックとして動的に積まれた let変数の束縛は消えてしまうので，トップレベルの x が元のまま見えるようになる．この挙動は関数 foo をコンパイルしても同様であり，let変数はコンパイラ言語の変数とも異なるLisp独特のものであることを教える．

*CLtL2*と*HyperSpec*ではdescribeの引数はobjectとされている．通常のlet変数は見かけはシンボルのように見えるけれども実はdescribeでいうオブジェクトではなくて，特殊変数であるlet変数はdescribeでいうオブジェクトなのだろうか．ついでながら，*CLtL2*および*HyperSpec*で，関数symbol-valueの引数はsymbolとされているが，レキシカル変数をアクセスできないという但し書きがある．だから以下のような振る舞いはその説明に合致している．Lispシステムを立ち上げ直して，

```
cl-user(2): (defun foo ()
              (let ((x 1))
                (print x)
                (symbol-value 'x)))
foo
cl-user(3): (foo)

1
Error: Attempt to take the value of the unbound variable `x'.
[condition type: unbound-variable]
cl-user(4): (defun bar (x)
              (print x)
              (symbol-value 'x))
bar
cl-user(5): (bar 1)

1
Error: Attempt to take the value of the unbound variable `x'.
[condition type: unbound-variable]
```

ここで，実施例3では let 変数の値を印刷してから，シンボル x の symbol-value を取り出そうとしたがトップレベルでの定義がないので未束縛エラーとなり，実施例5ではラムダ変数の値を印刷してから，シンボル x の symbol-value を取り出そうとして未束縛のエラーとなった．

ところが，特殊変数であるlet変数やラムダ変数ならば，その変数の値を symbol-valueで見ることができる．

```
cl-user(6): (defun foo ()
              (let ((x 1))
                (declare (special x))
                (print x)
                (symbol-value 'x)))
foo
cl-user(7): (foo)

1
1
cl-user(8): (defun bar (x)
              (declare (special x))
              (print x)
              (symbol-value 'x))
bar
cl-user(9): (bar 1)

1
1
cl-user(10): x
```

```
Error: Attempt to take the value of the unbound variable `x'.
[condition type: unbound-variable]
```

動的変数が必要な場面というのがあまり想像できないかもしれないが，プログラム中のテキストと関係なく，実行時の値がほしいとか，実行時モードによって場合分けするとかの場合には有効である．特に，Lisp以外のオブジェクト指向言語で，メソッドの第1引数に self を置くものがあるが，クラスに定義されるメソッド中で将来生成されるであろう任意のインスタンスオブジェクトを実行時に指定するにはそれを動的変数にするしかない．

以上見てきたように，Common Lispの変数というのはまことに分かりにくい．同時に同じ名前を持つレキシカル変数と特殊変数があり得て「任意の時点で，同一の名前を持った変数は両者またはいずれかが値を持つことができる」(*CLtL2*, 5.1.2 変数)．ところが，変数の識別子に相当するものをシンボルとよびしかもレキシカル変数はあきらかにシンボルとは言えないが，特殊変数では本当にシンボルのように振る舞うので訳が分からなくなる．以下は Common Lisp HyperSpec からの引用である．ただし，太字は筆者による．

> *In the following example, the **symbol x** is used, at different times, as the name of a lexical variable and as the name of a dynamic variable.*

```
(let ((x 1))            ;Binds a special variable x
  (declare (special x))
  (let ((x 2))          ;Binds a lexical variable x
    (+ x                ;Reads a lexical variable x
       (locally (declare (special x))
                x))))   ;Reads a special variable x
=>  3
```

上記 *HyperSpec*からの引用例では，1 が束縛された x は let 変数ではあるが宣言によって特殊変数となり，2 が束縛された x はレキシカル変数のままであり，`(locally ... )`の中の x は特殊変数と宣言されたので，この場合たまたま最初に特殊変数と宣言された let 変数を指していることになる．さらに詳細な悩ましい技術的議論は第7章のクロージャとエクステントを参照されたい．第11章では Scheme の実装を学ぶが，束縛の完全な理解はそこで得られるであろう．

特殊変数と大域変数

ラムダ変数でも let変数でも，それらを特殊変数とすると関数symbol-valueの引数となりえてしまうが，それでも**ラムダ変数やlet変数である特殊変数はトップレベルにおけるシンボルとは別のオブジェクトのように振る舞う**．しかし，トップレベルにおけるシンボルは本来大域的である．そして，let変数やラムダ変数による変数の隠ぺいがなければ，関数の中やlet構文中でもその大域的なシンボルを参照したりそれに値をセットしたりできてしまう．

```
cl-user(19): (defun foo ()
               (setq x 1)
               (symbol-value 'x))
foo
cl-user(20): (foo)
```

```
1
cl-user(21): x
1
```

上記例は大域的なシンボルについてであるが，局所的な束縛によって生じた変数によって大域的なシンボルが隠ぺいされる場合には，setqはその局所的なlet変数あるいはラムダ変数について行われる．

```
cl-user(3): (setq x 1)
1
cl-user(4): (defun foo (x)
              (setq x 100)
              x)
foo
cl-user(5): x
1
cl-user(6): (foo x)
100
cl-user(7): x
1
cl-user(8): (let ((x 2))
              (setq x 100)
              x)
100
cl-user(9): x
1
```

上記例は静的変数についてであるが，同じことは動的変数についても同様である．

```
cl-user(2): (setq x 1)
1
cl-user(3): (defun foo (x)
              (declare (special x))
              (setq x 100)
              x)
foo
cl-user(4): x
1
cl-user(5): (foo x)
100
cl-user(6): x
1
cl-user(7): (let ((x 2))
              (declare (special x))
              (setq x 100)
              x)
100
cl-user(8): x
1
```

symbol-valueはレキシカル変数には使えなかったが，setq はレキシカル変数にも特殊変数にも使われることに注意されたい．

4.4 defvar, defparameter, defconstant

上記では let 変数を導入し，静的変数と動的変数を説明した．静的変数ではテキストにおける入れ子構造を見て，内から外へと値を探せばよく，動的変数ではプログラム実行時に積まれるスタックをイメージして，上から下へと値を探せばよい．では静的でも動的でも let 変数が見つからなかったら，どうなるのか．すなわち，静的では一番外側の Lisp 環境まで到達したら，動的ではスタックの床まで到達したらどうなるのか．そのときは静的でも動的でもその変数に大域的にどんな値が与えられているかあるいは与えられていないか，で決定される．そのような変数はコードのどこからでもグローバルに見えるので**大域変数**と呼ばれる．トップレベルで変数に setq で値を与えることは対話的には普通に行われるが，プログラムテキストとしてはあまりよいマナーではない．また，関数の中でラムダ変数でも let 変数でもない変数に値をセットすることも，非常に特殊な例であり，それが是非とも必要でかつ効果的であるという場合にしか行わない．明示的に大域変数を定義するために， defvar，defparameter，defconstant というマクロが用意されている．その書式は次のとおりである．ここで《初期値》はいずれも評価される．これらのマクロで定義された変数はいずれも特殊変数となる．

```
(defvar       《シンボル名》 [《初期値》[《ドキュメンテーション》]])
(defparameter 《シンボル名》  《初期値》[《ドキュメンテーション》])
(defconstant  《シンボル名》  《初期値》[《ドキュメンテーション》])
```

普通はトップレベルでこれらの書式を使って定義して，定義された変数はコードのどこからでもその値を参照する．ただし，これらの三者には微妙な違いがあり，特に defconstant は文字通り定数として使われて，コンパイル時にはひょっとしたら，コード中の即値となっている可能性がある．だから defconstant で定義した変数は使うだけで決して値を変えようとしてはならない． defparameter では値をセットしなおすことはできるが，普通はプログラムの中で頻繁に値を変更するということはあまりないようなものに使う．システムパラメータみたいなものによい．defvar は普通にプログラム中で値をセットする大域変数であるが，なんと明示的に使うまでは初期値は書いてあってもそこでは評価されず，すでに値がセットされていなければ，いつまでも未束縛のままである．この特殊な機能は評価時に大きなコストを伴うようなフォームをセットする場合には都合がよい．ところが注意すべきは，値が未束縛の場合にのみ初期値がセットされることである．だから，一旦プログラムを動かしたのち，プログラムを再ロードしてもその時の defvar は働かない．一方， defparameter は宣言されたら必ずその場でその初期値がセットされる．そういうこともあって， defvar 宣言の初期値はオプショナルであり省略もできるが（その場合は初期値はセットされずに未束縛のままである），これを使わずに，変数に確かにその値がセットされていることを確実にするために，時々 defvar 宣言の直後に値をわざわざ setq するようなコードも見かける．それは実質的には defparameter と同じことである．もし， defvar で挙動がおかしいと思ったら， defparameter に変えてみると解決することがあるかもしれない．

```
cl-user(1): (defvar *val1*)
*val1*
cl-user(2): (defvar *val2* 2)
*val2*
cl-user(3): *val1*
Error: Attempt to take the value of the unbound variable *val1*'.
  [condition type: unbound-variable]
```

```
Restart actions (select using :continue):
 0: Try evaluating *val1* again.
 1: Set the symbol-value of *val1* and use its value.
 2: Use a value without setting *val1*.
 3: Return to Top Level (an "abort" restart).
 4: Abort entirely from this (lisp) process.
[1] cl-user(4): :continue 3
cl-user(5): *val2*
2
cl-user(6): (setq *val1* 1)
1
cl-user(7): (setq *val2* 'two)
two
cl-user(8): *val1*
1
cl-user(9): *val2*
two
cl-user(10): (defvar *val2* 2)
*val2*
cl-user(11): *val2*
two
cl-user(12): (setq *val2* 2)
2
cl-user(13): *val2*
2
```

このように大域変数はアスタリスクで囲むことが，習慣となっている．定義としてはアスタリスクをつけてもつけなくてもよいし，あくまでもアスタリスクまで含めて一つのシンボルなのであるが，見るだけでこれは大域変数だと知らしめることがこの慣習の目的である．

実施例1では初期値が定義してないために，実施例3で値を得ようとしてもエラーとなる．一方，実施例2では定義してあるため，その値を実施例5で得ようとして，ここで初めて初期値がセットされる．一旦値がセットされてしまうと，実施例10のように再度 defvar で定義してもそれは働かない．ただし実施例12のように setq などでセットしなおすことはできる．要するに defvar では大域変数の宣言のみをして，setq などで値をセットして使うということが原則と考えたほうが安全である．

一方，defparameter では下記のように，初期値を与えれば，その場でそれが有効となり，再定義も可能である．もちろん setq などでセットし直すことも可能である．defparameter では必ず初期値を与えなければならない．

```
cl-user(14): (defparameter *val3* 3)
*val3*
cl-user(15): *val3*
3
cl-user(16): (defparameter *val3* 'three)
*val3*
cl-user(17): *val3*
three
cl-user(18): (setq *val3* 3)
3
```

```
cl-user(19): *val3*
3
```

defconstant も初期値が必須であり，ACL では defconstant 宣言された変数に値をセットしようとすると，エラーとなる．defconstant の大域変数には defvar や defparameter と区別するためにアスタリスクではなく + 記号で囲む習慣の人がいる．これも一目でそれが定数であると知らしめるためのものである．

```
cl-user(20): (defconstant +val4+ 4)
+val4+
cl-user(21): (constantp '+val4+)
t
cl-user(22): (setq +val4+ 'four)
Error: Cannot setq +val4+ -- it is a constant.
  [condition type: program-error]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(23): :continue 0
cl-user(24):
```

大域変数は動的変数

defvar などの大域変数も前述の特殊変数と同じく動的スコーピングである．しかもそれは大域的である．動的スコーピングであることを利用した特殊なコードを紹介する．プログラム中において過去はどういう値の変遷をたどってきたか知らないけれど，今現在はこの値を使いたい，そして今後どう使われるか分からないけれど，使用後はその元の値に自動的に戻しておきたい，あるいはとにかく大域変数を副作用なしで使いたい，というような場合には，let 構文中で大域変数（特殊変数）に大域変数の値を束縛するという使い方がしばしばされる．たとえば標準入出力ストリームである *standard-input* , *standard-output* や，現在の名前空間を示す *package* などは，大域変数であるが，次のようなコードをしばしば見かける．パッケージについては第20章で学ぶが，関数 intern の二番目の引数が省略されたときには，カレントパッケージがその規定値となるということだけここでは述べておく．

```
cl-user(25): (defun foo ()
               (let ((*package* *package*))
                 (setq *package*
                       (make-package :my-package))
                 (intern "foo")))
foo
cl-user(26): *package*
#<The common-lisp-user package>
cl-user(27): (foo)
my-package::foo
nil
cl-user(28): *package*
#<The common-lisp-user package>
```

ここでは実施例25の let 構文に注目されたい．let 変数 *package* に *package* の値を束縛するという一見不自然なコードに見えるかもしれないが，*package* は動的束縛なので，let 変数の値としてとられるもの

（2番目の *package* の値）は，関数 foo 実行時の *package* の値である．それを let 変数である *package* に束縛している．この関数が実行されるとき，これで *package* のスタックにそのときのこの変数とその値が動的に積まれることになる．そして *package* の値がプログラム中で変更されるが，関数実行から抜けるとすなわちこの let 構文を抜けると，スタックはとりのぞかれて，前の *package* の値が見えるようになる．もしこのいささか不自然な let 構文が使えないと，関数 foo 実行時に本当に大域変数の *package* の値を変更することになってしまい，関数が終了してもその副作用が残ってしまう．大域変数の値を副作用なしで保全するにはスマートなコードである．

第5章 Lisp の内幕

これまで，なるべく副作用がないプログラムコードを紹介してきた．ここからは一歩 Lisp の中に入って，副作用アリアリのコードを紹介する．と言っても紹介するのはあくまでも Lisp の実装をモデル化したセルとポインタのモデルである．大域変数は副作用を前提としているが，ラムダ変数や let 変数を使う限りでは，いわゆる変数の参照透明性が確保される．副作用はバグの温床となりやすいため，普通は参照透明性を重視してプログラムするが，どうしても計算速度が遅くなりやすい．ソフトウェアの世界では「プログラムの処理にかかる時間の80%はコード全体の20%の部分が占める」という経験則があるが，プログラムの実行速度が問題となったとき，効率を考えてプログラムを組むには，どうしてもこの20%の部分について副作用も利用したプログラミングが必要であり，そのためにはここで紹介するセルとポインタのモデルでの仕事を理解しなければならない．

5.1 consセル

試しに各自の Lisp 処理系で (room t) と打ち込んでみよう．結果は処理系依存だが，いずれにしても下のような結果が得られるはずである．

```
cl-user(1): (room t)
area area  address(bytes)        cons         other bytes
  #  type                    16 bytes each
                              (free:used)      (free:used)
     Top #x100b90000
     New #x100a80000(1114112)    0:4391      579648:375664
     New #x100970000(1114112)   -----           -----
   0 Old #x100001230(9891280)  496:80624    2158912:6380272
Root pages: 118
  Lisp heap:    #x100000000  pos: #x100b90000 resrve: #x10f000000
Aclmalloc heap: #x8000000000  pos: #x800000c000 resrve: #x8001000000
 Pure space:    #xff640000  end: #xffff6dc0

code   type                               items     bytes
112: (simple-array t)                      8105   2733344 33.8%
  1: cons                                 85522   1368352 16.9%
 11: symbol                               20152   1289728 15.9%
136: function                              9586   1203168 14.9%
133: sv-vector                            20344    675456  8.4%
119: (simple-array code)                    185    223392  2.8%
140: standard-instance                     3784    121088  1.5%
117: (simple-array character)              1671    117856  1.5%
137: closure                               2618     91264  1.1%
143: structure                             1023     82560  1.0%
126: (simple-array (unsigned-byte 16))      145     76560  0.9%
125: (simple-array (unsigned-byte 8))         7     49536  0.6%
120: (simple-array fixnum)                    4     14224  0.2%
138: hash-table                             109     10464  0.1%
127: (simple-array (unsigned-byte 32))        4      7568  0.1%
135: symbol-locative                        202      6464  0.1%
111: (short-simple-array foreign)            59      3744  0.0%
 18: bignum                                  90      2944  0.0%
 16: single-float                           153      2448  0.0%
```

```
 17: double-float                          123      1968  0.0%
 96: (short-simple-array t)                 48      1536  0.0%
130: mv-vector                               5       880  0.0%
240: (simple-array (unsigned-byte 64))       3       416  0.0%
118: (simple-array bit)                     12       416  0.0%
 20: complex                                11       352  0.0%
 80: (array t)                               7       336  0.0%
193: short-array                             5       240  0.0%
139: readtable                               7       224  0.0%
241: (simple-array (signed-byte 64))         1       176  0.0%

 total bytes = 8086704
aclmalloc arena:
    max size    free bytes    used bytes         total
          96          2688           384          3072
        2016          2016          2016          4032
        5088         15264         25440         40704
 total bytes:        19968         27840         47808
cl-user(2):
```

ここで code 1 のところに cons というタイプがある．全部で 16.9% のメモリスペースをとっているが，それが今ここで説明しようとしている cons セルである．cons セルというのは，二つのポインタからできているが，上記例では 1 cons セル当たり16バイト，1ポインタ当たり8バイトから成っているということが分かる．8 byte × 8 bit = 64 bit で，余談だがこのマシンは 64 bit マシンであることが分かる．もちろんこの値はマシンのみならず Common Lisp処理系依存であり，マシンの1ワード長が Lisp の1ポインタ用アドレスだとは必ずしも言えないが（特に昔はMC68000系でそうだった），同じことを32 bit マシン上のACLで行うと，1ポインタ当たり4バイトということが分かる．

さて cons セルは Lisp のリストの基本構成要素である．昔の Lisp には構造体などがなく，文字列と cons セルしかなくて，すべてをこれで構成した(Lisp 1.5 では配列(array)はあったようだが)．だから cons セルは文字通り Lisp の細胞みたいなものだった．cons セルを Lisp の教科書では次のような絵で説明する．

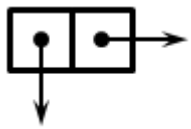


図5.1 cons セル

矢印はポインタを表しているが，左のポインタを car，右のポインタを cdr と言う．リストはこのような cons セルの連なりである．たとえば (this is a list) は次のようにモデル化できる．

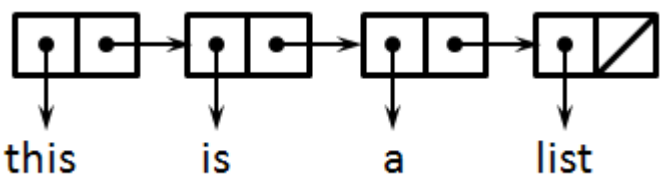


図5.2 cons セルによるリスト

最後の斜めに線の入った箱は nil を表す．その実装としてはほとんどの場合オール0ビットのメモリである．car ポインタの先はシンボルになっているが，(setq example1 '(this is a list)) みたいなことをすると，こんな具合になる．

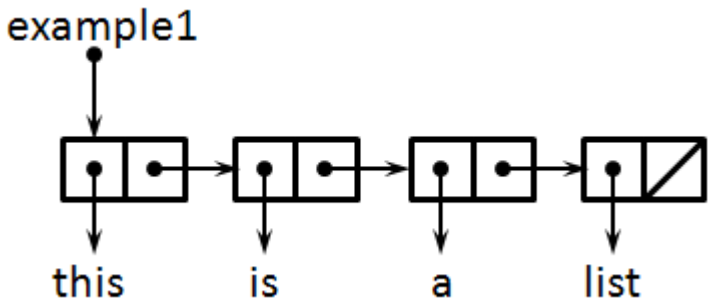


図5.3 (setq example1 '(this is a list)) 実行結果

シンボル example1 から出ている矢印は，シンボルの持つ value ポインタである．(setq example1 '(this is a list)) と言っただけでこれだけの結果を残すためには，必要であれば Lisp 処理系はexample1, this, is, a, listなどのシンボルを作りだして，それらをこのように consセルでつなぎ，example1 の value ポインタがそのリストの先頭を指すようにする．だから，シンボルを評価するというのは，そのシンボルの value ポインタを取り出すことであるし，リストの car を取るというのは，リストの先頭のセルの car ポインタを取り出すということであるし，リストの cdr を取るというのは，リストの先頭のセルの cdr ポインタを取り出すということである．

```
cl-user(3): (setq example1 '(this is a list))
(this is a list)
cl-user(4): example1
(this is a list)
cl-user(5): (car example1)
this
cl-user(6): (cdr example1)
(is a list)
```

リストの入れ子構造はこのような cons セルの car がまた別の cons セルをポイントするということで実現される．たとえば，次のようにしたとき，

```
cl-user(7): (setq example2 '((this is a list) (of two)))
((this is a list) (of two))
cl-user(8): (car example2)
(this is a list)
cl-user(9): (car (cdr example2))
(of two)
```

このとき，処理系内部では次のような構造が実現されている．

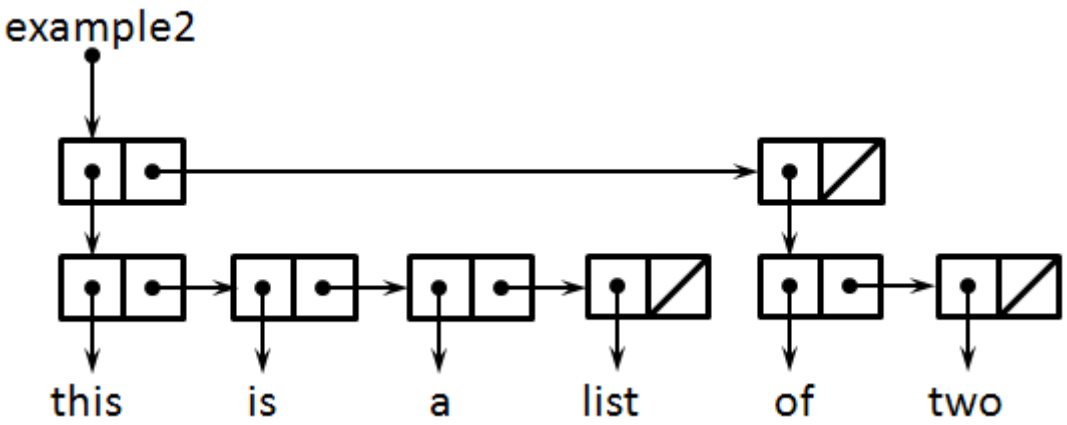


図5.4 リストのリスト構造

関数の cons というのは，上記 cons セルの未使用のプールから一つのセルをとってきて，その car が第1引数をポイントするようにし，cdr がその第2引数をポイントするようにすることである．

```
cl-user(10): (setq example3 (cons 'it (cdr example1)))
(it is a list)
cl-user(11): example3
(it is a list)
```

問題5.1 上記実施例10の実行経過と結果のセルとポインタの構造を示せ．

問題5.2 第3章で定義した our-member のコードから，なぜ答えがその要素を先頭に含むリストになるのかを考え，それをセルとポインタの構造から説明せよ．

関数 cons の実行時にその第2引数には多くの場合リストが与えられるが，リストでないと駄目なのかというとそんなことはない．ただし，そうするとリストのような拡張の柔軟性はなくなって，できることは限られてくるが，逆に key-value ペアを作りたいというような場合には，積極的に cons の第2引数にリスト以外の値を与えることもできる．

```
cl-user(12): (setq key-val-1 (cons 'key1 'val1))
(key1 . val1)
cl-user(13): (setq key-val-2 (cons 'key2 'val2))
(key2 . val2)
cl-user(14): (setq key-val-3 (cons 'key3 'val3))
(key3 . val3)
cl-user(15): (setq list-of-key&values
                   (list key-val-1 key-val-2 key-val-3))
((key1 . val1) (key2 . val2) (key3 . val3))
cl-user(16): (assoc 'key3 list-of-key&values)
(key3 . val3)
```

このように，cdr が cons セルではないときの構造をその表示方法から dotted pair と言う．しかし，ドットを使って通常のリスプ構造を入力することだってできる．次の例を見られたい．

```
cl-user(17): (cons 'this
                   (cons 'is (cons 'a (cons 'list nil))))
```

```
(this is a list)
cl-user(18): '(this . (is . (a . (list))))
(this is a list)
```

5.2 rplaca，rplacd，そして nconc

さて，いよいよあたかも Lisp の脳内に手をいれて手術するような関数を説明しよう．普段の Lisp の使い方では副作用を起こさないような関数群のみを使うようにするのが正しい態度なのであるが，後々人工知能プログラミングで紹介する queue を理解するには，どうしても rplaca，rplacd，nconc などの危険な関数を理解する必要がある．また，これらの関数を含めて，cons セルの取り扱いに関して各関数のすることを理解すれば，副作用なく効率的なプログラムを組むこともできる．rplaca，rplacd，nconc の説明の前に，copy-list や append のすることを説明する．

第3章において，我々の copy-list を定義した．再掲すると以下のとおりである．Lisp 処理系が行うことも全く同じである．

```
(defun our-copy-list (list)
  (cond ((endp list) nil)
        (t (cons (car list)
                 (our-copy-list (cdr list))))))
```

関数 cons は consセルを一つ取ってくる．だから次の実施例のようにすると，その結果は図5.5のようになる．

```
cl-user(19): (setq example1 '(this is a list))
(this is a list)
cl-user(20): (setq foo (copy-list example1))
(this is a list)
cl-user(21): (eq example1 foo)
nil
cl-user(22): (eq (last example1) (last foo))
nil
cl-user(23): (eq (car (last example1)) (car (last foo)))
t
```

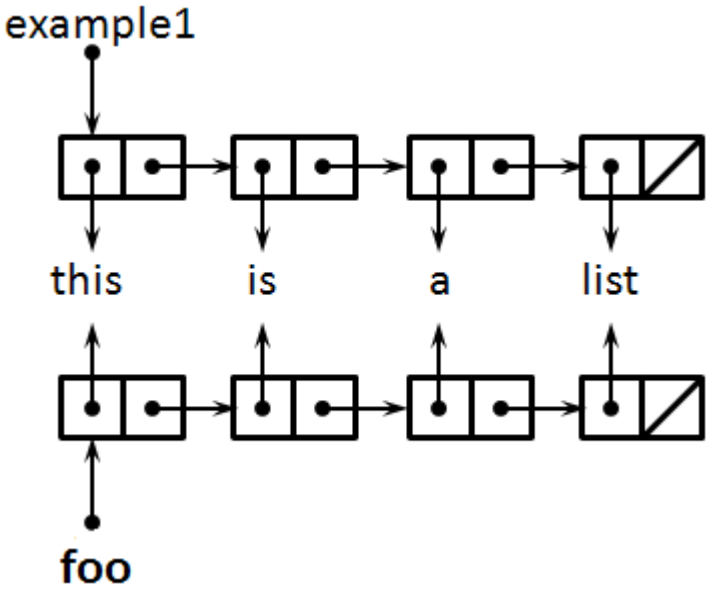


図5.5 リストのコピー結果

行番号21と22においてとりあげて比較するポインタは別物なので，eq の結果は nil となる．同様に append では最後の引数以外について，cons セルのコピーをとり，コピーされた最後の nil ポインタを次の引数の値にポイントするように変更する．

```
cl-user(1): (setq foo '(a b c))
(a b c)
cl-user(2): (setq bar '(x y z))
(x y z)
cl-user(3): (setq foobar (append foo bar))
(a b c x y z)
cl-user(4): foo
(a b c)
cl-user(5): bar
(x y z)
cl-user(6): (eq (cdddr foobar) bar)
t
```

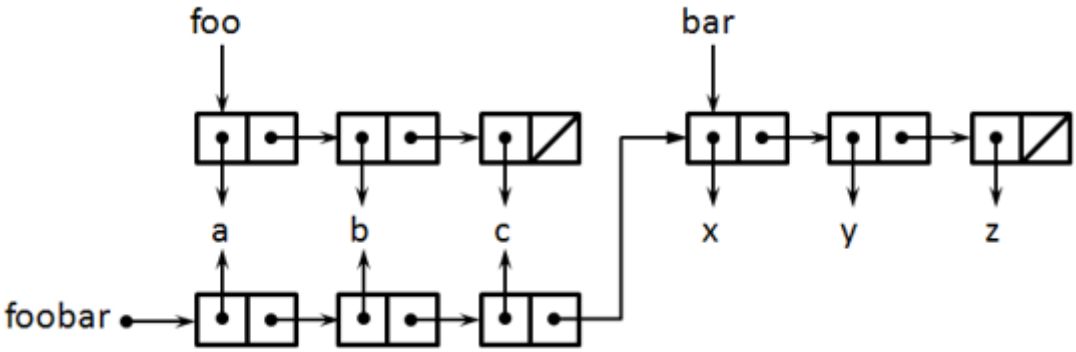


図5.6 リストの append 結果

cdddr というのは，cdr を3回施すことと同じである．この実行例では，(cdddr foobar) により得られるポインタの指すものは，bar が指すものと同じであることが分かるであろう．

copy-list でも append でも，その引数の値は変化しない．ところが nconc がすることは，一見得られた結果は append のそれと似ているが，実は副作用を伴っていて，引数の値がリストとしては異なったものになってしまう．次の実行例を見られたい．

```
cl-user(9): (setq foobar (nconc foo bar))
(a b c x y z)
cl-user(10): foobar
(a b c x y z)
cl-user(11): foo
(a b c x y z)
cl-user(12): bar
(x y z)
cl-user(13): (eq foo foobar)
t
```

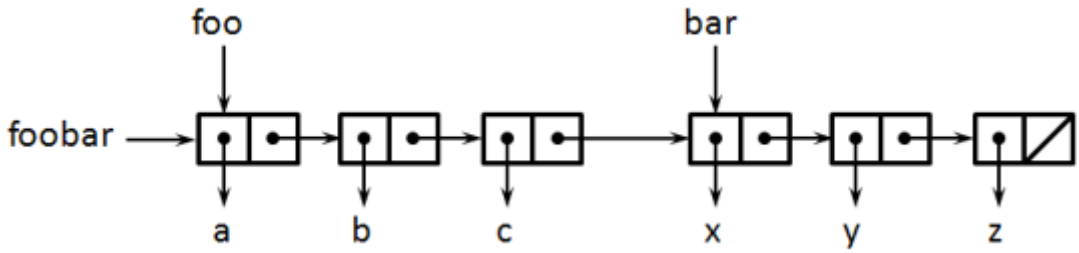


図5.7 リストの nconc 結果

nconc は任意個のリストを引数にとり，最後以外のリストの nil ポインタを破壊的に変えて，リストをつなげてしまう．だから，foo の値はこの実行例では foobar と同じことになり，リストとして見ると foo の値は変わってしまう．

同様に，cons セルの car ポインタを書き変えてしまうのが rplaca であり，cdr ポインタを書き変えてしまうのが rplacd である．

```
cl-user(14): foobar
(a b c x y z)
cl-user(15): (rplaca foobar 'AA)
(AA b c x y z)
cl-user(16): (rplaca (cdr foobar) 'BB)
(BB c x y z)
cl-user(17): foobar
(AA BB c x y z)
cl-user(18): (rplacd (cdr foobar) '(CC XX YY ZZ))
(BB CC XX YY ZZ)
cl-user(19): foobar
(AA BB CC XX YY ZZ)
```

問題5.3 上記行番号15から19までの経過をセルとポインタの構造から説明せよ．

5.3 破壊的関数

このように，nconc や rplaca，rplacd のようなリスト構造を変えてしまうような関数を使うときは，その副作用が悪さをしないよう注意しなければならない．通常はその引数がほかのどこからもポイントされていないことが明らかな場合に限って使うのが正しい使い方である．Common Lisp には副作用のない関数と共に，ほぼ同じことをするのであるが，副作用を持つ関数がしばしば用意されている．たとえば，remove には副作用はないが，delete には副作用がある．reverse には副作用はないが nreverse には副作用がある．copy-list 直後やクォートしたリストを明示的に与えた場合のように，他では絶対にそのリストがポイントされていないことが明らかな場合には，無駄な cons セルを消費することがないので，効率的なコーディングのために副作用ありの関数を積極的に使うことができる．誤解しやすいのは関数 sort である．これには副作用がある．便利な関数であるがソートされると元のリスト構造は変化してしまう．安全だと分かっていなければ，通常は copy-list した結果を sort に与える．sort と copy-list を一緒にした `sort*` を定義して使う人もいる．

破壊的関数を使うときでも，他のプログラミング言語とは異なり，常に関数の返値を使うようにコーディングしなければならない．以下の例を参照されたい．

```
cl-user(3): (setq foo '(a b c d))
(a b c d)
cl-user(4): (delete 'a foo)
(b c d)
cl-user(5): foo
(a b c d)
```

fooの値は変わらない．何故だろうか．実はdeleteの実装はおよそこんな具合になっている．

```
cl-user(6): (defun our-delete (item list)
              (setq list (cons 'dummy list))
              (%our-delete item list)
              (cdr list))
our-delete
cl-user(7): (defun %our-delete (item list)
              (cond ((null (cdr list)) nil)
                    ((eq item (second list))
                     (rplacd list (cddr list)))
                    (t (%our-delete item (cdr list)))))
%our-delete
```

内部関数%our-deleteの中で，2番目のアイテムが探しているものだと分かると，1番目のconsセルについてrplacdでそのlistの2番目のconsセルを飛ばして3番目のconsセルにつなげてしまう．これでlistの中間にあるアイテムは削除されるのだが，問題は探すべきものが一番最初にあった場合だ．だから外部関数our-deleteではわざわざ先頭にダミーの要素を追加して，その代わりに返す値はcdrの値にしているわけだが，返す値はこれで正しくても，もともとのシンボルfooにセットされていたポインタは削除されない．

```
cl-user(8): (setq foo '(a b c d))
(a b c d)
cl-user(9): (our-delete 'c foo)
(a b d)
cl-user(10): foo
(a b d)
cl-user(11): (setq foo '(a b c d))
(a b c d)
cl-user(12): (our-delete 'a foo)
(b c d)
cl-user(13): foo
(a b c d)
```

この状態でのセルの構造を下図に示す．

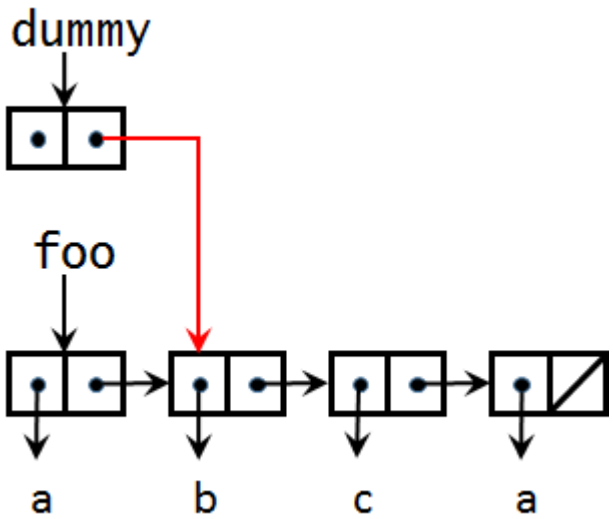


図5.8　先頭要素をdeleteしたときのセル構造

だからほとんどの破壊的関数において，返値を使うようにコーディングされなければならない．そのことはnreverse関数やsort関数においても同様である．

```
cl-user(14): (setq foo (delete 'a foo))
(b c d)
cl-user(15): foo
(b c d)
cl-user(16): (nreverse foo)
(d c b)
cl-user(17): foo
(b)
cl-user(18): (setq foo '(b c d))
(b c d)
cl-user(19): (setq foo (nreverse foo))
(d c b)
cl-user(20): foo
(d c b)
cl-user(21): (sort foo #'(lambda (x y) (string< x y))
                    :key #'symbol-name)
(b c d)
cl-user(22): foo
(d)
```

```
cl-user(23): (setq foo '(d c b))
(d c b)
cl-user(24): (setq foo
                   (sort foo #'(lambda (x y) (string< x y))
                         :key #'symbol-name))
(b c d)
cl-user(25): foo
(b c d)
```

問題5.4 問題 3.8ではinsert-rightを非破壊的に定義した．破壊的関数を使ってinsert-rightを定義せよ．

問題5.5 問題 3.7ではinsert-leftを非破壊的に定義した．破壊的関数を使ってinsert-leftを定義せよ．(hint: 左に入れるためには一個手前のセルポインタが必要である．)

問題5.6 上記行番号17でなぜそのような結果になるかは，自分でプログラムしてみれば分かる．破壊的リスト処理関数を使って our-nreverse を定義せよ．

5.4 シンボルとキーワード

ここでもう少し Lisp のシンボルとは何かを説明しておこう．次の実施例を見られたい．

```
cl-user(3): (inspect 'a-symbol)
A new symbol a-symbol @ #x100ae4e1b
  which is an internal symbol in the common-lisp-user package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #x00
   2 package ------> The common-lisp-user package
   3 value --------> ..unbound..
   4 global-value -> ..unbound..
   5 function -----> ..funbound..
   6 hash ---------> Bit field: #xad1d7017
   7 name ---------> A simple-string (8) "a-symbol"
   8 plist --------> symbol nil
   9 lock-index ---> symbol nil
[1i] cl-user(4): :i q
a-symbol
```

第4章でも述べたが，Lisp のシンボルは Java や C++ などの変数とは異なって，単なるプログラムコード上の存在（識別子）ではなく，REPL 環境中に存在する物理的なオブジェクトである．上記実施例のようにシンボルには多くのスロットがあるが，これまでも出てきた symbol-value，symbol-function のほかに，Common Lisp では属性リスト (property list, plist)，印字名 (print name)，パッケージセル (package cell) があって，それぞれ symbol-plist，symbol-name，symbol-package という関数で中身を取り出すことができる．

```
cl-user(7): (symbol-plist 'a-symbol)
nil
cl-user(8): (symbol-name 'a-symbol)
"a-symbol"
cl-user(9): (symbol-package 'a-symbol)
#<The common-lisp-user package>
```

属性リスト

属性リストはLispに当初から存在した機能で，昔のLispでは簡単なデータベース的な使い方に多用されたが，ハッシュが用意された現在では，ほとんど使われなくなってしまった．しかし，シンボルとシンボルを直接リンクするには便利かもしれない．

シンボルの属性を得るには関数getを使い，属性を与えるには次の例に示すように，5.5節で説明するsetfとgetを使う．

```
cl-user(4): (setf (get 'George 'sex) 'male)
male
cl-user(5): (setf (get 'Wendy 'sex) 'female)
female
cl-user(6): (get 'George 'sex)
male
cl-user(7): (get 'Wendy 'sex)
female
```

問題5.7 次のような男女のグループがある．この各メンバーに対して，上記のように男女の属性を付与せよ．

```
> (setq boys&girls
        '(George Wendy Sala John Steve Naomi))
```

問題5.8 複数の男女を含むリストから男性を取り出す関数 `make-boys-group-from` と女性を取り出す関数 `make-girls-group-from` を定義せよ．

問題5.9 並び順が不定な複数の男女を含むリストから男女を交互に並べる非破壊的関数 `boys-meet-girls*` を定義せよ．

問題5.10 並び順が不定な複数の男女を含むリストから男女を交互に並べる破壊的関数 `boys-meet-girls` を定義せよ．

パッケージ

パッケージについては第20章でもシステム構築の観点から述べるが，ここではシンボルとの関係に注目して述べる．新しいシンボルを生成するには，通常はREPL上でそのシンボルを（クォートをつけたりつけなかったりして）ただ単にタイプ入力すればよい．Lispリーダはシンボルらしきものを読み取ると，それを印字名とするシンボルをカレントパッケージ中に見つけることができれば（直接存在しているかあるいは異なるパッケージから継承されていれば），その実体を取り出してEVALに渡すが，なければ新しいシンボルをその時点で生成し，カレントパッケージに登録する．REPL上でタイプ入力するかわりに，以上のこと（パッケージ中でのシンボル検索と登録）を関数internで明示的に行うこともできる．メモリ番地から成るIDとvalueの値に注目しながら以下の実施例を見られたい．

```
cl-user(10): (setq new-thing nil)
nil
cl-user(11): (inspect 'new-thing)
A new symbol new-thing @ #x100adcbdb
  which is an internal symbol in the common-lisp-user package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #x00
   2 package ------> The common-lisp-user package
   3 value --------> symbol nil
   4 global-value -> symbol nil
   5 function -----> ..funbound..
   6 hash ---------> Bit field: #x600dc005
   7 name ---------> A simple-string (9) "new-thing"
   8 plist --------> symbol nil
   9 lock-index ---> symbol nil
[1i] cl-user(12): :i q
new-thing
cl-user(13): (intern "new-new-thing")
new-new-thing
nil
cl-user(14): (inspect 'new-new-thing)
A new symbol new-new-thing @ #x100aeb8db
  which is an internal symbol in the common-lisp-user package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #x00
   2 package ------> The common-lisp-user package
   3 value --------> ..unbound..
   4 global-value -> ..unbound..
   5 function -----> ..funbound..
   6 hash ---------> Bit field: #xf33ced71
   7 name ---------> A simple-string (13) "new-new-thing"
   8 plist --------> symbol nil
   9 lock-index ---> symbol nil
[1i] cl-user(15): :i q
new-new-thing
cl-user(16): (setq new-new-thing nil)
nil
cl-user(17): (inspect 'new-new-thing)
A new symbol new-new-thing @ #x100aeb8db
  which is an internal symbol in the common-lisp-user package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #x00
   2 package ------> The common-lisp-user package
   3 value --------> symbol nil
   4 global-value -> symbol nil
   5 function -----> ..funbound..
   6 hash ---------> Bit field: #xf33ced71
   7 name ---------> A simple-string (13) "new-new-thing"
   8 plist --------> symbol nil
   9 lock-index ---> symbol nil
[1i] cl-user(18): :i q
new-new-thing
```

上記10行目では，新しいシンボルの生成と登録が自動的に行われているが，16行目では一旦13行目で生成登録されたシンボルが取り出されて使用されている．

内部シンボル，外部シンボル

関数 intern の正式な関数呼び出し形式は以下のようである．ついでに関数 find-symbol も覚えておこう．

```
(intern      《文字列》 [ 《パッケージ》 ] )
(find-symbol 《文字列》 [ 《パッケージ》 ] )
```

2番目の引数はオプショナルであり，省略した場合はカレントパッケージが用いられる．両者の関数は二つの値を返す．intern では最初の返値がシンボルであり，2番目の返値はそのシンボルの指定パッケージにおける状態を意味する以下のようなものである．find-symbol は，もしシンボルを発見できなければシンボルを生成しないで nil を返すことを除いては，intern と同様である．

- nil ——— intern 時でシンボルが新たに生成されて内部シンボルとなった，あるいはfind-symbol 時でシンボルを生成しないとき．
- :internal — シンボルは内部シンボルとして存在していた．
- :external — シンボルは外部シンボルとして存在していた．
- :inherited - シンボルはuse-packageを介して継承されていた．

あるパッケージで新たに生成されたシンボルは明示的に**移出**(export) されないかぎり，内部シンボルである．内部シンボルは移出されると，外部シンボルとなる．あるパッケージ中の外部シンボルは他のパッケージ中で use-package されることでそのパッケージに継承されて（所有はされていないが）そのパッケージにおいて利用可能となる．とはいえ，内部シンボルと外部シンボルの違いは，外部シンボルはほかのパッケージにおいても利用されることが最初から予想されているということだけである．以下の実施例を見られたい．（Allegro mlispならば上の例．SBCLならば下の例．）

```
cl-user(1): (defpackage pkg1)
#<The pkg1 package>
cl-user(2): (defpackage pkg2)
#<The pkg2 package>
cl-user(3): (intern "foo" :pkg1)
pkg1::foo
nil
cl-user(4): (export 'pkg1::foo :pkg1)
t
cl-user(5): (intern "foo" :pkg1)
pkg1:foo
:external
cl-user(6): (use-package :pkg1 :pkg2)
t
cl-user(7): (intern "foo" :pkg2)
pkg1:foo
:inherited
cl-user(8): (in-package :pkg1)
#<The pkg1 package>
pkg1(9): (find-symbol "foo")
foo
:external
pkg1(10): (in-package :pkg2)
#<The pkg2 package>
pkg2(11): (find-symbol "foo")
foo
:inherited
―――――――――――――――――
```

```
CL-USER(1): (defpackage pkg1)

#<PACKAGE "PKG1">
CL-USER(2): (defpackage pkg2)

#<PACKAGE "PKG2">
CL-USER(3): (intern "FOO" :pkg1)

PKG1::FOO
NIL
CL-USER(4): (export 'pkg1::foo :pkg1)

T
CL-USER(5): (intern "FOO" :pkg1)

PKG1:FOO
:EXTERNAL
CL-USER(6): (use-package :pkg1 :pkg2)

T
CL-USER(7): (intern "FOO" :pkg2)

PKG1:FOO
:INHERITED
CL-USER(8): (in-package :pkg1)

#<COMMON-LISP:PACKAGE "PKG1">
PKG1(9): (cl:find-symbol "FOO")

FOO
:EXTERNAL
PKG1(10): (cl:in-package :pkg2)

#<COMMON-LISP:PACKAGE "PKG2">
PKG2(11): (cl:find-symbol "FOO")

FOO
:INHERITED
```

ここで実施例6では use-package を用いて pkg2 が pkg1 を使用するように設定していることを注意しておく．

use-package することで継承されたパッケージの外部シンボルは，カレントパッケージにおいてシンボルパッケージを指定することなく利用することができる．一方，カレントパッケージ内部でアクセス可能ではない他パッケージの外部シンボルであっても pkg1:foo のように一つのコロンを伴ったパッケージ指定のシンボルとして表示されるし，どのカレントパッケージにおいてもそのシンボルを指定できる．さらに注意すべきは，通常のプログラミング言語の常識では，内部シンボルは外部からは一切アクセス不可能と思われるところであるが，Common Lisp では，他パッケージ内部シンボルでも，pkg1::foo のように，ダブルコロン「::」を付けることで指定できてしまう．

パッケージの利用とシンボルの移出の関係はややこしくて奥が深い．第20章では改めてパッケージの利用方法について述べる．

キーワードシンボル

これまでも先頭にコロン「:」のついた字句表記がしばしば登場してきたが，それらは特殊なシンボルであり，キーワードとして特殊な使い方に用いられる．

```
cl-user(19): ':new-keyword
:new-keyword
cl-user(20): (inspect ':new-keyword)
A new symbol :new-keyword @ #x100aef4fb
  which is an external symbol in the keyword package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #xc1
   2 package ------> The keyword package
   3 value --------> symbol :new-keyword
   4 global-value -> symbol :new-keyword
   5 function -----> ..funbound..
   6 hash ---------> Bit field: #x2a0f4ad9
   7 name ---------> A simple-string (11) "new-keyword"
   8 plist --------> symbol nil
   9 lock-index ---> symbol nil
[1i] cl-user(21): :i q
:new-keyword
cl-user(22): (intern "new-new-keyword" :keyword)
:new-new-keyword
nil
cl-user(23): (inspect ':new-new-keyword)
A new symbol :new-new-keyword @ #x100af141b
  which is an external symbol in the keyword package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #xc1
   2 package ------> The keyword package
   3 value --------> symbol :new-new-keyword
   4 global-value -> symbol :new-new-keyword
   5 function -----> ..funbound..
   6 hash ---------> Bit field: #xe48e6889
   7 name ---------> A simple-string (15) "new-new-keyword"
   8 plist --------> symbol nil
   9 lock-index ---> symbol nil
[1i] cl-user(24): :i q
:new-new-keyword
```

キーワードシンボルはシンボルには違いないが，上記のようにそれはキーワードパッケージに登録されるシンボルであり，しかもその値はそれ自身である．だからプログラム中では，t や nil のように，クォートをつけてもつけなくてもよい．パッケージなしでコロンから始まるシンボルは自動的にキーワードシンボルとなる．ただし，ACL のトップレベル（そしてSBCL でも sb-aclrepl をロードした場合）では，はだかの単独のキーワードが UNIX のようなコマンドとして解釈されるようになっているので注意されたい．

5.5 汎参照（汎変数）

これまで変数の値をセットするのに 古い Lisp のように setq しか用いてこなかった．一方，属性リストに値をセットするのには，setf を用いたが，昔はシンボルの属性リストに値をセットするのには putprop という関数を用いた．ところが Common Lisp では一般の変数という概念を拡張した汎変数(generalized variable)

なるものが導入されて，putprop が仕様からは除かれてしまった．汎変数(generalized varible)という用語も ANSI Common Lisp では汎参照(generalized reference)というもっとよく分からない名前に変わった．以下に *CLtL2*における汎変数の説明を引用する．

> Lisp では，変数は 1 つのデータ，すなわち 1 つの Lisp オブジェクトを覚えることができる．変数の上での主な操作は，そのオブジェクトを取り出すことと，またその変数に新しいオブジェクトを覚えるように変更させることである．これらの操作はしばしば，アクセス(access)操作と変更(update)操作と呼ばれる．シンボルで名づけられた変数という概念は，1 つのデータを覚えることができる任意の記憶場所へと一般化できる．これはその場所が名づけられているかどうかとは無関係である．そうした記憶場所の例は，コンスの car および cdr，配列の要素や構造体の要素などである．（井田監訳）

Lisp の内部構造に言及することなく汎変数を説明しようとすると，このように言うしかないが，セルとポインタ，シンボルの構造などを理解したあとでは，変数の（symbol-value で得られる）ポインタにアクセスしたり変更したりするのと同様に，およそ Lisp で得られるポインタについて変数と同様のやり方で値を取り出したり変更したりできるようにしたもの，というように理解することができる．

```
cl-user(11): (setq foo '(one two three))
(one two three)
cl-user(12): (setf bar '(one two three))
(one two three)
cl-user(13): foo
(one two three)
cl-user(14): bar
(one two three)
cl-user(15): (rplaca foo 1)
(1 two three)
cl-user(16): (setf (car bar) 1)
1
cl-user(17): foo
(1 two three)
cl-user(18): bar
(1 two three)
cl-user(19): (rplacd (cdr foo) '(3))
(two 3)
cl-user(20): (setf (cddr bar) '(3))
(3)
cl-user(21): foo
(1 two 3)
cl-user(22): bar
(1 two 3)
```

汎参照(汎変数)に値をセットする呼び出し形式は次のようである．

```
(setf {《場所》 《更新する値》}* )
```

ここで《場所》というのは要するに評価するとポインタが得られるような書式のことであるが，マクロ setf は正しく値をセットする書式に展開されなければならないから，《場所》に使用できる（ポインタを取り出す）オペレータはもちろんあらかじめ事前に決められている．そのほかに構造体やCLOSオブジェクトのスロットなども，ユーザが定義したスロットは自動的に setf 可能になる．また，そういう汎変数を受け

付けるマクロというのもいくつか決まっていて，push, pushnew, remf, incf, decf, rotatef, psetf, pop などではsetfを用いることなく，汎変数の値を更新することができる．

```
cl-user(23): (setf (nth 1 bar) 2)
2
cl-user(24): bar
(1 2 3)
cl-user(25): (setq bii #(one two three))
#(one two three)
cl-user(26): (setf (svref bii 1) 2)
2
cl-user(27): bii
#(one 2 three)
cl-user(28): (incf (second bar))
3
cl-user(29): (incf (svref bii 1))
3
cl-user(30): bar
(1 3 3)
cl-user(31): bii
#(one 3 three)
```

defsetf

もし自分で新しく汎変数を定義したかったら，値を得るためのアクセス関数を定義したのち，そのある「場所」がsetf されたときに展開されて正しく値が更新されるようなマクロ展開書式を defsetf で定義する．あるいはもっと簡単には，アクセス関数と更新関数の二つを定義してそれらを defsetf でまとめればよい．

```
(defsetf 《アクセスオペレータ》 《ラムダリスト》 (《保存変数》)
   {《ドキュメンテーション》}*
   {《書式》}* )
(defsetf 《アクセスオペレータ》 《更新オペレータ》 [《ドキュメンテーション》])
```

このマクロ展開書式が返す値は常に更新される値でなければならない．たとえば，アクセス関数 car の defsetf 書式は以下のようになる．

```
(defsetf car (list) (new-value)
  `(progn (rplaca ,list ,new-value) ,new-value))
```

defsetf の書き方はマクロの書き方に似ているが，マクロについては第 8 章にて詳述する．《アクセスオペレータ》には関数名かマクロ名がくる．

*CLtL2*になって，関数定義時の関数名として「(setf シンボル)」という表記を与えることで setf ができるようになった．たとえば，(setf cadr) と全く同様な (setf my-cadr) を定義するには，次のようにすればよい．

```
cl-user(1): (defun my-cadr (list) (cadr list))
my-cadr
cl-user(2): (defun (setf my-cadr) (new-value list)
```

```
              (rplaca (cdr list) new-value) new-value)
(setf my-cadr)
cl-user(3): (setq foo '(one two three))
(one two three)
cl-user(4): (setf (my-cadr foo) 2)
2
cl-user(5): foo
(one 2 three)
cl-user(6): (my-cadr foo)
2
```

第6章 高階関数と MAP 関数

6.1 関数は第1級のクラス

Common Lisp では Scheme と異なって，シンボルは値のほかに関数を持つことができる．シンボルからシンボルの値を取り出すには symbol-value が用いられるが，関数を取り出すには関数 symbol-function が用いられる．

```
cl-user(1): (symbol-value 'pi)
3.141592653589793d0
cl-user(2): (symbol-function 'cons)
#<Function cons>
cl-user(3): #'cons
#<Function cons>
cl-user(4): (function cons)
#<Function cons>
```

ここで #' は特殊形式である function のためのシンタックスシュガーである．

Lisp が関数型言語であるという最大の特徴は，関数それ自体を関数引数とすることができるところにある．Lisp では関数も処理される対象なのである．このように関数それ自体も Lisp のオブジェクトであることを，昔は *first class citizen* と言ったが，今ではこのような言葉が避けられて，「政治的に正しい」(politically correct)言い方として，第1級のオブジェクト(*first class object*)という言葉が使われる．引数に関数を取ることのできる関数は，高階関数と呼ばれる．

これまでの関数呼び出しと同じことを，funcall という関数と apply という関数に関数引数を与えて実行できる．

```
cl-user(5): (funcall #'cons 'a '(b))
(a b)
cl-user(6): (apply #'cons '(a (b)))
(a b)
cl-user(7): (apply #'cons 'a '((b)))
(a b)
```

ただしその使い方は異なっていて，funcall ではその関数の引数を一つづつ並べて与えるが，apply では引数の最後に残りの与えるべき引数をまとめてリストとして与えることができる．状況に応じて便利に使い分ければよい．

さて，関数を引数にとる関数は，Common Lisp には多くあって，他の言語よりも多様でコンパクトな，見やすいコードを提供することに成功している．以下の実施例を見られたい．

```
cl-user(8): (setq foo '(a b 1 2 c 3))
(a b 1 2 c 3)
cl-user(9): (member-if #'numberp foo)
(1 2 c 3)
cl-user(10): (find-if #'numberp foo)
1
```

```
cl-user(11): (remove-if #'numberp foo)
(a b c)
```

member-if は member と似ているが，1引数の述語をとって，その述語の結果でメンバーシップを判定する．find-if は find と似ているが，1引数の述語をとって，その述語を真とする要素を返す．remove-if は remove と似ているが，条件にマッチした要素をリストから削除する．条件にマッチした要素をリストから取り出すには，remove-if-not がよく用いられる．

```
cl-user(12): (remove-if-not #'numberp foo)
(1 2 3)
```

これらの例では関数 numberp を用いたが，ここにはもちろん任意の1引数述語を置くことができ，無名関数でもラムダ式として書くことができる．

6.2 ラムダ式再び

すでに 2.1 節にてラムダ式について簡単に説明したが，ここであらためて，ラムダ式(lambda expression)とは何かについてきちんと説明し，複雑な関数呼び出しとそのラムダリストについて，その構文をきちんと説明する．

ラムダ式とは最初の要素がlamdaであるようなリスト形式のことであり，それは関数として働く．以下の例を見られたい．

```
cl-user(2): (apply (lambda (list)
                     (append (rest list)
                             (list (first list))))
                   '(a b c d) nil)
(b c d a)
cl-user(3): (apply #'(lambda (list)
                       (append (rest list)
                               (list (first list))))
                   '(a b c d) nil)
(b c d a)
cl-user(4): (apply (function (lambda (list)
                               (append (rest list)
                                       (list (first list)))))
                   '(a b c d) nil)
(b c d a)
```

前節で述べたように，#'consは(function cons)と同じことである．したがって上記実施例の３と４では同じく関数オブジェクトがapplyに渡される．では実施例2は何かというと，実はANSI Common Lispではlambdaはマクロになっていて，それが返すものは(function (lambda ...))という特殊形式だ．だから実施例2ではマクロ展開された結果がapplyに渡されるので，実施例3あるいは4と同じことになる．

apply や funcall の第１引数に与えるのは，直接の lambda式だけではない．関数オブジェクトのほかに，グローバルな環境における関数名シンボルでもよいし，次のように，関数名や関数オブジェクトを与えるシンボルや let 変数でもよい．

```
cl-user(5): (apply 'cons 'a '(b c) nil)
(a b c)
cl-user(6): (setq foo 'cons)
cons
cl-user(7): (apply foo 'a '(b c) nil)
(a b c)
cl-user(8): (setq bar (lambda (car cdr) (cons car cdr)))
#<Interpreted Function (unnamed) @ #x1023d4c52>
cl-user(9): (apply bar 'a '(b c) nil)
(a b c)
cl-user(10): (let ((foo 'cons))
               (apply foo 'a '(b c) nil))
(a b c)
cl-user(11): (let ((foo (lambda (car cdr) (cons car cdr))))
               (apply foo 'a '(b c) nil))
(a b c)
```

問題6.1 手持ちの Common Lisp 処理系を使って，次の書式を実行せよ．結果は上記と同様に計算されるか，あるいはエラーとなるかどちらかだが，手元の処理系ではどうだったろうか．

```
(apply '(lambda (list)
          (append (rest list)
                  (list (first list))))
       '(a b c d) nil)
```

リストの第1要素で関数として働くには，ラムダ式を与えるときに関数名をクォートしないようにラムダ式もクォートしてはならない．REPLにおいてリストの第1要素の扱いとその他の扱いは異なるのだ．

```
cl-user(12): ((lambda (list)
                (append (rest list)
                        (list (first list))))
              '(a b c d))
(b c d a)
cl-user(13): ('(lambda (list)
                 (append (rest list)
                         (list (first list))))
              '(a b c d))
Error: Illegal function object:
       '(lambda (list) (append (rest list) (list (first list)))).
  [condition type: type-error]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(14): :continue 0
cl-user(15): (#'(lambda (list)
                 (append (rest list)
                         (list (first list))))
              '(a b c d))
Error: Illegal function object:
       #'(lambda (list) (append (rest list) (list (first list)))).
  [condition type: type-error]

Restart actions (select using :continue):
```

```
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(16):
```

関数とラムダ式の定義式は以下のとおりである．

```
(defun 《関数名》 《ラムダリスト》
  [ {《宣言》}* | 《ドキュメンテーション》]
  {《関数本体》}* )
(lambda 《ラムダリスト》
  [ {《宣言》}* | 《ドキュメンテーション》]
  {《関数本体》}* )
```

《ラムダリスト》が関数引数のリストなのであるが，この仕様が結構複雑であり，これをどう書けばいいかを以下で説明する．以下では &optional，&rest，&key と三つのラムダリストキーワードと呼ばれる字句が出てくるが，キーワードと呼ばれてもこの三つはキーワードシンボルではないので注意されたい．また，それらに続くパラメータ記述においてユーザが定義する「キーワード」という言葉が登場するが，それは多くの場合はキーワードシンボルではあるが，そうでなくてもよいことになっている．非常に紛らわしいので注意されたい．

付加パラメータ

関数呼び出し時に必ず与えなければならない引数を，必須パラメータと呼ぶ．一方，必須パラメータをすべて書いたあと，省略したり，付け加えたりできる引数を，付加パラメータと呼ぶ．以下の例を見られたい．

```
cl-user(12): (log 100.0)
4.6051702
cl-user(13): (log 100.0 10)
2.0
cl-user(14): (log 8.0)
2.0794415
cl-user(15): (log 8.0 2)
3.0
cl-user(16): (log 2.7182818284590452353602874713526624977572470936999595749669)
0.99999994
cl-user(17): (log (exp 1))
0.99999994
```

log は対数を計算するが，第2引数はそのときの底であり，省略したときはネイピア数 e となって自然対数を計算し，10を与えたときは常用対数を計算する．

自分で付加パラメータを有する関数を定義するときは，ラムダリストを次のように書く．

```
( { 《必須パラメータ》 }*
    &optional { 《付加パラメータ》 |
               (《付加パラメータ》 [《規定値》 [《供給フラグ》]]) }* )
```

すなわち，必須パラメータ並びの次に &optional と書き，付加パラメータを並べる．関数呼び出し時の実引数は，必須パラメータとの対応が取られたのち，順番に付加パラメータ並びとの対応が取られるので，中間を飛ばして後ろのパラメータに値を対応させることはできない．対応する実引数が与えられないパラメータには，規定値が定義されていればそれが，定義されていなければ nil が与えられる．もし供給フラグがあれば，そこには実引数が与えられたときには真が，さもなければ偽が与えられる．

butlast は実は付加パラメータとして，最後の部分で除く要素の数を与えることができる．我々の butlast を書けば次のようなものになるであろう．

```
(defun our-butlast (list &optional (n 1))
  (%our-butlast list n (length list)))

(defun %our-butlast (list n len &optional butlast)
  (cond ((= n len) butlast)
        (t (%our-butlast (cdr list)
                         n
                         (1- len)
                         (append butlast (list (car list)))))))
```

our-butlast の内部関数として %our-butlast が定義されている．our-butlast の呼び出し時に list に (a b c d) を与えたとき，(length list) は4である．%our-butlast が最初に呼び出されるときにはその付加パラメータbutlastが与えられていないので，最初の呼び出し時にはその値は nil である．%our-butlast は再帰するが，その最初のときには (append nil (list (car list))) の結果として (a) がパラメータ butlast に与えられる．一方，その時 list には (cdr list) の値として (b c d) が与えられる．かくして list の頭から順に要素が butlast に移されるたびに，len が1づつ減じられ，その値が n の値と等しくなったとき，再帰が止まって蓄積された butlast の値が一気に返される．

```
cl-user(23): (our-butlast '(a b c d) 2)
 0[5]: (%our-butlast (a b c d) 2 4)
   1[5]: (%our-butlast (b c d) 2 3 (a))
     2[5]: (%our-butlast (c d) 2 2 (a b))
     2[5]: returned (a b)
   1[5]: returned (a b)
 0[5]: returned (a b)
(a b)
```

問題6.2 関数 last は実は付加パラメータとして最後から取り出す要素の個数を指定できる．その規定値は1である．それと似ているが先頭から任意個の要素を取り出す firsts を，butlast と付加パラメータを使って定義せよ．それは次のようなものである．

```
(firsts '(1 2 3 4 5 6 7 8 9 10)) -> (1)
(firsts '(1 2 3 4 5 6 7 8 9 10) 3) -> (1 2 3)
```

レストパラメータ

関数 + や list のように，任意個のパラメータを受け取りたいときには，レストパラメータを使う．

```
( {《必須パラメータ》}* &rest 《レストパラメータ》 )
```

レストパラメータを使えば，我々の list 関数は驚くほど簡単になる．

```
(defun our-list (&rest list) list)
```

しかし，このようなコードはラムダ式の実装によっては容易に副作用の問題を引き起こす．上記例で言えば，値として返されるlistはもとのリスト構造のコピーかも知れないし，そうでないかも知れない．それは処理系依存である．仕様書的に言えば，ユーザはレストパラメータによって受け取るリストの構造がもとの構造のコピーであるとも，コピーではないとも期待してはならないとされている．

キーワードパラメータ

1個からたかだか2,3個のパラメータなら付加パラメータとしても間に合うが，それが多くなると中間飛ばしをしたいという要求に合わなくなる．そういう場合にはキーワードパラメータにすると，多くのパラメータから必要なものだけを指示して与えることができる．典型的な例はファイルをオープンする関数であるが，ここでは remove 関数を紹介する．remove 関数をもし定義するとすると，そのラムダパラメータは意味的にはこんな具合である．

```
(defun remove (item sequence
               &key (from-end nil) (test #'eql) (test-not nil) (start 0)
                    (end (length sequence)) (count nil) (key #'identity)))
   ... )
```

関数 remove は，列 sequence からテスト条件を満足する item を取り除く．こんなに沢山あるキーワードパラメータがどう働くかを説明すると，テスト条件にはキーワードパラメータ test で与えられる述語が用いられる．その規定値は #'eql である．from-end はもしそれが真ならば，remove 操作がリスト末尾から実施されるが，さもなければリストの先頭から行われる．規定値は nil だから，通常は先頭からである．start と end で列全体のうちの対象となる範囲を指定できる．規定値は 0（先頭）と (length sequence) だから列の全体である．count は取り除く個数を指定する．これが nil というのは，いわば無限個，あるだけすべてという意味である．key には1引数の関数を与えて，sequence 中の各要素に適応されて本当に検査対象とするものを取り出す関数を与える． #'identity というのは，与えられた引数をおうむ返しにかえす関数である．実際に少し実施例を示してみる．ただし，キーワードパラメータというとおり，パラメータを指定するのにキーワードで指定する．

```
cl-user(1): (identity 'a)
a
cl-user(2): (setq foo '(1 2 3 a 4 5 a 6))
(1 2 3 a 4 5 a 6)
cl-user(3): (remove 'a foo)
(1 2 3 4 5 6)
cl-user(4): (remove 'a foo :count 1)
(1 2 3 4 5 a 6)
```

```
cl-user(5): (remove 'a foo :count 1 :from-end t)
(1 2 3 a 4 5 6)
cl-user(6): (remove 'a foo :start 0 :end 4)
(1 2 3 4 5 a 6)
cl-user(7): (remove 'a foo :start 4)
(1 2 3 a 4 5 6)
cl-user(8): (remove '(this is list) '(1 2 3 (this is list) 4 5))
(1 2 3 (this is list) 4 5)
cl-user(9): (remove '(this is list) '(1 2 3 (this is list) 4 5)
                    :test #'equal)
(1 2 3 4 5)
```

test-not キーワードを使えば，反対の条件を指定できるが，これはANSI Common Lispでは非推奨(deprecated)とされた．remove-if-notを使えば同じことができる．

```
cl-user(10): (remove '(this is list) '(1 2 3 (this is list) 4 5)
                     :test-not #'equal)
((this is list))
```

key キーワードを使えば，構成された要素の内部についてでも作用させることができる．

```
cl-user(11): (remove 'four '((one 1) (two 2) (three 3) (four 4)
                     (five 5)) :key #'car)
((one 1) (two 2) (three 3) (five 5))
```

自分でキーワードパラメータを使って関数定義するときのために，その正式な書式を紹介しておく．

```
( { 《必須パラメータ》 }*
  &key { 《変数》 |
        ({《変数》 | (《キーワード》 《変数》)} [《初期値》 [《供給フラグ》]]
        ) }* )
```

初期値と供給フラグについては，付加パラメータと同様であるが，キーワードパラメータではパラメータ指示に使われるべきキーワードとプログラム中の変数を分けて指定できるので，キーワードと関係なく関数内部の変数名を自由に与えることができる．それが(《キーワード》 《変数》)である．これをただの《変数》とした場合には，変数名と類似のキーワードがもちいられる．すなわち

```
(defun remove (item sequence
               &key (from-end nil) (test #'eql) . . .
                    . . . ))
   ... )
```

というのは，

```
(defun remove (item sequence
               &key ((:from-end from-end) nil) ((:test test) #'eql) . . .
                    . . . ))
   ... )
```

というのと同じである．

付加パラメータとレストパラメータの組み合わせ

これらの必須パラメータ以外のパラメータを複数組み合わせることもできるが，ここでは比較的簡単な付加パラメータとレストパラメータの組み合わせについて説明し，それ以外の組み合わせについては読者の研究課題とする．

さて，以下の実施例を参照されたい．

```
cl-user(7): (defun foo (n &optional o &rest r)
              (format t "~%N: ~S" n)
              (format t "~%O: ~S" o)
              (format t "~%R: ~S" r))
foo
cl-user(8): (foo 1)

N: 1
O: nil
R: nil
nil
cl-user(9): (foo 1 2)

N: 1
O: 2
R: nil
nil
cl-user(10): (foo 1 2 3)

N: 1
O: 2
R: (3)
nil
cl-user(11): (foo 1 2 3 4)

N: 1
O: 2
R: (3 4)
nil
```

すなわち，必須パラメータ，付加パラメータ，レストパラメータの順に並べた場合，関数呼び出しにおいて並べた実引数は順番に割り当てられる．Practical Common Lispの作者であるPeter Seibelは付加パラメータとキーワードパラメータの併用は絶対に避けるべきだと述べている．

レストパラメータとキーワードパラメータの併用

Peter Seibelはレストパラメータとキーワードパラメータの併用において，パラメータの順序はレストパラメータの次にキーワードパラメータを置くと述べている．ここでもその指定に従ってやってみよう．

```
cl-user(5): (defun key-arg-test0 (x y &rest rest &key a b)
              (format t "~%rest: ~S" rest)
              (format t "~%x, y: ~S, ~S" x y)
              (format t "~%a, b: ~S, ~S" a b)
              (setq x (+ a b))
              (setq y (* a b))
```

```
              (list x y))
key-arg-test0
cl-user(6): (key-arg-test0 'foo 'bar :a 1 :b 2)

rest: (:a 1 :b 2)
x, y: foo, bar
a, b: 1, 2
(3 2)
cl-user(7):
```

ご覧のように，restで指定されたパラメータには，呼び出し時の必須パラメータ割り当て以外のフォーム，この場合キーワードパラメータとして与えたフォームが与えられた．一方，キーワードパラメータには期待通りの値が与えられた．

実はこのいささか無意味に思われる例をあえてあげたのは、後述のマクロ定義におけるラムダリストとの比較をしたいがためであった。

6.3 MAP 関数

先に再帰プログラムの説明（第3章）において，lat や member のように cdr で横方向に再帰する場合と，一般のリストでは加えて car で縦方向に再帰する場合があることを述べた．このうち横方向については map 関数に置き換えできることがしばしばある．リストの各要素に平等に関数を働かせるというのが map 関数のイメージである．map 関数には色々な関数があるが，その主な違いは各要素に関数を働かせたあとの処理にある．mapcar はリストの各要素に働いて，得られる結果は要素数が同じリストになる．

```
cl-user(1): (mapcar #'1+ '(1 2 3))
(2 3 4)
```

問題6.3 関数 square-list (問題3.4参照)を mapcar を使って再定義せよ．

先に 3.1節でコードを見せたcount-elements は再帰を使ったが，mapcar でも作ることができる．

```
cl-user(2): (defun count-elements (list)
              (apply #'+ (mapcar #'(lambda (e) 1) list)))
count-elements
cl-user(3): (count-elements '(a b c d e))
5
cl-user(4): (count-elements '())
0
cl-user(5): (+ )
0
```

リスト (a b c d e) の各要素がラムダ関数によってすべて 1 になって，リスト (1 1 1 1 1) が返され，それを加算して 5 が得られるというわけである．得られるリストに任意個の引数をとる + を適応するために apply が用いられていることに注意されたい．

プログラムのバグはループの始めとか終わりとか，極限のときに持ち込まれることが多いのだが，Lispはその辺がよくできていて感心してしまう．ここではmapcarは空リストに対してnilを返すとか，空リストに対する加算は0を返すなどの仕様がうまく機能している．

一般のツリー状のリスト中のアトムの数を数える関数count-atomsを問題 3.9で出題した．今度はその横方向にはmap関数で処理し，縦方向には再帰させるやりかたでプログラムしてみよう．

```
cl-user(6): (defun count-atoms (s)
              (cond ((null s) 0)
                    ((atom s) 1)
                    (t (apply #'+ (mapcar #'count-atoms s)))))
count-atoms
cl-user(7): (count-atoms '(1 2 (3 (4 5) 6)))
6
cl-user(8): (count-atoms '(() () (() (() ()) ())))
0
cl-user(9): (count-atoms '(nil nil (nil (nil nil) nil)))
0
```

mapcarの関数として自分自身を与えているが，その停止条件として`(null s)`と`(atom s)`が与えられている理由がお分かりだろうか．このコードはもしリスト要素がnilだったら，それはアトムではなく空リストと解釈するコードである．もし`(null s)`のcond節がないと，空リストもアトムと解釈して行番号8および9では6が返されることになる．第1章ではnilはアトムでもありリストでもあり，と書いたが，それをどう解釈するかはプログラマー次第である．

問題6.4 リストを引数とし，その中の数アトムの数を返すcount-numbersをmapcarを用いて定義せよ．

mapcarで得られるnilをつぶしたいときはmapcan (mappend) を使う

mapcarで得られるリストの構造は与える引数のそれと変わらない．要素が変わるだけである．リスト中の数の要素だけ取り出したいと思って，次のように書いたとしよう．

```
cl-user(16): (setq foo '(1 2 3 a b c 4 x))
(1 2 3 a b c 4 x)
cl-user(17): (mapcar #'(lambda (e)
                         (cond ((null e) nil)
                               ((numberp e) e)
                               (t nil)))
                       foo)
(1 2 3 nil nil nil 4 nil)
```

残念ながらmapcarではリスト構造は変わらない．ここで生じるnilを消したいときにはmapcarではなくmapcanを使う．ただし，mapcanは各要素をnconcでつなげてしまうので，つぶされるnilはよいとして，数を返す時にはあらかじめそれをリストにして返す．

```
cl-user(18): (nconc '(1) '(2) '(3) nil nil nil '(4) nil)
(1 2 3 4)
cl-user(19): (mapcan #'(lambda (e)
                         (cond ((null e) nil)
```

```
                        ((numberp e) (list e))
                        (t nil)))
                foo)
(1 2 3 4)
```

しかし，mapcan は副作用を伴う nconc を使うので，mapcan を使うときは慎重に注意しなければならない．上記では必要があって数をlistしたが，listする必要がなくむしろ括弧を一つ削除したい場合にもし要素をそのまま返してしまうと，他でもその要素をポイントしているとコードによっては副作用が生じてしまう．そのためにあとの人工知能プログラミングでは，mapcan に類似であるが nconc ではなく，append を使う高階関数 mappend を定義してこちらを標準的に使用する．

mappend の簡単な定義は次のとおりである．

```
(defun mappend (fn list)
  "Append the results of calling 《fn》 on each element of 《list》.
   Like 《mapcon》, but uses 《append》 instead of 《nconc》."
  (apply #'append (mapcar fn list)))
```

問題6.5リストを引数とし，その中のシンボル not をすべて取り除く関数 remove-not を mapcanを用いて定義せよ．（hint: 要素が not だったら，それを除きたいのだけれど，そういうコードは難しい．とりあえず nil を与えておいて返ってきたものを append すると nil はつぶれる．けれども not 以外のところはつぶれては困るから・・・）．

問題6.6 リストを引数とし，その中のアトムだけを取り出してならべたリストを返す関数 squash を mapcan を用いて定義せよ．(hint: ちなみに問題3.13の答えは次のとおりである).

```
(defun squash (s)
  (cond ((null s) nil)
        ((atom s) (list s))
        (t (append (squash (first s))
                   (squash (rest s))))))
```

6.4 その他の MAP 関数

some と every

リスト中に一つでも条件を満たすものがあるか調べる時は some 関数を，すべてが満たすかどうか調べる時は every 関数を使う．ただし，mapcar や mapcan と異なり，some と every は文字列のような列にも有効に使うことができることを注意しておく．問題 3.1 の is-there-num は some を使えば簡単に次のように書ける．

```
cl-user(11): (defun is-there-num (lat)
               (some #'numberp lat))
is-there-num
cl-user(12): (is-there-num '(a b five c))
nil
```

```
cl-user(13): (is-there-num '(a b 5 c))
t
```

問題 3.2 の is-lat は every を使って次のように書ける.

```
cl-user(14): (defun is-lat (lst)
               (every #'atom lst))
is-lat
cl-user(15): (is-lat '(a b five c))
t
cl-user(16): (is-lat '(a b () c))
t
cl-user(17): (is-lat '(a b (c) d))
nil
cl-user(18): (atom '())
t
```

空リスト「( )」は nil に等しく，ここではアトムと見做されることに注意されたい.

問題6.7 問題3.3 と同じであるが，リストを引数としてその要素すべてが数字なら t を返す関数 is-lnum を定義せよ.

MAP と REDUCE

少し前にビッグデータで流行ったMAP-REDUCEであるが，その原型は Common Lisp にあった．以下のように map 関数は文字列やリストを引数として，その各要素に引数として与えた関数を平等に適応する.

```
cl-user(1): (map 'list #'identity
                 "Change to a list of chars")
(#\C #\h #\a #\n #\g #\e #\Space #\t #\o #\Space ...)
cl-user(2): (map 'string #'identity *)
"Change to a list of chars"
cl-user(3): (map 'string #'(lambda (str) (char str 0))
                 '("common" "lisp" "-" "artificial"
                   "intelligence" "programming"))
"cl-aip"
```

ここで map 関数の第 1 引数には結果の型を指定することに注意されたい．上記例では，文字列が構成する各文字のリストに変換されたり，文字のリストが文字列に変換されたりした．文字と文字列については 12.5節にて詳述する.

一方，reduce では引数となる関数には二引数を受け付けることができる関数を与えるが，列の要素の最初の二つに対して適応したのちは，その結果と 3 番目の列要素を引数として関数適応し，さらに同様に 4 番目，5 番目と次々に演算する．最後に返される結果はそういう演算で縮約されたものとなる．以下の例を参照されたい.

```
cl-user(4): (reduce #'+ '(1 2 3 4))
10
cl-user(5): (reduce #'- '(1 2 3 4))
-8
cl-user(6): (reduce #'list '(1 2 3 4))
```

```
(((1 2) 3) 4)
cl-user(7): (reduce #'list '(1 2 3 4) :from-end t)
(1 (2 (3 4)))
```

キーワードパラメータ :from-end に真を与えると，関数の適応はリストの終わりから前に向かって行われる．

前述の mappend も reduce を使って次のように定義できる．ただし，このmappendは複数のリストを引き受けることができる．

```
(defun mappend (fn &rest lists)
  "Apply 《fn》 to respective elements of 《lists》, and append results."
  (reduce #'append (apply #'mapcar fn lists) :from-end t))
```

reduce にキーワードパラメータ :initial-value を使って初期値を与えると，最初にそれと1番目の要素が用いられる．

```
cl-user(8): (reduce #'+ '(2 3 4) :initial-value 1)
10
cl-user(9): (reduce #'cons '(1 2 3 4) :from-end t :initial-value nil)
(1 2 3 4)
```

この map と reduce 両方を組み合わせて，文書全文検索エンジン Lucene では単語の出現場所をインデックスとしたときの転置インデックスを生成する．本来であれば最初に形態素解析などで文書をパーズするのであるが，簡単のために今はパーズされた文書が与えられたものとしよう．たとえば，以下のとおりである．

```
cl-user(10): (setq doc1 '(さくら 学校 の ページ))
(さくら 学校 の ページ)
cl-user(11): (setq doc2 '(かえで 学校 の ページ))
(かえで 学校 の ページ)
```

単語「さくら」は文書 doc1 の最初(index 0)にしか現れないし，単語「かえで」は文書 doc2 の最初(index 0)にしか現れない．

関数 wordmap は文書IDを受け取ってそのコンテンツの語彙について，文書ID，単語，その単語のインデックスの三つ組みのリストを作り出す．ここでインデックスというのはある文書中に出現する単語の順番のことである．ここで用いられる loop マクロについては第17章で詳述する．

```
cl-user(12): (defun wordmap (docID)
               (loop for wd in (contents-of docID)
                   as i from 0
                        to (1- (length (contents-of docID)))
                   collect (cons docID (cons wd i))))
wordmap
cl-user(13): (defun contents-of (docID)
               (symbol-value docID))
contents-of
cl-user(14): (map 'list #'wordmap '(doc1 doc2))
(((doc1 さくら . 0) (doc1 学校 . 1) (doc1 の . 2) (doc1 ページ . 3))
 ((doc2 かえで . 0) (doc2 学校 . 1) (doc2 の . 2) (doc2 ページ . 3)))
```

検索でほしいものは，単語を与えたときに，その単語を含む文書IDと出現場所(インデックス)である．そのために単語を検索キーとするような，上記行列の転置行列を作る．ここに map-reduce が用いられる．map 関数で wordmap を用いて複数の文書について各々の文書の三つ組みリストが作られるが，そのリストを統合するのに reduce を用いる．shuffle-wordkeys 関数は単語をキーとして，初期値に与えたハッシュ表に文書インデックスを次々に統合してゆく．make-hash-table やgethashについては12.7節で説明している．loopについては第17章を参照されたい．

```
cl-user(15): (defun make-inverted-array (doclist)
               (reduce #'shuffle-wordkeys
                       (map 'list #'wordmap doclist)
                       :initial-value *shuffled-array*))
cl-user(16): (defun shuffle-wordkeys (shuffled wordkeys)
               (loop for (docID word . key) in wordkeys
                   do (push (cons docID key)
                            (gethash word shuffled)))
               shuffled)
shuffle-wordkeys
cl-user(17): (setq *shuffled-array* (make-hash-table))
#<eql hash-table with 0 entries @ #x20288ef82>
cl-user(18): (make-inverted-array '(doc1 doc2))
#<eql hash-table with 5 entries @  #x20288ef82>
```

こうしてできた転置行列 *shuffled-array* に検索ワードで検索すれば，それが含まれる文書とインデックスが得られるので，その周りを切りだしたり，統計処理をすることができる．

```
cl-user(19): (gethash 'さくら *shuffled-array*)
((doc1 . 0))
t
cl-user(20): (gethash 'かえで *shuffled-array*)
((doc2 . 0))
t
cl-user(21): (gethash '学校 *shuffled-array*)
((doc2 . 1) (doc1 . 1))
t
cl-user(22): (gethash 'ページ *shuffled-array*)
((doc2 . 3) (doc1 . 3))
t
cl-user(23): (defun show-inverted-array (hashtable)
               (loop for key being each hash-key
                     in hashtable using (hash-value val)
                  collect (cons key val)))
show-inverted-array
cl-user(24): (show-inverted-array *shuffled-array*)
((の (doc2 . 2) (doc1 . 2))
 (ページ (doc2 . 3) (doc1 . 3))
 (学校 (doc2 . 1) (doc1 . 1))
 (かえで (doc2 . 0))
 (さくら (doc1 . 0)))
```

Common Lisp では map-reduce プログラムはもちろんシングルスレッドで実行されるので，特にビッグデータ向きというわけではない．Lucene の map-reduce では大量のデータを互いに独立に処理できることを前提として，map-reduce が実行される．一般の map-reduce のフレームワークでは，データ間の依存関係の

見極めや効率の良い分散処理へのデータの分割が重要となるが，map 処理に用いられる関数や reduce に用いられる関数はアプリ依存にならざるをえない．それはもちろん Common Lisp の範疇の話ではないのでここまでとする．

第7章 関数閉包とエクステント

7.1 関数閉包

第4章で特殊変数，大域変数を導入し，静的束縛と動的束縛の議論をした．本章ではさらにその議論を深めて，最初に**関数閉包**(closure)について知ってもらい，そのうえで変数の参照に関するスコープと値確立に関するエクステントの概念を理解してもらう．

論理の世界では論理式に現れる論理変数が全称記号∀や存在記号∃により限量されているとき，これを変数束縛と言う．論理変数がその式中では変数束縛されていないとき，その変数を自由変数と言う．論理式では自由変数があるとそれだけでは論理式の真偽値は定まらない．ラムダ計算における式においても，λ変数（たとえば $\lambda x\,\mathrm{P}(x)$ の場合の x）を束縛変数と呼び，そうでないもの（たとえば $\lambda x\,\mathrm{P}(x, y)$ の場合の y）を自由変数と言う．

Lisp 関数においても自由変数の値が定まらない限りは，関数が評価できない．たとえば，次のように関数 foo と関数 bar があった場合，変数 x は関数 foo にとっては束縛変数だが，関数 bar にとっては自由変数である．

```
cl-user(13): (defun foo (x)
               (print x))
foo
cl-user(14): (defun bar ()
               (print x))
bar
```

だから，関数 foo は常に実行できても，関数 bar は実行できない場合がある．

```
cl-user(15): (foo 1)

1
1
cl-user(16): (bar)
Error: Attempt to take the value of the unbound variable `x'.
[condition type: unbound-variable]
```

では，次のようなコードではどうだろうか．

```
cl-user(1): (defun make-foo-bar ()
              (let ((x 10))
                (defun foo (x) (print x))
                (defun bar () (print x))))
make-foo-bar
cl-user(2): (make-foo-bar)
bar
cl-user(3): (foo 1)

1
1
cl-user(4): (bar)
```

```
10
10
```

Common Lisp では関数の中で関数を定義することもできる．defun による定義は必ずしもトップレベルで行わなくてもよいが，その効果は常にグローバルである．つまりdefunで定義された関数はどこからでも参照され得る．この例では関数 bar における x は bar にとっては自由変数であるが，それは静的スコープにおいて make-foo-bar 中に現れる let 変数 x であることは実施例からも明らかである．つまり，関数 bar を定義した時，その自由変数 x として，その時の**静的スコープで決まる変数が捕捉される**．このとき値が捕捉されるのではなく，あくまでもその変数そのものが捕捉されることに注意されたい．次の例を見られたい．

```
cl-user(5): (defun make-bar1 ()
              (let ((x 10))
                (defun bar1 ()
                  (print x))
                (setq x 100)
                ))
make-bar1
cl-user(6): (make-bar1)
100
cl-user(7): (bar1)

100
100
```

let 変数としての x は初期値 10 であるが，それが関数 bar1 に捕捉される．これはもちろん make-bar1 が実行されたときに，関数 bar1 が定義されて起こることである．同時にそのとき， x の値は 100 に関数 bar1 の外で，設定しなおされる．結果はご覧のとおり．つまり，静的変数ならば関数定義時の自由変数はその関数によってそのときの値が捕えられるのではなく，変数そのものが捕えられて，関数ごとどこへでも持ちまわられることになる．それが関数閉包というものである．

7.2 エクステントとは何か？

ここで理解が難しい変数のエクステントという概念について説明してみよう．

誰も触れない秘密の変数

let 変数の特殊な使い方として，システム中にあって直接にはだれにもさわれない，保守用関数を通してでないとアクセスできない変数を容易に作ることができる．次のコードを見られたい．

```
cl-user(1): (let (x)
              (defun my-confidential-key-get () x)
              (defun my-confidential-key-set (val)
                (setq x val)))
my-confidential-key-set
cl-user(2): (my-confidential-key-set 123)
123
cl-user(3): (my-confidential-key-get)
123
cl-user(4): (setq x 0)
```

```
0
cl-user(5): (my-confidential-key-get)
123
cl-user(6): (my-confidential-key-set 0)
0
cl-user(7): (my-confidential-key-get)
0
```

このコードで用いられた let 変数 x はシステム中のどこかにその場所が確保されているわけだが，いったん let 構文を完成させてしまうと，もうどこからもその変数に直接はアクセスすることはできず，このコードでは my-confidential-key-get と my-confidential-key-set を経由するしかアクセスする方法がない．昔 Lisp にオブジェクト指向プログラミングの機能がないころには，このようなコードで変数隠ぺいを行って，オブジェクト指向っぽいプログラムを書くというようなことが行われた．

さて前述の make-bar1 の let 変数もそうであるが，この「誰にも触れない変数」も何か不思議に思われなかっただろうか．変数 x は静的スコープだから外側からは触れないというのは分かる．でも my-confidential-key-set でセットした値が，その関数を終えたあとでも残っている．そしてあとからその値を取り出すことも，セットしなおすこともできる．つまり，スコープの外にあっても実はその存在は「生きている」．スコープとは変数の参照つまり見える見えないに関する概念だが，これとは別に，変数の値の確立が「生きている生きていない」の概念がある．それがエクステントである．Common Lisp の変数は，変数の参照と値の確立が分離している．ラムダ変数や let 変数は特殊変数の宣言がないかぎり静的変数であり，変数への参照の範囲すなわちスコープはその構文の字面上で確立された範囲であるが，「参照の可能性が存在し続ける限り束縛が存在し続けるので，時間的な制約はない」(*CLtL2*, 5.1.2)．そういうものを**無限エクステント**(indefinite extent)と呼ぶ．「let 変数の束縛はレキシカルスコープと無限(indefinite)エクステントを持つ」(*CLtL2*, 7.5, let Special Form)．スペシャル宣言のないラムダ変数も同様である(*CLtL2*, 第3章)．一方，スペシャル宣言されたラムダ変数や let 変数（特殊変数または動的変数）では，その値の確立は構文要素が実行完了した時に解除される．これを**動的エクステント**と呼ぶ．もちろん束縛が解除されれば，参照には意味がないから，特殊変数では変数の参照と値の確立は一致していると言ってもよいが，ややこしいことに，動的変数はその値は動的ではあるが，参照は束縛が有効であるかぎりどこからでもその値を見ることができる．これを**無限スコープ**の動的エクステントと呼ぶ．**動的スコープ**とはこの無限スコープの動的エクステントのことである．上記の議論は同じ名前の変数による値の隠ぺいとは別の話であり，実際にはこれに静的でも動的でも隠ぺいの議論が加わる．

```
cl-user(17): (defun foo0 ()
               (let ((x 0))
                 (declare (special x))
                 (foo1)))
foo0
cl-user(18): (defun foo1 ()
               (declare (special x))
               (print x)
               (let ((x 1))
                 (declare (special x))
                 (foo2)))
```

```
foo1
cl-user(19): (defun foo2 ()
               (declare (special x))
               (print x)
               (let ((x 2))
                 (declare (special x))
                 (foo3)))
foo2
cl-user(20): (defun foo3 ()
               (declare (special x))
               (print x)
               (let ((x 3))
                 (declare (special x))
                 (foo4)))
foo3
cl-user(21): (defun foo4 ()
               (declare (special x))
               (print x))
foo4
cl-user(22): (foo0)

0
1
2
3
3
```

この例では変数 x はすべてスペシャルである．関数実行に入るたびに，let 変数であるが特殊である変数 x の値がスタックに積まれていくが，それぞれの関数の冒頭で無限スコープによる参照によって関数実行時にそれ以前に積まれた値を参照して印刷している．

トップレベルで変数に値を与えるには，setq（あるいは set か setf）するしかないが，関数の内部であっても自由変数への setq はその変数がラムダ変数や let 変数**以外の変数**ならばトップレベルにおける変数の扱いと同様に無限スコープとなって，特殊変数と区別がつかなくなってしまう．自由変数の使用は思わぬバグを潜ませることにもなるので，無限スコープの変数については，4.4節で説明したような明示的なグローバル宣言をすることが望ましい．

第８章 マクロを書く

マクロはプログラムを書くプログラムだ．この文には非常に深い意味が込められている．
[Paul Graham, 1993]（On Lisp，野田訳）

さて，いよいよ中級 Lisper への最大の難関であるマクロについて説明する．再帰プログラムをクリアした多くの人が次にマクロで挫折する．しかし，Lisp のマクロこそ他のプログラム言語が決して真似ができないところであり，一人前の Lisper となるためにも，また Lisp 上でいわゆるドメイン特化言語(DSL)を開発するためにも必須の技術である．

8.1 マクロ展開

Common Lisp の when や until などはマクロである．そのコードを書けば，きっとこんな具合である．

```
(defmacro when (test &body forms)
  (if ,test (progn ,@forms) nil))
```

マクロ定義には一見すると defun によく似た形式の defmacro を用いるが，実のところ，その目的も働きも全然違う．Lisp 処理系がマクロ呼び出し書式を読み込むと，それを解釈する前にマクロ書式がインラインで展開され，そのマクロ展開されたコードがマクロ形式の前後の文脈の中で評価されたり（解釈時），コンパイルされたり（コンパイル時）する．マクロを理解しプログラムするためには，マクロ展開時とコード実行時（あるいはコンパイル時）を明確に区別することが大事である．たとえば，次のような書式を Lisp 処理系が受け取ると，

```
(when (equal *wheather-forecast* '(it will rain))
   (have 'umbrella)
   (go-out))
```

その場でwhenのマクロ展開が行われ，そのマクロの効果は次のような書式を与えた場合と全く同一のこととなる．

```
(if (equal *wheather-forecast* '(it will rain))
   (progn (have 'umbrella)
          (go-out))
  nil)
```

ここでアンダーラインの部分がそれぞれマクロ定義におけるパラメータ `test` と `forms` に相当するところとなる．

関数 not-null は次のようなものである．

```
(defun not-null (x) (not (null x)))
```

しかし，こんな簡単なコードでいちいち関数呼び出しをするのは気が引ける．これをマクロにしてみよう．マクロ定義するときには defun の代わりに defmacro を使うが，マクロ本体に書くものは，コードその

ものではなくそのようなコードを生成するプログラムである．Lisp におけるマクロはプログラムを生成するプログラムなのである．

```
cl-user(2): (defmacro not-null (x)
              (list 'not (list 'null x)))
not-null
cl-user(3): (not-null t)
t
cl-user(4): (not-null nil)
nil
cl-user(5): (not-null 'foo)
t
```

問題8.1 第2章の問題にある last1 のマクロ版を書け．

8.2 バッククォート・テンプレート

上記のように，マクロ本体には展開後のコードを生成するようなコードを書けばよいが，通常はバッククォートとカンマを使って，もっと簡単にマクロを書くことができる．たとえば上記の場合にはこんな風に書く．

```
(defmacro not-null (x) `(not (null ,x)))
```

バッククォートがすることは，クォートされた場合と同様な書式を生成することだが，バッククォートテンプレート内部におけるカンマはそれに続く書式（上記例では x）をその場で評価した結果で置き換える．もしカンマがなければバッククォートの結果は単にクォートの場合と変わらない．

バッククォートを使いこなすには，まず最初に生成されるべきS式を書きだしてから，その前にクォートの代わりにバッククォートを書き，次にその文脈において評価されるべき部分にカンマをつければよい．バッククォートはマクロと一緒によく用いられるが，別にマクロにしか使えないというわけではない．コードパターンが欲しくて，その一部分のみ（変数や書式の評価結果）を状況に応じたものにしたいというところには，どこでも有効に利用できる．

問題8.2 上記 last1 マクロをバッククォートを使って書け．

let 構文を自作する

今の let は特殊形式になっているが，昔の Lisp には let がなく，lambda 構文を使ったマクロで実装された．昔のようにマクロを使って our-let を定義してみる．次のような our-let 文があったとき，

```
(our-let ((arg1 val1) (arg2 val2) ... )
  form1 form2 ...)
```

実際に生成されるべきS式は，こんなものでなければならない．

```
((lambda (arg1 arg2 ... )
    form1 form2 ...)
```

```
 val1 val2 ...)
```

もし次のように our-let 定義を書きだしたとすると，

```
(defmacro our-let (parms &rest body) ... )
```

parmsは "((arg1 val1) (arg2 val2) ...)" のパターンに該当し，bodyには "form1 form2 ..." のパターンが該当する．難しいのは lambda 変数の仮引数と実引数を分離してラムダ文を作らなければならないところである．それには次のようにする．

```
(defmacro our-let (parms &rest body)
  `((lambda ,(mapcar #'car parms) ,@body)
       ,@(mapcar #'cadr parms)))
```

上記の場合には args の各要素の car のみを取り出すものが (mapcar #'car args) だが，その前にカンマが付いている．すなわち，処理系がマクロ定義された our-let を書式の第1要素として読み込んだとき，マクロ展開が起こって，そのときのマクロの引数 parms について (mapcar #'car parms) を実行し，得られた結果でそのパターンを置き換える． ただのカンマは単純に置き換えるだけだが，「,@」は直後の書式を評価したら，その結果を該当部分にリストの一部としてスプライスする．同様に，(mapcar #'cadr parms) は "val1 val2 ... " のパターンを作りだすが，それが最後にスプライスされる．期待通りに動くことを macroexpand で確かめることができる．

```
cl-user(1): (defmacro our-let (parms &rest body)
              `((lambda ,(mapcar #'car parms) ,@body)
                  ,@(mapcar #'cadr parms)))
our-let
cl-user(2): (macroexpand '(our-let ((arg1 val1) (arg2 val2))
                           (form1) (form2)))
((lambda (arg1 arg2) (form1) (form2)) val1 val2)
t
```

ついでに，our-let を用いることを前提に，我々の let* についても定義しておこう．

```
cl-user(3): (defmacro our-let* (parms &rest body)
              (if (null parms)
                  `(progn ,@body)
                `(our-let (,(first parms))
                   (our-let* ,(rest parms) ,@body))))
our-let*
```

ここではマクロについての再帰のプログラムとなっていることに注意されたい．マクロの再帰でも展開が進んだとき，最後には展開が終了するようにプログラムされなければならない．さもないと永遠に展開し続けてしまう．以下に (our-let* ((arg1 val1) (arg2 arg1)) ...) という書式について，マクロ展開を順次トレースしてみる．（関数のようなトレース機能はないので一つづ macroexpand する．）

```
cl-user(4): (macroexpand
              '(our-let* ((arg1 val1) (arg2 arg1))
                 (form1)
                 (form2)))
((lambda (arg1) (our-let* ((arg2 arg1)) (form1) (form2))) val1)
```

```
t
cl-user(5): (macroexpand
              '(our-let* ((arg2 arg1)) (form1) (form2)))
((lambda (arg2) (our-let* nil (form1) (form2))) arg1)
t
cl-user(6): (macroexpand '(our-let* nil (form1) (form2)))
(progn (form1) (form2))
t
```

だから，出来上がる展開形はこんなものになる．

```
((lambda (arg1) ((lambda (arg2) (progn (form1) (form2))) arg1)) val1)
```

これで確かに val1 の値が arg1 にわたって，それがさらに arg2 にわたっていくというのがよく分かる．

```
cl-user(7): (our-let* ((x 1) (y x)) (list x y))
(1 1)
```

次に示すように，色々な目的にマクロが利用される．

1. 関数呼び出しの数を抑えて高速化したい．
   コンパイルされるのはマクロ展開後の書式なので，関数呼び出しが1回減少することになる．比較的簡単な関数だが頻繁に呼び出されるようなものはマクロにする．ただし，マクロだとそれを引数で高階関数に渡すことはできなくなる．
2. 関数引数が評価されるのを抑制したい．
   Common Lisp の if は特殊形式で cond がマクロだが，昔は cond が特殊形式で if はなかった．だから if 文が欲しければ，マクロで実現した．もし if を関数としたら，テストコンディションの真偽にかかわらず，必ずその then部や else部に相当する書式が評価されてしまう．AND や OR もマクロであったことを思い出していただきたい．
3. コードを抽象化したい．
   同じことをマクロを使用せずに書くことはできるが，プログラムがごちゃごちゃと見通しが悪くなってきたときや，プログラムを作っていて何か同じようなコードパターンが繰り返し出てきたとき，それらをマクロにするとより抽象的なレベルでコードを書くことができて，プログラム全体の見通しも改善される．繰り返しをサポートする構文には do とか dotimes があるが，第17章で述べるように Common Lisp では C や Java よりも豊富な繰り返しの loop 構文がマクロで実装されて提供される．
4. 新しいLispの構文を追加したい．
   Common Lisp には with-何ちゃら という構文がたくさんあるが，これらはすべてマクロである．マクロを使えばドメイン特有のプログラム言語，いわゆるDSLを容易に開発することができる．

問題8.3　5.4節で属性リストを紹介した．Georgeのsex属性を取り出すには （get 'George 'sex） だが，値を与えるには (setf （get 'George 'sex) 'male) だった．これを　putマクロを新しく定義して (put 'George 'sex 'male） とできるようにせよ．

問題8.4　put/getの第1引数はシンボルで第2引数もシンボルであることがほとんどである．次のような putqq/setqqをマクロで定義せよ．

(macroexpand '(getqq George sex)） ==> (get 'George 'sex)

(macroexpand '(putqq George sex 'male)) ==> (setf （get 'George 'sex） 'male)

問題8.5　Schemeのdefineと同様な働きをするマクロを定義せよ.

(macroexpand-1 '(define my-name 'Seiji-Koide)) ==> (defvar my-name 'Seiji-Koide)

((macroexpand-1 '(define (pythagoras x y) (sqrt (+ (* x x) (+ y y))) ==>

　　(defun pythagoras (x y)　(sqrt (+ (* x x) (+ y y))))

問題8.6　次のような展開をするマクロ defperson を定義せよ。

(macroexpand-1 '(defperson George 28 male)) ==> (progn (setf (get 'George 'age) 28) (setf (get 'George 'sex) 'male))

8.3 変数の衝突と once-only

変数の衝突

　たとえば，defstruct とか defmethod のような汎用的で複雑なマクロを書くようになると，どうしてもマクロ中で局所変数を導入せざるを得なくなり，そういう場合には関数の場合と違ってマクロでは変数の衝突に注意しなければならない（変数の衝突の衝突は局所変数だけでなくラムダ変数でも起こるが）．なぜならばマクロそのものにはスコープがないからである．REPL は書式の第1要素がマクロ定義のシンボルであれば，そのマクロ定義を使ってその文脈でマクロ展開を実行する．言ってみればマクロの変数はいつもその文脈において大域的だから，もしマクロ定義中にマクロ用の変数を持ち込むとそれがマクロ展開時の文脈においてすでに使用中の変数であれば変数の衝突が起こる．変数の衝突を説明する最適な例は swap である．

　もし swap を関数型言語でないプログラム言語（参照呼出し型）で書けば次のようなものであろう．

```
(defun swap (x y)
  (let ((tmp x))
    (setq x y)
    (setq y tmp)))
```

　しかし，このコードは Lisp では動かない.

```
cl-user(18): (defun swap (x y)
  (let ((tmp x))
    (setq x y)
    (setq y tmp)))
swap
cl-user(19): (setq foo 1 bar 2)
2
cl-user(20): (swap foo bar)
1
cl-user(21): foo
1
cl-user(22): bar
2
```

　その理由は，Lisp 関数の呼び出し形式は値呼び出し（WinstonはLisp第1版では**引き渡しによる呼び出し**と名付け，第3版では値呼び出しとしている）であり，上記であれば実施例20において関数パラメータの x と y にはそれぞれ1と2がわたってくるが，関数swap 中の x と y はその外側の x と y とは全く関係がないからで

ある．Lisp で swap を実装したければどうしてもマクロにする必要がある．それで次のようにマクロ定義したとする．

```
cl-user(32): (defmacro swap (x y)
  `(let ((tmp ,x))
    (setq ,x ,y)
    (setq ,y tmp)))
swap
cl-user(33): foo
1
cl-user(34): bar
2
cl-user(35): (swap foo bar)
1
cl-user(36): foo
2
cl-user(37): bar
1
```

しかし，今ここで let 変数として tmp を使っていることに留意されたい．この例ではあまり不具合が起こりそうにもないが，一般に，マクロで let 変数を一時的に導入した場合，その変数がマクロ引数に渡される書式中に使われていたり，マクロ中のマクロで使われているとき，何が起こるか分からない．これを**マクロにおける変数の衝突**と呼ぶ．

変数の衝突の起こる簡単な例を，Successful Lisp (David B. Lamkins, 2005) からの引用で以下に紹介する．今簡単に，our-repeat を次のようにマクロ定義したとする．

```
cl-user(38): (defmacro our-repeat (times &body body)
               `(dotimes (x ,times) ,@body))
our-repeat
cl-user(39): (our-repeat 3 (print 'Hi))

Hi
Hi
Hi
nil
```

一見良さそうに見えるが，書式にシンボル x を用いた場合には，変数の衝突が起こって正しく働かない．

```
cl-user(40): (setq x 'Hi)
Hi
cl-user(41): (our-repeat 3 (print x))

0
1
2
nil
```

自分一人でプログラムして自分だけが使うときには，決してあり得ないような変数名を使えばよいが，マクロ定義で変数の衝突を避ける正しい方法は gensym を使う方法である．gensym は呼び出すたびに新しいシンボルを作りだす．

```
cl-user(42): (gensym)
#:g1256
cl-user(43): (gensym)
#:g1258
cl-user(44): (defmacro repeat (times &body body)
               (let ((x (gensym)))
                 `(dotimes (,x ,times) ,@body)))
repeat
cl-user(45): x
Hi
cl-user(46): (repeat 3 (print x))

Hi
Hi
Hi
nil
```

マクロ呼び出しパラメータに副作用がある時は once-only

マクロプログラミングにおいて，もう一つ重大な注意すべき点がある．次の例を見られたい．これも Successful Lisp (David B. Lamkins, 2005) からの引用である．与えられた引数の３乗を返すマクロを，関数と同じようにうっかり書いてしまったとする．

```
cl-user(47): (defmacro cube (n)
               `(* ,n ,n ,n))
cube
cl-user(48): (cube 3)
27
cl-user(49): (let ((n 2))
               (cube (incf n)))
60
```

上記実施例49は明らかに期待と異なるものであるが，その原因はマクロ展開結果を見れば一目瞭然である．ここで， macroexpand-1 はマクロ展開を１回だけ行うものである．実は incf もマクロであるため，macroexpand を使うと incf まで展開されてしまうのでここでは macroexpand-1 を使う．

```
cl-user(50): (macroexpand-1 '(cube (incf n)))
(* (incf n) (incf n) (incf n))
t
```

incf は副作用があるため，n の値は呼ばれるたびに1ずつ増加する．ここではそれが3回も呼ばれている．正しくは次のようにコード化されなければならなかった．

```
cl-user(55): (defmacro cube (n)
  (let ((x (gensym)))
    `(let ((,x ,n))
       (* ,x ,x ,x))))
cube
cl-user(56): (let ((n 2))
               (cube (incf n)))
27
```

この場合のマクロ展開は比較的簡単であるが，複雑な場合に正しくマクロをコード化するのは大変である．第2部の人工知能プログラミングでは，副作用のあるマクロ書式をうまく処理してくれる once-only という名前のマクロが用いられる．このマクロを用いると，cube マクロは次のように定義すればよい．

```
cl-user(59): (defmacro cube (x)
               (once-only (x) `(* ,x ,x ,x)))
cube
cl-user(60): (let ((n 2))
               (cube (incf n)))
27
```

問題8.7 once-only の定義は次のようなものである．上記 cube のマクロ定義と例題を例に，once-only の働きを考察せよ．

```
(defmacro once-only (variables &rest body)
  "Returns the code built by BODY.  If any of VARIABLES
  might have side effects, they are evaluated once and stored
  in temporary variables that are then passed to BODY."
  (assert (every #'symbolp variables))
  (let ((temps nil))
    (dotimes (i (length variables)) (push (gensym) temps))
    `(if (every #'side-effect-free? (list ,variables))
         (progn ,body)
       (list 'let
             `(list ,@(mapcar #'(lambda (tmp var)
                                  `(list ',tmp ,var))
                              temps variables))
             (let ,(mapcar #'(lambda (var tmp) `(,var ',tmp))
                           variables temps)
               ,body)))))

(defun side-effect-free? (exp)
  "Is exp a constant, variable, or function,
  or of the form (THE type x) where x is side-effect-free?"
  (or (atom exp) (constantp exp)
      (starts-with exp 'function)
      (and (starts-with exp 'the)
           (side-effect-free? (third exp)))))

(defun starts-with (list x)
  "Is x a list whose first element is x?"
  (and (consp list) (eql (first list) x)))
```

8.4 Elizaルール

第22章で本格的に説明するが，Elizaとはロジャー学派の精神科医の模倣をするプログラムのことである．PAIPによるElizaプログラムにはルール定義のためのマクロは記述されていないが，マクロ定義のちょうどよい例題として，Elizaのルール定義のためのマクロをここで書いてみる．そのルールがどのように使われるのかは，第22章を参照されたい．

一つのElizaルールは次のように，患者に相当するユーザが入力する文章とパターンマッチさせるためのパターン部と，パターンが一致したときの応答部から成っている．応答部は１個の場合もあるが普通は複数あり，Elizaプログラムはその中の一つをランダムに選んで応答する．次のルールはユーザが「についての夢を見る」あるいは「の夢を見る」という入力をしたときに対応するルールである．

```
(((?* ?x) dream about (?* ?y))
 (How do you feel about ?y in reality?))
(((?* ?x) dream (?* ?y))
 (What does this dream suggest to you?) (Do you dream often?)
 (What persons appear in your dreams?)
 (Do not you believe that dream has to do with your problem?))
```

このようなルールが大域変数*eliza-rules*に蓄積されるとする．

```
cl-user(4): (defparameter *eliza-rules* nil)
```

そこで，次のように入力したら，上記ルールがルール蓄積部に蓄積されるようなプログラムを考える．

```
(defrule rule1
  (?* ?x) dream about (?* ?y) ->
  (What does this dream suggest to you?)
  (Do you dream often?)
  (What persons appear in your dream?)
  (Don't you believe that dream has to do with your problem?))
```

defruleの構文は評価されてはならないので，マクロで定義するしかない．defruleのすぐ次のシンボルはルール名でありそのあとに，パターン部と応答部が続く．上記からすぐ分かるように，「->」を境にしてパターン部と応答部が分割される．マクロdefruleのプログラム例を以下に示す．

```
cl-user(5): (defmacro defrule (name &rest body)
              (let ((pattern (gensym))
                    (responces (gensym))
                    (pos (gensym)))
                `(let ((,pos (position '-> ',body)))
                   (let ((,pattern
                          (butlast ',body
                                   (- (length ',body) ,pos)))
                         (,responces
                          (cdr (last ',body
                                     (- (length ',body) ,pos)))))
                     (setf (get ',name 'eliza-rule)
                       (cons ,pattern ,responces))
                     (setq *eliza-rules*
                           (append *eliza-rules
                                   (list (get ',name 'eliza-rule))))
                     ',name))))
defrule
```

bodyはバッククォート中でカンマにより評価されるが，さらにその前にクォートが付いていることに注意されたい．ルール例をいくつか以下に示す．

```
cl-user(6): (defrule rule1
                (?* ?x) hello (?* ?y) ->
```

```
              (How do you do. Please state your problem.))
rule1
cl-user(7): (defrule rule2
                (?* ?x) computer (?* ?y) ->
              (Do computers worry you?)
              (What do you think about machines?)
              (Why do you mention computers?)
              (What do you think machines have to do with your problem?))
rule2
cl-user(8): (defrule rule3
                (?* ?x) name (?* ?y) ->
              (I am not interested in names.))
rule3
cl-user(9): (defrule rule4
                (?* ?x) sorry (?* ?y) ->
              (Please don't apologies.)
              (Apologies are not necessary.)
              (What feelings do you have when you apologies?))
rule4
cl-user(10): *eliza-rules*
((((?* ?x) hello (?* ?y)) (How do you do. Please state your problem.))
 (((?* ?x) computer (?* ?y)) (Do computers worry you?) (What do you think about machines?)
(Why do you mention computers?)
  (What do you think machines have to do with your ...))
 (((?* ?x) name (?* ?y)) (I am not interested in names.))
 (((?* ?x) sorry (?* ?y)) (Please don 't apologies.) (Apologies are not necessary.) (What
feelings do you have when you apologies?)))
```

上記5行目のdefruleの定義はこれで正しいのであるが，よく見るとパターンに対する(butlast ',body (- (length ',body) ,pos)と応答に対する(cdr (last ',body (- (length ',body) ,pos))はマクロ展開時に計算可能であることが分かる．必要なデータはすべてその文脈で与えられていて即時に計算できるからである．そのような場合にはマクロ展開時に計算してしまって，より具体的でコンパクトな展開形式にすることができる．

```
cl-user(11): (defmacro defrule (name &rest body)
               (let ((pattern (gensym))
                     (responces (gensym)))
                 `(let ((,pattern
                          ',(butlast body
                                     (- (length body)
                                        (position '-> body))))
                        (,responces
                          ',(cdr (last body
                                       (- (length body)
                                          (position '-> body))))))
                    (setf (get ',name 'eliza-rule)
                      (cons ,pattern ,responces))
                    (setq *eliza-rules*
                          (append *eliza-rules*
                                  (list (get ',name 'eliza-rule))))
                    ',name)))
defrule
```

ここでも(butlast ...)と(cdr (last ...)の前にはカンマとクォートが付いていることに注意されたい．

8.5 マクロにおけるラムダパラメータ

マクロ定義の形式は関数定義とよく似ている．関数定義時のラムダパラメータの使い方については既に説明した．マクロ定義時も基本的には関数定義時と同様である．レストパラメータとキーワードパラメータの併用について以下の例を見られたい．

```
cg-user(6): (defmacro key-arg-test1 (x y &rest rest &key a b)
              `(list ',rest ',(list x y) ',(list a b)))
key-arg-test1
cg-user(7): (key-arg-test1 foo bar :a 1 :b 2)
((:a 1 :b 2) (foo bar) (1 2))
```

問題8.9　次のような展開をするマクロ defperson を定義せよ。
(macroexpand-1 '(defperson George :age 28 :sex male)) ==> (progn (setf (get 'George :age) 28) (setf (get 'George :sex) 'male))

マクロ定義ではこれまで述べたように、変数の衝突や多重評価 (once-only) に気を付け無ければならないという面倒くささはあるが，呼び出し形式を定義時パターンにあわせて分解・分配するなど，関数定義より高度な機能もある．以下の例を見られたい．

```
cg-user(16): (defmacro macro-reverse ((head . tail))
               `(cond ((null ',tail) (list ',head))
                      (t (append (macro-reverse ,tail) (list ',head)))))
macro-reverse
cg-user(17): (macro-reverse (a b c d e))
(e d c b a)
```

マクロ `macro-reverse` の呼び出しでは引数として一つのリストが与えられた．このマクロ定義ではパラメータは一つのリストであり，その car 部分をパラメータ head，cdr 部分をパラメータ tail としている．すると呼び出し時に与えられたリストが car と cdr に分解され，それぞれのラムダ変数に束縛される．これがマクロ呼び出しにおける分解・分配機能である．こんなことは、関数呼び出しではできない（明示的に destructuring-bind をどこかに使うことはできる）．

8.6 マクロを書くマクロ

上記 defperson ではあらかじめ defperson と George と，属性 :age と :sex は固定であった． defperson はそのままとしても、人の名前と人の属性は自由に拡張設定したい．こんなとき，利用できるのがマクロ定義を生成するマクロだ。マクロ定義を生成するマクロになると、多重のカンマとクォートとバッククォートをどう組み合わせて使うのかという問題が生じやすい．

例えば、

問題8.10　次のようなマクロ定義マクロ gen-defperson を定義せよ。
(gen-defperson name &rest props))として

(gen-defperson George :age :sex :email)としたら
(defperson George :age 28 :sex male :email nobody@gmail.com)とできる defperson マクロを生成するような gen-defperson マクロ

Shiro Kawai(Gaushe の作者) さんの記事が一番コンパクトでいいのだけれど、コンパクトすぎるきらいもある。シローさんの記事を読んでから、次の問題にチャレンジしてみよう。どこまでできるだろうか.

第9章 コンパイラ

歴史的にLispのコンパイラはインタプリタのあとに開発された．インタプリタがコンパイルされたプログラムより遅いことは当たり前である．短い期間ではあるがLispにはインタプリタしかない時代があり（当時存在していたFORTRANはコンパイラで動いた），そのためかLispは遅いという誤った神話がいまだに根強く残っているように思われる．

9.1 loadとcompileとcompile-file

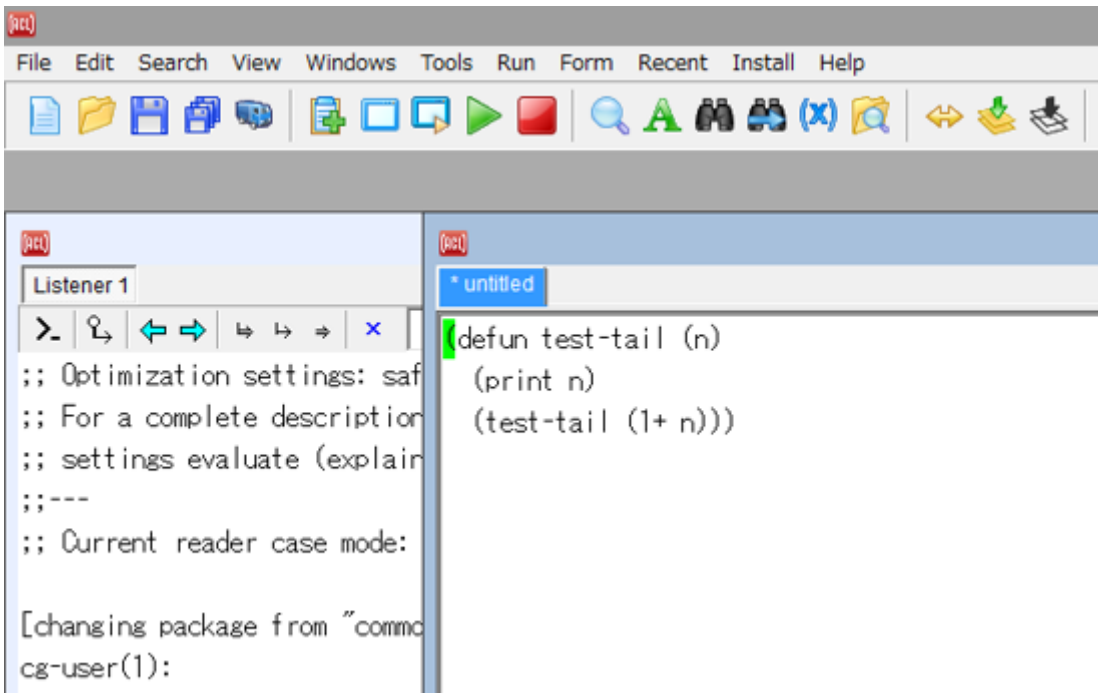
単独の関数をコンパイルする

対話的にREPL環境でプログラムしていくときはインタプリタを利用している．ただし、SBCLではREPL上でdefunとタイプ入力しただけで自動的にコンパイルされる．関数1個をREPL環境でコンパイルするには，通常、関数compileを用いる．第10章の末尾再帰では，末尾再帰の関数をコンパイルする必要があるため，次のように関数compileが用いられる．

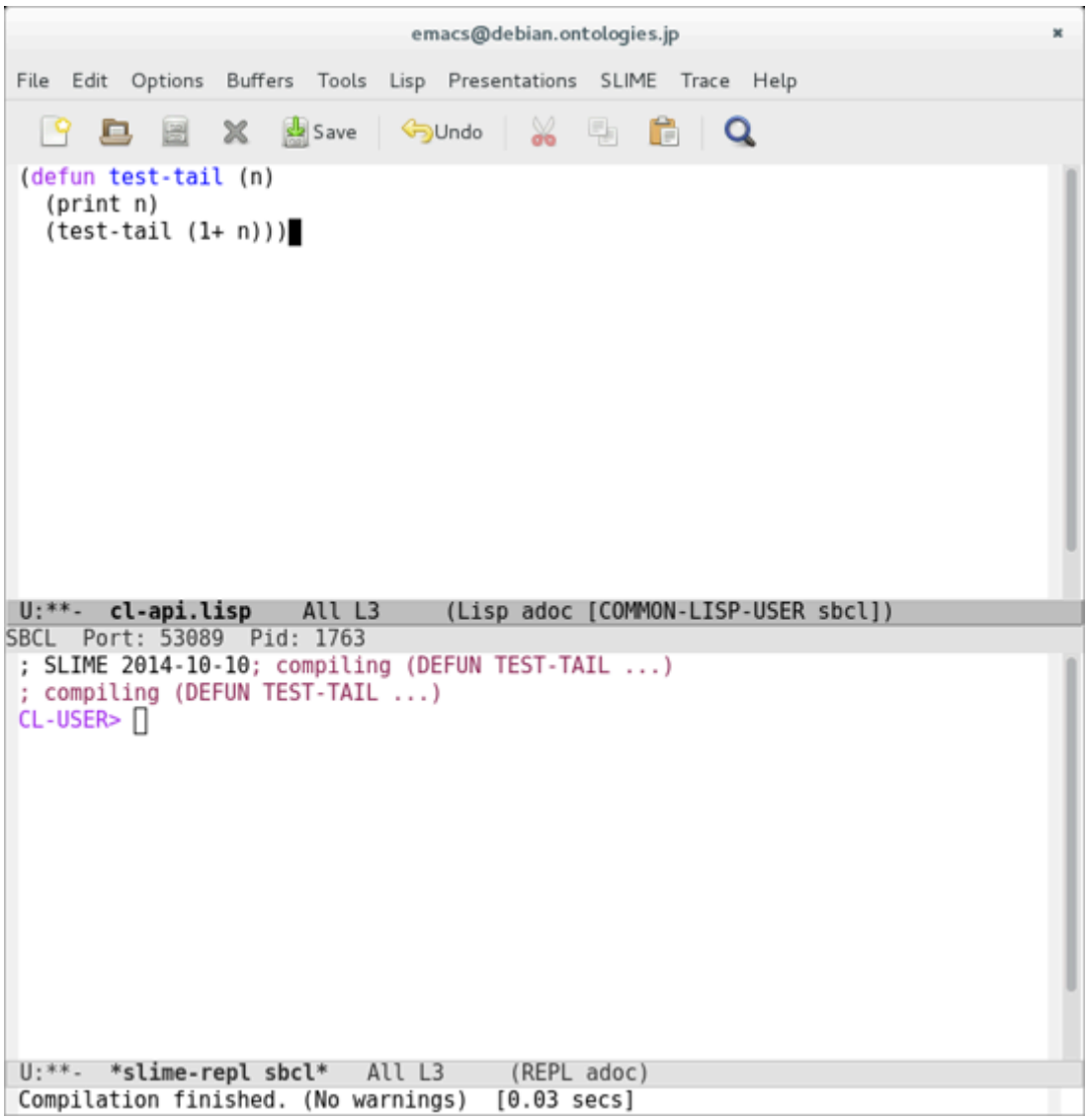
```
cl-user(2): (defun test-tail (n)
              (print n)
              (test-tail (1+ n)))
test-tail
cl-user(3): (compile 'test-tail)
test-tail
nil
nil
```

本書籍では関数定義を説明するのに，トップレベルでの直打ちのような記載をしているが，それは記述をコンパクトにする都合上のことであり，実際にこんなやりかたをするのはごく簡単な関数を定義するときぐらいである．ACLに付属のLispエディタでも，あるいはemacsをLisp用に仕立てたSlime利用のemacsエディタでも，プログラム編集中に任意の書式をエディタ上で評価したりコンパイルしたりすることができるので，普通はエディタを開いた状態でその上で対話的にプログラム開発が行われる．

ACL with IDEでは書式を選択しておいて，Ctrl-E (Ctrlキーを押した状態でeキーを押す)で選択された書式を評価できるし，Ctrl+Dで選択した部分のコンパイルができる．



Slime+emacs では，エディタ環境中で選択された関数をC-c C-c（Ctrl キーを押した状態で c キーを 2 回押す）キーコマンドでコンパイルすることができる.



ソースファイルをコンパイルする

複数の関数や defvar などの変数を定義したソースファイルは，ファイルタイプに lisp とか lsp とか，ACL では cl というタイプが付いて保存されるが，それを再び Lisp 環境中に取り込むには，関数 load を用いる．しかし，この関数 load はLispソースコードを自動的にコンパイルすることはしない．だからロードされてもそのプログラムはインタープリタで解釈される．

Lisp のソースコードファイルをファイルごとコンパイルするには，compile-file を用いる．Lispのソースファイルをコンパイルすると，fasl というファイルタイプのファイルが生成される．関数 load はこの fasl ファイルをロードすることもできる．

```
cl-user(3): (compile-file "test-tail.cl")
;;; Compiling file test-tail.cl
;;; Writing fasl file test-tail.fasl
;;; Fasl write complete
#P"C:\\acl100express\\test-tail.fasl"
nil
nil
cl-user(4): (load "test-tail.fasl")
```

```
; Fast loading C:\acl100express\test-tail.fasl
t
```

ファイルタイプなしで指示した場合には，fasl ファイルがあればそれが，無ければソースコードファイルがロードされるはずだ．

ソースコードのコンパイルとコンパイルコードのロードは一応別物ではあるが，ほとんどの場合，それらは一体のものとして実行される．ACLプロジェクトではプロジェクトをオープンすると，ソースファイルの更新日がコンパイルファイルよりも新しければ自動的にコンパイルされてからロードされるし，quicklispの場合でもそのようになる．Slime+emacs では C-c C-k (slime-compile-and-load-file) が頻繁に使用される．
個々の処理系に特有な処理方法についてはこれ以上言及しないが，各自研究されたい．

しかし，たまにREPL環境では動くのにコンパイルしてうまく動かないときがある．あるいは，ソースコードファイルをロードした場合は動くのに，コンパイルファイルをロードしても動かないという場合もある．そういう場合には，ソースコードとコンパイルコードでの挙動の違いを突き止めて，問題を解決しなければならない．それらのほとんどは，コンパイラがすることを細部まで分かっていないためであり，それは特にマクロに関連する場合が多い．

もしマクロを変更したら，そのマクロを利用しているソースコードもコンパイルし直さなければならないのは，当然のことである．**コンパイルされるのはマクロ展開されたあとのコード**だからである．

同様に，ファイルBがファイルAに依存していて(use していて)，ファイルBの中からファイルA中の関数を利用しようとして，ファイルA中でその関数を外部シンボルとして移出しても，コンパイル済みのAはそれを知る方法がないので，再度コンパイルしなければならない．

9.2 eval-when で評価時期のコントロール

前章のマクロの説明で，「マクロを理解しプログラムするためには，マクロ展開時とコード実行時（あるいはコンパイル時）を明確に区別することが大事である」と述べた．インタプリタによるコード実行時には，マクロ形式がインラインで展開されなければならないから，マクロ利用時よりも前にそのマクロが定義されていなければならない．同様に，ファイルをコンパイルするときには，そのファイル中で利用するマクロはコンパイル時にはすでにシステムに定義済みでなければならない．だから，通常マクロ定義は本体関数群よりも前にロードされるユーティリティファイル中に定義したりする．しかし，どうしてもモジュールのまとまりとしてマクロを利用する関数と同じファイルにマクロを書いておきたいというような場合には，eval-when を使ってそれができる．

CLtL1 と *CLtL2* ではこの eval-when の仕様が若干異なる．以下はPAIPのユーティリティファイル auxfns.lisp ファイルの冒頭部分にある eval-when であるが，これは*CLtL1*で書かれており，今日では eval は :execute に，compile は :compile -toplevelに，load は :load-toplevel に書き直したほうがよい．

```
(eval-when (eval compile load)
  ;; Make it ok to place a function definition on a built-in LISP symbol.
```

```
  #+(or Allegro EXCL)  (dolist (pkg '(excl common-lisp common-lisp-user))
    (setf (excl:package-definition-lock (find-package pkg)) nil))
  ;; Don't warn if a function is defined in multiple files --
  ;; this happens often since we refine several programs.
  #+Lispworks  (setq *PACKAGES-FOR-WARN-ON-REDEFINITION* nil)
  #+LCL    (compiler-options :warnings nil)
  )
```

すなわち，上記コードは下記のように修正されるべきである．

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  ;; Make it ok to place a function definition on a built-in LISP symbol.
  #+(or Allegro EXCL)  (dolist (pkg '(excl common-lisp common-lisp-user))
    (setf (excl:package-definition-lock (find-package pkg)) nil))
  ;; Don't warn if a function is defined in multiple files --
  ;; this happens often since we refine several programs.
  #+Lispworks  (setq *PACKAGES-FOR-WARN-ON-REDEFINITION* nil)
  #+LCL    (compiler-options :warnings nil)
  )
```

ここで eval-when はその囲まれたフォームをいつ評価したらよいのかを指定するための特殊形式で，:compile-toplevel，:load-toplevel，:execute などは**状況**と呼ばれている．その意味は*CLtL1*と*CLtL2*の記述にあるように，「eval-when 特殊形式は，コードの断片の実行をコンパイル時だけ，あるいはロード時だけ，あるいは解釈されるがコンパイルされないときだけ，というように指定することができる」のであるが，「その使い方はかなり奥が深い」．*CLtL2*を見れば，Common Lisp 実装者に便利なように，詳細な記載がされているが，はたしてどう使ったらよいかという観点からは理解することが難しい．この三種類の個別状況の組み合わせで7種類の組み合わせを作ることができるが，実際にもっぱら用いられるのは上記のように，:compile-toplevel，:load-toplevel，:execute がすべて並んだものだ．ちなみに，手元の様々なCommon Lispソースファイル群でこの三種類の組みあわせの数を調べてみると，三つすべてを並べたものが171，:load-toplevel と :execute の組み合わせが24，:compile-time+:execute が 5，:load-toplevel のみが 4，その他は 0 であった．

defun，defvar，defparameter，defconstantしかないようなコードでは通常はコンパイラが面倒を見てくれるので，システム依存の特有なことをしたいとかのような特殊なこと以外では，何も心配することはない．しかし何か結果的に副作用を期待するようなコードではそれが実行時にもコンパイル時にも同様であってほしいことであれば，これらの三つの状況を並べる．コンパイル時にはしてほしくない副作用だったら，:load-toplevel と :execute の組み合わせになる．ロード時のみに欲しい副作用だったら，:load-toplevel 一つになる．ただし，defpackage，in-package などは当然のこととしてコンパイラが面倒を見てくれるので心配する必要はない．手元のコードからいくつかの例を挙げると以下のようである．

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defmacro base-char-code-p (code)
    `(and (eq 0 (ldb (byte 1 7) ,code))
          (eq 0 (ldb (byte 1 15) ,code))))

(eval-when (:compile-toplevel :load-toplevel :execute)
  (require 'sb-introspect))
```

```
(eval-when (:load-toplevel :execute)
  (defparameter *owl-directory*
    (make-pathname :host (pathname-host *load-truename*)
                   :device (pathname-device *load-truename*)
                   :directory (pathname-directory *load-truename*))))

#+allegro
(eval-when (:compile-toplevel :execute)
  (when (boundp 'excl:*warn-on-nested-reader-conditionals*)
    (setf excl:*warn-on-nested-reader-conditionals* *acl-warn-save*)))

(eval-when (:load-toplevel)
  (let ((*default-pathname-defaults* *load-pathname*))
    (read-rdf-file #'addRdfXml "OWL:OWL.rdf")))
```

9.3 #. 形式でコンパイル時の値を固定する

#. は標準ディスパッチングマクロ文字（付3.3も参照のこと）の一種であり，その機能はそれに続く書式を最初の読み込み時に，実行時でもファイルをコンパイルする場合でも，評価するというものである．この機能により，コンパイル時にその値を固定してしまうことができる．get-time.cl あるいは get-time.lisp という名前で，次のような内容のファイルを作ってみよう．

```
(defun get-compiled-time ()
  (decode-universal-time #.(get-universal-time) -9))

(defun get-current-time ()
  (decode-universal-time (get-universal-time) -9))
```

そして以下のようにファイルのコンパイルとロードを行い， get-compiled-time の呼び出しを行ってみよう．

```
cl-user(14): (compile-file "get-time.cl")
;;; Compiling file get-time.cl
;;; Writing fasl file get-time.fasl
;;; Fasl write complete
#P"C:\\acl100express\\get-time.fasl"
nil
nil
cl-user(15): (load "get-time.fasl")
; Fast loading C:\acl100express\get-time.fasl
t
cl-user(16): (get-current-time)
43
23
13
12
10
2015
0
nil
-9
cl-user(17): (get-compiled-time)
22
23
```

```
13
12
10
2015
0
nil
-9
cl-user(18): (get-compiled-time)
22
23
13
12
10
2015
0
nil
-9
cl-user(19):
```

　すると，get-compile-time の結果はいつまでたっても変化しない．この関数の定義を眺めてみると，`#.(get-universal-time)`という記述がある．get-universal-time は14.1節で出てくるが，`#.`《書式》 はこの書式をインタープリットする場合でもコンパイルする場合でもリード時の値を固定する．したがってファイルをコンパイルすると**コンパイル時に実行**されたときの評価値がここに与えられる．だからこのようにコンパイル時の値をそこで凍結することができる．この性質を積極的に利用して，一度評価したら再び値が変わることがないようなもの，そしてコンパイル時に計算可能でかついつ何度評価しても変わるはずはないようなものにはこの形式を使って，実行速度を向上させることができる．

第10章 末尾再帰，継続，コルーチン

> 継続とは，動作中のプログラムを凍結したものだ．すなわち計算処理の状態を含んだ一つの関数オブジェクトだ．保存された計算処理は，それが中断された時点から再開する．
>
> [Paul Graham, 2006]（On Lisp，野田(訳), 2007）

10.1 末尾再帰

末尾再帰(tail recursive)とは3.1節で述べた our-member のように，再帰の結果が何も加工されずに，そのまま関数の結果として返されるようなことを言う．以下に our-member を再掲しておく．

```
(defun our-member (item list)
  "This is almost the same function as common lisp's member."
  (cond ((endp list) nil)                ; empty list
        ((equal item (car list)) list)    ; matched to the first
        (t (our-member item (cdr list))))); else next
```

なぜ末尾再帰をわざわざ取り上げる価値があるかといえば，Common Lisp ではコンパイラを利用して，それが後述の**継続**(continuation)とか**コルーチン**(coroutine)を実装する手段となるからである．末尾再帰そのものは，本来コンパイラとは別の議論であり，特にSchemeにおいては第11章で説明するようにコンパイラのみならずインタプリタにおいても末尾再帰に特別な処置が実装されるが，Common Lisp では末尾再帰についての仕様記述はない．しかし実際にはほとんどの主要な処理系のコンパイラは末尾再帰を見つけるとこれを特別に処理する．ただし，明確な仕様記述がないために，すべての末尾再帰が処理されるというわけではなく，その程度は難易度に依存して処理系によってまちまちである．

一般に，関数呼び出しでは戻り番地がスタックに積まれるという計算モデルで正しいが，Common Lisp のコンパイラは末尾再帰を見つけるとそれを通常の再帰とはことなり，無条件jump のようにコンパイルする．手元の処理系で以下のように実験してみてほしい．見れば分かるように，これは停止することなく延々と数字を印刷するプログラムである．

```
cl-user(1): (defun test-tail (n)
              (print n)
              (test-tail (1+ n)))
test-tail
cl-user(2): (test-tail 0)
```

やってみればすぐ分かるが，ACLでは途中で Stack Overflow でブレークするか止まってしまう．SBCLではREPLでも常にコンパイルされてしまうのでブレークしない

ACLだったら今度はコンパイル後に同じことをしてみる．

```
cl-user(1): (defun test-tail (n)
              (print n)
              (test-tail (1+ n)))
test-tail
```

```
cl-user(2): (compile 'test-tail)
test-tail
nil
nil
cl-user(3): (test-tail 0)
```

すると今度は止まらずに延々と数字をプリントし続ける．これが末尾再帰の効果である．つまりコンパイルされた末尾再帰はスタックを消費しない．ACLのコンパイラはSBCLに比べて末尾再帰処理の程度が緩いものであるが，それでもこのような簡単な末尾再帰はコンパイラがきちんと処理をしてくれる．だから，末尾再帰の効果を得るために，末尾再帰ではないプログラムをわざわざ書き直して末尾再帰にするということも行われる．たとえば，3.1節に述べた count-elementを以下に再掲すると次のようなものであった．

```
(defun count-elements (list)
  (cond ((endp list) 0)
        (t (1+ (count-elements (cdr list))))))
```

これを末尾再帰にするには，次のようにすればよい．こうすることで，これを**コンパイルすると**膨大なリストでもスタックを消費することなく処理できるようになる．

```
cl-user(28): (defun count-elements (list)
               (%count-elements list 0))
count-elements
cl-user(29): (defun %count-elements (list n)
               (cond ((endp list) n)
                     (t (%count-elemens (cdr list) (1+ n)))))
%count-elements
cl-user(30): (trace %count-elements)
(%count-elements)
cl-user(31): (count-elements '(one two three four five))
 0[5]: (%count-elements (one two three four five) 0)
   1[5]: (%count-elemens (two three four five) 1)
     2[5]: (%count-elemens (three four five) 2)
       3[5]: (%count-elemens (four five) 3)
         4[5]: (%count-elemens (five) 4)
           5[5]: (%count-elemens nil 5)
           5[5]: returned 5
         4[5]: returned 5
       3[5]: returned 5
     2[5]: returned 5
   1[5]: returned 5
 0[5]: returned 5
5
```

ここでは末尾再帰の理解のためにトレースしたが，もちろん，トレースを掛けている状態では，スタックは消費されるであろう．一言注意しておく．

問題10.1 先の問題3.9の count-atomsの答えは次のようなものである．

```
(defun count-atoms (list)
  (cond ((null list) 0)
        ((atom list) 1)
        (t (+ (count-atoms (car list))
              (count-atoms (cdr list))))))
```

これを末尾再帰に修正せよ.

問題10.2 リストを逆順にする our-reverse はこんなものだった.

```
(defun our-reverse (list)
  (cond ((null list) '())
        (t (append (our-reverse (cdr list))
                   (list (car list))))))
```

これを末尾再帰に修正せよ.

10.2 継続

継続(continuation)とは，中断された計算を再開するために呼ばれるような関数オブジェクトのことである．次の関数は 1 から任意の数までの合計を求める関数である． labels についてはこのあとコルーチンのところで説明する.

```
(defun sum-1-to-n (-c- n)
  (labels ((sum-loop (nn sum)
             (cond ((= nn 0) (funcall -c- sum))
                   (t (sum-loop (1- nn) (+ sum nn))))))
    (sum-loop n 0)))
```

ここではまだあまり「中断された計算を再開する」という雰囲気はないが，-c- としたのが，継続である．Common Lisp の継続は常に関数で持ち回られる．この場合最初に -c- に与えるものは最後のfuncall されるときにつじつまが合えばどんな関数でもよい．最も典型的には values である（valuesについては19.1節を参照のこと）．以下の実施例を参照されたい.

```
cl-user(4): (sum-1-to-n #'values 10)
55
```

これをもう少し継続らしくしてみる.

```
cg-user(2): (setq next nil)
nil
cg-user(3): (defun sum-1-to-n (-c- n)
              (labels ((sum-loop (nn sum)
                         (cond ((= nn 0) (funcall -c- sum))
                               (t (setq next (lambda ()
                                               (format t "Got ~S!" nn)
                                               (sum-loop (1- nn) (+ sum nn))))
                                  (values)))))
                (sum-loop n 0)))
sum-1-to-n
cg-user(4): (sum-1-to-n #'values 3)
cg-user(5): (funcall next)
Got 3!
cg-user(6): (funcall next)
Got 2!
cg-user(7): (funcall next)
Got 1!
6
```

今度は計算をすることなく，クロージャーを作ってそれをシンボル next にセットする．next には format 印刷文とちょっとだけ計算して再帰するものをクロージャーとしたものがセットされるが、それを funcall で実行すれば御覧の通りだ。

factorial は再帰プログラミングの代表例として，よく引用されるが，通常次のようにコード化される．

```
(defun factorial (n)
  (if (zerop n) 1
    (* n (factorial (1- n)))))
```

一方，継続を用いた factorial は次のようにコード化される．

```
(defun actorial (-c- n)
  (if (zerop n) (funcall -c- 1)
    (actorial #'(lambda (z) (funcall -c- (* n z)))
              (1- n))))
```

関数 actorial はパラメータとして n に加えて継続を受け取る．そして通常は受け取った継続に対して次の factorial にふさわしい新しい継続を作りだして再帰している．actorial の再帰呼び出しのたびに n が 1 づつ減じられ，n がゼロのときに再帰は止まり，その時点でもらった継続の (funcall -c- 1) の結果が一気に計算されて返される．ちょっと試してみよう．

```
cl-user(9): (actorial #'values 50)
30414093201713378043612608166064768844377641568960512000000000000
```

簡単な場合で n が 3 の場合に再帰を追跡するとこうなっている．

```
(actorial #'values 3)
(actorial #'(lambda (z)
              (funcall #'values (* 3 z)))
          2)
(actorial #'(lambda (z)
              (funcall #'(lambda (z)
                           (funcall #'values (* 3 z)))
                       (* 2 z)))
          1)
(actorial #'(lambda (z)
              (funcall #'(lambda (z)
                           (funcall #'(lambda (z)
                                        (funcall #'values (* 3 z)))
                                    (* 2 z)))
                       (* 1 z)))
          0)
```

だから，最終的に再帰が止まったときに実行される継続は次のようなものになる．

```
(funcall #'(lambda (z)
             (funcall #'(lambda (z)
                          (funcall #'(lambda (z)
                                       (funcall #'values (* 3 z)))
                                   (* 2 z)))
                      (* 1 z)))
         1)
```

ACL9.0 (without IDE) では actorial のパラメータが 5000 でも計算したし，ACL9.0 (with IDE)ではスタックオーバーフローとなったが，それでもそれをコンパイルすればさらに計算できる．actorial の再帰呼び出しは末尾再帰だからコンパイルすると actorial の再帰部分は実質ループと同じことになる．ACL9.0 でもそれはループになったようだ．ループするたびに継続の形はどんどん長くなっていくのが問題であるが，これもその元の形 `#'(lambda (z) (funcall -c- (* n z)))` がうまくコンパイルされて，機械語のレベルでそれほどメモリを消費しなければ，それほどは問題とならない．ただし，このあたりは大いにコンパイラの性能に依存するし，継続 `-c-` は関数パラメータでわたってくるものなので，コンパイル時にはその実態は不明であって，その動的な継続をコンパイルで効率化するのは難しい．

10.3 コルーチン

コルーチン(coroutine) とは， A が B を呼び， B がまた A を呼ぶように相互に呼びあうようなものを言う．再帰であればこれは相互再帰となるが，コルーチンでは再帰と違って制御を返さないというところが違う．相互再帰が末尾再帰になっていると，AとBが互いに呼びあっても両者が制御を返さないので実質 GOTO となって形の上では相互再帰であるが実質はコルーチンとなる．継続を使うとこのコルーチンプログラミングが簡単に実装できる．

generator-consumer とか reader-writer というプログラミング・パラダイムもコルーチンの一種である．このパラダイムでは，片方がデータを作りだして，もう片方がそれを処理するだけであるが，処理するデータの 1 単位ごとに片方が片方に制御を渡す．

以下にもっとも簡単だと思われるコルーチンプログラムを示す．これは Charniak, Riesbeck, McDermott による「人工知能プログラミング」の Reader-Writer プログラムを Common Lisp 用に書き換えて（もとのプログラムは SCHUM という Scheme 風言語で書かれている），さらに簡単化したものである．

```
cl-user(11): (defun reader (-corou-)
  (labels ((mainloop (-corou- obj)
             (funcall -corou-
                      #'(lambda (-c-)
                          (mainloop -c- (read)))
                      obj)))
    (funcall #'mainloop -corou- (read))))
reader
cl-user(12): (defun writer (-corou-)
  (labels ((writerloop (-corou- thing)
             (cond ((eql thing :done) :done)
                   (t (print thing)
                      (print '>)
                      (funcall -corou- #'writerloop)))))
    (funcall -corou- #'writerloop)))
writer
cl-user(13): (writer #'reader)
This

This
> is
```

```
is
> a

a
> test.

test.
> :done
:done
```

見ての通り，このリーダ・ライタは端末からLispのS式を読み込み（上記ではシンボルアトム），それをおうむ返しにプリントして，:done を読み込んだら終了する．

labels はローカルにのみ有効な関数群を定義するのに用いられる．同様なものに flet があるが， flet のスコープがちょうど let と同様に，関数定義が flet の入れ子の本体で有効になるのに対して， labels は labels 自身の環境にその定義が見えるようにする．だから，局所関数として再帰関数を定義したいときには labels にしなければならない．

さて，上記実施例では何が起こったのだろうか？実施例13ではコルーチンとして reader を writer に渡している．だから，writer の定義における -corou- は最初は reader だ．その内部では今度はその -corou- すなわち reader に writerloop を渡している．だから reader定義における -corou-は writerloop だ． mainloopが実行されるとき， obj には読み込まれたオブジェクト， -corou- には writerloop がわたってくる．その -corou- すなわち writerloop が実行されるときの thing には読み込まれた obj の値と新しいコルーチン #'(lambda (-c-) (mainloop -c- (read))) が渡される． :done が読み込まれるまでは thing の値とプロンプトを印刷して， (mainloop -c- (read)) を実行するがそのときの -c- は writerloop である．ここでは，局所関数を名前で引くときはローカルにしか引けないが，局所的であっても局所関数本体は引数で引き渡されることに注意されたい．Common Lisp では (funcall -corou- ...) と書かなければならないが，これが Scheme だと (-corou- ...)でよくなるから，ずっと見やすくなるし，プログラミング時にもやりやすい．

このプログラムをベースに，ファイルのコピープログラムを作ってみる．それは以下のようなプログラムだ．

```
(defun reader (-corou- in-stream)
  (labels ((mainloop (-corou- char)
             (funcall -corou-
               #'(lambda (-c-)
                   (mainloop -c- (read-char in-stream nil nil)))
               char)))
    (funcall #'mainloop -corou- (read-char in-stream))))

(defun writer (-corou- out-stream)
  (labels ((writerloop (-corou- char)
             (cond ((null char) :done)
                   (t (write-char char out-stream)
                      (funcall -corou- #'writerloop)))))
    (funcall -corou- #'writerloop)))
```

こんな単純なコピープログラムでも，コンパイルしないままだと，ACL9.0(with IDE)では，なんと5，6行でスタックオーバーフローになってしまう．そこで，これをコンパイルすればスタックオーバーフローすることなく，大容量のファイルでも処理可能となる．

```
cl-user(3): (compile 'reader)
reader
nil
nil
cl-user(4): (compile 'writer)
writer
nil
nil
cl-user(5): :cd
C:\Users\誠二\
cl-user(6): (with-open-file (in-stream "test1.txt")
    (writer #'(lambda (-corou-) (reader -corou- in-stream))
            *standard-output*))
Mrs. Rachel Lynde lived just where the Avonlea main road dipped down into a little hollow,
fringed with alders and ladies' eardrops and traversed by a brook that had its source away
back in the woods of the old Cuthbert place; it was reputed to be an intricate, headlong
brook in its earlier course through those woods, with dark secrets of pool and cascade; but
by the time it reached Lynde's Hollow it was a quiet, well-conducted little stream, for not
even a brook could run past Mrs. Rachel Lynde's door without due regard for decency and
decorum; it probably was conscious that Mrs. Rachel was sitting at her window, keeping a
sharp eye on everything that passed, from brooks and children up, and that if she noticed
anything odd or out of place she would never rest until she had ferreted out the whys and
wherefores thereof.
:done
```

次は言語処理プログラムやファイル変換プログラムでは必須となるパーザを，このリーダ・ライタで書いてみよう．このパーザは，ファイル入力から英数字の並びを切り出して一つのトークンとし，リストにして返すようなプログラムである．英数字トークン以外の特殊文字については出力しない．

```
(defun reader (-corou- in-stream)
  (labels ((mainloop (-corou- char)
             (cond ((null char)
                    (funcall -corou- #'values nil))
                   ((alphanumericp char)
                    (funcall -corou-
                             #'(lambda (-c-)
                                 (mainloop -c- (read-char in-stream nil nil)))
                             char))
                   (t ; special character such that #\Newline or #\Space
                    (specloop -corou- char (read-char in-stream nil nil)))))
           (specloop (-corou- prev-char char)
             (cond ((null char)
                    (funcall -corou- #'values nil))
                   ((not (alphanumericp char))
                    (specloop -corou- char (read-char in-stream nil nil)))
                   (t ;; new token started
                    (funcall -corou-
                             #'(lambda (-c-) (mainloop -c- char))
                             prev-char)))))
    (funcall #'mainloop -corou- (read-char in-stream))))

(defun writer (-corou-)
```

```
  (let ((token nil)
        (token-list nil))
    (labels ((writerloop (-corou- char)
               (cond ((null char)
                      (unless (null token)
                        (push (coerce (nreverse token) 'string) token-list))
                      (setq token nil)
                      (funcall -corou- (nreverse token-list)))
                     ((alphanumericp char)
                      (push char token)
                      (funcall -corou- #'writerloop))
                     (t ; special char such that #\Newline or #\Space
                      (unless (null token)
                        (push (coerce (nreverse token) 'string) token-list))
                      (setq token nil)
                      (funcall -corou- #'writerloop))
                     )))
      (funcall -corou- #'writerloop))))
```

英数字の場合には１文字づつ writerloop に引き渡されて，token に蓄積される．そうでない場合には，次の文字を読んで，specloop に行く．specloop では，特殊文字が続くかどうかを判断し，続く間はループする．新しいトークンの始まりを検知したら，その新しい英文字で mainloop に戻ってくるような継続を作りだして writerloop に行く．このとき，writerloop に渡される特殊文字は単に，トークンの終わりを知らせるためだけのものである．

```
cl-user(19): (compile 'reader)
reader
nil
nil
cl-user(20): (compile 'writer)
writer
nil
nil
cl-user(21): :cd
C:\Users\誠二\
cl-user(22): (with-open-file (in-stream "test1.txt")
  (writer #'(lambda (-corou-) (reader -corou- in-stream))))
("Mrs" "Rachel" "Lynde" "lived" "just" "where" "the" "Avonlea" "main" ...)
cl-user(22): (pprint *)

("Mrs" "Rachel" "Lynde" "lived" "just" "where" "the" "Avonlea" "main" "road" "dipped" "down"
"into" "a" "little" "hollow" "fringed" "with" "alders" "and" "ladies" "eardrops" "and"
"traversed" "by" "a" "brook" "that" "had" "its" "source" "away" "back" "in" "the" "woods"
"of" "the" "old" "Cuthbert" "place" "it" "was" "reputed" "to" "be" "an" "intricate"
"headlong" "brook" "in" "its" "earlier" "course" "through" "those" "woods" "with" "dark"
"secrets" "of" "pool" "and" "cascade" "but" "by" "the" "time" "it" "reached" "Lynde" "s"
"Hollow" "it" "was" "a" "quiet" "well" "conducted" "little" "stream" "for" "not" "even" "a"
"brook" "could" "run" "past" "Mrs" "Rachel" "Lynde" "s" "door" "without" "due" "regard" "for"
"decency" "and" "decorum" "it" "probably" "was" "conscious" "that" "Mrs" "Rachel" "was"
"sitting" "at" "her" "window" "keeping" "a" "sharp" "eye" "on" "everything" "that" "passed"
"from" "brooks" "and" "children" "up" "and" "that" "if" "she" "noticed" "anything" "odd" "or"
"out" "of" "place" "she" "would" "never" "rest" "until" "she" "had" "ferreted" "out" "the"
"whys" "and" "wherefores" "thereof")
```

問題10.3 自分の好きな文章（英語）についてファイルを作り，上記reader-writerで単語の切り出しを行え.

問題10.4 見ての通り，このリーダ・ライタはアポストロフィもコンマもピリオドも無視してしまう．これらの特殊文字も１文字だけのトークンとして切り出すように，上記リーダ・ライタを変更せよ.

問題10.5 自分の好きな計算機言語について，そのプログラムコードをトークンに切り出すパーザを設計し，実装せよ.

第11章 Lisp で作る Scheme

> 私達は，関数とアクタをサポートするLispインタプリタを作りました．引数をつけて関数を呼び出したり，メッセージをアクタへ送ることができます．そしてそのインタプリタにアクタが実装された仕方を見て，私達は，たいへん驚きました．ラムダクロージャの実装とアルファアクタの実装は，同一のコードだったのです．
>
> [Guy L. Steele Jr.]（Scheme 過去・現在・未来，井田(訳)）

> Pascalはピラミッド---重い石を定位置へ運ぶ軍隊が作った堂々とした, 息もつけぬ静的構造---を作るためである. Lispは組織---変動する数万の, より単純な器官を定位置へはめ込む連隊が作った堂々とした, 息もつけぬ動的構造---を作るためである. 使われた組織化の原則は二つの場合で同じである. ただ極めて重要な違いがある: 個々のLispプログラマに任された自由裁量で輸出出来る機能は, Pascalの事業で見出されるそれより桁違いに多い. Lispプログラムライブラリはその効用がそれを生み出した応用を超越するような関数で膨張する.
>
> [Alan J. Perlis] (SICP第 2 版序文, 和田(訳))

Gerry Sussman と Guy Steele, Jr. はアクタモデルを理解しようと Carl Hewitt の提唱する言語を Lisp で実装しようとして，たまたま継続を発見し，それが Scheme 開発のきっかけとなった．Lisp コミュニティにおけるそのインパクトは大きかったらしく，その後の Lisp 教科書では，Charniak/Riesbeck/McDermott, Winston，Norvig と3冊の教科書で Lisp による Scheme 実装が取り上げられている．とは言え，その内容は三者三様である．ここでは Norvig の PAIP の内容(http://norvig.com/paip/interp1.lisp, http://norvig.com/paip/interp2.lisp, http://norvig.com/paip/interp3.lisp)にしたがって記述する．ただし，オリジナルではパッケージを使っていないが，ここでは scheme という名前のパッケージを用意して，Scheme 関係のいくつかのシンボルはそちらに intern するようにした．

Lisp で Lisp 風な言語を実装してどうするのかと思われるかも知れないが，この章の目的は，1) REPL の構造を示すことによって Lisp を理解すること，2) Scheme の実装を Common Lisp で示すことで，call/cc（継続）を理解すること，3) 実装を通じて動的束縛と静的束縛を理解すること，などである．実際，Lisp のインタープリタも今では Lisp で実装されている．さらに読者がここで紹介する scheme とは異なる自分流の領域固有のインタープリタを作りたいと思ったときは，本章の内容をベースにそれを開発することができる．なお，Brooks は Lisp の動作を教えるために，Common Lisp で Lisp の簡単なインタプリタを教科書に書いている．

11.1 末尾再帰なしのインタープリタ

Scheme 解釈器の説明に入る前に，必要なシンボルを scheme パッケージ中の外部シンボルとして定義しておこう．defpackage については第20章で説明する．

```
(defpackage scheme
  (:shadow #:if #:lambda #:write)
  (:use :common-lisp)
  (:export #:scheme #:macro #:macro-expand #:begin #:set!
```

```
            #:quote #:null? #:eq? #:equal? #:if #:lambda
            #:write #:display #:newline #:define #:letrec)
  (:documentation "Scheme implemented by Common Lisp."))
```

最初に示すのは，[末尾再帰なしの Scheme 実装]である．以下の補助関数と一緒に，トップレベルの関数 scheme を次のように定義する．

```
(in-package :scheme)

(defun length=1 (x)
  "Is x a list of length 1?"
  (and (consp x) (null (cdr x))))

(defun last1 (list)
  "Return the last element (not last cons cell) of list"
  (first (last list)))

(defun rest2 (x)
  "The rest of a list after the first TWO elements."
  (rest (rest x)))

(defun starts-with (list x)
  "Is x a list whose first element is x?"
  (and (consp list) (eql (first list) x)))

(defun maybe-add (op exps &optional if-nil)
  "For example, (maybe-add 'and exps t) returns
  t if exps is nil, exps if there is only one,
  and (and exp1 exp2...) if there are several exps."
  (cond ((null exps) if-nil)
        ((length=1 exps) (first exps))
        (t (cons op exps))))
```

maybe-add は exps が複数の要素を持つリストであれば，引数 op を exps の最初に追加した書式を返す．exps が要素数 1 のリスト（すなわちシングルトン）であれば，逆にその要素を返す．もし exps が空リストなら， if-nilの値をそのまま返す．

トップレベルで呼び出す関数は次のようなものである．

```
(defun scheme:scheme (&optional x)
  "A Scheme read-eval-print loop (using interp)"
  ;; Modified by norvig Jun 11 96 to handle optional argument
  ;; instead of always going into a loop.
  (init-scheme-interp)
  (cl:if x
      (interp x nil)
    (loop (format t "~&==> ")
          (cl:print (interp (cl:read) nil)))))
```

トップレベルで引数無しで Scheme システムを呼び出すが，上記に示す関数 scheme の最後の行がいわゆる Read-Eval-Print Loop (REPL)の部分である．ここでは Common Lisp の read と print が流用されている．

Schemeシステムを実行すると，プロンプトを印刷してからユーザからの入力待ちとなり，ユーザが何かを入力するとそれはinterpで解釈され，その結果が印刷されて，再びユーザからの入力待ちとなる．

さて，末尾再帰なしのScheme実装における解釈器は以下のようになっている．

```
(defun interp (x &optional env)
  "Interpret (evaluate) the expression <x> in the environment ‹env›."
  (cond
   ((symbolp x) (get-var x env))
   ((atom x) x)
   ((case (first x)
      (cl:quote      (second x))
      (scheme:begin  (last1 (mapcar #'(cl:lambda (y) (interp y env))
                            (rest x))))
      (scheme:set!   (set-var! (second x) (interp (third x) env) env))
      (scheme:if     (cl:if (interp (second x) env)
                        (interp (third x) env)
                      (interp (fourth x) env)))
      (scheme:lambda (let ((parms (second x))
                           (code (maybe-add 'scheme:begin (rest2 x))))
                       #'(cl:lambda (&rest args)
                           (interp code (extend-env parms args env)))))
      (otherwise  ;; a procedure application
       (apply (interp (first x) env)
              (mapcar #'(cl:lambda (v) (interp v env))
                (rest x)))))))
```

これを見ればschemeのインタープリタが何をしているのかは一目了然である．すなわち，もし与えられた書式が，

1. シンボルならget-varでその値を取り出して返す．このとき，もしシンボルが関数名であれば，そのScheme関数名に対応したCommon Lispにおける関数実体が取りだされることを注意しておく．
2. シンボル以外のアトム（たとえば数）なら，それをそのまま返す．
3. リストのcarがcl: quoteだったら，2番目の要素を返す．
4. リストのcarがscheme:beginだったら，残りを順番に解釈して最後の結果を返す．
5. リストのcarがscheme:set!だったら，3番目の解釈した結果を2番目（シンボルのはず）に set-var!で束縛する．
6. リストのcarが scheme:ifだったら，2番目の解釈した結果が真なら3番目を解釈し，偽だったら4番目を解釈する．
7. リストのcarがscheme:lambdaだったら，2番目をパラメータリストとし，それ以降の本体について必要ならscheme:beginをプッシュしたものをコード本体とし，パラメータリストにextend-envで束縛を拡張した結果を新しく 環境envとして本体の解釈をするようなCommon Lisp無名関数を作る．ここで返される無名関数には関数interpが含まれることと，無名関数全体が次のotherwise節のapplyで処理されることを注意しておく．
8. 最後のotherwise節がもっとも一般的な場合であるが，リストのcarがシンボルならそれを関数名としてそのときの環境から関数定義を取りだし，残りをパラメータとして各々を解釈した結果を引数として関数適応する．リストのcarがscheme:lambdaのリストならば返されるCommon Lisp無名関数について関数適応することに注意されたい．

ここで分かるように，第1要素がscheme:set!とか scheme:ifであるリストは，このインタープリタ内で処理される．このようにインタープリタでの処理が特別にコード化されている書式を特殊形式と呼ぶ．ちな

みに，ここで関数適応の apply は Common Lisp の関数であるが，関数適応する前に scheme としての関数について interp で解釈しているから，そのinterp 解釈結果が Common Lisp として apply できさえすればどんなものでもよい（つまりとても複雑な計算の結果としてラムダ文が得られるようなものでもよいという意味．通常は単にシンボルにセットされた関数定義を取り出すだけ）．またここでは，シンボルから `(get-var x env)` で取り出される値が関数適応されるが，これはこの Scheme システムにおけるシンボルの値を取り出すことなので，Scheme では関数用の名前空間はないということも分かる．

最低限の関数

さて最低限の Scheme 関数を実装しなければならないが，上で分かるようにCommon Lisp による Scheme 実装では Common Lisp の関数を Scheme の関数として流用する．以下のコードはそれを行うための仕組みである．

```
(in-package :scheme)

(defparameter *scheme-procs*
  '(+ - * / = < > <= >= cons car cdr not append list read member
      (null? null) (eq? eq) (equal? equal) (eqv? eql)
      (write prin1) (display princ) (newline terpri)))

(defun init-scheme-interp ()
  "Initialize the scheme interpreter with some global variables."
  ;; Define Scheme procedures as CL functions:
  (mapc #'init-scheme-proc *scheme-procs*)
  ;; Define the boolean `constants'. Unfortunately, this won't
  ;; stop someone from saying: (set! t nil)
  (set-global-var! t t)
  (set-global-var! nil nil))

(defun init-scheme-proc (f)
  "Define a Scheme procedure as a corresponding CL function."
  (cl:if (listp f)
      (set-global-var! (first f) (symbol-function (second f)))
    (set-global-var! f (symbol-function f))))
```

`*scheme-procs*` には Scheme の関数から Common Lisp の関数へのマッピング情報が書かれている．ただし，その対応表記は Scheme 関数名と Common Lisp 関数名の二つの関数名を持つリストであるが，同名の関数であればリストではなく関数名アトムとなっている．ここでは，一部 Common Lisp における + や − などの関数シンボルとscheme におけるそれらとを区別していないことを注意しておく．init-scheme-proc はその情報にしたがって，Scheme のグローバル変数として関数定義を行う．init-scheme-interp ではそのほかに t と nil の scheme における定義をする．本当の Scheme なら本来は #t と #f となるべきところであるが，ここでは Lisp と同様の t と nil である．

環境と変数

ここで，普段は目に触れることのない**環境**というものについて説明する．束縛というのは変数とその値のペアのことであり，環境はその時点でアクセス可能な束縛の集まりである．この実装では環境は第1章で説明した連想リストが用いられる．

```
(defun set-var! (var val env)
  "Set a variable to a value, in the given or global environment."
  (cl:if (assoc var env)
      (setf (second (assoc var env)) val)
    (set-global-var! var val))
  val)

(defun get-var (var env)
  "Get the value of a variable, from the given or global environment."
  (cl:if (assoc var env)
      (second (assoc var env))
    (get-global-var var)))

(defun extend-env (vars vals env)
  "Add some variables and values to an environment."
  (nconc (mapcar #'list vars vals) env))
```

get-var は環境中から var の値を取り出す．もし環境中に値が無ければ，get-global-var により大域変数としての値を取り出す．set-var! はもし環境中に値があれば古い値を書き換えてしまう．環境中に値がなければ set-global-var! により大域変数としてその値をセットする．

大域変数とは見ようと思えばどこからでもその値が見える変数のことであるが，この実装では実は Lisp シンボルの gloval-val 属性の値である．大域変数の値を取り出そうとしても値がなければ，未束縛エラーとなる．

```
(defun set-global-var! (var val)
  (setf (get var 'global-val) val))

(defun get-global-var (var)
  (let* ((default "unbound")
         (val (get var 'global-val default)))
    (cl:if (eq val default)
        (error "Unbound scheme variable: ~a" var)
      val)))
```

環境には extend-env により新しい束縛が追加される．この extend-env が実行されるのは，Scheme の REPL でラムダ文が与えられたときであるが，後述のScheme マクロ let と define においてもそれがラムダ文に展開されることを注意しておく．

extend-env は変数名（シンボル）とその値を引数にもらって，そのペアを作って環境の先頭にプッシュする．nconc が用いられているが，define と let の実行時には先頭に追加されるだけでそれまでにプッシュされてきた値が変更されることはなく，もし同名の変数束縛がプッシュされると assoc で取り出されるのは先頭のペアであるので，古い束縛は結果的に隠ぺいされる．ただし，新しい変数束縛ではない変数に関数中で set! を実行すれば，その関数中では自由変数であるが環境中にある値が副作用として変更される．ここでも

し nconc ではなく，push を用いると，scheme:begin のコードが mapcarであるため，うまく副作用が蓄積されないことを注意しておく．

ここまでのところで，以下のように Scheme を実行してみよう．

```
cl-user(17): (scheme:scheme)
==> 486

486
==> (+ 137 349)

486
==> (- 1000 334)

666
==> (* 5 99)

495
==> (/ 10 5)

2
==> (/ 10 6)

5/3
==> (+ 2.7 10)

12.7
==> (+ (* 3 5) (- 10 6))

19
```

問題 11.1 SBCL を使っている方は，一見上記のようには印刷と入力のタイミングが合わなくて，ずれてしまうことを発見してびっくりするかもしれない．これはSBCL では標準出力 *standard-output* 用バッファとターミナル入出力バッファが異なるためである．上記のように印刷するには，(force-output) をある場所においてやれば直る．正しい位置を見つけて印刷と入力のタイミングを修正せよ．

Scheme マクロ

これでこの Scheme インタプリタが働くことは分かったが，まだシンボルの値や関数を定義することはできない．Scheme の define はマクロであり，and，or，cond なども同様にマクロである．次のコードは Scheme マクロを組み込んだ解釈器である．

```
(defun interp (x &optional env)
  "Interpret (evaluate) the expression <x> in the environment <env>.
  This version handles macros."
  (cond
   ((symbolp x) (get-var x env))
   ((atom x) x)
   ((scheme:macro (first x))              ;***
    (interp (scheme:macro-expand x) env)) ;***
   ((case (first x)
      (cl:quote      (second x))
```

```
      (scheme:begin  (last1 (mapcar #'(cl:lambda (y) (interp y env))
                            (rest x))))
      (scheme:set!   (set-var! (second x) (interp (third x) env) env))
      (scheme:if     (cl:if (interp (second x) env)
                            (interp (third x) env)
                          (interp (fourth x) env)))
      (scheme:lambda (let ((parms (second x))
                           (code (maybe-add 'scheme:begin (rest2 x))))
                       #'(cl:lambda (&rest args)
                           (interp code (extend-env parms args env)))))
      (otherwise  ;; a procedure application
       (apply (interp (first x) env)
              (mapcar #'(cl:lambda (v) (interp v env))
                (rest x))))))))
```

Common Lispの関数scheme:macroはシンボルがマクロ定義されているかどうかを調べる．もし書式の第１要素にマクロ定義があれば，scheme:macro-expandでマクロを展開し，その結果について解釈する．

このSchemeシステムにおけるマクロ定義はシンボルのscheme:macro属性に保存されるが，マクロ展開は単に保存された書式を取り出して関数適応してその結果をさらにマクロが無くなるまで，繰り返し展開するだけである．

```
(defun scheme:macro (symbol)
  (and (symbolp symbol) (get symbol 'scheme:macro)))

(defmacro def-scheme-macro (name parmlist &body body)
  "Define a Scheme macro."
  `(setf (get ',name 'scheme:macro)
     #'(cl:lambda ,parmlist .,body)))

(defun scheme:macro-expand (x)
  "Macro-expand this Scheme expression."
  (cl:if (and (listp x) (scheme:macro (first x)))
      (scheme:macro-expand
       (apply (scheme:macro (first x)) (rest x)))
    x))
```

ここで，Schemeマクロの展開形として実際に保存されるものは，実はCommon Lispのクロージャであるが，それはscheme:macro-expandの中のapplyで処理されてSchemeの書式になるということを注意しておく．すなわち，def-scheme-macroするときにはそれに与える書式はScheme書式であるが，それは保存されるときにはラムダ式でラップされてCommon Lispのクロージャとなる．しかし展開されるときにapplyによって中のScheme書式が取り出され，それが解釈器によって解釈される．

次にschemeマクロをいくつか定義する．

```
(def-scheme-macro let (bindings &rest body)
  `((scheme:lambda ,(mapcar #'first bindings) . ,body)
    .,(mapcar #'second bindings)))

(def-scheme-macro let* (bindings &rest body)
  (cl:if (null bindings)
      `(scheme:begin .,body)
    `(let (,(first bindings))
```

```
        (let* ,(rest bindings) . ,body))))

  (def-scheme-macro and (&rest args)
    (cond ((null args) 't)
          ((length=1 args) (first args))
          (t `(scheme:if ,(first args)
                (and . ,(rest args))))))

  (def-scheme-macro or (&rest args)
    (cond ((null args) 'nil)
          ((length=1 args) (first args))
          (t (let ((var (gensym)))
               `(let ((,var ,(first args)))
                  (scheme:if ,var ,var (or . ,(rest args))))))))

  (def-scheme-macro cond (&rest clauses)
    (cond ((null clauses) nil)
          ((length=1 (first clauses))
           `(or ,(first clauses) (cond .,(rest clauses))))
          ((starts-with (first clauses) 'else)
           `(scheme:begin .,(rest (first clauses))))
          (t `(scheme:if ,(first (first clauses))
                (scheme:begin .,(rest (first clauses)))
                (cond .,(rest clauses))))))

  (def-scheme-macro case (key &rest clauses)
    (let ((key-val (gensym "KEY")))
      `(let ((,key-val ,key))
         (cond ,@(mapcar
                    #'(cl:lambda (clause)
                        (cl:if (starts-with clause 'else)
                            clause
                          `((member ,key-val ',(first clause))
                            .,(rest clause))))
                 clauses)))))

  (def-scheme-macro define (name &rest body)
    (cl:if (atom name)
        `(scheme:begin (scheme:set! ,name . ,body) ',name)
      `(define ,(first name)
         (scheme:lambda ,(rest name) . ,body))))

  (def-scheme-macro delay (computation)
    `(scheme:lambda () ,computation))

  (def-scheme-macro letrec (bindings &rest body)
    `(let ,(mapcar #'(cl:lambda (v) (list (first v) nil)) bindings)
       ,@(mapcar #'(cl:lambda (v) `(scheme:set! .,v)) bindings)
       .,body))
```

では，scheme:macro-expand，interp，set-var!にトレースを掛けて，動き方を追跡してみよう．ここではまずin-packageを用いてscheme名前空間に入ってから，scheme実行していることを注意しておく．だから最初の (let ((x 1) (y 2)) (list x y)) の let はCommon LispのletではなくてSchemeのletである．

```
cl-user(2): (in-package :scheme)
#<The scheme package>
scheme(3): (trace macro-expand interp set-var!)
```

```
(set-var! interp macro-expand)
scheme(4): (scheme)
==> (let ((x 1) (y 2)) (list x y))
 0[5]: (interp (let ((x 1) (y 2)) (list x y)) nil)
   1[5]: (macro-expand (let ((x 1) (y 2)) (list x y)))
   1[5]: returned ((lambda (x y) (list x y)) 1 2)
   1[5]: (interp (list x y) ((x 1) (y 2)))
   1[5]: returned (1 2)
 0[5]: returned (1 2)

(1 2)
==> (define size 2)
 0[5]: (interp (define size 2) nil)
   1[5]: (macro-expand (define size 2))
   1[5]: returned (begin (set! size 2) 'size)
   1[5]: (set-var! size 2 nil)
   1[5]: returned 2
 0[5]: returned size

size
==> size
 0[5]: (interp size nil)
 0[5]: returned 2

2
==> (define (square x) (* x x))
 0[5]: (interp (define (square x) (* x x)) nil)
   1[5]: (macro-expand (define (square x) (* x x)))
   1[5]: returned (begin (set! square (lambda (x) (* x x))) 'square)
   1[5]: (set-var! square #<Closure (:internal interp 0) @ #x10242a492> nil)
   1[5]: returned #<Closure (:internal interp 0) @ #x10242a492>
 0[5]: returned square

square
==> (square 3)
 0[5]: (interp (square 3) nil)
   1[5]: (interp (* x x) ((x 3)))
   1[5]: returned 9
 0[5]: returned 9

9
==>
```

Schemeのトップレベルで `(let ((x 1) (y 2)) (list x y))` を与えると，`let` にはSchemeのマクロ定義があるので展開されて `((lambda (x y) (list x y)) 1 2)` となるが，これの第1要素 `(lambda (x y) (list x y))` にはもちろんマクロ定義はなく，展開はそこまでとなる．`(lambda (x y) (list x y))` が解釈器で解釈されて，そこで返されるものは実は

```
#'(cl:lambda (&rest args)
    (interp '(list x y)
            (extend-env '(x y) args env)))
```

というクロージャである．それが返されると結局のところ，`(interp ((lambda (x y) (list x y)) 1 2) nil)` に対して解釈器の最後で次のような Common Lisp の呼び出しが実行される．

```
(cl:apply #'(cl:lambda (&rest args)
              (interp '(list x y)
                  (extend-env '(x y) args nil)))
  '(1 2))
```

一方，(define size 2) では，(begin (set! size 2) 'size) が展開形だが，次の (define (square x) (* x x)) では (begin (set! square '(lambda (x) (* x x))) 'square) が展開形となり，これが解釈実行される．実際に square に保存されるのは，

```
#'(cl:lambda (&rest args)
    (interp '(begin (* x x))
            (extend-env '(x) args env)))
```

であるが，これを実際に実行するときには

```
(apply #'(cl:lambda (&rest args)
           (interp '(begin (* x x))
                   (extend-env '(x) args nil)))
  '(3))
```

から取りだされた次の書式が実行される．

```
(interp '(begin (* x x)) '((x 3)))
```

問題 11.2 ACL ではなぜか scheme:lambda を定義することができず，それはcl:lambda と同じとされてしまう．各々のLisp処理系で以下のようにして，どうなるか試してみよう．

```
cl-user(1): (defpackage scheme (:export #:lambda))
#<The scheme package>
cl-user(2): (inspect 'scheme:lambda)
A tenured symbol lambda @ #x10001c28b
  which is an external symbol in the common-lisp package
   0 type ---------> Bit field: #x0b
   1 flags --------> Bit field: #x00
   2 package ------> The common-lisp package
   3 value --------> ..unbound..
   4 global-value -> ..unbound..
   5 function -----> #<macro lambda @ #x100207b72>
   6 hash ---------> Bit field: #x742d9e1c
   7 name ---------> A simple-string (6) "lambda"
   8 plist --------> (comp::.x-con-handler. ...), a proper list with 14 elements
   9 lock-index ---> symbol nil
[1i] cl-user(3): :i q
lambda
```

問題 11.3 SICPとはエーベルソンとサスマンによる著名な教科書「プログラムの構造と実行」(Structure and Interpretation of Computer Programs)のことである．この教科書はSchemeで書かれているが，長年MITでプログラミングの講義に使われた．SICPの1.1.2項，1.1.4項，1.1.6項の実行例を順に実行せよ．

11.2 末尾再帰のインタープリタ

Common Lisp ではコンパイルしないと末尾再帰の効果はなかった．Scheme ではインタープリタでも末尾再帰を処理する．そのために interp を次のように改造する．

```
(defun interp (x &optional env)
  "Evaluate the expression x in the environment env.
   This version is properly tail-recursive."
  (prog ()
    :INTERP
    (return
      (cond
       ((symbolp x) (get-var x env))
       ((atom x) x)
       ((scheme:macro (first x))
        (setf x (scheme:macro-expand x)) (go :INTERP))
       ((case (first x)
          (quote (second x))
          (scheme:begin (pop x) ; pop off the BEGIN to get at the args
                        ;; Now interpret all but the last expression
                        (loop while (rest x) do (interp (pop x) env))
                        ;; Finally, rename the last expression as x
                        (setf x (first x))
                        (go :INTERP))
          (scheme:set!  (set-var! (second x) (interp (third x) env) env))
          (scheme:if    (setf x (cl:if (interp (second x) env)
                                       (third x)
                                      (fourth x)))
              ;; That is, rename the right expression as x
              (go :INTERP))
          (scheme:lambda (make-proc :env env :parms (second x)
                           :code (maybe-add 'scheme:begin (rest2 x))))
          (otherwise   ;; a procedure application
           (let ((proc (interp (first x) env))
                 (args (mapcar #'(cl:lambda (v) (interp v env)) (rest x))))
             (cl:if (proc-p proc)
                    ;; Execute procedure with rename+goto
                    (progn
                      (setf x (proc-code proc))
                      (setf env (extend-env (proc-parms proc) args
                                            (proc-env proc)))
                      (go :INTERP))
                ;; else apply primitive procedure
                (apply proc args))))))))))

(defstruct (proc (:print-function print-proc))
  "Represent a Scheme procedure"
  code (env nil) (name nil) (parms nil))

(defun print-proc (proc &optional (stream *standard-output*) depth)
  (declare (ignore depth))
  (format stream "{~a}" (or (proc-name proc) '??)))
```

改造部分にはアンダースコアをつけた．(loop ...) のかわりに (prog () :INTERP ...) としたが，scheme:macro の最後，scheme:begin の最後，if の最後，そして cond 文最後の otherwise の一部で (go :INTERP) となっていることに注目されたい．たとえば，scheme:macro ではマクロ展開の結果を interp

することなく，それを持って return を飛び越えて :INTERP に直接 go しているし，scheme:begin では最後の書式を残して順番に interp したのち，最後の書式は return を飛び越えて :INTERP に go している．

このバージョンではラムダ文を読み込むと，Common Lisp のクロージャ（interpを含む）を作るのではなく，構造体 proc の実現を作る．それには関数本体と関数パラメータリストのほかにそのときの環境が保存される．そして関数適応時には，もしそれが構造体 proc の実現ならば本体コードと関数パラメータを取り出して，保存されていた環境に関数パラメータと実引数の値を束縛して追加したものを新しい環境として，interp せずに return を飛び越えて :INTERP に直接 go する．

defstruct定義におけるオプショナルパラメータ:print-functionについては付3.2を参照されたい．

問題 11.4 上記末尾再帰版の解釈器を読み込ませて，scheme がきちんと働くことを確認せよ．

10.3節ではCommon Lispによるコルーチンを実装したが，Schemeであればコルーチンはより自然な形でプログラムできる．

```
scheme(8): (scheme)
==> (define (reader cc)
      (letrec ((mainloop
                 (lambda (obj cc)
                   (cc obj (lambda (-c-)
                             (mainloop (read) -c-))))))
        (mainloop (read) cc)))

reader
==> (define (writer cc)
      (letrec ((writerloop
                 (lambda (thing cc)
                   (cond ((eqv? thing 'done) 'done)
                         (else (write thing)
                               (newline)
                               (write '>)
                               (cc writerloop))))))
        (cc writerloop)))

writer
==> (writer reader)
this
this
>is
is
>a
a
>test.
test.
>done

done
==>
```

問題 11.5 SICPの1.1.7項を実行せよ．ただし，squareは1.1.4項に，absは1.1.6項にあるので，これらを先に定義しておく．

11.3 call/cc によるインタープリタ

最後に，継続すなわちcall/cc を用いたバージョンを次に示す．

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :interp1))

(defpackage scheme
  (:export #:call/cc #:call-with-current-continuation))

(in-package :scheme)

(defun interp (x env cc)
  "Evaluate the expression x in the environment env,
   and pass the result to the continuation cc."
  (cond
   ((symbolp x) (funcall cc (get-var x env)))
   ((atom x)    (funcall cc x))
   ((scheme:macro (first x))
    (interp (scheme:macro-expand x) env cc))
   ((case (first x)
      (cl:quote     (funcall cc (second x)))
      (scheme:begin (interp-begin (rest x) env cc))
      (scheme:set!  (interp (third x) env
                            #'(cl:lambda (val)
                                (funcall cc (set-var! (second x)
                                                      val env)))))
      (scheme:if    (interp (second x) env
                            #'(cl:lambda (pred)
                                (interp (cl:if pred (third x) (fourth x))
                                        env cc))))
      (scheme:lambda (let ((parms (second x))
                           (code (maybe-add 'scheme:begin (rest2 x))))
                       (funcall
                        cc
                        #'(cl:lambda (cont &rest args)
                            (interp code
                                    (extend-env parms args env)
                                    cont)))))
      (otherwise (interp-call x env cc))))))
```

このバージョンでは `interp` は常に継続 `cc` を持ちまわるが，最初の版で再帰しないところ (symbolp, atom，quote，lambda) では，しかるべき値で `cc` が実行され（これが値を返すことに相当する），`scheme:macro` では初版と同様にマクロ展開した結果を持って再帰する．一方，`scheme:set!` ではその書式の3番目の値（セットされる値）を評価するが，そこに与えられる `cc` は値を受け取ったらそれを `set-var!` するような継続だ．同様に，`if` 書式では条件文を `interp` するがそこに与えられる継続はその結果を受け取ってthen部かelse部を実行して再帰するような継続だ．一般の関数実行書式 (`otherwise`部分)では以下のようなコードを実行する．（呼出し可能なように以下を追記する．）

```
(defun interp-call (call env cc)
  "Interpret the call (f x...) and pass the result to CC."
  (map-interp call env
              #'(cl:lambda (fn-and-args)
                  (apply (first fn-and-args)
                         cc
                         (rest fn-and-args)))))

(defun map-interp (list env cc)
  "Interpret each element of LIST, and pass the list to CC."
  (cl:if (null list)
      (funcall cc nil)
    (interp (first list) env
            #'(cl:lambda (x)
                (map-interp (rest list) env
                            #'(cl:lambda (y)
                                (funcall cc (cons x y))))))))
```

map-interp は実はリストの要素を先頭から順に解釈しながら，それを継続に渡していって最後にリストとして返すような関数である．interp-call はそれを呼び出すだけである．

interp-begin のやることも map-interp と似ている．

```
(defun interp-begin (body env cc)
  "Interpret each element of BODY, passing the last to CC."
  (interp (first body) env
          #'(cl:lambda (val)
              (cl:if (null (rest body))
                  (funcall cc val) ;; fix, hsuc 2/20/93; forgot to call cc
                (interp-begin (rest body) env cc)))))
```

さてこれまでのところ，Schemeにおける継続 call/cc は現れていなかった．ここでSchemeプログラマーが明示的に call/cc を呼び出せるように，追加修正を行う．

```
(defun scheme:scheme (&optional x)
  "A Scheme read-eval-print loop (using interp).
   Handles call/cc by explicitly passing continuations."
  ;; Modified by norvig Jun 11 96 to handle optional argument
  ;; instead of always going into a loop.
  (init-scheme-interp)
  (cl:if x
      (interp x nil #'cl:print)
    (loop (format t "~&==> ")
      (interp (cl:read) nil #'cl:print))))

(defun init-scheme-proc (f)
  "Define a Scheme primitive procedure as a CL function."
  (cl:if (listp f)
      (set-global-var! (first f)
                       #'(cl:lambda (cont &rest args)
                           (funcall cont (apply (second f) args))))
    (init-scheme-proc (list f f))))

(defun call/cc (cc computation)
  "Make the continuation accessible to a Scheme procedure."
  (funcall computation cc
```

```
            ;; Package up CC into a Scheme function:
            #'(cl:lambda (cont val)
                (declare (ignore cont))
                (funcall cc val))))

  ;; Now install call/cc in the global environment
  (set-global-var! 'scheme:call/cc #'call/cc)
  (set-global-var! 'scheme:call-with-current-continuation #'call/cc)
```

interp には三番目の引数として継続が与えられなければならない．scheme はトップの interp の呼び出しで print を与えるように修正された．init-scheme-proc の引数には第1要素が Scheme 関数名，第2要素が対応する Common Lisp 関数名のリストだが，Common Lisp 関数はそれが第1引数に継続を引き受けるようなクロージャに変更された．関数 call/cc はCommon Lisp 関数だが，その第1引数が継続，第2引数は計算オブジェクトである．

この解釈器の動きをトレースしてみよう．

```
cl-user(2): (in-package :scheme)
#<The scheme package>
scheme(3): (trace macro-expand interp interp-call call/cc)
(call/cc interp-call interp macro-expand)
scheme(4): (scheme)
==> (+ 1 20 300)
 0[5]: (interp (+ 1 20 300) nil #<Function print>)
   1[5]: (interp-call (+ 1 20 300) nil #<Function print>)
     2[5]: (interp + nil #<Closure (:internal map-interp 0) @ #x102443682>)
       3[5]: (interp 1 nil #<Closure (:internal map-interp 0) @ #x102443a32>)
         4[5]: (interp 20 nil #<Closure (:internal map-interp 0) @ #x102443de2>)
           5[5]: (interp 300 nil #<Closure (:internal map-interp 0) @ #x102444192>)

321
           5[5]: returned 321
         4[5]: returned 321
       3[5]: returned 321
     2[5]: returned 321
   1[5]: returned 321
 0[5]: returned 321
```

call/cc を使う簡単なSchemeプログラム例を以下に示す．

```
==> (+ 1 (call/cc (lambda (cc) (+ 20 300))))
 0[5]: (interp (+ 1 (call/cc (lambda (cc) (+ 20 300)))) nil #<Function print>)
   1[5]: (interp-call (+ 1 (call/cc (lambda (cc) (+ 20 300)))) nil #<Function print>)
     2[5]: (interp + nil #<Closure (:internal map-interp 0) @ #x1024955c2>)
       3[5]: (interp 1 nil #<Closure (:internal map-interp 0) @ #x102495972>)
         4[5]: (interp (call/cc (lambda (cc) (+ 20 300))) nil #<Closure (:internal map-interp
0) @ #x102495d22>)
           5[5]: (interp-call (call/cc (lambda (cc) (+ 20 300))) nil
                              #<Closure (:internal map-interp 0) @ #x102495d22>)
             6[5]: (interp call/cc nil #<Closure (:internal map-interp 0) @ #x102496402>)
               7[5]: (interp (lambda (cc) (+ 20 300)) nil #<Closure (:internal map-interp 0)
@ #x1024967b2>)
                 8[5]: (call/cc #<Closure (:internal map-interp 0) @ #x102495d22>
                                #<Closure (:internal interp 2) @ #x102496b02>)
                   9[5]: (interp (+ 20 300) ((cc #<Closure (:internal call/cc 0) @
```

```
#x102496e72>))
                                  #<Closure (:internal map-interp 0) @ #x102495d22>)
                      10[5]: (interp-call (+ 20 300) ((cc #<Closure (:internal call/cc 0) @
#x102496e72>))
                                     #<Closure (:internal map-interp 0) @ #x102495d22>)
                        11[5]: (interp + ((cc #<Closure (:internal call/cc 0) @ #x102496e72>))
                                       #<Closure (:internal map-interp 0) @ #x102497492>)
                          12[5]: (interp 20 ((cc #<Closure (:internal call/cc 0) @
#x102496e72>))
                                        #<Closure (:internal map-interp 0) @ #x102497842>)
                            13[5]: (interp 300 ((cc #<Closure (:internal call/cc 0) @
#x102496e72>))
                                          #<Closure (:internal map-interp 0) @ #x102497bf2>)

321
                            13[5]: returned 321
                          12[5]: returned 321
                        11[5]: returned 321
                      10[5]: returned 321
                    9[5]: returned 321
                  8[5]: returned 321
                7[5]: returned 321
              6[5]: returned 321
            5[5]: returned 321
          4[5]: returned 321
        3[5]: returned 321
      2[5]: returned 321
    1[5]: returned 321
 0[5]: returned 321
```

call/ccによる非局所的脱出

(call/cc 《手続き》)というSchemeの手続き呼び出しで，それが実行された時点での継続，すなわちその時点でやり残されている計算手続き，を獲得し，それが引数である手続きに渡される．ここで渡される継続というものは通常は一つの引数を値として受けとるようなものである．上記実行例では，`(call/cc (lambda (cc) (+ 20 300)))` で計算途中の継続が `cc` に渡されるが，特に何をすることもなく加算の計算が実行される．しかし，継続を利用して，プログラムによって任意の計算途中時点で非局所的な脱出をプログラムできる．以下のプログラム例を見られたい．

3.1節でour-memberを示した．ここではそれをcall/ccを用いてプログラムしてみよう．

```
==> (define (member/cc itm lst)
      (call/cc
       (lambda (cc)
         (cond ((null? lst) nil)
               ((equal? itm (car lst)) (cc lst))
               (else (member/cc itm (cdr lst)))))))

member/cc
==> (member/cc 'a '(a b c))

(a b c)
==> (member/cc 'd '(a b c))
```

```
nil
==> (member/cc 'b '(a b c))

(b c)
==>
```

プログラムの構造は3.1節に示したour-memberと類似しているが，リストの第1要素にメンバーを発見した時にはすぐさま継続にそのリストを渡している．そこでそのリストの値が継続すなわち上記例ではSchemeのREPLにおけるmember/ccの呼び出し手続きに返される．

10.2節では Common Lisp における継続による局所脱出例として sum-1-to-n をあげた．ここでは call/cc を用いて同様なことをプログラムしてみる．

```
==> (set! sum 0)

0
==> (define (sum-1-to-n/cc n)
        (call/cc
         (lambda (cc)
                 (cond ((= n 0) (cc sum))
                       (else (display n)
                             (set! sum (+ n sum))
                             (set! next (lambda () (sum-1-to-n/cc (- n 1))))
                             t)))))
sum-1-to-n/cc
==> (sum-1-to-n/cc 5)
5
t
==> (next)
4
t
==> (next)
3
t
==> (next)
2
t
==> (next)
1
t
==> (next)

15
==>
```

call/ccを用いたreader/writer

call/ccは実行されるとその時点での将来の計算を捕捉する．call/ccの引数として与えるのは任意の計算オブジェクトであるが，(call/cc (lambda (cc) ...)という形式は，与えられた計算オブジェクトであるlambda式がcall/ccで捕捉された将来の計算を引数に受け取る．将来の計算というのは，Lispでは計算は常に内側から外側に向かって実行されるため，実のところcall/ccが実行されたときの呼び出しの外側の計算のことである．call/ccを用いると先のreader/writerの書式はさらに簡単になる．

```
cl-user(1): (in-package :scheme)
#<The scheme package>
scheme(2): (scheme)
==> (define (reader)
      (call/cc (lambda (cc) (cc (read)))))

reader
==> (define (writer reader)
      (call/cc (lambda (cc)
                 (let ((thing (reader)))
                   (cond ((eqv? thing 'done) (cc 'done))
                         (else (write thing)
                               (newline)
                               (write '>)
                               (writer reader)))))))

writer
==> (writer reader)
this
this
>is
is
>a
a
>test.
test.
>done

done
==>
```

上記例では，`(writer reader)` のトップレベルの呼び出しでwriter側が受け取る継続はschemeにおけるREPLのことになるが，reader側が受け取る継続は `(let ((thing (reader)))` のreader呼び出し部分であり，その継続に `(cc (read))` のところで `read` で読み込んだ結果でもって継続実行している．これが結果的に `thing` の束縛値となる．ただし，このscheme実装ではインタプリタとしては最後のwriter末尾再帰をうまくGOTO文に処理してくれるわけではなく，call/ccの呼び出しスタックもどんどん深くなるので，入出力を繰り返しているといずれスタックオーバーフローになる．

11.4 動的束縛の実装

これまでも述べてきたように，Common Lisp以前のLispでは変数の束縛形式は動的束縛であった（厳密にはNILという名前の処理系以前のLispというべきであるが）．Lisp1.5の後継となるLisp2において静的束縛が実装されたようだが，これは失敗したプロジェクトと言われている．その後，Schemeでは静的束縛が標準となり，Common Lispにも影響を及ぼして，今日ではLispでも静的束縛が標準となっている．

WinstonのLisp教科書でも，第1版は動的束縛のMacLispをベースに第23章の「LISPで書くLISP」において静的束縛についても実装しているし，第3版では静的束縛のCommon Lispをベースに第18章の「Lispで書くLisp」において動的束縛について実装している．ところが，ここで取り上げたPAIPの例では，動的束縛の実

現について一切情報がない．そこで，Scheme実装について豊富な経験をお持ちの初谷久史氏にお願いして，PAIPに合わせた動的束縛実装についてプログラムを提供して頂いた．

動的束縛のみの実装

最初は静的束縛にかえて，動的束縛のみの実装である．11.1節のSchemeマクロ版インタープリタをベースに静的束縛ではなく動的束縛のコードにしたものが，以下のコードである．

```
(defun interp (x &optional env)
  "Interpret (evaluate) the expression <<x>> in the environment
   <<env>>. This version handles macros."
  (cond
        ((symbolp x) (get-var x env))
        ((atom x) x)
        ((scheme:macro (first x))
         (interp (scheme:macro-expand x) env))
        ((case (first x)
           (quote  (second x))
           (scheme:begin  (last1 (mapcar #'(lambda (y) (interp y env))
                                 (rest x))))
           (scheme:set!   (set-var! (second x) (interp (third x) env) env))
           (if            (if (interp (second x) env)
                              (interp (third x) env)
                            (interp (fourth x) env)))
           (scheme:lambda (let ((parms (second x))
                                (code (maybe-add 'scheme:begin (rest2 x))))
                            #'(lambda (env &rest args) ; ++++
                                (interp code (extend-env parms args env)))))
           (otherwise ;; a procedure application
                (apply (interp (first x) env)
                           env                          ; ++++
                           (mapcar #'(lambda (v) (interp v env))
                             (rest x)))))))
```

ここで「++++」のコメントをつけた行が修正したところである．すなわち，scheme:lambdaを検出すると関数interpは意味的にはSchemeのクロージャとなるところのCommon Lispクロージャを生成するが，ここで環境envを受け取るようにする．だから`(interp code (extend-env parms args env))`の書式におけるenvはこのCommon Lispクロージャ実行時に渡ってくる環境である．そしてcase文のotherwise節で関数実行時の環境を渡してやる．

この変更に伴って，SchemeマクロからCommon Lisp関数マッピングにおいても，追加の動的環境を受け取るように変更する．

```
(defun init-scheme-proc (f)
  "Define a Scheme primitive procedure as a CL function."
  (if (listp f)
      (set-global-var! (first f)
                       #'(lambda (env &rest args)       ; ++++
                           (apply (second f) args)))    ; ++++
      (init-scheme-proc (list f f))))                   ; ++++
```

これだけの変更で，動的束縛のSchemeができあがる．ただし，この実装は静的束縛を基本とするCommon Lispで動的束縛を実現するときのやり方であることを重ねて注意しておく．

4.1節の自由変数の説明において，ラムダ変数であるrateが静的束縛であるためにエラーとなる例を示した．ところが動的束縛では関数呼び出し時のラムダ変数は動的にスタックに積まれることになり，その関数内部で呼び出される関数では自由変数であるはずのものでもそれが動的環境で見えればエラーとはならない．動的束縛Schemeでの以下の例を参照されたい．

```
cl-user(10): (in-package :scheme)
#<The scheme package>
scheme(11): (scheme)
==> (define (interest bal)
        (* bal rate))

interest
==> (define (newbalance balance rate)
        (+ balance (interest balance)))

newbalance
==> (newbalance 100.0 0.1)

110.0
==>
```

動的束縛をオプションとするScheme

Common Lispでは特殊変数とspecial宣言することによって動的変数を作りだすことができた．ところが静的束縛を原則とするSchemeにおいては，動的束縛を作りだす標準的な指定方法がない．ここではSICP第1版問題4.18の書き方による動的変数の指定の仕方を採用することにした．すなわち，動的変数としたい変数にはdynamicという修飾子をつけてやり，それを参照するときにはdynamic-referenceという特殊形式を用いることにする．そのために，解釈関数interpを次のように変更する．

```
(defun interp (x &optional lenv denv)
  "Interpret (evaluate) the expression <<x>> in the static environment <<lenv>> and the
dynamic environment <<denv>>.
   This version handles macros."
  (cond
   ((symbolp x)
    (let ((p (assoc x lenv)))
      (if (and p (eq (second p) :lexical))
          (third p)
        (get-var x denv))))
   ((atom x) x)
   ((scheme:macro (first x))
    (interp (scheme:macro-expand x) lenv denv))
   ((case (first x)
      (quote  (second x))
      (scheme:begin  (last1 (mapcar #'(lambda (y) (interp y lenv denv))
                                    (rest x))))
      (scheme:set!
       (let ((var (second x))
             (val (interp (third x) lenv denv)))
```

```
      (let ((p (assoc var lenv)))
        (if (and p (eq (second p) :lexical))
            (setf (third p) val)
          (set-var! var val denv)))))
    (if
        (if (interp (second x) lenv denv)
            (interp (third x) lenv denv)
          (interp (fourth x) lenv denv)))
    (scheme:lambda (make-proc :env lenv :parms (second x)
                              :code (maybe-add 'scheme:begin (rest2 x))))
    (scheme:dynamic-reference (get-var (second x) denv))
    (otherwise ;; a procedure application
     (let ((proc (interp (first x) lenv denv))
           (args (mapcar #'(lambda (v) (interp v lenv denv)) (rest x))))
       (if (proc-p proc)
           (interp-proc proc args denv)
         (apply proc args))))))))
```

この実装では静的束縛のための環境 lenv に加えて，動的束縛のための環境 denv がある．関数interpはこの両方の環境を持ちまわるが，ラムダ文として手続きが定義されるときにはその時点での静的環境が手続きに与えられる．束縛を調べる時は，必ず最初に静的環境を調べて，そこに情報がないときにのみ次に動的環境を調べることとする．しかも字句スコーピングで調べたときに変数が静的か動的かを判別できるように，静的環境 lenv において静的か動的かの情報も変数に持たせるようにする．たとえば次のような lenv の場合を考えてみよう．

```
((x :lexical 2) (y :lexical 1) (x :lexical 0) (x :dynamic))
```

最初に動的変数 x としての環境があり，次に通常の静的束縛 (x 0) と (y 1) があり，その環境の更に内側で (x 2) の束縛があったことになる．この束縛環境で通常の参照であれば x の値は 2 であるが，特に (scheme:dynamic-reference x) であれば，動的環境における束縛値が得られる．

静的束縛がなく動的束縛だけの場合には，静的環境の代わりに動的環境のみ用意して解釈すればよかったが，静的束縛と動的束縛の両方ある場合には，ある手続きの定義時と実行時の両方の環境を管理しなければならない．そのためにこの実装では手続きprocを構造体としてそこに静的束縛環境情報を付与している．

```
(defstruct (proc (:print-function print-proc))
  "Represent a Scheme procedure"
  code (env nil) (name nil) (parms nil))

(defun print-proc (proc &optional (stream *standard-output*) depth)
  (declare (ignore depth))
  (format stream "#<proc:~s>" (proc-name proc)))
```

手続きを実行する際には，その手続きに付与された静的環境を取り出して，実行時の動的環境と一緒にして環境を適性な状態に戻してから，パラメータによる環境の拡張をする．それを実行するのが以下の関数である．

```
(defun interp-proc (proc args denv)
  (let ((envs (extend-env (proc-parms proc) args
                          (proc-env proc) denv)))
```

```
    (interp (proc-code proc) (first envs) (second envs))))

(defun extend-env (parms args lenv denv)
  (if (null parms)
      (list lenv denv)
    (let ((p (first parms)))
      (cond ((symbolp p)
             (extend-env (rest parms) (rest args)
                         (cons (list p :lexical (first args)) lenv)
                         denv))
            ((and (consp p) (eq (first p) 'scheme:dynamic))
             (let ((var (second p)))
               (extend-env (rest parms) (rest args)
                           (cons (list var :dynamic) lenv)
                           (cons (list var (first args)) denv))))
            (t
             (error "invalid parameter specifier" p))))))
```

先の利率計算の実行例は以下のようになる.

```
cl-user(2): (in-package :scheme)
#<The scheme package>
scheme(3): (scheme)
==> (define (interest bal)
      (* bal rate))

interest
==> (define (newbalance balance (dynamic rate))
      (+ balance (interest balance)))

newbalance
==> (newbalance 100.0 0.1)

110.0
==>
```

この実行例では関数 newbalance のパラメータ rate は動的束縛と宣言されたが，ほかに静的束縛の同名の変数がないため，参照時に特に (dynamic-reference rate) としなくてもよかった．しかし，動的束縛と静的束縛の両方があるときには，動的静的にかかわらず最も内側のスコープの変数が優先されることを注意しておく.

第12章 配列，文字列，列

12.1 配列

Common Lispの**配列**(array)は非常に汎用にできており，いかなるデータ型の要素を持つ多次元行列でも実現できるし，その要素に異なるデータ型が混在していてもよい.

九九の表を配列で作るには次のようにする.

```
cl-user(3): (setq multiplication-table
                  (make-array '(9 9) :element-type 'integer))
#2A((nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil))
cl-user(4): (loop for i from 1 to 9 do
            (loop for j from 1 to 9 do
              (setf (aref multiplication-table (1- i) (1- j))
                    (* i j))))
nil
cl-user(5): multiplication-table
#2A((1 2 3 4 5 6 7 8 9)
    (2 4 6 8 10 12 14 16 18)
    (3 6 9 12 15 18 21 24 27)
    (4 8 12 16 20 24 28 32 36)
    (5 10 15 20 25 30 35 40 45)
    (6 12 18 24 30 36 42 48 54)
    (7 14 21 28 35 42 49 56 63)
    (8 16 24 32 40 48 56 64 72)
    (9 18 27 36 45 54 63 72 81))
```

このように，もし要素の型が決まっていれば，その型をキーワードパラメータ `:element-type` で指定する．配列の要素を取り出すには `(aref 《配列》 i j)` のように関数arefを用いるが，上記例4行目で分かるように，Common Lispでは配列の添字は常に0から始まることに注意されたい.

配列の次元数を求めるにはarray-rank，そのある次元の大きさを求めるにはarray-dimension，あるいはarray-dimensions，配列の要素の型情報を求めるにはarray-element-typeを用いる.

```
cl-user(6): (array-rank multiplication-table)
2
cl-user(7): (array-dimension multiplication-table 0)
9
cl-user(8): (array-dimension multiplication-table 1)
9
cl-user(9): (array-dimensions multiplication-table)
(9 9)
```

```
cl-user(10): (array-element-type multiplication-table)
t
```

おやおや，実施例10では integer が返されると思いきや，そうではない．型 t は第13章で詳しく述べるが，型階層の頂点にある型である．では，次のようにして指定とは異なるデータ型をセットしてみる．

```
cl-user(11): (setf (aref multiplication-table 0 0)
                   'this-is-not-an-integer)
this-is-not-an-integer
cl-user(12): multiplication-table
#2A((this-is-not-an-integer 2 3 4 5 6 7 8 9)
    (2 4 6 8 10 12 14 16 18)
    (3 6 9 12 15 18 21 24 27)
    (4 8 12 16 20 24 28 32 36)
    (5 10 15 20 25 30 35 40 45)
    (6 12 18 24 30 36 42 48 54)
    (7 14 21 28 35 42 49 56 63)
    (8 16 24 32 40 48 56 64 72)
    (9 18 27 36 45 54 63 72 81))
```

少なくとも，ACLとSBCLでは既に作られた配列の要素に値をセットする場合，要素の型指定は制約とはなっていない．ただし，make-array 時に各要素の初期値を同時に :initial-element でセットする場合には，ACL では make-array 関数の中ではチェックされるが :initial-contents の場合にはやはり素通りである．

```
cl-user(13): (make-array '(3 3) :initial-element 0.1
                                :element-type 'integer)
Error: Initial element 0.1 is not of type integer
[condition type: type-error]
cl-user(14): (make-array '(3 3)
                :initial-contents '((1 2 3) (2 4 6) (3 6 9))
                :element-type 'float)
#2A((1 2 3) (2 4 6) (3 6 9))
```

これ以上はアドバンストな議論になるので，13.2 節の Type Upgrading の項で詳述する．ちなみに，Common Lisp では配列の最大の大きさには上限があり，ある処理系における上限は次のような定数を見れば分かる．

```
cl-user(15): array-rank-limit
65536
cl-user(16): array-dimension-limit
1152921504606846975
cl-user(17): array-total-size-limit
1152921504606846975
cl-user(18): (array-total-size multiplication-table)
81
```

ここで，array-total-size というのはその配列の全要素数を返す関数である．

12.2 ベクタ

ベクタ(vector)とは，1次元配列のことである．

```
cl-user(23): (setq foo (make-array '(10)))
#(nil nil nil nil nil nil nil nil nil nil)
```

```
cl-user(24): (typep foo 'vector)
t
```

したがって，vector は array の副型(subtype)であるが，それ自体として取り上げる価値があるものとして，make-array のかわりに関数 vector でベクタを生成することができ，aref の代わりにアクセッサとして svref が用意されている．

```
cl-user(25): (setq bar (vector 0 1 2 3 4 5 6 7 8 9))
#(0 1 2 3 4 5 6 7 8 9)
cl-user(26): (aref bar 9)
9
cl-user(27): (svref bar 9)
9
cl-user(28): (typep bar 'vector)
t
cl-user(29): (typep bar 'array)
t
cl-user(30): (subtypep 'vector 'array)
t
t
```

今までのところ，出てきた配列やベクタはすべて**単純配列**(simple array)であり，**単純ベクタ**(simple vector)である．単純配列というのは何かというと，「他の配列により共有されず，フィルポインタを持たず，また生成後動的に大きさが変わらない配列」(*CLtL2*, Section 2.5)のことであり，単純ベクタというのはそういうベクタのことである．実は svref というのは simple vector reference の意味であり，単純ベクタでないと使えない．以下に単純ではない配列について各々説明する．

12.3 配列の共有，フィルポインタ，可変配列

displaced-to

配列の共有をするには，新しく配列を定義 (make-array) するときに displaced-to キーワードパラメータに既存の配列を指示してやればよい．配列の共有といってもそれは，実のところ次元数やサイズなどを除いた配列の実体へのポインタの共有である．何次元の配列であろうと，そのメモリ上の実体はメモリサイズ単位で順番に並んだアドレスになっている．だからアクセスするときの添え字さえ計算のつじつまが合えば，次元数を変えても望みの要素にアクセスできる．

```
cl-user(40): multiplication-table
#2A((1 2 3 4 5 6 7 8 9)
    (2 4 6 8 10 12 14 16 18)
    (3 6 9 12 15 18 21 24 27)
    (4 8 12 16 20 24 28 32 36)
    (5 10 15 20 25 30 35 40 45)
    (6 12 18 24 30 36 42 48 54)
    (7 14 21 28 35 42 49 56 63)
    (8 16 24 32 40 48 56 64 72)
    (9 18 27 36 45 54 63 72 81))
cl-user(41): (setq serial-table
                   (make-array '(81)
                     :displaced-to
                     multiplication-table))
```

```
#(1 2 3 4 5 6 7 8 9 2 ...)
cl-user(42): (aref serial-table 80)
81
cl-user(43): (setf (aref multiplication-table 8 8)
                   '9by9)
9by9
cl-user(44): (aref serial-table 80)
9by9
```

フィルポインタ

フィルポインタ(fill pointer)とはベクタ中に順番に値を保存したり取り出したりするときに用いられる添字を表す数であり，通常は先入れ後出しのスタックをベクタで実現するときに用いられる．

```
cl-user(58): (setq one2ten
                   (make-array '(10) :fill-pointer 0))
#()
cl-user(59): (loop for i from 1 to 10
                do (vector-push i one2ten))
nil
cl-user(60): one2ten
#(1 2 3 4 5 6 7 8 9 10)
cl-user(61): (vector-pop one2ten)
10
cl-user(62): (vector-pop one2ten)
9
cl-user(63): one2ten
#(1 2 3 4 5 6 7 8)
cl-user(64): (vector-push 'nine one2ten)
8
cl-user(65): (vector-push 'ten one2ten)
9
cl-user(66): one2ten
#(1 2 3 4 5 6 7 8 nine ten)
cl-user(67): (vector-push 'eleven one2ten)
nil
```

フィルポインタを備えた配列は make-array 時にキーワードパラメータ :fill-pointer を使って初期値を与えると作られる．ただし，フィルポインタが通常の使い方で役に立つのは1次元の配列，すなわちベクタである．興味深いのは長さ10のベクタを定義しても，フィルポインタを有するベクタはそれがフィルポインタの長さしか持たないかのように印刷されることである．上記実施例の23と58を比較されたい．関数 vector-push と vector-pop ではフィルポインタの増減は自動的に行われることを上記例で観察されたい．vector-pop の返す値はもちろん取り出されたスタックの中身であるが，vector-push の返す値は直前のフィルポインタの値である．

フィルポインタとは常に次にフィルすべき場所を示す数値であることに注意されたい．ベクタの物理的に可能な長さを超えてプッシュすることはできず，その場合には vector-push は nil を返す．

可変配列

可変配列(adjustable array)とは make-array 後でも必要とあらばその大きさを変更できるような配列のことである．ただし，そのためには最初から adjustable であると言っておかなければならず，adjustable でない配列に対してあとから adjustable にすることはできない．

```
cl-user(68): (adjust-array one2ten '(20))
#(1 2 3 4 5 6 7 8 nine ten)
cl-user(69): (vector-push 'eleven one2ten)
nil
cl-user(70): one2ten
#(1 2 3 4 5 6 7 8 nine ten)
cl-user(71): (vector-push-extend 'eleven one2ten)
Error: Vector-push-extend cannot extend a non adjustable array: #<Vector @ 1x1022833c2>
[condition type: simple-error]
```

最初から adjustable なベクタに対してなら，vector-push-extend は必要とあらば adjust-array を用いて自動的にサイズを(適当に)拡張し，プッシュを実行してくれる．

```
cl-user(72): (setq one2ten
                   (make-array '(10) :fill-pointer 0
                                     :adjustable t))
#()
cl-user(73): (loop for i from 1 to 20
                 do (vector-push i one2ten))
nil
cl-user(74): one2ten
#(1 2 3 4 5 6 7 8 9 10)
cl-user(75): (loop for i from 11 to 20
                 do (vector-push-extend i one2ten))
nil
cl-user(76): one2ten
#(1 2 3 4 5 6 7 8 9 10 ...)
cl-user(77): (pprint one2ten)

#(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20)
```

12.4 配列の型階層と特殊配列

ここまで，一般の配列と一次元の配列すなわちベクタと，単純配列および単純ベクタについて述べてきた．単純配列は一般配列の副型(subtype)であり，単純ベクタは一般ベクタの副型である．一般のベクタはもちろん一般配列の副型である

```
cl-user(1): (subtypep 'vector 'array)
t
t
cl-user(2): (subtypep 'simple-array 'array)
t
t
cl-user(3): (subtypep 'simple-vector 'vector)
t
t
cl-user(4): (subtypep 'simple-vector 'simple-array)
t
t
```

この型階層の議論は主に配列の要素型以外の属性に関する議論だが，これとは別に配列の要素の型を指定した配列やベクタを**特殊化配列**(specialized array)とか**特殊化ベクタ**(specialized vector)と呼ぶ．しかし make-array で要素の型を指定しようとも，それでその特殊化された型が自動的に生じるわけではない．だから，理論的にはしかるべき型指定子 (array integer) に所属するべきあるいはするべきではない場合でも，実際の型判定では理想とは異なる現実がある．とくに ACL の型判定では注意が必要である．

```
cl-user(12): (typep (make-array '(9 9) :element-type 'integer)
                    '(array integer))
t
cl-user(13): (typep (make-array '(9 9) :element-type 'float)
                    '(array integer))
t
```

詳細な議論は13.2節の Type Upgrading の項で述べる．ただし，要素が文字であるベクタは特別に**文字列**(string)として定義され，ビットの並びは特別に**ビット配列**(bit array)として扱われる．ただし，bit-array という型はなく型指定子としてあるのは bit-vector だけである．**ビットベクタ**(bit vector)というのは1次元のビット配列のことである．**単純文字列**(simple string)とは単純ベクタでかつその要素が文字であるようなものであり，**単純ビットベクタ**(simple bit vector)というのは単純ベクタでかつその要素がビットであるようなものである．simple-string型は string型の副型で，string型は vector型の副型であるにもかかわらず simple-string型は simple-vector型の副型ではない．また，simple-bit-vector型は bit-vector型の副型で，bit-vector型は vector型の副型であるけれども，simple-bit-vector型は simple-vector の副型ではない．このあたりの型階層は大変ややこしい．13.2節で Common Lisp 全体の型階層について言及するが，その一部となる配列の部分だけについて以下に示す．

```
(array (vector (bit-vector (simple-bit-vector))
               (string (base-string
                        (simple-string (simple-base-string))))
               (simple-vector))
       (simple-array (simple-vector)
                     (simple-string)
                     (simple-bit-vector)))
```

文字列ならわざわざ make-string や vector を使わなくても直接ダブルクォーテーションを使って書き下すことができるように，ビット列は `#*` を頭につけて書き下すことができる．

```
cl-user(7): (+ (* 192 256 256 256) (* 168 256 256) (* 1 256) 1)
3232235777
cl-user(8): (format nil "~B" *)
"11000000101010000000000100000001"
cl-user(9): (typep #*11000000101010000000000100000001
                   'simple-bit-vector)
t
cl-user(10): (typep #*11000000101010000000000100000001
                    'simple-vector)
nil
cl-user(11): (typep "This is a string."
                    'simple-string)
t
cl-user(12): (typep "This is a string."
```

```
                    'simple-vector)
nil
```

12.5 文字と文字列

Common Lisp の**文字列**(string)は単なるメモリ上における文字コードを表すバイトの並びではなく，文字オブジェクトの1次元配列(ベクタ)である．だから文字列中の文字にアクセスするのに aref を用いることもできるが，通常はアクセッサとして関数 char を用いる．

```
cl-user(15): (aref "This is a string." 0)
#\T
cl-user(16): (char "This is a string." 1)
#\h
```

Common Lisp における文字の印字表現は上記のように #\ に続く文字表記であるが，そのような表記を与えて直接文字を生成することもできる．

```
cl-user(19): (setq txt "This is a string.")
"This is a string."
cl-user(20): (setf (char txt 0) #\t)
#\t
cl-user(21): txt
"this is a string."
```

文字スペースの印字表現は #\Space であり，改行は #\Newline である．文字列は後述する列(sequence)の副型でもあるので，文字列の操作には通常シーケンスの関数が用いられる．

```
cl-user(22): (nsubstitute #\Newline #\Space txt)
"this
is
a
string."
cl-user(23): txt
"this
is
a
string."
```

文字には等しいとか大小比較の概念がある．これらの比較には文字コードが用いられるが，今日では個々の文字の文字コードはUNICODEと考えてよい．

```
cl-user(25): (char= #\A #\A)
t
cl-user(26): (char< #\A #\あ)
t
cl-user(27): (char< #\A #\a)
t
cl-user(28): (char-code #\A)
65
cl-user(29): (code-char #x0031)
#\1
cl-user(30): (code-char #x0041)
#\A
cl-user(31): (code-char #x0391)
```

```
#\A
cl-user(32): (char-name #\A)
"greek_capital_letter_alpha"
cl-user(33): (char-name #\あ)
"hiragana_letter_a"
```

ふだんはダブルクォーテーションを使って文字列を生成すればよいが，たまにはプログラム中で動的に文字列を生成することが必要になるかもしれない．そんなときのためにいくつかの手法をここで紹介する．

- 文字のリストを文字列にするには coerce を用いる．

```
(coerce (list #\a #\b #\c) 'string) => "abc"
```

- make-string は残念なことに一つの文字でしか初期化できない．

```
(make-string '3 :initial-element #\a) => "aaa"
```

- 無理やりこんな風にプログラムすることはできる．

```
(loop with txt = (make-string '3 :initial-element #\Null)
      with chc = (char-code #\a)
      for i from 0 to 2
      do (setf (char txt i) (code-char (+ chc i)))
      finally (return txt))
 => "abc"
```

- make-array を使えば初期化は簡単

```
(make-array '(3) :element-type 'character
                 :initial-contents (list #\a #\b #\c))
 => "abc"
```

リストのコピーには copy-list が用意されている．しかし copy-string という関数はない．かわりに copy-seq を使う．

```
cl-user(17): (setq text1 "これは文字列")
"これは文字列"
cl-user(18): (setq text2 (copy-seq text1))
"これは文字列"
cl-user(19): (string= text1 text2)
t
cl-user(20): (equal text1 text2)
t
cl-user(21): (eql text1 text2)
nil
cl-user(22): (loop for i from 0 to (1- (length text1))
                   always (eql (char text1 i) (char text2 i)))
t
```

しかし，こうやって作られる文字列の長さは固定であり，拡張することはできない．

```
cl-user(23): (length text1)
6
cl-user(24): (setf (char text1 6) #\で)
Error: Array index 6 not of type (integer 0 5) while accessing "これは文字列".
[condition type: type-error]
```

動的に拡張可能な文字列を作るにはフィルポインタと可変配列を利用して，このようにする．

```
cl-user(27): (loop with txt = (make-array 0
                                          :element-type 'character
                                          :fill-pointer 0
                                          :adjustable t)
                  with chc = (char-code #\a)
                for i from 0 to 2
                 do (vector-push-extend (code-char (+ chc i)) txt)
                 finally (return txt))
"abc"
```

エスケープ文字

REPLでは，文字列はダブルクォーテーションで括ってその場で直接に生成することができるが，文字列の中にダブルクォーテーションを文字として入れたい場合には，それと分かるように逆スラッシュ「\」でエスケープしなければならない．逆スラッシュはCommon Lispの文字列表現において，それに続く文字一つを文字として指定するためのエスケープ文字である．以下の実施例を参考にされたい．

```
cl-user(8): "\" is a double quotation."
"\" is a double quotation."
cl-user(9): (coerce * 'list)
(#\" #\Space #\i #\s #\Space #\a #\Space #\d #\o #\u ...)
cl-user(10): (describe #\")
#\" is an immediate character.
cl-user(11): (coerce "\\ is a backslash." 'list)
(#\\ #\Space #\i #\s #\Space #\a #\Space #\b #\a #\c ...)
cl-user(12): (describe #\\)
#\\ is an immediate character.
cl-user(13): (coerce "\a\b\c" 'list)
(#\a #\b #\c)
cl-user(14): (coerce * 'string)
"abc"
cl-user(15): (coerce (coerce "\a\b\c" 'list) 'string)
"abc"
```

12.6 列（シーケンス）

列（シーケンス）というのはリストとベクタを合わせた型概念である．実装上は全く異なる両者ではあるが，両方とも順番や長さ等の共通の属性を持ち，逆順にするとか何番目を取り出す等，共通する操作がある．文字列は特殊なベクタだからもちろんシーケンスでもある．だから，find や reverse や nreverse などの列用の関数は文字列にも使える．しかし，似たようで異なる関数がリスト用とシーケンス用にあって注意が必要である．mapcar や mapcan はリスト専用であるが，map はシーケンスにも使える．subst はリスト専用だが，substitute はシーケンス用としてリストにも文字列にも使える．だからリストでツリーの枝すべてに置換をかけるときは subst で，リストのトップレベルだけで置換するときは substitute になる．

前述のように，配列とベクタを作るには make-array を使えばよい．文字列には make-string が用意されている．しかし make-vector というのはなくて，それをするなら make-sequence になる．以下はすべて同じ

結果が得られる．

```
(make-array '(3) :element-type 'character :initial-element '#\a) => "aaa"

(make-sequence 'string 3 :initial-element '#\a) => "aaa"

(make-string 3 :initial-element '#\a) => "aaa"
```

もし配列のコンテンツを指定したいのなら，前述のようにベクタでも make-array を用いるしかない．

シーケンスのアクセスには elt が用意されている．しかし，配列(array) の要素にアクセスするには aref，文字列にアクセスするには char が用意されているので，これらに対しては通常は elt は使わない．残念ながら文字列アクセスに svref は使えない．文字列は simple-vector ではないのだ．

```
cl-user(30): (elt '(a b c) 2)
c
cl-user(31): (elt "abc" 2)
#\c
cl-user(32): (nth 2 '(a b c))
c
cl-user(33): (char "abc" 2)
#\c
cl-user(34): (aref "abc" 2)
#\c
cl-user(35): (svref "abc" 2)
Error: Illegal vector object passed to svref: "abc"
[condition type: simple-error]
```

12.7 ハッシュ表

以下は岩波情報科学事典からの引用である．

> 〝ハッシュ表とはハッシュ法において用いられるデータ探索，格納用の表をいう．ハッシュ法とはデータを登録するために探索鍵の内部コードを用いて格納アドレスを決定する方法をいう．探索鍵の内部コードからアドレスを求めることを鍵-アドレス変換という．そのための関数をハッシュ関数といい，得られるアドレスをハッシュアドレスという．データを格納するハッシュ表は初めは空であり，データを登録するにつれて詰まってくる．この詰まりの程度，すなわち表の大きさに対するデータの個数の割合を表占有率という．ハッシュ表の探索費用は，表へのアクセス回数すなわち平均探査回数で表すことが多い．平均探査回数は，原理的にはデータ総数ではなく表占有率に依存する．表占有率が小さいときはほぼO(1)である．異なるデータが同じアドレスに変換されて衝突が生じたときどのように処置するかによって，開アドレス法や連鎖法など種々のハッシュ法がある．〟（岩波情報科学事典より）

6.4節の map-reduce では，インデックスの転置行列を作るためにハッシュ表を利用した．

```
cl-user(17): (setq *shuffled-array* (make-hash-table))
#<eql hash-table with 0 entries @ #x20288ef82>
```

このように， make-hash-table は引数無しで使うことが多いが，キーワードパラメータを使って，**キーの同値関係**を調べる関数として eq, eql, equal, equalp のいずれかを指定することができる．既定値では eql

が用いられるので，シンボルならそれでよいが，もし大文字小文字を区別して文字列をキーとしたければ equal を，もし大文字小文字を区別せずに文字列をキーとしたければ， equalp を用いなければならない．

```
cl-user(7): (setq foo (make-hash-table :test #'equal))
#<equal hash-table with 0 entries @ #x1022e0172>
cl-user(8): (setf (gethash "one" foo) 1)
1
cl-user(9): (setf (gethash "two" foo) 2)
2
cl-user(10): (gethash "one" foo)
1
t
cl-user(11): (gethash "two" foo)
2
t
cl-user(12): (gethash "One" foo)
nil
nil
cl-user(13): (setq bar (make-hash-table :test #'equalp))
#<equalp hash-table with 0 entries @ #x1023126b2>
cl-user(14): (setf (gethash "one" bar) 1)
1
cl-user(15): (gethash "one" bar)
1
t
cl-user(16): (gethash "One" bar)
1
t
```

上記シンボル foo には equal が，シンボル bar には equalp がテスト関数として与えられたが．実施例12と16では挙動が異なることを理解されたい．

ハッシュ表がほぼ満杯になると，そのサイズが自動的に拡張される．キーワードに何も指定しなかった場合，最初にどのくらいのサイズのハッシュ表が定義され，どのくらいの大きさに拡張されるのかは処理系依存とされているが，ユーザが積極的にサイズと拡張の大きさを指定することができる．

```
cl-user(23): (setq small-hashtable
                   (make-hash-table :test #'equal
                                    :size 70
                                    :rehash-size 1.5))
#<equal hash-table with 0 entries @ #x102467642>
cl-user(24): (hash-table-size small-hashtable)
107
cl-user(25): (loop for x from 0 to 90
                 do (setf (gethash x small-hashtable) x))
nil
cl-user(26): (hash-table-size small-hashtable)
163
```

この例では，最初サイズ70の指定に対して，実際には107が取られ，100個のデータ保存の結果，サイズ拡張が起こって，サイズが163になった．

ハッシュ保存された全データを一気にクリアするには，clrhash を用いる．保存されたデータのエントリ数は hash-table-count で分かる．

```
cl-user(27): (hash-table-count small-hashtable)
91
cl-user(28): (clrhash small-hashtable)
#<equal hash-table with 0 entries @ #x1022afe32>
cl-user(29): (hash-table-count small-hashtable)
0
```

ハッシュ表のエントリの値を消すには nil 値をセットすればよいが，エントリそのものを消す場合には remhash を用いる．

```
cl-user(30): (gethash "one" small-hashtable)
nil
nil
cl-user(31): (setf (gethash "one" small-hashtable) 1)
1
cl-user(32): (gethash "one" small-hashtable)
1
t
cl-user(33): (setf (gethash "one" small-hashtable) nil)
nil
cl-user(34): (gethash "one" small-hashtable)
nil
t
cl-user(35): (hash-table-count small-hashtable)
1
cl-user(36): (remhash "one" small-hashtable)
t
cl-user(37): (hash-table-count small-hashtable)
0
cl-user(38): (gethash "one" small-hashtable)
nil
nil
```

gethash は2値を返すが，その二番目の値はエントリがあるときに真，そのエントリがないときには偽である．上記実施例34の返す `nil`，`t` はエントリはあって，その値は `nil` であることを表している．しかるに，最後の実施例38ではエントリそのものが削除されたことを表している．

第13章 Lisp の型システム

Lisp の変数は型を持たないが，言ってみればそれはデータへのポインタであり，ポイントされるデータそのものは型を持つ．初期の代表的な Lisp 処理系である Lisp 1.5においても，シンボルやコンス以外にも，整数，浮動小数点数，配列などのデータ型はあった．Common Lisp では更に文字，文字列，配列，ハッシュ，ビット，ストリーム，パスネーム，パッケージなどがあり，それらの抽象型や複合型も定義されて，全体に整合的な型階層を形成している．

13.1 データ型，型指定子

データ型とは，Lisp 中のオブジェクト，たとえば整数 0，1，2 とか，あるいは文字列 "This is a string.", "Lisp is a language." など，構造において共通の特徴を持つデータの集合のことである．これまでも関数，シンボル，数，配列，文字列，シーケンスなどについて説明してきたが，本章ではその他のデータ型およびデータ型に関する種々の関数について説明する．

型指定子(type specifier) とは，以下のように定義される．

- データオブジェクトの型を表すシンボルとして表記される素型指定子(atomic type specifier)
- 素型指定子，あるいはアトム satisfies，あるいは member を第1要素に含むリスト
- それらの and，or，not を用いた論理結合である．

Common Lisp 処理系に組み込みで定義されている素型指定子を以下に示す．

arithmetic-error	function	simple-condition
array	generic-function	simple-error
atom	hash-table	simple-string
base-char	integer	simple-type-error
base-string	keyword	simple-vector
bignum	list	simple-warning
bit	logical-pathname	single-float
bit-vector	long-float	standard-char
broadcast-stream	method	standard-class
built-in-class	method-combination	standard-generic-function
cell-error	nil	standard-method
character	null	standard-object
class	number	storage-condition
compiled-function	package	stream
complex	package-error	stream-error
concatenated-stream	parse-error	string
condition	pathname	string-stream
cons	print-not-readable	structure-class
control-error	program-error	structure-object

division-by-zero	random-state	style-warning
double-float	ratio	symbol
echo-stream	rational	synonym-stream
end-of-file	reader-error	t
error	readtable	two-way-stream
extended-char	real	type-error
file-error	restart	unbound-slot
file-stream	sequence	unbound-variable
fixnum	serious-condition	undefined-function
float	short-float	unsigned-byte
floating-point-inexact	signed-byte	vector
floating-point-invalid-operation	simple-array	warning
floating-point-overflow	simple-base-string	
floating-point-underflow	simple-bit-vector	

任意のオブジェクトの型を知るには type-of 関数を用いればよい.

```
cl-user(1): (type-of 3)
fixnum
cl-user(2): (type-of 100000000000000000000000000)
bignum
cl-user(3): (type-of 100.0)
single-float
cl-user(4): (type-of 100.0d0)
double-float
cl-user(5): (type-of "This is a string.")
(simple-array character (17))
cl-user(6): (type-of #\A)
character
```

あるオブジェクトについて，その型判定をするには typep を用いる.

```
cl-user(7): (typep 1 'fixnum)
t
cl-user(8): (typep 1 'integer)
t
cl-user(9): (typep 1 'number)
t
cl-user(10): (typep "one" 'number)
nil
cl-user(11): (typep "a string" 'string)
t
cl-user(12): (typep "a string" 'array)
t
cl-user(13): (typep #\A 'standard-char)
t
cl-user(14): (typep #\A 'character)
t
```

整数 1 の型は fixnum であるが，fixnum は integer の副型(subtype)であるので，整数 1 は型 integer にも所属する．同様に型 integer は型 number の副型であるので，整数 1 は型 number にも所属する．また，string は array の副型．standard-char は character の副型なので，上記の結果が得られる.

13.2 Lisp の型階層

仕様書に規定されていないデータ型については処理系によって実装依存の細部の型の違いはあるが，仕様とされている部分については，Common Lisp に共通な組み込みの素型指定子の型階層が定義されている．以下に示す型階層は CLOS システムとしての型階層を求めるルーチン(後述)を使って得られた結果について，ACL 固有の部分を削除し，上記に掲載の型指定子に照らして足らないところは追記して修正したものである．

```
(t (atom)
   (hash-table)
   (readtable)
   (function
    (compiled-function)
    (generic-function (standard-generic-function)))
   (symbol (null)
           (keyword))
   (character (base-char (standard-char (extended-char))))
   (array (vector (bit-vector (simple-bit-vector))
                  (string (base-string
                           (simple-string (simple-base-string))))
                  (simple-vector))
          (simple-array (simple-vector)
                        (simple-string)
                        (simple-bit-vector)))
   (sequence (vector)
             (list (null)
                   (cons)))
   (number
    (real (rational (ratio)
                    (integer (fixnum)
                             (bignum)
                             (signed-byte
                              (unsigned-byte (bit)))))
          (float (double-float)
                 (single-float)
                 (short-float)
                 (long-float))))
    (complex))
  (structure-object
   (package)
   (pathname (logical-pathname))
   (random-state)
   (logical-pathname)
   (restart))
  (standard-object
   (condition
    (serious-condition
     (storage-condition)
     (error
      (type-error (simple-type-error))
      (parse-error (reader-error))
      (stream-error (end-of-file))
      (package-error)
      (syscall-error (file-error))
      (cell-error
       (undefined-function)
```

```
      (unbound-variable)
      (unbound-slot))
     (print-not-readable)
     (arithmetic-error
      (floating-point-inexact)
      (floating-point-invalid-operation)
      (floating-point-underflow)
      (floating-point-overflow)
      (division-by-zero))
     (program-error)
     (control-error)
     (simple-error)))
   (warning
    (simple-warning)
    (style-warning))
   (simple-condition
    (simple-type-error)
    (simple-error)
    (simple-warning)))
  (stream
   (string-stream)
   (file-stream)
   (concatenated-stream)
   (echo-stream)
   (synonym-stream)
   (two-way-stream))
  (class
   (structure-class)
   (standard-class)
   (built-in-class))
  (method-combination)
  (method (standard-method))
  (generic-function)))
```

ここで，立体文字の型指定子は CLOS システムとして認識されるもので，CLOSのクラス定義があるが，イタリック体文字の型指定子は CLOS クラスはなく，Common Lisp の型としてのみ認識されるものである．たとえば，型 *atom* は (not cons) として型定義されるが，そういう CLOS クラスがあるわけではない．同様に fixnum はなぜかそういうクラス定義があるが，*bignum* は (and integer (not fixnum)) として型定義される．

型階層は束

vector は array の副型でもありかつ sequence の副型でもある．string は vector の副型なので array の副型かつ sequence の副型でもある．つまりこの型階層は数学的にはツリーではなくて束(lattice)である．型指定子 null は symbol の副型でもあり，list の副型でもあることに注意されたい．オブジェクト nil の型は null であるが，上記にあるように，null はシンボルの副型でもあるし，リストの副型でもある．すなわち nil はシンボルでもありリストでもある．

型指定子としての t はすべての型の最上位 (top) を表し，束の最下位 (bottom) には型指定子である nil がある．どんなオブジェクトも型 t に所属し，どんなオブジェクトも型 nil には所属しない．型 nil とは実は空集

合に相当する概念である．なお，null 型のインスタンスは nil しかない．

```
cl-user(37): (typep 't 't)
t
cl-user(38): (typep '123 't)
t
cl-user(39): (typep 'nil 't)
t
cl-user(40): (typep 't 'nil)
nil
cl-user(41): (typep '123 'nil)
nil
cl-user(42): (typep 'nil 'nil)
nil
cl-user(43): (typep 'nil 'null)
t
```

紛らわしいかも知れないが，シンボルとしての t および nil (の指示するもの)と，型としての t および nil をきちんと区別してほしい．上記37行目はシンボル t (の指示するもの)は型 t に所属するという意味であり，42行目はシンボル nil (の指示するもの)は型 nil には所属しないという意味である．

網羅的和集合と網羅的分割

ある型のすべてのオブジェクトがその副型のいずれかに必ず所属するとき，その型は副型の**網羅的和集合**(exhaustive union)と呼ぶ．有理数(rational)は整数(integer)と分数(ratio)の網羅的和集合である．

網羅的和集合でかつ副型が**互いに素**(disjoint)すなわち副型の間で共有されるメンバーはないとき，それを**網羅的分割**(exhaustive partition)と呼ぶ．most-positive-fixnum 以下かつ most-negative-fixnum 以上の整数は fixnum であるが，それ以外の整数は bignum に所属する．fixnum と bignum は互いに素であり共通のメンバーを持たない．また fixnum と bignum の合併（和集合）は integer に等しい．すなわち，fixnum と bignum は integer の網羅的分割である．同様に，ratio と integer は rational の網羅的分割であり，rational と float も real の網羅的分割である．

上下関係が計算できない？

二つの型指定子どうしの関係は論理的に次のものしかない．

1. どちらかがどちらかの(直接のあるいは間接の)副型である．すなわち，副型のすべてのインスタンスは必ずもう片方の型のインスタンスでもある．両方が両方の副型の場合はその二つの型は等価である．
2. ある型の一部のインスタンスのみが他の型のインスタンスでもあるが，すべてのインスタンスがそうではない．
3. 両者のインスタンスに共通のメンバーはない．

たとえば，網羅的分割にある副型はその上位の型と第 1 の関係にある．symbol と sequence は nil を共有するので，上記第2の関係にある．number と sequence は上記第 3 の関係にある．

関数 subtypep は二つの型指定子を引数に受け取って，両者の関係を調べる．

```
cl-user(23): (subtypep 'simple-bit-vector 'array)
t
t
cl-user(24): (subtypep 'string 'array)
t
t
cl-user(25): (subtypep 'fixnum 'bignum)
nil
t
```

関数 subtypep は 2 値を返す．二つ合わせて次のような意味である．

第1返値	第2返値	備考
t	t	type1 は確実に type2 の副型である
nil	t	type1 は確実に type2 の副型ではない
nil	nil	決められない

<t，nil> の組み合わせが返ることは決してない．それにしても「決められない」ことがあるとは驚きだが，論理上はっきり分かる場合でも，Common Lisp は計算できない場合があるようだ（処理系依存）．特に，(not 《型指定子》) という型指定をしたときにそれが著しい．Bakerによれば，後述の satisfies の場合も計算が難しいそうである．

```
cl-user(26): (subtypep 'fixnum '(not bignum))
nil
nil
cl-user(27): (subtypep 'fixnum '(not fixnum))
nil
nil
```

論理的には，上記26行目に対しては確実に真であるし，27行目に対しては確実に偽であるにもかかわらず，ACL ではそうなっていない．ACL に限らず，論理的に正しく型の包含関係を計算する満足な Common Lisp 処理系がないのは驚きである．この話題に興味がある方は拙書論文を参照されたい．

今の Common Lisp は型については意味論上の議論ではなく，プログラム言語実装上の議論という側面が強い．たとえば，1.0d0 と 1.0 は数学上では意味するところは同じであるが，型としては別物であり，オブジェクトの実装としても異なるものである．

```
cl-user(19): (typep 1.0d0 'double-float)
t
cl-user(20): (typep 1.0 'single-float)
t
cl-user(21): (equal 1.0d0 1.0)
nil
cl-user(22): (= 1.0d0 1.0)
t
```

このあたりをしっかりと掴むには REPL における入力の字句表記と，システム上の実装オブジェクトと，Common Lisp の意味論上でオブジェクトが意味するものを区別する（付3.1参照のこと）ことが重要であるが，Common Lisp の意味論上の定義がないのは問題である．

Type Upgrading

第12章の配列において，期待と異なって要素型を指定してもそれが制約となっていないことを述べた．そのわけをここで説明する．次の実施例を見られたい．

```
cg-user(1): (setq mixed-type-vector (make-array '(4)))
#(nil nil nil nil)
cg-user(2): (setf (aref mixed-type-vector 0) 3)
3
cg-user(3): (setf (aref mixed-type-vector 1) 3.14)
3.14
cg-user(4): (setf (aref mixed-type-vector 2) pi)
3.141592653589793d0
cg-user(5): (setf (aref mixed-type-vector 3) 'pi)
pi
cg-user(6): (loop for i from 0 to 3 collect (type-of (aref mixed-type-vector i)))
(fixnum single-float double-float symbol)
cg-user(7): (describe mixed-type-vector)
#(3 3.14 3.141592653589793d0 pi) is a new (simple-array t (4)).
 There are 4 elements
cg-user(8):
```

４要素数のベクターを作って，順番に整数，単長浮動小数点数，倍長浮動小数点数，シンボルのオブジェクトを保存した．そのベクターの型は，要素型は t の4要素のベクターである．

今度は同じことを要素型として fixnum を指定して行ってみる．

```
cg-user(8): (setq simple-type-vector (make-array '(4) :element-type 'fixnum))
#(0 0 0 0)
cg-user(9): (setf (aref simple-type-vector 0) 3)
3
cg-user(10): (setf (aref simple-type-vector 1) 3.14)
Error: Attempt to store the wrong type of a value in an array.
[condition type: simple-error]
cg-user(11): (setf (aref simple-type-vector 1) 4)
4
cg-user(12): (setf (aref simple-type-vector 2) "pi")
Error: Attempt to store the wrong type of a value in an array.
[condition type: simple-error]
cg-user(13): (setf (aref simple-type-vector 2) 5)
5
cg-user(14): (setf (aref simple-type-vector 3) 'pi)
Error: Attempt to store the wrong type of a value in an array.
[condition type: simple-error]
cg-user(15):
```

今度は当初の期待通り，fixnum 以外のデータは受け付けない．実はinteger型には， fixnum もあれば bignum もある．fixnum のデータが必要とするメモリー長は固定だが，bignum では必要なだけのメモリー

が用いられる．だから要素型が fixnum であれば，intrinsicな実装もできるが，integerでは要素の実装にはポインターセルを使うしかなく，すべての要素の型を包含するような型としているわけだ．仕様としては make-array 時に :element-type で指定した型が必ずしも実際の型判定時に用いられるわけではないということなのである．それは実装依存であり，指定の型よりも上位の型であってよいということらしい．それを ANSI Lisp では，Type Upgrading と呼んでいる．それでも型付きλ計算の理想としては型 t ではなく，型 integer としてほしいところではある．なお，配列の要素型について実際の配列の型判定で用いられる型は upgraded-array-element-type で求めることができる．

13.3 新しい型の定義

Common Lisp では deftype マクロを用いて，プログラマーが既存の型定義をベースにして，新しい派生型を作ることもできるし，まったく新しい型を新規に作り出すこともできる．

論理結合の型指定子

任意の型指定子の論理結合として新しい型を定義することができる．ここでの型の意味は外延的に解釈される．つまり引用される個々の型を集合名と考え，その集合の補集合，積集合，和集合である型を定義できる．

- (not《型》) 指定された型ではないすべてのオブジェクトの集合を表す．
- (and《型1》《型2》《型3》...) 指定された型の共通部分の集合を表す．
- (or 《型1》《型2》《型3》 ...) 指定された型の和集合を表す．

メンバー指定の型指定子

個別に型集合のメンバーを指定する型指定子がある．

- (member《オブジェクト1》《オブジェクト2》《オブジェクト3》 ...)

たとえば，光の三原色を型にするには `(member red blue green)` とすればよい．

第26章のルールベースシステムでは Mycin のパラメータの型として新しく yes/no 型を定義するが，その書式は次のようなものである．

```
(deftype yes/no () '(member yes no))
```

数の範囲指定の型指定子

Common Lisp に正の整数の型や自然数の型はないが，必要とあらば自分で定義することができる．

- (《numberの副型》《下限値》《上限値》)

たとえば，自然数は (integer 0 *) と定義され，負の整数型は (integer * -1) と定義される．ここでアスタリスク「*」は無指定を表す．(integer 0 *) は (integer 0) と書くことができ，integer とは (integer * *) のことである．第26章で説明されるが MYCIN の確信度の型は (float -1.0 1.0) である．

unsigned-byte および signed-byte 型の場合にはその bit 数を指定して特殊化することができる．

- (unsigned-byte 《ビット数》)
- (signed-byte 《ビット数》)

型指定子 bit 型とは (unsigned-byte 1) のことである．

配列要素の型と長さを特殊化した型指定子

Common Lisp の配列はその要素に任意の型のデータを許し，しかもその型が混在してもよい．しかし，3次元空間の点とか，直方体のサイズなど，配列の要素の型と長さを指定したい場合もある．長さ3の倍精度浮動小数点を要素とする型は (vector double-float 3) と定義される．整数のみを含む3次元配列は (array integer (* * *)) と定義される．

- (array《要素型》《ディメンジョン》)

新規の型指定子

これまでの型指定子は member による型指定子を除き，必ず素型指定子が含まれていた．既存の型指定子を用いることなく，新規にデータ型を定義することもできる．それにはまずそのための型判定述語を準備して，次のような型指定子を作ればよい．

- (satisfies《型判定述語》)

たとえば，(satisfies numberp) 型というのは number 型に等しい．

```
cl-user(34): (numberp 1234)
t
cl-user(35): (typep 1234 'number)
t
cl-user(36): (typep 1234 '(satisfies numberp))
t
```

型判定述語があるのに，なぜわざわざその型を定義しなければならないのかと思うかもしれないが，typecase を用いる場合に必要なものは述語ではなく，型指定子である．だから述語を満足する型が欲しくなる場合があるのだ．

deftype

新しい型指定子を考えても，その定義が丸見えのリストのままでは見栄えが悪い．型指定子に名前を付けてプリミティブな型と同じように使うことができる．そのために用いられるのが deftype である．その一般

的な形式は以下のとおり．

```
(deftype 《名前》 《ラムダリスト》
  [{《宣言》}* | 《ドキュメンテーション》]
  {《書式》}* )
```

しかし，deftypeの書き方は少々複雑である．以下に個別に説明する．

- ラムダリストの引数なしの場合には，書式には型指定子をそのままクォートをつけて書く．

XML Schema データ型を定義する

XML Schemaで定義されるデータ型をCommon Lispのデータ型と比較すると，xsd:stringやxsd:integerなど，そのまま名前だけ変えればよいものもあれば，xsd:byte，xsd:short，xsd:int，xsd:longのように範囲制限を付けるものもある．deftypeを用いてCommon Lisp環境にXML Schemaデータタイプを用意できる．

以下を参照されたい．

```
(deftype xsd:unsignedByte () "0 <= x <= 255"
  '(unsigned-byte 8))
(deftype xsd:unsignedShort () "0 <= x <= 65535"
  '(unsigned-byte 16))
(deftype xsd:unsignedInt () "0 <= x <= 4294967295"
  '(unsigned-byte 32))
(deftype xsd:unsignedLong () "0 <= x <= 18446744073709551615"
  '(unsigned-byte 64))

(deftype xsd:byte () "-128 <= x <= 127"
  '(signed-byte 8))
(deftype xsd:short () "-32768 <= x <= 32767"
  '(signed-byte 16))
(deftype xsd:int () "-2147483648 <= x <= 2147483647"
  '(signed-byte 32))
(deftype xsd:long ()
  "-9223372036854775808 <= x <= 9223372036854775807"
  '(signed-byte 64))
(deftype xsd:integer ()
  "any number of digits without point but with or without + or -"
  'integer)

(deftype xsd:positiveInteger () "integer greater than 0"
  '(integer 1 *))
(deftype xsd:nonPositiveInteger () "integer smaller than 1"
  '(integer * 0))
(deftype xsd:negativeInteger () "integer smaller than 0"
  '(integer * -1))
(deftype xsd:nonNegativeInteger () "integer greater than -1"
  '(integer 0 *))

(deftype xsd:float () "floating point number in lisp"
  'single-float)
(deftype xsd:double () "double floating point number in lisp"
  'double-float)
```

ただし，ここでは先に xsd パッケージが定義されていて，そのパッケージ中で各種シンボルが移出されているものとしている．詳細は第20章を参照されたい．

素型指定子のように振る舞う新しい型を定義したいときには，deftype のパラメータリストに必須パラメータがあってはならない．satisfies を使えば，そのような任意の型を定義できる．dotted-pair という新しい型を定義してみよう．

```
cl-user(1): (defun dotted-pair-p (x)
              (and (consp x) (not (null (cdr x)))))
dotted-pair-p
cl-user(2): (deftype dotted-pair ()
              '(satisfies dotted-pair-p))
dotted-pair
cl-user(3): (typep '(a . b) 'dotted-pair)
t
```

引数を受け付ける型指定子を定義することもできる．ただし，その引数というのは型指定の定義で用いられるものであり，型判定が施されるオブジェクトを引数とすることはできない．*CLtL2*で説明されている例は，各次元のサイズが等しい配列の例であり，配列のオブジェクトを型定義で必要としないため，こういうものでは比較的簡単に引数を持つ型指定子を定義できる．

```
cl-user(2): (setq foo (make-array '(3 3 3)))
#3A(((nil nil nil) (nil nil nil) (nil nil nil))
    ((nil nil nil) (nil nil nil) (nil nil nil))
    ((nil nil nil) (nil nil nil) (nil nil nil))
cl-user(3): (array-dimensions foo)
(3 3 3)
cl-user(4): (defun equidimensional (a)
              (or (< (array-rank a) 2)
                  (apply #'= (array-dimensions a))))
equidimensional
cl-user(5): (equidimensional foo)
t
cl-user(6): (deftype square-matrix (&optional type size)
              `(and (array ,type (,size ,size))
                    (satisfies equidimensional)))
square-matrix
cl-user(7): (setq bar (make-array '(9 9)))
#2A((nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil)
    (nil nil nil nil nil nil nil nil nil))
cl-user(8): (typep bar 'square-matrix)
t
cl-user(9): (typep foo 'square-matrix)
nil
```

ここで，deftype の定義では必須パラメータではなくオプショナルとしていることに注意されたい．それは square-matrix を素型指定子のように使えるようにするためである．

第14章 構造体

セルとポインターからなるリストはどんなデータ構造も表現できる非常に柔軟な装置であるが，それ故にプログラミングのパラダイムとして素朴すぎるという場合もある．第12章では配列を学んだが，Common Lisp の配列は要素に異なるデータ型を混在できるという柔軟性はあっても，それへのアクセス手段は添え字を用いるしかない．Common Lisp の構造体の機能を使えば，構成要素へのアクセス手段が提供されるばかりでなく，ユーザは Common Lisp では提供されていない任意のデータ型を新しく作ることができる．しかもその新しい型は Common Lisp のデータ型として組み込みの型と何の区別もなく扱うことができる．

本章では最初にXMLデータ型の dateTime を Common Lisp の構造体として実装する例題を通して，構造体の初歩的な説明をする．その後，数多くあるdefstructオプションのうち:listと:includeについて，それぞれ例題を通して説明する．

14.1 Common Lisp の Universal Time

Common Lisp では時刻は "Universal Time" として実現される．これは最小単位を秒とする非負の整数である．これとは別に "Internal Time" というものがあるが，これは秒以下の最小単位（定数 internal-time-units-per-second を見られたい）の非負の整数であり，通常はコンピュータの実行時間の計測などに用いられるものであって，日常生活と密接に結びついている時刻は "Universal Time" の方である．

現在時刻の "Universal Time" を得るには，関数 get-universal-time を用いればよく，get-decoded-time はそれを秒，分，時間，日付，月，年，週日，夏時間フラグ，タイムゾーン，に分けたデータにして返す．

```
cl-user(2): (setq foo (get-universal-time))
3643230645
cl-user(3): (get-decoded-time)
51
30
9
14
6
2015
6
nil
-9
cl-user(4): (decode-universal-time foo -9)
45
30
9
14
6
2015
6
nil
-9
```

上記例は日本で2015年6月14日の日曜日に実行されたが，とすると日本のタイムゾーンは−9(協定世界時UTCより9時間早い，ただし通常我々がよく見る表記はUTC＋09:00であり何故か符号が逆である)である．2015年6月14日の日曜日が週日6ということは，月曜日を起点0とする0から6までの数字ということらしい．

グリニッチ標準時GMTと協定世界時UTC

Common Lisp でカレンダー時刻形式と言ったとき，そこで扱われる時刻や暦は何を使っているのだろうか？*CLtL2*ではタイムゾーンに関しては明確にグリニッチ標準時GMTと書いてある．ANSI Common Lispにおいてもそれは同様である．世界的にかっては地球の自転周期に基づいた平均太陽暦が共通の時間基準として用いられ，子午線０度であるグリニッチ天文台の位置を基準としてその時間からの差異を定めた時間が各国で標準として使われていた．

旧協定世界時とは，1964年から正式に採用されて1971年末まで使われた標準電波の報時信号に同期する，国際協定に基づいて使われた時刻のことである． 一方，原子時計による時刻として原子時(TA)なるものが一部に使われていたが，1967年にセシウム原子時計を基準に１秒が定められ，それを基準に各国で調整されてきた時間が国際原子時 (TAI) である．そして1972年には国際原子時 (TAI) と協定世界時 (UTC) との調整が行われた．このときの調整により協定世界時 (UTC) は国際原子時 (TAI) に対して丁度10秒遅れとなった．あらためて整理すると，協定世界時 (UTC) は、国際原子時 (TAI) に由来する原子時系の時刻だが，世界時の UT1（したがって地球の自転）に歩調を合わせるように調整された基準時刻のことである．UTCは原子時計由来で正確であるがこれを天文系のUT1と合わせるために現在ではときどきうるう秒がUTCに挿入される．本原稿執筆時では2015年7月1日0時にうるう秒が挿入された．ただし，相変わらずUTCとTAIの違いは存在し，1972年調整時には10秒だったものが，現在では37秒となっている（この秒差はうるう秒が挿入されるたびに1ずつ増える）．

さて，今日ではグリニッチ標準時 (GMT) というものは協定世界時 (UTC) にとって変わられたと考えてよいらしいが，Common Lisp でもそう考えてよいのだろうか．decorded-time はuniversal-time を decode-universal-time すれば得られるものだから，問題は universal-time が何かということになる．そこは処理系依存になるが，多分今ではマシンの時刻をとってきているであろうし，マシンの時刻というのはNTPサービスに依存してセットされるであろう．だからマシンの内臓時計をNTPに合わせた場合には，unversal-time はUTCあるいはGMTより9時間早い日本標準時 (JST) と考えられる．実際下図に示すようなNICTの提供する日本標準時を見ながら (get-decoded-time) を実行すると，秒単位まで等しい結果が得られる．



しかし，分からないのはここからだ．Common Lisp では universal-time は「うるう秒を無視して」，1900年1月1日0時 (GMT) を起点に計算することになっている．とすると，2015年6月30日23時59分59秒 (UTC) から物理的に10秒後に get-universal-time を実行すると，それは2015年7月1日0時0分9秒 (UTC)が得られるのであろうか．うるう秒が挿入される時刻をまたがって，数秒前から起動されて internal-time で数十秒後に get-decoded-time を実行するようなプログラムを書いて実行すると面白そうだ．

問題は逆に時間をさかのぼってもある．(decode-universal-time 0 0) を実行すると，次のような結果にはなるが，それは定義上だけの話であって，実際に今 (get-universal-time) で得られる秒だけ時間軸をさかの

ぼることができたとしても，そのときが天文的な，すなわちGMTとしての1900年1月1日の0時0分0秒になるわけではない．

```
cl-user(1): (decode-universal-time 0 0)
0
0
0
1
1
1900
0
nil
0
```

もし，今日 Common Lisp の仕様書を書くとすれば，get-uiversal-time で得られるものは当然 GMT ではなく UTC と書くことになるであろう．

問題 14.1 各自の所有するマシンの内臓時計をNICTの提供する時間に合わせて，上図に示すウェブ画面を見ながら，get-decoded-time を実行して，両者が一致することを確認せよ．

14.2 date 構造体をオプションなしで定義する

構造体を定義するには defstruct マクロを用いるが，これには様々なオプションがあってややこしい．しかしオプションなしで使う分には，ユーザは新しい型のデータを楽に便利に定義することができる．たとえば，Common Lisp にはXML Data Schema にあるような date のデータ型がない．そこで Common Lisp でも使えるような date 型を作ってみる．

```
cl-user(3): (defstruct date year month day timezone)
date
```

このようにしただけで，date の実現体を作るための make-date，データ型をチェックする date-p，各々のスロット値にアクセスするためのアクセッサ date-year，date-month，date-day，date-timezone が自動的に定義され，使うことができる．

```
cl-user(6): (multiple-value-bind
                  (second minute hour day month year day-of-week daylight-p zone)
                (get-decoded-time)
              (assert (eq daylight-p nil))
              (setq foo
                    (make-date :year year :month month :day day :timezone zone)))
#S(date :year 2015 :month 6 :day 25 :timezone -9)
cl-user(7): (date-p foo)
t
cl-user(8): (date-year foo)
2015
cl-user(9): (date-timezone foo)
-9
```

このようにしてできた構造体定義の名前は Common Lisp の型システムにおいて型名として扱うことができるし，実現体はその型に所属するものとなって，組み込みのデータと同様に type-of や typep で有効とな

る.

```
cl-user(10): (type-of foo)
date
cl-user(11): (typep foo 'date)
t
```

実現体のスロットに値をセットするには，アクセッサを用いて setf すればよいが，スロット値の既定値を定義したいときには，defstruct 時に以下のようにすればよい.

```
cl-user(13): (setf (date-timezone foo) 9)
9
cl-user(14): (defstruct date
               (year 0) (month 0) (day 0) (timezone 0))
date
cl-user(15): (setq bar (make-date))
#S(date :year 0 :month 0 :day 0 :timezone 0)
```

14.3 defstructオプション

オプションなしで構造体を使うには，上記のように簡単に使えるが，オプションを使ってコンストラクタの名前を変更したり，型判定述語の名前を変更したりすることができる．詳細は*CLtL2*を参照されたい．ここでは*CLtL2*の説明だけでは何に使うかよく分からないタイプオプションについて，形態素解析器chasenの辞書であったIPADICを例に取り上げて説明する.

IPADICの辞書ファイルのエントリは，次に示すようにリスプのＳ式みたいな括弧で構成されている．以下は一般名詞「知能」に関する記述である.

```
(品詞 (名詞 一般)) ((見出し語 (知能 3486)) (読み チノウ) (発音 チノー) )
```

ここで，「品詞」は品詞情報であることを示すラベル，その次の「(名詞 一般)」が品詞名であり，以下の「(見出し語 (知能 3486))」「(読み チノウ)」「(発音 チノー)」がそれぞれ見出し語情報，読み情報，および発音情報である．こんなファイルを読み込むには，まずエントリの１行を取り込んでから，read関数を使って括弧の構造そのまま，「(品詞 (名詞 一般))」と「((見出し語 (知能 3486)) (読み チノウ) (発音 チノー))」に分けて取り込めばよい．以下のコードはそれをする.

```
(defun read-ipadic-file (instream)
  (loop for line = (read-line instream nil nil)
      while line
      do (with-input-from-string (s line)
           (let* ((hinshi (read s nil nil))
                  (others (read s nil nil)))
             ...
             ))))
```

これで，hinshiには(品詞 (名詞 一般))が束縛され，othersには((見出し語 (知能 3486)) (読み チノウ) (発音 チノー))が束縛される．othersの最初は必ず見出し語情報であり，今この場合は(見出し語 (知

能 3486))であるが，その次には任意個の形態素情報が並ぶ．それを分解するのにはdestructuring-bindを使うと便利である．詳しくは17.11節を参照されたい．

```
(destructuring-bind (midashigo &rest morphs) others ... )
```

これで，midashigoには(見出し語 (知能 3486))が束縛される．これから「知能」を取り出すには(first (second midashigo))とすればよいし，3486を取り出すには(second (second midashigo))とすればよいが，そんなコードがならぶとコード断片を見たときにそれが一体何をしているのか判断するのに時間がかかる．そういうときには専用のリーダを定義する．それはこんな具合になる．

```
(defun 品詞見出し語 (midashigo)
  (first (second midashigo)))

(defun 品詞見出し生起コスト (midashigo)
  (second (second midashigo)))
```

構造体のdefstructオプションにある:listを用いると，同様のことをさらに徹底して実行できる．以下に示すコードはリストを構造体として扱う2種類の方法だ．

```
(defstruct (見出し語 (:type list) :named) word-cost)

(defstruct (品詞見出し (:type list)) word cost)
```

いずれも構造体を生成するとそれはリスト構造となるが，:namedつきのほうは名前付きになるが，そうでないなら名前なしである．名前付きというのは以下に示すように，構造体の名前がリストの最初に置かれるものだ．

```
ipadic(6): (setq foo (make-見出し語 :word-cost '(脚気衝心 3999)))
(見出し語 (脚気衝心 3999))
ipadic(7): (setq bar (make-品詞見出し :word '脚気衝心 :cost 3999))
(脚気衝心 3999)
```

しかし，いずれも構造体としてのアクセッサが定義されるので，リスト処理としてではなく，構造体への処理としてアクセッサを次のように利用することができる．

```
ipadic(8) (見出し語-word-cost foo)
(脚気衝心 3999)
ipadic(9) (品詞見出し-word bar)
脚気衝心
ipadic(10) (品詞見出し-cost bar)
3999
```

だから，もともとのリスト構造に合わせて，リストの構造体を定義すると，自動的に定義される構造体のアクセッサを利用してリスト構造を処理することができるようになる．先の二つのリーダは次のようにしてよい．

```
(defun 品詞見出し語 (midashigo)
  (品詞見出し-word (見出し語-word-cost midashigo)))
```

```
(defun 品詞見出し生起コスト (midashigo)
  (品詞見出し-cost (見出し語-word-cost midashigo)))
```

問題 14.2 IPADICの辞書ファイル(dicタイプファイル)を読み込むコードは次のようなものである．このコードが正しく動くように，名前付きリスト構造体「品詞」と品詞名を取り出す関数「品詞名」を定義して，IPAdic legacyよりIPADICをダウンロードし，下記コードを実行せよ．SBCLではEUCファイルを読み込むことができないので、UTF-8のIPADIC辞書ファイル(dicタイプファイル)を用意して external-format 指定なしで使ってください。

```
ipadic(2) (with-open-file (ipadic-stream "Noun.dic" :external-format :EUC)
            (read-ipadic ipadic-stream *standard-output*))

仕舞い（名詞 一般）3999（読み シマイ）（発音 シマイ）
綺（名詞 一般）3999（読み アヤギヌ）（発音 アヤギヌ）
洋裁（名詞 一般）3649（読み ヨウサイ）（発音 ヨーサイ）
  ...
```

ただし，read-ipadic は次のような関数である．

```
(defun read-ipadic (instream outstream)
  (loop for line = (read-line instream nil nil)
      while line
      do (with-input-from-string (s line)
           (let* ((hinshi (read s nil nil))
                  (others (read s nil nil)))
             (destructuring-bind (midashigo &rest morphs) others
               (format outstream "~%~S ~S ~S ~{~S ~}"
                 (品詞見出し語 midashigo) (品詞名 hinshi) (品詞見出し生起コスト midashigo)
morphs))))))
```

14.4 includeオプションによる型階層定義

構造体として人型を次のように定義する．

```
cl-user(2): (defstruct person
              (sex nil)
              (age nil))
person
```

すると，関数make-personで人型のインスタンスを作ることができる．

```
cl-user(3): (setq foo (make-person :sex 'male :age 33))
#S(person :sex male :age 33)
cl-user(4): (person-p foo)
t
```

defstructオプションの一つである:includeを用いると，人型の副型としてビジネスパーソンを定義することができる．人の属性である性別と年齢に加えて，ビジネスパーソンの属性である電話番号とタイトルと年収を追加する．

```
cl-user(5): (defstruct (business-person (:include person))
              (tel nil)
```

```
              (title nil)
              (annual-income nil))
business-person
```

すると，新しくビジネスパーソンのインスタンスを定義することができるようになる．

```
cl-user(6): (setq bar
                  (make-business-person
                    :sex 'female :age 43
                    :tel '+81-3-123-4567
                    :title 'executive-officer
                    :annual-income 1000000))
#S(business-person :sex female :age 43 :tel +81-3-123-4567 :title executive-officer :annual-
income ...)
```

このビジネスパーソンはもちろんビジネスパーソン型であるが，同時に人型でもある．

```
cl-user(7): (person-p bar)
t
cl-user(8): (business-person-p bar)
t
```

そしてCommon Lispの型階層において，business-person型はperson型の副型になる．

```
cl-user(9): (subtypep 'business-person 'person)
t
t
```

ビジネスパーソンの属性値と人型の属性値がビジネスパーソンにはあるが，それぞれの属性値を同一のインスタンスから次のようにして，取り出したり，セットしたりすることができる．

```
cl-user(10): (business-person-title bar)
executive-officer
cl-user(11): (person-sex bar)
female
cl-user(12): (setf (business-person-title bar) 'CEO)
CEO
cl-user(13): (business-person-title bar)
CEO
cl-user(14): bar
#S(business-person :sex female :age 43 :tel +81-3-123-4567 :title CEO :annual-income ...)
```

第15章 CLOS とメタオブジェクト・プロトコル

> 多重継承は悪だ．みんなそう言っているから．オペレータのオーバロードは悪だ．みんなそう言っているから．私はその理由をぼんやりと理解してはいたが，本当にではなかった．それから私は最低なのは多重継承ではないと認識するようになった．最低だったのは私だ．
>
> [Steve Yegge, 2004]（バベル案内，青木(訳)）

Common Lisp Object System (CLOS) は *CLtL2* から導入されたため，あまりよい資料や教科書がない．PAIP でも使われていないし，AIMA 用の Lisp コードでは本来 CLOS を使うべきところで CLOS 風の独自オブジェクトシステムを開発して使っているありさまである．きっと AIMA 原稿執筆時には CLOS が出ていなかったのであろう．初心者向きに CLOS のよい教科書と言われているものに Sonya Keene の *Object-Oriented Programming in Common Lisp* があるとされているが，全くお薦めできない．一方，メタオブジェクト・プロトコル (MetaObject Protocol，略して MOP) ではCLOS実装の立役者である Kiczales 自身になる *The Art of the Metaobject Protocol* がある．MOPは ANSI Common Lisp の仕様には入っていないが，今日ほとんどの Common Lisp に実装されている．本章では現在でも最も先進的なオブジェクトシステムであるCLOSについて，初心者向けに解説する．最後にメタオブジェクト・プロトコルについて簡単に紹介する．

15.1 オブジェクト指向とは何か？

かつて Wikipedia では，オブジェクト指向プログラミングの特徴として以下のように記述していた．

1. カプセル化（振る舞いの隠蔽とデータ隠蔽）
2. インヘリタンス（継承） -- クラスベースの言語
3. ポリモフィズム（多態性、多相性） -- 型付きの言語
4. ダイナミックバインディング（動的束縛） -- 動的型付言語

しかも，「これらの機能のうち、オブジェクト指向の考え方で不可欠なのは『カプセル化』の機能だけである」とさえ述べていた．もしそうならば，CLOS には継承と多相性はあっても変数隠ぺいの機能はないから，CLOSはオブジェクト指向言語ではないことになってしまう．Java で頭が固まった人にとってはCLOSをオブジェクト指向言語と言ったら，目をむくかもしれないが，確かにCLOSはオブジェクト指向言語の流れを引き継いで生まれたものだ．C++やJavaが出現する前にCLOSがあって，CLOSはSmalltalk-80の影響を強く受けて生まれた（メタクラスのアイデアは多分Smalltalk-80から来ているのであろう．CLOSの参照モデルとなったPCL(Portable Common Loops)もSmalltalkも，Xerox PARC でほぼ同時期に開発された．何しろKiczalesはPARCにいたのだ）．しかも，CLOS はほんの少しだけ遅れて開発されたため，Smalltalk-80よりもずっとメタクラス構造などはよいものになっている．

Smalltalk ではメソッドディスパッチを決定するものは，メッセージが送られるオブジェクトの型（クラス）一つのみである．しかしCLOSではすべての関数引数の型（クラス）がメソッドの決定に関与しうる．また，Javaでは多重継承は禁じられているが，CLOSは多重継承を許している．序章で述べたように，Lisp

はプログラマーに最大限の自由を許しているが，それはCLOSにおいても同様であり，CLOS の標準仕様さえメタオブジェクト・プロトコル (MOP) を用いて自分の好きなように変更できるようになっている．

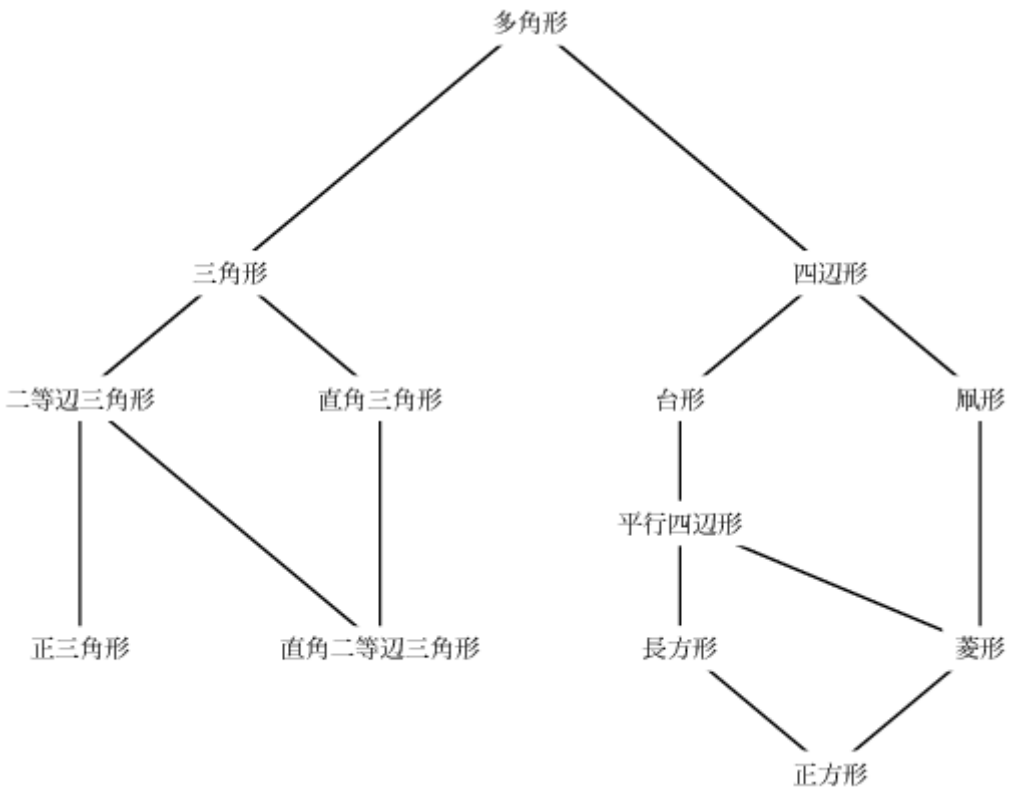
15.2 総称関数とメソッド

総称関数(generic function)とは名前と関数引数の数(arity)のみを共通とした関数群を指すものである．つまり，CLOSでは同一の名前を持つメソッドが複数あって，似ているが厳密には異なるデータに対する処理をまとめることができる．CLOSメソッドではないがこれまで出てきた例ではそのようなものとして，列（シーケンス）型のデータに対するアクセッサ elt がある．これは列型（シーケンス）に対する関数であり，列型の副型であるリストでもベクタでも同様に働く．総称関数とはそのように，様々な副型について共通する関数を抽象化した概念なのである．

```
cl-user(1): (elt '(zero one two three) 2)
two
cl-user(2): (elt #(zero one two three) 2)
two
```

もちろん，実際の関数の動作は，異なる型（クラス）についてはそれぞれ具体的に定義する必要があるが，ここでは関数名と呼び出し形式が共通していることに注目されたい．

これから説明するときの例題として，図形を取り上げる．厳密に言えば2次元平面に描かれる単純かつ閉じた図形である．三角形や四角形は多角形の特殊なもの（副型，下位クラス）と考えることができる．下図はエーベルソンとサスマンの著名なプログラミングの教科書SICPに記載された多角形の階層図である．



多角形のクラス階層図（計算機 プログラム の 構造 と 解釈 第二版, 図.2.26より）

このような多角形や楕円といった閉じた平面図形のどんなものにも，周長と面積があるので，どのように計算するかという細かいことは置いておいて，平面図形の周長と面積を求める総称関数を定義してよい．

```
cl-user(2): (defgeneric circumference (plain-figure))
#<standard-generic-function circumference>
cl-user(3): (defgeneric area (plain-figure))
#<standard-generic-function area>
```

総称関数の定義ではdefun ではなく，defgeneric と書くが，関数本体はない．

```
(defgeneric 《関数名》 《gfラムダリスト》
  [{《オプション》|《ドキュメンテーション》}* ]
  )
```

上記実施例で，二つの総称関数の名前と引数の数を規定したことになる．あるいはこの場合にはメソッド circumference と area のインターフェース仕様を定義したと考えてもらってもよい．

実際の計算手続きは defmethod で定義するが，そのときには各々のメソッドがどのクラスに対するものかを指定しなければならず，メソッド定義時には必要なクラスが定義済みでなければならない．たとえば，円のクラス定義を次のように行う．円には中心点座標と半径があるべきだと考え，そのように circle を定義した．

```
cl-user(4): (defclass plain-figure () ())
#<standard-class plain-figure>
cl-user(5): (defclass circle (plain-figure)
              (center-point radius))
#<standard-class circle>
```

ここでは plain-figure については上位クラス（スーパークラス）とスロットを定義せず，circle について上位クラスを plain-figure とし，center-point と radius を circle **実現体が持つ**べきスロットとして定義した．CLOS ではオブジェクトが持つデータ（javaでいうフィールド）をスロットと呼んでいる．スロットの定義はクラスに書くが，実際のスロットデータはそのクラスの実現体にあることを注意しておく．defclass の呼び出し形式は次のとおりである．

```
(defclass 《クラス名》 ({《直接上位クラス名》}*)
   ( {《スロット特定子》}* )
   [ 《クラスオプション》 ])
```

次に円の周長と面積を求めるメソッド circumference と area を次のように定義する．

```
cl-user(6): (defmethod circumference ((a-circle circle))
              (* 2 pi (slot-value a-circle 'radius)))
#<standard-method circumference (circle)>
cl-user(7): (defmethod area ((a-circle circle))
              (let ((r (slot-value a-circle 'radius)))
                (* pi r r)))
#<standard-method area (circle)>
```

ここで a-circle はメソッドの引数であり，circle は第 1 引数のクラス（型）を指定するクラス名である．CLOS では (slot-value 《オブジェクト》《スロット名》) でスロットの値を取り出すことができる．defmethod の定義形式は以下のとおり．

```
(defmethod 《関数名》{《メソッド修飾子》}* 《特殊化ラムダリスト》
  [{《宣言》}*|《ドキュメンテーション》]
  {《書式》}* )
```

特殊化ラムダリストには，**引数特定子**(parameter-specializer)，すなわち引数名のアトム，あるいは引数名とクラス名の対リスト，が並ぶ．アトムの場合にはその引数についてクラスによる制限はなく，通常の関数と同じことになるが，引数名とクラス名の対リストの場合には，実行時に渡ってきた実引数が指定のクラスに合致したクラスの実現体であるメソッドが選ばれて適応される．メソッド修飾子については15.4節にて説明する．

実際にあるクラスの実現体(instance)を作るにはクラスを指定してmake-instance を用いる．

```
cl-user(9): (setq my-circle (make-instance 'circle))
#<circle @ #x102439d52>
cl-user(10): (setf (slot-value my-circle 'radius) 3)
3
```

make-instance の呼び出し形式は以下のようであるが，上記実施例では初期化引数リストはなく，(setf slot-value) を使って半径 radius の値をセットしている．

```
(make-instance 《クラス》 &rest 《初期化引数リスト》 &key &allow-other-keys)
```

実際にこの circle 実現体の周長と面積を求めてみよう．

```
cl-user(11): (circumference my-circle)
18.84955592153876d0
cl-user(12): (area my-circle)
28.274333882308138d0
```

楕円の形状は楕円の中心点からの長軸半径と短軸半径が決まれば決まる．長軸直径と短軸直径の交点が楕円の中心点である．長軸半径 *a* と短軸半径 *b* の比を楕円率と呼ぶ．楕円率を $c = b/a$ とすると，長軸半径 *a* を主にして短軸半径 *b* は $b = ca$ で求めるものとする．楕円の面積が nab と簡単に求めることができるのに対して，楕円の周長は厳密には積分式となる．しかしその簡易近似計算式があるので，今回はそれを用いることとする．circle の場合と同様にして，ellipse とそのメソッドを定義する．

```
cl-user(13): (defclass ellipse (plain-figure)
               (center-point long-radius ellipcity))
#<standard-class ellipse>
cl-user(14): (defmethod short-radius ((an-ellipse ellipse))
               (* (slot-value an-ellipse 'long-radius)
                  (slot-value an-ellipse 'ellipcity)))
#<standard-method short-radius (ellipse)>
cl-user(15): (defmethod area ((an-ellipse ellipse))
               (* pi (slot-value an-ellipse 'long-radius)
                     (short-radius an-ellipse)))
```

```
#<standard-method area (ellipse)>
cl-user(16): (defmethod circumference ((an-ellipse ellipse))
               (let ((a (slot-value an-ellipse 'long-radius))
                     (b (short-radius an-ellipse)))
                 (let ((abab (/ (- a b) (+ a b))))
                   (let ((3abab (* 3 abab abab)))
                     (* pi (+ a b)
                        (+ 1
                           (/ 3abab
                              (+ 10
                                 (sqrt (- 4 3abab))))))))))
#<standard-method circumference (ellipse)>
```

計算式の確認のため，試しに楕円率 1 の楕円（すなわち実態は円）を定義して，計算してみる．

```
cl-user(17): (setq my-ellipse (make-instance 'ellipse))
#<ellipse @ #x1022c2ee2>
cl-user(18): (setf (slot-value my-ellipse 'long-radius) 3)
3
cl-user(19): (setf (slot-value my-ellipse 'ellipcity) 1)
1
cl-user(20): (short-radius my-ellipse)
3
cl-user(21): (area my-ellipse)
28.274333882308138d0
cl-user(22): (circumference my-ellipse)
18.84955592153876d0
```

小数点以下すべて先の半径 3 の円の計算値と同じ結果が得られた．

楕円率を 0.5 に置き換えて計算してみる．

```
cl-user(23): (setf (slot-value my-ellipse 'ellipcity) 0.5)
0.5
cl-user(24): (area my-ellipse)
14.137166941154069d0
cl-user(25): (circumference my-ellipse)
14.532672204015787d0
```

先のウェブ上で長軸半径 3，短軸半径 1.5 としての計算値と比較して有効桁8桁まで同一である．どちらがより正確な値なのかは不明だが，いずれも近似計算式としてはよいようだ．

多相性（polymorphism）

ここまでのところで，平面図形の下位クラス（サブクラス）として，円と楕円を定義し，それぞれに対する周長と面積を計算するメソッドを定義した．上記例で分かるように，同じ関数名で呼び出しを行っても，引数に与えるオブジェクトの種類（クラス）によって異なるメソッドが適応されて，それぞれ正しく周長と面積が計算された．このように**同じ呼び出しでも対象となるオブジェクトの型によって異なる手続きが実施される**ことを**多相性**という．

スロット特定子を使ってスロットアクセッサを定義

楕円の短軸半径を求めるためにメソッド short-radius を定義した．スロット特定子でスロットのアクセッサを指定すると，直接ユーザが関数 slot-value を使うことなく，short-radius と同様なメソッド long-radius を使うことができるようになる（内部では slot-value が使われている）．

```
《スロット特定子》::=《スロット名》|(《スロット名》[《スロットオプション》])
《スロット名》::=《シンボル》
《スロットオプション》::= {:reader 《リーダ関数名》}* |
                          {:writer 《ライタ関数名》}* |
                          {:accessor 《リーダ関数名》}* |
                          {:allocation 《アロケーションタイプ》} |
                          {:initarg 《初期化引数》}* |
                          {:initform 《書式》} |
                          {:type 《型指定子》} |
                          {:documentation 《文字列》}
```

先の実施例13に代わって，次のように定義したとする．

```
cl-user(3): (defclass ellipse (plain-figure)
              ((center-point :accessor center-point)
               (long-radius :accessor long-radius)
               (ellipcity :accessor ellipcity)))
#<standard-class ellipse>
```

すると，アクセッサ long-radius を使ってスロット値を取り出したり，(setf long-radius) でスロット値をセットしたりすることができるようになる．スロットオプションで :reader を指定したときはその名前のリーダ関数が，:writer 関数を指定した時はその名前のライタ関数が定義される．ライタ関数を指定した時はその関数の第1引数が新しい値，第2引数がオブジェクトである．アクセッサを指定した時は上記のように (setf 《関数名》) で新しい値をセットする．

```
cl-user(4): (setq my-ellipse (make-instance 'ellipse))
#<ellipse @ #x1024824c2>
cl-user(5): (setf (long-radius my-ellipse) 3)
3
cl-user(6): (long-radius my-ellipse)
3
```

with-slots でスロットアクセス

上記実施例16の代わりに，あたかもスロット名をローカル変数のように見せかける with-slots マクロを使って次のようにしてもよい．with-slots マクロでは既存のアクセッサは使われず，slot-value が使われることを注意しておく．

```
cl-user(15): (defmethod circumference ((an-ellipse ellipse))
               (with-slots (long-radius ellipcity) an-ellipse
                 (let ((short-radius (* long-radius ellipcity)))
                   (let ((abab (/ (- long-radius short-radius)
                                  (+ long-radius short-radius))))
                     (let ((3abab (* 3 abab abab)))
                       (* pi (+ long-radius short-radius)
                         (+ 1
                            (/ 3abab
                               (+ 10 (sqrt (- 4 3abab))))))))))
```

```
#<standard-method circumference (ellipse)>
cl-user(16): (circumference my-ellipse)
14.532672204015787d0
```

初期化引数を使う(initarg)

構造体のときに初期値を定義できたように，クラスの実現体を生成するときにスロットの値を指定したい場合には，あらかじめクラス定義時にスロット特定子で次のように初期化時に用いられる引数を指定しておく．

```
cl-user(18): (defclass circle (plain-figure)
               ((center-point :initarg :center-point)
                (radius       :initarg :radius)))
#<standard-class circle>
cl-user(19): (setq my-circle (make-instance 'circle :center-point 0 :radius 3))
#<circle @ #x1024b0632>
cl-user(20): (slot-value my-circle 'radius)
3
```

ここで，初期化引数は必ずしもキーワードシンボルでなくてもよいのだが，通常はキーワードシンボルとする．キーワードシンボルでない場合には，インスタンス定義時には初期化引数をクォートして与えなければならない．

スロット規定値を定義する(initform)

スロット値を与えないかぎり，オブジェクトのスロット値は未束縛(unbound)である．未束縛スロット値にアクセスしようとするとエラーとなる．スロット値をオブジェクト定義時に明示的に与えなくても，あるいは slot-value でセットしなくても，あるクラスの実現体に共通な規定値を定義したいときには次のようにすればよい．

```
cl-user(21): (defclass ellipse (plain-figure)
               ((center-point :initform 0)
                (long-radius :initform nil)
                (ellipcity :initform 1)))
#<standard-class ellipse>
cl-user(22): (setq my-ellipse (make-instance 'ellipse))
#<ellipse @ #x1022f21b2>
cl-user(23): (slot-value my-ellipse 'center-point)
0
cl-user(24): (slot-value my-ellipse 'long-radius)
nil
cl-user(25): (slot-value my-ellipse 'ellipcity)
1
```

ここで，スロット規定値として nil を設定することには意味がある．スロット値を取りに行ったとき，エラーを起こすことなく，まだしかるべきスロット値が与えられていないと解釈することができるからである．

問題 15.1 自分の好きなやり方で，抽象クラスである立体 solid と，その下位クラスである球 sphere と，立方体 cube と，円錐 cone を定義し，その実現体を生成せよ．

問題 15.2 球と，立方体と，円錐について，その表面積と体積を求めるメソッドを定義し，計算せよ．

15.3 上位クラスと下位クラス

クラス階層構造を説明する例題として，動物を考える．哺乳類も鳥も動物である．食肉獣も偶蹄類も哺乳類である．アホウドリもペンギンもダチョウも鳥である．シマウマとキリンは偶蹄類であるが，トラとチータは食肉獣である．これらの動物の概念をCLOSの機能を使って次のように定義する．

```
(defclass 動物 () ())
(defclass 哺乳類 (動物)
  ((毛がある :initform t)
   (授乳する :initform t)))
(defclass 鳥 (動物)
  ((羽がある :initform t)
   (抱卵する :initform t)))
(defclass 食肉獣 (哺乳類)
  ((肉を食べる :initform t)
   (鋭い歯がある :initform t)))
(defclass 偶蹄類 (哺乳類)
  ((ひずめがある :initform t)
   (反芻する :initform t)))
(defclass チータ (食肉獣)
  ((褐色である :initform t)
   (黒い斑点がある :initform t)))
(defclass トラ (食肉獣)
  ((褐色である :initform t)
   (黒い縞がある :initform t)))
(defclass キリン (偶蹄類)
  ((首が長い :initform t)
   (足が長い :initform t)
   (黒い斑点がある :initform t)))
(defclass シマウマ (偶蹄類)
  ((黒い縞がある :initform t)))
(defclass ダチョウ (鳥)
  ((首が長い :initform t)
   (足が長い :initform t)
   (黒と白の色である :initform t)))
(defclass ペンギン (鳥)
  ((黒と白の色である :initform t)
   (泳ぐ :initform t)))
(defclass アホウドリ (鳥)
  ((飛ぶ :initform t)))
```

問題 15.3 上記記述を参考に，この動物のクラス階層図を作成せよ．

CLOS クラスは Common Lisp の型である

一匹のチータを make-instance してみる．そのチータはもちろん食肉獣であり哺乳類である．15.8節で詳細に述べるが，CLOS クラスは Common Lisp の型システムと統合されており，新しく作られたクラスは Common Lisp の型となる．

```
cl-user(10): (setq a-cheeta
                   (make-instance 'チータ))
#<チータ @ #x10248c682>
cl-user(11): (type-of a-cheeta)
チータ
cl-user(12): (typep a-cheeta 'チータ)
t
cl-user(13): (typep a-cheeta '食肉獣)
t
cl-user(14): (typep a-cheeta '哺乳類)
t
cl-user(15): (typep a-cheeta '動物)
t
```

下位クラスは上位クラスの副型となる．

```
cl-user(4): (subtypep (find-class 'チータ) (find-class '哺乳類))
t
t
cl-user(5): (subtypep (find-class 'チータ) (find-class '動物))
t
t
```

ここで，find-class はそういう名前のクラスのLispオブジェクトを返す関数である．

実現体はそのクラスとすべての上位クラスからスロットを継承する

チータクラスに直接には「毛がある」とか「授乳する」などの情報は記述されていないが，チータの上位クラスである哺乳類にはそういう情報が記述してある．するとあるクラスの実現体はそのクラスとすべての上位クラスの情報をすべて継承しているので，次のような答えが返ってくる．

```
cl-user(6): (defun is-this-true? (animal property)
              (cond ((find-class property nil)
                     (typep animal property))
                    ((and (slot-exists-p animal property)
                          (slot-value animal property)))))
is-this-true?
cl-user(7): (is-this-true? (make-instance 'チータ) '食肉獣)
t
cl-user(8): (is-this-true? (make-instance 'チータ) '毛がある)
t
cl-user(9): (is-this-true? (make-instance 'チータ) '泳ぐ)
nil
```

(find-class《クラス名》nil) はそういう名前のクラスがあればそのクラスを返し，さもなければ nil を返す．ここに nil を与えないと，クラスがなければエラーとなってしまう．(slot-exists-p《オブジェクト》《スロット名》) はもしそのオブジェクトにそういう名前のスロットがあれば真を返し，なければ偽を返す．

下位クラスの情報は上位クラスに優先する

上記の定義では鳥のところで飛ぶ飛ばないを記述せずに，アホウドリは飛ぶがペンギンとダチョウは飛ばないとした．ここで，一般的な常識として鳥は飛ぶとして，ただしペンギンとダチョウは飛ばないという定義をする．そしてアホウドリは上位クラスの鳥から飛ぶ性質を継承させるものとする．

```
cl-user(32): (defclass 鳥 (動物)
               ((羽がある :initform t)
                (抱卵する :initform t)
                (飛ぶ     :initform t)))
#<standard-class 鳥>
cl-user(33): (defclass ペンギン (鳥)
               ((黒と白の色である :initform t)
                (飛ぶ :initform nil)
                (泳ぐ :initform t)))
#<standard-class ペンギン>
cl-user(34): (defclass ダチョウ (鳥)
               ((首が長い :initform t)
                (足が長い :initform t)
                (黒と白の色である :initform t)
                (飛ぶ     :initform nil)))
cl-user(35): (defclass アホウドリ (鳥)
               ())
#<standard-class アホウドリ>
```

すると，鳥とペンギンについて相反する定義がされているわけであるが，同じスロット名のスロットがある場合には，下位クラスの情報が上位クラスの情報に優先（オーバライド）するので，次のような結果になる．

```
cl-user(36): (setq a-penguin
                   (make-instance 'ペンギン))
#<ペンギン @ #x102310022>
cl-user(37): (is-this-true? a-penguin '飛ぶ)
nil
cl-user(38): (is-this-true? a-penguin '泳ぐ)
t
cl-user(39): (setq an-albatross
                   (make-instance 'アホウドリ))
#<アホウドリ @ #x10249a242>
cl-user(40): (is-this-true? an-albatross '飛ぶ)
t
cl-user(41): (is-this-true? an-albatross '泳ぐ)
nil
```

このように鳥は飛ぶという知識を鳥全般の既定値としておいて，しかしペンギンは飛ばないという知識を個別に与えるような方法は，一般に**デフォールト推論**と呼ばれている．デフォールト推論を論理として実装することは大変難しいが，フレームとかオブジェクト指向のようにクラスの間で下位クラスのオーバライドがある場合には簡単である．

円と楕円，どちらが上位概念か？

さて，動物のクラス定義では，その分類学上の上下関係がクラス定義における上下関係と矛盾なく合致していた．つまり，一般に下位クラスは上位クラスのスロットを継承するが，そのオブジェクト指向プログラミングの基本原則が動物の属性の継承にうまく利用できた．それでは円と楕円の関係ではどうだろうか．これまでのところでは円も楕円もそれぞれ独立に平面図形の下位クラスとして定義された．これに上位下位の関係付けをしたい．円は楕円の特殊である下位クラスか，それとも楕円が円の拡張概念なので下位クラスか．円は楕円率が1の場合に相当する楕円の特殊なものとすれば，円は楕円の下位クラス（サブクラス）とするのが適当に思える．一方，円の中心点および半径に対して楕円は楕円率が加わったものと考えれば，楕円が円の下位クラスとするのが妥当なようにも思える．もちろん円と楕円は別のものと定式化することも可能である．実現体を作ることのできるクラスが本物のクラスであって，その上位クラスは直接実現体を作ることはできない抽象クラスであるとすれば，円と楕円は別物とすることが合理的になる．チータでもなくペンギンでもない抽象的な動物というものは存在しないが，円も楕円も存在することはできるから．

上位クラスのメソッドの継承と下位クラスのメソッドの優先

円は楕円の下位クラスとする場合の定義の仕方を，以下に説明する．まず最初に，平面図形-->楕円-->円の階層構造を定義するというのはもちろんのことだが，ここでは楕円の定義で長軸半径(long-radius)とならんでただの半径(radius)という名前を初期化引数にもアクセッサにも定義したというところを見てほしい．先に述べたように，楕円において短軸半径は長軸半径から楕円率で計算されることにして，楕円においてもただの半径といえば，それは長軸半径のこととしたわけである．そして円についても楕円から継承されるスロット名は実は long-radius であるが，アクセッサとしてただの radius でそれにアクセスするようにした．

```
cl-user(1): (defclass plain-figure () ())
#<standard-class plain-figure>
cl-user(2): (defclass ellipse (plain-figure)
              ((center-point :initarg :center :accessor center)
               (long-radius :initarg :long-radius :initarg :radius
                            :accessor long-radius :accessor radius)
               (ellipcity :initarg :ellipcity :initform 1
                          :accessor ellipcity)))
#<standard-class ellipse>
cl-user(3): (setq foo (make-instance 'ellipse :long-radius 3))
#<ellipse @ #x102272862>
cl-user(4): (long-radius foo)
3
cl-user(5): (ellipcity foo)
1
cl-user(6): (defclass circle (ellipse) ())
#<standard-class circle>
cl-user(7): (setq bar (make-instance 'circle :radius 3))
#<circle @ #x102288b42>
cl-user(8): (radius bar)
3
cl-user(9): (ellipcity bar)
1
```

次に，周長と面積を求める計算式を楕円について定義する．すると円についての計算でも楕円の計算式を使って（楕円率を1として）計算される．実は必ずしも最初に総称関数を定義する必要はない．先に個々のメソッドを定義してもよいのである．

```
cl-user(10): (defmethod short-radius ((an-ellipse ellipse))
               (* (long-radius an-ellipse)
                  (ellipcity an-ellipse)))
#<standard-method short-radius (ellipse)>
cl-user(11): (defmethod area ((an-ellipse ellipse))
               (* pi (long-radius an-ellipse)
                     (short-radius an-ellipse)))
#<standard-method area (ellipse)>
cl-user(12): (defmethod circumference ((an-ellipse ellipse))
              (with-slots (long-radius ellipcity) an-ellipse
                (let ((short-radius (* long-radius ellipcity)))
                  (let ((abab (/ (- long-radius short-radius)
                                 (+ long-radius short-radius))))
                    (let ((3abab (* 3 abab abab)))
                      (* pi (+ long-radius short-radius)
                         (+ 1
                           (/ 3abab
                              (+ 10 (sqrt (- 4 3abab))))))))))
#<standard-method circumference (ellipse)>
cl-user(13): (area bar)
28.274333882308138d0
cl-user(14): (area foo)
28.274333882308138d0
cl-user(15): (circumference foo)
18.84955592153876d0
cl-user(16): (circumference bar)
18.84955592153876d0
```

このように，**上位クラスで定義されたメソッドはスロットと同様に下位クラスに継承される**．

またスロットのときと同様に，**下位クラスでメソッドを定義すると，それは上位クラスのメソッドに優先する**．以下のコードは，円で楕円率を変更しようとしたときにエラーを起こすようなコードである．

```
cl-user(18): (defmethod (setf ellipcity)
               (new-value (a-circle circle))
               (error "You cannot modify the ellipcity of circle."))
#<standard-method (setf ellipcity) (t circle)>
cl-user(19): (setf (ellipcity bar) 0.5)
Error: You cannot modify the ellipcity of circle.
cl-user(20): (setf (ellipcity foo) 0.5)
0.5
```

上記では楕円に対しては `(defmethod (setf ellipcity) (new-value (an-ellipse ellipse)) ... )` のような，円に対しては `(defmethod (setf ellipcity) (new-value (a-circle circle)) ... )` のようなメソッドが定義された．複数の似たメソッドが一つの総称関数およびそれに関連する一群のメソッドとして認識されるために大事なことは，関数名が同一であることと関数引数の必須パラメータや付加パラメータの数と順番が一致していることであり，関数引数の名前やその指定クラスなどは問題ではない．この考えを総称関

数に属するメソッドの適合ラムダリストと呼んでいる．詳しくはその仕様を参照されたい．なお，付加パラメータやキーワードパラメータを引数特定子(parameter-specializer)にすることはできない．

15.4 メソッド修飾子

これまで述べてきたメソッドは**基本メソッド**(primary method)と呼ばれるもので，CLOSにはこの基本メソッドに加えて**補助メソッド**(auxiliary method)と呼ばれるものがある．CLOSの標準メソッド結合(standard method combination)では，それらは次の三種類である．

1. :before — 基本メソッドの実行前に実行される．主に前準備とか条件検査のような副作用に期待してプログラムされる．
2. :after　— 基本メソッドの実行後に実行される．主に後始末とか結果確認のような副作用に期待してプログラムされる．
3. :around — このメソッドの中で囲うようにして基本メソッドを呼び出す．

基本メソッドと :before，:after に関する次の実行例を見られたい．

```
cl-user(2): (defclass class1 () ())
#<standard-class class1>
cl-user(3): (defclass class2 (class1) ())
#<standard-class class2>
cl-user(4): (defmethod foo ((x class2))
              (format t "~%;; This is primary.")
              x)
#<standard-method foo (class2)>
cl-user(5): (defmethod foo :before ((x class2))
              (format t "~%;; ~S is given." x))
#<standard-method foo :before (class2)>
cl-user(6): (defmethod foo :after ((x class2))
              (format t "~%;; Type of ~S is ~S." x (type-of x)))
#<standard-method foo :after (class2)>
cl-user(7): (setq bar (make-instance 'class2))
#<class2 @ #x2159696a>
cl-user(8): (foo bar)

;; #<class2 @ #x2159696a> is given.
;; This is primary.
;; Type of #<class2 @ #x2159696a> is class2.
#<class2 @ #x2159696a>
```

最初に :before メソッドが実施され，そのあと基本メソッドが実施され，最後に :after メソッドが実施された．返された値は基本メソッドで返す値であり，:before メソッドの値を基本メソッドで受け取る方法はないことに注意されたい．同様に，基本メソッドで返す値を :after メソッドで受け取る方法もない．

もし基本メソッドを書いてしまってから，その中身を書き換えることなく柔軟に拡張したい場合には，:around メソッドを用いることができるが，後で述べるように上位クラスで定義されたメソッドとの結合の順番（優先順位）には十分な注意が必要である．ここでは単独クラスでの :around メソッドの働きを示す．

まずは上記の :before，:after と同じことを :around でやってみよう．

```
cl-user(2): (defclass class1 () ())
#<standard-class class1>
cl-user(3): (defclass class2 (class1)
              ((name :initarg :name)))
#<standard-class class2>
cl-user(4): (defmethod foo ((x class2))
              (format t "~%;; This is primary.")
              (format t "~%;; Its name is ~S."
                        (slot-value x 'name))
              x)
#<standard-method foo (class2)>
cl-user(5): (defmethod foo :around ((x class2))
              (format t "~%;; ~S is given." x)
              (setf (slot-value x 'name) 'bbbbar)
              (prog1 (call-next-method)
                (format t "~%;; Type of ~S is ~S." x (type-of x))))
#<standard-method foo :around (class2)>
cl-user(6): (setq bar (make-instance 'class2 :name 'bar))
#<class2 @ #x215944b2>
cl-user(7): (foo bar)

;; #<class2 @ #x215944b2> is given.
;; This is primary.
;; Its name is bbbbar..
;; Type of #<class2 @ #x215944b2> is class2.
#<class2 @ #x215944b2>
```

prog1 は progn と似ているが返す値は，最初の書式の値，しかも多値の場合は**多値の最初の値だけ**である．

:around メソッド中の call-next-method は基本メソッドを呼び出すが，このとき**与える引数に変更がないならば**，このように引数を明記することなく，呼び出すことができる．その場合にはシステムはメソッド結合において**効率の良い実行コードを生成する**ことができる．ここで，オブジェクトの名前として最初に与えた bar という名前が call-next-method の前で変更されたが，オブジェクトそのものは同一のオブジェクトが基本メソッドに引き渡されていることを注意しておく．

しかし，もしオブジェクトそのものを変更せざるを得ないような場合にはどうしたらよいだろうか．そんな時にこそ :around メソッドの出番となる．:around メソッドでは基本メソッドの周りに，どんなコードでも書くことができて，基本メソッドとのやり取りも自由自在である．

```
cl-user(11): (defmethod foo :around ((x class2))
               (format t "~%;; ~S is given." x)
               (setq x (make-instance (class-of x) :name 'baaaaa))
               (let ((result (call-next-method x)))
                 (format t "~%;; Type of ~S is ~S."
                           result (type-of result))
               result))
#<standard-method foo :around (class2)>
cl-user(12): (setq bar (make-instance 'class2 :name 'bar))
#<class2 @ #x215e1dfa>
cl-user(13): (foo bar)
```

```
;; #<class2 @ #x215e1dfa> is given.
;; This is primary.
;; Its name is baaaaa.
;; Type of #<class2 @ #x215ed43a> is class2.
#<class2 @ #x215ed43a>
```

このコードでは，ユーザが与えたオブジェクトとは異なるオブジェクトが作られて，それが基本メソッドに与えられた．

あるいは，基本メソッドそのものには手を付けずに，大域変数などその実行環境を変化させるとか，18.3節のコンディションシステムを用いてエラーハンドリングを行うというような場合には:aroundメソッドが最適となる．最後に言うまでもないが，自分だけではなく他のどんなメソッドでも :around メソッド中で呼び出すこともできるので，見かけの呼び出し形式や基本メソッドを変えずに大きくデリゲーションを導入するというようなことにも有効である．

15.5 多重継承のクラス優先順位

オブジェクト指向言語における多重継承問題には様々な議論があった．Java にはインターフェースというものはあるが，関数の多重継承はない．C++ に多重継承はあるが，そのやり方は CLOS とは大いに異なる．多重継承問題は数学的に言えば，半順序列の集合をどうやって全順序列に直すかという問題であり，これをCLOSでは**各クラスから上位クラスを見たときのトポロジカル・ソーティング**という考え方で解決した．トポロジカル・ソーティングの利点は，どの半順序列についても，半順序列における順序と全順序列の順序の間に（中間に他のエンティティが入ることがあっても）矛盾はなく，半順序の上下関係が全順序においても保たれているということである．もちろんその前提として，優先順位に巡回があってはならない．その場合には，CLOSシステムはエラーを発する．

```
cl-user(2): (defclass A (B) ())
#<standard-class A>
cl-user(3): (defclass B (C) ())
#<standard-class B>
cl-user(4): (defclass C (A) ())
#<standard-class C>
cl-user(5): (make-instance 'A)
Error: While computing the class precedence list of the class named A.
The class #<standard-class C> has a circular superclass dependency.
[condition type: program-error]
cl-user(6): (make-instance 'B)
Error: While computing the class precedence list of the class named B.
The class #<standard-class A> has a circular superclass dependency.
[condition type: program-error]
cl-user(7): (make-instance 'C)
Error: While computing the class precedence list of the class named C.
The class #<standard-class B> has a circular superclass dependency.
[condition type: program-error]
```

しかし，それでもまだいくつかの微妙に考慮しなければならない点が残されている．たとえば，Employee クラスと Student クラスがあったとする．

```
cl-user(8): (defclass Employee () ())
#<standard-class Employee>
cl-user(9): (defclass Student () ())
#<standard-class Student>
```

生協の従業員でかつ学生であるクラスは第1の身分として従業員であり，次の資格として学生なので，次のように定義する．

```
cl-user(10): (defclass Coop-Student (Employee Student) ())
#<standard-class Coop-Student
```

クラス定義における上位クラスの記述では，直接上位クラスを複数記述できるが，それらの直接上位クラス間では，**左にあるほうがより特定的で優先順位が高い**のだ．

大学院学生は（米国では）教授に雇われる研究生であるが，第1の身分としては学生であり，次の属性として従業員なので，次のように定義する．

```
cl-user(11): (defclass Research-Student (Student Employee) ())
#<standard-class Research-Student>
```

ここまでは，何の問題もない．そういう人材をいくらでも作ることができる．ところが，生協の社会人学生でありかつ研究生としようとすると，これはエラーを引き起こす．なぜならば，Coop-Studentにおける局所優先順位(Employee Student)とResearch-Studentにおける局所優先順位(Student Employee)が矛盾するからだ．

```
cl-user(12): (defclass Coop-Resarch-Student (Coop-Student Resarch-Student) ())
#<standard-class Coop-Resarch-Student>
cl-user(13): (make-instance 'Coop-Resarch-Student)
Error: While computing the class precedence list of the class
       named Coop-Resarch-Student.
The class named Resarch-Student is a forward referenced class.  The class named Resarch-
Student is a direct superclass of the class named Coop-Resarch-Student.
[condition type: program-error]
```

たとえエラーが起こらない場合でも，上位クラス名の並び順が適応されるメソッド選択に影響するので，似たような優先度にあるメソッド間で微妙な問題が起こる可能性がある．たとえば，次の例を見てみよう．

```
cl-user(2): (defclass Curtain () ())
#<standard-class Curtain>
cl-user(3): (defclass Heat-Insulating-Object () ())
#<standard-class Heat-Insulating-Object>
cl-user(4): (defclass Sound-Insulating-Object () ())
#<standard-class Sound-Insulating-Object>
cl-user(5): (defclass Heat-Insulation-Curtain
                (Curtain Heat-Insulating-Object)
              ())
#<standard-class Heat-Insulation-Curtain>
cl-user(6): (defclass Sound-Insulation-Curtain
               (Curtain Sound-Insulating-Object)
              ())
#<standard-class Sound-Insulation-Curtain>
cl-user(7): (defclass Heat-Sound-Insulation-Curtain
               (Curtain Heat-Insulating-Object Sound-Insulating-Object)
```

```
              ())
#<standard-class Heat-Sound-Insulation-Curtain>
cl-user(8): (defclass Sound-Heat-Insulation-Curtain
              (Curtain Sound-Insulating-Object Heat-Insulating-Object)
              ())
#<standard-class Sound-Heat-Insulation-Curtain>
cl-user(9): (setq curtain1
                  (make-instance 'Heat-Sound-Insulation-Curtain))
#<Heat-Sound-Insulation-Curtain @ #x216e415a>
cl-user(10): (setq curtain2
                   (make-instance 'Sound-Heat-Insulation-Curtain))
#<Sound-Heat-Insulation-Curtain @ #x215a47ea>
```

このときもしHeat-Insulating-ObjectとSound-Insulating-Objectにメソッド定義がそれぞれメソッド１とメソッド２となっていれば，その実行順序はHeat-Sound-Insulation-Curtainのインスタンスに対してはメソッド１，メソッド２の順であり，Sound-Heat-Insulation-Curtainのインスタンスに対してはその逆である．だから，両者のメソッドが直行していればよいが，共通のスロットや共通の変数を参照したり変更したりするような相互干渉があると，問題となる．

先の動物のタクソノミーでは，オントロジー的に抽象のクラスから特殊なクラスへ上下関係が記述された．それは上位クラスから下位クラスへ末広がりにツリーが広がっていくイメージだ．上記の防音カーテンと断熱カーテンの例では，カーテンに対して防音性と断熱性の性質を追加することで，防音カーテンや断熱カーテンを作ることができた．防音断熱カーテンは防音性と断熱性を兼ね備えたカーテンだ．こちらの方法は mix-in と呼ばれる方法で，複雑な性質を有するクラスは基本となるクラスに様々な属性を任意に追加する．属性のクラスは単純でその上位クラスがないのがふつうである．この場合は実クラスが下位にあって，上位の方へ単純な性質を有するクラスへ向かって広がっていくイメージになる．この両者は，オブジェクトをどう組織化するかという点において典型的な二つのプログラミング・パラダイムである．

15.7節では，タクソノミー的でもなければmix-in的でもないオブジェクト指向プログラミングの例を，文献検索システムを例題に取り上げる．

15.6 メソッド結合の優先順位

前節で述べたように，スロットとメソッドの継承は個々のクラスごとに作られるところの，そのクラスから見た上位クラスの全順序リストに従って実行される．このリストを**クラス優先リスト**(class-precedence-list)と呼ぶ．クラス優先リストはそのクラスのインスタンスを作成するときにはじめて作られる．一方，15.4節で述べたように，標準メソッド結合では，基本メソッド，:before メソッド，:after メソッド，:around メソッドと4種類のメソッドがある．一つのメソッド呼び出しにおいてそのラムダリストに適合するメソッドが上位クラスにも複数あったとき，どのような順番で各メソッドが実行されるのかは一見複雑である．

1. もし基本メソッドしかないのなら，クラス優先リストにおいて最も特定的な（最も下位クラスにある）メソッドが一つ実行される．その実行メソッドの中から上位クラスのメソッドを起動するに

は call-next-method が用いられる．call-next-method がなければ，上位クラスのメソッドは下位クラスのメソッドによってオーバーライドされる．

2. :beforeと :after メソッドはないものとする．もしどこかに :around メソッドがあれば，そのうちで最も特定的な :around メソッドが一つ実行される．その :around メソッドよりも特定的な基本メソッドはそのときは無視される．call-next-method の呼び出しがなければ，その :around メソッドの返す値が総称関数の返す値となる．:around メソッドにおいて call-next-method が呼ばれると，次に特定的な :around メソッドがあればそれが実行される．そのときより特定的な基本メソッドがあっても無視される．もしそのような :around メソッドがなければ，そこであらためて全体の中から最も特定的な基本メソッドが実行される．このときその基本メソッドが最も特定的な :around メソッドよりも特定的であっても実行される．最も特定的でない :around メソッドが最も特定的な基本メソッドの返す値を受け取る．
3. :before と :after メソッドはすべてのメソッドが集められて，最初にある基本メソッドが実行される直前ですべての :before メソッドが特定的なものから順番に実行され，:after メソッドは基本メソッド実行後に一般的なものから順番に実行される．その基本メソッドよりも特定的な :before と :after メソッドも実行される．ただし，:beforeや:afterメソッドが実行されるためには，どこかに必ず基本メソッドがなければならない．

:around メソッドにも基本メソッドにも call-next-method があるが，最初は :around メソッドの一連の call-next-method が実行され，次に基本メソッドの一連の call-next-method が実行される．複数ある:around メソッドと基本メソッドの使い分けは，:before，:afterメソッドと比べて難しく，もっと実用上研究されてもよい課題である．

15.7 文書検索，もう一つのオブジェクト指向的プログラミング

先にオブジェクト指向プログラミングのパラダイムとして，タクソノミーと mix-in を挙げた．タクソノミー的パラダイムでは，階層構造は深くツリー状になり，上位クラスから下位クラスに向かって末広がりに広がる．一方，mix-in では一つの深い幹に軽い枝が何本もついて下位クラスから上位クラスに広がっていくイメージとなる．ここではもう一つのパラダイムとして，階層構造はほとんどなく，単独のメソッドを持つ単独のクラスが分散して存在し，プログラマーが必要に応じてそれらのモジュールを自在に結合して使うようなものを紹介する．自在に組み合わせる時に利用されるのが，複合 CLOS クラス定義時の局所直接上位クラスリストである．

全文検索エンジン Lucene の開発者である Doug Cutting 氏は，Java による Lucene 開発以前に，CLOS を利用した文書検索のプロトタイプを発表した．それにプログラムの全リストが掲載されている．全部合わせても高々150行ほどであり，ここで紹介するのはそのリストの内容である．

トークナイザは最初にトークンを切り出す

入力ストリームから最初にトークンを切り出すのはトークナイザの仕事である．プログラムは10.3節コルーチンの場合とは少し異なって，連続するアルファベット文字のみをトークンとして切り出すが，それ以外の文字は捨ててしまう．

with-output-to-string は新しく出力ストリームを生成して与えられた変数にそれを束縛し，そのストリームに出力された結果を文字列として返すようなCommon Lispのマクロである．次の例を見られたい．

```
cl-user(9): (with-output-to-string (string-stream)
              (format string-stream "Hello, World!"))
"Hello, World!"
```

トークナイザは独立した単独のクラスで，それはスロットとして文字の入力ストリームを提供する．トークナイザに密接に結びついたメソッド next-token は，トークナイザ・インスタンスから文字入力ストリームをもらって，その入力ストリームからトークンを切り出す．この実装ではアルファベット文字であればそれは string-stream に1文字づつ出力され蓄積されるが，そうでなければ何もされない．つまり捨てられてしまう．最初に非アルファベット文字が続いたときは捨ててしまうだけだが，トークン蓄積の最中に非アルファベット文字を読んだときはそれまでに蓄積されたトークンを文字列として返す．

```
(defclass tokenizer ()
  ((char-stream :initarg :char-stream)))
(defmethod next-token ((token-stream tokenizer))
  (with-slots (char-stream) token-stream
    (with-output-to-string (string-stream)
      (let ((in-token-p nil))
        (loop (let ((char (read-char char-stream nil)))
                (cond ((null char) ; EOF
                       (if in-token-p (return)
                         (return-from next-token nil)))
                      ((alpha-char-p char)
                       (write-char char string-stream)
                       (setq in-token-p t))
                      (t (if in-token-p (return)))))))))))
```

以下に実施例を示す．

```
cl-user(2): (with-open-file (stream "test1.txt")
              (let ((tokeninput
                      (make-instance 'tokenizer
                                     :char-stream stream)))
                (loop for token = (next-token tokeninput)
                    while token
                    collect token)))
("Mrs" "Rachel" "Lynde" "lived" "just" "where" "the" "Avonlea" "main" "road" ...)
cl-user(3): (pprint *)

("Mrs" "Rachel" "Lynde" "lived" "just" "where" "the" "Avonlea" "main"
 "road" "dipped" "down" "into" "a" "little" "hollow" "fringed" "with"
 "alders" "and" "ladies" "eardrops" "and" "traversed" "by" "a" "brook"
 "that" "had" "its" "source" "away" "back" "in" "the" "woods" "of"
 "the" "old" "Cuthbert" "place" "it" "was" "reputed" "to" "be" "an"
 "intricate" "headlong" "brook" "in" "its" "earlier" "course" "through"
 "those" "woods" "with" "dark" "secrets" "of" "pool" "and" "cascade"
 "but" "by" "the" "time" "it" "reached" "Lynde" "s" "Hollow" "it" "was"
 "a" "quiet" "well" "conducted" "little" "stream" "for" "not" "even"
 "a" "brook" "could" "run" "past" "Mrs" "Rachel" "Lynde" "s" "door"
 "without" "due" "regard" "for" "decency" "and" "decorum" "it"
 "probably" "was" "conscious" "that" "Mrs" "Rachel" "was" "sitting"
 "at" "her" "window" "keeping" "a" "sharp" "eye" "on" "everything"
```

```
  "that" "passed" "from" "brooks" "and" "children" "up" "and" "that"
  "if" "she" "noticed" "anything" "odd" "or" "out" "of" "place" "she"
  "would" "never" "rest" "until" "she" "had" "ferreted" "out" "the"
  "whys" "and" "wherefores" "thereof")
```

ノルマライザがある

ノルマライザは解析に都合がよいように文字列の標準化を行うものであるが，ここでは単に英大文字を英小文字にするだけである．

```
(defclass normalizer () ())
(defmethod next-token ((token-stream normalizer))
  (let ((token (call-next-method)))
    (if token (string-downcase token) nil)))
```

ノルマライザが提供するのはメソッドだけである．ここで元となる**トークンはcall-next-methodを呼び出して上位クラスからもらってくる**．

文書アナライザがある

ノルマライザの出力は次に解析にかけられる．ただし，解析と言ってもここで行うのは冠詞や前置詞など，文書検索の観点から見て，扱う価値がないものとして無視されるべき単語を見つけて削除するだけである．文書検索では意味の解釈は行われないことに留意されたい．

ストップリストはストップワードを提供するが，next-tokenメソッドは**上位クラスからトークンをもらって**，それがストップワードであれば何もせずに次のトークンに進む．さもなければ，もらったトークンを値として返す．

```
(defclass stop-list ()
  ((stop-words :initform '("a" "an" "and" "or" "by" "for" "of" "the" "to" "with"))))
(defmethod next-token ((token-stream stop-list))
  (with-slots (stop-words) token-stream
    (loop (let ((token (call-next-method)))
            (cond ((null token) (return nil))                 ; EOF
                  ((member token stop-words :test #'string=)) ; nothing done
                  (t (return token)))))))
```

ここで，stop-list のtoken-stream は他のものとは違って，実はストリームではなくてストップリストになっていることを注意しておく．call-next-methodで得られるtokenは**上位クラスを呼び出すことで得られる文字列**である．

パイプラインを構成するのはプログラマーの仕事

これで文書アナライザを構成する主要なクラスが出そろったわけだが，最後にそれらを組み合わせてアナライザを構成するのは，プログラマーの仕事である．ここでCLOSクラス定義における直接上位クラスのリストが利用される．

```
(defclass simple-analysis-pipeline (stop-list normalizer tokenizer) ())
```

このクラスは stop-list と normalizer と tokenizer をパイプラインのように接続する．ストップリストもノルマライザも上位クラスを call-next-method で呼び出して，トークンをもらっていたことを思い出してほしい．また，ストップワードはすべて英小文字で登録されていたことも思い出してほしい．だからこの上位クラスリストの順番はこれ以外にはないわけだ．一般に，オブジェクト指向プログラミングではタクソノミーパラダイムでも mix-inパラダイムでも，何とか相互干渉することのないように機能をモジュール化することにプログラマーは知恵を傾ける．しかしここでのやり方は真逆で，上下関係でクラスメソッド間のやりとりがあることを前提としてプログラムされている．それでもうまく機能分離ができているために，それぞれのメソッドで高機能化を図っても，それほど他のメソッドには影響しない．少し気持ち悪いのは，next-token の引数が，stop-listの場合とtokenizerの場合で全く意味するものがことなることだ．しかも，normalizer の token-stream は一切使われない．もしこのようなプログラミングをするのなら，ドキュメンテーションをしっかりやらないと大規模システムでは容易に破たんしてしまうだろう．

この単純なパイプラインを利用した単純なアナライザは次のようになる．

```
(defclass simple-analyzer () ())
(defmethod make-token-stream ((analizer simple-analyzer) char-stream)
  (make-instance 'simple-analysis-pipeline :char-stream char-stream))
```

前述の実施例と同じテキストによるアナライザの実施例を以下に示す．大文字が小文字に変換されていること，ストップワードが無くなっていることに注目されたい．

```
cg-user(3): (with-open-file (stream "test1.txt")
  (let ((tokeninput (make-token-stream
                     (make-instance 'simple-analyzer)
                     stream)))
    (loop for token = (next-token tokeninput)
        while token
        collect token)))
("mrs" "rachel" "lynde" "lived" "just" "where" "avonlea" "main" "road" ...)
cg-user(4): (pprint *)

("mrs" "rachel" "lynde" "lived" "just" "where" "avonlea" "main" "road"
 "dipped" "down" "into" "little" "hollow" "fringed" "alders" "ladies"
 "eardrops" "traversed" "brook" "that" "had" "its" "source" "away"
 "back" "in" "woods" "old" "cuthbert" "place" "it" "was" "reputed" "be"
 "intricate" "headlong" "brook" "in" "its" "earlier" "course" "through"
 "those" "woods" "dark" "secrets" "pool" "cascade" "but" "time" "it"
 "reached" "lynde" "s" "hollow" "it" "was" "quiet" "well" "conducted"
 "little" "stream" "not" "even" "brook" "could" "run" "past" "mrs"
 "rachel" "lynde" "s" "door" "without" "due" "regard" "decency"
 "decorum" "it" "probably" "was" "conscious" "that" "mrs" "rachel" "was"
 "sitting" "at" "her" "window" "keeping" "sharp" "eye" "on" "everything"
 "that" "passed" "from" "brooks" "children" "up" "that" "if" "she"
 "noticed" "anything" "odd" "out" "place" "she" "would" "never" "rest"
 "until" "she" "had" "ferreted" "out" "whys" "wherefores" "thereof")
```

コーパスがある

コーパスとは，本来は「自然言語処理の研究に用いるため、自然言語の文章を構造化し大規模に集積したもの」(Wikipedia)であるが，ここではコーパスとはある一つのファイル(ファイルコーパス)あるいはディレクトリ下にあるファイル群(ディレクトリコーパス)のことである．

文字列コーパスとは文字列のリストを有しているもののことで，メソッドopen-documentはIDを与えて文字列コーパスからID番目の文字列を取り出してその入力ストリームを作りだす．

```
(defclass string-corpus ()
  ((strings :initarg :strings)))
(defmethod open-document ((corpus string-corpus) id)
  (with-slots (strings) corpus
    (make-string-input-stream (nth id strings))))
```

コーパスは文書の集積であり，IDは本来はコーパス中の文書を指し示すものであるが，後述のデモにおいては単にコーパスディレクトリ下にあるファイルを順番に指し示すものにすぎない．

インデックス・ストア

ある単語がコーパス中のどの文書に存在するかを明らかにするものが，インデックス・ストアである．以下のプログラムではハッシュテーブルが用いられるが，ハッシュのキーとなるのが単語であり，値となるのがIDである．MAP-REDUCEで紹介した単語IDとは異なることを注意しておく．

```
(defclass hash-store ()
  ((table :initform (make-hash-table :test #'equal))))

(defmethod get-mapping ((store hash-store) term)
  (with-slots (table) store
    (gethash term table)))

(defmethod (setf get-mapping) (value (store hash-store) term)
  (with-slots (table) store
    (setf (gethash term table) value)))
```

map-tokenはトークンストリームから順番にトークンを取り出して，引数にもらった関数を適応するが，index-documentではその関数というのはトークンをキーにしてIDを保存するものである．

```
(defun map-tokens (function token-stream)
  (loop (let ((token (next-token token-stream)))
          (if token (funcall function token) (return)))))

(defclass binary-index () ())

(defmethod index-document ((index binary-index) id)
  (map-tokens #'(lambda (token) (pushnew id (get-mapping index token)))
              (make-token-stream index (open-document index id))))
```

メソッドget-mappingが有効であるためにはその第１引数はhash-storeでなければならないが，ここのコードではそれは明らかではない．実はあとのデモではdemoクラスが定義され，それはhash-storeでもあ

り，かつbinary-indexでもあるということが局所直接上位クラスリストの中に定義される．このあたりも通常のオブジェクト指向プログラミングのマナーからは逸脱して気持ち悪いところではある．

上述のindex-documentの結果，すべてのトークンについてそれがどの文書に含まれていたか，コーパスIDを値としてハッシュテーブルに保存される．get-binary-postingsはトークンをキーとして，それを含むコーパスIDをリストにして返す．get-frequency-postingsは，get-binary-postingsを用いてトークンが含まれる文書について，改めてその出現回数を計算する．get-term-frequencyはその名前と相違して文書中の出現回数ではなく，そのトークンが含まれる文書の数を返すことに注意されたい．

```
(defmethod get-binary-postings ((index binary-index) term)
  (get-mapping index term))

(defmethod get-term-frequency ((index binary-index) term)
  (length (get-binary-postings index term)))

(defmethod get-frequency-postings ((index binary-index) term)
  (mapcar #'(lambda (id)
              (let ((freq 0))
                (map-tokens #'(lambda (token)
                                (if (string= token term) (incf freq)))
                            (make-token-stream index
                                               (open-document index id)))
                (cons id freq)))
    (get-binary-postings index term)))
```

boolean-searchは複数の検索キーワードのANDとORの組み合わせの式について，検索結果を計算する．後述のデモを参照されたい．

```
(defmethod boolean-search ((app t) expr)
  (labels ((resolve (x)
                    (if (listp x)
                        (case (first x)
                          (and (intersection (resolve (second x))
                                             (resolve (third x))))
                          (or (union (resolve (second x))
                                     (resolve (third x)))))
                      (get-binary-postings app x))))
    (resolve expr)))
```

relevance-searchでは，複数の検索キーワードの出現回数を考慮して，関連度の高い順にIDを並べて出力する．ただしこの計算では連語(collocation)については考慮されてなくて，各単語の出現回数を計算するのみであることを注意しておく．

```
(defmethod relevance-search ((app t) query &optional (threshold 10))
  (let ((terms ())
        (scores ()))
    (map-tokens #'(lambda (token)
                    (pushnew token terms :test #'string=))
                query)
    (dolist (term terms)
      (let ((weight (/ 1.0 (get-term-frequency app term))))
        (dolist (freq-pair (get-frequency-postings app term))
```

```
          (let* ((id (first freq-pair))
                 (freq (rest freq-pair))
                 (score-pair (assoc id scores)))
            (unless score-pair
              (setq score-pair (cons id 0.0)
                    scores (cons score-pair scores)))
            (incf (cdr score-pair) (* weight freq))))))
    (mapcar #'car
            (subseq (sort scores #'> :key #'cdr)
                    0
                    (min (length scores) threshold)))))
```

プロジェクト・グーテンベルグ

デモ用の素材として，プロジェクト・グーテンベルグより，いくつかの素材をダウンロードし，平文テキストとして用意した．クラスdemoはこれらの素材を一か所に集めたディレクトリの情報を持っているが，そのインスタンスを生成した時にこれらの素材の内容がfile-to-stringによってファイル単位で文字列になって，文字列コーパスのstringスロットの値となる．さらに，initialize-instance:afterによって文字列（コーパス）ごとにコーパスIDが与えられてインデックスとしてトークンをキーとしてコーパスIDがストアされる．

```
(defclass demo (string-corpus simple-analyzer hash-store binary-index) ()
  (:default-initargs
    :strings (mapcar #'file-to-string (directory "D:/ProjectGutenberg/"))))
(defmethod initialize-instance :after ((app demo) &key &allow-other-keys)
  (dotimes (id (length (slot-value app 'strings)))
    (index-document app id)))

(defun file-to-string (pathname)
  (with-output-to-string (string-stream)
    (with-open-file (file-stream pathname)
      (loop (let ((char (read-char file-stream nil)))
              (if char (write-char char string-stream) (return)))))))
```

この例題では，各ファイルの第１行目にはそのファイルのタイトルが書かれていることを前提としており，print-titlesは与えられた複数のコーパスIDに相当する文書をオープンして，その第１行目のみを順に印刷する．

```
(defmethod print-titles ((app demo) ids)
  (dolist (id ids (values))
    (format t "~&~3D ~A~%" id (read-line (open-document app id)))))
```

以下はデモ実施例である．

```
cl-user(2): (setq app (make-instance 'demo))
#<demo @ #x1022e03b2>
cl-user(3): (print-titles
 app
 (boolean-search app '(and "america" "constitution")))
  4 THE CONSTITUTION OF THE UNITED STATES OF AMERICA, 1787
cl-user(4): (print-titles
 app
 (boolean-search app '(and "american" "constitution")))
  1 Lincoln's First Inaugural Address
```

```
cl-user(5): (print-titles
 app
 (boolean-search app '(and (or "america" "american") "constitution")))
  4 THE CONSTITUTION OF THE UNITED STATES OF AMERICA, 1787
  1 Lincoln's First Inaugural Address
cl-user(6): (print-titles
 app
 (boolean-search app '(and "japan" "constitution")))
  3 THE CONSTITUTION OF JAPAN, 1946
cl-user(7): (print-titles
 app
 (boolean-search app "people"))
  5 The United States Bill of Rights.
  4 THE CONSTITUTION OF THE UNITED STATES OF AMERICA, 1787
  3 THE CONSTITUTION OF JAPAN, 1946
  1 Lincoln's First Inaugural Address
  0 JFK's Inaugural Address, January 20, 1961, 12:11 EST
cl-user(8): (print-titles
 app
 (boolean-search app "war"))
  5 The United States Bill of Rights.
  4 THE CONSTITUTION OF THE UNITED STATES OF AMERICA, 1787
  3 THE CONSTITUTION OF JAPAN, 1946
  2 Lincoln's Second Inaugural Address
  1 Lincoln's First Inaugural Address
  0 JFK's Inaugural Address, January 20, 1961, 12:11 EST
cl-user(9): (print-titles
  app
  (relevance-search
   app
   (make-token-stream
    app
    (make-string-input-stream "war"))))
  2 Lincoln's Second Inaugural Address
  0 JFK's Inaugural Address, January 20, 1961, 12:11 EST
  3 THE CONSTITUTION OF JAPAN, 1946
  4 THE CONSTITUTION OF THE UNITED STATES OF AMERICA, 1787
  1 Lincoln's First Inaugural Address
cl-user(10): (print-titles
 app
 (boolean-search app "emperor"))
  3 THE CONSTITUTION OF JAPAN, 1946
cl-user(11): (print-titles
 app
 (boolean-search app '(and "prime" "minister")))
  3 THE CONSTITUTION OF JAPAN, 1946
cl-user(12): (print-titles
  app
  (relevance-search
   app
   (make-token-stream
    app
    (make-string-input-stream "prime minister"))))
  3 THE CONSTITUTION OF JAPAN, 1946
```

このプログラムは実用的な観点から見ると，問題が多い．まず第1に，文字列コーパスはすべての文書中の文字列を取り込んで保持していることがある．これではスケールしない．また，Luceneでは文書IDに加

えて文書中のトークンIDを記録するが，ここではトークンIDはない．だからGoogle検索結果表示のように文書中の検索キーワードのまわりの文書を取りだすことができないし，出現頻度を求めるためには`get-frequency-postings`では再度該当文書を読み込む必要があった．しかしここに示されたプログラミングのアイデアをもとに，全文検索エンジンLuceneが作られ，さらにLuceneをベースにSolrが作られたことを考えると，アイデアを現実化するプロトタイプ開発用としてのCLOSの価値はここでも証明されたということができる．

15.8 CLOS と型システム

CLOSクラスは型である

構造体のときと同様に，CLOSのクラス定義をすると，それはCommon Lispの型システムの一部となる．具体的には，typep，type-of，subtypepなどの関数で有効になる．上記「円と楕円，どちらが上位概念か？」の以下の例を見られたい．

```
cl-user(2): (defclass plain-figure () ())
#<standard-class plain-figure>
cl-user(3): (defclass ellipse (plain-figure)
              ((center-point :initarg :center :accessor center)
               (long-radius :initarg :long-radius :initarg :radius
                            :accessor long-radius :accessor radius)
               (ellipcity :initarg :ellipcity :initform 1
                          :accessor elipcity)))
#<standard-class ellipse>
cl-user(4): (setq foo (make-instance 'ellipse :long-radius 3))
#<ellipse @ #x216d594a>
cl-user(5): (defclass circle (ellipse) ())
#<standard-class circle>
cl-user(6): (setq bar (make-instance 'circle :radius 3))
#<circle @ #x216e6082>
cl-user(7): (typep bar 'circle)
t
cl-user(8): (typep bar 'ellipse)
t
cl-user(9): (typep foo 'ellipse)
t
cl-user(10): (typep foo 'circle)
nil
cl-user(11): (type-of foo)
ellipse
cl-user(12): (type-of bar)
circle
cl-user(13): (subtypep 'circle 'ellipse)
t
t
cl-user(14): (subtypep 'ellipse 'circle)
nil
t
```

構造体に対するメソッド定義

メソッド定義はクラスに対してだけ有効であるわけではない．構造体定義についてもメソッド定義ができるのだ．14.4節ではincludeオプションを使って人型とビジネスパーソンを次のように定義した．

```
cl-user(16): (defstruct person
               (sex nil)
               (age nil))
person
cl-user(17): (setq foo (make-person :sex 'male :age 33))
#S(person :sex male :age 33)
cl-user(18): (defstruct (business-person (:include person))
               (tel nil)
               (title nil)
               (annual-income nil))
business-person
cl-user(19): (setq bar
                   (make-business-person
                     :sex 'female :age 43
                     :tel '+81-3-123-4567
                     :title 'executive-officer
                     :annual-income 1000000))
#S(business-person :sex female :age 43 :tel +81-3-123-4567 :title executive-officer :annual-
income ...)
```

健常者であれば，人は歩いて移動できる．年収50万ドル以上のビジネスパーソンなら社用飛行機で移動できるとしよう．それを次のようにメソッドで定義する．

```
cl-user(20): (defmethod move-on-foot-p ((person person))
               t)
#<standard-method move-on-foot-p (person)>
cl-user(21): (defmethod move-by-business-jet-p ((biz-person business-person))
               (>= (business-person-annual-income biz-person) 500000))
#<standard-method move-by-business-jet-p (business-person)>
```

すると，人型のインスタンスである foo や bar は歩いて移動できるが，ビジネスパーソンのインスタンスである bar には move-by-business-jet-p が適応できても，そうではない foo に対してはエラーとなる．

```
cl-user(22): (move-on-foot-p foo)
t
cl-user(23): (move-on-foot-p bar)
t
cl-user(24): (move-by-business-jet-p foo)
Error: No methods applicable for generic function #<standard-generic-function move-by-
business-jet-p> with args (#S(person :sex male :age 33)) of
       classes (person)
[condition type: program-error]
cl-user(25): (move-by-business-jet-p bar)
t
```

データ型に対するメソッド定義

そればかりではない．既存のCommon Lisp組み込みのデータ型についてもメソッド定義ができるのだ．

```
cl-user(26): (defmethod this-number-is? ((number fixnum))
               (foramt nil "~D is a fixnum."))
#<standard-method this-number-is? (fixnum)>
```

```
cl-user(27): (defmethod this-number-is? ((number fixnum))
               (format nil "~D is a fixnum." number))
#<standard-method this-number-is? (fixnum)>
cl-user(28): (defmethod this-number-is? ((number integer))
               (format nil "~D is an integer but not a fixnum."
                           number))
#<standard-method this-number-is? (integer)>
cl-user(29): (defmethod this-number-is? ((number single-float))
               (format nil "~D is a single-float." number))
#<standard-method this-number-is? (single-float)>
cl-user(30): (defmethod this-number-is? ((number double-float))
               (format nil "~D is a float but not a single."
                           number))
#<standard-method this-number-is? (double-float)>
cl-user(31): (defmethod this-number-is? ((number number))
               (error "This cannot happen!"))
#<standard-method this-number-is? (number)>
```

ここではすべてのメソッドが同じ総称関数になっているが，データ型には上下関係があることに注意されたい．メソッド内部で call-next-method は呼ばれていないので，下位のデータに対して適応されるメソッドがあれば，その上位データに対するメソッドが適応されることはない．データの上下関係については13.2節を参考にされたい．

```
cl-user(32): (this-number-is? 1.0)
"1.0 is a single-float."
cl-user(33): (this-number-is? 1.0d0)
"1.0d0 is a float but not a single."
cl-user(34): (this-number-is? 1)
"1 is a fixnum."
cl-user(35): (this-number-is? 100000000000000000000)
"100000000000000000000 is an integer but not a fixnum."
```

だから，実は上記の抽象的なnumber型に対して定義したメソッドは，決して起動されることはない．もっとも，ここでは複素数については何も言っていないから，実は複素数については次のようになってしまう．

```
cl-user(36): (this-number-is? #C(1 1))
Error: This cannot happen!
```

15.9 複数引数におけるメソッドディスパッチ

これまで見てきたように，クラスを有する特定化された引数が一つであれば，基本メソッドと:aroundメソッドは特定化引数に適合するメソッドのうち，その特定化のクラスがもっとも特定的なメソッドが実行される．このロジックは複数引数において特定化がある場合でも同様である．以下のいかにもわざとらしい実施例を見られたい．

```
cl-user(2): (defmethod multiple-method-test ((x number) (y number))
              (format nil "x and y are numbers."))
#<standard-method multiple-method-test (number number)>
cl-user(3): (defmethod multiple-method-test ((x integer) (y number))
              (format nil "x is an integer, and y is a number"))
#<standard-method multiple-method-test (integer number)>
cl-user(4): (defmethod multiple-method-test ((x number) (y integer))
```

```
              (format nil "x is a number, and y is an integer."))
#<standard-method multiple-method-test (number integer)>
cl-user(5): (multiple-method-test 1 1)
"x is an integer, and y is a number"
cl-user(6): (defmethod multiple-method-test ((x integer) (y integer))
              (format nil "x and y are integers."))
#<standard-method multiple-method-test (integer integer)>
cl-user(7): (multiple-method-test 1 1)
"x and y are integers."
```

上記2行目の定義よりも，3行目と4行目のほうがより特定的な引数になっているが，3行目と4行目を比較すると実は3行目の方がより特定的な組み合わせになる．なんとなれば複数のメソッドの優先順位は第1引数から優先的に考慮されるからだ．6行目の定義をすると6行目の方が3行目と4行目よりも，特定的なのでそれが適応されるようになる．

15.10 CLOSプログラミングとMOP

これまでのところでは，defclassやdefmethodを使ってどのようにオブジェクト指向プログラミングを実施するかという説明をしてきた．そこでは特にプログラム中でクラス情報を取り出したり，クラス定義やメソッド定義を変更したりということはしていない．そのような機能はC#ではリフレクションと呼ばれているが，オブジェクトのタイプをプログラム中で取り出したり，クラスに関する諸データをプログラムで利用したりすることはCommon Lispにおいてはあたりまえの機能であり，CLOSではプログラマ・インタフェース機能と呼ばれている．CLOSにおけるリフレクションといえば，CLOS自身がCLOSオブジェクトで実装されているが，その実装をユーザが変更するようなことを言う．

MOPとはメタオブジェクト・プロトコルのことであるが，CLOSではクラスそのものもCLOSオブジェクトとなっており，通常のインスタンスオブジェクトに対してクラスオブジェクトのことをメタオブジェクトと呼んでいる．クラスに対するクラスをメタクラスと呼んでいるが，CLOSのクラスとメタクラスにおけるプログラミング（あるいはメッセージ・パッシングの通信）の約束事がメタオブジェクト・プロトコルである．

CLOSプログラマ・インタフェース機能

MOPの説明に入る前に，標準のCLOSプログラマ・インタフェース機能について説明する．CLOSメタオブジェクトの構造をCLOSで書くとすると，それは次のようなものである．

```
(defclass standard-class ()
  ((name :initarg :name
         :accessor class-name)
   (direct-superclasses :initarg :direct-superclasses
                        :accessor mop:class-direct-superclasses)
   (direct-slots :accessor mop:class-direct-slots)
   (class-precedence-list :accessor mop:class-precedence-list)
   (effective-slots :accessor mop:class-slots)
   (direct-subclasses :initform ()
                      :accessor mop:class-direct-subclasses)
```

```
    (direct-methods :initform ()
                    :accessor mop:class-direct-methods)))
```

すなわち，standard-class をメタクラスとするすべての標準的クラスは，名前(name)，局所上位クラス(direct-superclasses)，局所スロット(direct-slots)，クラス優先順位リスト(class-precedence-list)，実効スロット(effective-slots)，局所下位リスト(direct-subclasses)，および局所メソッド(direct-methods)を有している．defclassの宣言において定義された各クラスの局所優先順位リストの内容が上記局所上位クラスとなり，同様に宣言中のスロット定義が少しだけ変形されて，上記direct-slotsの内容になる．これらのクラス定義におけるスロット定義によって，クラスのインスタンスが持つべきスロットが決定される．15.5節で述べたように，該当クラスの局所上位クラスをたどって，すべての上位クラスにおけるクラスの上位下位関係が該当クラスごとにトポロジカル・ソーティングのアルゴリズムにより計算されて，その結果がクラス優先順位リスト(class-precedence-list)の内容となる．一方，該当クラスも含めてすべての上位クラスにあるスロットが合計されて，実行スロット(effective-slots)の情報となる．

標準的なCLOSクラス定義として，クラスには名前がある．多くのプログラマ・インタフェースにおいて，CLOSメタオブジェクトの代理として名前を置くことができるが，あるクラスメタオブジェクトの名前は上記のようにアクセッサclass-nameで取り出すことができる．また，これまで何回も出てきたように，システムに存在するクラスはその名前を使って関数 find-class で発見することができる．

```
cl-user(2): (find-class 'standard-object)
#<standard-class standard-object>
cl-user(3): (class-name (find-class 'standard-object))
standard-object
```

このように，通常ではfind-classとclass-nameで名前は循環的な呼び出しで一致するが，そのような名前は正規の名前(proper name)と呼ばれている．CLOSクラスのそのほかの情報も，次のようにそれぞれのアクセッサを用いて取りだすことができる．

```
cl-user(6): (mop:class-direct-superclasses
              (find-class 'standard-object))
(#<built-in-class t>)
cl-user(7): (mop:class-direct-slots
              (find-class 'standard-object))
nil
cl-user(8): (mop:class-precedence-list
              (find-class 'standard-object))
Error: Class #<standard-class standard-object> is not yet finalized.
[condition type: program-error]
cl-user(9): (mop:class-direct-subclasses
              (find-class 'standard-object))
(#<standard-class common-graphics-user::test-person> #<standard-class ide.trace::profile-
tree> #<standard-class ide.trace::profile-item> #<standard-class cg.base:project> #<standard-
class cg.base:module> #<standard-class ide.patch-dialog::patch> #<standard-class
ide.inspector::inspector-grid-info> #<standard-class ide.inspector::inspector-history-info> #
<standard-class ide.inspector::inspector> #<standard-class ide.inspector::process-bindings>
...)
```

すなわち，クラスに定義してある局所上位クラス(direct super class)はmop:class-direct-superclasses(SBCLではsb-mop:class-direct-superclasses)で取り出すことができるし，局所下位クラスはmop:class-direct-subclasses(SBCLではsb-mop:class-direct-subclasses)で取り出すことができる．

これらのプログラマ・インターフェースを用いて，CLOSの型階層を印刷する関数は次のように定義できる．このコードはRiesbeckとSchankによる*Inside Case-Based Reasoning* に記載のアルゴリズムをCLOS用に書き換えたものである．

```
cl-user(2): (defun dah (cls)
  "dah 《cls》
   prints all the specializations under 《cls》.
   The name is short for
   'display abstraction hierarchy'"
  (pprint (tree->list cls #'specs->list nil)))
dah
cl-user(3): (defun tree->list (cls fn visited)
  "tree->list 《cls》《function》《cls-list》
   returns a list starting with 《cls》, followed by the elements
   of the list returned by calling 《function》with 《cls-list》
   updated to include 《cls》. If 《cls》is already in 《cls-list》,
   just a list with 《cls》is returned."
  (cond ((member cls visited) (list (class-name cls)))
        (t (push cls visited)
           `(,(class-name cls) ,@(funcall fn cls visited)))))
tree->list
cl-user(4): (defun specs->list (cls visited)
  "specs->list 《cls》《cls-list》
   returns a list starting with 《cls》, followed by the
   specialization tree under each specializations of 《cls》."
  (loop for spec in (mop:class-direct-subclasses cls)
        when (tree->list spec #'specs->list visited)
        collect it))
specs->list
```

Common Lispのデータ型のいくつかにはそれに対応するCLOSオブジェクトが定義してある．Common Lispの型階層の最上位には t が位置しているが，データ型 t に対応するCLOSオブジェクトは `(find-class 't)` で得ることができる．それに上記関数 dah を適応すると ACL Free Express 10.0 の mlisp の場合には以下のような結果が得られる．

```
cl-user(5): (dah (find-class 't))

(t (agraph-upi) (agraph-triple) (excl::instance1) (excl::instance)
 (hash-table) (readtable)
 (function
  (aclmop:funcallable-standard-object
    (generic-function
     (excl::basic-constructor (excl::auto-constructor)
      (excl::constructor))
     (standard-generic-function)))
 (symbol (null)) (character) (array (vector (bit-vector) (string)))
 (sequence (vector (bit-vector) (string)) (list (null) (cons)))
 (number
  (real (rational (ratio) (integer (fixnum)))
```

```
  (float (double-float) (single-float)))
 (complex))
(structure-object (excl::loop-universe) (acl-socket::ipv6-address)
 (inspect::inspect-level) (inspect::field-def)
 (profiler::woven-stream) (profiler::aggregate) (profiler::node)
 (profiler::current-profile) (excl.scm::eval-info)
 (foreign-functions::sized-ftype
  (foreign-functions::sized-ftype-complex
   (foreign-functions::sized-ftype-bits)
   (foreign-functions::sized-ftype-array)
   (foreign-functions::sized-ftype-struct))
  (foreign-functions::sized-ftype-ref)
  (foreign-functions::sized-ftype-function)
  (foreign-functions::sized-ftype-prim))
 (foreign-functions::iforeign-type) (debugger::environment)
 (compiler::census-point) (compiler::census) (compiler::bb)
 (compiler::creg) (compiler::treg) (compiler::quad)
 (compiler::reg-struct) (compiler::source-struct)
 (compiler::source-label) (compiler::source-ref) (compiler::faslobj)
 (compiler::lap) (compiler::varrec) (compiler::keyrec)
 (compiler::optrec) (compiler::closure-vec)
 (compiler::node (compiler::xthrow) (compiler::tag)
  (compiler::xtagbody) (compiler::symeval) (compiler::simple-case)
  (compiler::xsetq) (compiler::rfrom) (compiler::xprogv)
  (compiler::xprogn) (compiler::mvprog1) (compiler::mvc)
  (compiler::xlet) (compiler::xgo) (compiler::constant)
  (compiler::xcond) (compiler::xcatch) (compiler::call) (compiler-let)
  (compiler::xblock) (compiler::xlambda))
 (excl::entry) (excl::pprint-dispatch-struct) (excl::anno)
 (excl::xpstruct) (system::join-data)
 (system::chain (system::clock-event) (system::thread))
 (excl::source-context) (restart) (excl::census-point) (excl::shlib)
 (excl::non-lisp-value) (excl::frame-descriptor) (system::cl-option)
 (external-format) (excl::composing-functions) (excl::prim-ftype)
 (package) (excl::cached-translation-info)
 (pathname (logical-pathname))
 (system::augmentable-evaluation-environment)
 (system::augmentable-compilation-environment)
 (system::augmentable-compiler-environment)
 (system::augmentable-environment)
 (system::augmentable-environment-base) (system::ha$h-table)
 (excl::lockable-structure-base
  (synchronizing-structure (multiprocessing:process-lock)
   (basic-lock))
  (excl::lockable-structure (random-state) (excl::basic-spin-lock)))
 (random-state) (package) (pathname (logical-pathname))
 (logical-pathname))
(standard-object (net.uri:uri (net.uri:urn))
 (acl-socket::pf-datagram
  (acl-socket::pf-datagram-internet
   (acl-socket:socket-datagram-internet-active))
  (acl-socket::pf-datagram-file))
 (acl-socket::pf-stream
  (acl-socket::hiper-socket-stream-internet-active)
  (acl-socket:socket-stream-internet-passive)
  (acl-socket:socket-stream-internet-active)
  (acl-socket::pf-passive-stream-file)
  (acl-socket::pf-passive-stream-internet
   (acl-socket:socket-stream-internet-passive)))
```

```
(acl-socket::con-passive (acl-socket:socket-datagram-internet-active)
 (acl-socket:socket-stream-internet-passive)
 (acl-socket::pf-passive-stream-file)
 (acl-socket::pf-passive-stream-internet
  (acl-socket:socket-stream-internet-passive)))
(acl-socket::con-active
 (acl-socket::hiper-socket-stream-internet-active)
 (acl-socket:socket-stream-internet-active))
(acl-socket::af-file (acl-socket::pf-datagram-file)
 (acl-socket::pf-passive-stream-file))
(acl-socket::af-internet
 (acl-socket::hiper-socket-stream-internet-active)
 (acl-socket:socket-datagram-internet-active)
 (acl-socket:socket-stream-internet-passive)
 (acl-socket:socket-stream-internet-active)
 (acl-socket::pf-datagram-internet
  (acl-socket:socket-datagram-internet-active))
 (acl-socket::pf-passive-stream-internet
  (acl-socket:socket-stream-internet-passive)))
(excl::default-latin1-base-ef
 (acl-socket:socket (acl-socket::hiper-socket-stream-internet-active)
  (acl-socket:socket-datagram-internet-active)
  (acl-socket:socket-stream-internet-passive)
  (acl-socket:socket-stream-internet-active)))
(profiler::generic-profile (profiler::count-only-profile)
 (profiler::space-profile) (profiler::real-time-profile)
 (profiler::time-profile))
(cross-reference::basic-relation (cross-reference::complex-relation)
 (cross-reference::simple-relation
  (cross-reference::exhaustive-relation)))
(lep::background-stream-error-handling
 (lep::basic-evaluation-request (lep::compile/load-file-request)
  (lep::evaluation-request))
 (excl.scm::list-changed-definitions))
(excl.scm::basic-source-part (excl.scm::source-part)
 (excl.scm::basic-definition-group
  (excl.scm::string-definition-group)
  (excl.scm::file-definition-group)))
(lep:basic-session
 (lep::basic-query (lep::completing-read)
  (lep:simple-query (excl.scm::signal-transaction-file-error)
   (excl.scm::get-buffer-status) (lep::eval-in-emacs-query)
   (lep::buffer-region-query) (lep::show-clman-entry)
   (lep::create-listener-stream) (lep:display-string-somewhere)
   (lep::output-to-hidden-buffer) (lep:display-string-in-buffer)
   (lep::prompt-in-editor) (lep:find-file-in-editor)))
 (lep:compute-reply-session
  (excl.scm::find-buffer-definition-session)
  (excl.scm::file-definitions-session)
  (lep::basic-evaluation-request (lep::compile/load-file-request)
   (lep::evaluation-request))
  (excl.scm::list-changed-definitions) (lep::trace-definition-reply)
  (lep::gdb-session) (lep::eval-from-emacs-session)
  (lep::list-all-processes-session) (lep::local-session)
  (lep::disassemble-session) (lep::pprint-frame-session)
  (lep::tpl-command-session) (lep::zoom-session) (lep::toggle-trace)
  (lep::undefine-reply) (lep::bug-report-session)
  (lep::list-unused-functions-session)
  (lep::list-undefined-functions-session) (lep::list-fspecs-session)
```

```
  (lep::describe-something-session) (lep::macroexpand-session)
  (lep::function-documentation-session)
  (lep::disassemble-fspec-session) (lep::arglist-session)
  (lep::apropos-session)
  (lep:iterative-session (excl.scm::edit-sequence-session)
   (excl.scm::metadot-session))
  (lep::null-package-reply-session
   (lep::list-all-completions-session))))
(lep::editor-connection (lep::lemacs-editor-connection)
 (lep::epoch-editor-connection))
(foreign-functions::foreign-object)
(foreign-functions:foreign-pointer)
(defsystem::dependency-graph
 (defsystem::module
  (defsystem:default-module
   (defsystem::foreign-module (defsystem:fortran-module)
    (defsystem:c-module))
   (defsystem:text-module) (defsystem:lisp-module)))
 (defsystem::module-group (defsystem:default-module-group))
 (defsystem::system (defsystem:default-system)))
(defsystem::defsystem-base-class
 (defsystem:module-container (defsystem:default-module-group)
  (defsystem:default-system))
 (defsystem::module
  (defsystem:default-module
   (defsystem::foreign-module (defsystem:fortran-module)
    (defsystem:c-module))
   (defsystem:text-module) (defsystem:lisp-module)))
 (defsystem::module-group (defsystem:default-module-group))
 (defsystem::system (defsystem:default-system)))
(multiprocessing:queue) (excl::locale) (excl::encrypted-string)
(lockable-object (multiprocessing:process))
(condition (excl::compiler-note) (compiler::ics-dependent-passc)
 (compiler::ics-dependent-passa)
 (excl::utf8-generating-surrogate-pair) (socket-chunking-end-of-file)
 (excl::advance-warning) (excl::input-edit)
 (excl::non-error-operating-system-signal
  (excl::non-error-asynchronous-operating-system-signal
   (excl::non-erroring-interrupt-signal)))
 (serious-condition (storage-condition)
  (error (multiprocessing::process-join-error)
         (excl::license-support-expired)
         (excl::locale-missing-category) (void-external-format)
         (fasl-casemode-mismatch)
         (type-error (case-failure) (simple-type-error))
         (parse-error (reader-error (excl::extra-right-paren-error)))
         (stream-error (errno-stream-error (socket-error))
          (stream-closed-error) (end-of-file)
          (reader-error (excl::extra-right-paren-error)))
         (package-error (package-locked-error)) (winapi-error)
         (syscall-error
          (file-error (file-incompatible-fasl-error)
           (file-does-not-exist-error
            (autoload-file-does-not-exist-error))))
         (cell-error (undefined-function) (unbound-variable)
          (unbound-slot))
         (print-not-readable)
         (arithmetic-error (floating-point-inexact)
          (floating-point-invalid-operation)
```

```
          (floating-point-underflow) (floating-point-overflow)
          (division-by-zero))
         (purespace-write-error)
         (operating-system-signal
          (synchronous-operating-system-signal)
          (asynchronous-operating-system-signal (interrupt-signal)))
         (no-external-format-error) (compiler-not-available-error)
         (program-error) (control-error)
         (simple-error
          (lep:lep-error (lep:lep-operation-aborted)
           (lep:no-lep-connection)))))
 (break-condition (simple-break))
 (warning
  (foreign-functions::strings-convert-warning
    (foreign-functions::strings-convert-rt-warning)
    (foreign-functions:strings-convert-def-warning
     (foreign-functions::strings-convert-noclosure-warning)
     (foreign-functions::strings-convert-noconv-warning)))
  (compiler::dynamic-extent-argument-specification-change)
  (incompatible-conformance-change-warning)
  (excl::ef-precomp-mismatch-warning)
  (excl::foreign-strings-convert-warning)
  (compiler-not-available-warning) (simple-warning)
  (style-warning (compiler-undefined-functions-called-warning)
   (compiler-unreachable-code-warning)
   (compiler-no-in-package-warning)
   (compiler-inconsistent-type-declaration-warning)
   (compiler-inconsistent-name-usage-warning)
   (compiler-free-reference-warning)
   (ineffective-declaration-warning
    (excl::float-declaration-used-warning))))
 (simple-condition (simple-type-error)
  (simple-error
   (lep:lep-error (lep:lep-operation-aborted)
    (lep:no-lep-connection)))
  (simple-break) (simple-warning)))
(stream
 (excl::line-buffered-mixin (excl::line-buffered-terminal-stream)
  (excl::line-buffered-file-stream))
 (simple-stream
  (string-simple-stream
   (string-output-simple-stream
    (bidirectional-character-encapsulating-stream)
    (annotation-output-simple-stream) (xp-simple-stream)
    (excl::limited-string-output-simple-stream)
    (fill-pointer-output-simple-stream))
   (string-input-simple-stream
    (bidirectional-character-encapsulating-stream)
    (excl::ring-input-simple-stream))
   (composing-stream))
  (dual-channel-simple-stream
   (excl::hiper-socket-stream
    (acl-socket::hiper-socket-stream-internet-active))
   (socket-base-simple-stream)
   (socket-simple-stream (acl-socket:socket-stream-internet-active))
   (terminal-simple-stream
    (excl::lazy-encapsulating-stream (excl::background-stream))
    (excl::line-buffered-terminal-stream)))
  (single-channel-simple-stream
```

```
   (file-simple-stream (excl::line-buffered-file-stream))
   (null-simple-stream)
   (direct-simple-stream (buffer-output-simple-stream)
    (buffer-input-simple-stream)))
  (probe-simple-stream))
 (string-stream
  (string-simple-stream
   (string-output-simple-stream
    (bidirectional-character-encapsulating-stream)
    (annotation-output-simple-stream) (xp-simple-stream)
    (excl::limited-string-output-simple-stream)
    (fill-pointer-output-simple-stream))
   (string-input-simple-stream
    (bidirectional-character-encapsulating-stream)
    (excl::ring-input-simple-stream))
   (composing-stream)))
 (file-stream (file-simple-stream (excl::line-buffered-file-stream))
  (probe-simple-stream)))
(excl::plist-mixin
 (standard-method
  (aclmop:standard-accessor-method (aclmop:standard-writer-method)
   (aclmop:standard-reader-method)))
 (excl::documentation-mixin
  (class (foreign-functions::foreign-structure)
   (excl::clos-class (structure-class)
    (excl::std-class (standard-class)
     (aclmop:funcallable-standard-class))
    (excl::nonstd-class (aclmop:forward-referenced-class)
     (built-in-class)))))
 (excl::dependee-mixin
  (generic-function
   (excl::basic-constructor (excl::auto-constructor)
    (excl::constructor))
   (standard-generic-function))
  (class (foreign-functions::foreign-structure)
   (excl::clos-class (structure-class)
    (excl::std-class (standard-class)
     (aclmop:funcallable-standard-class))
    (excl::nonstd-class (aclmop:forward-referenced-class)
     (built-in-class))))))
(aclmop:metaobject
 (aclmop:specializer (aclmop:eql-specializer)
  (class (foreign-functions::foreign-structure)
   (excl::clos-class (structure-class)
    (excl::std-class (standard-class)
     (aclmop:funcallable-standard-class))
    (excl::nonstd-class (aclmop:forward-referenced-class)
     (built-in-class)))))
 (aclmop:slot-definition
  (excl::structure-slot-definition
   (excl::structure-effective-slot-definition)
   (excl::structure-direct-slot-definition))
  (aclmop:standard-slot-definition
   (aclmop:standard-effective-slot-definition)
   (aclmop:standard-direct-slot-definition))
  (aclmop:effective-slot-definition
   (excl::structure-effective-slot-definition)
   (aclmop:standard-effective-slot-definition))
  (aclmop:direct-slot-definition
```

```
    (excl::structure-direct-slot-definition)
    (aclmop:standard-direct-slot-definition)))
  (method-combination
   (excl::standard-method-combination (excl::short-method-combination)
    (excl::long-method-combination)))
  (method
   (standard-method
    (aclmop:standard-accessor-method (aclmop:standard-writer-method)
     (aclmop:standard-reader-method))))
  (generic-function
   (excl::basic-constructor (excl::auto-constructor)
    (excl::constructor))
   (standard-generic-function)))
 (aclmop:funcallable-standard-object
  (generic-function
   (excl::basic-constructor (excl::auto-constructor)
    (excl::constructor))
   (standard-generic-function)))
 (excl::class-lock-mixin
  (aclmop:specializer (aclmop:eql-specializer)
   (class (foreign-functions::foreign-structure)
    (excl::clos-class (structure-class)
     (excl::std-class (standard-class)
      (aclmop:funcallable-standard-class))
     (excl::nonstd-class (aclmop:forward-referenced-class)
      (built-in-class)))))
  (excl::plist-mixin
   (standard-method
    (aclmop:standard-accessor-method (aclmop:standard-writer-method)
     (aclmop:standard-reader-method)))
   (excl::documentation-mixin
    (class (foreign-functions::foreign-structure)
     (excl::clos-class (structure-class)
      (excl::std-class (standard-class)
       (aclmop:funcallable-standard-class))
      (excl::nonstd-class (aclmop:forward-referenced-class)
       (built-in-class)))))
    (excl::dependee-mixin
     (generic-function
      (excl::basic-constructor (excl::auto-constructor)
       (excl::constructor))
      (standard-generic-function))
     (class (foreign-functions::foreign-structure)
      (excl::clos-class (structure-class)
       (excl::std-class (standard-class)
        (aclmop:funcallable-standard-class))
       (excl::nonstd-class (aclmop:forward-referenced-class)
        (built-in-class))))))
  (method-combination
   (excl::standard-method-combination (excl::short-method-combination)
    (excl::long-method-combination))))))
```

メタクラスとは？

これまで出現した例題として出現したクラスは，すべて個々のクラスの実現体（インスタンス）を作るためのものであった．たとえば，円や楕円の実現体を作るためにそのクラスを定義して，実現体をmake-

instanceで作成した．これと同様の関係をクラスとメタクラスの間で持つことができる．すなわち，クラスをその実現体とするクラスがメタクラスと呼ばれるものである．

メタクラス・プログラミング

defclassを用いたクラス定義において，上位クラスを省略した時にはstandard-objectがその既定値となる．また，そのときメタクラスを指定しなければstandard-classが既定値として定義されるクラスのメタクラスとなる．

```
cl-user(2): (defclass foo () ())
#<standard-class foo>
cl-user(3): (subtypep (find-class 'foo) 'standard-object)
t
t
cl-user(4): (class-of (find-class 'foo))
#<standard-class standard-class>
```

メタクラスを使ったプログラミングにおいてはstandard-classの下位クラスとしてメタクラスを定義し，その自分用メタクラスをメタクラスとして自分用クラスを定義する．

```
cl-user(6): (defclass my-own-standard-class (standard-class) ())
#<standard-class my-own-standard-class>
cl-user(7): (defclass my-own-object-class () ()
              (:metaclass my-own-standard-class))
#<my-own-standard-class my-own-object-class>
```

このようにしてできた自分用メタクラスはstandard-classの属性とメソッドをすべて継承しているし，自分用クラスはstandard-objectの属性とメソッドをすべて継承している．ここで自分用のメタクラスとクラスを利用して，標準のCLOS機能では無い機能を追加したり，標準のCLOSメソッドをオーバライドして変更したりすることができる．メタクラスにある機能とは，実はクラスの生成と制御に関するものであるが，アドバンストな議論となるのでその実例紹介は省略する．

リフレクションとは？

一般にリフレクションとは，その言語の範囲を離れることなくその言語の実装を変更できることを言う．CLOSでは上で述べたように，メタオブジェクト・プロトコルがきっちり決められていて，しかもその変更手段までプログラマーに公開されている．プログラマーはメタオブジェクト・プロトコルに従いつつ，それをCLOSの枠組みで変更することができる．

standard-classのメタ巡回（circularity)と解釈可能クラス階層

上記の実施例4でも分かるように，standard-classのクラスは再びstandard-classである．すなわち，standard-classに関してはクラス・インスタンス関係が止まることなく巡回している．これを**メタ巡回**という．もし，ボトムの実現体をオーダ0とし，オーダ0のオブジェクトのクラスをオーダ1とし，同様にオーダnのオブジェクトのクラスをオーダn+1とすると，standard-classのオーダはn=2から無限大まで伸びること

になる．これを**standard-classにおける無限メタ巡回**と呼ぼう．さらに，上位下位関係にあるクラスのオーダは等しいとしよう．これを**解釈可能なクラス階層**と呼ぶことにする．すると，standard-classを上位クラスとするメタクラスも，必要に応じてオーダはn=2から無限大までありえることになる．

standard-objectとstandard-classの奇妙な関係

一方，standard-objectのクラス（メタクラス）は他のクラスと同様にstandard-classであるが，standard-classの上位クラス（スーパークラス）はstandard-objectである．これを**standard-objectとstandard-classのねじれた関係**と呼ぼう．この特殊な関係があるために，結果的にstandard-objectはstandard-objectのインスタンスということにもなる．確かに以下に示す通りであるが，その理由は分かるだろうか．

```
cl-user(8): (typep (find-class 'standard-class) 'standard-class)
t
cl-user(9): (typep (find-class 'standard-object) 'standard-class)
t
cl-user(10): (subtypep (find-class 'standard-class)
                       (find-class 'standard-object))
t
t
cl-user(11): (typep (find-class 'standard-object) 'standard-object)
t
```

このねじれた関係のすごいところは，結果的にすべてのクラスがstandard-classとstandard-objectの実現体になってしまうところだ．

```
cl-user(12): (typep (find-class 'my-own-object-class)
                    'standard-class)
t
cl-user(13): (typep (find-class 'my-own-object-class)
                    'standard-object)
t
cl-user(14): (typep (find-class 'my-own-standard-class)
                    'standard-class)
t
cl-user(15): (typep (find-class 'my-own-standard-class)
                    'standard-object)
t
```

すなわち，オーダ0の実現体のみならず，n=1から無限大までのすべてのクラスがstandard-objectの属性とメソッドを継承することになる．

第16章 Format の呪文

Common Lisp のフォーマット文はもう一つの言語と言われるほど，Lispとは異なる文法で豊富な機能を持っていて，それだけに使いこなすのは大変である．網羅的に記述しようとすると，分量も多くなるので，ここでは実際に使うときに有用なものとか，はまりやすいポイントを中心的に取り上げて解説する．

16.1 format 構文

format 文の呼び出し形式は次のようなものである．

```
(format《出力ストリーム》《制御文字列》{《出力項目》}* )
```

《出力ストリーム》は nil あるいは t あるいは第19章で説明する出力ストリームである．《出力ストリーム》が nil の場合には，format の結果はどこにも出力されずに，format 文の返値となる．t の場合には標準出力ストリーム(19.3節で説明する*standard-output*)に出力されるが，何もしていなければそれは通常ターミナルウィンドウやコンソールである．《出力ストリーム》に出力ストリームが与えられると format 文の結果はその出力ストリームに出力される．《出力ストリーム》が nil 以外の場合には，format 文の返値は nil であることに注意されたい．

出力したい内容が固定的なら，《出力項目》なしで，《制御文字列》にその固定文を書けばよい．

```
cl-user(2): (format t "Hello, World!")
Hello, World!
nil
```

《出力項目》がある場合には，それに対応する制御文字を制御文字列の中に書く．対応はそれぞれの先頭から順に対応が取られる．

```
cl-user(3): (setq state 'Wonderful)
Wonderful
cl-user(4): (format t "Hello, ~S World!" state)
Hello, Wonderful World!
nil
```

制御文字「~S」(チルダ（U+007E）と大文字 S または小文字 s)は，REPL におけるオブジェクトの印字表現をそのまま出力する．だから，シンボルや数字なら便利だが，文字列は文字列としてダブルクォーテーション付きになる．もしそうしたくなれければ，文字列に対しては「~A」を使えばよい．

```
cl-user(5): (format t "Hello, ~S World!" (string state))
Hello, "Wonderful" World!
nil
cl-user(6): (format t "Hello, ~A World!" (string state))
Hello, Wonderful World!
nil
cl-user(7): (format t "Hello, ~S World!" *package*)
Hello, #<The common-lisp-user package> World!
nil
```

```
cl-user(8): (format t "This is PI, ~S." pi)
This is PI, 3.141592653589793d0.
nil
```

文字列に対して「~S」と「~A」の効果は同じではない．特にその文字列がダブルクォーテーションを含むような場合には，それはエスケープされなければならないが，「~S」ではエスケープ文字が印刷されるのに対して「~A」は印刷されない．やはり「~A」は文字列向きの印刷用になっているのだ．

```
cl-user(9): (setq foo
                  "So you were \"accidentally\" at the restaurant with her?")
"So you were \"accidentally\" at the restaurant with her?"
cl-user(10): (format t "~S" foo)
"So you were \"accidentally\" at the restaurant with her?"
nil
cl-user(11): (format t "~A" foo)
So you were "accidentally" at the restaurant with her?
nil
```

16.2 文字列出力制御

「~A」に色々な修飾をして，出力を整形することができる．固定カラム数の中に左詰めあるいは右詰めに印刷するには，「~」と「A」の間に，カラム数「n」あるいは「n@」を挿入する．文字数が溢れたときにはそのまま印刷される．

```
cl-user(1): (setq mw "Supercalifragilisticexpialidocious")
"Supercalifragilisticexpialidocious"
cl-user(2): (length mw)
34
cl-user(3): (format nil "~40A" mw)
"Supercalifragilisticexpialidocious      "
cl-user(4): (format nil "~40@A" mw)
"      Supercalifragilisticexpialidocious"
cl-user(5): (format nil "~30A" mw)
"Supercalifragilisticexpialidocious"
```

詰め物にされる文字を指定することもできる．制御文字列における**文字指定にはクォート文字**が用いられる．

```
cl-user(6): (format nil "~40,,,'_A" mw)
"Supercalifragilisticexpialidocious______"
```

たとえカラム数が足りない場合でも詰め物の文字を最低数つけたい場合には，次のように指定できる．

```
cl-user(7): (format nil "~20,,3,'_A" mw)
"Supercalifragilisticexpialidocious___"
```

さらに，指定したカラム数を越すまで，ある個数単位(以下では10個単位)で詰め文字を詰めたい場合には，このようにする．

```
cl-user(15): (format nil "~40,10,3,'_A" mw)
"Supercalifragilisticexpialidocious__________"
cl-user(16): (length *)
47
```

```
cl-user(17): (format nil "~40,,3,'_A" mw)
"Supercalifragilisticexpialidocious______"
cl-user(18): (length *)
40
```

大文字小文字変換

「~@(《文字列》~)」は文字列の最初だけを大文字にする．

```
cl-user(10): (setq foo '(this is a camel case))
(this is a camel case)
cl-user(11): (format nil "~@(~A~)~@(~A~)~@(~A~)~@(~A~)~@(~A~) ~A ~A ~A ~A."
               (first foo) (second foo) (third foo) (fourth foo)
               (fifth foo) (cadr foo) (caddr foo) (cadddr foo)
               (nth 4 foo))
"ThisIsACamelCase is a camel case."
```

「~:(《文字列》~)」は文字列中のすべての単語について最初の文字を大文字にする．「~:@(《文字列》~)」は文字列中のすべての小文字を大文字にする．「~(《文字列》~)」は文字列中のすべての大文字を小文字にする．

```
cl-user(17): (format nil "~:(china japan korea vieatnum~)")
"China Japan Korea Vieatnum"
cl-user(18): (format nil "~:@(china japan korea vieatnum~)")
"CHINA JAPAN KOREA VIEATNUM"
cl-user(19): (format nil "~(CHINA JAPAN KOREA VIEATNUM~)")
"china japan korea vieatnum"
```

16.3 数出力制御

整数の出力にはその出力形式によって「~D」(10進表示)，「~X」(16進表示)，「~O」(8進表示)，「~B」(2進表示)を用いる．

```
cl-user(25): (setq ip-address 3232235777)
3232235777
cl-user(26): (format nil "~D" ip-address)
"3232235777"
cl-user(27): (format nil "~X" ip-address)
"c0a80101"
cl-user(28): (format nil "~O" ip-address)
"30052000401"
cl-user(29): (format nil "~B" ip-address)
"11000000101010000000000100000001"
```

強制的に符号も付ける場合には「@」修飾子をつける．カラム数に対して数字を右詰めにして詰め文字を指定するには，次のようにすればよい．なお，詰め文字の既定値は空白スペースである．

```
cl-user(39): (format nil "~10,'_@D" 12345)
"____+12345"
```

三桁ごとにカンマを印刷したければ，コロンで修飾する．

```
cl-user(40): (format nil "~10,'_:@D" 12345)
"___+12,345"
```

「~D」以外の「~X」，「~O」，「~B」では，三桁ごとではなく，任意の桁ごとに任意の文字を挿入できる．

```
cl-user(60): (format nil "~,,'.,2:X" ip-address)
"c0.a8.01.01"
```

特殊な数字の出力方法として，「~R」がある．これを見れば，Common Lisp の format はそこまでするのかと思うはずだ．

```
cl-user(70): (format nil "~R" 1234567890)
"one billion two hundred thirty-four million five hundred sixty-seven thousand eight hundred
ninety"
cl-user(71): (format nil "~@R" 1234567890)
"1234567890"
cl-user(72): (format nil "~@R" 1)
"I"
cl-user(73): (format nil "~@R" 2)
"II"
cl-user(74): (format nil "~@R" 3)
"III"
cl-user(75): (format nil "~@R" 4)
"IV"
cl-user(76): (format nil "~@R" 5)
"V"
cl-user(77): (format nil "~@R" 10)
"X"
cl-user(78): (format nil "~@R" 11)
"XI"
cl-user(79): (format nil "~@R" 12)
"XII"
```

コロンで修飾すると印刷されるのは序数になる．

```
cl-user(80): (format nil "This chapter is ~:R Chapter." 16)
"This chapter is sixteenth Chapter."
```

浮動小数点数の出力には「~F」と「~E」が使われる．

```
cl-user(81): (format nil "~F" 1234.0)
"1234.0"
cl-user(82): (format nil "~F" 1234.5678)
"1234.5677"
cl-user(83): (format nil "~F" pi)
"3.141592653589793d0"
cl-user(84): (format nil "~E" 1234.0)
"1.23399995e+3"
cl-user(85): (format nil "~E" 1234.5678)
"1.23456776e+3"
cl-user(86): (format nil "~E" pi)
"3.141592653589793d+0"
```

浮動小数点数に対する「~G」制御は，見た目に楽なように「適当に」固定小数点表現や，浮動小数点表現で印刷してくれる．

```
cl-user(12): (format nil "~G" 0.1234)
"0.1234    "
cl-user(13): (format nil "~G" 0.000001234)
"1.233999950e-6"
cl-user(14): (format nil "~G" 1234000000000000000000000000.0)
"1234000000000000000000000000.    "
```

カラム数の指定や印刷桁数の指定など，およそ必要なものはそのような出力を印刷することができるが，ここでは省略する．詳しくは *CLtL2* 22.3.3 を参照されたい．

英語の複数形

数出力制御の最後に英語の複数形についての出力制御を説明する．「~P」は英語名詞の語幹に対して複数形を作るのに便利な機能である．

```
cl-user(29): (setq foods
                   '(apple pear plum strawberr orange))
(apple pear plum strawberr orange)
cl-user(30): (loop for iii from 1 to 5
                   for food in foods
                 do (if (eql food 'strawberr)
                        (format t "~%He ate ~R ~A~@P at ~:R day."
                                  iii food iii iii)
                      (format t "~%He ate ~R ~A~P at ~:R day."
                                iii food iii iii)))

He ate one apple at first day.
He ate two pears at second day.
He ate three plums at third day.
He ate four strawberries at fourth day.
He ate five oranges at fifth day.
nil
```

「~P」は数字 1 のときには何もしないが，それ以外では"s"を出力する．「~@P」は数字 1 のときには何もしないが，それ以外では"ies"を出力する．いちごが"strawberr"になっていることに注意されたい．残念ながら不規則な複数形についての機能はない．

上記例では "one apple" や "two pears" を出力するのに「~R ~A~P」とコード化して「iii food iii」を当てた．これを「iii food」としたい．「~:P」は直前の引数を利用して「~P」と同じことをする．

```
cl-user(45): (format nil "~R apple~:P" 1)
"one apple"
cl-user(46): (format nil "~R pear~:P" 2)
"two pears"
```

ところが，制御文字列に固定的な文字として英語の名詞が書かれなければならないのにもかわらず，今ここでは apple も pear も引数になっている．そこで，format制御文字を出力するformat文を書いて，この食べ物を場合に応じて正しく出力したものを制御文字とするようなformat文を作って，その出力を制御文として利用することとする．

```
cl-user(42): (loop for iii from 1 to 5
                   for food in foods
                do (format t
                     (if (eql food 'strawberr)
                         (format nil
                                 "~~%He ate ~~R ~A~~:@P at ~~:R day."
                                 food)
                       (format nil
                               "~~%He ate ~~R ~A~~:P at ~~:R day."
                               food))
                     iii iii))

He ate one apple at first day.
He ate two pears at second day.
He ate three plums at third day.
He ate four strawberries at fourth day.
He ate five oranges at fifth day.
nil
```

二重のチルダを持つ「~~《制御文字》」は最初の format 文では制御文字とは処理されずに「~《制御文字》」と変わるだけである．だから，最初の format 文で出力されるものは，次のようなものである．

```
cl-user(49): (loop for food in foods
                collect (if (eql food 'strawberr)
                            (format nil
                                    "~~%He ate ~~R ~A~~:@P at ~~:R day."
                                    food)
                          (format nil
                                  "~~%He ate ~~R ~A~~:P at ~~:R day."
                                  food)))
("~%He ate ~R apple~:P at ~:R day."
 "~%He ate ~R pear~:P at ~:R day."
 "~%He ate ~R plum~:P at ~:R day."
 "~%He ate ~R strawberr~:@P at ~:R day."
 "~%He ate ~R orange~:P at ~:R day.")
```

ついでに，上記42行目実施例で，同じ数字を二つ与えているのも煩わしい．これを一つにしてみよう．

```
cl-user(50): (loop for iii from 1 to 5
                   for food in foods
                do (format t
                     (if (eql food 'strawberr)
                         (format nil
                                 "~~%He ate ~~R ~A~~:@P at ~~:*~~:R day."
                                 food)
                       (format nil
                               "~~%He ate ~~R ~A~~:P at ~~:*~~:R day."
                               food))
                     iii))

He ate one apple at first day.
He ate two pears at second day.
He ate three plums at third day.
He ate four strawberries at fourth day.
He ate five oranges at fifth day.
nil
```

「~:*」は直前の引数をバックアップしてもう一度利用可能にする．

```
cl-user(51): (loop for food in foods
                collect (if (eql food 'strawberr)
                            (format nil
                                    "~~%He ate ~~R ~A~~:@P at ~~:*~~:R day."
                                    food)
                          (format nil
                                  "~~%He ate ~~R ~A~~:P at ~~:*~~:R day."
                                  food)))
("~%He ate ~R apple~:P at ~:*~:R day."
 "~%He ate ~R pear~:P at ~:*~:R day."
 "~%He ate ~R plum~:P at ~:*~:R day."
 "~%He ate ~R strawberr~:@P at ~:*~:R day."
 "~%He ate ~R orange~:P at ~:*~:R day.")
```

format 文はもう一つの言語と言われることが少しお分かりだろうか．ほかにも次に説明するように，繰り返し構文や， if ... then ... に相当するものなど，色々な機能があって，それらを駆使するととても複雑なことが処理できるようになるが，あまりやりすぎるとまるで暗号文のようになって，format 文でやっていることを理解することが大変になる．実際，関係者によってこんなに高機能にするんじゃなかったという反省がされているくらいである．

16.4 繰返し構文

「~{《文字列》~}」で一つのリストの要素について繰り返し制御を当てはめて実行できる．

```
cl-user(20): (setq FatherOfLisp
                   '(名前 John_McCarthy
                        生誕 1927年9月4日
                        死没 2011年10月24日（満84歳没）
                        居住 アメリカ合衆国
                        国籍 アメリカ合衆国
                        出身校 プリンストン大学-カリフォルニア工科大学))
(名前 John_McCarthy 生誕 1927年9月4日 死没 2011年10月24日（満84歳没） 居住 アメリカ合衆国 国
籍 アメリカ合衆国 ...)
cl-user(21): (format t "~{~A: ~A~%~}" FatherOfLisp)
名前: John_McCarthy
生誕: 1927年9月4日
死没: 2011年10月24日（満84歳没）
居住: アメリカ合衆国
国籍: アメリカ合衆国
出身校: プリンストン大学-カリフォルニア工科大学
nil
```

Lisp のリストを JSON 形式に直したい．丸括弧は波括弧に直せばよいが，問題はカンマの扱いである．中間にカンマは入れられるが，最後にカンマを付けないようにするには，次のようにコードする．

```
cl-user(23): (format t "{~{\"~A\": \"~A\"~^,~% ~}}" FatherOfLisp)
{"名前": "John_McCarthy",
 "生誕": "1927年9月4日",
 "死没": "2011年10月24日（満84歳没）",
 "居住": "アメリカ合衆国",
 "国籍": "アメリカ合衆国",
```

```
 "出身校": "プリンストン大学-カリフォルニア工科大学"}
nil
```

リスト中の要素について繰り返し「\"~A\": \"~A\"~^,~% 」が実施されるが，最後にリストの中身が尽きたときのみ，「~^」が働いてそこで繰り返しが終了する．

16.5 選択構文

「~:[《偽文字列》~;《真文字列》~]」は該当項目に真偽値を置いて，その結果が偽のときにはそのあと「《偽文字列》」が選択実行され，真のときには「《真文字列》」が選択実行される．以下の実施例では，もし食べ物が strawberry であれば，「~2*strawberries」が実行されるが，そうでなければ「~A~P」が food と iii について実行される．ここで，「~2*」は続く2項目を無視する制御文字である．

```
cl-user(3): (setq foods
                  '(apple pear plum strawberry orange))
(apple pear plum strawberry orange)
cl-user(4): (loop for iii from 1 to 5
                  for food in foods
               do (format t
                    "~%He ate ~R ~:[~A~P~;~2*strawberries~] at ~:R day."
                    iii (eql food 'strawberry) food iii iii))

He ate one apple at first day.
He ate two pears at second day.
He ate three plums at third day.
He ate four strawberries at fourth day.
He ate five oranges at fifth day.
nil
```

「~[《文字列0》~;《文字列1》~;《文字列2》~; ... ~]」は該当項目に自然数を置いて，その数に該当する文字列を実行する．選択は数字の 0 から始まることに留意されたい．

```
cl-user(5): (setq 食べ物
                  '(りんご 洋ナシ プラム いちご オレンジ))
(りんご 洋ナシ プラム いちご オレンジ)
cl-user(6): (loop for iii from 0 to 4
                  for food in 食べ物
               do (format t
                    "~%~Aを~[ひとつ~;ふたつ~;みっつ~;よっつ~;いつつ~]食べました。"
                    food iii))

りんごをひとつ食べました。
洋ナシをふたつ食べました。
プラムをみっつ食べました。
いちごをよっつ食べました。
オレンジをいつつ食べました。
nil
```

16.6 幅揃え

表のように印刷したいときには，「~《カラム数》< ~>」の構文を使う．九九を表のように印刷する次の例を見られたい．

```
cl-user(53): (setq multiplication-table
                  (loop for i from 1 to 9 collect
                     (loop for j from 1 to 9 collect
                        (* i j))))
((1 2 3 4 5 6 7 8 9) (2 4 6 8 10 12 14 16 18)
 (3 6 9 12 15 18 21 24 27) (4 8 12 16 20 24 28 32 36)
 (5 10 15 20 25 30 35 40 45)
 (6 12 18 24 30 36 42 48 54)
 (7 14 21 28 35 42 49 56 63)
 (8 16 24 32 40 48 56 64 72)
 (9 18 27 36 45 54 63 72 81))
cl-user(54): (format t
 "~{~{~3<~S~>~3<~S~>~3<~S~>~3<~S~>~3<~S~>~3<~S~>~3<~S~>~3<~S~>~3<~S~>~}~%~}"
 multiplication-table)
  1  2  3  4  5  6  7  8  9
  2  4  6  8 10 12 14 16 18
  3  6  9 12 15 18 21 24 27
  4  8 12 16 20 24 28 32 36
  5 10 15 20 25 30 35 40 45
  6 12 18 24 30 36 42 48 54
  7 14 21 28 35 42 49 56 63
  8 16 24 32 40 48 56 64 72
  9 18 27 36 45 54 63 72 81
nil
```

実は整数の印刷に限って言えば，右詰めで印刷するには「~D」においてカラム数で修飾すればよい．

```
cl-user(55): (format t
 "~{~{ ~2D ~2D ~2D ~2D ~2D ~2D ~2D ~2D ~2D~}~%~}"
 multiplication-table)
  1  2  3  4  5  6  7  8  9
  2  4  6  8 10 12 14 16 18
  3  6  9 12 15 18 21 24 27
  4  8 12 16 20 24 28 32 36
  5 10 15 20 25 30 35 40 45
  6 12 18 24 30 36 42 48 54
  7 14 21 28 35 42 49 56 63
  8 16 24 32 40 48 56 64 72
  9 18 27 36 45 54 63 72 81
nil
```

しかし，数字以外のデータについて右詰めをしたり，数字も含めて一般のデータで左詰めしたいような場合には，やはり「~《カラム数》< ~>」を用いるのが便利である．

明示的に右詰めを指示するときは「~《カラム数》:< ~>」，左詰めを指示するときは「~《カラム数》@< ~>」，中央揃えを指示するときは「~《カラム数》:@< ~>」を使う．次の例を見られたい．

```
cl-user(4): (setq FatherOfLisp
                  '((Name "John McCarthy")
                    (Born "September 4, 1927")
                    (BirthPlace "Boston, Massachusetts, U.S.")
                    (Died "October 24, 2011 (aged 84)")
                    (DeathPlace "Stanford, California, U.S.")
                    (Residence "United States")
                    (Nationality  "American")
```

```
                    (Fields "Computer technology")
                    (Institutions
                     "Stanford University; Massachusetts Institute of Technology; Dartmouth
College; Princeton University")))
((Name "John McCarthy") (Born "September 4, 1927")
 (BirthPlace "Boston, Massachusetts, U.S.") (Died "October 24, 2011 (aged 84)")
 (DeathPlace "Stanford, California, U.S.") (Residence "United States")
 (Nationality "American") (Fields "Computer technology")
 (Institutions
  "Stanford University; Massachusetts Institute of Technology; Dartmouth College; Princeton
University"))
cl-user(5): (format t "~{~{~14:<~S~> ~A~}~%~}" FatherOfLisp)
         Name: John McCarthy
         Born: September 4, 1927
   BirthPlace: Boston, Massachusetts, U.S.
         Died: October 24, 2011 (aged 84)
   DeathPlace: Stanford, California, U.S.
    Residence: United States
  Nationality: American
       Fields: Computer technology
 Institutions: Stanford University; Massachusetts Institute of Technology; Dartmouth College;
Princeton University
nil
cl-user(6): (format t "~{~{~14@<~S~> ~A~}~%~}" FatherOfLisp)
Name:          John McCarthy
Born:          September 4, 1927
BirthPlace:    Boston, Massachusetts, U.S.
Died:          October 24, 2011 (aged 84)
DeathPlace:    Stanford, California, U.S.
Residence:     United States
Nationality:   American
Fields:        Computer technology
Institutions:  Stanford University; Massachusetts Institute of Technology; Dartmouth College;
Princeton University
nil
cl-user(7): (format t "~{~{~14:@<~S~> ~A~}~%~}" FatherOfLisp)
     Name      John McCarthy
     Born      September 4, 1927
  BirthPlace   Boston, Massachusetts, U.S.
     Died      October 24, 2011 (aged 84)
  DeathPlace   Stanford, California, U.S.
   Residence   United States
  Nationality  American
    Fields     Computer technology
 Institutions  Stanford University; Massachusetts Institute of Technology; Dartmouth College;
Princeton University
nil
```

16.7 プリティプリント

プリティプリントとは何か？

ACL with IDE上のREPLにおいて，システムからの印刷がウィンドウ幅によって変化するのは，プリティプリントの働きである．以下の出力を参照されたい．

```
cg-user(1): (setq multiplication-table
                  (loop for i from 1 to 9 collect
```

```
                        (loop for j from 1 to 9 collect
                              (* i j))))
((1 2 3 4 5 6 7 8 9) (2 4 6 8 10 12 14 16 18) (3 6 9 12 15 18 21 24 27) (4 8 12 16 20 24 28
32 36) (5 10 15 20 25 30 35 40 45)
 (6 12 18 24 30 36 42 48 54) (7 14 21 28 35 42 49 56 63) (8 16 24 32 40 48 56 64 72) (9 18 27
36 45 54 63 72 81))
```

そのままで，もう一度印刷すると以下の通りだが，

```
cg-user(2): multiplication-table
((1 2 3 4 5 6 7 8 9) (2 4 6 8 10 12 14 16 18) (3 6 9 12 15 18 21 24 27) (4 8 12 16 20 24 28
32 36) (5 10 15 20 25 30 35 40 45)
 (6 12 18 24 30 36 42 48 54) (7 14 21 28 35 42 49 56 63) (8 16 24 32 40 48 56 64 72) (9 18 27
36 45 54 63 72 81))
```

ウィンドウ幅を狭めて印刷すると，以下のようになり，

```
cg-user(3): multiplication-table
((1 2 3 4 5 6 7 8 9) (2 4 6 8 10 12 14 16 18)
 (3 6 9 12 15 18 21 24 27) (4 8 12 16 20 24 28 32 36)
 (5 10 15 20 25 30 35 40 45) (6 12 18 24 30 36 42 48 54)
 (7 14 21 28 35 42 49 56 63) (8 16 24 32 40 48 56 64 72)
 (9 18 27 36 45 54 63 72 81))
```

更に狭めて印刷すれば，次のようになる．

```
cg-user(4): multiplication-table
((1 2 3 4 5 6 7 8 9)
 (2 4 6 8 10 12 14 16 18)
 (3 6 9 12 15 18 21 24 27)
 (4 8 12 16 20 24 28 32 36)
 (5 10 15 20 25 30 35 40 45)
 (6 12 18 24 30 36 42 48 54)
 (7 14 21 28 35 42 49 56 63)
 (8 16 24 32 40 48 56 64 72)
 (9 18 27 36 45 54 63 72 81))
```

残念ながらSlimeではこうはならない．

また，長いリストや深いリストを印刷するときに「#」とか「..」と省略されるのも，プリティプリントの働きである．ファイルに印刷する場合には，そのような機能はあまり必要がないかもしれないが，REPLでは有効な機能であり，さらにdefun構文やlet構文において，自動的に見やすく行変えとインデンテーションが付けられるのもプリティプリントのおかげである．*CLtL1*ではユーザがこのプリティプリントを制御する機能はなかったが，*CLtL2*ではそれが可能になった．

プリティプリントには重要な働きをする大域変数が五つある．

```
cl-user(2): *print-pretty*
t
cl-user(3): *print-pprint-dispatch*
#<excl::pprint-dispatch-struct @ #x201a133a>
cl-user(4): *print-right-margin*
nil
cl-user(5): *print-miser-width*
```

```
40
cl-user(6): *print-lines*
nil
```

*print-pretty*の値が真の場合，REPL上の出力は*print-pprint-dispatch*のプリティプリントディスパッチ表に従って印刷される．たとえば，defunとかcondの構文が整形されるのはそのためである．*print-right-margin*は出力幅の右マージンを文字カラム数で与える．これがnilのときは出力ストリームが出力できる最大の1行の長さが用いられる．一方，左のマージンは通常は0であるが，それはその時々のプリティプリントの論理ブロック(後述)の中に制限されるし，その中でカウントされる．*print-miser-width*は右マージンまでの残りの文字数がこれ以下になったときに，プリティプリントはコンパクトな書式(マイザープリント)のモードに切り替わる．*print-lines*はプリティプリントが印刷する行数を制限するもので，これを超える時には改行せずに「..」(空白と二つのピリオド)に省略して印刷される．既定値はnilであり，それは制限なしを意味する．以下の実施例を見られたい．

```
cl-user(7): (let ((*print-right-margin* 40)
                  (*print-miser-width* nil)
                  (*print-lines* 3))
              (pprint '(progn (setq a 1 b 2 c 3 d 4))))

(progn (setq a 1 b 2 c 3 d 4))
cl-user(8): (let ((*print-right-margin* 25)
                  (*print-miser-width* nil)
                  (*print-lines* 3))
              (pprint '(progn (setq a 1 b 2 c 3 d 4))))

(progn (setq a 1
             b 2
             c 3 ..))
```

上記実施例では，40の印刷幅では1行に印刷できたが，25の印刷幅では小さすぎるためにプリティプリントによって25文字幅に収まるように印刷された例である．

*print-pprint-dispatch*もユーザが既定のプリティプリント形式を修正したり，新しく追加したりできるようになったが，これの取り扱いは本書籍では扱わない．詳細は*CLtL2*の第27章を参照されたい．また，*CLtL2*の第27章にはプリティプリントのためのマクロや関数と，それらを利用したフォーマット文における指示子が説明してあるが，本書籍ではよりエンドユーザ向きのフォーマット文におけるプリティプリントの指示方法について説明する．

フォーマット指示子によるプリティプリント

format文でプリティプリントするときは，指示子として「~W」と「~_」と「~I」を主に使用する．

- ~W
  ~Sと似ているがプリティプリント用制御変数の値を受け取るところが異なる．深さカウンタをクリアしないので，プリティプリントの場合にはすべての場所で~Sではなくこれを用いること．
- ~_
  条件付き改行で，これへの修飾子によってリニアプリント，マイザープリント，フィルプリント，強

制改行プリントの指定をする.

- ~I
  綺麗に印刷するための字下げを指示する．その場で字下げをするのではなく字下げの位置をマークして後続の改行時にそれが有効になると考えるべきである.

以下の例ではschemeのある書式に対して「~W」で出力しているが，*print-right-margin*の値によって出力が異なっていることに注目されたい.

```
cl-user(2): (setq foo
                  '(define (factorial n)
                     (our-if (= n 1) 1 (* n (factorial (- n 1))))))
(define (factorial n) (our-if (= n 1) 1 (* n (factorial (- n 1)))))
cl-user(3): (pprint foo)

(define (factorial n) (our-if (= n 1) 1 (* n (factorial (- n 1)))))
cl-user(4): (let ((*print-right-margin* 80)
                  (*print-miser-width* nil))
              (format t "~W" foo))
(define (factorial n) (our-if (= n 1) 1 (* n (factorial (- n 1)))))
nil
cl-user(5): (let ((*print-right-margin* 40)
                  (*print-miser-width* nil))
              (format t "~W" foo))
(define (factorial n)
 (our-if (= n 1) 1
  (* n (factorial (- n 1)))))
nil
cl-user(6): (let ((*print-right-margin* 20)
                  (*print-miser-width* nil))
              (format t "~W" foo))
(define
 (factorial n)
 (our-if (= n 1) 1
  (* n
     (factorial
      (- n 1)))))
nil
```

最後の実施例6の印刷はプリティプリントされてはいるが，デフォールトのプリティプリントの仕方になっていて，defineやour-ifを認識したものではない．以下では上記のフォーマット指定子を使ったより目的別のプリティプリントを考える.

プリティプリント論理ブロック

プリティプリントを理解するのに大事な概念がリスト構造に対応した論理ブロックの指定である．フォーマット指示子「~<《制御文字列》~:>」には一つのリストが対応して，プリティプリントのための論理ブロックを形成する．これに対応しないリストは論理ブロックではなくその中の要素にすぎない．「~<《制御文字列》~:>」の制御文字列にはさらに入れ子に「~<《制御文字列》~:>」を入れることができ，これによりプリティプリントに都合のよい多重の論理ブロックを作り上げる．このフォーマット指示子がない場合でも，出力全体が一番外側の論理ブロックとなる.

以下は最も一般的な論理ブロックの使用例である．「~<《プレフィックス》~;《本文》~;《サフィックス》~:>」という形式で本文の前後に任意の文字列を入れることができるし，特に丸括弧については「~:<《本文》~:>」とコロン修飾子をつけて印刷することができる．

```
cl-user(7): (let ((*print-right-margin* 80)
                  (*print-miser-width* nil))
              (format t "~<(~;~W ~W ~W~;)~:>" foo))
(define (factorial n) (our-if (= n 1) 1 (* n (factorial (- n 1)))))
nil
cl-user(8): (let ((*print-right-margin* 80)
                  (*print-miser-width* nil))
              (format t "~:<~W ~W ~W~:>" foo))
(define (factorial n) (our-if (= n 1) 1 (* n (factorial (- n 1)))))
nil
```

一つの論理ブロックの中は条件付き改行指示子「~_」によってセクションに分けられる．今define文の本体の直前でこの論理ブロックを二つのセクションに分割するとする．

```
cl-user(9): (let ((*print-right-margin* 80)
                  (*print-miser-width* nil))
              (format t "~:<~W ~W~1I ~_~W~:>" foo))
(define (factorial n) (our-if (= n 1) 1 (* n (factorial (- n 1)))))
nil
cl-user(10): (let ((*print-right-margin* 50)
                   (*print-miser-width* nil))
               (format t "~:<~W ~W~1I ~_~W~:>" foo))
(define (factorial n)
  (our-if (= n 1) 1 (* n (factorial (- n 1)))))
nil
```

ここで，*print-right-margin*が80の場合には全体が1行で印刷され，50になるとセクション分割により改行されて，それでも関数本体が1行で印刷されていることに留意されたい．これを更に小さく20にすると以下のようになる．下記実施例11では(factorial n)が途中で改行されてしまってあまり美しくない．そこでこれをdefineの直後に改行するように変更したのが実施例12だ．二番目の「~W」の前にコロンで修飾された改行子が挿入されていることに留意されたい．

```
cl-user(11): (let ((*print-right-margin* 20)
                   (*print-miser-width* nil))
               (format t "~:<~W ~W~1I ~_~W~:>" foo))
(define (factorial
         n)
  (our-if (= n 1) 1
   (* n
      (factorial
       (- n 1)))))
nil
cl-user(12): (let ((*print-right-margin* 20)
                   (*print-miser-width* nil))
               (format t "~:<~W~3I ~:_~W~1I ~_~W~:>" foo))
(define
    (factorial n)
  (our-if (= n 1) 1
   (* n
```

```
      (factorial
       (- n 1)))))
nil
```

「~《n》I」は現在の論理ブロック中での字下げの位置を指定する．「~I」ではn=0になる．

ただの条件付き改行指示子「~_」はリニアプリント指定であり，なるべく全体を１行で印刷しようとする．それができないときに改行するがその場合には，その論理ブロック中のすべてのリニアプリント指定の改行が実行される．「~:_」はフィルプリント指定であり，その前後のセクションのいずれかで1行で印刷できないときに改行され，前後のセクションをそれぞれなるべく１行に詰め込むように印刷する．「~@_」はマイザープリントを指定するが，それが有効に働く場合は非常に限られており，まず最初に*print-miser-width*が定義されていなければならない．ここでは説明を省略する．どんな場合でも必ずその場所で改行をするのが「~:@_」である．

それでもこれはまだ，our-ifを認識していない．次の実施例を見られたい．ここではour-ifではなく，ifとしている．

```
cl-user(17): (setq bar
                   '(define (factorial n)
                      (if (= n 1) 1 (* n (factorial (- n 1))))))
(define (factorial n) (if (= n 1) 1 (* n (factorial (- n 1)))))
cl-user(18): (let ((*print-right-margin* 20)
                   (*print-miser-width* nil))
               (format t "~:<~W~3I ~:_~W~1I ~_~W~:>" bar))
(define
    (factorial n)
  (if (= n 1)
      1
    (* n
       (factorial
        (- n 1)))))
nil
```

our-ifでなくifの場合にはCommon Lispで定義されたifのプリティプリントが働いてour-ifとは印刷の仕方が異なる．our-ifのプリティプリントをCommon Lispの場合と同様にするには，次のようにする．

```
cl-user(19): (let ((*print-right-margin* 20)
                   (*print-miser-width* nil))
                (format t
                  "~:<~W~3I ~:_~W~1I ~_~:<~W~3I ~W ~_~W~1I~^ ~_~W~:>~:>"
                  foo))
(define
    (factorial n)
  (our-if (= n 1)
      1
    (* n
       (factorial
        (- n 1)))))
nil
```

「~^」はリストの終わりを検出し，その場合にはそこで印刷を打ち切るためのものである．これがないとelse部がない場合にはエラーとなる．

ここまでで，defineとour-ifに対応したプリティプリントを明らかにしたが，本当ならばこれらの特別なプリティプリントの仕方をシステムに登録しなければならない．その具体的なやり方は*CLtL2*の27.6節を参照されたい．また，論理ブロック指定を単価記号で修飾した場合には(~@<　~:>)，format関数への印刷要素がその中の印刷制御指示子に一つ一つ対応付けられるのではなく，印刷されるべき残りの要素がひとまとまりで与えられる．任意個のリスト要素を処理する場合にはこちらを利用しなければならないが，その詳細についても省略する．

第17章 Loop の呪文

17.1 繰り返し

繰り返しの実行は関数の再帰を使っても書くことはできる．実際，3.1節ではリストの要素数を数えるのに，以下に再掲するように関数の再帰呼び出しを利用した．

```
(defun count-elements (list)
  (cond ((endp list) 0)
        (t (1+ (count-elements (cdr list))))))
```

場合によっては，リストの横方向の操作では繰り返しをMAP関数に置き換えることもできる．実際，6.3節では count element を次のように書いた．

```
(defun count-elements (list)
  (apply #'+ (mapcar #'(lambda (e) 1) list)))
```

しかしこれらのコードが読みやすいかといえば，そんなことはない．再帰呼び出しがあれば，何か高度なことをしているに違いないと身構えてしまう．なんとなれば，再帰呼び出しは単純な繰り返しよりも高度なコードが書けるからだ．それはMAP関数でも同様である．MAP関数と再帰の組み合わせを使うと，とても複雑なことが短いコードで書けてしまう．単なる繰り返しであったら，見てすぐ繰り返しと分かるコードの方がいいに決まっている．*CLtL1* の dolist マクロや dotimes マクロは，繰り返しのコードを書くための構文である．これらのマクロではリストの要素について繰り返しがあるとか，ある回数分の繰り返しがあるということが見てすぐ分かる．

```
cl-user(1): (defun count-element (list)
              (let ((n 0))
                (dolist (tmp list)
                  (incf n) (print tmp))
                n))
count-element
cl-user(2): (count-element '(one two three))

one
two
three
3
```

*CLtL1*にもloop マクロはあったがその機能は非常にプリミティブで，ただ単にその中の書式並びを順に繰り返し実行するというだけのものであった．第11章で示した Scheme のREPL 部分は次のようなものだった．

```
    (loop (format t "~&==> ")
          (print (interp (read) nil)))
```

見て分かるように，このバージョンではこのループは永遠に続いてしまう．プリミティブなloopマクロでは，繰り返しを終了するための条件や繰り返しの効果を得るためのコードは一切プログラマーに任されてお

り，*CLtL1*の loop マクロは dotimes や dolist では書けない複雑な繰り返しのコードを書くときしか使い道がなかった．

*CLtL1*の仕様でcount-elementを書けば次のようなコードになる．

```
cl-user(3): (defun count-element (list)
              (let ((n 0))
                (loop
                  (when (null list) (return n))
                  (incf n)
                  (setq list (cdr list)))))
count-element
cl-user(4): (count-element '(one two three))
3
```

CLtL2 になって loop 構文が大幅に拡張され，およそ考え得る限りの機能が詰め込まれたものになった．あまりに多機能で複雑になったため，ベテランの Lisper の中にはこれを嫌う人もいる．確かに熟達しないとloopのプログラムを書くときにはなかなか大変ではあるが，できあがったプログラムを読むときにははるかに読みやすく分かりやすい．将来の自分が読むときのことを思えば，このプログラミング時の労力は払っても価値のあるものである．一年後にコードを読む自分は赤の他人なのだ．以下のコードを見てほしい．特に最後のコードはとても単純だ．単純な仕事は単純に書くことができなければならない．

```
cl-user(5): (defun count-element (list)
              (loop for nil in list with n = 0
                 do (incf n)
                 finally (return n)))
count-element
cl-user(6): (count-element '(one two three))
3

cl-user(9): (defun count-element (list)
              (loop for nil in list count t))
count-element
cl-user(10): (count-element '(one two three)
3
```

本章ではこの複雑かつ多機能な loop 構文の書き方について，学ぶ．

17.2 無条件実行節

loop の本体には do (doing) か return の字句を書いて，無条件実行節を置くことができる．

```
{do|doing} {《書式》}+ | return {《書式》 | it }
```

以下のコードは loop マクロのもっとも簡単なコードだろう．

```
cl-user(1): (loop return nil)
nil
cl-user(2): (loop do (print :OK) return t)

:OK
t
cl-user(3): (loop do (print :OK) (return :OK))
```

```
:OK
:OK
```

do あるいは doing は一つ以上の書式の並びを従えて，それを順に実行する．return は無条件でその繰り返し構文を終了させ，書式の値を持ってその loop 構文の値とする(だから本当は無条件実行の return というのはあまり使い道がない)．上記実行例で下線のついたものが無条件実行節のためにloopマクロで処理されるものである．上記2行目の return で終了させられるのは必ず該当の loop であるが，3行目の return は通常の return である．ループキーワードの return と関数の return を区別してほしい．前者はループ本体中に裸で現れるが，後者は括弧に囲われている．この場所に return-from を用いれば，さらに上位での名前付き loop や関数ブロックからの脱出をコードすることができる．

17.3 蓄積節

loop 本体に置かれるものとして，無条件実行節以外にも蓄積節がある．蓄積節には，リストに関するものと数字に関するものの二種類がある．

```
{ collect | collecting | append | appending | nconc | nconcing }
  {《書式》| it } [ into 《変数》 ]

{ count | counting | sum | summing | maximize | maximizing | minimize | minimizing }
  {《書式》| it } [ into《変数》] [《型指定》]
```

リスト蓄積節

- collectまたはcollectingは繰り返し時の書式の結果をリストにして返す．
- appendまたはappendingは繰り返し時の書式の結果を append でつなぎ合わせて返す．

```
cl-user(6): (loop for x in '(1 2 3 4)
                  collect (1+ x))
(2 3 4 5)
cl-user(7): (loop for x in '((1) (2) (3) (4))
                  append x)
(1 2 3 4)
```

6.3 節で定義した mappend は loop 構文で次のようにも定義できる．

```
(defun mappend (function &rest lists)
  "Applies 《function》 to respective element(s) of each 《list》, append
  all the result lists to a single list. 《function》 must return a list."
  (loop for result in (apply #'mapcar function lists)
     append result))
```

- nconcまたはnconcingは繰り返し時の書式の結果を nconc でつなぎ合わせて返す．

数蓄積節

- countまたはcountingは繰り返し時の書式の結果が真となる回数を数える．

```
cl-user(8): (loop for x in '(one 1 two 2 three 3 four)
                  count (numberp x))
3
```

```
cl-user(9): (loop for x in '(one 1 two 2 three 3 four)
                count (not (numberp x)) into n
                finally (return n))
4
```

- sumまたはsummingは繰り返し時の書式の結果を合計したものを返す.

```
cl-user(11): (loop for num in '(1 2 3 4)
                  sum num)
10
```

- maximizeまたはmaximizingは繰り返し時の書式の結果のうち，最大値を返す.
- minimizeまたはminimizingは最小値を返す.

```
cl-user(12): (loop for num in '(2 1 5 4 3)
                  maximize num)
5
cl-user(13): (loop for num in '(2 1 5 4 3)
                  minimize num)
1
```

17.4 条件実行節

蓄積節の count または counting は，評価する書式が真のときしかカウントしなかった．これはたまたま設計者がどんな場合でもカウントするという仕様よりもこの方が価値があると思ったのであろう．一般に，条件に合致した時だけ無条件実行節や蓄積節を実行したいときは，ここで説明する条件実行節を用いる.

```
{ if | when | unless } 《書式》
        《無条件実行節/蓄積節/条件実行節》
          {and 《無条件実行節/蓄積節/条件実行節》}*
   [ else 《無条件実行節/蓄積節/条件実行節》
            {and 《無条件実行節/蓄積節/条件実行節》}* ]
 [ end ]
```

```
cl-user(10): (loop for num in '(1 2 3 4 5)
                  if (evenp num) count num into even-n
                    else count num into odd-n
                  finally (return (values even-n odd-n)))
2
3
```

《無条件実行節/蓄積節/条件実行節》 を複数実行したいときは and を使って並べることができる.

```
cl-user(13): (loop for num in '(1 2 3 4 5)
                  if (evenp num) count num into even-n and do (print num)
                  else count num into odd-n and summing num into odd-sum
                  finally (return (values even-n odd-n odd-sum)))

2
4
2
3
9
```

問題 17.1 ラムダ式において

17.5 コレクションについての for-as 節

(loop for 《変数》 in 《リスト書式》 ...) という loop 書式はこれまでも何回も出てきたが，これは《リスト書式》の評価結果のリストについて，繰り返しのたびにその各要素を順番に《変数》に束縛して loop を実行するというものである．リストの要素について繰り返す場合には in を使うが，ベクタの各要素について実行するときは in の代わりに across を使い，ハッシュ表の各要素について実行するときは being each hash-value of を使う．パッケージ内のシンボルについてすら，この for as loop 構文が使える．

リストについての for-as 節

リストについての for-as 節は正確には次のような構文である．

{for|as}《変数》[《型指定》] {in|on}《リスト書式》[by 《ステップ関数》]

ここで for を使おうと as を使おうと，その効果は全く同じであるが，前置詞 in ではリストの各要素が《変数》に束縛され，前置詞 on では繰り返しのたびに要素ではなく部分リストが《変数》に束縛される．

```
cl-user(15): (loop as x in '(1 2 3 4)
                collect x)
(1 2 3 4)
cl-user(16): (loop as x on '(1 2 3 4)
                collect x)
((1 2 3 4) (2 3 4) (3 4) (4))
```

《ステップ関数》を指定すると，一つずつ順番にではなく，1個飛びや 2個飛びなどを指定することができる．

```
cl-user(17): (loop as x in '(1 2 3 4) by #'cddr
                collect x)
(1 3)
cl-user(18): (loop as x on '(1 2 3 4) by #'cddr
                collect x)
((1 2 3 4) (3 4))
```

ベクタについての for-as 節

リストではなくベクタについて繰り返しを行う場合には，次の構文を用いる．

{for|as} 《変数》 [《型指定》] across 《ベクタ書式》

ベクタをリストに変換するには通常 coerce を用いるが，loop 構文を使えば次のようになる．

```
cl-user(19): (coerce #(1 2 3 4) 'list)
(1 2 3 4)
cl-user(20): (loop as x across #(1 2 3 4)
                collect x)
(1 2 3 4)
```

ベクタについて，リストの場合のように要素を飛び飛びに選択することはできない．

問題17.1 文字列もベクタの一種である．6.4節のMAP-REDUCEでは以下のようにMAP関数で文字列を文字のリストに変更した．loop-as節を使って同じことをせよ．

```
(map 'list #'identity "Change to a list of chars")
 -> (#\C #\h #\a #\n #\g #\e #\Space #\t #\o #\Space ...)
```

問題17.2 残念ながら，値の累積の結果をベクタで返すようなloop構文はない．6.4節では以下のようにMAP関数で文字列のリストからそれぞれの頭文字を取りだして省略語を作りだした．loop-asとcoerceを使って同じことをせよ．

```
(map 'string #'(lambda (str) (char str 0))
     '("common" "lisp" "-" "artificial" "intelligence" "programming"))
 -> "cl-aip"
```

ハッシュ表についての for-as 節

ハッシュ表についての繰り返しの場合，ハッシュキーについて変数束縛をしたり，値について変数束縛をしたり，あるいはその両方についても変数束縛をすることができる．そのため，書き方も少しだけ複雑になる．

```
{for|as} 《変数1》 [《型指定》] being {each|the}
  { {hash-key|hash-keys} {in|of} 《ハッシュ表》 [using (hash-value 《変数2》)] |
    {hash-value|hash-values} {in|of} 《ハッシュ表》 [using (hash-key 《変数2》)]
  }
```

```
cl-user(21): (setq foo (make-hash-table))
#<eql hash-table with 0 entries @ #x1023008d2>
cl-user(22): (loop for x in '(one two three four)
                   for i from 1 to 4
                do (setf (gethash x foo) i))
nil
cl-user(23): (gethash 'three foo)
3
t
cl-user(24): (loop for key being each hash-key
                   in foo using (hash-value val)
                collect (,val . ,key))
((4 . four) (1 . one) (2 . two) (3 . three))
cl-user(25): (sort * #'< :key #'car)
((1 . one) (2 . two) (3 . three) (4 . four))
```

ここでは，最初に one，two，three，four をキーにハッシュ表を作ったが，あとでそのハッシュ表から数字をキーに変えた連想リストを作っている．

複数形の字句はその前に前置詞 the と組み合わせて使うが，それで特に each の場合と効果が異なるわけではない．

```
cl-user(26): (loop for key being the hash-keys
                   in foo using (hash-value val)
                collect (,val . ,key))
```

```
((4 . four) (1 . one) (2 . two) (3 . three))
```

パッケージについての for-as 節

5.4 節で説明したように，Common Lisp のシンボルはパッケージ毎に分かれて，各パッケージに登録されている．for-as 節を使って任意のパッケージ中のすべての登録シンボルを取り出すことができる．

```
{for|as} 《変数》 [《型指定》] being {each|the}
  {symbol|symbols|present-symbol|present-symbols|external-symbol|external-symbols} [{in|of}
《パッケージ》]
```

symbol(s) の字句を使った場合には，指定のパッケージで参照可能なすべてのシンボルについて，繰り返しが実施される．external-symbol(s) の場合には，指定のパッケージの外部シンボルについて繰り返しが行われる．present-symbol(s) の場合には，そのパッケージ中に存在する**外部シンボル以外の**シンボルについて繰り返しが実施される．

Allegro Common Lisp 9.0 の場合には，次のような結果が得られた．

```
cl-user(53): (loop for x being each symbol of :common-lisp-user
                count t)
1675
cl-user(54): (loop for x being each external-symbol of :common-lisp-user
                count t)
0
cl-user(55): (loop for x being each present-symbol of :common-lisp-user
                count t)
8
```

この例の具体的な結果は処理系によって異なることに注意されたい．

17.6 数字指定の for-as 節

for-as 節のもう一つの典型的パターンは，繰り返し数字を変数とともに指定する方法である．これには数を増加させるパターンと減少させるパターンがある．

```
{for|as} 《変数》 {from|upfrom} 《開始数書式》 {to|upto|below} 《終了数書式》 [by 《ステップ
数書式》]
{for|as} 《変数》 from 《開始数書式》 {downto|above} 《終了数書式》 [by 《ステップ数書式》]
{for|as} 《変数》 downfrom 《開始数書式》 {to|downto|above} 《終了数書式》 [by 《ステップ数書
式》]
```

```
cl-user(11): (loop for i from 0 to 5
                do (print i))

0
1
2
3
4
5
nil
```

```
cl-user(12): (loop for i from 1 to 10 by 2
                do (print i))

1
3
5
7
9
nil
cl-user(13): (loop for i downfrom 10 to 1 by 2
                do (print i))

10
8
6
4
2
nil
```

終了数まで実行されること，by 《ステップ数書式》を省略したときは1ずつ増加すること，に注意されたい.

17.7 多重のループ

for-as 節は一つのループでいくつでも重ねることができるが，それとループが入れ子になった構造の場合を区別してほしい.

```
cl-user(2): (loop for elm in '(one two three)
                  and j from 1 to 3
               collect `(,elm . ,j))
((one . 1) (two . 2) (three . 3))
cl-user(3): (loop for elm in '(one two three)
                  for j from 1 to 3
               collect `(,elm . ,j))
((one . 1) (two . 2) (three . 3))
cl-user(4): (loop for i from 1 to 9 collect
              (loop for j from 1 to 9 collect
                 (* i j)))
((1 2 3 4 5 6 7 8 9)
 (2 4 6 8 10 12 14 16 18)
 (3 6 9 12 15 18 21 24 27)
 (4 8 12 16 20 24 28 32 36)
 (5 10 15 20 25 30 35 40 45)
 (6 12 18 24 30 36 42 48 54)
 (7 14 21 28 35 42 49 56 63)
 (8 16 24 32 40 48 56 64 72)
 (9 18 27 36 45 54 63 72 81))
```

上記例 collect では1次元構造か2次元構造になるかでその違いが分かりやすいが，そうでない場合の方が普通であり，注意しないと多重の for-as を入れ子構造に書いてしまう.

17.8 変数の for-as 節

より自由な書き方を許す for-as 節として変数の値を更新する次のようなパターンがある.

```
{for|as} 《変数》 = 《初期書式》 [then 《更新書式》]
    {and 《変数》 = 《初期書式》 [then 《更新書式》]}*
```

この書き方では loop 変数として《変数》を導入し，ループ開始時の値として《初期書式》の値を，繰り返し時には《更新書式》の値を《変数》に束縛する．then 《更新書式》 が無い場合には，《初期書式》が繰り返し用いられる．多数の loop 変数を導入するときは，and を用いて導入する．ただし，その場合は変数への束縛は let 変数と同様に並列に行われる．以下に示すように，and なしで，複数の loop 変数を導入することもできるが，その場合には let* と同様に，変数束縛が順番に行われる．

```
cl-user(33): (loop repeat 5
                   for x = 0 then y
                   for y = 1 then (+ x y)
                do (print (,x ,y)))

(0 1)
(1 2)
(2 4)
(4 8)
(8 16)
nil
cl-user(34): (loop repeat 5
                   for x = 0 then y
                   and y = 1 then (+ x y)
                do (print (,x ,y)))

(0 1)
(1 1)
(1 2)
(2 3)
(3 5)
nil
```

17.9 While と Until ループ

while はテスト書式が真である間はループが実行される．until はテスト書式が真になったらループを終了させる．

```
cl-user(6): (loop for x = (read)
                  initially (print '>)
                  while x
                  collect x
                  do (prin1 x)
                  (print '>))

> This
This
> is
is
> a
a
> test.
test.
> nil
(This is a test.)
```

```
cl-user(7): (loop for x = (read)
                  initially (print '>)
                  until (eql x :quit)
                  collect x
                  do (prin1 x)
                  (print '>))

> This
This
> is
is
> also
also
> a
a
> test.
test.
> :quit
(This is also a test.)
```

initially 節はループが始まる前に評価され，これまでもしばしば出てきたが finally 節はループが終了したあとに評価される．ただし，finally節のformで該当loopの値を返すときはreturn関数で値を返さなければならない．

17.10 alwaysとthereisとnever

while 節や until 節には無条件実行節や蓄積節を書くことができるが，それらが必要ない場合には，while や until の代わりに always（条件が真の場合には繰り返しが行われるが，偽になったときには繰り返しが終了する）や thereis（条件が偽の場合には繰り返しが行われるが，真になったときに繰り返しが終了する）を使えばよい．この loop節が返す値は always なら nil（条件が満足されなかったとき）か t であり（最後まで満足された時），thereis ならその条件節の評価結果（真のとき）か nil（最後まで満足されなかった時）である．neverでは条件が満足されたときに繰り返しを終了して nil を返し，それ以外の時に t を返す．

これらの繰り返しでは，途中でループを抜ける時にはfinally節は書いてあっても実行されないことに注意されたい．ループが最後まで実行されて抜ける時にはfinally節が実行される．

```
cl-user(2): (loop for i in '(1 2 3 4)
                  always (plusp i))
t
cl-user(3): (loop for i in '(1 -2 3 4)
                  thereis (minusp i))
t
cl-user(4): (loop for i in '(1 2 3 4)
                  never (minusp i)
                  finally (print 'OK))

OK
t
cl-user(5): (loop for i in '(1 -2 3 4)
                  never (minusp i)
                  finally (print 'OK))
nil
```

```
cl-user(6): (loop for i from 1 to 10
                  thereis (> i 11)
                  finally (print i))

11
nil
```

17.11 一時的変数の導入

for-as 変数のパターンでは，常にループのたびに変数の値が更新されるが，そうではなくて，ループの中だけで使う let 変数的なものがほしい場合は，以下のように with 節を使えばよい.

```
with 《変数》 = 《初期値》{and 《変数》 = 《初期値》}*
```

```
cl-user(8): (loop with one = 1 and two = 2 and three = 3
                  return (list one two three))
(1 2 3)
cl-user(9): (loop with one = 1
                  with two = (1+ one)
                  with three = (1+ two)
                  return (list one two three))
(1 2 3)
```

上記例から分かるように，`with ... = ... and ... = ...` の場合には，ちょうど let 変数のように束縛は並列に行われるが，`and` がない場合には，let* 変数のように，束縛は順番に行われ，先に束縛された変数をあとの with 書式で使うことができる.

17.12 分配束縛

分配束縛とは destructuring-binding のことである．この機能は*CLtL1*のときから暗黙のうちにマクロのパラメータ束縛に使われていたが，*CLtL2*のloopでも使われるようになるとともに，便利な機能であるため明示的に `destructuring-binding` マクロを使うことで，コードのどこにでも使えるようになった.

分配束縛ではリスト構造に着目した対応関係で，パラメータと実オブジェクトとの束縛関係を作ることができる．簡単かつ典型的な例は連想リストの場合である.

分配束縛を使わないとすると，こんな風にプログラムしなければならない.

```
cl-user(2): (loop for elm in '((one . 1) (two . 2) (three . 3))
                  with key and val
                do (setq key (car elm))
                   (setq val (cdr elm))
                   (format t "~%~A: ~D" key val))

one: 1
two: 2
three: 3
nil
```

しかし，分配束縛を使えば次のように簡単になる.

```
cl-user(3): (loop for (key . val) in '((one . 1) (two . 2) (three . 3))
              do (format t "~%~A: ~D" key val))

one: 1
two: 2
three: 3
nil
```

分配束縛では構造パターンに対応して変数が束縛されるが，一部の変数を使わないという場合もある．そういう場合は使わない変数部分を nil に置き換えればよい．ビットバケットみたいなものである．

```
cl-user(4): (loop for (nil . val) in '((one . 1) (two . 2) (three . 3))
              do (format t "~%~D" val))

1
2
3
nil
```

destructuring-bind マクロを使うと，コード中の任意の場所で複雑な構造のデータについて変数との対応関係を取ることができる．inverse-family-treeは，子から親への家系図であるが，子は常に父方の逆家系図と母方の逆家系図を持つ構造になっている．

```
cl-user(5): (setq inverse-family-tree
                  '(安倍晋三 (安倍晋太郎 (安倍寛 (安倍彪助)
                                                 (安倍タメ))
                                          (安倍静子 (本堂恒次郎)
                                                    (本堂秀子)))
                             (安倍洋子 (岸信介 (佐藤秀助)
                                               (佐藤茂世))
                                       (岸良子 (岸信政)
                                               (岸千代)))))
(安倍晋三 (安倍晋太郎 (安倍寛 (安倍彪助) (安倍タメ)) (安倍静子 (本堂恒次郎) (本堂秀子))) (安倍洋子 (岸信介 (佐藤秀助) (佐藤茂世)) (岸良子 (岸信政) (岸千代))))
```

先祖を求める関数は再帰しているが，分配束縛によって逆家系図が繰り返し一人の子供と母方逆家系図と父方逆家系図に分配されているところを見られたい．

```
cl-user(6): (defun get-ancestor (inv-tree sex-trail &optional trail)
              (cond ((null inv-tree) (nreverse trail))
                    (t (destructuring-bind
                           (child &optional father-tree
                                            mother-tree)
                           inv-tree
                         (ecase (first sex-trail)
                           (m (get-ancestor father-tree
                                            (cdr sex-trail)
                                            (cons child trail)))
                           (f (get-ancestor mother-tree
                                            (cdr sex-trail)
                                            (cons child trail))))))))
get-ancestor
cl-user(7): (get-ancestor inverse-family-tree '(f m m m))
(安倍晋三 安倍洋子 岸信介 佐藤秀助)
```

第18章 エラー，警告，コンディションシステム

> 幸福な家庭はどれも似たものだが、不幸な家庭はいずれもそれぞれに不幸なものである。
>
> [トルストイ，1877]（アンナカレーニナ，中村訳）

Common Lisp以前にもエラー処理はあったが，その取扱いは各実装者まかせであった．Common Lisp 策定時にはエラー処理についても議論されたが，当時はまだ流動的な状態であって，*CLtL1*ではエラーを発報する仕方は標準化されたが，エラーをどう処理するかについての機能の標準化はお預けとなった．その後，*CLtL2*では異常状態の処理方法がコンディションシステムとして標準化され，エラーはそのコンディションシステムの一部の機能となった．*CLtL1*の第24章エラーはたかだか8ページであったのが，*CLtL2*では第24章エラーの11 ページに加えて，第29章コンディションは53ページもある（いずれも日本語訳で）．本章では，最初にエラーと警告について学び，次にコンディションシステムについて解説する．*CLtL2*のコンディションシステムを利用すると，異常時の対処方法までプログラムできる．不確実な環境で生き残る人工知能システムの開発には，Common Lispのコンディションシステムを中心としたプログラム開発方法がもっと研究されてよい．

18.1 エラー

継続不可能なエラー

以下は整数のバイトサイズを返す関数だが，256バイト以上についてはエラーとする．

```
cl-user(19): (defun integer-size (integer)
  (let ((n-bytes (ceiling (integer-length (abs integer)) 8)))
    (unless (< n-bytes (expt 2 8))
      (error "Integer too large: ~D" integer))
    (1+ n-bytes)))
integer-size
cl-user(20): (integer-size
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000)
256
cl-user(21): (integer-size
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000)
Error: Integer too large:

100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000
```

関数 error の呼び出し形式は以下のとおりである．

(error《出力制御文字列》{《出力項目》}*)

《出力制御文字列》と《出力項目》の関係はformat文と類似であるが，実際にどのように出力されるかについては処理系依存のところがある．たいていは最初に "Error" などと固定的に出力される．この関数はトップレベルに無条件に制御を移してしまうので，処理途中でユーザがデバッグ作業などを行うことはできない．デバッグ終了後，再度最初からやり直しとなる．

継続可能なエラー

関数errorではユーザがその場で誤りを修正する手段がない．cerrorではプログラムがエラーを訂正する手段を提供して，計算を継続することができる．

たとえば，関数internを呼び出すときにパッケージは必ず存在していなければならない．以下に示す関数 intern* は，もし誤って間違ったパッケージを指定すると，正しいパッケージ名の入力を促す．下線の部分がエラーメッセージとしてプログラムされたところである．

```
cl-user(1): (defun intern* (name &optional (package *package*))
              (if (find-package package)
                  (intern name package)
                (progn (cerror "Enter an existing one."
                               "There is no ~S package."
                               package)
                  (format t "package? ")
                  (intern* name (read)))))
intern*
cl-user(2): (intern* "foo")
foo
nil
cl-user(3): (intern* "foo" :mystery)
Error: There is no :mystery package.

Restart actions (select using :continue):
 0: Enter an existing one.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] cl-user(4): :continue 0
package? :cl-user
foo
:internal
```

関数 cerror の呼び出し形式は以下のとおりである．

(cerror《継続用制御文字列》《出力制御文字列》{《出力項目》}*)

しかし cerror では比較的簡単な場合しか処理できない．より複雑な例外状況に対処するには後述の condition-handler を用いることができる．

問題 18.1 上記例で正しいパッケージ名をタイプ入力をさせるのではなく，その名前のパッケージを作ってよいか聞いて，選択されたらそのようにする cerror プログラムを書け．

18.2 警告

警告は，単にユーザにメッセージを出力するのみで，プログラムの実行には何ら影響を及ぼさない．システムはその状況を受け入れるが，望ましい状態ではない，というような場合に使われる．

```
cl-user(4): (defun warn-obsolete-argument (old-arg new-arg)
              (warn "Keyword ~S is obsolete, please use ~S"
                    old-arg new-arg))
warn-obsolete-argument
cl-user(5): (warn-obsolete-argument 'eval :execute)
Warning: Keyword eval is obsolete, please use :execute
nil
```

関数 warn の呼び出し形式は以下のとおりである．

(warn《出力制御文字列》{《出力項目》}*)

18.3 コンディションシステム

先に述べたように，*CLtL1*におけるerrorやwarnは，*CLtL2*になって修正され，コンディションシステムの一部となった．error と cerror は内部で simple-error の実現体を作り，関数 signal にそれが与えられる．warn は simple-warning の実現体を作り関数 signal にそれが与えられる．以下の実施例を確認されたい．

```
cl-user(1): (trace signal)
(signal)
cl-user(2): (error "This is a test for making an error.")
 0[2]: (signal #<simple-error @ #x100aec612>)
 0[2]: returned nil
Error: This is a test for making an error.

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(3): :continue 0
cl-user(4): (cerror "continue?" "This is a test for making an cerror.")
 0[2]: (signal #<simple-error @ #x100aef462>)
 0[2]: returned nil
Error: This is a test for making an cerror.

Restart actions (select using :continue):
 0: continue?
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] cl-user(5): :continue 1
cl-user(6): (warn "This is a test for making a warn.")
 0[2]: (signal #<simple-warning @ #x100af11c2>)
 0[2]: returned nil
```

```
 0[2]: (signal #<excl::advance-warning @ #x1002cf462>)
 0[2]: returned nil
Warning: This is a test for making a warn.
nil
```

ここで，コンディションシステムを利用する場合の error と cerror と warn の正式な呼び出し形式を挙げておく．

(error《データ》{《アーギュメント》}*)

(cerror《継続用制御文字列》《データ》{《アーギュメント》}*)

(warn《データ》{《アーギュメント》}*)

しかし実際に《データ》と《アーギュメント》に何を与えるかは*CLtL2*を読んでも分からない．実は，《データ》にはコンディションクラス名を与える．すると与えたコンディションクラス名の実現体が内部で作られるが，その実現体生成の際に初期化パラメータとして用いられるのが《アーギュメント》なのである．だから，アーギュメントに何が来るべきかというのは《データ》に与えるコンディションクラス名に依存する．simple-error の場合にはそれは次のように，:format-controlとその値，および:format-argumentsとその値である．

```
cl-user(1): (defun my-error (format-control &rest format-arguments)
              (error 'simple-error
                :format-control format-control
                :format-arguments format-arguments))
my-error
cg-user(2): (my-error "Integer too large: ~D" 1000000)
Error: Integer too large: 1000000
[condition type: simple-error]
```

継続的エラー cerror の場合には，《継続用制御文字列》の取り扱いが少し難しく，たとえば次のようになる．ここで，continue-format-controlが《継続用制御文字列》である．

```
cl-user(1): (defun my-cerror (continue-format-control format-control
                                        &rest format-arguments)
              (cerror continue-format-control
                      'simple-error
                      :format-control format-control
                      :format-arguments format-arguments))
my-cerror
cl-user(2): (defun real-sqrt (n)
              (when (minusp n)
                (setq n (- n))
                (my-cerror "Return sqrt(~*~*~*~{~D~}) instead."
                           "Tried to take sqrt(-~D)."
                           n))
              (sqrt n))
real-sqrt
cl-user(3): (real-sqrt 4)
2.0
cl-user(4): (real-sqrt -9)
Error: Tried to take sqrt(-9).
  [condition type: simple-error]
```

```
Restart actions (select using :continue):
 0: Return sqrt(9) instead.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] cl-user(5): :continue 0
3.0
```

ここで，《継続用制御文字列》における「~*~*~*」は，cerrorに与える引数のうち，:format-controlに始まる三つの実引数をスキップするためのものである．ここは*CLtL2*の記述とは異なるが，ACLでもSBCLでもこのとおりである．つづく「~{~D~}」についても同様にこれで正しいが，ここではformat-argumentsの値は(9)となっていることに留意されたい．

コンディションシステムはCLOSで実装されているが，コンディションシステムにおける最上位のクラスはconditonである．その最上位クラスからのクラス階層図はACLの場合以下のようになっている．この階層図はstandard-objectの下にあり，コンディションシステムはCLOS標準メソッド結合のプロトコルに従っている．下記階層図の中で，simple-error と simple-warning の占める位置を確認されたい．見て分かるように，simple-errorの直接上位クラスはerrorとsimple-condition，simple-warningの直接上位クラスはwarningとsimple-conditionである．

```
(condition
  (excl::compiler-note)
  (compiler::ics-dependent-passc)
  (compiler::ics-dependent-passa)
  (excl::utf8-generating-surrogate-pair)
  (socket-chunking-end-of-file)
  (excl::advance-warning)
  (excl::input-edit)
  (excl::non-error-operating-system-signal
    (excl::non-error-asynchronous-operating-system-signal
      (excl::non-erroring-interrupt-signal)))
  (serious-condition
    (storage-condition)
    (error
      (multiprocessing::process-join-error)
      (excl::license-support-expired)
      (excl::locale-missing-category)
      (void-external-format)
      (fasl-casemode-mismatch)
      (type-error (case-failure)
                  (simple-type-error))
      (parse-error (reader-error (excl::extra-right-paren-error)))
      (stream-error
        (socket-error)
        (errno-stream-error)
        (stream-closed-error)
        (end-of-file)
        (reader-error
          (excl::extra-right-paren-error)))
      (package-error (package-locked-error))
      (winapi-error)
      (syscall-error
        (file-error
```

```
          (file-incompatible-fasl-error)
          (file-does-not-exist-error
            (autoload-file-does-not-exist-error))))
      (cell-error (undefined-function)
                  (unbound-variable)
                  (unbound-slot))
      (print-not-readable)
      (arithmetic-error
        (floating-point-inexact)
        (floating-point-invalid-operation)
        (floating-point-underflow)
        (floating-point-overflow)
        (division-by-zero))
      (purespace-write-error)
      (operating-system-signal
        (synchronous-operating-system-signal)
        (asynchronous-operating-system-signal (interrupt-signal)))
      (no-external-format-error)
      (compiler-not-available-error)
      (program-error)
      (control-error)
      (simple-error
        (lep:lep-error (lep:lep-operation-aborted)
          (lep:no-lep-connection)))))
  (break-condition (simple-break))
  (warning
    (foreign-functions::strings-convert-warning
      (foreign-functions::strings-convert-rt-warning)
      (foreign-functions:strings-convert-def-warning
        (foreign-functions::strings-convert-noclosure-warning)
        (foreign-functions::strings-convert-noconv-warning)))
    (compiler::dynamic-extent-argument-specification-change)
    (incompatible-conformance-change-warning)
    (excl::ef-precomp-mismatch-warning)
    (excl::foreign-strings-convert-warning)
    (compiler-not-available-warning)
    ( )
    (style-warning
      (compiler-undefined-functions-called-warning)
      (compiler-unreachable-code-warning)
      (compiler-no-in-package-warning)
      (compiler-inconsistent-name-usage-warning)
      (ineffective-declaration-warning
        (excl::float-declaration-used-warning))))
  (simple-condition
    (simple-type-error)
    (simple-error
      (lep:lep-error (lep:lep-operation-aborted)
        (lep:no-lep-connection)))
    (simple-break)
    (simple-warning)))
```

実はここで用いられた，format-controlとformat-argumentsスロットはconditionクラスに定義されていて，simple-errorもsimple-warningもそのスロット定義を継承してきている．大規模システムを開発するときには，エラーの体系を自分なりにこのsimple-errorやsimple-warningの下に構築することで，エラーの詳細

化，特殊化を実装できる．特殊化されたエラーにおいては，出力形式を特殊化してカスタマイズすることもできるし，新しいスロットを追加して，ユーザに提供する情報量を上げることもできる．

第19章 多値，ファイル，入出力ストリーム

19.1 多値

昔のLispでは，関数が返す値は一つのみであり，もしいくつもの結果を返したければリストにするしかなかった．Common Lispでは関数は複数の値を返すことができる．これまでもいくつか多値を返す関数が登場した．

```
cl-user(4): (round (/ 22 7))
3
1/7
cl-user(5): 'foo
foo
cl-user(6): (intern "foo")
foo
:internal
cl-user(7): (subtypep 'vector 'array)
t
t
cl-user(8): (setq *db* (make-hash-table))
#<eql hash-table with 0 entries @ #x216c7fa2>
cl-user(9): (setf (gethash 'one *db*) 1)
1
cl-user(10): (gethash 'one *db*)
1
t
```

上記のように，round，intern，subtypep，gethashは二つの値を返す．ただし，多値を返す関数であっても多値を受け取るコードを書かないかぎり，1番目の値しか受け取れない．そのため1番目の値はprimary valueと呼ばれる．

プログラム中で多値を受け取るには二つの方法がある．let変数に似たmultiple-value-bindとsetqに似たmultiple-value-setqである．

```
cl-user(11): (defun test1 (key)
               (multiple-value-bind (val present)
                                    (gethash key *db*)
                 (if present
                     (format nil "The value of ~S is ~S." key val)
                   (format nil "There is no value on ~S." key))))
test1
cl-user(12): (test1 'one)
"The value of one is 1."
cl-user(13): (test1 'two)
"There is no value on two."
cl-user(14): (defun test2 (key)
               (let (val present)
                 (multiple-value-setq (val present)
                                      (gethash key *db*))
                 (if present
                     (format nil "The value of ~S is ~S." key val)
                   (format nil "There is no value on ~S." key))))
```

```
test2
cl-user(15): (test2 'one)
"The value of one is 1."
cl-user(16): (test2 'two)
"There is no value on two."
cl-user(17): (setf (gethash 'two *db*) 2)
2
cl-user(18): (setf (gethash 'one *db*) nil)
nil
cl-user(19): (test2 'one)
"The value of one is nil."
cl-user(20): (test2 'two)
"The value of two is 2."
```

multiple-value-bindでは新しい局所変数が導入されてその変数束縛による本体(body)の書式を持つが，multiple-value-setqはそうではなくsetqと同様に振る舞うことに注意されたい．

multiple-value-callを用いると，変数を経由することなく直接関数に多値を引き渡すことができる．

```
cl-user(21): (defun test3 (key)
               (multiple-value-call
                   #'(lambda (val present)
                       (if present
                           (format nil "The value of ~S is ~S."
                                       key val)
                         (format nil "There is no value on ~S."
                                     key)))
                 (gethash key *db*)))
test3
cl-user(22): (test3 'two)
"The value of two is 2."
cl-user(23): (test3 'three)
"There is no value on three."
```

一方，自分のコードで多値を返すときには関数 valuesを用いる．この関数は任意個の引数をとってそれらを多値として返すが，引数がない場合には何も値を返さない．

```
cl-user(25): (values 1)
1
cl-user(26): (values 1 2)
1
2
cl-user(27): (values 1 2 3)
1
2
3
cl-user(28): (values)
cl-user(29):
```

関数内のコード中では，値を返すべきところ，すなわち暗黙のPROGNの最後とか，retern-from の呼び出し場所などにこの values のコードを書く．次のプログラム ensure-external は intern と似ているが，必ず移送(export)して，引数に与えられたシンボルを外部シンボルとするようなプログラムである．

```
cl-user(29): (defun ensure-external
                    (symbol &optional (package *package*))
```

```
                "ensures 《symbol》 to be external. If it is
                 a string and such a symbol does not exist, it is made."
                (multiple-value-bind (sym status)
                    (intern (string symbol) package)
                  (ecase status
                    (:external (values sym status))
                    (:internal (export sym package)
                               (values sym :external))
                    (:inherited (error "Not Yet!"))
                    ((nil)                ; newly created
                     (export sym package)
                     (values sym :external)))))
ensure-external
cl-user(30): (ensure-external "strange-symbol" (make-package :mystery))
mystery:strange-symbol
:external
```

上記において引数として与えたシンボルが他のパッケージから継承したもの(inherited)だった場合，エラーで"Not Yet!"とメッセージを出力している．この場合，新しく指定パッケージ内にシンボルを作るべきかそれともそのままにしてさらに移送すべきかどうか，まだプログラマーが決めかねているわけだが，このように仕様が未確定の場合には，上記のようにとりあえずエラーを起こすようにしておいて，あとから考えようとするのは，よくやる手である．

通常ならば，関数や書式中から抜ける場所に多値を返すvaluesやvalues-listを書けば，多値が返る．ところがprog1だけはprimary valueしか返さない．もし多値を返すprog1が欲しければ，multiple-value-prog1を使う．

```
cl-user(3): (progn
              (setq foo nil)
              (values-list (list 1 2 3)))
1
2
3
cl-user(4): (prog1
                (values-list (list 1 2 3))
              (setq foo nil))
1
cl-user(5): (multiple-value-prog1
                (values-list (list 1 2 3))
              (setq foo nil))
1
2
3
```

multiple-value-bindやmultiple-value-setqでは扱う返値の数が固定的である．もし，任意個の返値を扱いたいと思ったら，values-listとmultiple-value-listを使うしかない．values-listでは任意個の返値を作りだすことができるし，multiple-value-listは任意個の返値をリストにして受け取ることができる．

```
cl-user(6): (defun foo (&rest vals)
              (values-list vals))
foo
cl-user(7): (foo 1 2 3 4)
1
```

```
2
3
4
cl-user(8): (multiple-value-list (foo 1 2 3 4))
(1 2 3 4)
```

19.2 ファイル

今日一般的に利用される計算機は，Unix/LinuxかMS WindowsかAppleのMacがほとんどであり，これら三種類のことを考えれば十分であるが，Common Lisp 制定前の米国では，DECのTOPS-10，TOPS-20，ITS，TENEX，VAX/VMS，それにUNIXと様々な計算機があり，加えてLisp専用機であるSymbolicsマシンとXeroxマシンがあった．当時WindowsマシンとMacintoshはまだ無かった．これらの計算機によるファイルシステムの差異を吸収するために，Common Lispでは各種マシンごとのファイル指定方法を抽象化したパスネームというものを考え出した．ファイル操作の関数のほとんどで，文字列によるマシン固有のファイル表記が有効ではあるが，それでは今日でもLinux用プログラムをWindowsにトランスポートして使うということはできない．だから，移植性を考慮すればパスネームによるファイル操作をマスターしなければならない．

Windows上のACLでは実ファイル名を指定して，次のようにprobe-fileを使ってファイルの有無を調べることができる．ただし，逆スラッシュ「\」はCommon Lisp文字列においてはエスケープ文字なので，文字列中では常に二重にしなければならない．

```
cl-user(2): :cd
C:\Users\誠二\
cl-user(3): :cd Documents
C:\Users\誠二\Documents\
cl-user(4): (probe-file "seiji.txt")
#P"C:\\Users\\誠二\\Documents\\seiji.txt"
cl-user(5): (describe *)
#P"C:\\Users\\誠二\\Documents\\seiji.txt" is a structure of type pathname.  It has these
slots:
  excl::host           nil
  excl::device         "C"
  directory            (:absolute "Users" "誠二" "Documents")
  excl::name           "seiji"
  type                 "txt"
  excl::version        :unspecific
  namestring           "C:\\Users\\誠二\\Documents\\seiji.txt"
  excl::hash           nil
  excl::dir-namestring nil
  excl::plist          nil
cl-user(6): (pathname-directory **)
(:absolute "Users" "誠二" "Documents")
cl-user(7): (pathname-name ***)
"seiji"
```

ここで，星一つはREPLにおいて一つ前の結果を表し，星二つは二つ前の，星三つは三つ前の結果を表している．

上記ではWindowsファイルシステムであることを前提としたが，同じことをパスネームにより行うと次のようになる．

```
cl-user(8): *default-pathname-defaults*
#P"C:\\Users\\誠二\\Documents\\"
cl-user(9): (setq foo
                  (make-pathname
                    :name "seiji"
                    :type "txt"
                    :defaults *default-pathname-defaults*))
#P"C:\\Users\\誠二\\Documents\\seiji.txt"
cl-user(10): (probe-file foo)
#P"C:\\Users\\誠二\\Documents\\seiji.txt"
```

Debian 上のSBCLならば，たとえば次のようになる．

```
CL-USER(2): :cd
/home/seiji/
CL-USER(3): :cd ドキュメント
/home/seiji/ドキュメント/
CL-USER(4): (probe-file "seiji.txt")

#P"/home/seiji/ドキュメント/seiji.txt"
CL-USER(5): (describe *)

#P"/home/seiji/ドキュメント/seiji.txt"
  [structure-object]

Slots with :INSTANCE allocation:
  HOST       = #<SB-IMPL::UNIX-HOST {10002A1453}>
  DEVICE     = NIL
  DIRECTORY  = (:ABSOLUTE "home" "seiji" "ドキュメント")
  NAME       = "seiji"
  TYPE       = "txt"
  VERSION    = NIL
CL-USER(6): (pathname-directory **)

(:ABSOLUTE "home" "seiji" "ドキュメント")
CL-USER(7): (pathname-name ***)

"seiji"
CL-USER(8): *default-pathname-defaults*

#P"/home/seiji/ドキュメント/"
CL-USER(9): (setq foo
                  (make-pathname
                    :name "seiji"
                    :type "txt"
                    :defaults *default-pathname-defaults*))
; in: SETQ FOO
;     (SETQ FOO
;             (MAKE-PATHNAME :NAME "seiji" :TYPE "txt" :DEFAULTS
;                            *DEFAULT-PATHNAME-DEFAULTS*))
;
; caught WARNING:
;   undefined variable: FOO
;
; compilation unit finished
;   Undefined variable:
;     FOO
;   caught 1 WARNING condition
```

```
#P"/home/seiji/ドキュメント/seiji.txt"
CL-USER(10): (probe-file foo)

#P"/home/seiji/ドキュメント/seiji.txt"
```

上記実施例9(Windows)と9(Debian)の記述が同じであることに留意されたい．どんな処理系であっても，パスネームの構成は次のようになっていて，そのスロット値を実際のファイルシステムに適応することで，パスネームとしてのコードの可搬性を確保している．

- ホスト　　Common Lisp処理系から見たマシン上のファイルシステム名称
- 装置　　ファイルを含むデバイス名
- ディレクトリ　フォルダやディレクトリ（複数）の連続
- 名前　　ファイル名
- 型　　ファイルタイプ
- バージョン　ファイルのバージョンであるが，今では明示的にバージョンを持つファイルシステムはほとんどない

make-pathnameは指定されていないスロット値については:defaultsキーワードパラメータの値から必要な値をとってくる．だからデフォールトのパスネームをうまく扱うことで，移植性の高いコードにすることができる．

またmerge-pathnamesでは，*default-pathname-defaults*の値がオプショナルパラメータであるdefaulsの既定値として用いられるので，*default-pathname-defaults*の値をしかるべき値に設定することで，ファイル名とファイルタイプのみを指定して次のように処理系に依存しないコードをプログラムできる．上がWindows上ACL，下がDebian上SBCLである．

```
cl-user(11): (probe-file (merge-pathnames "seiji.txt"))
#P"C:\\Users\\誠二\\Documents\\seiji.txt"

CL-USER(11): (probe-file (merge-pathnames "seiji.txt"))

#P"/home/seiji/ドキュメント/seiji.txt"
```

物理パスネームと論理パスネーム

これまで説明してきたパスネームは物理パスネームと呼ばれるものである．処理系の違いを吸収するとは言え，たとえばホスト名やディレクトリの各要素名を見ればそれがUnix/LinuxなのかWindowsなのかは推測が容易である．ここで論理パスネームなるものを定義して使うと更に処理系の差異を吸収できる．

試しに，ACLで次のようにしてみよう．残念ながらSBCLではこうはいかない．

```
cl-user(4): (logical-pathname-translations "SYS")
((#P";**;*.*" #P"C:\\acl100express\\"
  #S(excl::cached-translation-info :host nil :device nil :directory nil :name nil :type ...))
 (#P"**;*.*" #P"C:\\acl100express\\"
  #S(excl::cached-translation-info :host nil :device nil :directory nil :name nil :type
...)))
cl-user(5): (pprint *)
```

```
((#P";**;*.*" #P"C:\\acl100express\\"
  #S(excl::cached-translation-info :host nil
                                   :device nil
                                   :directory nil
                                   :name nil
                                   :type nil
                                   :version nil))
 (#P"**;*.*" #P"C:\\acl100express\\"
  #S(excl::cached-translation-info :host nil
                                   :device nil
                                   :directory nil
                                   :name nil
                                   :type nil
                                   :version nil)))
cl-user(6): (probe-file "SYS:alisp.dxl")
#P"C:\\acl100express\\alisp.dxl"
```

ここで「SYS」は論理ホストと呼ばれるもので，その定義は処理系依存であるが，定義してある場合はたいていはLisp処理系の置かれているディレクトリを指示するものである．同様にしてWindows用ACLでは次のようにすると，自分のホームディレクトリを論理ホストに設定できる．ここではホームの定義と同時に，物理パスネームのディレクトリDocumentsを論理パスネームのディレクトリDOCUMENTSに対応させた．

```
cl-user(11): (let ((home (user-homedir-pathname)))
               (setf (logical-pathname-translations "HOME")
                 `(("DOCUMENTS;**;*.*"
                    ,(namestring
                       (make-pathname
                         :directory (append (pathname-directory home)
                                            (list "Documents")
                                            (list :wild-inferiors))
                         :defaults home)))
                   ("**;*.*"
                    ,(namestring
                       (make-pathname
                         :directory (append (pathname-directory home)
                                            (list :wild-inferiors))
                         :defaults home))))))
(("DOCUMENTS;**;*.*" "C:\\Users\\誠二\\Documents\\**\\") ("**;*.*" "C:\\Users\\誠二\\**\\"))
cl-user(12): (probe-file "HOME:DOCUMENTS;seiji.txt")
#P"C:\\Users\\誠二\\Documents\\seiji.txt"
cl-user(13): (probe-file "HOME:wordnet.rdf")
#P"C:\\Users\\誠二\\wordnet.rdf"
cl-user(14): (probe-file "HOME:eBook;metadata.db")
#P"C:\\Users\\誠二\\eBook\\metadata.db"
```

ここで，user-homedir-pathnameはユーザのホームディレクトリをパスネームで返す関数である．「"**;*.*"」よりも「"DOCUMENTS;**;*.*"」を先に置いているのには理由がある．先頭からパターンマッチが行われるので，特殊なものを先にする必要があるのだ．

同様のことはSBCLでもできる．以下の例を見られたい．

```
CL-USER(1): (let ((home (user-homedir-pathname)))
              (setf (logical-pathname-translations "HOME")
                `(("DOCUMENTS;**;*.*"
                   ,(namestring
```

```
                    (make-pathname
                      :directory (append (pathname-directory home)
                                         (list "ドキュメント")
                                         (list :wild-inferiors))
                      :defaults home)))
                ("**;*.*"
                 ,(namestring
                    (make-pathname
                      :directory (append (pathname-directory home)
                                         (list :wild-inferiors))
                      :defaults home))))))
(("DOCUMENTS;**;*.*" "/home/seiji/ドキュメント/**/") ("**;*.*" "/home/seiji/**/"))
CL-USER(2): (probe-file "HOME:DOCUMENTS;seiji.txt")

#P"/home/seiji/ドキュメント/seiji.txt"
CL-USER(3): (probe-file "HOME:quicklisp;asdf.lisp")

#P"/home/seiji/quicklisp/asdf.lisp"
CL-USER(4): (probe-file "HOME:quicklisp.lisp")

#P"/home/seiji/quicklisp.lisp"
```

論理パスネーム表記の構文は以下のとおりである．

```
[《ホスト》：][;]{《ディレクトリ》}*[《名前》][.《タイプ》[.《バージョン》]]
```

「*」すなわち「:wild」はワイルドカードで，ファイル名やタイプ名に用いれば，任意のファイル名やタイプ名にマッチするし，ディレクトリ名に用いれば1個の任意のディレクトリ名にマッチする．「**」すなわち「:wild-inferiors」は多重ワイルドカードで任意個のディレクトリのエントリにマッチする．ただし，UNIXシステムの中には多重ワイルドカードをサポートしないものがある．Debian上のSBCLでは上記のように大丈夫なようだ．

ロード時のディレクトリをアプリのホームにする

アプリケーションの配布は通常あるディレクトリの階層構造を保ったまま配布されるが，そのような場合にはどこのディレクトリに配布されたアプリケーションをロードしてもそのまま実行できるのが望ましい．ロード時のディレクトリを取り出して利用する方法を以下に示す．

最初に次のような内容のファイルを作って，適当なディレクトリに配置しておく．*load-pathname*はload関数実行中にその第１引数に合致するパスネームがその値として束縛される．だからアプリケーションのルートとなるべきディレクトリのソースファイルに以下のようなコードを入れておいて，最初に必ずそれがロードされるようにしておけば，その後のアプリケーションのどこでも，定まった配布されるアプリのディレクトリ構造を前提として，可搬性の高いアプリケーションを開発することができる．

```
(defparameter *application-home*
  (make-pathname :name nil :type nil :defaults *load-pathname*))

(setf (logical-pathname-translations "APPLICATION-HOME")
  `(("BIN;*.*"
     ,(make-pathname
```

```
        :directory (append (pathname-directory *application-home*)
                           (list "bin"))
        :defaults *application-home*))
    ("SRC;**;*.*"
     ,(make-pathname
        :directory (append (pathname-directory *application-home*)
                           (list "src")
                           (list :wild-inferiors))
        :defaults *application-home*))
    ("DOC;**;*.*"
     ,(make-pathname
        :directory (append (pathname-directory *application-home*)
                           (list "doc")
                           (list :wild-inferiors))
        :defaults *application-home*))))
```

いまここでは，C:\acl90-smp.64\のディレクトリにおいたものとする．以下はその時の実施例である．

```
cl-user(2): (load "aphome.cl")
; Loading C:\acl90-smp.64\aphome.cl
t
cl-user(2): (probe-file "APPLICATION-HOME:BIN;bundle.exe")
#P"C:\\acl90-smp.64\\bin\\bundle.exe"
cl-user(3): (probe-file "APPLICATION-HOME:DOC;contents.htm")
#P"C:\\acl90-smp.64\\doc\\contents.htm"
cl-user(4): (probe-file "APPLICATION-HOME:SRC;asdf.lisp")
#P"C:\\acl90-smp.64\\src\\asdf.lisp"
```

19.3 入出力ストリーム

これまでも色々なところでREPLにおける出力ストリームとして(format t ...)という書式が用いられてきた．本節では改めてCommon Lispにおける入出力ストリームについて解説する．最初に，ストリームには入力用，出力用，双方向用（対話用）とその役割に応じて3種類あることを述べておく．

定義済み入出力ストリーム

Common Lispには以下のように，七つの組み込みの入出力ストリームがある．

- *standard-input*
  read関数やread-char関数など，多くの入力用関数においてオプショナルの入力ストリームの既定値として用いられる．REPLのトップレベルループにおいては，これに束縛されるストリームが入力ストリームとなる．
- *standard-output*
  print関数やwrite-char関数など，多くの出力用関数においてオプショナルの出力ストリームの既定値として用いられる．REPLのトップレベルループにおいては，これに束縛されるストリームが出力ストリームとなる．
- *error-output*
  エラーや警告の出力先ストリームである．たとえば，上記*standard-output*をファイルストリームにしても，このストリームの値を変えない限り，端末にエラー出力されるし，もしこれにファイルストリームを束縛すればそのファイルがいわゆるエラーログファイルとなる．
- *query-io*
  yes-or-no-p関数のように，直接ユーザにメッセージ出力して答えを得たいような場合に用いられる．これによって，ファイル入出力と質問応答用ストリームが分離できる．

- *debug-io*
  デバッグ時に対話的に用いられるためのストリームである．
- *terminal-io*
  ターミナルへの入出力ストリームであり，通常ならば出力はコンソールまたはターミナルウィンドウに出力されるし，入力はキーボードからの入力である．
- *trace-output*
  trace関数の出力先ストリームである．

これら七つのストリームの値は，最初Common Lisp処理系を立ち上げたとき，**標準初期ストリーム**になっている．それは通常の立ち上げであればコンソールまたはターミナルへの出力とキーボードからの入力である．

format関数の第1引数には出力用ストリームとしてストリーム以外にも `t` や `nil` を与えることができるが，その `t` は常に*standard-ouput*のことである．`nil` の場合にはどこにも出力されずにformat構文の結果がその関数の返値として返される．

with-open-fileを使う

上記の既存ストリームを使う限りにおいてはファイルの取り扱いは必要がない．しかしファイルとの入出力を行う場合にはファイルのオープン，クローズと結びついたストリームが必要となる．それには関数のopenとcloseが対応する．しかし，実際にはプログラムコード中でそれらはほとんど使われず，もっぱらwith-open-fileが利用される．何故with-open-fileかといえば，ファイル入出力処理中にプログラムが異常終了した場合でも，入出力バッファとか排他的処理のロックとかそういうものをすべてきれいにクリアして資源を開放してくれるからである．関数openの場合，もし入出力中に異常終了すると，明示的にそのストリームをユーザ側がcloseしてやらなければならない．with-open-fileを使えば，正常終了の場合でも異常終了の場合でもシステム側がちゃんと面倒を見てくれる．10.3節のコルーチンでもwith-open-fileを使ったが，ここで改めてwith-open-fileの引数などを説明する．最初は入力用ストリームの場合である．

```
(with-open-file (《入力ストリーム》《ファイル》
  [:direction :input]
  [:element-type《トランザクション型》]
  [:if-does-not-exists {:error | nil}]
  [:external-format《文字コード指定子》) ... )
```

ここで，《ファイル》はオープンされるべきファイルを指定する文字列，またはファイルパス，またはファイルストリームそのものである．:directionキーの値に:inputを与えたときが入力の指定となるが，もし:directionの入力がなければ既定値としてファイルオープンは入力用となる．:element-typeにて入力のトランザクション単位を決めるが，既定値としてcharacterであり，文字入力では省略してよい．バイナリ入力では通常の使い方ならば「`'(unsigned-byte 8)`」を指定する．:if-does-not-existsの既定値は:errorであり，その場合指定のファイルが見つからなければエラーとなるが，この値にnilを指定するとエラーとせずに，nilが返される．:external-formatにて入力ファイルの文字コードを指定するが，これも処理系ごとの指定方法となる．ACLではiac.htmの3.0 External formatsに説明があり，SBCLについてはSBCL Manual の10.1 External

Formatsに詳細な説明がある．UTF-8では:utf8，Shift-JISならば:shiftjisまたは:shift_jisを指定する．:932または:cp932というのはMS-Windows系でのShift JISのことである．何も指定しなければ:defaultであり，それは処理系依存の文字コード指定となる．次に出力用ストリームの場合である．

```
(with-open-file (《出力ストリーム》《ファイル》:direction :output
  [:element-type《トランザクション型》]
  [:if-exists {:error | :supersede | :overwrite | :append | :rename | nil}]
  [:external-format《文字コード指定子》) ... )
```

出力の場合には:directionキーの値として:outputを明示的に与えなければならない．:if-existsの既定値は:errorであり，もし同じ名前のファイルが存在していればエラーとなる．この値にnilを指定するとエラーとせずに，nilが返る．:supersedeではもし既存のファイルがあればそれが結局新しいファイルに置き換わるが，異常終了の時には古いファイルがなるべく壊されないように保存される．:overwriteが一見良さそうに見えるが，もし新しいファイルの方が古いファイルより長さが短い場合には，ファイルの末尾に古いファイルが残ったままになるので，通常の使い方では:supersedeを用いるべきである．:overwriteはfilepositionを使ってユーザがファイル内容を部分的に書き換えるような場合にのみ効果的である．:appendでは古いファイルの最後から新しい内容が付け加えられる．:renameでは古いファイルが保存され，指定のファイル名が新しく作成される．:external-formatにて出力ファイルの文字コードを指定する．

問題19.1 Common Lispにはファイルをコピーする関数がない．与えられたファイルをコピーする関数を定義せよ．文字単位でコピーするにはread-charとwrite-charを用いる．もし１行づつコピーするならread-lineとwrite-lineを用いる．

問題19.2 UTF-8とShif-JISでコードを変換するようなファイルコピー関数を定義し，日本語や特殊文字（特にハイフン，マイナス記号，EM-Dash，EN-Dashなど）についてその挙動を見よ．サイクリックにUTF-8->Shif-JIS->UTF-8やShif-JIS->UTF-8->Shift-JISとしたとき，元のファイルと同一にならなければならないが，どうだろうか？

その他のストリーム

13.2節の型階層において掲載されたCommon Lispの型階層からストリームに関するところを抜き出して以下に再掲する．

```
(t
 . . .
 (standard-object
  . . .
  (stream
   (string-stream)
   (file-stream)
   (concatenated-stream)
   (echo-stream)
   (synonym-stream)
   (two-way-stream))
  . . .
```

これまで述べたストリームは主にファイルストリームであった．with-open-fileからファイルストリームは作られるが，ストリームを引数とするwith-open-streamからもファイルストリームが作られる．複数の入力ストリームからそれらを一本につなげたようなconcatenated-streamをmake-concatenated-stream関数で作りだすことができる．入力ストリームと出力ストリームを引数に入力ストリームからの入力を自動的に出力ストリームに送るエコーストリームをmake-echo-streamで作ることができる．make-synonym-streamは一つのシンボルを引数にしてシノニムストリームを返す．その使い方はHyperSpecを参照されたい．make-two-way-streamは入力ストリームと出力ストリームの二つの引数から双方向ストリームを作りだす．以下ではstring-streamについて説明する．

関数make-string-output-streamによって出力用string-streamが作られ，これへの出力は蓄積されて最後にその結果をget-output-stream-stringで取り出すことができるが，それよりはwith-output-to-stringマクロを用いるのが普通である．その呼び出しの一般形式は次のとおり．

```
(with-output-to-string (《変数》[《フィルポインタ文字列》[:element-type《型》]])
  {《宣言》}* {《書式》}* )
```

《フィルポインタ文字列》は通常フィルポインタを持つ空の文字列である．この《変数》には《フィルポインタ文字列》から生成されるstring-streamが束縛される．この書式の中でその《変数》に対する出力が行われると，その《フィルポインタ文字列》にちょうどvector-push-extendを用いているかのように出力文字が蓄積される．以下の実施例を見られたい．

```
cl-user(2): (setq foo
                  (make-array '(0) :element-type 'character
                                   :fill-pointer 0 :adjustable t))
""
cl-user(3): (with-output-to-string (string-stream foo)
              (format string-stream "Hello, World!"))
nil
cl-user(4): foo
"Hello, World!"
```

《フィルポインタ文字列》を省略すると，内部で空のフィルポインタ文字列が作られてそれが用いられ，そのとき蓄積された文字列がこの全体の書式の結果として返される．以下は15.7節のコーパス検索事例において提示された例である．

```
cl-user(9): (with-output-to-string (string-stream)
              (format string-stream "Hello, World!"))
"Hello, World!"
```

with-output-to-stringに対応する入力がwith-input-from-stringである．これを使うと文字列を与えてそれからread関数を用いてリスプオブジェクトを容易に得ることができる．14.3節に実例があるので，参照されたい．

第20章 パッケージとモジュール

5.4節で説明したように，シンボルには印字名がある．Lispの読込み関数(関数read)は新しいシンボルの印字名を読むとその印字名を持つシンボルオブジェクトを作りだし，システムに登録する．二度目からの読み込みではそのシステムに登録されたシンボルオブジェクトを返す．だからシンボルは常にLispのある環境において物理的に唯一なオブジェクトである．シンボル印字名からシンボルオブジェクトへの写像は一種の辞書引きと思えばよい．

MacLispにシンボルを作る関数internはあったが，シンボルが登録される辞書，つまりシンボルの名前空間は一つだけであった．Common Lispではシンボルをインターンする別々の空間をいくつでも好きなだけ持てる．それがパッケージである．自分一人で何かを開発する場合には，パッケージがなくてもなんとかなるかもしれないが，チームでシステム開発するとか，独立した複数のプログラマーが開発したプログラムを組み合わそうとすると，名前の衝突を防ぐためにパッケージが必要になる．これまでも本書籍では関数internを使ったりパッケージという用語を使ってきたが，本格的なパッケージの説明はしてこなかった．第11章のschemeの説明では，PAIPにおいてはscheme-macroと称していたシンボルと，scheme-macro-expandと称していたシンボルを，schemeパッケージなるものを導入してscheme:macroとscheme:macro-expandという名前に変更したが，その理由はmacroとmacro-expandというシンボルがいかにも名前の衝突を起こしそうだったからだ（実際にはこれまでのところはそうはなっていない）．世界で唯一のパッケージ名を使えば，その中のシンボルについてどんな名前を使おうが名前の衝突を気にする必要はない．本章の前半では，このパッケージの使い方について学ぶ．

20.1 パッケージの定義方法と使い方

手元にあるCommon Lispで今どういうパッケージが存在しているかを知るには，list-all-packagesを使う．パッケージ名を指定してそれが有るか無いかを調べるにはfind-packageを使う．立ち上げた直後のAllegro Common Lisp free Express with IDEでは次のようになる．

```
cg-user(1): (list-all-packages)
(#<The acl-socket package> #<The aclmop package> #<The cg-dde-utils package> #<The cg.base
package> #<The cg.bitmap-pane package>
 #<The cg.bitmap-pane.clipboard package> #<The cg.bitmap-stream package> #<The cg.button
package> #<The cg.calendar package>
 #<The cg.caret package> ...)
cg-user(2): (find-package :cl)
#<The common-lisp package>
cg-user(3): (find-package :cl-user)
#<The common-lisp-user package>
```

「cl」はcommon-lispパッケージの別名（ニックネーム）であり，「cl-user」はcommon-lisp-userパッケージの別名である．

カレントパッケージと他パッケージの利用

Common Lispの実行環境においては，必ず一つのパッケージ（シンボルの名前空間）の中にいる．それをカレントパッケージという．カレントパッケージが何かはグローバル変数 *package* の値を見れば分かる．通常Common Lispを立ち上げたとき，REPLにおけるカレントパッケージは common-lisp-user（別名cl-user）である（Allegro Common LispのIDE環境ではcg-user）．この環境中ではユーザは common-lisp パッケージにおいて移出(export) されているシンボル（関数名や大域変数名）をあたかもその環境に存在しているかのごとくパッケージ指定なしに利用できる．しかしこれは一般的にいつでもそうだというわけではない．独自パッケージを定義しても common-lisp パッケージをuseしないかぎり，Common Lispの機能をパッケージ名なしでは利用できない．以下の実施例を見られたい．9行目でmy-own-packageを定義し，10行目でそれをカレントパッケージとしたが，clというパッケージ名なしではCommon Lispの機能を使えずに，すぐにエラーを起こしている．

```
cl-user(1): *package*
#<The common-lisp-user package>
cl-user(2): (describe *package*)
#<The common-lisp-user package> is the package named common-lisp-user.
It has nicknames: cl-user, user.
It has 0 external symbols, 1 internal symbol, and shadows 0 names.
It uses packages: common-lisp, excl.
cl-user(3): (loop for x being each external-symbol of :common-lisp-user
                  count t)
0
cl-user(4): (loop for x being each present-symbol of :common-lisp-user
                  count t)
8
cl-user(5): (+ 1 2)
3
cl-user(6): (cons 'a '(b c))
(a b c)
cl-user(7): (describe *package*)
#<The common-lisp-user package> is the package named common-lisp-user.
It has nicknames: cl-user, user.
It has 0 external symbols, 11 internal symbols, and shadows 0 names.
It uses packages: common-lisp, excl.
cl-user(8): (loop for x being each present-symbol of :common-lisp-user
                  count t)
11
cl-user(9): (defpackage my-own-package (:use))
#<The my-own-package package>
cl-user(10): (in-package :my-own-package)
#<The my-own-package package>
my-own-package(11): *package*
Error: Attempt to take the value of the unbound variable *package*'.
  [condition type: unbound-variable]

Restart actions (select using :continue):
 0: Try evaluating *package* again.
 1: Use the value of common-lisp:*package* instead.
 2: Set the symbol-value of *package* and use its value.
 3: Use a value without setting *package*.
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1] my-own-package(12): :continue 4
my-own-package(13): cl:*package*
```

```
#<The my-own-package package>
my-own-package(14): (+ 1 2)
Error: attempt to call `+' which is an undefined function.
  [condition type: undefined-function]

Restart actions (select using :continue):
 0: Try calling + again.
 1: Try calling common-lisp:+ instead.
 2: Return a value instead of calling +.
 3: Try calling a function other than +.
 4: Setf the symbol-function of + and call it again.
 5: Return to Top Level (an "abort" restart).
 6: Abort entirely from this (lisp) process.
[1] my-own-package(15): :continue 5
my-own-package(16): (cl:+ 1 2)
3
```

上記実施例9行目でわざわざ(defpackage my-own-package (:use))としたのには訳がある．defpackageはパッケージ定義のためのマクロだが，普通に(defpackage 《パッケージ名》)とすると，Allegro Common Lispでは(defpackage my-own-package (:use :common-lisp))と同じことになるからだ．つまりdefpackageにおいて利用するパッケージを何も指定しないと，それは暗黙の裡にcommon-lispパッケージを指定したことになる（SBCLではこれとは異なり，(defpackage my-own-package)でcommon-lispパッケージが指定されることはない）．逆に，common-lisp以外のパッケージを利用する場合には，ほとんどの場合(defpackage my-own-package (:use :common-lisp 《その他のパッケージ名》 ...))と明示的にcommon-lispパッケージをuseしなければならない．どんなパッケージにおいてもCommon Lispの諸機能はデフォールトとして使いたいというのが普通だからだ．わざとcommon-lispパッケージを使用しない例が13.3節で説明したXML Schema Datatypeの場合である．この例では実際にdeftypeによるXMLデータタイプの定義の前に次のようにパッケージ定義がされているものとした．(:use)では意図的にcommon-lispパッケージを利用しないと宣言している．

```
(cl:defpackage :xsd
  (:nicknames :xs)
  (:use ) ; supressing using common lisp package
  (:export "string" "boolean" "decimal" "float" "double" "dateTime"
           "time" "date" "gYearMonth" "gYear" "gMonthDay" "gDay"
           "gMonth" "hexBinary" "base64Binary" "anyURI"
           "normallizedString" "token" "language" "NMTOKEN" "Name"
           "NCName" "integer" "nonPositiveInteger" "negativeInteger"
           "long" "int" "short" "byte" "nonNegativeInteger"
           "unsignedLong" "unsignedInt" "unsignedShort" "unsignedByte"
           "positiveInteger" "simpleType" "anySimpleType" "true"
           "false" "duration" "duration-year" "duration-month"
           "duration-day" "duration-hour" "duration-minute"
           "duration-second")
  (:documentation "http://www.w3.org/2001/XMLSchema#"))
```

REPL中やプログラムコードの途中で，一時的に別パッケージで移出されたシンボルをあたかも自分のシンボルのように利用したいというときは，関数 use-package を使う．その詳細な実例は5.4節の移出シンボル実施例にあるので参照されたい．

パッケージの最も一般的な使用方法は，プログラムファイルの最初でdefpackageでパッケージを定義して，すぐにin-packageで自分のカレントパッケージとしてその中で自前の大域変数や関数を定義するというやり方である．そして，関数としてのuse-packageやexport，importなどは使わずにdefpackage中で定義する．そうすることでファイルのコンパイル時とそうでないときの挙動の違いを極力抑えることができる．これまでそのような本格的なプログラムコード例を示してこなかった．第11章のSchemeプログラムでさえ，パッケージschemeを定義したがin-packageは使わずに過ごした．ここで，そのちょうどよい例として，末尾再帰なしの Scheme 実装でdefpackageを用いた場合の典型的なコード例を以下に示そう．ただし，オリジナルに忠実というよりは，第11章の内容に沿ったコードになっていることをお断りしておく．

```
(defpackage scheme
  (:shadow #:quote #:if #:lambda #:write)
  (:use :common-lisp)
  (:export #:scheme #:macro #:macro-expand #:begin #:set!
           #:quote #:null? #:eq? #:equal? #:if #:lambda
           #:write #:display #:newline)
  (:documentation "Scheme implemented by Common Lisp."))

(in-package :scheme)

(defparameter *scheme-procs*
  '(+ - * / = < > <= >= cons car cdr not append list read member
    (null? null) (eq? eq) (equal? equal)
    (eqv? eql) (write prin1) (display princ)
    (newline terpri)))

(defun length=1 (x)
  "Is x a list of length 1?"
  (and (consp x) (null (cdr x))))

(defun last1 (list)
  "Return the last element (not last cons cell) of list"
  (first (last list)))

(defun rest2 (x)
  "The rest of a list after the first TWO elements."
  (rest (rest x)))

(defun starts-with (list x)
  "Is x a list whose first element is x?"
  (and (consp list) (eql (first list) x)))

(defun maybe-add (op exps &optional if-nil)
  "For example, (maybe-add 'and exps t) returns
  t if exps is nil, exps if there is only one,
  and (and exp1 exp2...) if there are several exps."
  (cond ((null exps) if-nil)
        ((length=1 exps) (first exps))
        (t (cons op exps))))

(defun scheme (&optional x)
  "A Scheme read-eval-print loop (using interp)"
  ;; Modified by norvig Jun 11 96 to handle optional argument
  ;; instead of always going into a loop.
  (init-scheme-interp)
```

```
  (if x
      (interp x nil)
    (loop (format t "~&==> ")
          (force-output)
          (cl:print (interp (cl:read) nil)))))

(defun interp (x &optional env)
  "Interpret (evaluate) the expression <<x>> in the environment <<env>>.
   This version handles macros."
  (cond
   ((symbolp x) (get-var x env))
   ((atom x) x)
   ((case (first x)
      (quote  (second x))
      (begin  (last1 (mapcar #'(lambda (y) (interp y env))
                             (rest x))))
      (set!   (set-var! (second x) (interp (third x) env) env))
      (if     (if (interp (second x) env)
                  (interp (third x) env)
                (interp (fourth x) env)))
      (lambda (let ((parms (second x))
                    (code (maybe-add 'begin (rest2 x))))
                #'(lambda (&rest args)
                    (interp code (extend-env parms args env)))))
      (otherwise ;; a procedure application
       (apply (interp (first x) env)
              (mapcar #'(lambda (v) (interp v env)) (rest x))))))))

(defun init-scheme-interp ()
  "Initialize the scheme interpreter with some global variables."
  ;; Define Scheme procedures as CL functions:
  (mapc #'init-scheme-proc *scheme-procs*)
  ;; Define the boolean `constants'. Unfortunately, this won't
  ;; stop someone from saying: (set! t nil)
  (set-global-var! t t)
  (set-global-var! nil nil))

(defun init-scheme-proc (f)
  "Define a Scheme procedure as a corresponding CL function."
  (if (listp f)
      (set-global-var! (first f) (symbol-function (second f)))
      (set-global-var! f (symbol-function f))))

(defun set-var! (var val env)
  "Set a variable to a value, in the given or global environment."
  (if (assoc var env)
      (setf (second (assoc var env)) val)
    (set-global-var! var val))
  val)

(defun get-var (var env)
  "Get the value of a variable, from the given or global environment."
  (if (assoc var env)
      (second (assoc var env))
    (get-global-var var)))

(defun extend-env (vars vals env)
  "Add some variables and values to an environment."
  (nconc (mapcar #'list vars vals) env))
```

```
(defun set-global-var! (var val)
  (setf (get var 'global-val) val))

(defun get-global-var (var)
  (let* ((default "unbound")
         (val (get var 'global-val default)))
    (if (eq val default)
        (error "Unbound scheme variable: ~a" var)
        val)))
```

(defpackage ...)の三行目にある(:use :common-lisp)は，ACLではいらないが，SBCLでは必要だ．ポータビリティを考えれば丁寧に書いた方がよい．同様なことは関数scheme中の(force-output)にも言える．ACLではあってもなくてもよいが，SBCLでは必須である．

(:export ...)には何を書くのか？

読者の中には，先のXML Schema Datatypeのところではexportで文字列指定だが，schemeのexportではアンインターンされたシンボルを指定したことの違いに疑問を持った方がいるかもしれない．これには訳がある．既に承知のように，Common Lispはシンボルについて通常は大文字と小文字を区別しない．厳密に言うと，小文字として入力したシンボル印字名は通常は大文字印字名に自動的に変換される．これが readtable case が :upcase であることの効果である．しかし標準ではそうであってもCommon Lispが小文字のシンボルが持てないわけではない．文字をエスケープしたりシンボル名を縦棒で括ったりすれば，小文字からなるシンボルも作って利用することはできる．SBCLでは以下の通りであるし，ACLならば（mlispではなく）ANSI lispで以下の通りである（mlispなら小文字を最初から認識する）．

```
CL-USER(1): 'abc
ABC
CL-USER(2): '\a\b\c
|abc|
CL-USER(3): '|abc|
|abc|
CL-USER(4): '誠二
誠二
CL-USER(5):
```

さて話を戻して，defpackageのexportで書くべきものは，文字列指示子(string designator)ということになっている．すなわち，文字列でもいいし，シンボルでもよい．文字列の場合には，それを印字名とするシンボルがそのパッケージからexportされる．このとき小文字大文字の変換は起こらない．XML Schema Datatypeでは小文字大文字の変換なしに，指定した通りの型の名前にしたかったので，文字列を与えた．Schemeプログラムの場合にはACLのmlispでもANSI lispでも，あるいはSBCLでも，共通に利用可能なプログラムコードにしたかった．つまりexportでシンボルを与えたとき，それはmlispならば小文字大文字の変換は起こらずにそのままシンボルとなるが，ANSI lispあるいはSBCLではreadtable caseの値が:upcaseである限りはすべてのシンボルは大文字となる．しかしそれでもそのファイル中のコードとしては，わざとエスケープなどでシンボルに小文字を指定しない限りではそれで問題ないわけだ．ACL mlispでは小文字で指定すれば

その通り小文字のシンボルとなるが，それ以外のCommon Lispでは大文字シンボルになる．両方にポータビリティのあるコードにするには，以下の何点かに気を付ければよい．

1. 同一のシンボルについてファイル中で，あるときには大文字を使ったり，あるときには小文字を使ったりしない．特に t と nil には注意が必要である．昔のコードには関数名は小文字で表記しても，コード中に t ではなく T を使ったものが多くあるが，これはmlispでは動かない．
2. スペルは同一でも大文字シンボルと小文字シンボルの両方を別のものとして利用するようなコードにしない．
3. スペルは同一でも大文字シンボルと小文字シンボルの両方を同じものとして利用するようなコードにしない．たとえば，同じシンボルについて小文字のみで書いたりCamelケースで書いたりすると，そのコードはmlispでは通用しない．
4. defpackage中でエクスポートされるシンボルには特に理由がないかぎりは文字列を使用せずに，アンインターンされたシンボルとする．

アンインターンされたシンボルとは，make-symbolで作られるような，オブジェクトとしてはシンボルではあるが，まだどのパッケージにも登録されていないようなシンボルのことである．

```
CL-USER(5): (make-symbol "FOO")
#:FOO
CL-USER(6): (intern "BAR")
BAR
NIL
CL-USER(7): (describe **)
#:FOO is a NEW SYMBOL.
  It is unbound.
  It is uninterned.
CL-USER(8): (describe **)
BAR is a NEW SYMBOL.
  It is unbound.
  It is INTERNAL in the COMMON-LISP-USER package.
```

アンインターン・シンボルはmake-symbolで作られるが，直接 #:《シンボル印字名》としても作りだすことができる．

```
CL-USER(11): '#:BAR
#:BAR
CL-USER(12): 'BAR
BAR
CL-USER(13): '#:BII
#:BII
```

実はdefpackage文中でexportするときにそこに書かれるものはインターンされたシンボルやキーワードシンボルでもコードとしてはちゃんと作動し何ら問題ないが，さてファイル中の最初にdefpackageがあってそこでシンボル形式で書いたとき，そのシンボルは一体どこに存在するのかという（必要がないかもしれないが余計な）心配がつきまとう．それがアンインターン・シンボルにする理由である．あるパッケージ中であるシンボルについて言及し，すでにそのパッケージ中に存在していることが明瞭な場合には，わざわざアンインターン・シンボルを使う理由はない．

```
CL-USER(1): (defpackage my-package (:use :common-lisp))
#<The MY-PACKAGE package>
```

```
CL-USER(2): (in-package :my-package)
#<The MY-PACKAGE package>
MY-PACKAGE(3): (cons 'foo '(bar bii))
(FOO BAR BII)
MY-PACKAGE(4): (export '(foo bar bii))
T
MY-PACKAGE(5): (describe 'foo)
FOO is a NEW SYMBOL.
  It is unbound.
  It is EXTERNAL in the MY-PACKAGE package.
```

シャドウとシャドウイング・インポート

具体的な例題を通じて，パッケージにおけるシャドウとシャドウイング・インポートについて説明する．

ストリームの下位クラスにはファイルからの入出力のためのクラスfile-streamがある．しかしこのクラスに入力ファイル中のどの位置にあるかを返すfile-position関数はあっても，現在読み込み中の行の行頭のファイル・ポジションを知りたいと思ってもそのような機能は用意されていない．そこで読み込み中の行の行頭のファイル・ポジションを常に記録して，要求に応じてそれを返すことができるような入出力クラスline-streamをfile-streamの下位クラスとして定義することにする．これを新しくline-readerパッケージなるものを定義して，その中でストリーム関連シンボルを次のように定義したとしよう．

```
cl-user(1): (defpackage :line-reader
              (:nicknames :line)
              (:use common-lisp)
              (:export #:stream #:with-open-file #:read-line #:read))
#<The line-reader package>
```

ここでは一見なんの問題もないように見える．しかし，行数のカウントと行頭のファイル・ポジションを保存できるように，line-readerパッケージ中のクラスline:streamを実際に次のように定義しようとするとたちまちエラーとなる．

```
cl-user(2): (in-package :line)
#<The line-reader package>
line(3): (defclass stream (file-stream)
           ((line-count :initform 0 :accessor line-count)
            (line-position :initform 0 :accessor line-position)))
Error: Attempt to make a type definition for the name stream as a
       defclass.  This name is in the common-lisp package and
       redefining it is a violation for portable programs.  Replacing
       the current definition of #<standard-class stream> may be
       dangerous.  The package common-lisp has
       excl:package-definition-lock set, which causes the system to
       signal this violation.
  [condition type: package-locked-error]

Restart actions (select using :continue):
 0: Set the type definition of the name stream anyway.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] line(4):
```

すなわち，common lispパッケージに定義済みのクラスを再定義しようとしたというエラーである．メッセージの表現こそ違えども，基本的にはSBCLでも同様なエラーとなる．なぜならば，最初にdefpackageにてcommon-lispパッケージをuseすると，common-lispパッケージ中の移出(export)されたシンボルがline-readerパッケージ中に取り込まれた状態(inherited)になり，ここのdefpackageコードによってstreamとかwith-open-fileとかのシンボルがさらにline-readerから移出された状態になったが，その実態としてはcommon-lisp中のstreamシンボルとline-reader中のstreamシンボルは同一のものである．そこで既存のCommon Lispクラスを再定義しようとしたことになったわけだ．こういうとき，すなわち全体としてはuseするが一部のシンボルだけ取り込みたくないというような場合に，**シンボルのシャドウイング**を行う．

```
cl-user(1): (defpackage :line-reader
              (:nicknames :line)
              (:shadow #:stream #:with-open-file #:read-line #:read)
              (:use common-lisp)
              (:export #:stream #:with-open-file #:read-line #:read))
#<The line-reader package>
cl-user(2): (in-package :line)
#<The line-reader package>
line(3): (defclass stream (file-stream)
           ((line-count :initform 0 :accessor line-count)
            (line-position :initform 0 :accessor line-position)))
#<standard-class stream>
```

シャドウイング宣言することで，そのパッケージ中に確かにそのシンボルが存在することになり，use-packageしても該当シンボルについては取り込まれない．だから上記三行目のdefclass宣言ではcommon-lispパッケージ中のcl:file-streamの下位クラスとして，line:streamクラスを定義したことになる．

それとは反対に，全体としてはuseしないがその一部だけを取り込みたいときに:import-fromを使うことができる．しかし，パッケージ名付きのシンボルにすれば何もわざわざ:import-fromする必要もない．どうしても:import-fromを使いたいというのは，あるパッケージの一部だけ取り込んで，しかもそれをあたかも自前のシンボルであるかのように移出したいときだけだ．本当に必要な場合があるのは**シャドウイング・インポート**だ．複数のパッケージ中で同じ印字名のシンボルを別々に定義したとしよう．第三のパッケージ中で両者のパッケージをuseしようとすると両方のシンボルが衝突することになる．そういう場合にどちらのパッケージから強制的にインポートするかを明示的に指示する必要がある（両方からインポートしないならただのシャドウイングでよい）．そういう場合に:shadowing-import-fromを使う．たとえば，アプリのプログラム中で全体的にはもちろんcommon-lispパッケージを:useするが，streamとかwith-open-fileなどのみをline-readerパッケージから取り込みたいときには:shadowing-import-fromを使えばよい．

20.2 モジュールとは？

名前空間とかモジュールという言葉は非常にあいまいで，言語横断的になんとなく共通の意味を持ちながらも，厳密には言語によって異なるものであったりする．Common Lispの名前空間は変数のスコープとは別物で，シンボルの名前からそのオブジェクトへの写像の空間であるときっちり言うことができる．これは比較的分かりやすい．シンボルの名前についての辞書が複数あって同じ印字名でありながらシステム中に複数

の異なるシンボルオブジェクトが存在できるが，それらは名前空間で区別される．一方，モジュールという言葉は他の言語と同様にCommon Lispでも明確な定義はないが，あえていえばグローバル変数*modules*に登録されるプログラムの単位ということができよう．それは名前空間とは直交した概念で，たとえば複数のモジュールに亘って同一の名前空間であってもよいし，一つのモジュール中で複数の名前空間があってもよい．大体においては，Common Lispでも一つのモジュールは一つのファイルに対応するが，実際には複数のファイルで構成される一つのモジュールであることもできる．すべてはシステム開発者の意図による．しかし，モジュールを*modules*に登録されるプログラムの単位とすると，それは結局requireとprovideをどう使うかという話に還元されることになる．本節ではその立場からCommon Lispのモジュール化について議論する．

試しに，各自のCommon Lisp処理系で*modules*の値がどうなっているか見てみよう．以下はACL mlispの場合である．

```
cl-user(1): *modules*
("efft-932-base" "ef-932" "develenv" "uri" "mcombin" "constructor"
 "loop" "socket" "sock" "disasm" ...)
cl-user(2): (pprint *modules*)

("efft-932-base" "ef-932" "develenv" "uri" "mcombin" "constructor"
 "loop" "socket" "sock" "disasm" "disx86-64" "inspect" "prof-s" "prof"
 "trace" "xref-s" "xref" "scm" "emacs" "eli" "sock-s" "lze" "lep"
 "proc2common" "process" "compftype" "defftype" "foreign"
 "foreign-functions" "defsystem" "defsys" "defsys-s" "proc2-s"
 "tpl-proc" "debug" "tpl-debug" "srecord" "regexp" "safeseq" "seq2"
 "list2" "rsa" "foreign-s" "compiler" "compdecl" "compiler-s" "excl"
 "pprint" "toplevel" "cond" "frame" "xframe" "ics" "ef-utf8" "ef-void"
 "ef-latin1" "ef-e-crcrlf" "ef-e-crlf" "ef-crcrlf" "ef-crlf" "smpdefs")
cl-user(3):
```

ここに登録されている各要素はモジュール名と呼ばれるが，関数provideは引数に与えられたモジュール名をこの*modules*に追加する．

```
cl-user(3): (provide :my-module1)
nil
cl-user(4): *modules*
("my-module1" "efft-932-base" "ef-932" "develenv" "uri" "mcombin"
 "constructor" "loop" "socket" "sock" ...)
```

一方，関数requireは与えられたモジュール名がこのグローバル変数*modules*にあるかどうかを調べ，すでにあればそれはシステムに該当モジュールが存在しているものとしてなにもしないが，なければそのモジュールをロードする．

```
cl-user(5): (require :my-module1)
nil
cl-user(6): (require :asdf)
; Fast loading C:\acl90-smp.64\code\asdf.005
;;; Installing asdf patch, version 5.
t
```

システムライブラリとして位置づけされているようなモジュールなら，requireにモジュール名を与えるだけであとは処理系がそれを見つけてロードしてくれるが，自分の開発したモジュールであればそのモジュールのありかをrequireの第2引数にパスとして与えなければならない．しかたがないとは言え，これではあまりうれしくない．自分の開発したモジュールも自分の環境中ではシステムライブラリ同様に扱いたいのが人情というものだ．幸いなことに，ACLではrequireされたモジュールについてどこに探索にいくのかは sys:*require-search-list*に書いてあることになっている．だから，これらのパスのいずれかに潜り込ませるか，あるいは自分専用のライブラリのパスをここに追加すればよい．

```
cl-user(9): sys:*require-search-list*
(:first #P"sys:;code;.fasl" #P"sys:;private;.fasl"
 #P"sys:;private;crypto;.fasl" #P"sys:;clos;.fasl" #P"sys:;sys;.fasl"
 #P"sys:;winapi;.fasl" #P"sys:;ole;.fasl" #P"sys:;zlib;.fasl"
 #P"sys:;uri;.fasl" ...)
cl-user(10): (pprint *)

(:first #P"sys:;code;.fasl" #P"sys:;private;.fasl"
 #P"sys:;private;crypto;.fasl" #P"sys:;clos;.fasl" #P"sys:;sys;.fasl"
 #P"sys:;winapi;.fasl" #P"sys:;ole;.fasl" #P"sys:;zlib;.fasl"
 #P"sys:;uri;.fasl" #P"sys:;tester;.fasl" #P"sys:;cl-env;.fasl"
 "sys:;net-xml-generator;.fasl" "sys:;xmlutils;.fasl"
 "sys:;aserve;.fasl" (:lib-bundle #P"sys:;code;.fasl")
 (:lib-bundle #P"sys:;private;.fasl")
 (:lib-bundle #P"sys:;private;crypto;.fasl")
 (:lib-bundle #P"sys:;clos;.fasl") (:lib-bundle #P"sys:;sys;.fasl")
 (:lib-bundle #P"sys:;winapi;.fasl") (:lib-bundle #P"sys:;ole;.fasl")
 (:lib-bundle #P"sys:;zlib;.fasl") (:lib-bundle #P"sys:;uri;.fasl")
 (:lib-bundle #P"sys:;tester;.fasl")
 (:lib-bundle #P"sys:;cl-env;.fasl"))
```

一方SBCLでは，requireにパスが与えられなかったときは，*MODULE-PROVIDER-FUNCTIONS*にバインドされている関数を順に呼び出すことになっている．現在手元にあるSBCLを確認すると次のようになっていた．

```
seiji@wordonto:~$ sbcl
This is SBCL 1.2.4.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
CL-USER(1): (apropos "MODULE-PROVIDER-FUNCTIONS")

*MODULE-PROVIDER-FUNCTIONS* (bound)
CL-USER(2): *MODULE-PROVIDER-FUNCTIONS*

(ASDF/OPERATE:MODULE-PROVIDE-ASDF SB-IMPL::MODULE-PROVIDE-CONTRIB)
CL-USER(3):
```

だからDebianだったらシステムのライブラリ・リポジトリである/usr/lib/sbcl/contribに自分のライブラリを潜り込ませるか，自前のライブラリ用のプロバイダー関数を定義してここに追加すればよい．

ファイル中にrequireはprovideの後，大体はdefpackageの前に書く．defpackageで他のパッケージをuseするときは，その前でそのモジュールをrequireをしておかねばならないし，provideとrequireを逆にすると次に述べるように循環する危険性があるからだ．しかもファイル中にrequireを書くときは，今ではeval-whenでラップしなければならない．*CLtL1*の仕様ではrequireはコンパイル時にも読み込んでくれたが，*CLtL2*になってからはそうではなくなったからだ．同様に*CLtL1*ではコンパイル時に処理したのに*CLtL2*ではしなくなったものにパッケージ関連の関数がある．前述のようにパッケージ関連の関数は使わずにdefpackageを使うことで対処できるが，requireにはそのようなものがない．ANSIではdefsystemのようなシステム定義機能を導入しようとしてrequireをdeprecatedとしたが，時間切れでdefsystemが入らなかったらしい．後述のASDFを使わない場合にはprovide，requireで対処するしかない．

モジュール化にあたってprovide宣言をファイルの最初に書くのか最後に書くのかは，常に議論の対象になっている．私自身のことで言えば，かっては最後に置いていたが，いまは最初に置くようにしている．問題はモジュールＡはモジュールＢを必要とし，モジュールＢはモジュールＣを必要としているが，そのモジュールＣはまたモジュールＡを必要としているといったように，依存関係がループするような場合である．おおざっぱに言えば，もしrequire宣言が最初にあってprovide宣言が最後にあるといつまでもrequireし続けることになる．もし最初にprovide宣言を置いてそのあとにrequire宣言を置くようにすれば，依存関係に循環があっても少なくとも永遠にrequireし続けるということはなくなる．ただし，パッケージやマクロの依存関係についてはこの方法では場合によっては問題となる．関数の依存関係は警告は出力されるかもしれないがそれでロードが止まるようなことはなく，最後につじつまが合えばよいが，パッケージの依存関係とマクロについてはコンパイル時でもロード時でも，必要とされるその時に必要だからだ．グローバル変数の依存関係は少なくともdefvarである限りでは，コンパイル時でもロード時でも定義時には問題とはならない．複雑なモジュール群になると，パッケージとマクロについては別建てのファイルに纏めるということが必要になるかも知れない．

20.3 ASDF3とquicklisp

Common Lispで他言語のIDEと同じようなものはないのかという要求が，一般ユーザの間で常に潜在的にある．Slimeを使うことでかなりの不満は解消できるが（付2.4参照），それでもプログラム開発環境として，Allegro Common LispのIDEによる開発環境やLispWorksのIDEによる開発環境と比べると，プロジェクト管理と検索機能に劣るところがあることは否めない．付1.3ではAllegro Common LispのIDEの使い方について紹介したが，本節ではSBCLでも有効に使えるASDFとquicklispについて紹介する．

ASDFはシステム定義とビルドのツールである．Common Lispには昔からいくつかのシステムビルド・ツールがあった．極端に言えば，compile-fileとloadを駆使すれば，自分で個別アプリに専用のシステムビルドツールを書くこともできた．ASDFは決して使いやすいものではないが，最近急速に普及して支配的になったquicklispのベースツールとなったために，今日ではASDFが事実上の標準Common Lispビルド・ツールといえる．もし自分のシステムをquicklispに載せようと思えば，ASDFを使いこなすことができなければならな

い．Allegro Common LispにおいてもIDEを使う限りではASDFやquicklispのお世話になることはないが，IDEなしの場合や，開発システムを他の処理系まで含めて配布したいと思えば，今日ではASDFとquicklispに依存せざるを得ない．

インストール直後ではSBCLでもACLでもasdfはロードされていないが，asdfを要求(require)することでライブラリにあるuiopとasdfをロードすることができる．以下はSBCLの場合だが，ACLでもasdfを要求してuiopとasdfがロードされることは同じである．

```
seiji@agents:~$ sbcl
This is SBCL 1.2.4.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* *modules*

NIL
* (require :asdf)

("ASDF" "asdf" "UIOP" "uiop")
* *modules*

("ASDF" "asdf" "UIOP" "uiop")
*
```

ファイルのモジュール化とシステム定義

もっとも簡単なASDFの利用例として，先に掲げた末尾再帰なしのScheme実装コードを通常のマナーに倣って，package.lispとutilities.lispとscheme-without-tail-without-macro.lispに分けてシステム定義した場合のASDF定義例を以下に示そう．

まず最初に，パッケージ部分だけを取りだして，以下のようなpackage.lispファイルとする．

```
seiji@server:~$ cat package.lisp
(defpackage scheme
  (:use :common-lisp)
  (:export #:scheme #:macro #:macro-expand #:begin #:set! #:lambda
           #:null? #:eq? #:equal? #:eqv? #:write #:display #:newline))
```

同様にユーティリティ部分だけを取りだして，以下のようなファイルとしておく．

```
seiji@server:~$ cat utilities.lisp
(in-package :scheme)

(defparameter *scheme-procs*
  '(+ - * / = < > <= >= cons car cdr not append list read member
    (null? null) (eq? eq) (equal? equal)
    (eqv? eql) (write prin1) (display princ)
    (newline terpri)))
```

```
(defun length=1 (x)
  "Is x a list of length 1?"
  (and (consp x) (null (cdr x))))

(defun last1 (list)
  "Return the last element (not last cons cell) of list"
  (first (last list)))

(defun rest2 (x)
  "The rest of a list after the first TWO elements."
  (rest (rest x)))

(defun starts-with (list x)
  "Is x a list whose first element is x?"
  (and (consp list) (eql (first list) x)))

(defun maybe-add (op exps &optional if-nil)
  "For example, (maybe-add 'and exps t) returns
  t if exps is nil, exps if there is only one,
  and (and exp1 exp2...) if there are several exps."
  (cond ((null exps) if-nil)
        ((length=1 exps) (first exps))
        (t (cons op exps))))
```

最後に残りの部分がschemeプログラム本体となる．ただし，お決まりのように，最初にin-packageが追加されていることを注意しておく．

```
seiji@server:~$ cat scheme-without-tail-without-macro.lisp
(in-package :scheme)

(defun scheme (&optional x)
  "A Scheme read-eval-print loop (using interp)"
  ;; Modified by norvig Jun 11 96 to handle optional argument
  ;; instead of always going into a loop.
  (init-scheme-interp)
  (if x
      (interp x nil)
    (loop (format t "~&==> ")
          (force-output)
          (print (interp (read) nil)))))

(defun interp (x &optional env)
  "Interpret (evaluate) the expression <<x>> in the environment <<env>>.
   This version handles macros."
  (cond
   ((symbolp x) (get-var x env))
   ((atom x) x)
   ((case (first x)
      (quote  (second x))
      (begin  (last1 (mapcar #'(lambda (y) (interp y env))
                             (rest x))))
      (set!   (set-var! (second x) (interp (third x) env) env))
      (if             (if (interp (second x) env)
                          (interp (third x) env)
                        (interp (fourth x) env)))
      (lambda (let ((parms (second x))
                          (code (maybe-add 'begin (rest2 x))))
                      #'(lambda (&rest args)
```

```
                          (interp code (extend-env parms args env)))))
      (otherwise ;; a procedure application
       (apply (interp (first x) env)
              (mapcar #'(lambda (v) (interp v env)) (rest x))))))))

(defun init-scheme-interp ()
  "Initialize the scheme interpreter with some global variables."
  ;; Define Scheme procedures as CL functions:
  (mapc #'init-scheme-proc *scheme-procs*)
  ;; Define the boolean `constants'. Unfortunately, this won't
  ;; stop someone from saying: (set! t nil)
  (set-global-var! t t)
  (set-global-var! nil nil))

(defun init-scheme-proc (f)
  "Define a Scheme procedure as a corresponding CL function."
  (if (listp f)
      (set-global-var! (first f) (symbol-function (second f)))
      (set-global-var! f (symbol-function f))))

(defun set-var! (var val env)
  "Set a variable to a value, in the given or global environment."
  (if (assoc var env)
      (setf (second (assoc var env)) val)
    (set-global-var! var val))
  val)

(defun get-var (var env)
  "Get the value of a variable, from the given or global environment."
  (if (assoc var env)
      (second (assoc var env))
    (get-global-var var)))

(defun extend-env (vars vals env)
  "Add some variables and values to an environment."
  (nconc (mapcar #'list vars vals) env))

(defun set-global-var! (var val)
  (setf (get var 'global-val) val))

(defun get-global-var (var)
  (let* ((default "unbound")
         (val (get var 'global-val default)))
    (if (eq val default)
        (error "Unbound scheme variable: ~a" var)
        val)))

seiji@server:~$
```

このファイル構造に対応するscheme.asdを次のように作る．このときファイルタイプをasdとすること．

```
seiji@server:~$ cat scheme.asd
(defpackage :scheme-system
  (:use :common-lisp :asdf))

(in-package :scheme-system)

(defsystem :scheme
  :name "Scheme"
```

```
  :author "Seiji Koide <koide@ontolonomy.co.jp>"
  :maintainer "Seiji Koide <koide@ontolonomy.co.jp>"
  :version "0.0.1"
  :license "PAIP"
  :description "Scheme written in Common Lisp from PAIP"
  :long-description "Modified version of Scheme in Common Lisp from 'Paradigms of Artificiall
Intelligence Programming' by Peter Norvig"
  :components
  ((:file "package")
   (:file "utilities")
   (:file "scheme-without-tail-without-macro")))

seiji@server:~$
```

asdfファイルを含めてこれら四つのファイルを同じディレクトリ，たとえばschemeディレクトリに一緒にいれておいて，そのschemeディレクトリをカレントディレクトとして，次のようにすれば，複数のファイルをまとめてコンパイル・ロードすることができる.

```
CL-USER(2): :ld scheme.asd
loading "scheme.asd"
CL-USER(3): (asdf:load-system :scheme)

; compiling file "/home/seiji/scheme/package.lisp" (written 07 AUG 2016 05:21:41 PM):
; compiling (DEFPACKAGE SCHEME ...)

; /home/seiji/.cache/common-lisp/sbcl-1.2.4.debian-linux-x64/home/seiji/scheme/package-
TMP.fasl written
; compilation finished in 0:00:00.003
; compiling file "/home/seiji/scheme/utilities.lisp" (written 07 AUG 2016 09:21:16 PM):
; compiling (IN-PACKAGE :SCHEME)
; compiling (DEFPARAMETER *SCHEME-PROCS* ...)
; compiling (DEFUN LENGTH=1 ...)
; compiling (DEFUN LAST1 ...)
; compiling (DEFUN REST2 ...)
; compiling (DEFUN STARTS-WITH ...)
; compiling (DEFUN MAYBE-ADD ...)

; /home/seiji/.cache/common-lisp/sbcl-1.2.4.debian-linux-x64/home/seiji/scheme/utilities-
TMP.fasl written
; compilation finished in 0:00:00.018
; compiling file "/home/seiji/scheme/scheme-without-tail-without-macro.lisp" (written 07 AUG
2016 09:26:44 PM):
; compiling (IN-PACKAGE :SCHEME)
; compiling (DEFUN SCHEME ...)
; compiling (DEFUN INTERP ...)
; compiling (DEFUN INIT-SCHEME-INTERP ...)
; compiling (DEFUN INIT-SCHEME-PROC ...)
; compiling (DEFUN SET-VAR! ...)
; compiling (DEFUN GET-VAR ...)
; compiling (DEFUN EXTEND-ENV ...)
; compiling (DEFUN SET-GLOBAL-VAR! ...)
; compiling (DEFUN GET-GLOBAL-VAR ...)

; /home/seiji/.cache/common-lisp/sbcl-1.2.4.debian-linux-x64/home/seiji/scheme/scheme-
without-tail-without-macro-TMP.fasl written
; compilation finished in 0:00:00.047
#<ASDF/LISP-ACTION:LOAD-OP >
```

```
#<ASDF/PLAN:SEQUENTIAL-PLAN {1005F0DBE3}>
CL-USER(4): (in-package :scheme)

#<PACKAGE "SCHEME">
SCHEME(5): (scheme)

==> (+ 1 2)

3
==>
```

プログラムの一部を修正したりすると，上記と同じようにasdf:load-systemしても更新されたファイルだけがコンパイルされてからロードされる．ここに提示したscheme.asdファイルでは:componentsにかかれた順にファイルがロードされるが，ファイルやモジュール依存関係を明示的に記述する方法もある．ASDFにおけるdefsystemの書き方の詳細を学ぶのは大変であるが，上記例のように簡単な場合から出発して必要に応じて少しずつ学んでいくしかない．その書き方やasdfファイルをロード後のオペレーション方法などは，ASDFのマニュアルを参照されたい．

quicklispとASDF

quicklispのインストール方法については付2.3で説明した．ここでは上記にてASDFを使って作成したモジュールやシステムをquicklispでどう使うか説明する．ディレクトリschemeの直下に上記のpackage.lisp，utilities.lisp，scheme-without-tail-without-macro.lisp，そしてscheme.asdがある状態で，schemeディレクトリを~/quicklisp/local-projects/の下に置く．その状態で(asdf:initialize-source-registry)を実行する．すると次のように，ライブラリと同じようにquicklispでロードできるようになる．

```
seiji@server:~$ ls ~/quicklisp/local-projects/scheme
package.lisp                           scheme.asd
scheme-without-tail-without-macro.lisp  utilities.lisp
seiji@server:~$ sbcl
This is SBCL 1.2.4.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
CL-USER(1): (asdf:initialize-source-registry)

CL-USER(2): (ql:quickload :scheme)
To load "scheme":
  Load 1 ASDF system:
    scheme
; Loading "scheme"

(:SCHEME)
CL-USER(3): (in-package :scheme)

#<PACKAGE "SCHEME">
SCHEME(4): (scheme)

==> (+ 1 2)
```

```
3
==>
```

20.4 処理系依存の条件付き読み込み

今日でも，入手して使用可能なCommon Lisp処理系にはAllegro Common Lisp，LispWorksのほかに，Steel Bank Common Lisp (SBCL)，CLISP，Clozure CL (CCL), CMU Common Lisp (CMUCL)など，多くの処理系がある．Common Lisp制定時にはもっと多くのLisp処理系があって，異なる処理系について同じプログラムを可搬にしようとすると，処理系ごとの細かな違いをコード上で吸収する仕組みが必要であった．それが`*features*`を利用した条件付きの読み込み機能である．通常ANSIの仕様の範囲ではほとんど移植性が問題になることはないが，それでも実際には微妙な違いがあったりして，やはり`*features*`を利用した条件付き読み込み機能を利用せざるを得ない．参考例としてCLOS標準化のときに参照モデルとなったPortable Common Loops (PCL)のdefsys.lispの一部のコードを以下に示す．

```
;;;
;;; When you get a copy of PCL (by tape or by FTP), the sources files will
;;; have extensions of ".lisp" in particular, this file will be defsys.lisp.
;;; The preferred way to install pcl is to rename these files to have the
;;; extension which your lisp likes to use for its files.  Alternately, it
;;; is possible not to rename the files.  If the files are not renamed to
;;; the proper convention, the second line of the following defvar should
;;; be changed to:
;;;     (let ((files-renamed-p nil)
;;;
;;; Note: Something people installing PCL on a machine running Unix
;;;       might find useful.  If you want to change the extensions
;;;       of the source files from ".lisp" to ".lsp", *all* you have
;;;       to do is the following:
;;;
;;;       % foreach i (*.lisp)
;;;       ? mv $i $i:r.lsp
;;;       ? end
;;;       %
;;;
;;;       I am sure that a lot of people already know that, and some
;;;       Unix hackers may say, "jeez who doesn't know that".  Those
;;;       same Unix hackers are invited to fix mv so that I can type
;;;       "mv *.lisp *.lsp".
;;;
(defvar *default-pathname-extensions*
  (car '(#+(and (not imach) genera)          ("lisp"  . "bin")
         #+(and imach genera)                ("lisp"  . "ibin")
         #+Cloe-Runtime                      ("l"     . "fasl")
         #+(and dec common vax (not ultrix)) ("LSP"   . "FAS")
         #+(and dec common vax ultrix)       ("lsp"   . "fas")
         #+KCL                               ("lsp"   . "o")
         #+IBCL                              ("lsp"   . "o")
         #+Xerox                             ("lisp"  . "dfasl")
         #+(and Lucid MC68000)               ("lisp"  . "lbin")
         #+(and Lucid VAX)                   ("lisp"  . "vbin")
         #+(and Lucid Prime)                 ("lisp"  . "pbin")
         #+(and Lucid SUNRise)               ("lisp"  . "sbin")
```

```
#+(and Lucid SPARC)                ("lisp"  . "sbin")
#+(and Lucid IBM-RT-PC)            ("lisp"  . "bbin")
#+(and Lucid MIPS)                 ("lisp"  . "mbin")
#+(and Lucid PRISM)                ("lisp"  . "abin")
#+(and Lucid PA)                   ("lisp"  . "hbin")
#+(and excl SPARC)                 ("cl"    . "sparc")
#+(and excl m68k)                  ("cl"    . "m68k")
#+excl                             ("cl"    . "fasl")
#+cmu ("lisp" . #.(c:backend-fasl-file-type c:*backend*))
#+HP-HPLabs                        ("l"     . "b")
#+TI ("lisp" . #.(string (si::local-binary-file-type)))
#+:gclisp                          ("LSP"   . "F2S")
#+pyramid                          ("clisp" . "o")
#+:coral                           ("lisp"  . "fasl")
#-(or symbolics (and dec common vax) KCL IBCL Xerox
      lucid excl :CMU HP TI :gclisp pyramid coral)
                                   ("lisp"  . "lbin"))))
```

Lispソースコードを読み組むとき，リーダー(read関数)は`#+`があったときにはその次の表現が`*features*`にある条件のどれかにマッチしたときのみ，その直後の書式を読み込む．逆に`#-`があったときには，その次の表現がマッチしないときのみ，その直後の書式を読み込む．たとえば，ACL express では`*features*`は次のようになっている．

```
cl-user(1): (pprint *features*)
(:allegro-cl-express :allegro-cl-trial :acldns net.aserve::queue-does-not-timeout
 :io-timeout :inflate-bug20472 :|YACC Release 0.98| :ide :common-graphics :ipv6
 :acl-socket :hiper-socket :profiler :compiler :use-structs-in-compiler :clos
 :atomic-subword-setf :atomic-setf :verify-stack :verify-car-cdr :dynload :dlwin
 :x86 :mswindows :microsoft :microsoft-32 :little-endian :encapsulating-efs
 :double-float-random-fix :relative-package-names  :module-versions :ieee
 :ieee-floating-point :conforming-ieee :ics :common-lisp :ansi-cl :draft-ansi-cl-2
 :x3j13 :allegro :excl :franz-inc :allegro-version>= :allegro-version=
 :new-environments :gsgc :flavors :multiprocessing :os-threads :dynload-acl-library
 :allegro-v8.2 :smp-macros :ssl-support)
```

その場合，読み込み条件によって任意の書式を読む読まないの制御をすることができる．以下は**Allegro での**実施例である．

```
cl-user(2): #+allegro (print 'OK)

OK
OK
cl-user(3): #-sbcl (print 'NG)

NG
NG
cl-user(4): #+sbcl (print 'Woops!)
#+allegro (print 'Woops!)

Woops!
Woops!
cl-user(5):
```

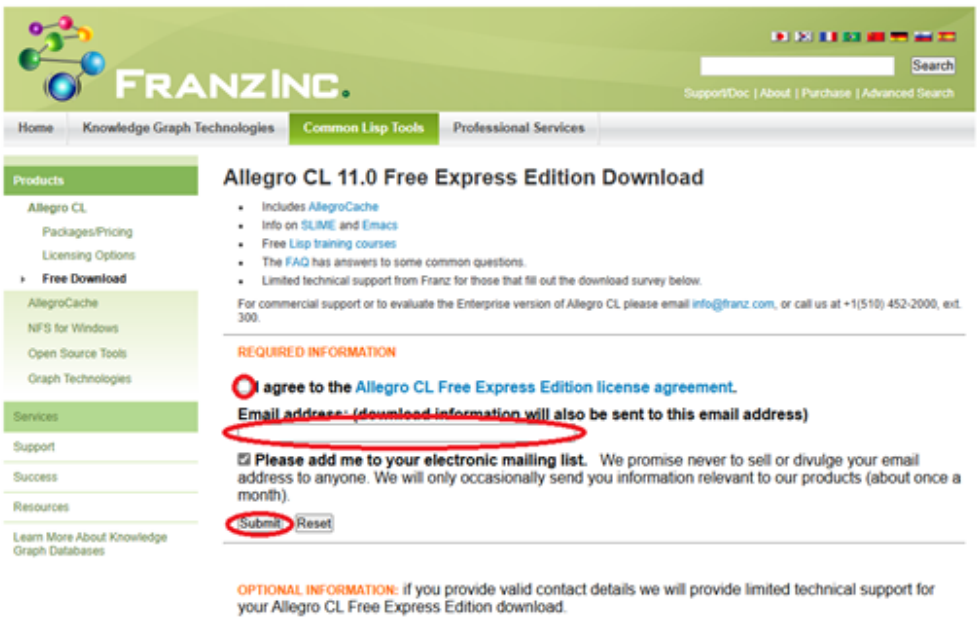
最後のところでは#+sbclでは書式が読み込まれずに#+allegroのところで(print 'Woops!)が読み込まれた．蛇足になるが，ACL mispでは処理系が大文字小文字を区別するので，:allegroとAllegro がマッチしないということを注意しておく

付録1 Allegro Express のインストール

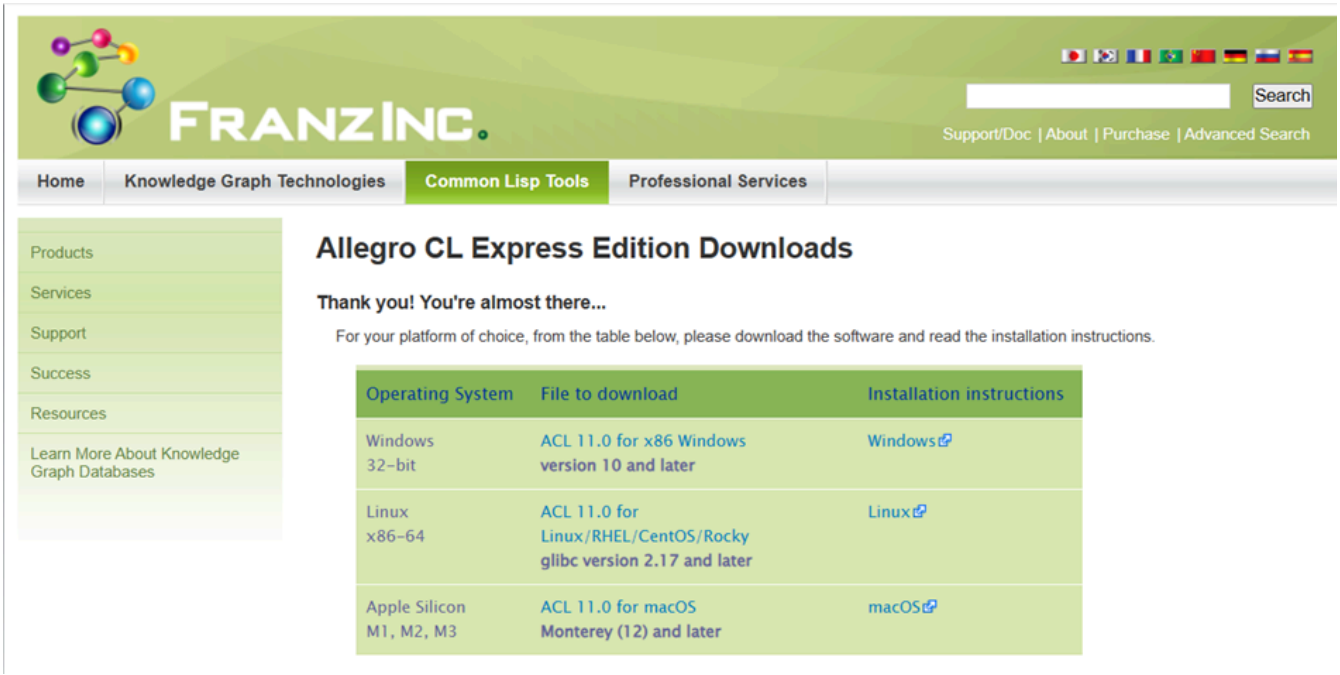
付1.1 ANSI Common Lisp with IDE のダウンロードとインストール

米国Franz社は商用Lisp処理系のベンダーとして唯一日本で販売代理店を持つ会社である．ほかに個人で利用するには LispWorks 社のものがあるが，メールのやり取りが英語で必要になるなど，企業や研究所などの法人が利用するには難しいところがあり，特に理由がない限り，日本では法人用にはFranz社のものが唯一の選択となる．Lisp の勉強用に Franz は Free Express という名で商用とほとんど変わらない性能の処理系を利用可能にしている．Windows 版 Express で提供されるのは 32 bit 版のみ（LinuxとMac版では64bit版のみ）だが，メモリ空間に限界があるにせよ，本書籍の勉強用には十分な性能だ．付属のライブラリなどを除いて Common Lisp として実商用版との性能上の違いは他には何もない．この付録 1 ではWindows 版，Linux 版，Mac 版の Allegro Common Lisp free Express のダウンロードとインストール方法を説明する．

Franz 社のダウンロードサイト http://www.franz.com/downloads/clp/survey に行って，次のような画面で Emailアドレスにインストラクションを受け取れるアドレスを書きこみ，"I agree to ... " のチェックボックスにチェックを入れて，「Submit」ボタンを押す．"Please add me ... " のチェックボックスにチェックを入れても入れなくてもかまわない．また，下記の画面には表示されていないが，その下にも詳細情報の記入欄があって，これも記入してもしなくてもかまわない．



すると，下のようなダウンロード画面が現れる．ここまでは Windows 版，Linux 版，Mac 版に共通である．

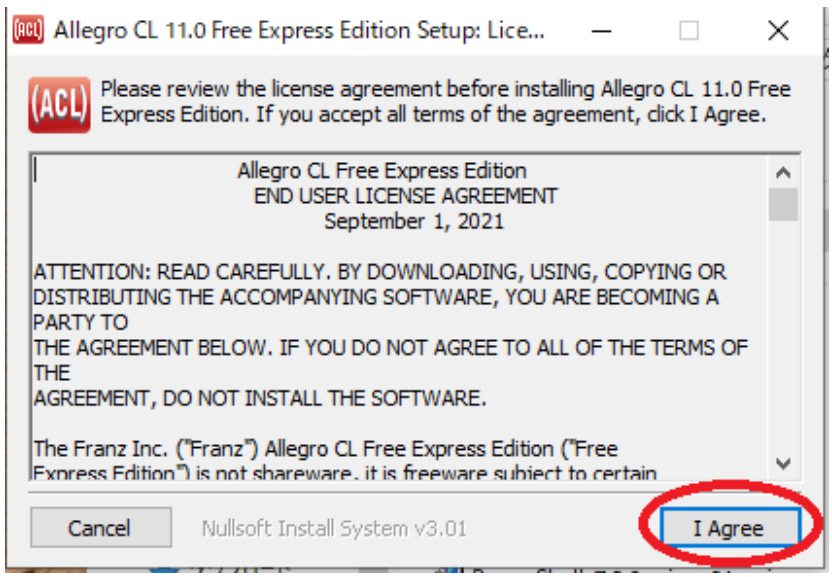


付1.1.1 Windows 32bit 版のインストール

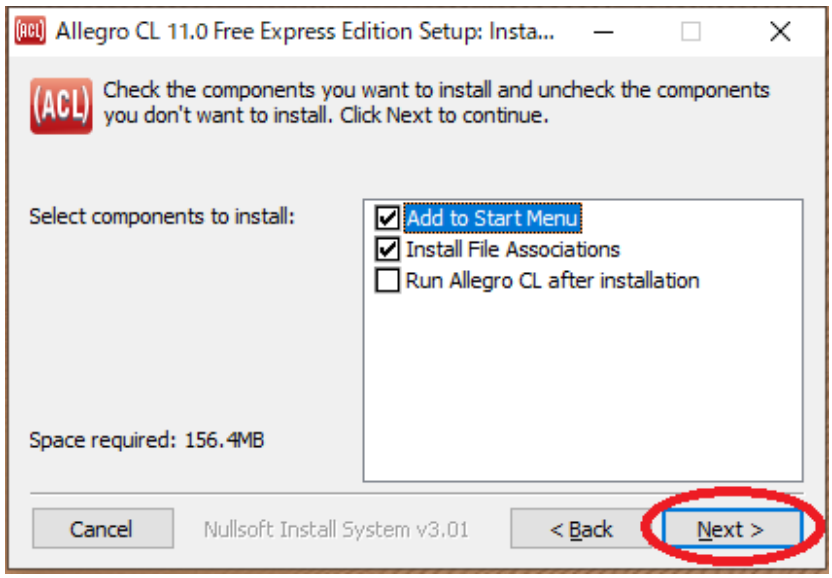
Windows 用ではこれまでと同様に32ビット版しか提供されていない．もちろん，32ビット版であっても x64 ベース プロセッサ，64 ビット オペレーティング システムで動かすことはできる．

ACL11.0 for x86 Windows を選んでクリックする．ウェブ経由でのファイルダウンロードになるが，いつものようにファイルを開くか「保存」するかを聞かれるので，「保存」を選んでインストーラーをダウンロードする．ダウンロード後はそれを実行すると，インストールが開始される．

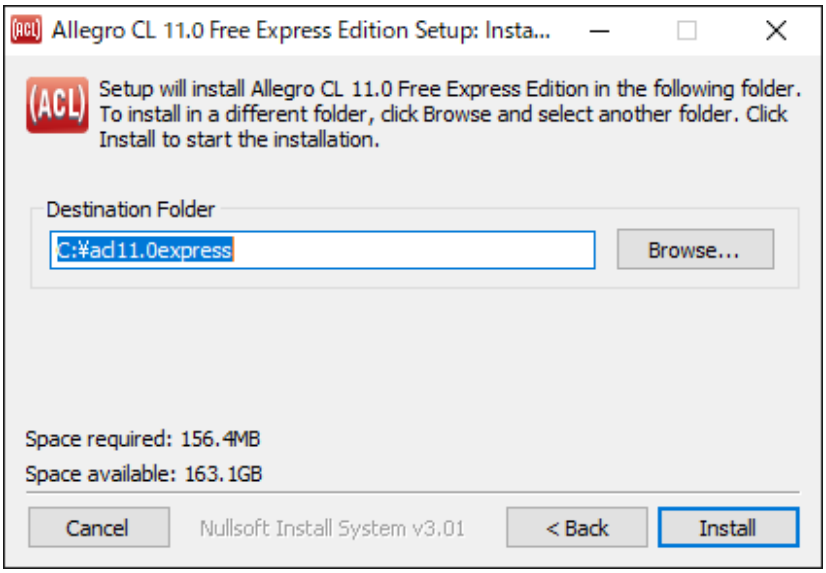
インストールが始まると，最初に以下のような画面が表示される．「I Agree」ボタンを押す．



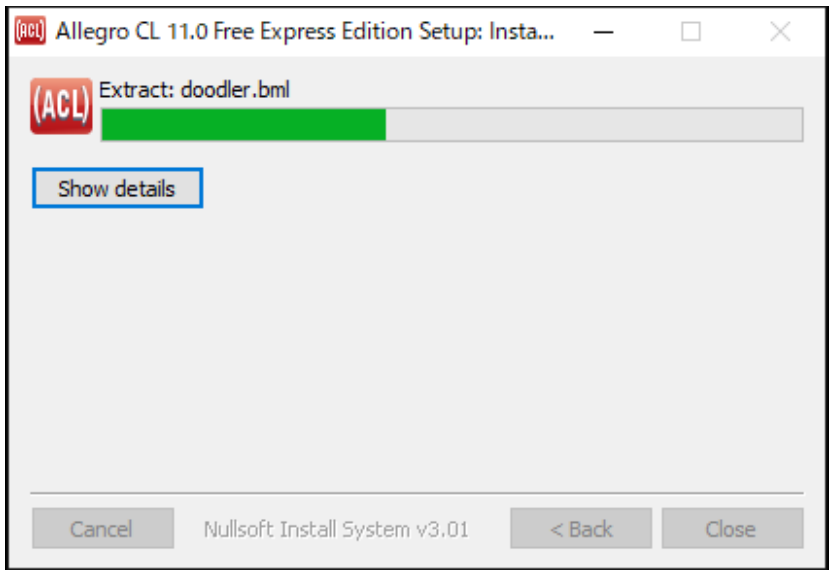
次に156.4MB必要という画面では，特に理由がなければ既定のままで「Next >」を押す．



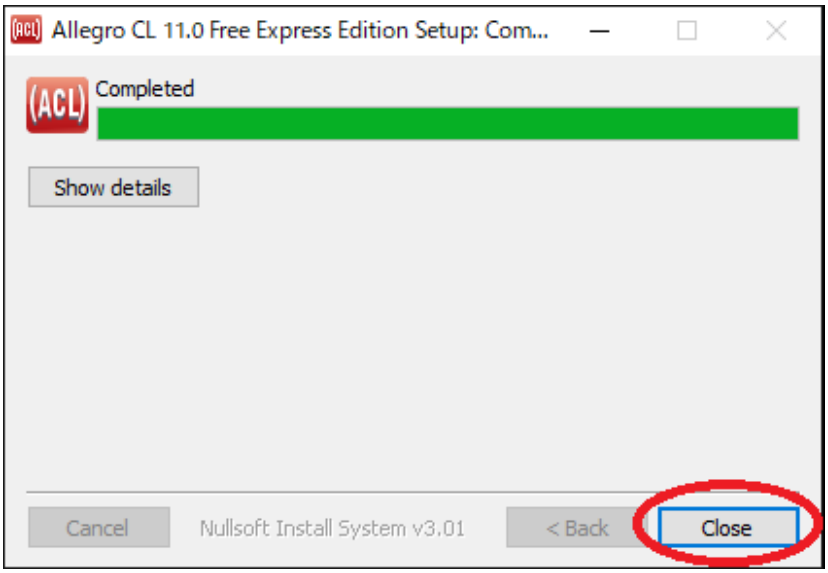
既定のインストール場所は C: 直下であるが，最初のインストールで特に理由がなければ，このままで「Install」を押す．二回目以降のインストールではたとえプログラムのアンインストールしたあとでも，残っているフォルダを削除しないと同じ場所には再インストールができないので注意しておく．



以下のようなインストール進行中の画面が出たあと

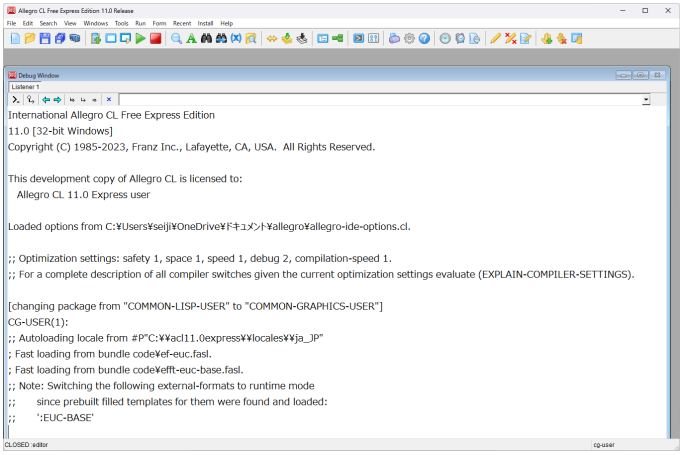


以下のように完了画面が出ればインストール終了だ.

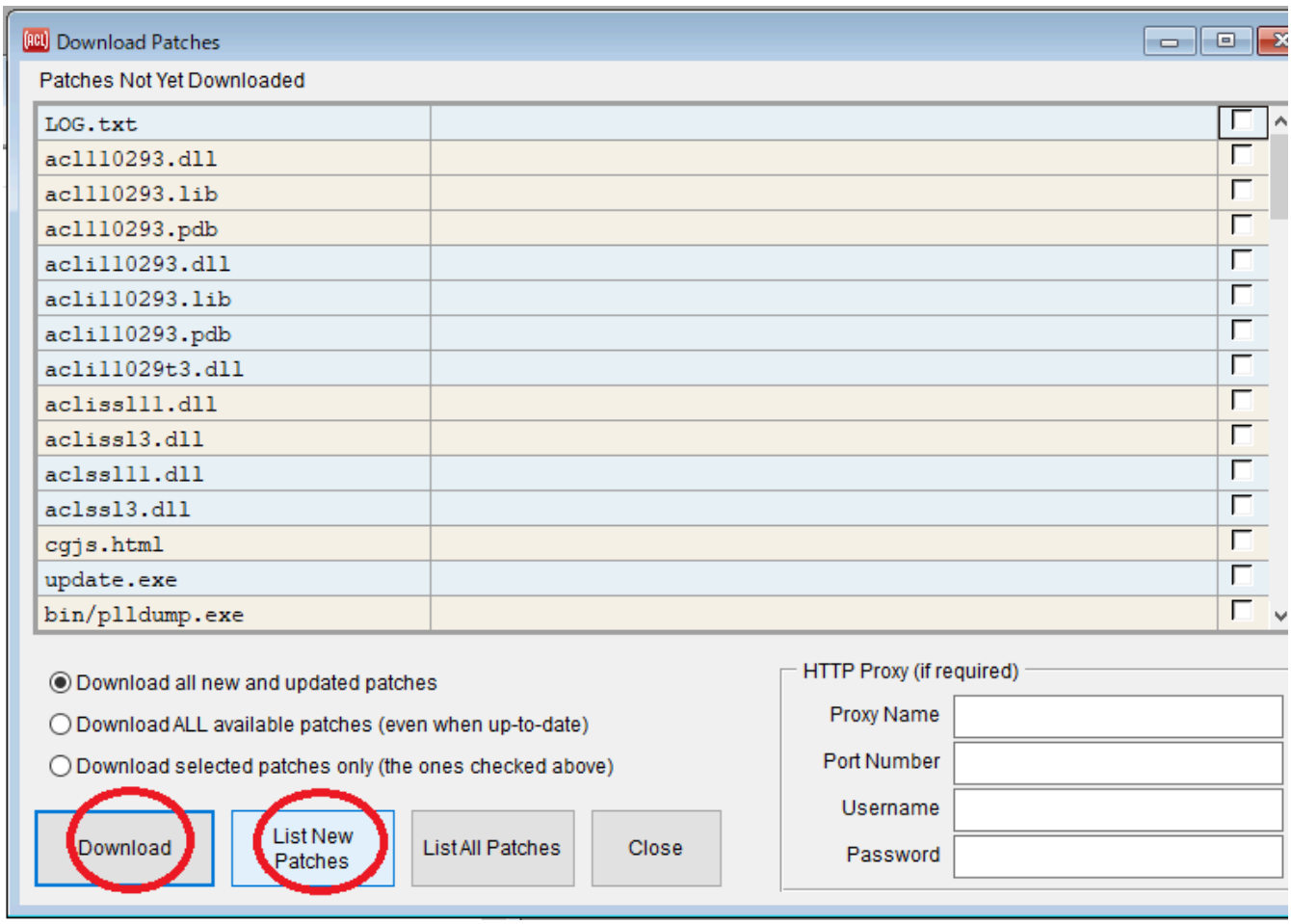


「Close」ボタンを押してインストールが終了する.

スタートメニューにはAllegro CL 11.0 Free Express Edition に Browser と Desktop の二つがある．どちらを選択してもかまわないが最初はパッチあての都合で．Desktop 版から初めていただきたい。Desktop 版は従来から Windows にあった種類のものであるが，Browser 版は 11.0 から新しく登場したもので，Mac版やLinux版と共通の技術を使ったものである．"Welcome to Allegro CL v11.0" のブラウザ画面はすぐ消してもらってかまわない．また，最初トップに出ている "IDE's built-in-editor" 画面も消してもらってかまわない．以下の画面はデスクトップ版の表示画面である。

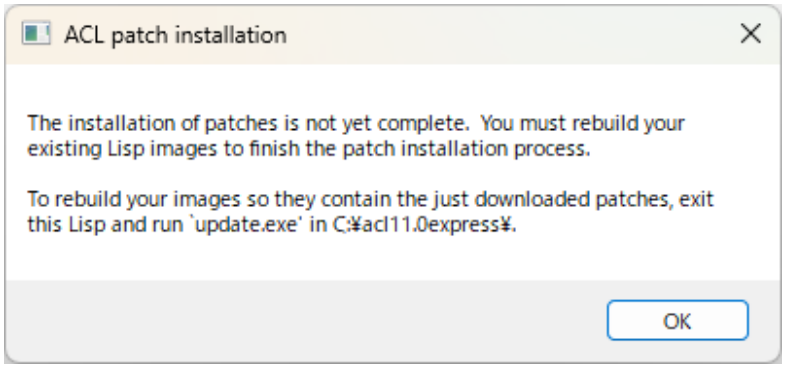


まず最初，デスクトップ版をつかって積極的にパッチをインストールして当てよう．最上部のメインメニューから "Install" を選び、サブメニューの "updates" をクリックすると "Download Patches"画面が現れるので，ここで「List New Pathes」ボタンを押してパッチ・リストを表示させる．



"Download all new ... "ラジオボタンを選択しておいて，「Download」ボタンを押そう．

以下のような画面が出ればパッチのダウンロード完了だ．「OK」ボタンを押してポップアップ・ウィンドウを閉じる．



しかし，この状態ではパッチはまだ当たっていないし，このまま再度ACLを立ち上げようとすると，パッチが当たっていないと叱られる．**ACL11.0 を終了した状態**で，通常の**Windowsプログラムを起動する要領でスタートメニューから ACL11.0 の update.exe を起動**すると，実際にパッチが当てられて，再びACLが使えるようになる．再度ACLを起動して最初と同様な画面が表示されれば，インストールとパッチ当てが終了となる．

付1.1.2 Ubuntu に Allegro Express をインストール

Linux には Debian系やら CentOS系やら各種のものがあるが，今では広く普及して使いやすいものとして Ubuntu がある．最近の Windows Subsystem for Linux (WSL) でもデフォールトとしてインストールされるのは Ubuntu になっている．ここでは Ubuntu に Allegro Express をインストールする方法を紹介する．

Franz社の Allegro Common Lisp はもともとWindowsを中心にバージョンアップを重ねてきた．そのため IDE や描画機能は Linux や MacOS では不十分であったり，提供されないということがあったが，最近デリバリの ACL11.0 ではこれまで使ってきた UNIX/Linux の GTK を捨てて，ブラウザーをIDEとして仕立てて使うようにした．そのためのミドルウェアとして Common Graphics on HTML5/JavaScript (CG/GS) を開発している．今では Linux でも MacOS でも Windows 版とほぼ同等な IDE が使えるようになった．そのため，GUI機能のない WSL 上の ubuntu に Allegro をインストールしても localhost のポート番号を指定することで，Windows 環境からこれが使えてしまうのだ．

ここでは最初に Ubuntu デスクトップ版へのインストール方法を紹介し、その後 Wsl/Ubuntu へのインストール方法を紹介する．本節の最初に述べたダウンロード画面までは Windows の場合と同様である．ダウンロード画面で **ACL 11.0 for Linux** を選んでダウンロードする．ダウンロードした **acl11.0express-linux-x64.tbz2** ファイルを自分のホームディレクトリに移動させる．下にあるように，**tar jxf** で解凍し，後々楽なように解凍してできた acl11.0express.64 ディレクトリ名を acl に改名する．

```
seiji@mini:~$ ls
acl11.0express-linux-x64.tbz2  テンプレート  ビデオ        公開
snap                           デスクトップ  ピクチャ
ダウンロード                   ドキュメント  ミュージック
seiji@mini:~$ tar jxf acl11.0express-linux-x64.tbz2
seiji@mini:~$ ls
acl11.0express-linux-x64.tbz2  ダウンロード  ドキュメント  ミュージック
acl11.0express.64              テンプレート  ビデオ        公開
```

```
snap                            デスクトップ  ピクチャ
seiji@mini:~$ mv acl11.0express.64  acl
```

まずは最初に alisp が動くのを確認しよう．**acl/alisp** とコマンド入力して，次のようになればOKだ．

```
seiji@mini:~$ acl/alisp
International Allegro CL Free Express Edition
11.0 [64-bit Linux (x86-64)]
Copyright (C) 1985-2023, Franz Inc., Lafayette, CA, USA.  All Rights Reserved.

This development copy of Allegro CL is licensed to:
   Allegro CL 11.0 Express user

;; Optimization settings: safety 1, space 1, speed 1, debug 2,
;; compilation-speed 1.
;; For a complete description of all compiler switches given the
;; current optimization settings evaluate (EXPLAIN-COMPILER-SETTINGS).
CL-USER(1): :exit
; Initial Lisp Listener will be killing processes:
;; processes killed
; Exiting
```

alisp （に限らず，すべての Allegro）を終了させるには，**:exit** とセミコロン付きのREPL用コマンドを打てばよい．その次は下記のように Allegro Express を立ち上げてみよう．立ち上げの途中でポート番号が印刷されるので注意されたい．

```
seiji@mini:~$ acl/allegro-express
;; ACLSSL: openssl command reports version (3 0)
International Allegro CL Free Express Edition
11.0 [64-bit Linux (x86-64)]
Copyright (C) 1985-2023, Franz Inc., Lafayette, CA, USA.  All Rights Reserved.

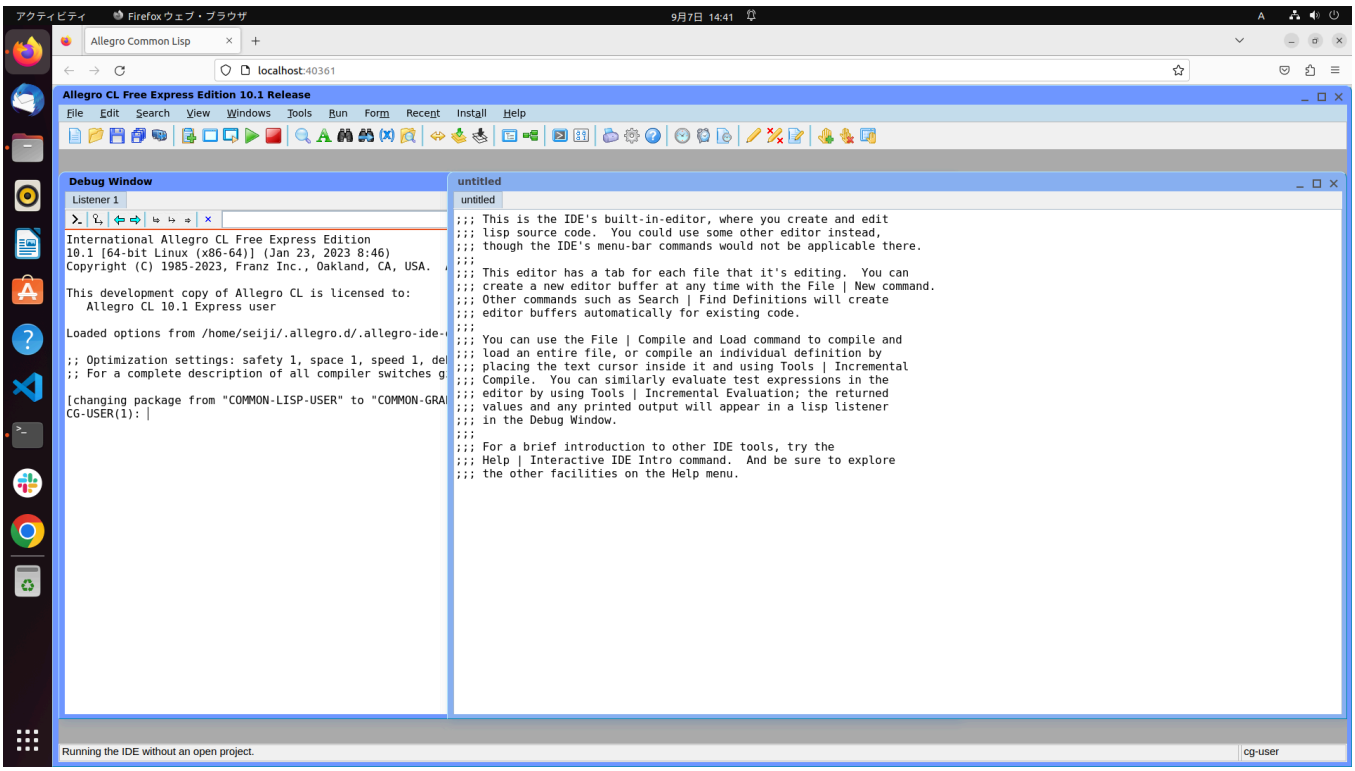
This development copy of Allegro CL is licensed to:
   Allegro CL 11.0 Express user

;; Optimization settings: safety 1, space 1, speed 1, debug 2,
;; compilation-speed 1.
;; For a complete description of all compiler switches given the
;; current optimization settings evaluate (EXPLAIN-COMPILER-SETTINGS).
port 35591

   [...]

127.0.0.1 - - [Thu, 08 May 2025 12:26:21 GMT] "GET / HTTP/1.1" 200 355342
127.0.0.1 - - [Thu, 08 May 2025 12:26:21 GMT] "GET /favicon.ico HTTP/1.1" 404 -1
; Autoloading for SHA1-STRING:
; Fast loading /home/seiji/acl/code/digest.fasl
```

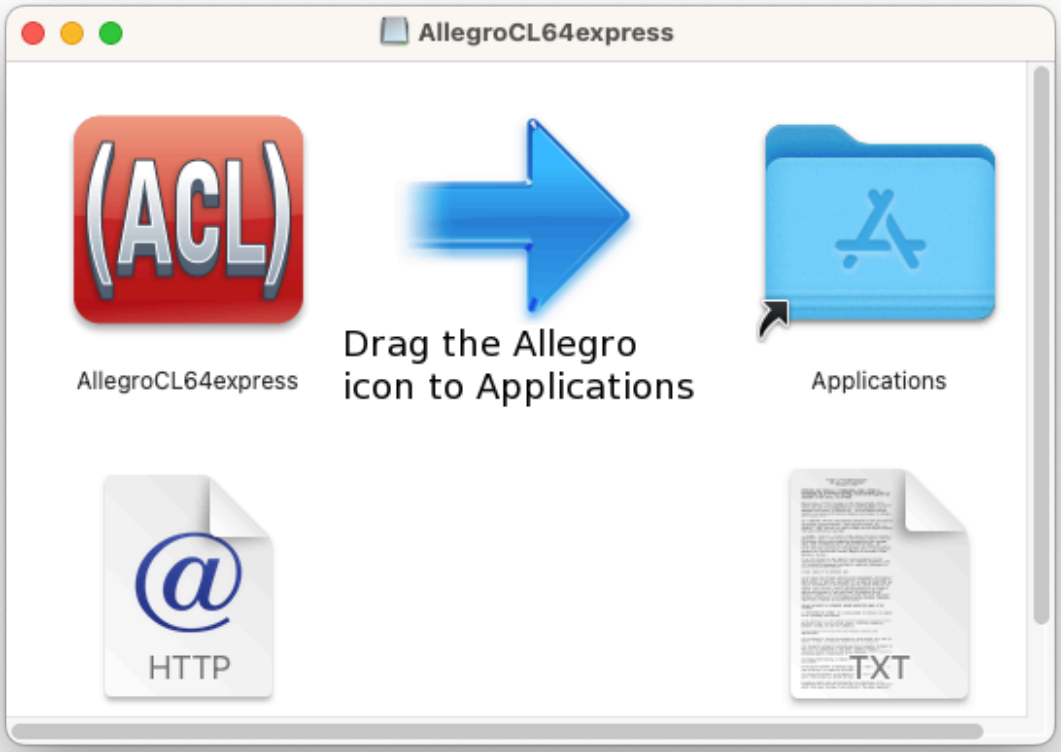
ここで Ubuntu デスクトップ版ではブラウザが自動的に立ち上がり、IDEの画面がブラウザの画面として使えるようになる．下図は Ubuntu デスクトップ版での IDE 表示画面である。

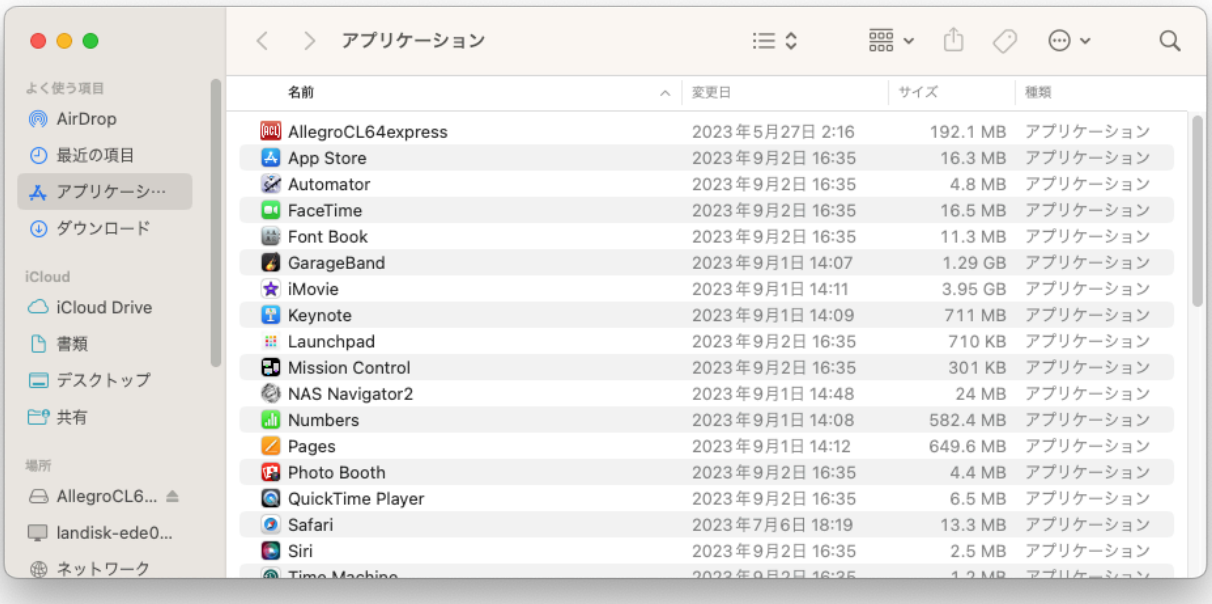


Allegro Express を終了させるには，IDEのREPL上で最初 **:exit** 打ち込んだ後、残るブラウザとターミナルを閉じればよい.

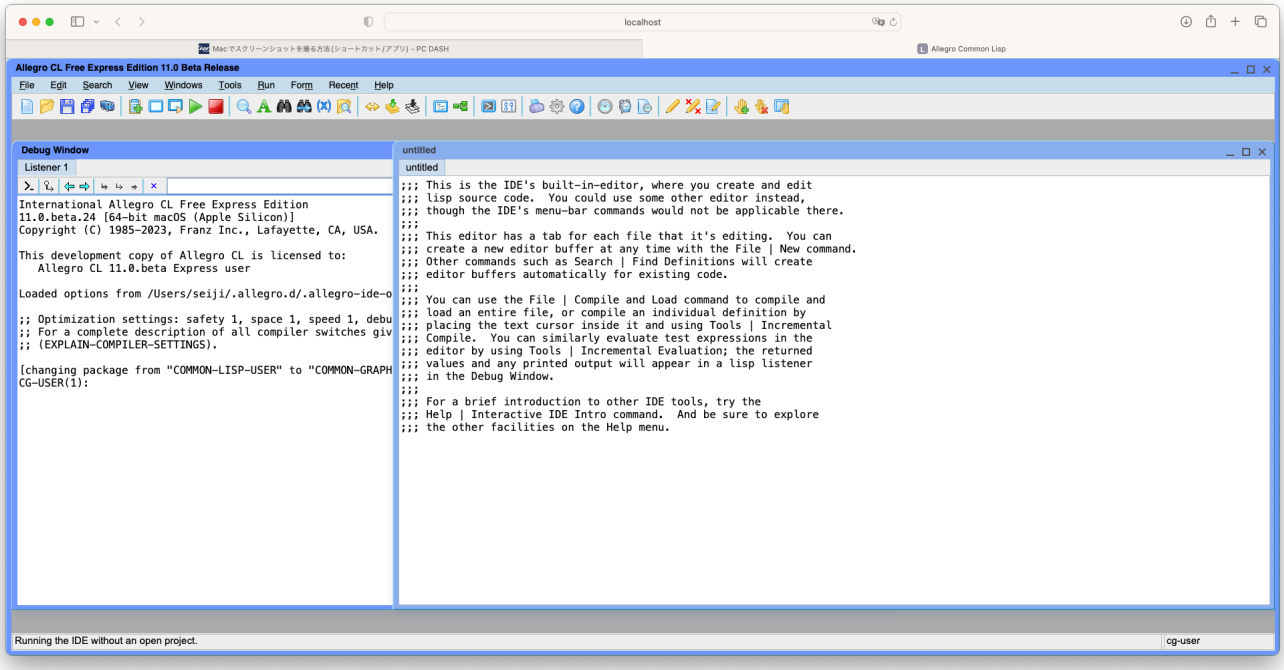
付1.1.3 Mac に Allegro express をインストール

本節の最初に述べたダウンロード画面まではWindowsの場合と同様である．ダウンロード画面で今度は **ACL 11.0 for macOS** を選んでダウンロードする．ただし，M1/M2/M3 プロセッサーが条件になるので注意されたい．ダウンロードした acl11.0express-macos-arm64.dmg をダブルクリックで起動すると，次のようなポップアップ画面が得られるので，この画面上で画面指示にしたがって左の AllegroCL64expressを右のフォルダ上にドラッグすると，アプリケーション・フォルダに AllegroCL64expressが現れるので直接それをダブルクリックで起動すればよい.





するとサファリ上に現れるのは Windows の場合とほぼ同様な IDE の画面である.



付1.1.4 Windows/Wsl に Allegro express をインストール

Windows/Wsl のインストール

WSL(Windows Subsystem for Linux) というのは，Windows 環境で従来のオラクル VirtualBox よりも，もっと Windows 環境と一体化された Linux を動かすことのできるウィンドウズサブシステムで，Windows 環境と Linux 環境を完全に分離して使いたい場合には適さないが，Linux 用アプリも Windows 環境で使えるとか，Windows で Linux アプリをダウンロードして Linux 環境で稼働させるなど，融通無碍に使える感のある優れモノだ．

ここでは，その WSL に Linux 用 Allegro Express をインストールする方法を紹介する．

最初にまだWSLをインストールしていなければ，ターミナルを管理者モードで開き，次のようにして WSL をインストールする．

```
C:\Users\seiji>wsl --uninstall

C:\Users\seiji>wsl --install
要求された操作には管理者特権が必要です。
ダウンロード中: Linux 用 Windows サブシステム 2.4.13
インストール中: Linux 用 Windows サブシステム 2.4.13
Linux 用 Windows サブシステム 2.4.13 はインストールされました。
この操作を正しく終了しました。
ダウンロード中: Ubuntu
インストール中: Ubuntu
ディストリビューションが正常にインストールされました。'wsl.exe -d Ubuntu' を使用して起動できます
```

最初に WSL をアンインストールしているのは，さもないと自動的に Ubuntu がインストールされない場合があるためで，念のため完全に wsl がインストールされてない状態にするためである．

そのあと一度は **wsl -d Ubuntu** とタイプ入力して動かしておいたほうがよいだろう．

```
C:\Users\seiji>wsl.exe -d Ubuntu
Provisioning the new WSL instance Ubuntu
This might take a while...
Create a default Unix user account: seiji
New password:
Retype new password:
passwd: password updated successfully
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 24.04.2 LTS (GNU/Linux 5.15.167.4-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/pro

 System information as of Sun May 11 00:54:07 JST 2025

  System load:  0.25                Processes:             31
  Usage of /:   0.1% of 1006.85GB   Users logged in:       0
  Memory usage: 9%                  IPv4 address for eth0: 172.21.171.94
  Swap usage:   0%


This message is shown once a day. To disable it please create the
/home/seiji/.hushlogin file.
seiji@GOFAI:/mnt/c/Users/seiji$
```

すぐ exit で抜けて以降は、スタートメニューや Windowsコマンドキー+R Ubuntu などで，Ubuntu を起動すればよい.

Wsl/Ubuntu に Allegro Express for Linux をインストール

付1.1で示したように，Allegro Express をダウンロードするが，当然のことながら for Linux をダウンロードしなければならないことに留意されたい．ダウンロードした acl11.0express-linux-x64.tbz2 を Windows 環境から Wsl/Ubuntu 環境に移動させるには，エクスプローラでもいいし，あるいは検索窓でもいいが **\\wsl$** と入力して検索する。するとエクスプローラに Ubuntu フォルダーが現れる。ここが Ubuntu 環境でのルートディレクトリになるので，そこから自分のホームディレクトリをエクスプローラで開いておいて，そこに acl11.0express-linux-x64.tbz2 を置けばよい．あとは付1.1.2 の記述のとおりである．ただし，ACL/Allegro-Express を起動しても，ブラウザは自動的には立ち上がらない．途中で印刷されるポート番号をよく見て，Windows 環境中で Edge でもよいから，ブラウザを立ち上げてURLアドレスとして **localhost:<port>** にアクセスすれば，Allegro Express の IDE 画面が現れる。

付1.2 Modern Common Lisp の作成

上記で得られる Lisp 環境は ANSI Common Lisp であり，シンボルで大文字小文字を区別しない．小文字で入力しても自動的に大文字とされて，返される値も大文字のシンボルだ.

```
CG-USER(1): t
T
CG-USER(2): nil
NIL
```

```
CG-USER(3): 'foo
FOO
CG-USER(4):
```

ここでは，プロンプトが大文字の CG-USER であること，入力されたシンボルは大文字で返されることを確認されたい．

Lisp はFORTRANと同じぐらい古い歴史の計算機言語であり，その当時は計算機に小文字を入力する手段がなかった．それを今も引きずっているわけだが，今日ではどの計算機言語もあたりまえのように大文字小文字を区別する．すべての計算機の文字コードにUNICODEがインストールされた今日では，これはあまりにも古すぎると言わざるを得ない．実は入力文字に関しては，Lisp リーダには大文字小文字をそのまま保つモードがある．ところが，大文字小文字を区別する preserve モードにおいても，Common Lisp に既存の関数やマクロのようなシンボルはACLのalisp (with IDE) ではやっぱり大文字のままなのだ．だから，せっかく preserve モードに変えても，コードを書くときには既存の Common Lisp 関数やマクロを大文字で書かざるをえず，とても使いにくい．ありがたいことに，Franz 社は既存の Common Lisp のすべてのシンボルを小文字に変えて preserve モードにした処理系を商用版ではバンドルしている．それが Modern Lisp と呼ばれるものだ．

商用版を購入すれば，ANSI Lisp (alisp)，Modern Lisp (mlisp)，それらの non-international 版，および with IDE 版と合計 6 種類の Lisp 処理系が配布される．上記でインストールした Free Express は ANSI Lisp with IDE だが，これを Modern Lisp with IDE に変換することができる．以下ではその方法を説明するが，先を急ぎたいかたは，あるいはLispで小文字入力を可能にするなんてけしからんという方はこれ以降の記述をスルーしても問題はない．

まずは自分がインストールした Free Express フォルダの中身をみて，allegro-express.dxl の存在を確認されたい．これがインストールした Free Express の Lisp イメージであり，これをもとに modern lisp版(with IDE) を作成する．

ACL free Express を起動した状態で（すなわち，allegro-express.dxl の利用状態で），Windows なら次のようにタイプ入力する．

```
CG-USER(1): (progn
  (build-lisp-image "sys:allegro.dxl" :case-mode :case-sensitive-lower)
  (when (probe-file "sys:allegro.exe") (delete-file "sys:allegro.exe"))
  (sys:copy-file "sys:allegro-express.exe" "sys:allegro.exe"))
done.
T
```

Linux(ubuntu) なら次のようにタイプ入力する．

```
CG-USER(1): (progn
              (build-lisp-image "sys:allegro.dxl" :case-mode :case-sensitive-lower :restart-app-
function nil)
              (when (probe-file "sys:allegro") (delete-file "sys:allegro"))
              (sys:copy-file "sys:allegro-express" "sys:allegro"))
done.
T
```

しばらくして，このように done. で終了したら，あらためて，Free Express フォルダの中身を見てみよう．新しく，allego.exe（またはallegro） と allegro.dxl があるはずだ．そしたら，Express を終了して今度は allegro.exe（またはallegro） を起動する．スタートメニューには現れていないので，直接フォルダを開いて実行する．こんな風になれば成功だ．

```
cg-user(1): t
t
cg-user(2): nil
nil
cg-user(3): 'foo
foo
cl-user(4): most-positive-fixnum
536870911
cg-user(5): (cons 'a '(b c))
(a b c)
```

ここでは，プロンプトが小文字に変わったこと，入力したシンボルが小文字のままに保たれていて，組み込みのシンボルの値が得られること．小文字で関数も実行できることを確認されたい．

ユーザの中には IDE を嫌って，alisp や mlisp を好みのエディタと連携して使いたいという人がいるかもしれない．そういう人たちのために，ついでにそういう IDE なしの mlisp や alisp の作り方も説明しよう．

Free Express を起動後，その中で次のようにすれば，大文字小文字を区別するが IDE のない mlisp を作成できる．

```
CG-USER(1): (progn
  (build-lisp-image "sys:mlisp.dxl" :case-mode :case-sensitive-lower
                    :include-ide nil :restart-app-function nil
                    :restart-init-function nil)
  (when (probe-file "sys:mlisp.exe") (delete-file "sys:mlisp.exe"))
  (sys:copy-file "sys:allegro-express.exe" "sys:mlisp.exe"))
done.
T
```

同様に，Free Express を起動後，その中で次のようにすれば，旧来のように大文字小文字を区別しないし IDE もない alisp を作成できる．

```
CG-USER(1): (progn
  (build-lisp-image "sys:alisp.dxl" :case-mode :case-insensitive-upper
                    :include-ide nil :restart-app-function nil
                    :restart-init-function nil)
  (when (probe-file "sys:alisp.exe") (delete-file "sys:alisp.exe"))
  (sys:copy-file "sys:allegro-express.exe" "sys:alisp.exe"))
done.
T
```

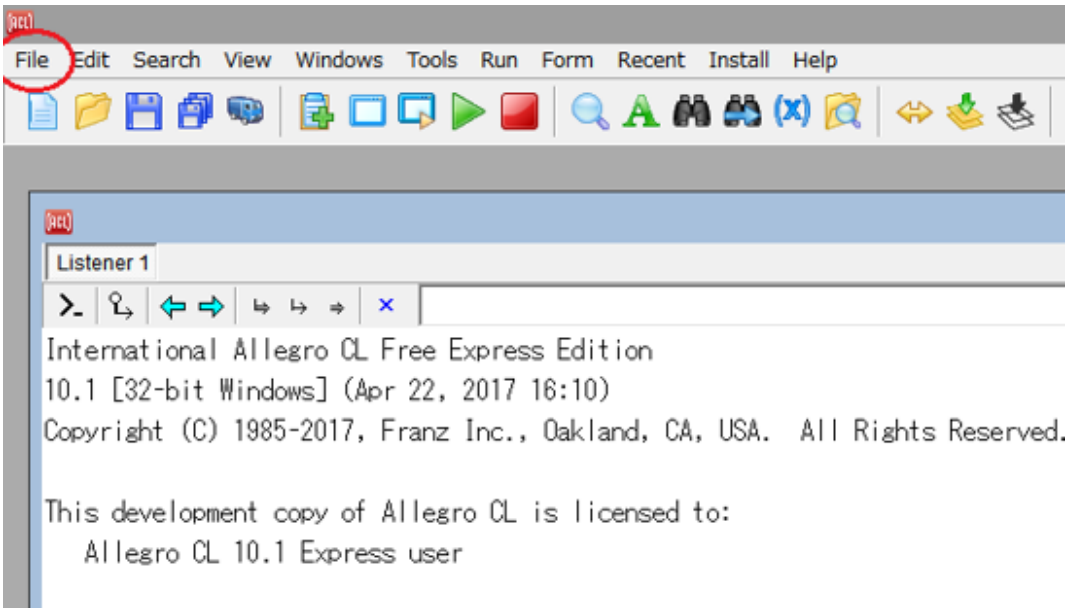
いずれにしても，このままではいちいちフォルダを開けないと Lisp を起動できないので，好みの処理系を起動したら，タスクバーにピン止めしておくのが便利である．

付1.3 ACL with IDE の使い方

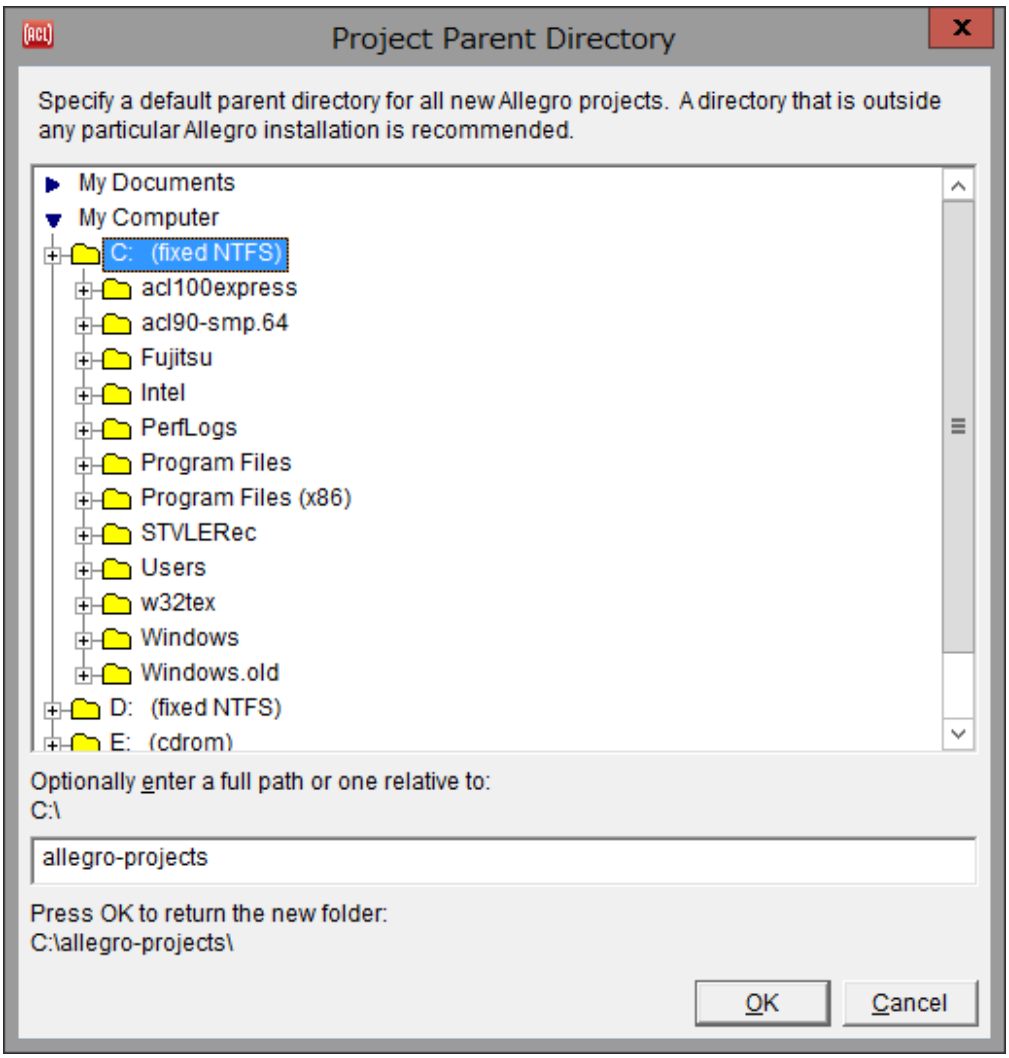
ベテランの Lisper はエディタとして emacs を便利に自分用に仕立てて使うが，ここでは初心者向きに ACL の GUI の使い方について説明する．IDE を使うと REPL でのプリティプリントとコピーペイストが可能になること，プロジェクトやフォルダを指定してファイル中のシンボルの検索ができること，などが非常に便利である．Lisp の CUI としての使い方は，このあとの SBCL で述べる．

ACL プロジェクト

ACL with IDE では Eclipse や Visual Studio などと同様に，プロジェクトとして開発システムをまとめることができる．

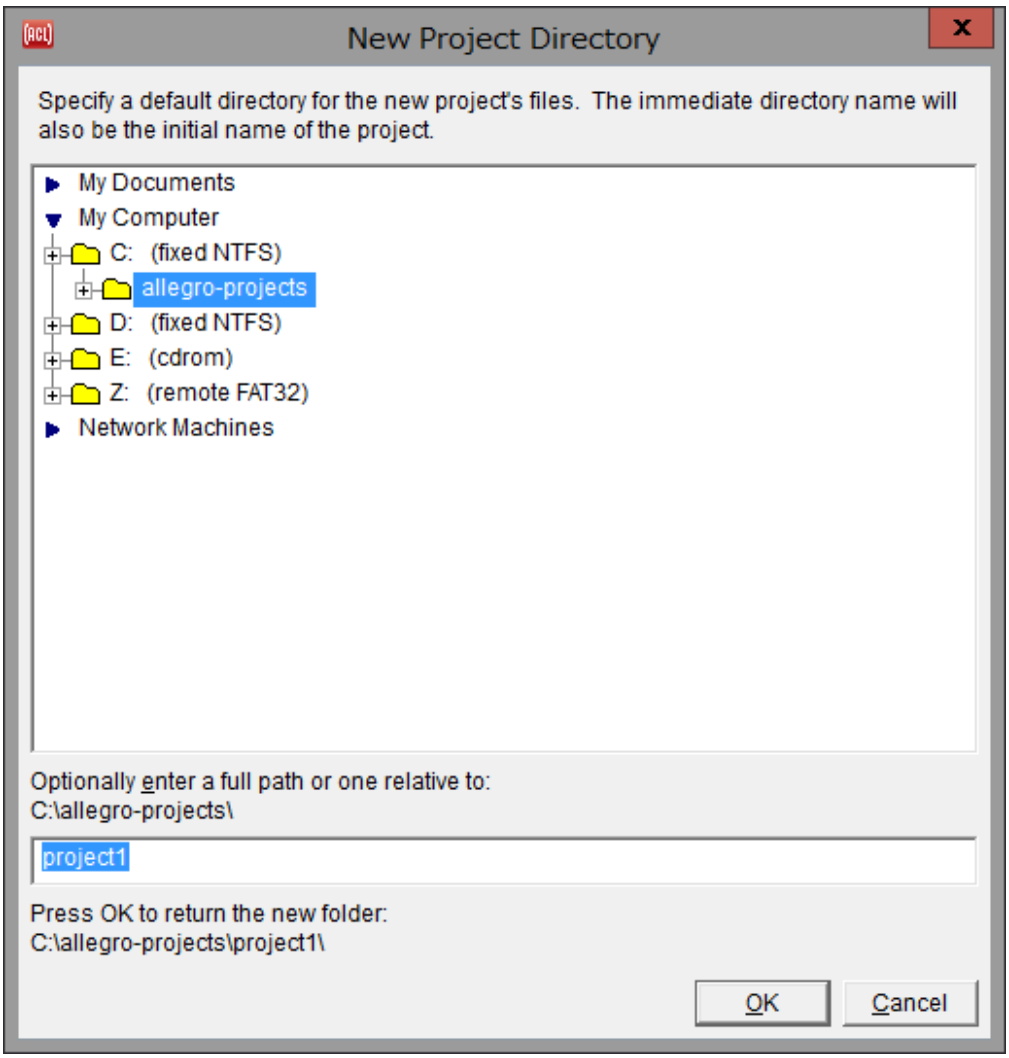


IDE のメニュー選択で，File --> New Project とすると，以下のような画面が現れる．

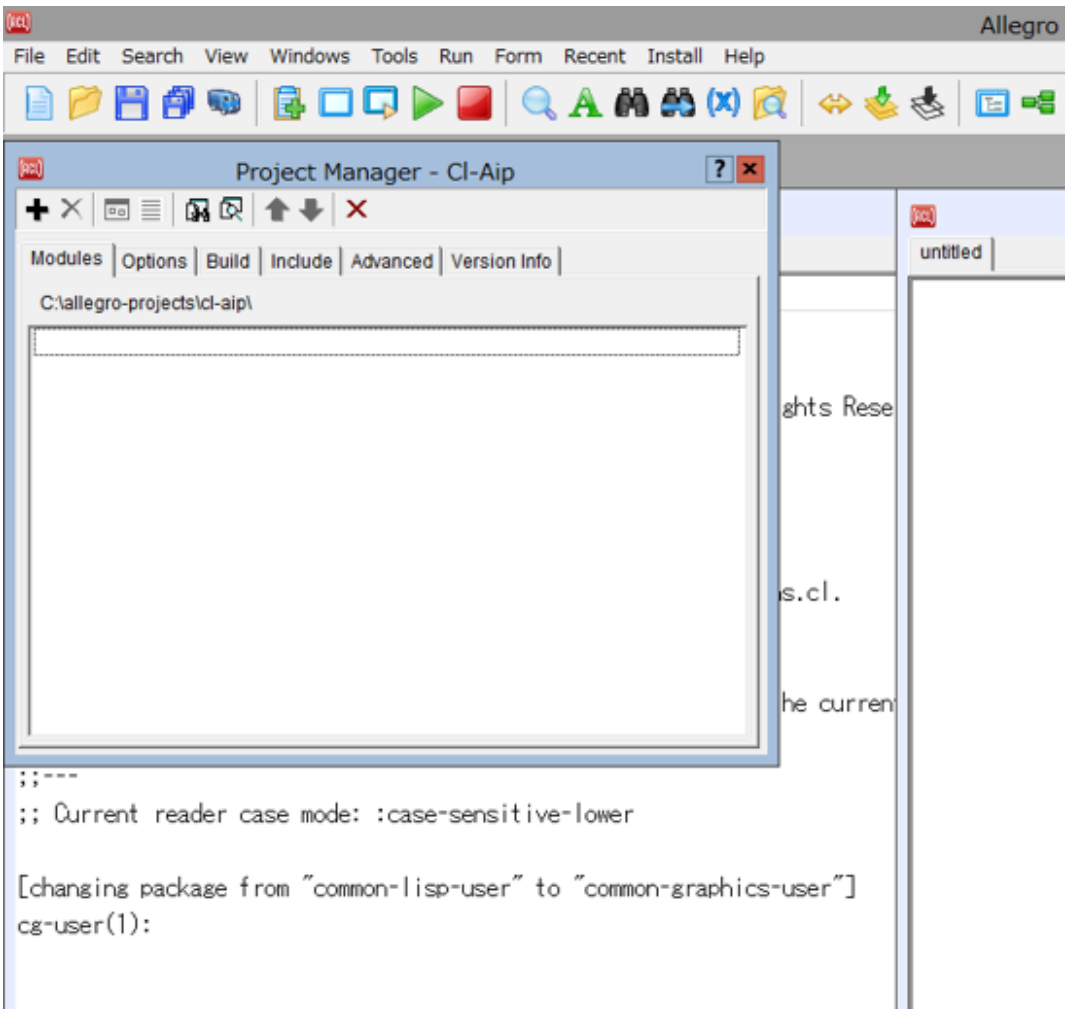


このまま OK を押すと，ACL プロジェクトがまとまって保存される場所が C: の直下の allegro-projects という名称のフォルダになる．これを嫌うのなら適当なフォルダ（たとえばDocumentsフォルダ）を選択して，自分の好きな名前のフォルダ名（たとえばmy-projects）にすればよい．いずれにしても以後そのフォルダがACLのプロジェクトフォルダとなって使われるようになる．

OK を押すとこのような画面になるので，ここで新しく作るプロジェクト名を入力する．



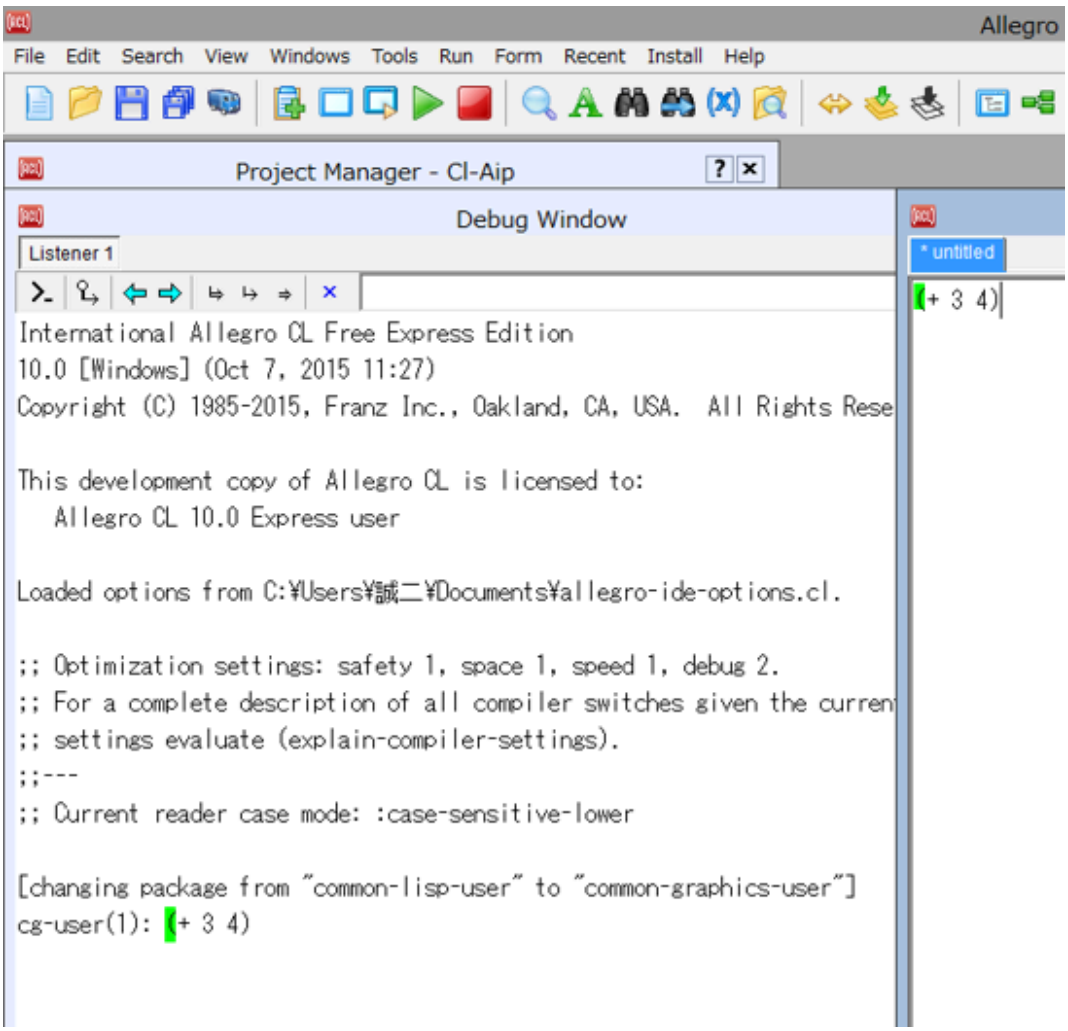
たとえばここで cl-aip という名前のプロジェクト名にして，OK を押せば，以下のような画面が得られる．



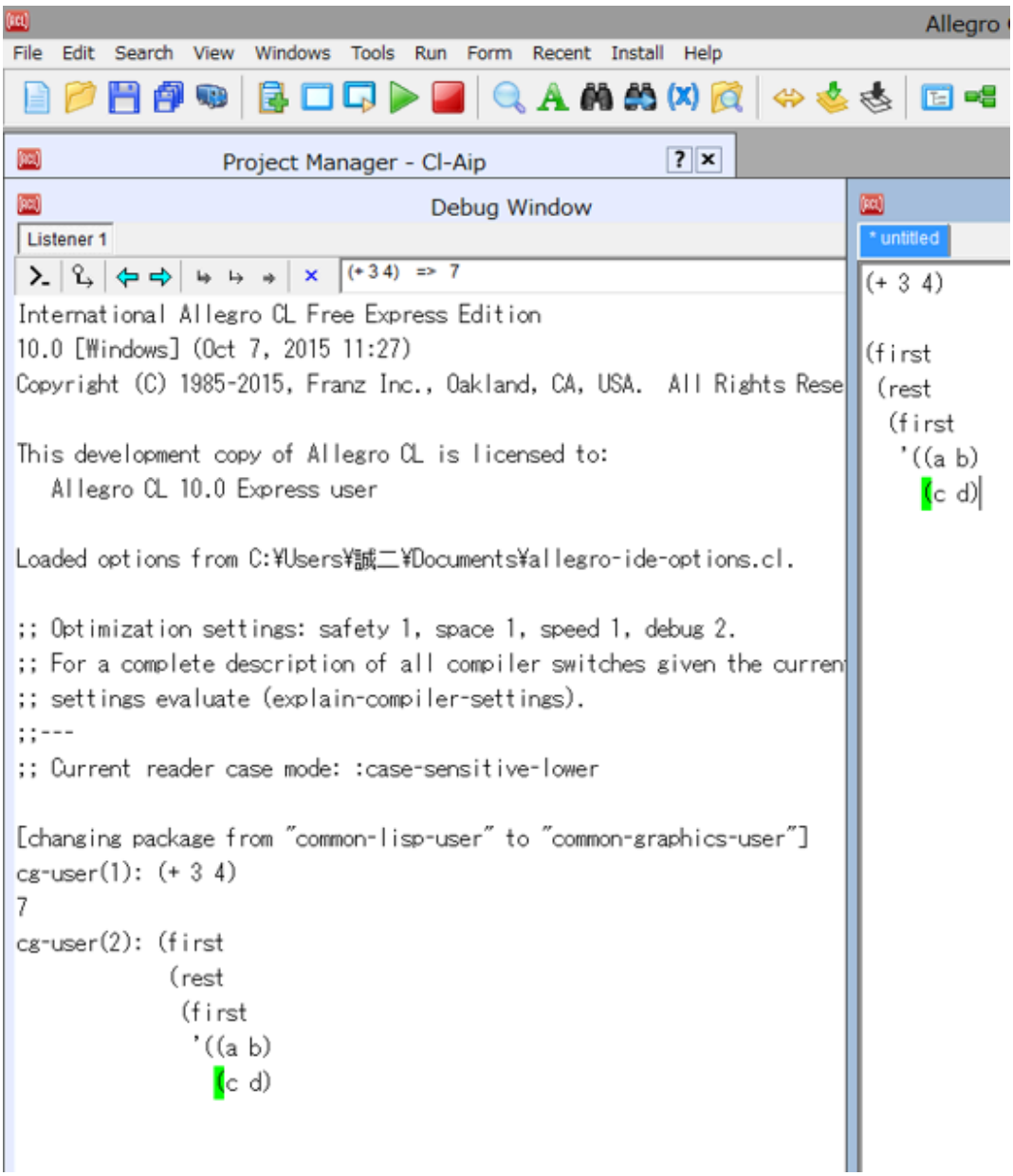
画面左にはREPLペイン，左上には Project Manager ペイン，右側にはエディタペインがある．

エディタと自動インデント

エディタ画面でもREPL画面でも，閉じ括弧をタイプした時に対応する開き括弧が画面表示される．



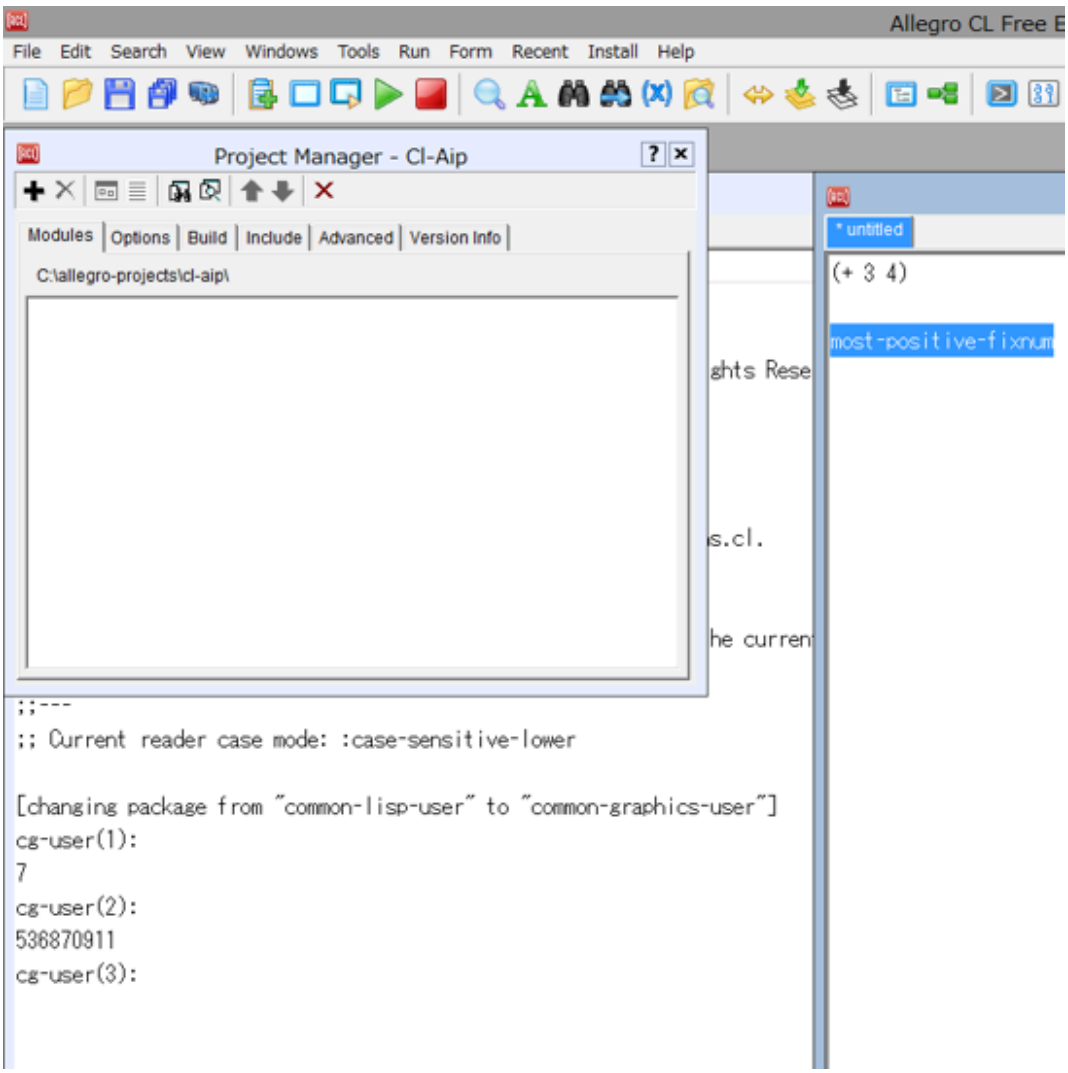
また，エディタ画面でもREPL画面でも，リターンキーで改行するとシステムが自動的に見やすくインデンテーションを付けてくれる．



だから一部の人たちが言う，括弧のお化けみたいな言い方は全く恐れるに足らず，こういうエディタさえあれば括弧の多さで心配するなどということは杞憂であることが了解できる．

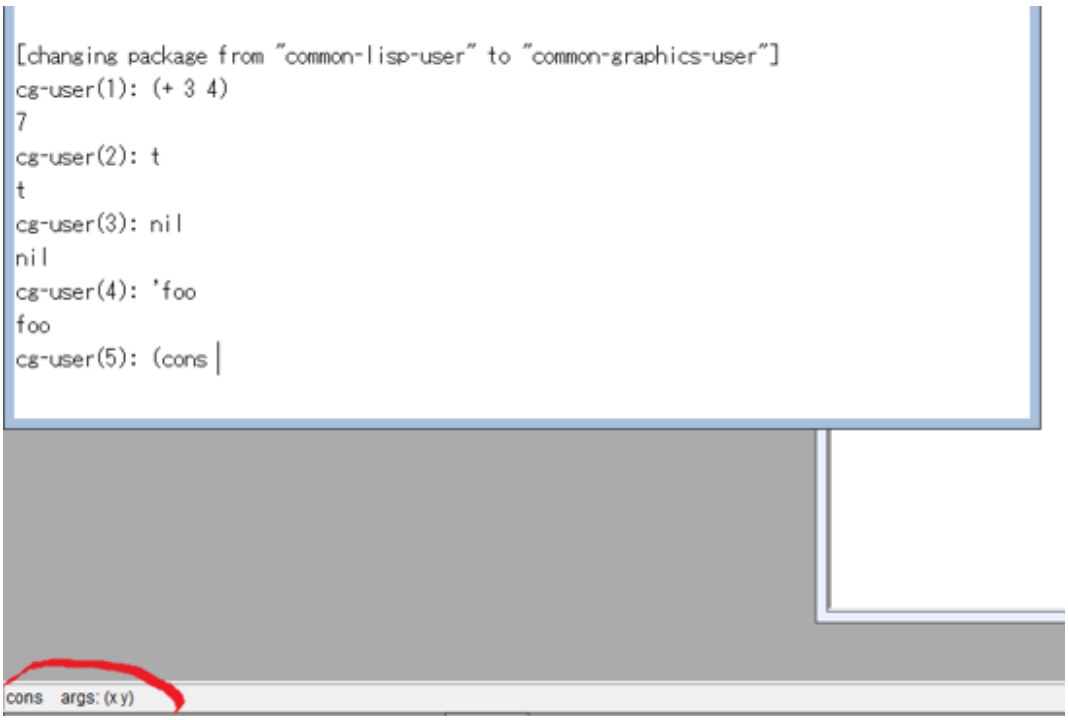
本書籍ではエディタの使い方については本文中では何も触れずに，REPL ですべて入力しているように記述しているが，それはエディタになると各処理系固有の記述が必要になるからである．ACL with IDE ではエディタで編集中に書式を評価したり，エディタから離れることなく関数定義を実行したりすることができる．

試しにエディタ画面上に (+ 3 4) とタイプ入力して，リターンを押さずにその書式が選択されている状態で ctrl-E（Ctrlキーが押された状態でEを押す）を押してみよう．すると REPL 画面上にその評価結果が表示されるはずである．残念ながらシンボルの値については，そのシンボルをマウスで指してクリックすることにより選択してから ctrl-E を押す必要がある．

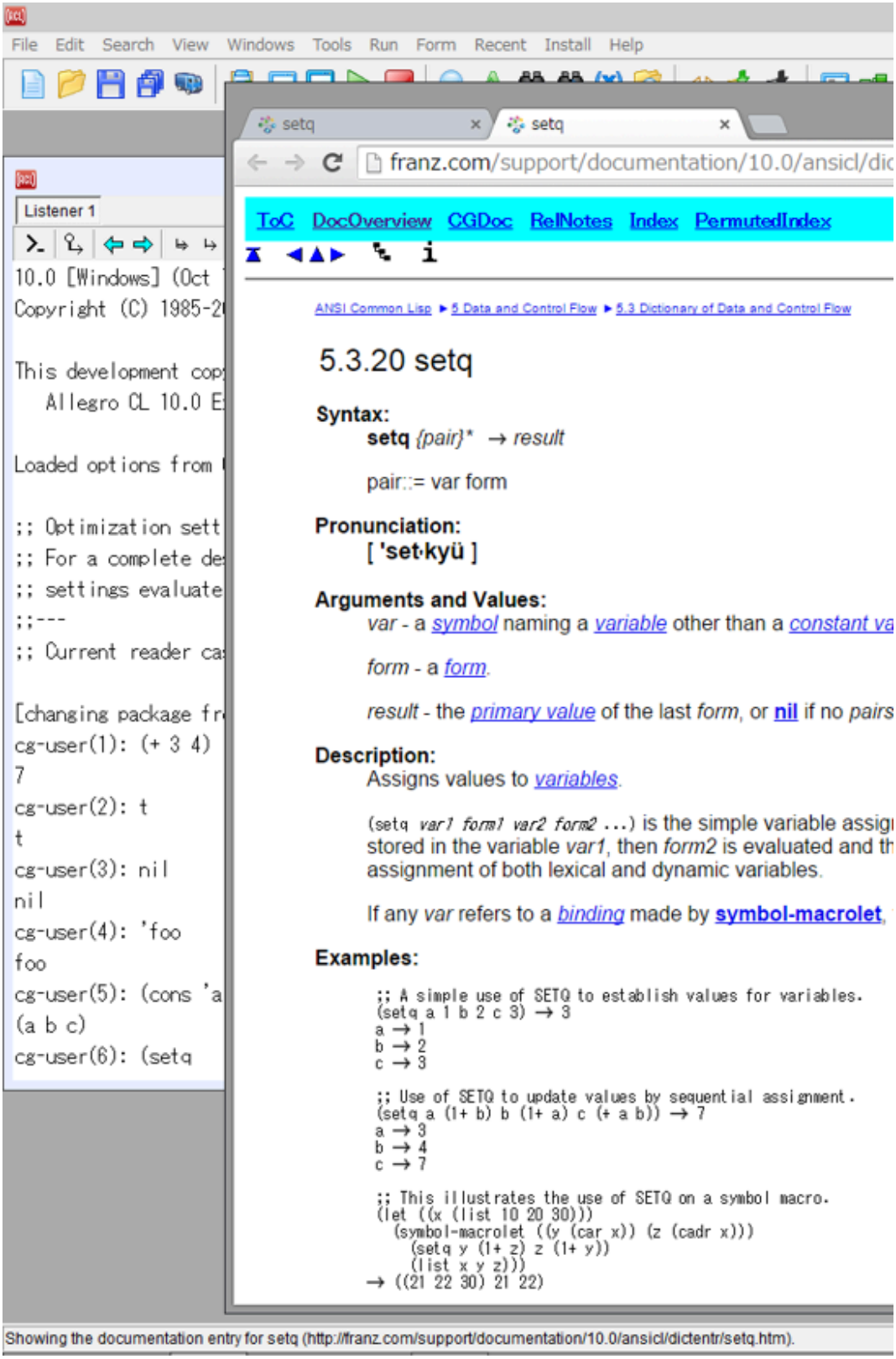


だから実のところ，本書籍のコードをまとめたファイルをダウンロードして，それをエディタで開いておけば，すべてのコードをタイプ入力することなく読み進むことができてしまうわけだ．しかし，これは初心者にはお勧めしない．タイプミスをしたり，関数名を思い出しながらタイプ入力することでしか，初心者の域を脱することはできないと考えるからである．

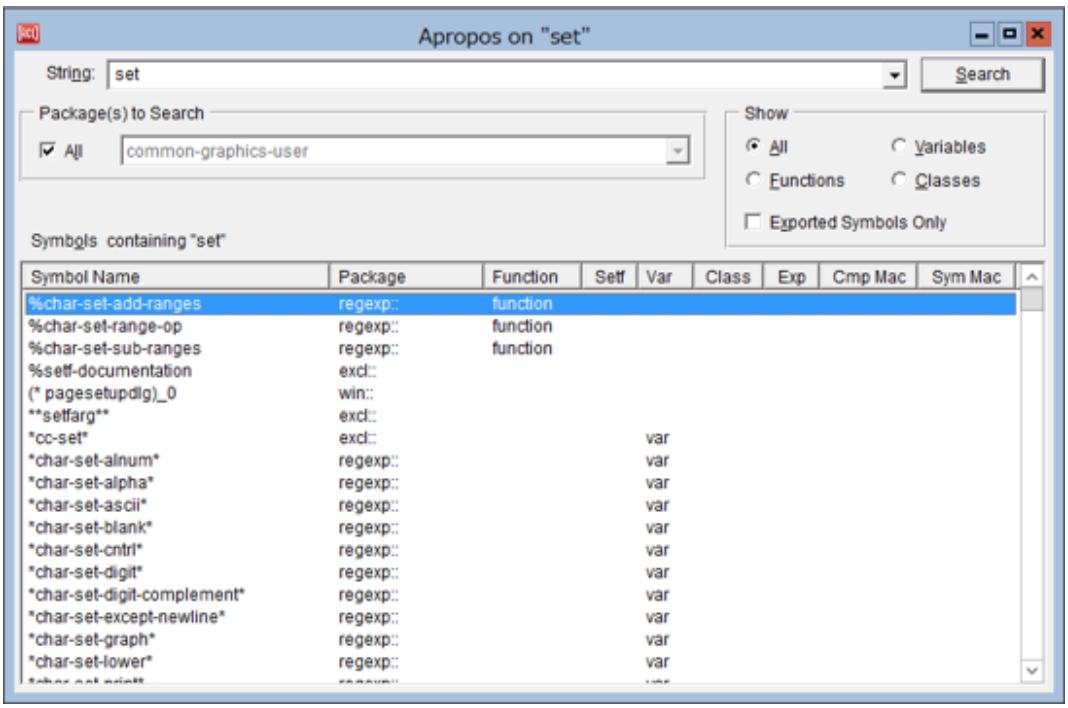
さて，今度はREPLあるいはエディタ画面上でゆっくりとタイプ入力しながら，全画面一番下にあるコラムに注目してほしい．すると評価結果が表示されるのに加えて，関数名を入力した直後にはその引数が表示されることが確認できる．これは非常に便利である．あやふやな関数名を入力してもそれが表示されなければ間違いだと分かるし，さらに引数の順序を逆にするなど間違った呼び出しをしていないかもこの欄で再確認できる．



さらに，関数や変数について詳細情報が欲しければ，そのシンボルを選択しておいてF1キーを押すと関連するドキュメントを閲覧することができる．本書籍の本文中でも多くの場所で Hyperspecへの外部リンクをつけているが，IDEのF1キー押下は直接 Hyerspec を見に行くよりもはるかに便利かつ印刷も綺麗である．

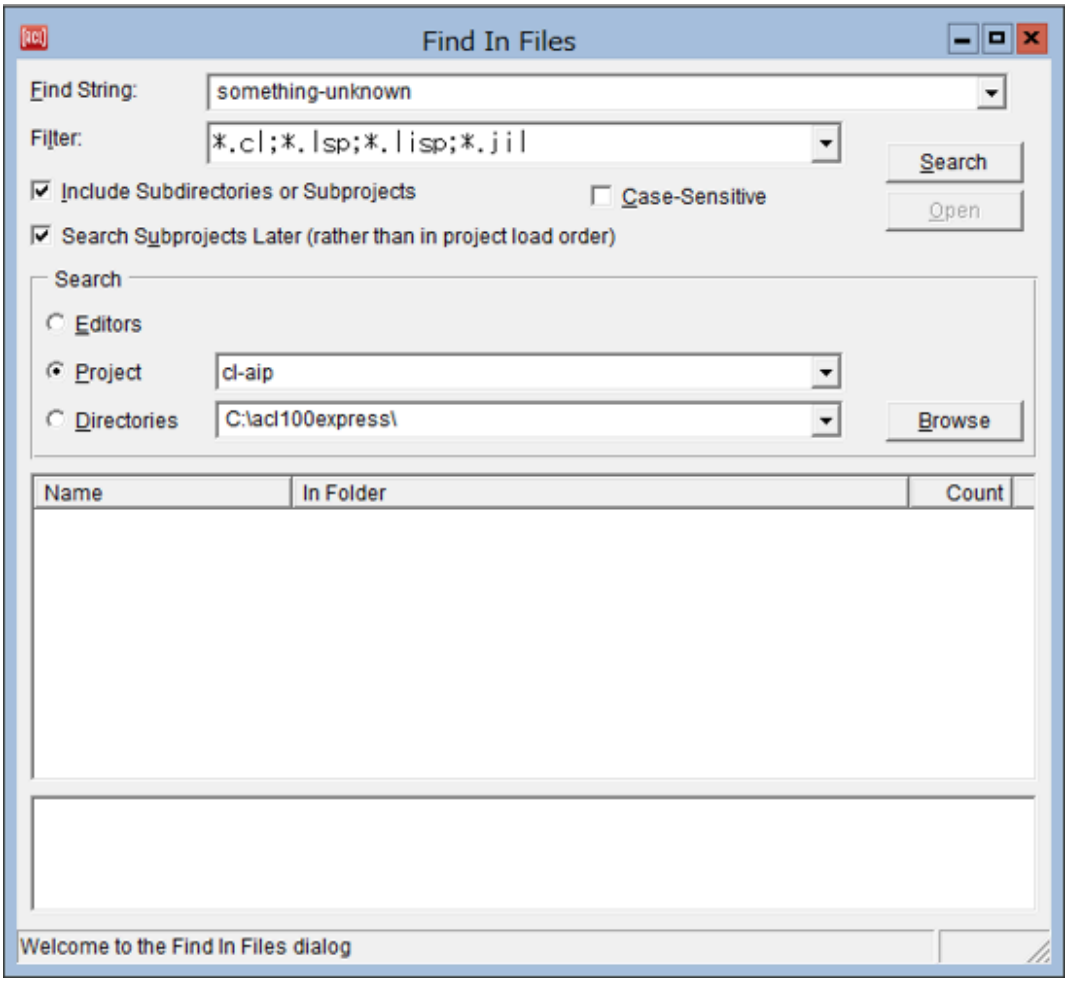


Common Lisp にはシンボルを検索するのに，その印字名に一部でもマッチすればそれをリストする apropos という便利な関数がある．ACL ではそれをメニューからの選択で Apropos ウィンドウで表示できる．ACL のメニュー選択で，Search --> Apropos ... と選択すれば，以下のような画面が表示される．そこで任意の文字列を入力して「Search」ボタンを押せばよい．



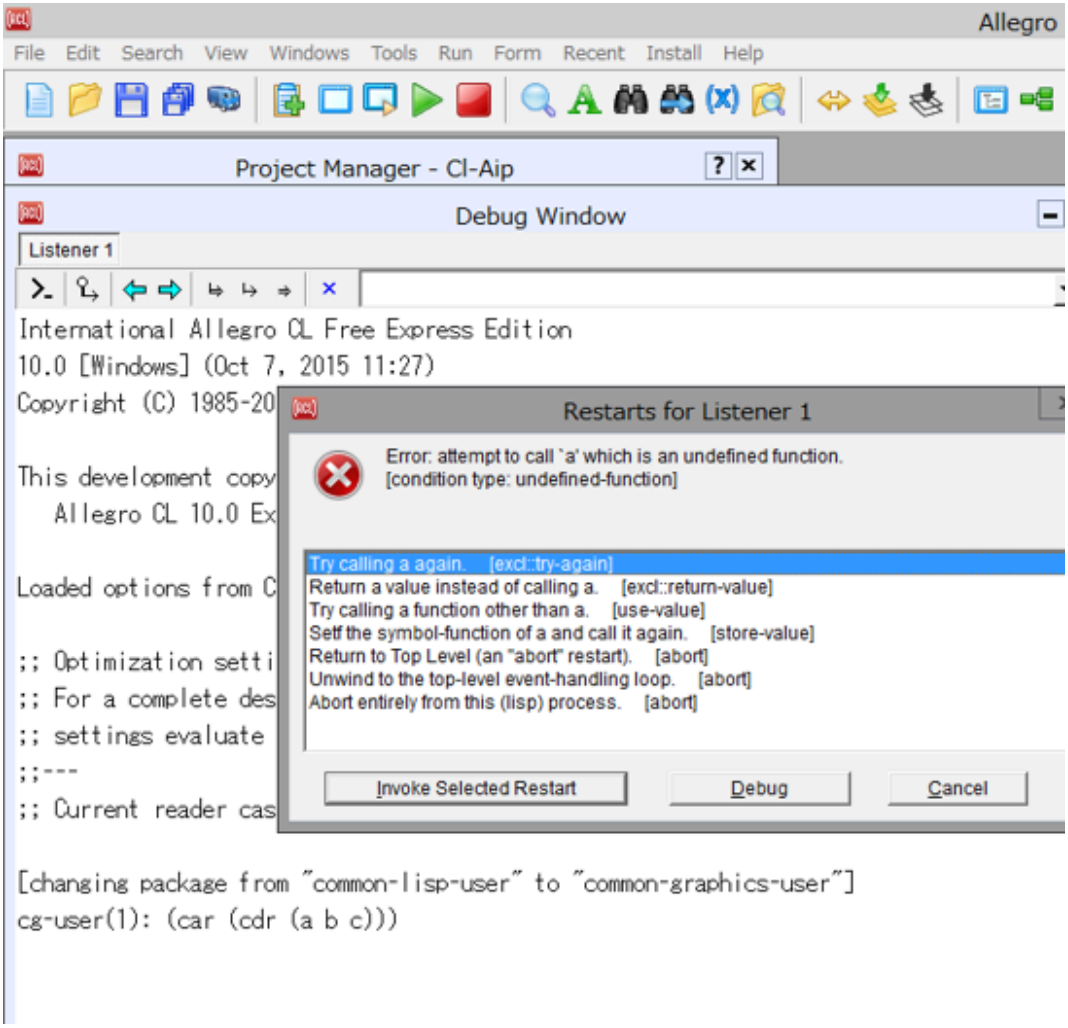
リストされたシンボルはそれがさらにメニューになっている．選択してマウス右クリックすれば，さらにそれについてF1することもできる．

さらに便利なのはプロジェクト単位やディレクトリ単位で，ファイルにあるシンボルをサーチする機能である．ACL に既存のシンボルの探索には apropos が使えるが，自分で開発中のプログラムについてはこの Find In Files を使う．ACL IDE メニューにおいて Search --> Find in Files ... と選択すると次のような画面が現れる．そこで任意の探したい文字列を入力すると，それを含むファイル名が上部に，その周りの情報が下部に表示されるので，これだと思われる部分をクリックすれば，自動的にエディタが開かれる．

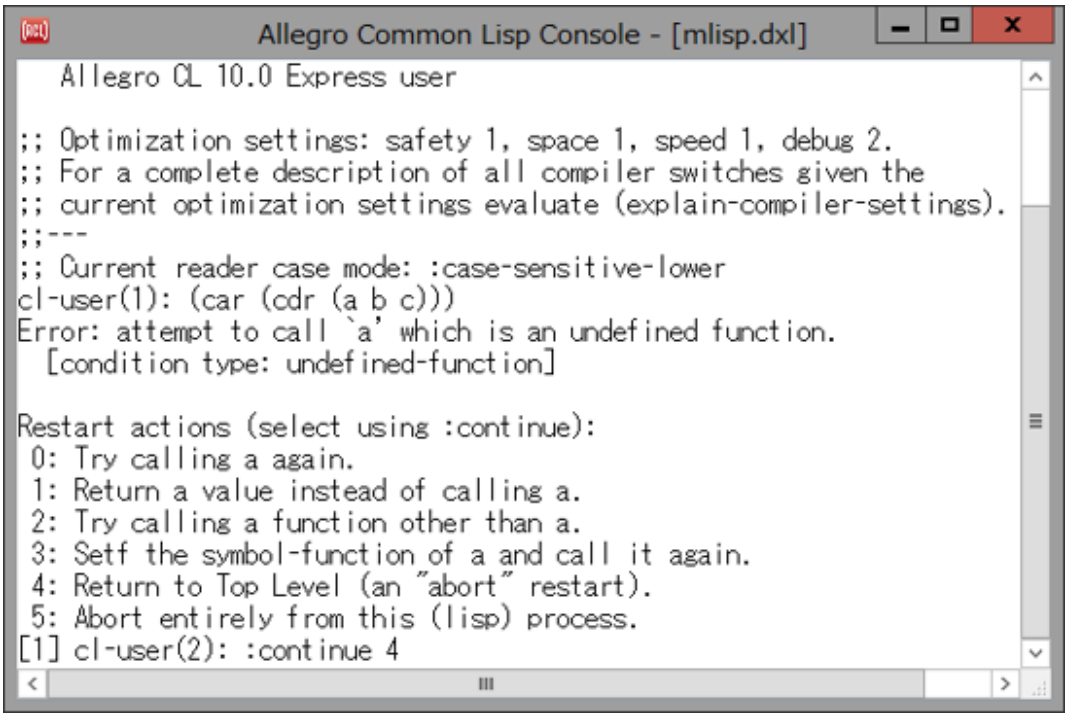


付1.4 ブレーク時の対処方法

ACL with IDE におけるブレーク時の対処方法は，コマンドラインではなく，開いたブレークウィンドウのメニュー選択で対処できるので簡単である．もっとも簡単な方法では，「Cancel」を押せばもとのトップレベルに戻ることができる．



mlisp や alisp の場合には，REPL 環境中でのタイプ入力による対処となる．最も簡単には，メニューからトップレベルに戻るためのメニュー番号を選択して，以下に示すように `:continue《番号》` とタイプ入力する．



付1.5 describeとinspect

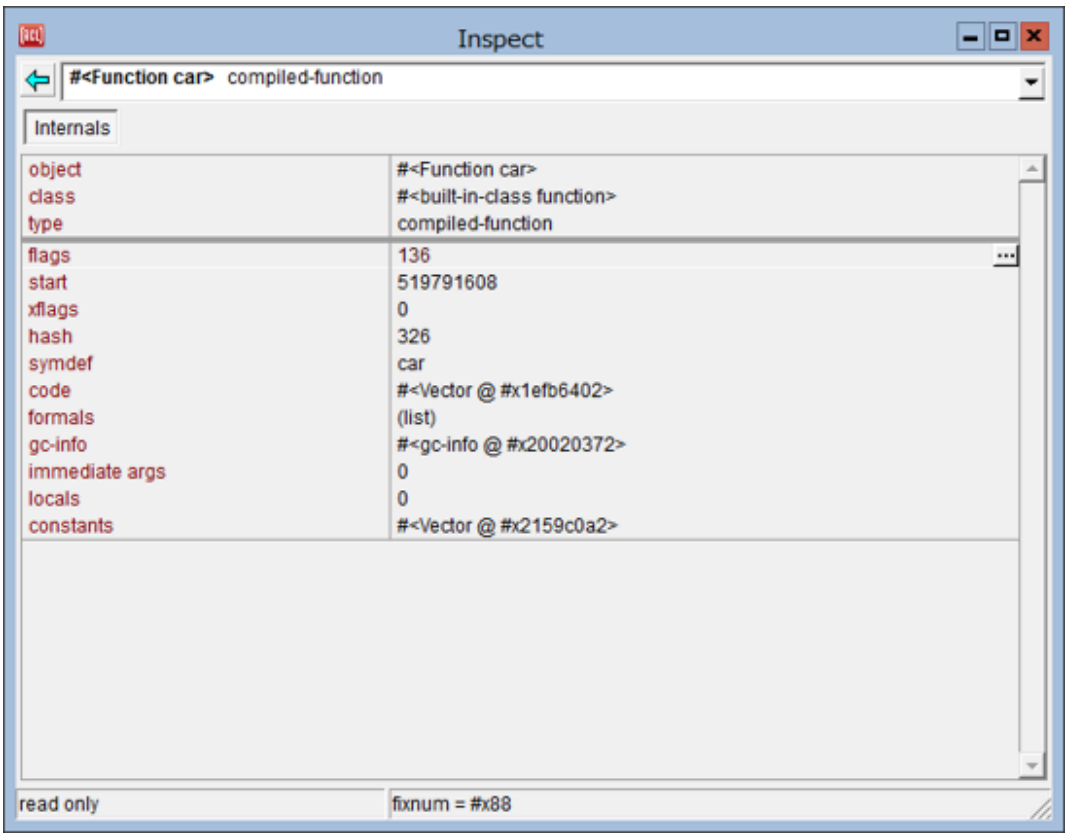
どんなLispオブジェクトでも処理系依存ではあるが，それなりの方法でdescribeによってオブジェクトの情報を *standard-output* に表示できる．ACL mlispでは以下を参照されたい．

```
cl-user(1): (describe #'car)
#<Function car> is a tenured compiled-function.
  The arguments are (list)
cl-user(2): (describe 'foo)
foo is a new symbol.
  It is unbound.
  It is internal in the common-lisp-user package.
```

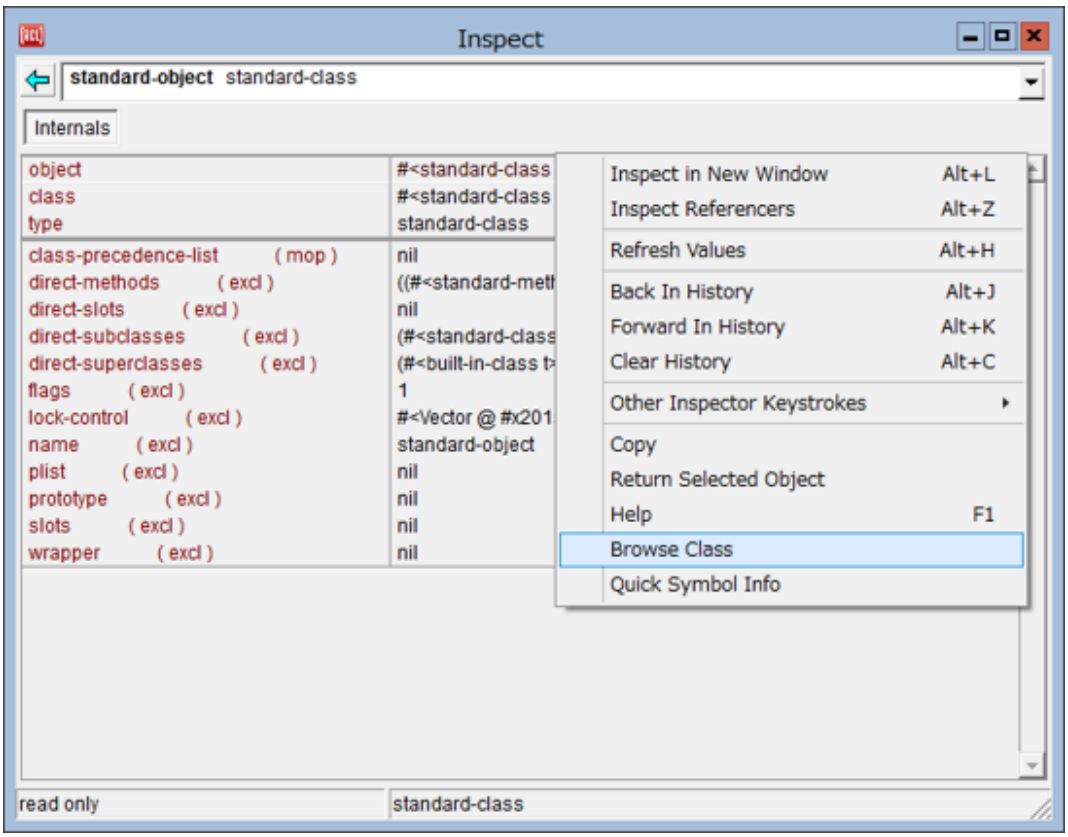
関数inspectではオブジェクトに関するさらに詳細な情報を対話型で得ることができるが，その方法は大いに処理系依存となる．ACLのmlispやalispでは，インスペクト用のコマンドが用意されている．それを使いこなすには，こちらを参照されたい．ここでは，インスペクト後の終了方法だけ記述しておく．

```
cl-user(3): (inspect #'car)
A tenured #<Function car>
  lambda-list: (list)
   0 type ---------> Bit field: #x08
   1 flags --------> Bit field: #x88
   2 start --------> Bit field: #x1efb63f8
   3 xflags -------> Bit field: #x00
   4 hash ---------> Bit field: #x146
   5 symdef -------> symbol car
   6 code ---------> simple code vector (31) = #(35157 33765 10476 ...)
   7 formals ------> (list), a proper list with 1 element
   8 gc-info ------> #<gc-info @ #x2001fd12>
   9 immed-args ---> fixnum 0 [#x00000000]
  10 locals -------> fixnum 0 [#x00000000]
[1i] cl-user(4): :i q
#<Function car>
```

Allegro with IDEでは関数inspectを呼び出すと，下図のようなウィンドウが現れて，それぞれの項目がメニューとなっているので，コマンドを知らなくてもメニュー項目を選択することでいくらでも詳細内容を調べることができる．



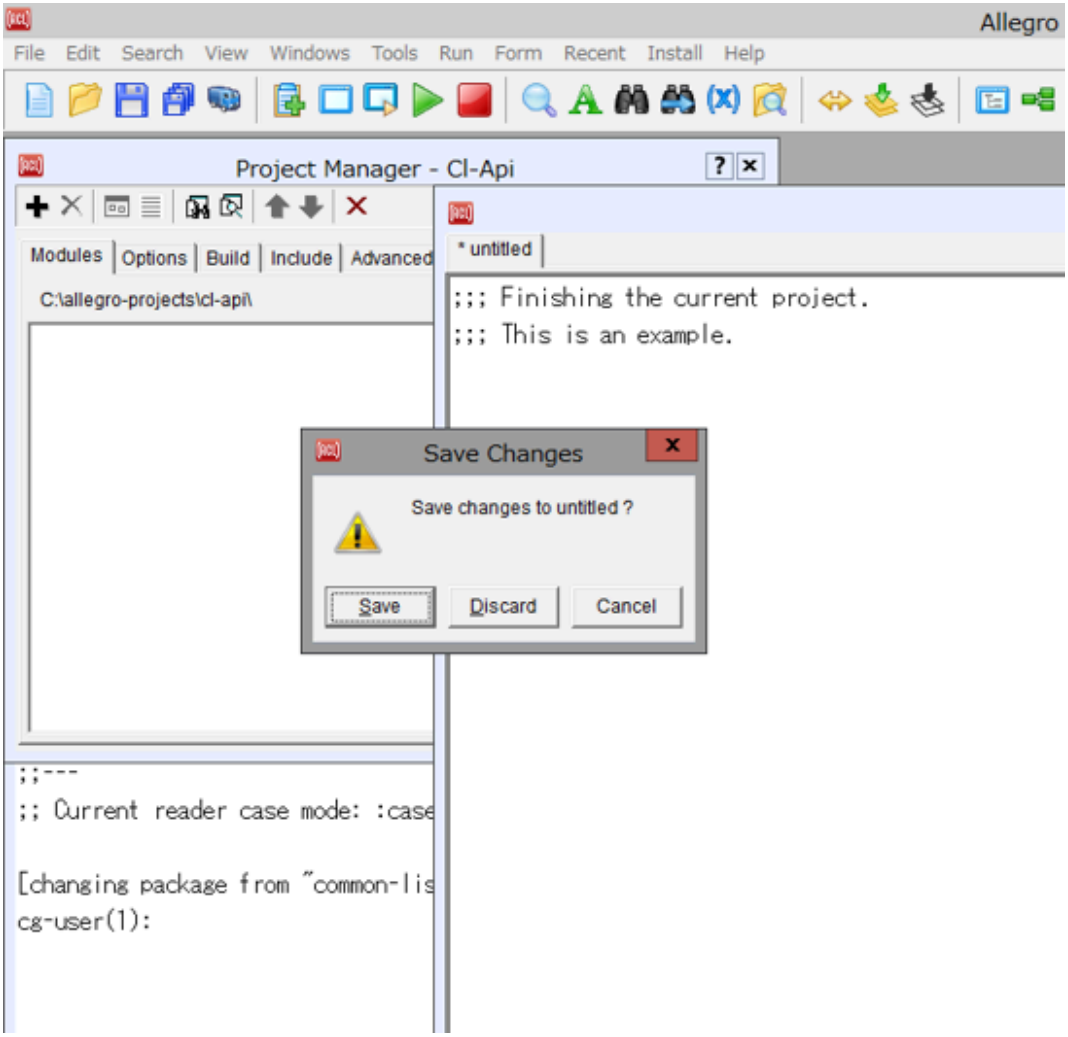
特に，CLOSクラスに関してインスペクトをすると，そのメニューから該当クラスについてのブラウズメニューを選択して，CLOSクラスの階層構造を表示したりスロット構造が分かったり，メソッド定義の場所や継承関係をグラフで表示することができる．下図はAllegro with IDEにおいて`(inspect (find-class 'standard-class))`としてインスペクト・ウィンドウを表示させ，`#<standard-class standard-object>`を選んで，右マウスクリックでメニューを表示させ，ブラウズしようとしているところである．同様のことを試して，色々操作しながら是非その素晴らしさを体感してほしい．



付1.6 システムの終わり方

一般に Lisp システムを終わらせるには，(exit) とか (quit) と入力する．中には (bye) といういうシステムもある．ACL は Unix/Linux ライクなコマンドを受け付けることができて，:cd とか :pwd などが使えるが，同様に :exit というコマンドでも終了できる．

プロジェクトを開いていて，untitled ファイルにコードを記述した場合には，それを閉じようとするとそれを保存するかどうか問われる．「Save」ボタンを押すと現在のプロジェクトに保存するかどうかを問われるので，「Yes」を押して，ファイル名として適当な名前をつけてプロジェクトに保存する．


Allegro
File Edit Search View Windows Tools Run Form Recent Install Help
Project Manager - Cl-Api
Modules Options Build Include Advanced
C:\allegro-projects\cl-api\
* untitled
;;; Finishing the current project.
;;; This is an example.
Save Changes
Save changes to untitled ?
Save
Discard
Cancel
;;---
;; Current reader case mode: :case
[changing package from "common-lis
cg-user(1):

付録2 SBCL のインストール

付2.1 Ubuntu にパッケージとしてインストール

SBCL を一番簡単にインストールする方法は，Debian や Ubuntu など Debian系の Linux に標準インストールするやり方である．ただし，Debian系でも openSUSE では YaST で SBCL を標準インストールできない．CentOS や Fedora のような RedHat系では標準パッケージとしては SBCL が提供されない．Debian 系標準パッケージへの SBCL 取り込みはどうしてもワンテンポ・ツーテンポ遅くなるので，最新版を使いたい場合や RedHat系 Linux にインストールしたい場合には，直接本家のサイトからダウンロード・インストールすることになるが，本書の範囲では最新版をインストールするほどのことはない．以下の説明では Ubuntu 系に標準パッケージとしてインストールする場合を説明したのち、最新版の手動インストール方法を説明する．

Ubuntu デスクトップ版であれば，Synaptic Package Manager をインストールして GUI を使ってインストールすることもできるが，そのためだけにわざわざ Synaptic Package Manager をインストールするまでのことはない。サーバー版と同様にすればよい．サーバー版であればroot 権限で「apt-get upgrade」を使ってシステムを最新状態にしてから，apt-get コマンドを使って SBCL と emacs 関連パッケージを次のようにインストールする．

```
seiji@NLU:~$ lsb_release -idrc
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 24.04.2 LTS
Release:        24.04
Codename:       noble
seiji@NLU:~$ sudo apt update && sudo apt upgrade -y
[sudo] seiji のパスワード:

   [...]

done
seiji@NLU:~$ sudo apt install emacs cl-swank slime sbcl -y
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています... 完了
状態情報を読み取っています... 完了
以下の追加パッケージがインストールされます:
  binutils binutils-common binutils-x86-64-linux-gnu dh-elpa-helper emacs-bin-common emacs-
common emacs-el emacs-gtk gsasl-common guile-3.0-libs libasan8
  libbinutils libctf-nobfd0 libctf0 libgc1 libgcc-14-dev libgccjit0 libgprofng0 libgsasl18
libgssglue1 libhwasan0 libitm1 liblsan0 libm17n-0 libmailutils9t64
  libmysqlclient21 libnsl2 libntlm0 libotf1 libpq5 libquadmath0 libsframe1 libtree-sitter0
libtsan2 libubsan1 libwebpdecoder3 m17n-db mailutils mailutils-common
  mysql-common postfix
提案パッケージ:
  binutils-doc gprofng-gui emacs-common-non-dfsg ncurses-term m17n-docs gawk mailutils-mh
mailutils-doc postfix-cdb postfix-doc postfix-ldap postfix-lmdb
  postfix-mta-sts-resolver postfix-mysql postfix-pcre postfix-pgsql postfix-sqlite procmail
sasl2-bin | dovecot-common sbcl-doc sbcl-source hyperspec
以下のパッケージが新たにインストールされます:
  binutils binutils-common binutils-x86-64-linux-gnu cl-swank dh-elpa-helper emacs emacs-bin-
common emacs-common emacs-el emacs-gtk gsasl-common guile-3.0-libs
```

```
  libasan8 libbinutils libctf-nobfd0 libctf0 libgc1 libgcc-14-dev libgccjit0 libgprofng0
libgsasl18 libgssglue1 libhwasan0 libitm1 liblsan0 libm17n-0
  libmailutils9t64 libmysqlclient21 libnsl2 libntlm0 libotf1 libpq5 libquadmath0 libsframe1
libtree-sitter0 libtsan2 libubsan1 libwebpdecoder3 m17n-db mailutils
  mailutils-common mysql-common postfix sbcl slime
アップグレード: 0 個、新規インストール: 45 個、削除: 0 個、保留: 0 個。
98.6 MB のアーカイブを取得する必要があります。
この操作後に追加で 399 MB のディスク容量が消費されます。
取得:1 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 libnsl2 amd64 1.3.0-3build3 [41.4
kB]

   [...]

Postfix (main.cf) was not set up.  Start with
  cp /usr/share/postfix/main.cf.debian /etc/postfix/main.cf
.  If you need to make changes, edit /etc/postfix/main.cf (and others) as
needed.  To view Postfix configuration values, see postconf(1).

After modifying main.cf, be sure to run 'systemctl reload postfix'.

Created symlink /etc/systemd/system/multi-user.target.wants/postfix.service →
/usr/lib/systemd/system/postfix.service.

   [...]

Install emacsen-common for emacs
emacsen-common: Handling install of emacsen flavor emacs
Install dictionaries-common for emacs
install/dictionaries-common: Byte-compiling for emacsen flavour emacs
Install slime for emacs
install/slime-2.28: Handling install of emacsen flavor emacs
install/slime-2.28: byte-compiling for emacs
emacs (1:29.3+1-1ubuntu2) を設定しています ...
seiji@NLU:~$ sbcl
This is SBCL 2.2.9.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

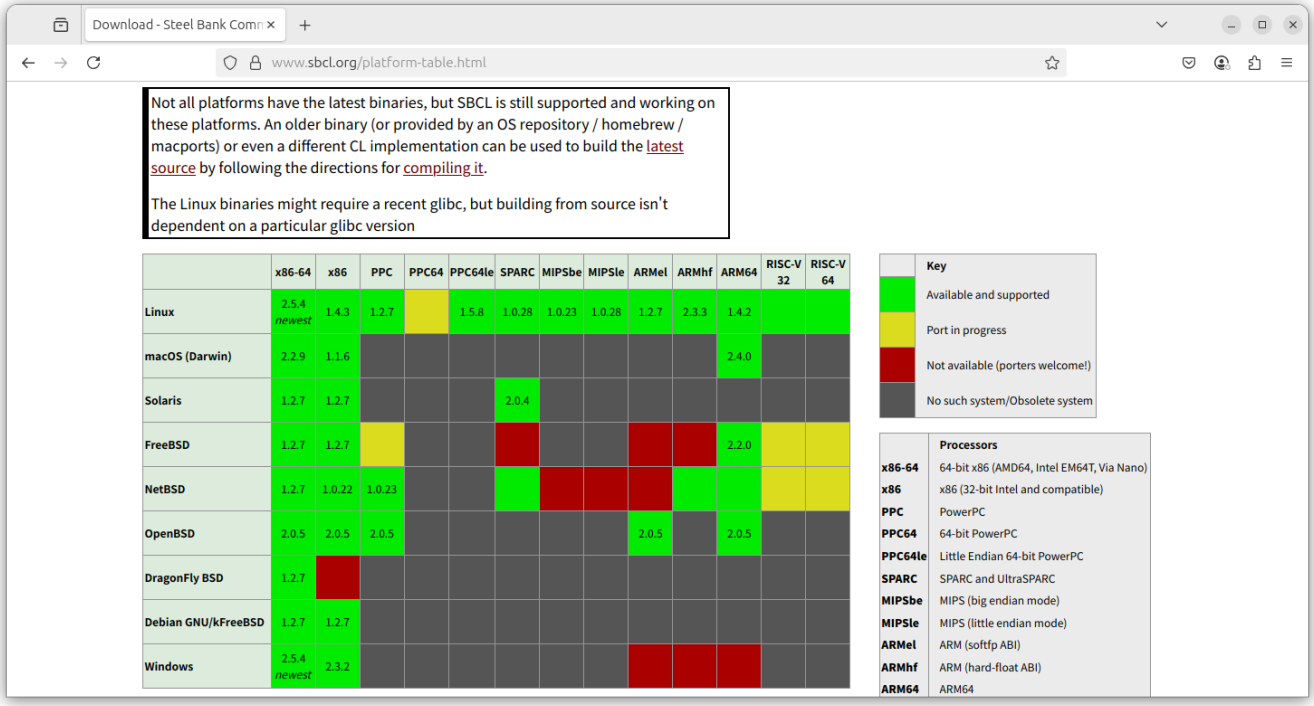
SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* t
T
* nil
NIL
* 'foo
FOO
* (exit)
seiji@NLU:~$
```

なぜかメールサーバーである Postfix がインストールされようとする．上記は No Configuration 設定なしを選択した結果である．ご覧のように，ここでインストールされた SBCL はバージョン2.2.9 である．この原稿執筆時における最新バージョンは2.5.4であるが，本書（Common Lisp と人工知能プログラミング）の記述範囲ではこれで十分であり，わざわざ最新版をインストールしなくてもよい．ただし，後続の自然言語理解の書籍では最新版が必要であり，次のバイナリーパッケージのインストール方法にしたがってインストールされたい．

付2.2 最新バイナリーパッケージのインストール

SBCL はソースから最新版をビルドすることもできるが、その場合でもビルドするための Common Lisp 実行環境が必要である．ここではビルド済みのバイナリ最新版の手動インストール方法を紹介する．

まずはSBCLプラットホームテーブルに行き，Linux 行の中から望みのバージョンのものを見つけ，該当セルをクリックしてバイナリシステムをダウンロードする．



ダウンローされたシステムはホームのダウンロードディレクトリに入る．その圧縮ファイルを次のように解凍し，標準的には以下のように **install.sh** を実行する．

```
seiji@NLU:~/ダウンロード$ ls
sbcl-2.5.4-x86-64-linux-binary.tar.bz2
seiji@NLU:~/ダウンロード$ tar jxf sbcl-2.5.4-x86-64-linux-binary.tar.bz2
seiji@NLU:~/ダウンロード$ ls
sbcl-2.5.4-x86-64-linux  sbcl-2.5.4-x86-64-linux-binary.tar.bz2
seiji@NLU:~/ダウンロード$ cd sbcl-2.5.4-x86-64-linux
seiji@NLU:~/ダウンロード/sbcl-2.5.4-x86-64-linux$ sudo sh install.sh
[sudo] seiji のパスワード:
/usr/local

SBCL has been installed:
 binary /usr/local/bin/sbcl
 core and contribs in /usr/local/lib/sbcl/

Documentation:
 man /usr/local/share/man/man1/sbcl.1
seiji@NLU:~/ダウンロード/sbcl-2.5.4-x86-64-linux$ cd
seiji@NLU:~$ sbcl
This is SBCL 2.5.4, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
```

```
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* t
T
* nil
NIL
* 'foo
FOO
* (exit)
seiji@NLU:~$
```

この原稿執筆時点では 2.5.4 がバイナリの最新版である．もしSBCLプラットフォームテーブルでより新しい版を見つけた時でも，同様の方法でインストールすればよい．

付2.3 SBCL ライブラリ sb-aclrepl を要求する

SBCL のライブラリは，/usr/local/lib/sbcl/ に保存されているが，ここにあるライブラリは単純に次のように require して使うことができる．

```
* (require :sb-aclrepl)

("SB-ACLREPL")
CL-USER(1): t

T
CL-USER(2): nil

NIL
CL-USER(3): :pwd
Lisp's current working directory is "/home/seiji/".
CL-USER(4): :cd ..
/home/
CL-USER(5): :cd ~
/home/seiji/
CL-USER(6): :exit
seiji@NLU:~$
```

これで見かけ上，SBCL も ACL と同じようになった．第1部のプログラム説明では，常にこのライブラリが有効に使用されているものと理解されたい（本当はACLでの結果であるが）．

付2.4 ライブラリマネージャ quicklisp を利用する

以前には Common Lisp には CPAN のような総合的で便利なライブラリ・リポジトリがないというのが問題とされていた．しかしこの状況は最近 quicklisp の登場によって劇的に改善された．本書においても，SBCL ではこの quicklisp を標準のライブラリ・リポジトリとして利用することにする．

Quicklisp 本家のサイトにあるインストール手続きに従って，以下のようにインストールする．ただし，後々のためにホームページ直下に common-lisp ディレクトリを作り，sbcl のプログラムはすべてその中に入れるようにする．また curl を利用するので，まだインストールしていない場合には，最初に curl をインストールしてほしい．

```
seiji@NLU:~$ mkdir common-lisp
seiji@NLU:~$ cd common-lisp
seiji@NLU:~/common-lisp$ curl
コマンド 'curl' が見つかりません。次の方法でインストールできます:
sudo snap install curl  # version 8.13.0, or
sudo apt  install curl  # version 8.5.0-2ubuntu10.6
他のバージョンについては 'snap info curl' を確認してください。
seiji@NLU:~/common-lisp$ sudo snap install curl
[sudo] seiji のパスワード:
curl 8.13.0 from hideo aoyama (aoilinux) installed
seiji@NLU:~/common-lisp$ curl -O https://beta.quicklisp.org/quicklisp.lisp
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 57144  100 57144    0     0   373k      0 --:--:-- --:--:-- --:--:--  374k
seiji@NLU:~/common-lisp$ ls
quicklisp.lisp
seiji@NLU:~/common-lisp$ sbcl --load quicklisp.lisp
This is SBCL 2.5.4, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.

  ==== quicklisp quickstart 2015-01-28 loaded ====

    To continue with installation, evaluate: (quicklisp-quickstart:install)

    For installation options, evaluate: (quicklisp-quickstart:help)

* (quicklisp-quickstart:install)
; Fetching #<URL "http://beta.quicklisp.org/client/quicklisp.sexp">
; 0.82KB
==================================================
839 bytes in 0.00 seconds (819.34KB/sec)
; Fetching #<URL "http://beta.quicklisp.org/client/2021-02-13/quicklisp.tar">
; 260.00KB
==================================================
266,240 bytes in 0.02 seconds (16250.00KB/sec)
; Fetching #<URL "http://beta.quicklisp.org/client/2021-02-11/setup.lisp">
; 4.94KB
==================================================
5,057 bytes in 0.00 seconds (0.00KB/sec)
; Fetching #<URL "http://beta.quicklisp.org/asdf/3.2.1/asdf.lisp">
; 628.18KB
==================================================
643,253 bytes in 0.03 seconds (24160.64KB/sec)
; Fetching #<URL "http://beta.quicklisp.org/dist/quicklisp.txt">
; 0.40KB
==================================================
408 bytes in 0.00 seconds (0.00KB/sec)
Installing dist "quicklisp" version "2024-10-12".
; Fetching #<URL "http://beta.quicklisp.org/dist/quicklisp/2024-10-12/releases.txt">
; 553.52KB
==================================================
566,809 bytes in 0.03 seconds (17855.63KB/sec)
; Fetching #<URL "http://beta.quicklisp.org/dist/quicklisp/2024-10-12/systems.txt">
; 429.87KB
==================================================
```

```
440,182 bytes in 0.03 seconds (16533.28KB/sec)

  ==== quicklisp installed ====

    To load a system, use: (ql:quickload "system-name")

    To find systems, use: (ql:system-apropos "term")

    To load Quicklisp every time you start Lisp, use: (ql:add-to-init-file)

    For more information, see http://www.quicklisp.org/beta/

NIL
* (ql:add-to-init-file)
I will append the following lines to #P"/home/seiji/.sbclrc":

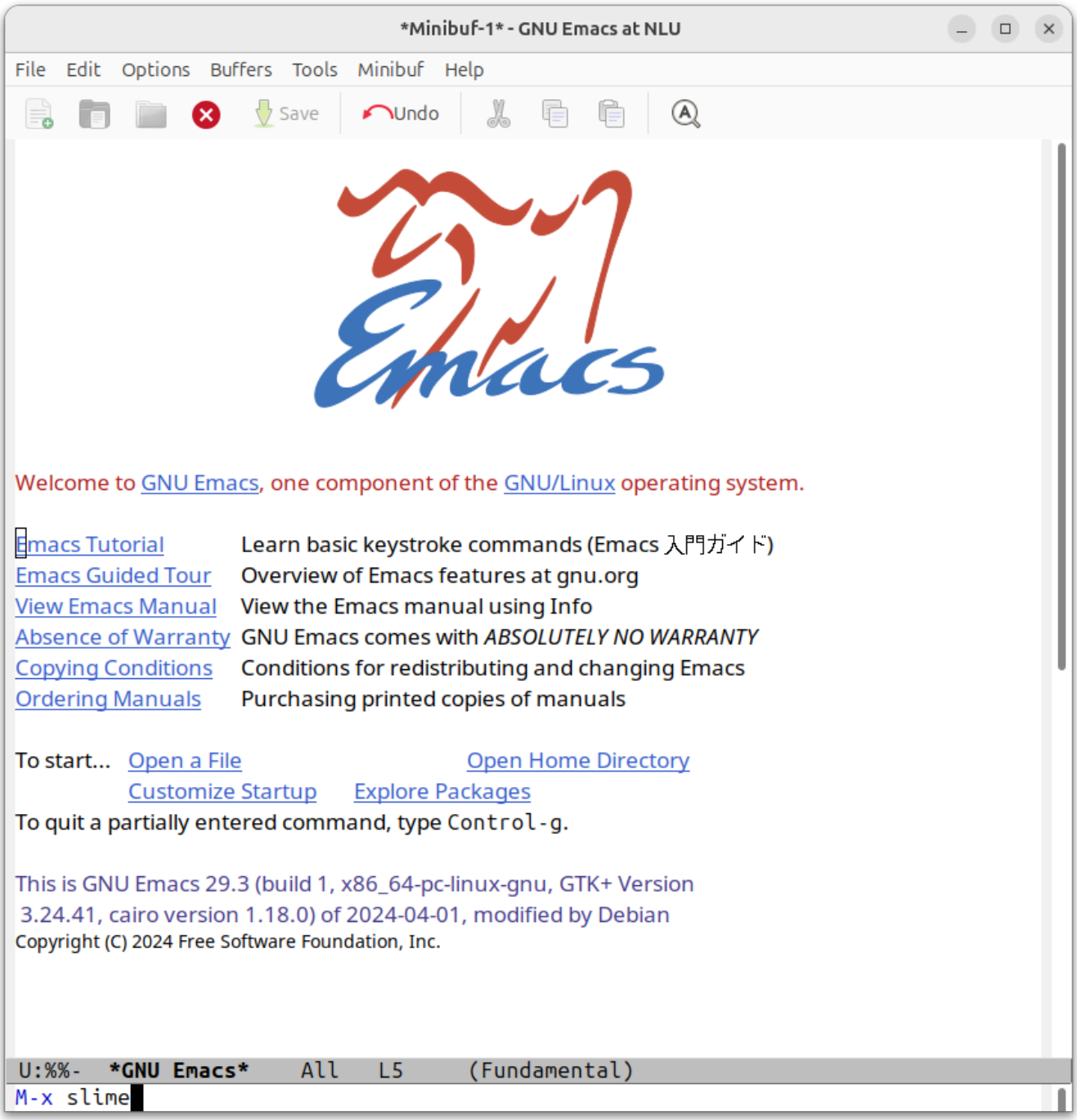
  ;;; The following lines added by ql:add-to-init-file:
  #-quicklisp
  (let ((quicklisp-init (merge-pathnames "quicklisp/setup.lisp"
                                         (user-homedir-pathname))))
    (when (probe-file quicklisp-init)
      (load quicklisp-init)))

Press Enter to continue.

#P"/home/seiji/.sbclrc"
* (exit)
seiji@NLU:~/common-lisp$
```

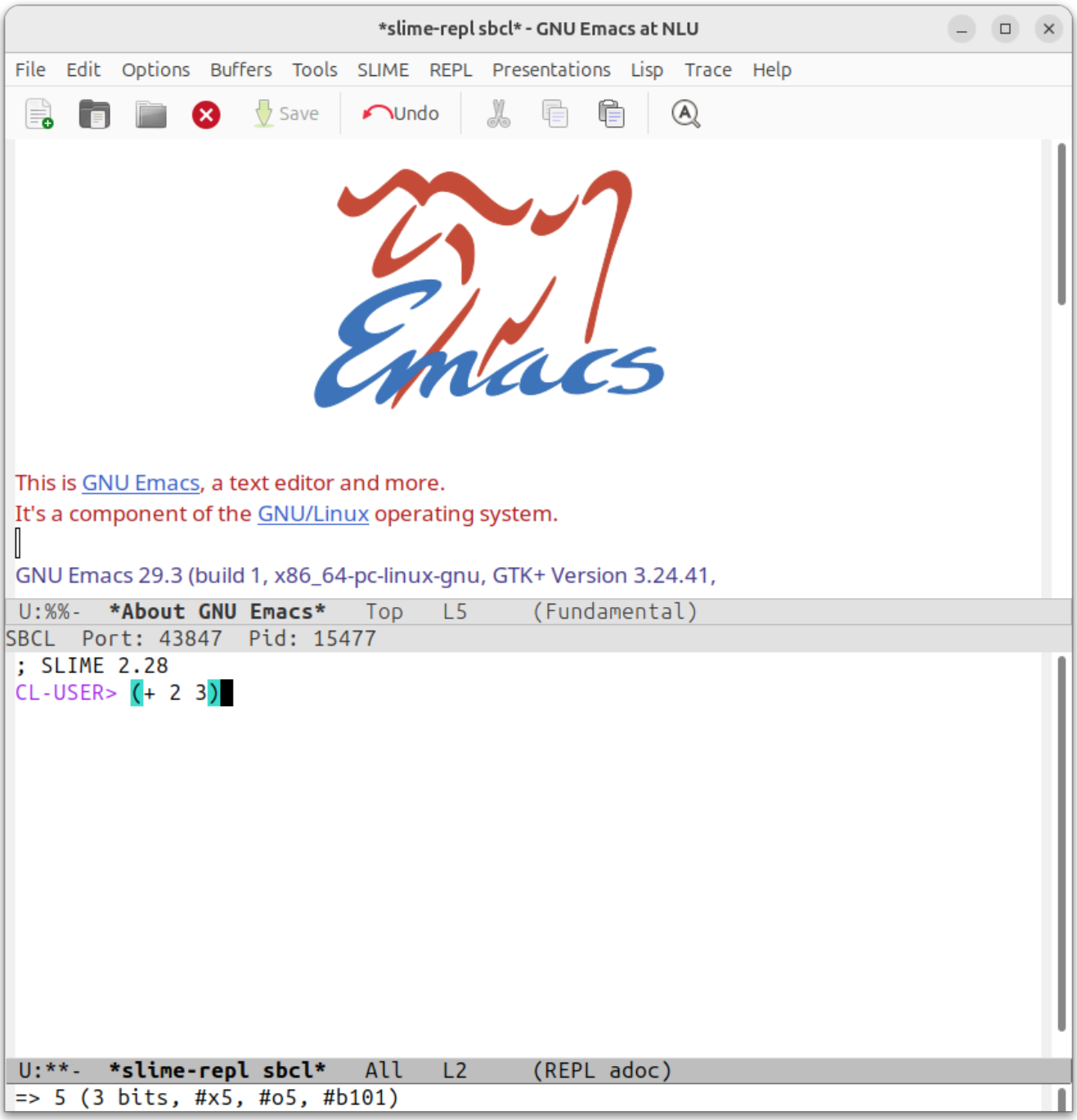
付2.5 開発環境 emacs+slime+sbcl

Allegro Common Lisp ほどの統合開発環境ではないが，それに近いものとして，emacs と slime の組み合わせで SBCL を利用する方法を紹介する．emacs のベテランならこのほうがよいであろう．付2.1では emacs と cl-swank と slime をインストールしている．そのあと emacs とタイプして emacs を起動し，emacs が立ち上がったら，<ESC>-x (M-x) の次に「slime」と打てば，emacs 環境の中で sbcl の REPL を使うことができるようになる．

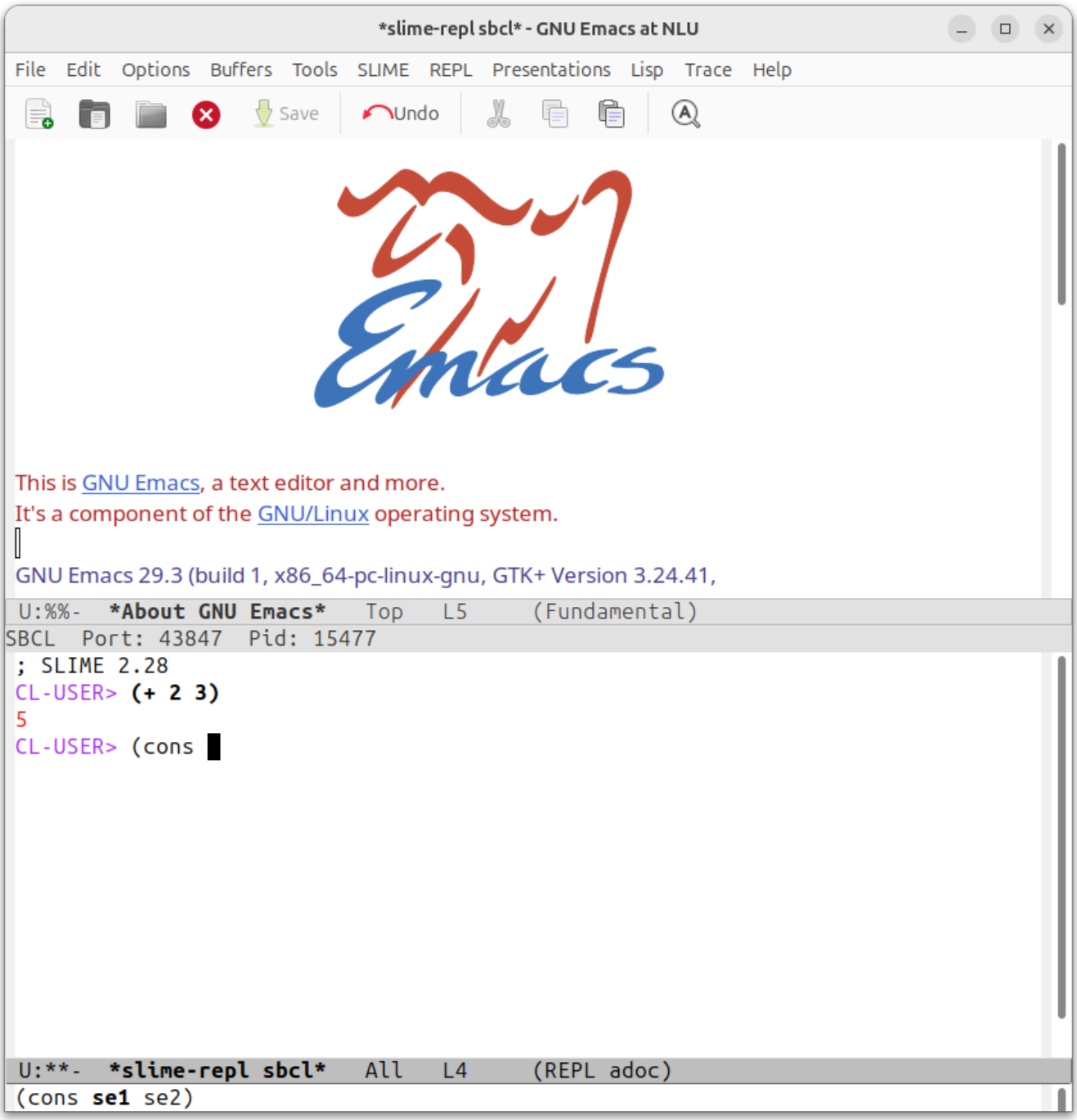


この環境の何が嬉しいかというと，Lisp のソースコードと REPL での実行が緊密に統合されていて，emacs 環境から離れずに書式を実行できるとか， REPL 環境で関数をタイプ入力した時，emacs エコー領域にその引数が表示されることである．このことで一々プログラム仕様書にまで戻ることなく，安心してプログラミングを進めることができる．

下図は **(+ 2 3)** の書式を emacs 上のREPLでタイプしたのち，C-x C-e した結果であるが，これと同様のことが，エディット用バッファ内でも実行できる．



下図では「(cons」の次に空白を一つ打ってエコー領域に cons のコーリングシーケンスが表示され，se1 の入力待ちであることがガイドされている．



また，任意の字句について**シンボルを選んでおいて，C-c C-d h とタイプ入力**すれば，そのシンボルについてのHyperspecドキュメンテーションが閲覧できる．ただし，これはインターネット環境にあるときでローカル環境でも同じことをするにはHyperspecをローカルにインストールしておまじないを掛ける必要がある．具体的なやりかたは「slime hyperspec」などで検索すればいくらでも情報が得られるので，そういうのが得意な人は頑張ってほしい．

Slime 環境では emacs と SBCL がソケット結合されるため，sb-aclrepl は残念ながら有効にならないが quicklisp は大丈夫である．本書では説明の都合上 emacs 環境を用いた記述ではないが，emacs を使いなれた人であれば是非この環境に慣れて SBCL のプログラミングに熟達しよう．

付2.6 ブレーク時の対処方法

Common Lisp でプログラミングしていると，ミススペルなどによって未定義の関数を呼び出そうとしたり，値が束縛されていないシンボルの値を読もうとしたりして，ブレークする瞬間が必ずある．そのような時には原因がすぐ同定できれば，ブレーク状態から脱出してやり直すことになるが，その方法は処理系依存である．そこで，ここでは SBCL におけるブレークからの脱出方法について説明する．

以下は未束縛シンボルの値を取りに行った場合である．

```
seiji@NLU:~$ sbcl
This is SBCL 2.2.9.debian, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.
* foo

debugger invoked on a UNBOUND-VARIABLE @52B4FD3F in thread
#<THREAD "main thread" RUNNING {1001348003}>:
  The variable FOO is unbound.

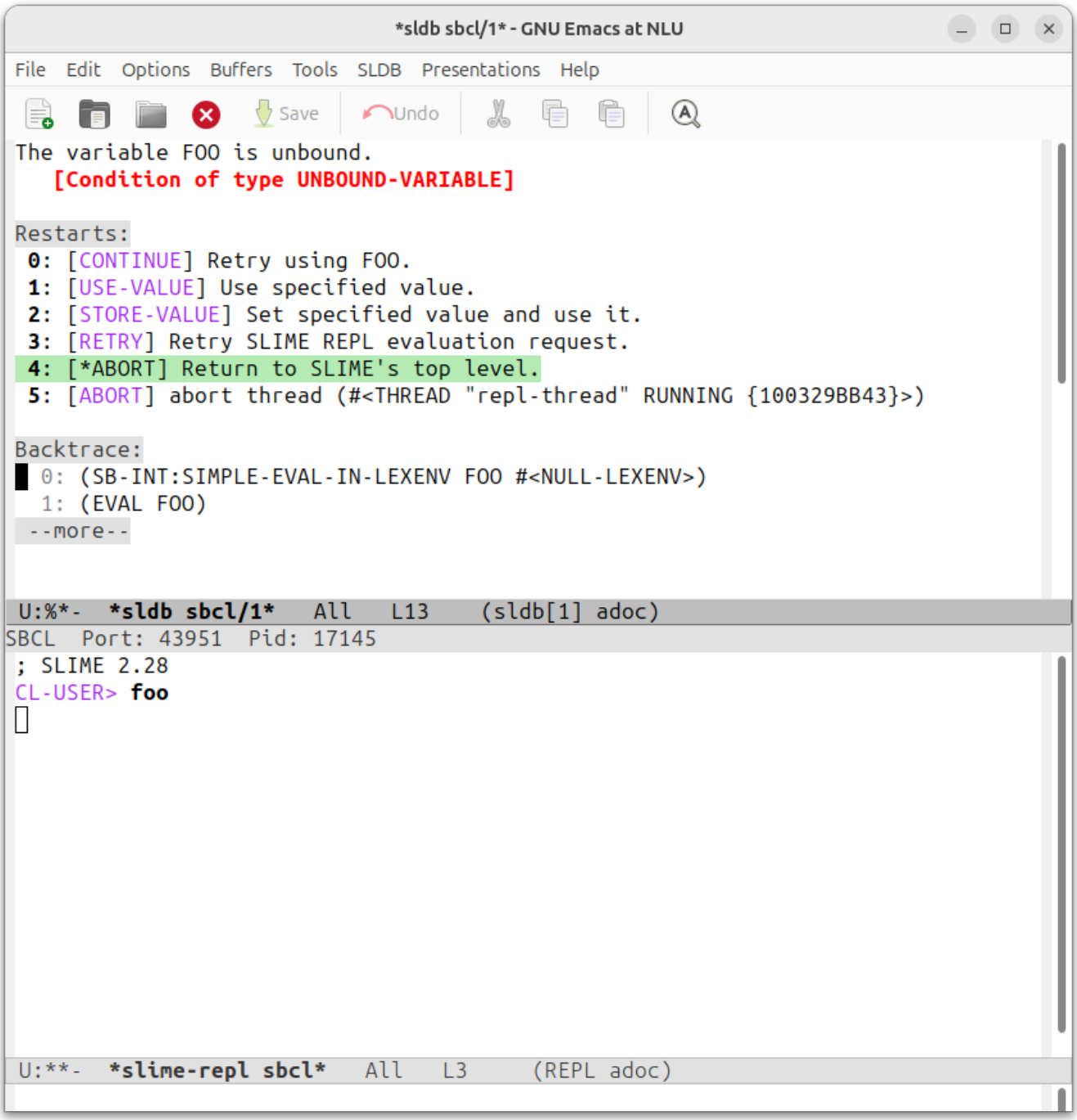
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0: [CONTINUE   ] Retry using FOO.
  1: [USE-VALUE  ] Use specified value.
  2: [STORE-VALUE] Set specified value and use it.
  3: [ABORT      ] Exit debugger, returning to top level.

(SB-INT:SIMPLE-EVAL-IN-LEXENV FOO #<NULL-LEXENV>)
0] 3
*
```

ブレークの原因を示すメッセージとともに，処理の選択肢が番号付きで示される．いずれにしても選択肢の中には必ず，"Exit debugger, returning to top level." というものがあるので，その該当する番号を入力すればよい．ここでは 3 を入力して，もとの REPL に戻った．トップレベルというのは，このブレーク前の REPL 環境のことである．

Slime と emacs を使った場合には，本質的には同じことだが，ちょっとだけ変化する．次の図を見られたい．



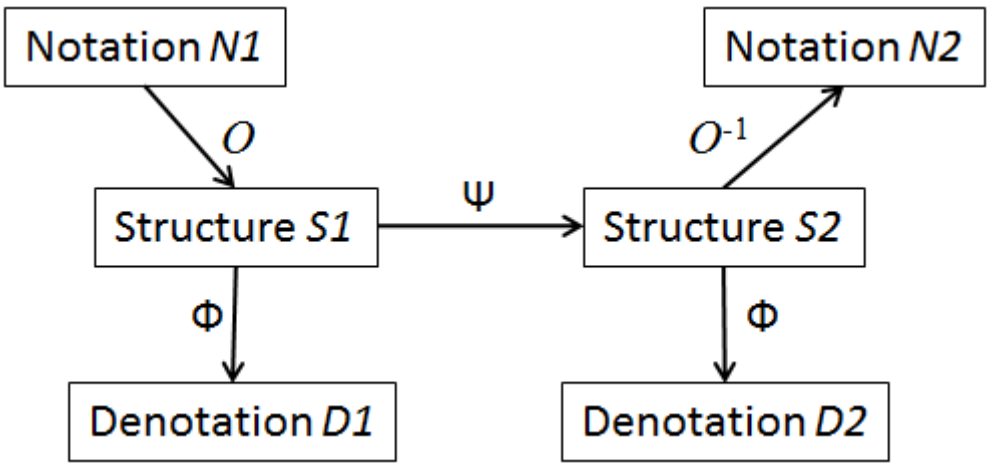
ここでは，"SLIME's top level" に戻るために，カーソルを *ABORT部の **4** のところまで持っていき，そこでマウスボタンを押せばよい.

付録3 マクロ文字とオブジェクト印字関数

付3.1 Homoiconic ということ

他の計算機言語と異なるLispの最大の特徴を一言で言えば，Lispのプログラムコードは Lispの基本的なデータそのものと同じであるということだ．他にこんな計算機言語はない．だからこそ Lisp がかつて持っていた対話型とか関数型とか，ラムダとかEVALといった諸特徴が後続の言語に盗まれていっても，いまだに Lisp のマクロに匹敵するものが他言語では生まれてこない．最近ではこの特長のことをhomoiconicと呼ぶらしい．

homoiconic言語の利点はマクロだけではない．第11章ではCommon LispでSchemeを実現した．では既存のScheme処理系でSchemeを実現できるだろうか．もちろんできる．第11章のSchemeを用いて，改めてSchemeのinterpをコードすることができるだろうか．もちろんできる．このようなものを，メタ巡回的評価子(meta-circular evaluater)と呼ぶ．homoiconic言語ではメタ巡回的評価子を作ることは非常に簡単だ．ではＣ言語でＣ言語文法に従ったREPLを実現することはできるだろうか．多分可能だがLispの場合ほど容易ではない（かつて私はロボット用のＣ文法による対話型の言語処理系を持っていた）．何故 homoiconic でないとダメなのか．Brian C. Smithは手続き型のreflectiveなシステムを当時のMacLispを用いて実装したが，Lisp特有の性質を次の図を用いて以下のように説明した．



付図1 SmithによるLispシステムの説明

一番上のNotationは，人間で言えば表面的な言語や会話の部分であり，計算機であればたとえば計算機に「`(sqrt 2)`」と入力した場合のその文字列表記にあたる．中段のStructureの部分は人間で言えば心，計算機であれば計算機内部の構造部分であり，計算機はその内部構造を操作して別の構造に変換したり，新たに生成したりする．一番下の部分は実世界に相当するところで，言ってみれば「平方根」という抽象的概念であったり，「2」という数の体系の中のある要素であったりする．関係Oは内部化(internalization)，O^{-1}は外部化(externalization)と呼ばれ，Lispであれば「`(sqrt 2)`」は内部化によってヒープ領域におけるconsセルとsqrtというシンボルとfixnumである2からなる構造になる．シンタックスはNotationのレベルの相互関係を規

定するが，セマンティックスはDenotationレベルの実世界を反映すべきものである．関係φは解釈(interpretation)と呼ばれるが，構造*S1*は表示物*D1*を表示(denote)するという．それに対して，表記*N1*は構造*S1*を表記(notate)するという．関係ψは人であれば心の中の動きであるが，論理であればそれは推論であり，Lispであればそれはevalである．

さて，話はここからだ．通常の計算機言語であれば計算ψ(computational process)と解釈φ(interpretation)は厳密に分離している．構造(+ 1 2)は計算ψによって構造3に変換されるが，その構造(+ 1 2)が表示(denote)するものは自然数の3であり，構造3の表示物も同じく3だ．ところが，Lispでは計算ψと解釈φが混然としている．何故ならばLispの計算コードはLispのデータであり，（計算モデルとして）Lispの構造がまたLispの構造を生み出すからだ．シンボル t は真であることをdenoteするが，シンボル t を評価しても t だ．構造 'a を評価すると構造シンボル a が得られるし，構造 (car '(a b)) を評価すれば構造シンボル a が得られる．逆に (cons 'a (list 'b))は構造 (a b) を生み出す．もちろん複雑な構造をどんどん計算して簡単な構造にリダクションしていって，最後は単純な還元された構造が世界の何かをdenoteしなければならないが，シンボル t が真を意味するとか，fixnumの3が数字の3を意味するというのはもはや計算の問題ではなくて人間の解釈の問題だ．

メタ巡回的評価子とはその言語で書かれたその言語の評価機構のことだ．もちろんＣ言語のインタープリタであってもきっとＣ言語で書かれているだろうが，その計算機構が自分自身のことを内察(introspect)したり，自らの計算機構を変更したりすることは不可能だ．しかしLispでは自分の評価機構を自分で変更することができる．ここまでくるとLispがいかに他の言語と異なっているか，そしてhomoiconicこそがLispのパワーの源泉であるということが理解される．Brian C. SmithはMacLisp上でメタ巡回的評価子を書いて，reflectiveなLispシステムを開発した．いつのことか分からないが，学習により自ら解釈を変えて成長することのできる全脳型エージェントはきっとLispで実装されているであろう．

付3.2 オブジェクト印字関数のカスタマイズ

11.2節のScheme末尾再帰のインタープリタでは，構造体procの定義にprint-functionオプションを与えたが，特に説明を与えなかった．このオプションを用いることで，構造体の印刷をユーザが自由に制御できる．前節で説明した外部化(externalization)を自由にプログラムできるわけだ．もし構造体procの定義にprint-functionオプションを定義しないと，通常の構造体の印刷のように印刷されてしまう．

```
(defstruct proc
  "Represent a Scheme procedure"
  code (env nil) (name nil) (parms nil))

scheme(7): (scheme)
==> (define (square x)
      (* x x))

square
==> square
```

```
#S(common-lisp-user::proc :code (* x x) :env nil :name nil :parms (x))
==>
```

ここではそれを嫌って{《手続き名》}と印刷されるようにしたわけだ.

```
(defstruct (proc (:print-function print-proc))
  "Represent a Scheme procedure"
  code (env nil) (name nil) (parms nil))

(defun print-proc (proc &optional (stream *standard-output*) depth)
  (declare (ignore depth))
  (format stream "{~a}" (or (proc-name proc) '??)))
```

ただしこのままだと，実際にはバグがある．実はこのコードでは手続きの名前をセットするところがどこにもない．したがっていつでもここの印刷では「{??}」のように印刷されてしまう．そこで，関数interpの中でグローバル変数に値をセットするところに，次のように値がもし手続きだったらグローバル変数名を手続き名にセットするコードを追加する.

```
(defun interp (x &optional env)
  "Evaluate the expression x in the environment env.
   This version is properly tail-recursive."
  (prog ()
    :INTERP
    (return
      (cond
       ((symbolp x) (get-var x env))
       ((atom x) x)
       ((scheme:macro (first x))
        (setf x (scheme:macro-expand x)) (go :INTERP))
       ((case (first x)
          (quote (second x))
          (scheme:begin (pop x) ; pop off the BEGIN to get at the args
                        ;; Now interpret all but the last expression
                        (loop while (rest x) do (interp (pop x) env))
                        ;; Finally, rename the last expression as x
                        (setf x (first x))
                        (go :INTERP))
          (scheme:set!  (let ((val (interp (third x) env)))
                          (when (proc-p val)
                            (setf (proc-name val) (second x)))
                          (set-var! (second x) val env)))
          (if           (setf x (if (interp (second x) env)
                                    (third x)
                                   (fourth x)))
              ;; That is, rename the right expression as x
              (go :INTERP))
          (scheme:lambda (make-proc :env env :parms (second x)
                           :code (maybe-add 'scheme:begin (rest2 x))))
          (otherwise   ;; a procedure application
           (let ((proc (interp (first x) env))
                 (args (mapcar #'(lambda (v) (interp v env)) (rest x))))
             (if (proc-p proc)
                 ;; Execute procedure with rename+goto
                 (progn
                   (setf x (proc-code proc))
                   (setf env (extend-env (proc-parms proc) args
```

```
                                    (proc-env proc)))
                (go :INTERP))
              ;; else apply primitive procedure
              (apply proc args)))))))))))
```

これで手続きはその内容を隠して次のように印刷されるようになった．

```
scheme(6): (scheme)
==> (define (square x)
        (* x x))

square
==> square

{square}
==>
```

構造体印刷関数の他の例は第２部第25章のフレームシステムXRLにもあるが，Lispオブジェクトの印字を制御できるのは構造体だけではない．CLOSオブジェクトもそのクラス用の print-object メソッドを定義することで制御できる．その実例は第27章事例ベース推論のクラスmopにある．

付3.3 マクロ文字によるread構文のカスタマイズ

通常Lispからの出力はそのまま入力としても解釈できるように出力されるのが原則である．だから私自身はあまりそれに従ってはいないが，システムからのメッセージ出力もわざわざコメント形式で出力したりする．その実例はCLOSメソッド修飾子の説明時に掲載した例を参照されたい．文字列やベクタや構造体などに見られるように，普通はLispオブジェクトの表示はそのままLispのリーダーに読み取られて，そのようなしかるべきLispオブジェクト構造になる．

```
cl-user(2): (make-string '3 :initial-element #\a)
"aaa"
cl-user(3): "aaa"
"aaa"
cl-user(4): (make-sequence 'vector 3 :initial-element #\a)
#(#\a #\a #\a)
cl-user(5): #(#\a #\a #\a)
#(#\a #\a #\a)
cl-user(6): (defstruct triple subject predicate object)
triple
cl-user(7): (make-triple :subject #\a :predicate #\b :object #\c)
#S(triple :subject #\a :predicate #\b :object #\c)
cl-user(8): #S(triple :subject #\a :predicate #\b :object #\c)
#S(triple :subject #\a :predicate #\b :object #\c)
```

一方，CLOSクラスのインスタンスは通常では次のように印刷されるが，ここで「#<」という文字の組み合わせはリスプリーダにとっては，これを読んではならないという信号になる．

```
cl-user(1): (defclass quad () (subject predicate object graph))
#<standard-class quad>
cl-user(2): (make-instance 'quad)
#<quad @ #x100aee132>
cl-user(3): #<quad @ #x100aee132>
```

```
Error: Illegal sharp character #\<
  [condition type: reader-error]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(4):
```

read関数にとっては，文字は基本的にそれが**空白文字**(whitespace)か，**単一エスケープ文字**(single escape)か，**多重エスケープ文字**(multiple escape)か，**終端マクロ文字**(terminating macro)か，**非終端マクロ文字**(non-terminating macro)かのシンタックス上の文字の型が重要である．それら以外の文字は**構成要素文字**(constituent)である．以下に標準文字シンタックス型を示す．

```
<tab> whitespace          <page> whitespace <newline> whitespace
<space> whitespace        @ constituent     ` terminating macro
! constituent *          A constituent     a constituent
" terminating macro       B constituent     b constituent
# non-terminating macro   C constituent     c constituent
$ constituent             D constituent     d constituent
% constituent             E constituent     e constituent
& constituent             F constituent     f constituent
' terminating macro       G constituent     g constituent
( terminating macro       H constituent     h constituent
) terminating macro       I constituent     i constituent
* constituent             J constituent     j constituent
+ constituent             K constituent     k constituent
, terminating macro       L constituent     l constituent
- constituent             M constituent     m constituent
. constituent             N constituent     n constituent
/ constituent             O constituent     o constituent
0 constituent             P constituent     p constituent
1 constituent             Q constituent     q constituent
2 constituent             R constituent     r constituent
3 constituent             S constituent     s constituent
4 constituent             T constituent     t constituent
5 constituent             U constituent     u constituent
6 constituent             V constituent     v constituent
7 constituent             W constituent     w constituent
8 constituent             X constituent     x constituent
9 constituent             Y constituent     y constituent
: constituent             Z constituent     z constituent
; terminating macro       [ constituent *   { constituent *
< constituent             \ single escape   | multiple escape
= constituent             ] constituent *   } constituent *
> constituent             ^ constituent     ~ constituent
? constituent *           _ constituent     <rubout> constituent
<bkspace> constituent  <return> whitespace <linefeed> whitespace
```

リーダへの一連の入力は空白文字によってトークンに分割されるが，終端マクロ文字によってもその前で分割される．しかし非終端マクロ文字はトークンの中にあればトークンの一部となってトークンを終了させないが，トークンの最初にあるときはマクロ文字として特別な処置を受ける．エスケープ文字については小文字のシンボルについて述べたときのとおりである．

```
cl-user(1): (list 'abc"abc")
(abc "abc")
cl-user(2): (list 'abc(+ 1 2))
(abc 3)
cl-user(3): (list (+ 1 2)"abc")
(3 "abc")
```

```
cl-user(4): (list 'abc#efg)
(abc#efg)
cl-user(5): (list 'abc'defg)
(abc defg)
cl-user(6): (list 'abc'defg)
(abc defg)
```

マクロ文字の説明の前に非終端マクロ文字すなわち「#」について説明する．トークンの最初が#であったとき，これはディスパッチマクロ文字となり，続く文字が何かによって働きが異なる．以下に標準ディスパッチマクロ文字を示す．

```
#!  undefined *                #<backspace>  signals error
#"  undefined                  #<tab>        signals error
##  reference to #= label      #<newline>    signals error
#$  undefined                  #<linefeed>   signals error
#%  undefined                  #<page>       signals error
#&  undefined                  #<return>     signals error
#'  function abbreviation      #<space>      signals error
#(  simple vector              #+      read-time conditional
#)  signals error              #-      read-time conditional
#*  bit-vector                 #.      read-time evaluation
#,  load-time evaluation       #/      undefined
#0  used for infix arguments   #A, #a  array
#1  used for infix arguments   #B, #b  binary rational
#2  used for infix arguments   #C, #c  complex number
#3  used for infix arguments   #D, #d  undefined
#4  used for infix arguments   #E, #e  undefined
#5  used for infix arguments   #F, #f  undefined
#6  used for infix arguments   #G, #g  undefined
#7  used for infix arguments   #H, #h  undefined
#8  used for infix arguments   #I, #i  undefined
#9  used for infix arguments   #J, #j  undefined
#:  uninterned symbol          #K, #k  undefined
#;  undefined                  #L, #l  undefined
#<  signals error              #M, #m  undefined
#=  label following object     #N, #n  undefined
#>  undefined                  #O, #o  octal rational
#?  undefined *                #P, #p  pathname
#@  undefined                  #Q, #q  undefined
#[  undefined *                #R, #r  radix-n rational
#\  character object           #S, #s  structure
#]  undefined *                #T, #t  undefined
#^  undefined                  #U, #u  undefined
#_  undefined                  #V, #v  undefined
#`  undefined                  #W, #w  undefined
#{  undefined *                #X, #x  hexadecimal rational
#|  balanced comment           #Y, #y  undefined
#}  undefined *                #Z, #z  undefined
#~  undefined                  #<rubout> undefined
```

`#'foo`は`(function foo)`を作りだし，`#(1 2 3)`はベクタを作りだし`#|this is comment|#`はコメントを作りだし，等々であるが，そのほとんどはこれまでも本書籍の各所で説明されてきた．

マクロ文字はそれに関連する特別な関数を起動させ，後続の文字列を特別に処置する．たとえばダブルクォーテーションは後続の文字をもう一度ダブルクォーテーションが出てくるまで読み取って，読み取ったものをLisp文字列とする．シングルクォートは次のトークンのread関数読み取り結果に対してシンボルquoteを前においてリストとする．各マクロ文字を読んだときに起動される関数はリードテーブルに登録されてお

り，現在有効なリードテーブルは大域変数*readtable*の値となっている．マクロ文字に結びついた関数はget-macro-characterで得ることができる．

```
cl-user(1): *readtable*
#<readtable @ #x100ae5b92>
cl-user(2): (get-macro-character #\()
#<Function read-list>
nil
cl-user(3): (get-macro-character #\')
#<Function read-quote>
nil
cl-user(4): (get-macro-character #\")
#<Function read-string>
nil
cl-user(5): (get-macro-character #\<)
#<Function read-token>
t
cl-user(6): (get-macro-character #\#)
#<Function read-dispatch-char>
t
```

Lispプログラマーは必要ならば自分専用のマクロ文字を定義することができる．すなわち，付図1の内部化を自由にプログラムできる．たとえば，IRIは必ず文字「<」と「>」で囲まれて表記されるとして，これを直にread関数で読み込んでIRIオブジェクトにするには，#\<をマクロ文字としてその処理関数と一緒にリードテーブルに登録すればよい．ただし，プログラミング・マナーとして標準リードテーブルを直接操作するのではなく，一旦自分用のリードテーブルを作ってそれに対して set-macro-character 関数を用いて登録行為を行う．自分用のリードテーブルを作るには最初にcopy-readtableで標準リードテーブルをコピーする．ただし，以下のコード中の ns:intern-iri はIRIをシンボルと同様にインターンする独自の関数である．

```
(defun angle-bracket-reader (stream char)
  "This is used to read an IRI that is enveloped by opening and closing angle bracket.
   This is not intended to be used by users, instead it is used by lisp reader."
  (assert (char= char #\<))
  (let ((nc (peek-char cl:nil stream t cl:nil t)))
    (cond ((char= nc #\<)
           (error "Double angle bracket reader, Not Yet!"))
          (t
           (ns:intern-iri
            (coerce
             (loop with char until (char= #\> (setq char (read-char stream))) collect char)
             'cl:string))))))

(defparameter *rdf-readtable* (copy-readtable))

(eval-when (:execute :load-toplevel :compile-toplevel)
  (setq *readtable* *rdf-readtable*)
  (set-macro-character #\< #'angle-bracket-reader)
  )
```

IRIオブジェクトがCLOSオブジェクトであるとして，その場合にprint-objectでIRI表記の文字列を文字#\<と#\>で括って印刷するようにすれば，Common Lispに新しくIRIという型のオブジェクトがその他の

Common Lispに元からあるLispオブジェクトと全く同様に埋め込まれたことになる．その結果，たとえば次のようなことがREPLにおいて可能になるであろう．

```
cl-user(9): (setq *readtable* *rdf-readtable*)
*rdf-readtable*
cl-user(10): (ns:intern-iri "http://www.w3.org/2000/01/rdf-schema#Resource")
<http://www.w3.org/2000/01/rdf-schema#Resource>
cl-user(11): (eq * <http://www.w3.org/2000/01/rdf-schema#Resource>)
t
cl-user(12):
```

ただし，ある構成要素文字にあらたに終端マクロ文字や非終端マクロ文字としての属性を持たせたければ，set-syntax-from-charを用いてそのような属性を該当する文字に付与しなければならないことを注意しておく．

ディスパッチングマクロ文字についても，マクロ文字と同様にプログラマーが自分用のディスパッチングマクロ文字をset-dispatch-macro-characterを用いてリードテーブルに登録できるが，その方法はマクロ文字のときと同様である．あまりよい応用例を思いつかないが，#UXXXX で UNICODE を与えて，文字コードがXXXXである文字とする（Common Lispでは文字コードと文字は異なるオブジェクトである）のはよい試みかもしれない．ただし、SBCLならば #\U+XXXX で UNICODE 文字を指定できるが、残念ながら Allegro ではそうはならない．

付録４ Armed Bear Common Lisp (ABCL)のインストールとビルド

付4.1 Ubuntuにパッケージとしてインストール

ABCLは Java バーチャルマシン (JVM) で動くCommon Lisp だ．Ubuntuではいつからか定かではないが，apt-getでABCLがインストールできるようになった．jdkも必要とあれば自動的にインストールされるので，ABCLのインストールがSBCL並みに簡単になった．

```
seiji@NLU:~$ abcl
コマンド 'abcl' が見つかりません。次の方法でインストールできます:
sudo apt install abcl
seiji@NLU:~$ sudo apt install abcl
[sudo] seiji のパスワード:
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています... 完了
状態情報を読み取っています... 完了
以下の追加パッケージがインストールされます:
  ca-certificates-java default-jre default-jre-headless fonts-dejavu-extra java-common
libatk-wrapper-java libatk-wrapper-java-jni libjna-java libjna-jni openjdk-21-jre
  openjdk-21-jre-headless
提案パッケージ:
  fonts-ipafont-gothic fonts-ipafont-mincho fonts-wqy-microhei | fonts-wqy-zenhei fonts-indic
以下のパッケージが新たにインストールされます:
  abcl ca-certificates-java default-jre default-jre-headless fonts-dejavu-extra java-common
libatk-wrapper-java libatk-wrapper-java-jni libjna-java libjna-jni openjdk-21-jre
  openjdk-21-jre-headless
アップグレード: 0 個、新規インストール: 12 個、削除: 0 個、保留: 1 個。
60.0 MB のアーカイブを取得する必要があります。
この操作後に追加で 225 MB のディスク容量が消費されます。
続行しますか? [Y/n]
取得:1 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 ca-certificates-java all 20240118
[11.6 kB]
取得:2 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 java-common all 0.75+exp1 [6,798
B]
取得:3 http://jp.archive.ubuntu.com/ubuntu noble-updates/main amd64 openjdk-21-jre-headless
amd64 21.0.7+6~us1-0ubuntu1~24.04 [46.4 MB]
取得:4 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 default-jre-headless amd64
2:1.21-75+exp1 [3,094 B]
取得:5 http://jp.archive.ubuntu.com/ubuntu noble-updates/main amd64 openjdk-21-jre amd64
21.0.7+6~us1-0ubuntu1~24.04 [227 kB]
取得:6 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 default-jre amd64 2:1.21-75+exp1
[922 B]
取得:7 http://jp.archive.ubuntu.com/ubuntu noble/universe amd64 libjna-jni amd64 5.14.0-1
[33.9 kB]
取得:8 http://jp.archive.ubuntu.com/ubuntu noble/universe amd64 libjna-java all 5.14.0-1 [201
kB]
取得:9 http://jp.archive.ubuntu.com/ubuntu noble/universe amd64 abcl all 1.9.0-1 [11.1 MB]
取得:10 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 fonts-dejavu-extra all 2.37-8
[1,947 kB]
取得:11 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 libatk-wrapper-java all 0.40.0-
3build2 [54.3 kB]
取得:12 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 libatk-wrapper-java-jni amd64
```

```
0.40.0-3build2 [46.4 kB]
60.0 MB を 7秒 で取得しました (9,204 kB/s)
以前に未選択のパッケージ ca-certificates-java を選択しています。
(データベースを読み込んでいます ... 現在 155731 個のファイルとディレクトリがインストールされ
ています。)
.../00-ca-certificates-java_20240118_all.deb を展開する準備をしています ...
ca-certificates-java (20240118) を展開しています...


   [...]

done.
default-jre-headless (2:1.21-75+exp1) を設定しています ...
openjdk-21-jre:amd64 (21.0.7+6~us1-0ubuntu1~24.04) を設定しています ...
default-jre (2:1.21-75+exp1) を設定しています ...
abcl (1.9.0-1) を設定しています ...
seiji@NLU:~$ abcl
Failed to introspect virtual threading methods: java.lang.NoSuchMethodException:
java.lang.Thread.builder()
Armed Bear Common Lisp 1.9.0
Java 21.0.7 Ubuntu
OpenJDK 64-Bit Server VM
Low-level initialization completed in 0.197 seconds.
Startup completed in 0.938 seconds.
Type ":help" for a list of available commands.
CL-USER(1): :help

  COMMAND     ABBR DESCRIPTION
  apropos     ap   apropos
  bt               backtrace n stack frames (default 8)
  cd               change default directory
  cf               compile file(s)
  cload       cl   compile and load file(s)
  continue    cont invoke restart n
  describe    de   describe an object
  error       err  print the current error message
  exit        ex   exit lisp
  frame       fr   set the value of cl:* to be frame n (default 0)
  help        he   print this help
  inspect     in   inspect an object
  istep       i    navigate within inspection of an object
  ld               load a file
  ls               list directory
  macroexpand ma   macroexpand an expression
  package     pa   change *PACKAGE*
  pwd         pw   print current directory
  reset       res  return to top level
  rq               require a module
  trace       tr   trace function(s)
  untrace     untr untrace function(s)

Commands must be prefixed by the command character, which is ':' by default.

CL-USER(2): t
T
CL-USER(3): nil
NIL
CL-USER(4): 'foo
FOO
```

```
CL-USER(5): :exit
seiji@NLU:~$
```

Lispコードをコンパイルするとjavaのバイトコードを生み出すので，Javaバイトコードが分かる人にはコンパイラ勉強の良い材料になるだろう．またLispコードでJavaオブジェクトを作るとか，Javaのコードから Lisp関数を動かすなど，JavaとLispを一体化して使うことができる．JVM上で動くLisp系のものにはほかに Ritch Hickey による Clojure Lisp があるが，あちらはCommon Lispではない．一方，ABCLは完全なANSI Common Lisp の実現を目指していて，2010年以前から開発が進められ，2025年現在はバージョン1.9.0である．

付4.2 Apache ant によるソースコードからのビルド

java ソースコードからのビルドでは当然のことながら， Apach ant を利用する．これが比較的簡単なのでここで紹介する．

java に加えて，Apache ant をインストールして，https://abcl.org より，source をダウンロード，解凍する．

```
seiji@NLU:~$ java --version
openjdk 21.0.7 2025-04-15
OpenJDK Runtime Environment (build 21.0.7+6-Ubuntu-0ubuntu124.04)
OpenJDK 64-Bit Server VM (build 21.0.7+6-Ubuntu-0ubuntu124.04, mixed mode, sharing)
seiji@NLU:~$ ant
コマンド 'ant' が見つかりません。次の方法でインストールできます:
sudo snap install ant  # version 1.10.15, or
sudo apt  install ant  # version 1.10.14-1
他のバージョンについては 'snap info ant' を確認してください。
seiji@NLU:~$ sudo snap install ant
[sudo] seiji のパスワード:
error: This revision of snap "ant" was published using classic confinement and thus may
perform
       arbitrary system changes outside of the security sandbox that snaps are usually
confined to,
       which may put your system at risk.

       If you understand and want to proceed repeat the command including --classic.
seiji@NLU:~$ sudo snap install ant --classic
ant 1.10.15 from Apache Ant✓ installed
seiji@NLU:~$ sudo snap install curl
curl 8.13.0 from hideo aoyama (aoilinux) installed
seiji@NLU:~$ curl -O https://abcl.org/releases/1.9.1/abcl-src-1.9.1.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 2537k  100 2537k    0     0   915k      0  0:00:02  0:00:02 --:--:--  915k
seiji@NLU:~$ ls
abcl-src-1.9.1.tar.gz  acl  snap  ダウンロード  テンプレート  デスクトップ  ドキュメント  ビ
デオ  ピクチャ  ミュージック  公開
seiji@NLU:~$ tar zxf abcl-src-1.9.1.tar.gz
seiji@NLU:~$ ls
abcl-src-1.9.1  abcl-src-1.9.1.tar.gz  acl  snap  ダウンロード  テンプレート  デスクトップ
ドキュメント  ビデオ  ピクチャ  ミュージック  公開
seiji@NLU:~$ cd abcl-src-1.9.1
```

```
seiji@NLU:~/abcl-src-1.9.1$ ant
Buildfile: /home/seiji/abcl-src-1.9.1/build.xml

abcl.clean.maybe:
     [echo] Cleaning all intermediate compilation artifacts.
     [echo] Setting 'abcl.build.incremental' enables incremental compilation.

abcl.clean:

abcl.copy.lisp:
     [copy] Copying 4 files to /home/seiji/abcl-src-1.9.1/build/classes

abcl.stage:
     [copy] Copying 11 files to /home/seiji/abcl-src-1.9.1/build/classes
     [copy] Copying /home/seiji/abcl-src-1.9.1/COPYING to /home/seiji/abcl-

   [...]

BUILD SUCCESSFUL
Total time: 2 minutes 9 seconds
seiji@NLU:~/abcl-src-1.9.1$ ls
CHANGES  COPYING  README  abcl  abcl.asd  abcl.bat.in  abcl.in  abcl.properties.in  abcl.rdf
build  build.xml  ci  contrib  dist  doc  etc  examples  nbproject  src  test
seiji@NLU:~/abcl-src-1.9.1$ cd
seiji@NLU:~$ ls
abcl-src-1.9.1  abcl-src-1.9.1.tar.gz  acl  snap  ダウンロード  テンプレート  デスクトップ
ドキュメント  ビデオ  ピクチャ  ミュージック  公開
seiji@NLU:~$ mv abcl-src-1.9.1 abcl
seiji@NLU:~$ cd abcl
seiji@NLU:~/abcl$ abcl
Failed to introspect virtual threading methods: java.lang.NoSuchMethodException:
java.lang.Thread.builder()
Armed Bear Common Lisp 1.9.0
Java 21.0.7 Ubuntu
OpenJDK 64-Bit Server VM
Low-level initialization completed in 0.19 seconds.
Startup completed in 0.919 seconds.
Type ":help" for a list of available commands.
CL-USER(1): :help

  COMMAND     ABBR DESCRIPTION
  apropos     ap   apropos
  bt               backtrace n stack frames (default 8)
  cd               change default directory
  cf               compile file(s)
  cload       cl   compile and load file(s)
  continue    cont invoke restart n
  describe    de   describe an object
  error       err  print the current error message
  exit        ex   exit lisp
  frame       fr   set the value of cl:* to be frame n (default 0)
  help        he   print this help
  inspect     in   inspect an object
  istep       i    navigate within inspection of an object
  ld               load a file
  ls               list directory
  macroexpand ma   macroexpand an expression
  package     pa   change *PACKAGE*
  pwd         pw   print current directory
```

```
  reset       res  return to top level
  rq              require a module
  trace       tr   trace function(s)
  untrace     untr untrace function(s)

Commands must be prefixed by the command character, which is ':' by default.

CL-USER(2): :exit
seiji@NLU:~/abcl$
```

ここでは，最新版より二つマイナーバージョン下の 1.9.1 を選んでビルドした.

付録５ GNU CLISPのインストール

付5.1 Ubuntuにパッケージとしてインストール

GNU CLISPは湯淺太一先生と萩谷昌己先生による Kyoto Common Lisp(KCL) から生まれたにもかかわらず，今ではそのことが海外のリスプコミュニティであまり知られていないのが残念である．仕様もまだあまり確立されていなかった当時，Common Lisp 処理系が日本で C 言語にコンパイルされて生まれたというのが，アメリカのコミュニティに衝撃を与えた．昔はUnixで動くCommon Lispと言えば KCL しかなかったが，今ではLinuxでわざわざ KCL を動かそうとする人はいないだろう．

GNU CLISPをLinuxにインストールするのは簡単である．Ubuntuにおけるインストール実施例を以下に示す．

```
seiji@NLU:~$ clisp
コマンド 'clisp' が見つかりません。次の方法でインストールできます:
sudo apt install clisp
seiji@NLU:~$ sudo apt install clisp -y
[sudo] seiji のパスワード:
パッケージリストを読み込んでいます... 完了
依存関係ツリーを作成しています... 完了
状態情報を読み取っています... 完了
以下の追加パッケージがインストールされます:
  libffcall1b libsigsegv2
提案パッケージ:
  clisp-doc clisp-module-berkeley-db clisp-module-clx clisp-module-dbus clisp-module-fastcgi
clisp-module-gdbm clisp-module-libsvm clisp-module-pari clisp-module-postgresql
  clisp-module-zlib hyperspec
以下のパッケージが新たにインストールされます:
  clisp libffcall1b libsigsegv2
アップグレード: 0 個、新規インストール: 3 個、削除: 0 個、保留: 1 個。
5,618 kB のアーカイブを取得する必要があります。
この操作後に追加で 32.3 MB のディスク容量が消費されます。
取得:1 http://jp.archive.ubuntu.com/ubuntu noble/universe amd64 libffcall1b amd64 2.4-
2.1build1 [11.4 kB]
取得:2 http://jp.archive.ubuntu.com/ubuntu noble/main amd64 libsigsegv2 amd64 2.14-1ubuntu2
[15.0 kB]
取得:3 http://jp.archive.ubuntu.com/ubuntu noble/universe amd64 clisp amd64
1:2.49.20210628.gitde01f0f-3.1build3 [5,592 kB]
5,618 kB を 2秒 で取得しました (2,426 kB/s)
以前に未選択のパッケージ libffcall1b:amd64 を選択しています。
(データベースを読み込んでいます ... 現在 156151 個のファイルとディレクトリがインストールされています。)
.../libffcall1b_2.4-2.1build1_amd64.deb を展開する準備をしています ...
libffcall1b:amd64 (2.4-2.1build1) を展開しています...
以前に未選択のパッケージ libsigsegv2:amd64 を選択しています。
.../libsigsegv2_2.14-1ubuntu2_amd64.deb を展開する準備をしています ...
libsigsegv2:amd64 (2.14-1ubuntu2) を展開しています...
以前に未選択のパッケージ clisp を選択しています。
.../clisp_1%3a2.49.20210628.gitde01f0f-3.1build3_amd64.deb を展開する準備をしています ...
clisp (1:2.49.20210628.gitde01f0f-3.1build3) を展開しています...
libffcall1b:amd64 (2.4-2.1build1) を設定しています ...
libsigsegv2:amd64 (2.14-1ubuntu2) を設定しています ...
```

```
clisp (1:2.49.20210628.gitde01f0f-3.1build3) を設定しています ...
libc-bin (2.39-0ubuntu8.4) のトリガを処理しています ...
man-db (2.12.0-4build2) のトリガを処理しています ...
seiji@NLU:~$ clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `--__|__-'        8         8           8           8  8
        |             8     o   8           8     o     8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.49.93+ (2018-02-18) <http://clisp.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992-1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2018

Type :h and hit Enter for context help.

[1]> :h
You are in the top-level Read-Eval-Print loop.
Help (abbreviated :h) = this list
Use the usual editing capabilities.
(quit) or (exit) leaves CLISP.
[2]> t
T
[3]> nil
NIL
[4]> 'foo
FOO
[5]> (exit)
Bye.
seiji@NLU:~$
```

SBCLも含めてOSSのCommon LIsp処理系には日本語マニュアルがないのが普通である．そんな中で有志の努力によりCLISPについては比較的充実したものがある．こちらを見られたい．

付録６ Paul Grahamの継続関数

付6.1 継続渡しマクロ

Common Lisp に Scheme のような継続（call/cc）はないが，ポール・グレアムは Lisp関数とマクロを一体化させて，見かけ上Schemeに似た継続の仕組みを考えた．この書籍でも31章のプラニングにおいてそれを利用した非決定的プログラミングが使われるので，ここでポール・グレアムによるCommon Lispにおける**継続関数**について説明する．

説明に入る前の準備と下調べとして以下のコードを実行してみる．

```
(setq -cont- #'values)
(defun =test-tail (-cont- n) (print n) (funcall #'=test-tail -cont- (1+ n)))
(compile '=test-tail)
(=test-tail -cont- 0)
```

このコードは10.1節の末尾再帰のコードを若干変形したものだ．このコードでは Lisp関数 `=test-tail` の引数に継続 `-cont-` を持ちまわっている．また再帰呼び出しにはfuncallを介していることにも注意されたい．やってみればすぐわかるようにこのコードでも末尾再帰のコンパイル効果が得られているようだ．この継続 `-cont-` の持ち回りをユーザの目から隠すために，ポール・グレアムはマクロを利用して継続関数として `test-tail` を定義したら，その第一引数に常に継続を受け取るような `=test-tail` Lisp関数を定義するような仕組みを考えた．

以下において`-cont-`はオリジナルのコードでは`*cont*`となっていたものであるが，`*cont*`というシンボル名では特殊変数との誤解を招きやすいので，このテキストではこれまでの継続記述と同様にハイフン記号で耳当てをした．ここで `defvar` の宣言は決してしてはならない．なぜなら `-cont-` は通常の動的変数ではないからである．それをコード上でも明示するためにわざと `defvar` 宣言を書いて，決してそれが読まれないように `#+:never` を付けた．ポール・グレアムによれば「`-cont-`は何のためにあるのか？それは現在の継続に束縛される」だそうである．これを**継続パラメータ**と呼ぶことにしよう．

REPLに下記のコードを打ち込むときには順番にのみ気を付ければよいが，ファイルにする場合には `defpackage` 宣言部分だけを `package.lisp` というようなファイル名に別建てにして，順次読み込むようにする．これはSBCLに対応するためである．

```
(cl:defpackage :callcc
  (:use :cl)
  (:export =lambda =defun =bind =values =funcall =apply
           choose choose-bind fail))

(in-package :callcc)

#+:never (defvar -cont-)
```

```
(setq -cont- #'values)

(defmacro =lambda (parms &body body)
  "macro for continuous lambda form. It replaces the form with the stack of
closures."
  `#'(lambda (-cont- ,@parms) ,@body))

(defmacro =defun (name parms &body body)
  "defines macro and function named '=<name>' with the stack of closures."
  (let ((f (intern (concatenate 'string "=" (symbol-name name)))))
    `(progn
       (defmacro ,name ,parms
         `(,',f -cont- ,,@parms))
       (defun ,f (-cont- ,@parms) ,@body)
       #+:allegro (compile ',f))))

(defmacro =bind (parms expr &body body)
  "this is used for making parameter bindings in continuous functions instead
of let form."
  `(let ((-cont- #'(lambda ,parms ,@body))) ,expr))

(defmacro =values (&rest retvals)
  "this is used for making multiple value returning in continuous functions
instead of values form."
  `(funcall -cont- ,@retvals))

(defmacro =funcall (fn &rest args)
  "this is used for making a funcall form in continuous functions."
  `(funcall ,fn -cont- ,@args))

(defmacro =apply (fn &rest args)
  "this is used for making an apply form in continuous functions."
  `(apply ,fn -cont- ,@args))
```

上記 =defun はマクロであるが，引数に与えた名前の頭に「=」を追加した Lisp関数を定義する．その Lisp関数引数では =defun で与えられる関数引数に加えて継続-cont-を受け取る．こうして以下に示すように，マクロの表面上は継続は現れていないが，実際のLisp関数ではいつも継続を持ちまわることになる．

```
cl-user(1): (in-package :callcc)
#<The callcc package>
callcc(2): (=defun test-tail (n)
             (print n)
             (test-tail (1+ n)))
=test-tail
nil
nil
callcc(3): (macroexpand
'(=defun test-tail (n)
           (print n)
           (test-tail (1+ n))))
(progn (defmacro test-tail (n)
           (excl::bq-list '=test-tail `-cont- n))
  (defun =test-tail (-cont- n)
    (print n)
    (test-tail (1+ n)))
  (compile '=test-tail))
t
```

```
callcc(4): (test-tail 0)
0
1
2
3
4
5
...
```

このコードでは，継続関数 `test-tail` 定義時に継続パラメータ`-cont-`が補足されるが（だから何よりも最初に`(setq -cont- #'values)`が宣言されているのは非常に意味がある）、`-cont-`はどこにも使われないし更新もされない.

付6.2 継続をベースにしたマルチプロセスの実現

マルチタスクとは、複数の作業やプロセスを同時に進行させる能力や方法のこと（Copilot）らしい．計算機側から見れば，複数のタスク処理プロセスが同時に存在していても（concurrent process)，実際にはそれらを一つのCPUが順次逐次的に実行しているか（逐次処理プロセス），それとも複数のCPUが同時に実行しているか（平行処理プロセス）によって，大きな違いがある．過去に MIMD(Multiple Instruction Multiple Data) 用 Lisp 計算機と Lisp 言語の開発が行われたこともあったが，Common Lisp 規格としては SISD(Single Instruction Single Data)が前提となっている.

現在のPCはマルチスレッドがほとんどであり，マルチスレッドを生かした Common Lisp の並列ないし並行プログラミングの試みがあるのは至極当然のことといえよう.

ここで紹介するものは上記のようなものではなく，あくまでも SISD を前提とした逐次並列処理のマルチプロセスであり，ポール・グレアムのいうプロセスとはその実態としては実行可能なLisp関数であるが，属性として実行優先順位とwait式を持つようなものである.

以下のコードにおいてプロセスはすべて大域変数 `*procs*` に保存されるが，大域変数 `*proc*` には次に実行されるべき，あるいは実行中のカレントプロセスが保存される.

```
(cl:defpackage :multiproc
  (:nicknames :proc)
  (:use :common-lisp :callcc)
  (:export #:wait #:halt #:yield #:program #:fork #:program0)
  )

(in-package :proc)

(defstruct proc
  "a process has the priority (pri), continuation (state), and wait-function
(wait)."
  pri state wait)

(defvar *procs* nil
  "a list of interunted processes to be reinvoked.")
```

```
(defvar *proc* nil
  "current process, that is set by pick-process.")

(defvar *halt* (gensym)
  "this provides process halt mechanism together with lisp catch-and-throw")

(defvar *default-proc*
    (make-proc :state #'(lambda (x)
                          (declare (ignore x))
                          (format t "~%>> ")
                          (princ (eval (read)))
                          (pick-process)))
  "this is invoked when there is no other processes that should be invoked,
see most-urgent-process.")

(defmacro fork (expr pri)
  "make a new process of expr with priority pri, and push it into *procs*.
Note that expr should be a continuation function calling form that is defined
by =defun, see calcc."
  (let ((prm (gensym)))
    `(prog1 ',expr
       (push (make-proc
              :state #'(lambda (,prm)
                         (declare (ignore ,prm))
                         ,expr
                         (pick-process))
              :pri ,pri)
             *procs*)))
  )
```

実際にプロセスを生成するのはマクロ `fork` である．マクロ `fork` はこのマルチプロセッシング用REPL内部で評価されるべき書式 `expr` とその優先順位 `pri` が与えられて呼び出されるが，生成されたプロセスは大域変数 `*procs*` にプッシュされる．ここで書式 `expr` は実行可能な関数としてラップされるが，その最後に関数 `pick-process` の呼び出しが付け加えられていることに注意されたい．すなわちこのプロセス実行中に `pick-process` 内部で次のプロセスが選択され，実行される．`pick-process` 内部ではプロセス優先順位とwait式を考慮して，実行すべき最も緊急なプロセスが一つ選ばれ，それに制御が移される．

```
(defmacro program (name args &body body)
  `(=defun ,name ,args
           (setq *procs* nil)
           ,@body
           (catch *halt* (loop (pick-process)))))

(defun pick-process ()
  (multiple-value-bind (p val) (most-urgent-process)
    (setq *proc* p
          *procs* (delete p *procs*)
      )
    (funcall (proc-state p) val)))

(defun most-urgent-process ()
  (let ((proc1 *default-proc*)
        (max -1)
        (val1 t))
```

```
    (dolist (p *procs*)
      (let ((pri (proc-pri p)))
        (if (> pri max)
            (let ((val (or (not (proc-wait p))
                           (funcall (proc-wait p)))))
              (when val
                (setq proc1 p
                      max pri
                      val1 val))))))
    (values proc1 val1)))

(defun arbirator (test cont)
  (setf (proc-state *proc*) cont
        (proc-wait *proc*) test)
  (push *proc* *procs*)
  (pick-process))

(defmacro wait (parm test &body body)
  `(arbirator #'(lambda () ,test)
              #'(lambda (,parm) ,@body)))

(defmacro yield (&body body)
  `(arbirator nil #'(lambda (,(gensym)) ,@body)))

(defun setpri (n) (setf (proc-pri *proc*) n))

(defun halt (&optional val) (throw *halt* val))

(defun kill (&optional obj &rest args)
  (if obj
      (setq *procs* (apply #'delete obj *procs* args))
    (pick-process)))
```

最も緊急なプロセスを選ぶのは `most-urgent-process` であるが，いつもwait式を実行して，答えが得られたときのみを選択対象とする．だからwait式の中身に副作用が在ってはならない． `most-urgent-process` の返す値はプロセスおよび wait 式実行結果であるが，この wait 式の実行結果は `pick-process` 内部においてこのプロセスへの関数引数として与えられる．

ちょっとやってみよう．ここでは継続関数 `foo` を最初に定義し，その呼び出しフォーム `(foo 2)` をプロセスとしてフォークしておいて，4行目のいささか不自然な形式でそのプロセスを実行した．確かに `(pick-process)` でプロセスが実行され，最後には `*halt*` がキャッチされ，マルチプロセッシング用REPLにおいて `(halt)` でプロセス実行が終了したことがわかる．

```
cg-user(1): (in-package :proc)
#<The multiproc package>
proc(2): (=defun foo (x)
          (format t "Foo was called with ~A.~%" x)
          (=values (1+ x)))
=foo
nil
nil
proc(3): (fork (foo 2) 25)
(foo 2)
```

```
proc(4): (catch *halt* (loop (pick-process)))
Foo was called with 2.

>> (halt)
nil
proc(5):
```

ここで expr である (foo 2) の実行結果はどこにも受け取られないことを注意しておく．さらにこのままではwait式の値はNILのままであり，その場合は優先順位だけが有効であるが，いま他に競合プロセスがないためにこれが選ばれて実行されることを注意しておく．

それではwait式の例を以下に示す．3行目の

```
cg-user(1): (in-package :proc)
#<The multiproc package>
proc(2): (defvar *open-doors* nil)
*open-doors*
proc(3): (=defun pedestrian ()
  (wait d (car *open-doors*)
    (format t "Entering ~A~%" d)))
; While compiling #'=pedestrian at top level:
Warning: Variable callcc::-cont- is never used.
=pedestrian
t
nil
proc(4): (program ped ()
         (fork (pedestrian) 1))
=ped
nil
nil
proc(5): (ped)
 0[5]: (=ped #<Function values>)
   1[5]: (pick-process)
     2[5]: (most-urgent-process)
     2[5]: returned #S(proc :pri 1 :state #<Closure (:internal =ped 0) @
#x20501c862> :wait nil) t
     2[5]: (=pedestrian #<Function values>)
       3[5]: (pick-process)
         4[5]: (most-urgent-process)
         4[5]: returned
                 #S(proc :pri nil
                         :state #<Function (:internal (:top-level-form
"C:\\allegro-projects\\multiprocess\\multiproc.lisp" 2084) 0) @ #x201b89fb2>
                         :wait nil)
                 t

>> (push 'door2 *open-doors*)
(door2)
         4[5]: (pick-process)
           5[5]: (most-urgent-process)
           5[5]: returned #S(proc :pri 1 :state #<Function (:internal
=pedestrian 1)> :wait #<Function (:internal =pedestrian 0)>) door2
Entering door2
         4[5]: returned nil
       3[5]: returned nil
     2[5]: returned nil
     2[5]: (pick-process)
```

```
        3[5]: (most-urgent-process)
        3[5]: returned
                #S(proc :pri nil
                        :state #<Function (:internal (:top-level-form
"C:\\allegro-projects\\multiprocess\\multiproc.lisp" 2084) 0) @ #x201b89fb2>
                        :wait nil)
                t

>> (halt)
     2[5]: returned-by-throwing to tag g1256: nil
   1[5]: returned-by-throwing to tag g1256: nil
 0[5]: returned nil
nil
proc(6):
```

付6.3 継続をベースにした非決定的プログラミング

非決定的とは迷路の分かれ道にきたときに，決まった方法（アルゴリズム）で取るべき道を選択するのではなく，選択可能な複数の中から適当にそのうちの一つを選択するようなことを言う．ここでなぜ非決定的プログラミングの話をするかというと，Ghallab, Nau, Traverso による各種のプラニングの形式化に非決定的なアルゴリズムが採用されているからである．通常の言語では非決定的プログラムを記述するのは大変であるが，LispにおいてはGrahamによって既にCommon Lispにおける非決定的プログラミングの書き方が明らかにされている．ここではそのライブラリモジュールについて説明し，第31章にてそのモジュールを利用した古典的プラニングのいくつかについて説明する．

GitHubの cl-aip/callcc からプログラムをダウンロード・インストールし，以下のように実行してみよう．

```
cl-user(1): (in-package :callcc)
#<The callcc package>
callcc(2): (choose-bind x '(marrakesh strasbourg vegas)
             (format nil "Let's go to ~A." x))
"Let's go to marrakesh."
callcc(3): (fail)

Failed.
"Let's go to strasbourg."
callcc(4): (fail)

Failed.
"Let's go to vegas."
callcc(5): (fail)

Failed.
@
callcc(6):
```

ここでやっていることは，気持ちとしてはマラケシュとストラスブルグとラスベガスのいずれかを選択して，そこに行こうということだが，マラケシュに行こうとしてそれに失敗すると次はストラスブルグに行こうとして，それに失敗すると今度はラスベガスに行こうとし，さらに失敗するともう行くところはないと

no backtrack point であるアスタリスクを返している，ということだ．こんな仕組みを可能にしているのが，Common Lispの継続とマクロだ．

経路の非決定的選択を行うには，探索途中における探索行為の継続を保存するための大域変数 `*path*` を使う．マクロ `choose` は選択肢の集合 `choices` が nil ならば `(fail)` にマクロ展開するが，通常は選択肢の最初は実行するが残りの選択肢は各々の選択肢について将来実行されたときにその選択肢を返すような関数を作りだして，それを `*path*` に保存するようなコード展開をする．`choose` では単に値を返すだけの継続だが，`choose-bind` では与えられた変数に選択肢の最初を束縛してその環境でボディを実行するような継続だが，残りの選択肢については将来実行されたときにそれらを変数に束縛して実行するようなコードを作りだして，それを `*path*` に保存する．`choose-bind` のボディには，その束縛条件にて実行されるべき式を置く．関数 `fail` は呼び出されると保存されている残りの選択肢から先頭を 1 個取りだして，それを実行する．

```
(defparameter *paths* nil "stack of continuation for undeterministic choice.")
(defconstant failsym '@ "marker of no backtrack point.")

(defmacro choose (&rest choices)
  "if <choices> is not empty, undeterministically choose one of choices and
   evaluates it."
  (if choices
      `(progn
        ,@(mapcar #'(lambda (c)
                      `(push #'(lambda () ,c) *paths*))
            (reverse (cdr choices)))
        ,(car choices))
    '(fail)))

(defmacro choose-bind (var choices &body body)
  "undeterministically chooses one of <choices> and binds it to <var>, then
   evaluates <body> in the binding."
  `(cb #'(lambda (,var) ,@body) ,choices))

(defun cb (fn choices)
  "this is used in <choose-bind>."
  (if choices
      (progn
        (if (cdr choices)
            (push #'(lambda () (cb fn (cdr choices)))
                  *paths*))
        (funcall fn (car choices)))
    (fail)))

(defun fail ()
  "backtracks to the last point of continuation and calls the continuation."
  (format t "~%Failed.")
  (if *paths*
      (funcall (pop *paths*))
    failsym))
```

ただし非決定的な選択と言っても，ここで実際に行っていることは与えられた選択肢をリストから順番に選んでいて，`(fail)` で次の候補を選んで実行しているだけであり，外部条件が同じなら得られる結果はいつ

でも同じなので，結果がその時々で異なる非決定的なものではないが，探索におけるバックトラックがこの低レベルのライブラリに閉じ込められるため，探索のコードの見通しは格段によくなる．

非決定的探索の例を以下に示す．=defun，=values，=bindなどは継続を利用した非決定的探索のためのものだが，以下の例では新しい継続が作り出されるわけではない．今，0から5までの数字から任意に二つの数字を選び，合計がある数字（ここではsum）になるようなプログラムを考える．

```
callcc(7): (=defun two-numbers ()
             (choose-bind n1 '(0 1 2 3 4 5)
                (choose-bind n2 '(0 1 2 3 4 5)
                   (=values n1 n2))))
=two-numbers
callcc(8): (=defun parlor-trick (sum)
             (=bind (n1 n2) (two-numbers)
               (if (= (+ n1 n2) sum)
                   `(The sum of ,n1 ,n2)
                 (fail))))
=parlor-trick
callcc(9): (parlor-trick 7)

Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
Failed.
(The sum of 2 5)
callcc(10):
```

みて分かるように，このコードは非決定的な選択で二つの数字を選んだあとは，合計が7ならばよしとし，もしそうでなければ (fail) を呼び出しているだけである．実際には (fail) が実行されるたびにバックトラックが起こり，いつか望みの数字の組み合わせを発見して終了する．探索の詳細について気にすることなく，プログラマーは探索の結果得られるものについて，望ましい条件をプログラムするだけでよい．さもなければ失敗と言うだけのことである．

このような仕組みを可能にするために，特殊な継続を実装する．パラメータ -cont- は実行途中における次の継続を保管するためのものである．初期状態あるいは呼び出しスタックのボトムではここには関数 values がセットされている．マクロ =values はこの継続プログラムでは必ず =defun と組になって用いられるものであり，探索の最後にはLisp関数 values で値が返される．

探索の途中では関数 cb において cdr choices がある限り将来実行されるべき継続を作り出して *path* に置いておく．マラケシュの場合にはユーザが成功失敗の判定をしたが，最後の例では継続関数 parlor-trick が成功失敗の判定者である．

付録 7 Test Harnesの使用法

テスト駆動開発について

テスト駆動開発（Test-Driven Development: TDD）とは，プログラムを書き始める前にテストコードを書き（テストファースト），そのテストコードに適合するように実装とリファクタリングを進めていく方法のことらしい．

序章で最初に述べたように，最初から人工知能プログラム開発用として作られた Lisp は試行錯誤を伴う創造的プログラミングが主な開発手法であり．現在巷で喧伝される開発手法で一番近いものと言えばアジャイル開発になるであろう．その点では人工知能プログラミングにテスト駆動開発手法はあまり適当ではないことは確かであるが，それでも望みのプログラムが大体出来上がったあと，品質向上に向けてリファクタリングはつきものと言ってよく，この段階になるとリファクタリングの結果を保証するためにはテスト駆動開発の手法が有効である．

Common Lisp でテスト駆動開発をサポートするものがないわけではない．以下では最初に Allegro Common Lisp 用に Franz 社が提供する Test Harness を紹介したのち，その派生で Allegro 以外にも有効な ptester の使い方について述べる．

付7.1 Allegro Common Lisp の Test Harness

Allegro Common Lisp で Test Harnes を使うには，最初に require して tester システムをロードする．

```
cg-user(1): (require :tester)
; Fast loading from bundle tester\tester.fasl.
t
cg-user(2): (use-package :util.test)
t
cg-user(3):
```

ここでは対話型でテストを実行するために、defpackageではなく、関数 `use-package` を使った．これで以下のようにパッケージ指定なしで関数 test が実行できるようになる．

```
cg-user(3): (test 'fixnum (type-of 1))
t
cg-user(4): (test t (typep 1 'number))
t
cg-user(5): (test '(t t) (subtypep 'fixnum 'number) :multiple-values t)
t
cg-user(6):
```

このように関数 test の2番目の引数にテストを実行したいフォームを置き，その期待される実行結果（関数の返値）を 1 番目の引数に置けばよい．期待通りの値が返されればテストの結果は t である。返値が多値の場合でも通常はそのプライマリー値が用いられるが，多値のすべてについてチェックしたいときは期待される多値をリストで与え，test にキーワード引数として `:multiple-values t` を与える．

複数の test をまとめて実行するには with-tests マクロを用いる．

```
cg-user(7): (with-tests (:name :example)
              (test 'fixnum (type-of 1))
              (test t (typep 1 'number))
              (test '(t t) (subtypep 'fixnum 'number) :multiple-values t))
Begin example test
********************************
End example test
Errors detected in this test: 0
Successes this test: 3
nil
cg-user(8):
```

参考文献（上）

Abelson, Harold, Gerald J. Sussman: Structure and Interpretation of Computer Programs, MIT (1985), 元吉文男(訳), プログラムの構造と実行[上][下], マグロウヒル (1989), 第２版 (1996), 和田英一(訳), ピアソン　(2000)

Baker, Henry G.: A Decision Procedure for Common Lisp's SUBTYPEP Predicate, Lisp and Symbolic Computation, Vol.5, pp.157-190, Kluwer (1992)

Charniak, Riesbeck, McDermott: Artificial Intelligence Programming, LEA (1980), 白井英俊, 片桐恭弘(訳) 人工知能プログラミング, 日本コンピュータ協会 (1986)

Cutting, Doug, Jan Pedersen, and Per-Kristian Halvorsen: An Object-Oriented Architecture for Text Retrieval, In Conference Proceedings of RIAO'91, Intelligent Text and Image Handling, pp.285-298 (1991)

Graham, Paul:　On Lisp　(1993), 野田（訳）, On Lisp, オーム社 (2007)

Graham, Paul: Being Popular (2001), 川合史郎(訳), 人気の言語を作るには, http://practical-scheme.net/trans/being-popular-j.html(2004)

Hofstadter, Douglas R.: Metamagical Themas, Basic Books(1985), 竹内郁雄, 斉藤康己, 片桐恭弘(訳), メタマジック・ゲーム, 白揚社(1990), 第19章

Kiczales, Gregor, Jim des Rivieres, and Daniel G. Bobrow: *The Art of the Metaobject Protocol*, ISBN 0-262-61074-4, MIT(1991)

Koide, Seiji and Hideaki Takeda: Meta-circularity and MOP in common Lisp for OWL full, Proc. 6th European Lisp Workshop (ELW09), pp.28-34, ACM New York, NY, USA (2009), ISBN: 978-1-60558-539-0, doi>10.1145/1562868.1562872

Koide, Seiji and Hideaki Takeda: Revisiting CLOS MOP, Int. Lisp Conf. (ILC 2012), The Association of Lisp Users (2012)

McCarthy, John: *LISP 1.5 Programmer's Manual*, ISBN:0262130114, MIT (1962)

Norvig, Peter: Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp, Morgan Kaufmann (1992)

Seibel, Peter: Coders at Work, Apress(2009), 青木靖(訳), Coders at Work, オーム社(2011)

Smith, Brian C.: Reflection and Semantics in Lisp, POPL '84 of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp.23-35 (1984)

Steele Jr., Guy L., 井田(訳):Scheme過去・現在・未来, bit 28-5, pp.84-92 (1996)

小出誠二, 武田英明: 人工知能用言語Lispの今と将来, 24(5), 681-690, 人工知能学会誌(2009-09)

第21章 人工知能プログラミング環境

21.1 PAIPとAIMAのプログラミング環境

第20章まではCommon Lispプログラミングについての記述であった．序章では Lisp は最初から人工知能用言語として開発されたことを述べた．初期の Lisp に関する代表的文献は Lisp 1.5 プログラマーズマニュアル (1962)であるが，最初の Common Lisp に関する言語仕様書 *Common Lisp the Language* (*CLtL1*) が出版されたのが1984年，CLOSを含む第2版 *CLtL2* が出版されたのが1990年である．今日では Common Lisp 標準準拠といえば，ANSI Common Lisp (1994年) に準拠のことになるのだが，これのよいテキストがない．Paul Graham のANSI Common Lisp は初心者向けとは言えるが，記述に彼独自のくせがあってレファレンスとしては使えない．Kent Pitman による HyperSpec が唯一手軽に利用できるレファレンスではあるが，仕様を確認するのには使えてもANSI Common Lisp の何たるかを学ぶには適していない．本書の第1部はその穴を埋めることを目的の一つとしている．

本章からはいよいよ人工知能プログラミングについての記述である．Lisp による人工知能プログラミングのテキストと言えば，歴史的には色々あったが，今まで Peter Norvigによる Paradigms of Artificial Intelligence Programming (略称PAIP) に勝るものはなかった．PAIP ホームページには本に記述されているプログラムが公開されている．また，人工知能についての包括的なテキストである Artificial Intelligence, Modern Approach (略称AIMA) のホームページにも主なコードが Common Lisp で公開されて，今では GITHUBリポジトリからダウンロードできる．ところが，PAIPの出版は1992年，AIMA第1版が1995年，第 2 版が2003年，第3版が2010年, 第4版が2021年であるにもかかわらず，PAIPが書かれたのがほぼANSI Common Lisp策定時であるためにそれに準拠していないのはしかたがないとしても，AIMAプログラムも同様である．そのため，それらのコードを部分的に取り出して今のCommon Lispで使おうとしても，すぐ右から左に使えるというわけにはいかない．また，今日では当然CLOSを使うところであっても，残念ながらPAIPではCLOSを利用していないし，AIMAではCLOSを使わずに構造体を拡張した独自のCLOS風システムを使っているありさまである．また，両者ともシステム管理に今では事実上の標準とも言える ASDF や Quicklisp が使われていないなどの問題もある．本書ではその穴を埋めるべく，古いPAIPプログラムコードの現代化を行っている．

また，シンボルに大文字小文字を許すACL mlispを使おうとすると，PAIPコードやAIMAコードが，大文字小文字を区別するコードになっていないために，そのままでは動かない．ここでは最初からmlispでもSBCLでも稼働する人工知能用プログラムのソースコードを提供する．幸いなことに，現在ではPAIPとAIMAのソースコードはオープンソースとして公開されるようになったので，PAIPとAIMAのソフトウェア・ライセンス・アグリーメントに従う限りは，ここで現代化されたPAIPコードを掲載しても何ら著作権上の問題が発生しない．さらに本書籍では，PAIP／AIMA由来のコードだけでなく，その他の人工知能研究者らによるプログラムコードも，差し支えない範囲で現代化し提供している．

21.2 人工知能用ユーティリティプログラム

PAIP では auxfns.lisp ファイルにて，AIMA では utilities フォルダ中のファイルにてそれぞれのプログラムで利用される各種ユーティリティが一括して記述してある．PAIP あるいは AIMA のプログラムを利用するためには必ずこれらのプログラムを先にロードするということになるのだが，本書籍では各章ごとの独立性を考えて，それぞれの章で最低限必要なものだけをロードするようにした．第2部のプログラムは各章ごとにGitHubに置くようにしてユーティリティも各章ごとにファイルを用意したが，それらユーティリティプログラムの説明は各章における重複を避けるために，ここでまとめて説明する．

リスト用関数

- mappend は副作用のない mapcan と思ってもらってよい．実際よほどのことがないかぎり，本書籍でもmapcanではなくこちらを使っている．実際に以下の二つの実装があるが，引数に与えるリストが 1 個ならばどちらでも効果は同じである．

```
(defun mappend (fn list)
  "Append the results of calling 《fn》 on each element of 《lists》.
  Like mapcan, but uses append instead of nconc."
  (apply #'append (mapcar fn list)))
```

```
(defun mappend (fn &rest lists)
  "Apply 《fn》 to respective elements of 《lists》, and append results."
  (reduce #'append (apply #'mapcar fn lists) :from-end t))
```

- 要素数1のリスト（シングルトン）とその要素だけのアトムを区別なく扱いたいことがよくある．そういう場合にはシングルトンを標準として，たまたまアトムであるものをシングルトンに変更するが，それを一言でイディオムのように使うのがこの mklist である．

```
(defun mklist (x)
  "If 《x》 is a list return it, otherwise return the list of 《x》"
  (if (listp x) x (list x)))
```

- length=1 は要素数1のリストのときに t を返す．

```
(defun length=1 (x)
  "Is 《x》 a list of length 1?"
  (and (consp x) (null (cdr x))))
```

- starts-with は第 1 引数であるリストの最初の要素が第2引数にeqlのときに t を返す．

```
(defun starts-with (list x)
    "Is x a list whose first element is x?"
    (and (consp list) (eql (first list) x)))
```

- flatten はリストの表面だけを舐めて，一段だけ括弧をはずす．

```
(defun flatten (exp)
  "Get rid of imbedded lists (to one level only)."
  (mappend #'mklist exp))

(flatten '(a (a (a (a b))) (((a b) b) b) b))
 ⇒ (a a (a (a b)) ((a b) b) b b)
```

- squash はリストの深部まで含めてリストをはずして，得られる結果はアトムだけのリストである．

```
(defun squash (x)
  "flattens a nested list 《x》 and returns a list that includes only atoms."
  (cond ((consp x) (mappend #'squash x))
        (t (list x))))

(squash '(a (a (a (a b))) (((a b) b) b) b))
 ⇒ (a a a a b a b b b b)
```

- find-if はリストの表面だけしか調べないが，find-if-anywhereはリストの中まで調べる．

```
(defun find-if-anywhere (predicate tree)
  "Does 《predicate》 apply to any atom in the 《tree》?"
  (if (atom tree)
      (funcall predicate tree)
      (or (find-if-anywhere predicate (first tree))
          (find-if-anywhere predicate (rest tree)))))
```

- random-elt はシーケンスの要素をランダムに1個選択する．

```
(defun random-elt (seq)
  "Pick a random element out of a sequence 《seq》."
  (elt seq (random (length seq))))
```

- Common Lisp 標準の sort はリストの中の並びを破壊的に変えてしまう．普通は副作用のないこちらを使う．

```
(defun sort* (seq pred &key key)
  "Sort without altering the sequence 《seq》."
  (sort (copy-seq seq) pred :key key))
```

queue

リストをスタックと見立てたときのプッシュとポップは Common Lisp の push と pop を使えばよい．一方，queue というのは先入れ先出しの列のことである．待ち行列と言っていい．これを普通のリストで実現しようとすると，列の最後にオブジェクトを置くために，その列を頭からトラバースしなければならない．これはリストの長さが長いとコストが高い．それを避けるために，リストの最後のセルをポイントするポインタを持った特別なリストを準備する．queue へのアクセスは必ずこの専用の enqueue，dequeue を使う．

```
;;; A queue is a (last . contents) pair

(defun queue-contents (q) (cdr q))

(defun make-queue ()
  "Build a new queue, with no elements."
  (let ((q (cons nil nil)))
    (setf (car q) q)))

(defun enqueue (item q)
  "Insert item at the end of the queue."
  (setf (car q)
```

```
        (setf (rest (car q))
              (cons item nil)))
  q)

(defun dequeue (q)
  "Remove an item from the front of the queue."
  (pop (cdr q))
  (if (null (cdr q)) (setf (car q) q))
  q)

(defun front (q) (first (queue-contents q)))

(defun empty-queue-p (q) (null (queue-contents q)))

(defun queue-nconc (q list)
  "Add the elements of 《list》 to the end of the queue."
  (setf (car q)
        (last (setf (rest (car q)) list))))
```

reuse-cons

cons は実行されるたびにセルをプールから一つ取ってくる．our-copy-list ではconsセルをとってきて見かけ上は似たようなリストを作りだすが，このようにわざわざセルを取ってくるのが目的の場合は実はあまり多くない．実際には無駄に cons セルを消費しているのがほとんどである．cons セル節約のために，すててしまうようなセルをもう一度再利用して，最低限必要な場合だけセルを消費するのが reuse-cons である．

```
(defun reuse-cons (x y x-y)
  "Return (cons x y), or reuse x-y if it is equal to (cons x y)"
  (if (and (eql x (car x-y)) (eql y (cdr x-y)))
      x-y
      (cons x y)))
```

debug

人工知能プログラム中で何が起こっているのか調べたいときに，この debug 機能が用いられる．細かくデバッグしたいときは主要な関数にtrace を掛ければいいのだが，それだと細かすぎるとか，何が起こっているのか目的別に調べたいという場合に，コードの中に関数 dbg を仕込んでおいて，任意のスウィッチシンボルで必要な情報を印刷する．

```
;;;; The Debugging Output Facility:

(defvar *dbg-ids* nil "Identifiers used by dbg")

(defun dbg (id format-string &rest args)
  "Print debugging info if (DEBUG-ID ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (apply #'format *debug-io* format-string args)))

(defun debug-id (&rest ids)
  "Start dbg output on the given ids."
  (setf *dbg-ids* (union ids *dbg-ids*)))
```

```
(defun undebug-id (&rest ids)
  "Stop dbg on the ids.  With no ids, stop dbg altogether."
  (setf *dbg-ids* (if (null ids) nil
                      (set-difference *dbg-ids* ids))))

(defun undebug-all ()
  (undebug-id))
```

使い方は trace の場合とよく似ている．ただし，trace には関数名を与えるが，debug の場合は任意のシンボルでよく，デバッグ用 id としてユーザがそれを指定する．するとコード中に仕込まれた `(dbg id ...)` の *id* がマッチしたところで印刷がされるという仕組みだ．具体例は 24.3節を参照されたい．

delay と force

Lisp では関数引数はすべて関数実体に引き渡される前に評価される．そうではなく，if 書式のように，場合によって評価したりしなかったりするには，組み込みの特殊形式以外ではマクロが用いられた．最近のビッグデータのように膨大なデータを扱うようになると，一律に評価するというしくみではうまくない場合が出てくる．defvar の既定値のように本当に必要になるまで評価を実行しないということができるとうれしい場合がある．それが遅延評価である．Haskell は関数呼び出し時にパラメータの遅延評価をするプログラム言語として有名であるが，Common Lisp でもここで紹介する delay と force を明示的に用いることで Haskell ほど徹底してはいないが，Scheme と同程度のプログラミングにて遅延評価を行うことができる．

```
(defstruct delay value (computed? nil))

(defmacro delay (&rest body)
  "A computation that can be executed later by FORCE."
  `(make-delay :value #'(lambda () . ,body)))

(defun force (delay)
  "Do a delayed computation, or fetch its previously-computed value."
  (if (delay-computed? delay)
      (delay-value delay)
      (prog1 (setf (delay-value delay) (funcall (delay-value delay)))
             (setf (delay-computed? delay) t))))
```

パターンマッチ

人工無脳のElizaからプラニングやPrologまで，至る所で出てくるのが変数を含むパターンマッチの機能である．ユニフィケーション（単一化）の本格的な説明は第29章に譲るが，Elizaにも用いられるコードはここに掲げておく．

```
;;;; PATTERN MATCHING FACILITY

(defconstant fail nil)
(defconstant no-bindings '((t . t)))

(defun match-variable (var input bindings)
  "Does VAR match input?  Uses (or updates) and returns bindings."
```

```
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
          ((equal input (binding-val binding)) bindings)
          (t fail))))

(defun make-binding (var val) (cons var val))

(defun binding-var (binding)
  "Get the variable part of a single binding."
  (car binding))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))

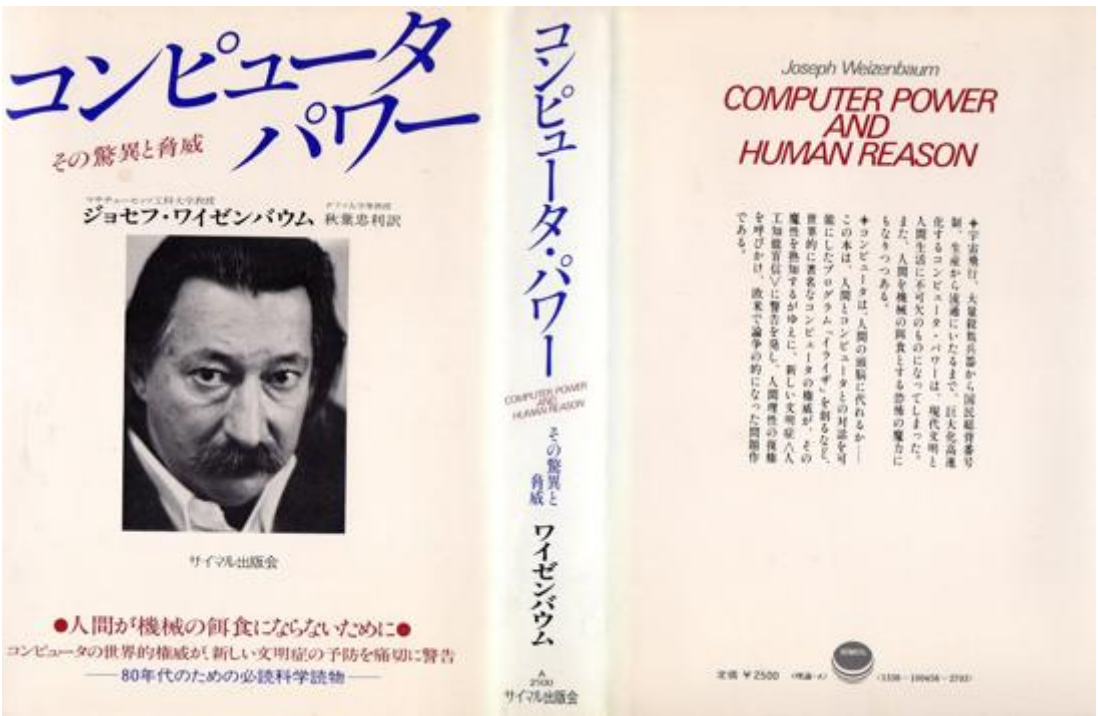
(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy no-bindings
        (if (eq bindings no-bindings)
            nil
            bindings)))
```

第22章 Elizaまたの名を人工無脳

> 私のショックは，[...]私の行った研究を誤って解釈し続ける人びとによってもたらされた．このように書いても，私は不平をいっているわけではなく，自己弁護しているのでもない．事実この誤った解釈は私の行った研究とその意義を，どちらかというと過大評価しているものなのである．
>
> [Weiznbaum, 1976]（コンピュータパワー，秋葉訳）

1966年頃にMITの教授だったジョセフ・ワイゼンバウムは，ロジャー学派の精神科医の模倣をするプログラム Eliza を作った．彼自身はたいそうなAIプログラムを作ったという意識はなかったが，それに対する人々の反響や誤解に基づく過大な期待，さらには彼がElizaを作る過程を知っているはずの秘書までもElizaに示した反応などにショックを受けて，「人工知能批判派に転じてしまった」と言われている．筆者自身は彼がいわゆる「人工知能批判派」とは思っていないが，人工知能歴史上のこのエピソードは，その後1980年代の第二次AIブームのとき，そして2015年の第三次AIブームにおいても度々繰り返されるところの，一般の人々からの人工知能に対する誤解，過大な評価・期待と，その裏返しである恐怖・失望の最初の事例だった．

この件について興味を持たれたのなら，ワイゼンバウム自身の手になる「コンピュータパワー」を入手して一読することをお勧めする．もしこれから人工知能に真剣に取り組もうとするのであれば，これは必読書の一つである．



22.1 Elizaとは何か？

GitHubリポジトリからダウンロードした eliza システムを実施すると，たとえば以下のような実施例が得られる．

```
cl-user(2): (eliza)

eliza> hello.
How do you do. Please state your problem.
eliza> My problem is about my father.
Your father
eliza> Yes, he is gone.
Go on
eliza> But, I am not moved by it.
Do you want to be not moved by it ?
eliza> No, no. I think people usually feel sad when their parents die.
You are being a bit negative
eliza> Is it so? Anyway I think I am not normal.
In what way are you not normal
eliza> I cannot cry after the death of my father.
Does he influence you strongly?
eliza> Well, I think not so much.
Go on
eliza>
```

この例は少しできすぎの感がある．如何にも流ちょうな会話をしているように見えるが，その理由は，eliza はロジャー流のサイコセラピストの模倣であり，ロジャー流ではセラピスト自身はあまりしゃべらず，患者になるべく多くを語らせるようにするが，eliza はパターンが見つからない場合は，「続けて」とか「興味深い」とか，多くの場合にあてはまるような，あたりさわりのないあいづちを打つようになっているのが，結局ロジャー流にもっともらしく見えるからである．

Elizaは会話の意味は理解していない．ただ単に特定の入力パターンに対して特定の文章を出力するだけである．たとえば，「hello」を含む入力文には「How do you do? Please state your problem.」を返し，「my father」を含む入力文には「Your father」とか「Does he influence you strongly?」という文章を返すだけである．ただし，単なる固定的なおうむ返しの反応だけでなく，「I am ～」に対して「In what way are you ～」とか「Do you want to be ～ ?」といった，入力文の一部を使って出力文を作ったりする．だからその返し方がツボにはまると会話がもっともらしく続くが，世界に対する知識は何も持っていないし，相手の問いに答えることもできない．

22.2 ダウンロードと eliza1 の実行

GitHubリポジトリから eliza システムをダウンロードし，まずは eliza1 を実行してみる．以下は Windows のドキュメントフォルダ中にGitHubフォルダを作り，そこにelizaのクローンを作ったときの ACL mlisp による実行例である．

```
cl-user(1): :cd
C:\Users\誠二\
cl-user(2): :cd Documents/GitHub/eliza
C:\Users\誠二\Documents\GitHub\eliza\
cl-user(3): :ld eliza.asd
; Loading C:\Users\誠二\Documents\GitHub\eliza\eliza.asd
;   Fast loading C:\acl90-smp.64\code\asdf.005
;;; Installing asdf patch, version 5.
cl-user(4): (asdf:load-system :eliza)
; Autoloading for excl::gray-open:
```

```
; Fast loading from bundle code\streamc.fasl.
;   Fast loading from bundle code\efft-932-base.fasl.
;   Fast loading from bundle code\efft-utf8-base.fasl.
;   Fast loading from bundle code\efft-void.fasl.
;   Fast loading from bundle code\efft-latin1-base.fasl.
; Autoloading for class echo-stream:
; Fast loading C:\acl90-smp.64\code\streama.002
;;; Installing streama patch, version 2.
;;; Compiling file C:\Users\誠二\Documents\GitHub\eliza\elizaaux.lisp
;;; Writing fasl file
;;;   C:\Users\誠二\AppData\Local\cache\common-lisp\acl-9.0ms-win-x64\C\Users\誠二
\Documents\GitHub\eliza\elizaaux-tmp2f8c10c2.fasl
;;; Fasl write complete
; Fast loading
;    C:\Users\誠二\AppData\Local\cache\common-lisp\acl-9.0ms-win-x64\C\Users\誠二
\Documents\GitHub\eliza\elizaaux.fasl
;;; Compiling file C:\Users\誠二\Documents\GitHub\eliza\eliza1.lisp
;;; Writing fasl file
;;;   C:\Users\誠二\AppData\Local\cache\common-lisp\acl-9.0ms-win-x64\C\Users\誠二
\Documents\GitHub\eliza\eliza1-tmpepqqcvc7.fasl
;;; Fasl write complete
; Fast loading
;    C:\Users\誠二\AppData\Local\cache\common-lisp\acl-9.0ms-win-x64\C\Users\誠二
\Documents\GitHub\eliza\eliza1.fasl
;;; Compiling file C:\Users\誠二\Documents\GitHub\eliza\eliza.lisp
;;; Writing fasl file
;;;   C:\Users\誠二\AppData\Local\cache\common-lisp\acl-9.0ms-win-x64\C\Users\誠二
\Documents\GitHub\eliza\eliza-tmppwgw9g2d.fasl
;;; Fasl write complete
; Fast loading
;    C:\Users\誠二\AppData\Local\cache\common-lisp\acl-9.0ms-win-x64\C\Users\誠二
\Documents\GitHub\eliza\eliza.fasl
t
cl-user(5): (eliza1)

eliza>
```

ASDFはコンパイルファイルが無ければソースファイルをコンパイルし，コンパイルファイル(faslファイル)をキャッシュに置く．このキャッシュ場所はWindows，Mac，Linuxで異なるのは当然であるが，また各Lisp処理系でも異なっているので，安心して同一コンピュータ上で各種のLisp処理系を使うことができる．ただし上記例はACL mlisp without IDEの場合であるが，ACL mlisp with IDE とは混在できないので留意されたい．ASDFはソースファイルが変更されれば，そのファイルのみをロード時に自動的にコンパイルしてからロードする．

これでとりあえず，eliza1 が実行できるようになった．試しに実行して色々入力してみよう．ただし，eliza1では括弧で入力文全体を囲うようにする．

```
eliza> (hello there)
(How do you do. Please state your problem.)
eliza> (I want to test this program)
(Suppose you got to test this program soon)
eliza> (OK. I am doing.)
(In what way are you doing.)
eliza> (are you OK?)
```

```
(Why are you interested in whether I am OK? or not?)
eliza>
```

(I want ...)の答えがこの通りでなくても気にしないように．Eliza は複数ある答え候補の中からランダムに答えを選択して返してくる．

eliza1のプログラムでは，この会話を終わる方法はないことに留意されたい．

22.3 eliza のコードウォークスルー

以下に eliza1 のコードを示す．

```
(defun eliza1 ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)(force-output)
    (write (flatten (use-eliza-rules (read))) :pretty t)))
```

関数 eliza1 のすることは，Lisp のREPLのすることとよく似ているが，出力するのは eval の結果ではなく，(flatten (use-eliza-rules input)) の結果であるリストである．ここでは出力にwriteを使っているが，キーワードパラメータ :pretty を真としている．さもないと出力リストの長さが長いと省略されてしまうからである．flattenについては前章で説明した．このプログラムでeliza特有のところはuse-eliza-rulesにある．それはこのようなコードになっている．

```
(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (some #'(lambda (rule)
            (let ((result (pat-match (rule-pattern rule) input)))
              (unless (eq result fail)
                (sublis (switch-viewpoint result)
                        (random-elt (rule-responses rule))))))
        *eliza-rules*))
```

*eliza-rules*にはルール集合がリストとして保持されている．一つのルールはパターン部と応答部からなっており，パターン部は入力文に対応する1個のパターンであるが，応答部は通常複数の応答文から成っている．ルール集合の最後だけはどんな入力にもマッチするようなパターンでなければならないことを注意しておく．

```
(setq *eliza-rules*
 '((((?* ?x) hello (?* ?y))
    (How do you do.  Please state your problem.))
   (((?* ?x) computer (?* ?y))
    (Do computers worry you?)
    (What do you think about machines?)
    (Why do you mention computers?)
    (What do you think machines have to do with your problem?))
      ...
   (((?* ?x) bye)
    (good bye and good luck))
   (((?* ?x))
    (Very interesting)
    (I am not sure I understand you fully)
```

```
    (What does that suggest to you?)
    (Please continue)
    (Go on)
    (Do you feel strongly about discussing such things?))))
```

use-eliza-rules はルールの一つ一つについて，そのパターン部と入力の pat-match をとり，その結果が失敗でなかったらルールの応答部を取り出して（リストになっている），random-elt でその中のどれかを選択し，パターンマッチの結果を switch-viewpoint したもので，要素の置き換えを行う．sublis はCommon Lisp の関数で，第1引数に連想リストという特殊な形式のリストを受け，第2引数のツリーリスト中の要素の置き換えを行う．以下に例を示す．

```
cl-user(4): (sublis '((this . that) (that . this))
                    '(this is (better (than that))))
(that is (better (than this)))
```

switch-viewpoint は引数リスト中にある “you”を“I”に，“I”を“you”に置き換え，“me”を“you”に，“am”を“are”に置き換えることだけである．ここでは目的語となる“you”を“me”には置き換えないことに注意してほしい．そもそも eliza に言語解析機能はないのである．

```
(defun switch-viewpoint (words)
  "Change I to you and vice versa, and so on."
  (sublis '((I . you) (you . I) (me . you) (am . are))
          words))
```

一番肝心なところはパターンマッチを実現する pat-match にある．そのコードは以下のようになっている．

```
(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((segment-pattern-p pattern)                ; ***
         (segment-match pattern input bindings))    ; ***
        ((and (consp pattern) (consp input))
          (pat-match (rest pattern) (rest input)
                     (pat-match (first pattern) (first input)
                                bindings)))
        (t fail)))
```

これも典型的な再帰構造をしている．pattern と input と bindings を入力として，pattern と input がリストだったら再帰している．再帰が終わるのはパターンマッチに失敗したとき(fail)と，成功した時だが，成功には 2 種類があって，pattern が変数でない場合には (eql pattern input)，bindings を返し，変数の場合には match-variable の実行結果を返している．要素がリストの場合には再帰するが，パターンマッチが成功しなかった場合には，最後に失敗(fail)を返している．segment-pattern が何をするかは今置いておいて，これを実行するとこんな感じになる．

```
cl-user(5): (pat-match '(this is easy) '(this is easy))
((t . t))
```

```
cl-user(6): (pat-match '(it is ?x) '(it is fine))
((?x . fine))
cl-user(7): (pat-match '(this is easy) '(this is not easy))
nil
cl-user(8): (pat-match '(it is ?x) '(it is not fine))
nil
cl-user(9): (pat-match '(it is ?x) '(it is ?y))
((?x . ?y))
```

先頭文字が?であるシンボルは何にせよこのパターンマッチにおける変数であり，それは何にでもマッチする．変数でないシンボルは，正確に同じものにしかマッチしない．比較する要素がリストであっても，それらがリストとして同じかどうかが試される．

```
cl-user(10): (pat-match '(this is (a list)) '(this is (a list)))
((t . t))
cl-user(11): (pat-match '(this is (a list)) '(this is also (a list)))
nil
cl-user(12): (pat-match '(this is ?) '(this is (a list)))
((? a list))
```

変数がなく完全一致したときに返されるものは `((t . t))` だが，変数でマッチした時にはその束縛情報が返されることに注意されたい．

さて次は segment-match である．これまでのパターンではリスト構造も含めて一つずつ要素ごとにマッチのための比較が行われた．segment-match は変数がリスト中の任意のセグメントとマッチすることを可能にする．たとえば，こんなふうになる．

```
cl-user(15): (pat-match '((?* ?x) jump) '(1 2 3 jump))
((?x 1 2 3))
cl-user(16): (pat-match '((?* ?x) need (?* ?y)) '(I and you need money and rest))
((?y money and rest) (?x I and you))
```

ここで，`(?* ?x)` は正規表現で言えば「{ ?x }*」みたいなもので，任意個の要素のならびにマッチさせるための表現である．segment-pattern-p はそれを判定する．

```
(defun segment-pattern-p (pattern)
  "Is this a segment matching pattern: ((?* var) . pat)"
  (and (consp pattern)
       (starts-with (first pattern) '?*)))
```

実際のセグメントパターンマッチのコードは少し複雑になる．

```
(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        ;; We assume that pat starts with a constant
        ;; In other words, a pattern can't have 2 consecutive vars
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
```

```
                fail
                (let ((b2 (pat-match
                           pat (subseq input pos)
                           (match-variable var (subseq input 0 pos)
                                           bindings))))
                  ;; If this match failed, try another longer one
                  (if (eq b2 fail)
                      (segment-match pattern input bindings (+ pos 1))
                      b2)))))))
```

セグメントパターンの場合には，入力中にパターン(pat)の第1要素と一致するものがあるか調べられる．あればその場所(pos)が得られる．`(subseq input pos)` によりセグメント変数にマッチした入力中のセグメントが取り出され，さらにパターンマッチが行われるが，このとき残りの後半部についてのパターンマッチが先に実行され（上記下線部），その束縛情報でもって前半のパターンマッチが実行される．もしそれが失敗なら，もっと前半のセグメントを長くとったら，後ろの部分でマッチするかも知れず，startに 1 をプラスして再帰する．成功だったのなら 得られた束縛情報を返す．

22.4 elizaプログラムの改良

先に実行した eliza1 は入力全体を括弧で括らなければならないとか，句読点を入力することができないなどの問題があった．それは Lisp のread 関数をそのまま利用していたからである．

```
cl-user(5): (eliza1)

eliza> (If I have a question, what should I do?)

Error: Comma not inside a backquote.
  [condition type: reader-error]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] cl-user(6):
```

read-line-no-punct はいったん括弧なしの入力を文字列として読み込んで，その中に句読点があったら，それを空白文字に置き換えたものを改めて括弧付きで読み込む．

```
(defun read-line-no-punct ()
  "Read an input line, ignoring punctuation."
  (read-from-string
    (concatenate 'string
                 "("
                 (substitute-if #\space #'punctuation-p (read-line))
                 ")")))

(defun punctuation-p (char) (find char ".,;:!?#-()\\\""))
```

このようにしても，elizaルールのパターンマッチのパターンに句読点が含まれないかぎり問題は生じない．

また，出力も括弧でくくられていた．ごく簡単な関数でこれらの制限を無くしたものが次の eliza のコードである．

```
(defun eliza ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)(force-output)
    (let* ((input (read-line-no-punct))
           (response (flatten (use-eliza-rules input))))
      (print-with-spaces response)
      (if (pat-match '((?* ?x) good bye (?* ?y)) response) (return)))))

(defun print-with-spaces (list)
  (format t "~{~a ~}" list))
```

このプログラムでは，応答文中に「good bye」があったら実行終了するようになっている．

問題 22.1 elizaを起動して，色々入力してみよう．elizaはうまく答えてくれるだろうか．最後終わるには，文章のどこかに「bye」を入れればよい．

22.5 ルールの追加

8.4節でelizaルールを追加するマクロを示した．ここに再掲すればそれは次のようなマクロである．ただし，既存のルールに追加するために，一番最後のどんなときにもマッチするルールを常に最後に置くために下線部の追加修正をしていることを注意しておく．

```
(defmacro defrule (name &rest body)
  (let ((pattern (gensym))
        (responces (gensym)))
    `(let ((,pattern
            ',(butlast body
                       (- (length body)
                          (position '-> body))))
           (,responces
              ',(cdr (last body
                           (- (length body)
                              (position '-> body))))))
       (setf (get ',name 'eliza-rule)
             (cons ,pattern ,responces))
       (setq *eliza-rules*
             (append (butlast *eliza-rules*)
                     (list (get ',name 'eliza-rule))
                     (last *eliza-rules*)))
       ',name)))
```

それではこのマクロを使ってルールを追加してみよう．私が追加したルールはたとえばこんなもの．

```
cl-user(8): (defrule 挨拶
               お願いします ->
              (どうかしましたか))
挨拶
cl-user(9): (defrule 不眠
               (?* ?z) 眠れないのです ->
              (気にかかっていることは 何ですか？))
```

```
不眠
cl-user(10): (defrule したいこと
                (?* ?z) したいのです ->
               (どうして ?z したいのですか？))
したいこと
cl-user(11): (defrule 否定的
                (?* ?x) ないのです ->
               (そんなに否定的に思う必要はありませんよ)
               (どうしてそう思うのですか？))
否定的
cl-user(12): (defrule 抽象的
                (?* ?x) 私は (?* ?y) ->
               (続けてください))
抽象的
cl-user(13): (defrule 思い
                (?* ?x) 思います ->
               (よく ?x 思うのですか？)
               (ほかにどんなことを思いますか？))
思い
cl-user(14): (defrule 終了
                (?* ?z) 失礼します ->
               (good bye ごきげんよう))
終了
```

これで如何にも作為的ではあるが，次のような実施例が得られる．

```
cl-user(33): (eliza)

eliza> お願いします
どうかしましたか
eliza> 最近 どうも 元気が ないのです
どうしてそう思うのですか？
eliza> 私は 小さいときから 親からしかられたことがないのですが，
続けてください
eliza> それで はたして よかったのかなと 思います
よく それで はたして よかったのかなと 思うのですか？
eliza> はい． よく そう 思います
ほかにどんなことを思いますか？
eliza> そうですね．もし兄弟がいたら 私は どうだったろうかとか
続けてください
eliza> そうそう．最近は よく 眠れないのです
気にかかっていることは 何ですか？
eliza> いや，時間がないんで，これで 失礼します
good bye ごきげんよう
nil
```

英語では単語が空白で分離しているため問題はないが，日本語の場合にはパターンマッチのキーとなる単語は分かち書きをしてやらなければならない．分かち書きしない日本語に対応するためには形態素解析が必要となる．

問題22.2 defruleを使って，自分流にルールを拡張せよ．

22.6 Eliza とは何であったか？

Elizaのしていることは，入力パターンに応じた出力をしているだけで，文の意味を理解しているわけではない．短期記憶も長期記憶も無ければ，世界に関する知識もない．短期記憶がなければ直前の会話の内容を踏まえた応答もできないわけで，人工知能ならぬ人工無脳と言われる所以である．しかし1960年代に作られたElizaは，人間の認知メカニズムについての知見が大きく進んだ現在でも，知能とは何かという根本的な問いを我々に投げかけている．対話エージェントの原型とも言えるこのプログラムに，形態素解析機能をつけ，構文解析機能をつけ，照応解析機能をつけ，意味解析機能をつけたら，それは人工知能と言えるのだろうか？

短期記憶のない認知症の患者でも，一見認知症とは思えないほどの会話のやりとりをするのを見ると，短期記憶のないElizaでもElizaルールを次々と拡張して，一体どこまで行けるのか，軽度の認知症患者を真似るところまで行けるのかという疑問がわく．逆に，何ができれば人工知能といえるのか，Elizaに何を追加すれば人工知能になるのか，記憶は必要だろうが，そのほかの機能は何が必要なのか，一体会話を理解するとはどういうことなのかという疑問も．

一方，この程度のプログラムでも，人間の振る舞いを真似るプログラムに人間の側が大きく動かされてしまうという事実も，人間と人工知能との関係に注意すべき点があるということを教える．会話動作を含めて外見上行動が人間的でありさえすれば，ヒトは人工物でも相手を同類と捉えてしまう．Siriは少し会話を進めるとたちまち馬脚を現わしてしまって，Siriに没入する人はいないが，会話型エージェントの代表格は今では映画「her」に登場するサマンサであろう．この映画では主人公とサマンサとの感情の交流とサマンサの成長・進化が軸となってストーリーが展開していく．つまり，ヒトは人工物と感情の交流が可能であることが前提とされている．この場合問題となるのが個体性の問題である．一見相手は自分と唯一の関係を持っているように見えながら，実はサマンサは同時に8316人の人間と個別の会話をし，641人の自分と同じような恋人がいることを知ったとき，主人公は愕然としてしまう．はたしてエモーショナル・エンジンを持つエージェントに個体性は必然なのか．

人間に分かりやすくするために，対話型システムでは時々擬人化エージェントが利用される．たとえば，映画「A.I.」では歓楽街Rouge Cityの中でどんな質問にも答えてくれるエージェントDr. Knowが登場するが，それはいかにもそれっぽい「博士」みたいな風貌をしているし，スカリー時代のAppleが作ったプロモーションビデオ「Knowledge Navigator」に登場するエージェントは「執事」という位置づけになっている．一方，スタートレックやSpace OdysseyのHALのように，宇宙船全体を統括するような全体制御システムは，インタフェースとしての対話機能があっても擬人化されないようだ．それはシステム全体を代表する存在なのだ．CortanaやSiri APIがデベロッパーにも公開されることになって，改めてこれから知的インタフェースとしての対話機能や個別タスク用擬人化エージェントの研究開発が進むであろうが，1960年代に開発されたElizaは対話機能の原型を提示したものとして，歴史的な意義がある．Elizaプログラムの系譜については Jeff Shrager によってまとめられたサイトがあるので，興味のある方は参考にされたい．

第23章 一般問題解決器

> GPS を覚えているかな？今では「GPS」はパズルを解く大いに愚かなプログラムを指す何の色もついてない用語だが，元々は「一般問題解決器， *General Problem Solver* 」を意味していて，当時はそれに誰もが不必要なほど興奮したり動揺した．だがそれは本来 LFGNS ― 「局所的特徴にガイドされたネットワーク探索器， *Local Feature-Guided Network Searcher* 」と呼ばれるべきだった．
>
> [McDermott, 1976]

23.1 手段・目標解析，Means-Ends Analysis

一般問題解決器(General Problem Solver)とは，まるでそれで何でも解くことができると，人々に容易に誤解を起こさせるようなおおげさな名前だが，人工知能の歴史の初期に生まれた人工知能システムである．**目標**（ゴール）と現在の状態と**手段**（オペレーション）が与えられたとき，ゴールの**サブゴール**を見つけて現状と目標との差を無くすように手段（オペレーション）を適用することから，GPS が問題を解くやりかたを，**手段・目標解析**と呼ぶようになった．アレン・ニューウェル と ハバート A. サイモン (1961) による GPS プログラムの説明では概略以下のように解説される(訳は阿部，横山監訳による)．

- 目標：対象Aを対象Bに変換せよ
    1. 対象Aと対象Bを照合し差Dを見つけよ => D
    2. Dを縮小というサブ目標を設定 => A'
    3. A'をBに変換というサブ目標を設定 => B
- 目標：対象Aと対象Bとの間の差Dを縮小せよ
    1. Dを縮小するのに適した作用子Qを探せ => Q
    2. 実行可能かどうかテストせよ => if 可 3 else 1
    3. AにQを適用というサブ目標を設定=> A' else go 1
- 目標：対象Aに作用子Qを適用せよ
    1. Qの条件をAと照合し差Dを見つけよ => D
    2. Dを縮小というサブ目標を設定 => A' else 失敗
    3. A'にQを適用というサブ目標を設定 => A'' else 失敗

対象AとBの間の差Dを発見し，その差Dを縮小することを新たなサブ目標とする．このサブ目標を達成すると新しい対象A'が得られる．このA'とBの間にはもはや差Dはない．次にA'をBに変換することを新たなサブ目標とする．こうして再帰的に対象Aは対象Bに変換される．Dを縮小するという目標は，最初にそれにふさわしい作用子を発見し，実行可能ならばそれを実行するが，実行不可であればそれが見つかるまで探索する．実際に適用すれば新たな対象が生成される．対象Aに作用子Qを適用するには，Qの付帯条件と対象Aを比較してもし差があればその差を縮小することを新たなサブ目標とする．そしてこの目標から得られた新たな対象にQを適用する．ここでも再帰的に実行される．詳細な記述は原著にあたってほしいが，オリジナルのプログラムは IPL で書かれていて，そのせいもあってプログラムは非常に複雑で分かりにくいものであった．それを Novig は PAIP の中でプラニングシステム STRIPS 流のいくつかの概念を用いて Common Lisp で書き直して簡単化した．以下では PAIP におけるプログラムをとりあげて GPS の説明をする．

23.2 状態，目標，作用子

GPS における主要なデータは，今現在の状態を示す *state*，スタートからエンドまで様々な計画段階での目標である goal，それに現在の状態 *state* に働いてそれを変更する作用子 op である．

```
(defvar *state* nil "The current state: a list of conditions.")

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))
```

state は個々の状態を指示するシンボルのリストである．たとえば，古典的な AI 問題である，Monkey & Banana では，最初は，おなかをすかせた猿がおもちゃのボールを持って，ドアの入口に立っており，ドアの近くには椅子があって，部屋の真ん中にはバナナが天井から吊るされている状態である．それをこんな感じのリストで表現する．

初期状態：(at-door on-floor has-ball hungry chair-at-door)

ゴールも同様にシンボルのリストで表現される．たとえば，Monkey & Banana 問題のゴールは猿がおなかがすいてない状態である．

ゴール：(not-hungry)

構造体 op のインスタンスとして，個々の作用子を定義する．たとえば，食べるという行為で，おなかがすいていた状態からすいていない状態に変化する作用子は次のようなものである．

```
eat-banana：(make-op :precond '(has-bananas)
                     :add-list '(empty-handed not-hungry)
                     :del-list '(has-bananas hungry))
```

この作用子を適応すると GPS は *state* から del-list に記述してあるシンボルを削除し，add-list にあるシンボルを *state* に加える．precond にはその作用子を適用するのに満たさなければならない条件を書いておく．この場合それは has-bananas である．もちろん手にもっていなければバナナは食べられない．

初期状態と最終ゴールの記述に加えて，必要十分な作用子をすべて記述して，対象領域の記述が完成する．Monkey & Banana のすべての作用子は以下のとおり．

```
(defparameter *banana-ops*
  (list
   (make-op :action 'climb-on-chair
            :preconds '(chair-at-middle-room at-middle-room on-floor)
            :add-list '(at-bananas on-chair)
            :del-list '(at-middle-room on-floor))
   (make-op :action 'push-chair-from-door-to-middle-room
            :preconds '(chair-at-door at-door)
            :add-list '(chair-at-middle-room at-middle-room)
            :del-list '(chair-at-door at-door))
   (make-op :action 'walk-from-door-to-middle-room
            :preconds '(at-door on-floor)
            :add-list '(at-middle-room)
```

```
          :del-list '(at-door))
    (make-op :action 'grasp-bananas
          :preconds '(at-bananas empty-handed)
          :add-list '(has-bananas)
          :del-list '(empty-handed))
    (make-op :action 'drop-ball
          :preconds '(has-ball)
          :add-list '(empty-handed)
          :del-list '(has-ball))
    (make-op :action 'eat-bananas
          :preconds '(has-bananas)
          :add-list '(empty-handed not-hungry)
          :del-list '(has-bananas hungry))))
```

action には eat-bananas とか climb-on-chair といった猿の行為を表すシンボルが与えられるが，これは全く人へのメッセージのために使われるだけであり，GPS から見て重要なものは，事前条件 preconds と，追加リスト add-list および削除リスト del-list に書かれる内容である．

23.3 GPS はどのように働くのか？(第 1 版)

前章と同様に，GitHubリポジトリから GPS システムをダウンロードし，まずは GPS1 を実行してみる．GPS1の第 1 引数が初期状態，第 2 引数がゴール，第 3 引数が作用子のリストである．

最初，猿はドアの入口に立っていて(at-door)，手にボールを持って(has-ball)，床の上にたっている(on-floor)．ドアのそばには椅子がある(chair-at-door)．ためしに GPS(第1版) を動かしてみると，以下のように実行される．以下すべての実行において，カレントパッケージを:gpsにすることに注意されたい．

```
cl-user(9): (in-package :gps)
gps(10): (GPS1 '(at-door on-floor has-ball hungry chair-at-door)
               '(not-hungry)
               *banana-ops*)

(executing push-chair-from-door-to-middle-room)
(executing climb-on-chair)
(executing drop-ball)
(executing grasp-bananas)
(executing eat-bananas)
solved
```

猿は椅子をドアの場所から部屋の真ん中まで持っていき，椅子に登って，ボールを手から落とし，バナナをつかみ，それを食べる．これで猿の状態はnot-hungryになった．

GPS の動きをしっかりとつかむには，STRIPS 流の探索木(search tree)を書くとよい．探索木のトップのノードには [S0, G0] を置く．ここで S0 は初期状態，G0 はトップのゴールだけを含むリストである．もしゴールの要素がすべて状態に含まれていればよいが普通はそんなことはない．作用子の precond がすべて状態に含まれているものがあれば，即時にその作用子を実行できるが，そのようなものが無ければ実行したらゴールを満足するような作用子(関連する作用子)を見つけ出し，その precond を新たなゴールとして G0 のリストの先頭にプッシュしてそれを G1 とする．S0 は変わらない．図23.1 に Monkey & Banana の探索木

を示すが，ゴールとして与えられた not-hungry について即時に実行できる作用子はないのでとりあえず関連する作用子 eat-bananas の precond を追加したノードが図に示されている．

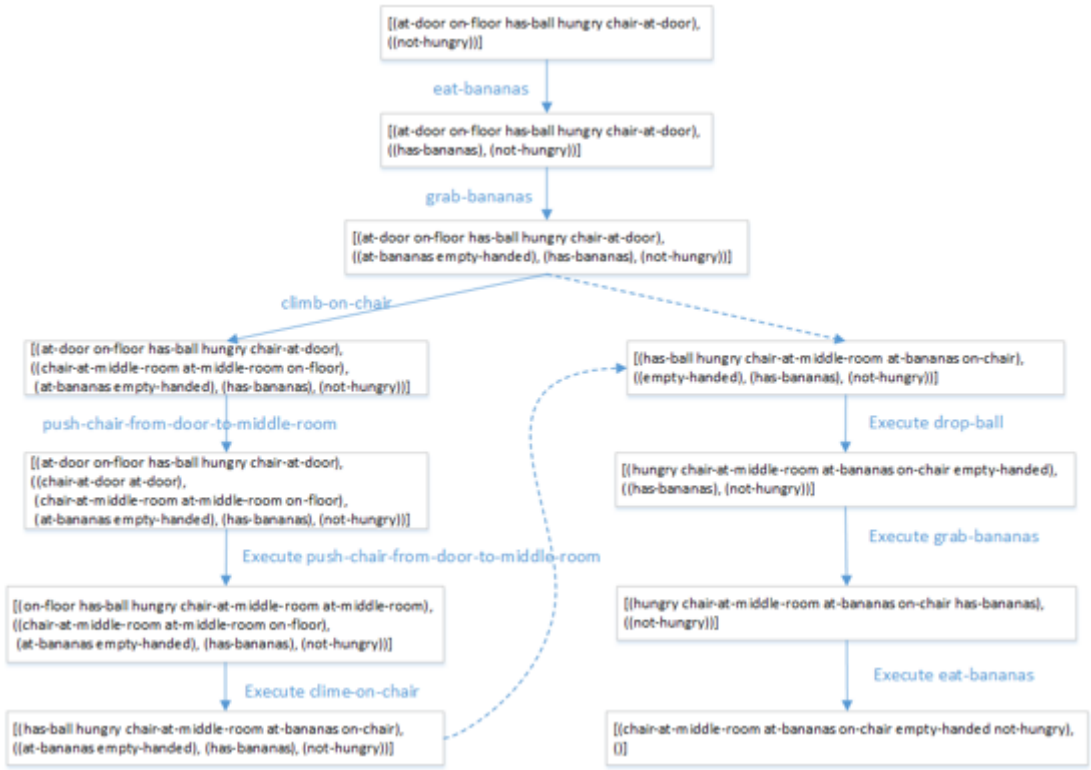


図23.1 Monkey & Banana 探索木

今度はその新たなゴール(サブゴール)を達成しようとする．今はそれは has-bananas である．同様にして，has-bananas が状態にあれば即作用子を実行できるが，なければまた関連する作用子を見つけ出し，その precond を再び新たなゴールとして G1 にプッシュする．それが G2 になる．同様なことを繰り返して push-chair-from-door-to-middle-room が関連する作用子となったときには，それを即時実行できる状態ということが判明し，それを実行する．ここで実行するとは，状態から del-list の要素を取り除き，add-list の要素を追加することである．それを状態 S1 とする．状態が変わったことで，以前は実行できなくてゴールとしたものが実行できるようになるかもしれない．そういう場合はそれを即時実行する．以後同様である．

最初に Norvig による GPS の簡単なバージョン(第1版)を示す．

```
(defvar *ops* nil "A list of available operators.")

(defun GPS1 (*state* goals &optional (*ops* *ops*))
  "General Problem Solver: achieve all goals using *ops*."
  (if (every #'achieve1 goals) 'solved)))
```

関数 GPS1 では複数のゴールをもらったら，それをすべて達成しようとする．すべてのゴールが達成されたときは，solved を返す．

```
(defun achieve1 (goal)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (or (member goal *state*)
      (some #'apply-op1
            (find-all goal *ops* :test #'appropriate-p1))))

(defun appropriate-p1 (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member goal (op-add-list op)))
```

achieve1 はゴール一つを受け取って，それが *state* にあれば達成済みとしてすぐに帰るが，なければそのゴール達成に直接に関連する作用子をすべて取り出して，そのどれか一つでも使ってゴールが達成できればよしとする．ゴール達成に直接関連する作用子とはゴールをすぐ達成するのに適切なもので，それはそのゴールが作用子の add-list に含まれるもののことである．

```
(defun apply-op1 (op)
  "Print a message and update *state* if op is applicable."
  (when (every #'achieve1 (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op)))
    (setf *state* (union *state* (op-add-list op)))
    t))
```

apply-op1 は受け取った作用子の preconds をサブゴールとして，そのすべてを達成しようとする．関数 GPS1では (every #'achieve1 goals) となっていたが，ここでは (every #'achieve1 (op-preconds op)) となっていることに注意されたい．達成できたときは executing ... とメッセージを印刷し，*state* から del-list の内容を削除し，add-list の内容を追加する．

23.4 世界をきっちりと記述しよう，宇宙ロボットによるORU交換

上記の Monkey & Banana は一見スマートに見えるかもしれないが，実はその手順は一通りしか可能性はなく，世界の記述も単純である．本当のところはあまりスマートとは言えない．猿は椅子に登ってボールを手から放したあと，バナナをつかむが，もしバナナが床に置いてあったら，もし猿がボールを持っていなかったら，もし猿がボールを投げてバナナを取ることができたら，どうしたらよいだろうか．作用子の数を増やしたり，より世界を豊富にしようとすると，計画を成功させるためには世界をより詳細に，作用子もより緻密に記述する必要がある．ボールを手から放したあと猿は手を伸ばしてバナナに触ってそれを握らなければならないが，それがこの例では grasp-banana の一言で記述されている．猿や人間では何がプリミティブな作業かが定かではないので，話を分かりやすくするために，ここでは宇宙ステーションにおける軌道上交換器(ORU)の交換を例に，宇宙ロボットの作業とタスクコマンドを解析する．

宇宙ステーションにおけるほとんどの機器は，簡単に着脱できるようになっており，故障した機器や中身を消費したタンクのような場合，該当の機器を同種のものと交換する．それが軌道上交換器 ORU である．国際宇宙ステーション ISS の船外実験プラットフォームでは宇宙飛行士の船外活動でORUを交換するよりも，ロボットアームでしかも可能であれば地上から遠隔操作で操作できることが望ましい．このORU交換のポイントは，宇宙ロボットアームは地上ロボットのような器用な手(dextrous hand)ではなく，機械的な着脱装置に助けられて作業を行うという点である．ハンドにはORUを把持したときに一体となるような着脱補助装置がついているし，ORUを本体装置に固定するには便利なラッチ機構がついている．着脱補助装置つきのハンドでORUを把持するには，宇宙船のドッキングの場合と同様に，ハンドを正しい位置(approach point)と姿勢(approaching posture)に持って行き，正しくORUに接触させる必要がある．そこでORU交換ロボットのプリミティブコマンドとして以下のようにする．今ORUは一個しかなく，ポート1にあるORUを空のポート2に装着する場合を考える．

1. approach HAND to ORU
   ロボットホームの場所からロボットハンドをORUのアプローチポイントまで持っていく．
2. align HAND to ORU
   アプローチポイントにおいてロボットハンドをORUのアプローチ姿勢にする．
3. attach HAND to ORU
   アプローチポイントとアプローチ姿勢にあるロボットハンドをORUに接近させて接触させる．
4. grasp ORU
   接触状態にあるORUを把持する．
5. unlatch ORU
   把持状態にあるORUのラッチをはずす．
6. detach ORU from PORT1
   把持状態かつラッチがはずれているORUをORUポートから離れた位置まで後退させる．
7. approach ORU to PORT2
   ORUをPORT2のアプローチポイントまで持っていく．引数がHANDとORUではないことに留意されたい．
8. align ORU to PORT2
   ORUをPORT2のアプローチ姿勢にする．
9. attach ORU to PORT2
   ORUをPORT2に接近させ接触させる．
10. latch ORU
    ORUをラッチする．
11. ungrasp ORU
    ラッチ状態にあるORUを手から放す．
12. detach HAND from ORU
    ロボットハンドをORUから離れた位置に移動させる．

初期状態として，ORUはラッチ状態(latched ORU)でポート1に置かれていて(on ORU PORT1)(aligned-to ORU PORT1)，ロボットハンドはホーム位置(at HAND HOME)(aligned-to HAND SELF)にあって手には何も持っていない(has NOTHING)とする．

上記ロボットコマンドを次のように定義する．

```
(defparameter *space-robot-ops*
  (list
   (make-op :action 'latch-ORU
            :preconds '((unlatched ORU) (has ORU))
            :add-list '((latched ORU))
            :del-list '((unlatched ORU)))
   (make-op :action 'unlatch-ORU
            :preconds '((latched ORU) (has ORU))
            :add-list '((unlatched ORU))
            :del-list '((latched ORU)))
   (make-op :action 'grasp-ORU
            :preconds '((has NOTHING) (on HAND ORU) (latched ORU) (aligned-to HAND ORU))
            :add-list '((has ORU))
            :del-list '((has NOTHING)))
   (make-op :action 'ungrasp-ORU
            :preconds '((has ORU) (latched ORU))
            :add-list '((has NOTHING))
            :del-list '((has ORU)))
   (make-op :action 'attach-HAND-to-ORU
            :preconds '((has NOTHING) (at HAND ORU) (aligned-to HAND ORU))
            :add-list '((on HAND ORU))
            :del-list '((at HAND ORU)))
   (make-op :action 'attach-ORU-to-PORT1
            :preconds '((has ORU) (at ORU PORT1))
            :add-list '((on ORU PORT1))
            :del-list '((at ORU PORT1)))
   (make-op :action 'attach-ORU-to-PORT2
            :preconds '((has ORU) (at ORU PORT2))
            :add-list '((on ORU PORT2))
            :del-list '((at ORU PORT2)))
   (make-op :action 'detach-HAND-from-ORU
            :preconds '((has NOTHING) (on HAND ORU))
            :add-list '((at HAND ORU))
            :del-list '((on HAND ORU)))
   (make-op :action 'detach-ORU-from-PORT1
            :preconds '((has ORU) (unlatched ORU) (on ORU PORT1))
            :add-list '((at ORU PORT1))
            :del-list '((on ORU PORT1)))
   (make-op :action 'detach-ORU-from-PORT2
            :preconds '((has ORU) (unlatched ORU) (on ORU PORT2))
            :add-list '((at ORU PORT2))
            :del-list '((on ORU PORT2)))
   (make-op :action 'approach-HAND-to-ORU-from-HOME
            :preconds '((has NOTHING) (at HAND HOME))
            :add-list '((at HAND ORU))
            :del-list '((at HAND HOME)))
   (make-op :action 'approach-HAND-to-HOME-from-ORU
            :preconds '((has NOTHING) (at HAND ORU))
            :add-list '((at HAND HOME))
            :del-list '((at HAND ORU)))
   (make-op :action 'approach-ORU-to-PORT2-from-PORT1
            :preconds '((has ORU) (at ORU PORT1) (unlatched ORU))
            :add-list '((at ORU PORT2))
            :del-list '((at ORU PORT1)))
   (make-op :action 'approach-ORU-to-PORT1-from-PORT2
            :preconds '((has ORU) (at ORU PORT2) (unlatched ORU))
            :add-list '((at ORU PORT1))
            :del-list '((at ORU PORT2)))
   (make-op :action 'align-HAND-to-ORU-from-SELF
```

```
            :preconds '((has NOTHING) (aligned-to HAND SELF))
            :add-list '((aligned-to HAND ORU))
            :del-list '((aligned-to HAND SELF)))
    (make-op :action 'align-HAND-to-SELF-from-ORU
            :preconds '((has NOTHING) (aligned-to HAND ORU))
            :add-list '((aligned-to HAND SELF))
            :del-list '((aligned-to HAND ORU)))
    (make-op :action 'align-ORU-to-PORT2-from-PORT1
            :preconds '((has ORU) (aligned-to ORU PORT1))
            :add-list '((aligned-to ORU PORT2))
            :del-list '((aligned-to ORU PORT1)))
    (make-op :action 'align-ORU-to-PORT1-from-PORT2
            :preconds '((has ORU) (aligned-to ORU PORT2))
            :add-list '((aligned-to ORU PORT1))
            :del-list '((aligned-to ORU PORT2)))
    ))
```

ここでは preconds や add-list, del-list にアトムではなくリストを使った．それに対応するように achieve1, appropriate-p1, apply-op1 を次のように変更する．

```
(defun achieve1 (goal)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (or (member goal *state* :test #'equal)
      (some #'apply-op1
            (find-all goal *ops* :test #'appropriate-p1))))

(defun appropriate-p1 (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member goal (op-add-list op) :test #'equal))

(defun apply-op1 (op)
  "Print a message and update *state* if op is applicable."
  (when (every #'achieve1 (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op) :test #'equal))
    (setf *state* (union *state* (op-add-list op) :test #'equal))
    t))
```

実際に上記のように変更してから，次のようにやってみよう．

```
gps(21): (GPS1 '((on ORU PORT1) (latched ORU) (has NOTHING)
                 (at HAND HOME) (aligned-to HAND SELF))
               '((unlatched ORU) (has ORU))
               *space-robot-ops*)
Error: Stack overflow (signal 1000)
[condition type: synchronous-operating-system-signal]
```

おやおや，スタックオーバーフローである．そこで次のように探索の経路をスタックして，何が起こっているのか見ることにする．

```
(defun GPS1 (*state* goals &optional (*ops* *ops*))
  "General Problem Solver: achieve all goals using *ops*."
  (when (every #'(lambda (g) (achieve1 g nil)) goals) 'solved))

(defun achieve1 (goal goal-stack)
  "A goal is achieved if it already holds,
```

```
  or if there is an appropriate op for it that is applicable."
  (cond ((member goal *state* :test #'equal))
        ((member goal goal-stack) (break))
        (t (some #'(lambda (op) (apply-op1 op (cons goal goal-stack)))
             (remove-if-not #'(lambda (op) (appropriate-p1 goal op)) *ops*)))))

(defun apply-op1 (op goal-stack)
  "Print a message and update *state* if op is applicable."
  (print (list 'applying (op-action op)))
  (when (every #'(lambda (g) (achieve1 g goal-stack)) (op-preconds op))
    (print (list 'executing (op-action op)))
    (setf *state* (set-difference *state* (op-del-list op) :test #'equal))
    (setf *state* (union *state* (op-add-list op) :test #'equal))
    t))
```

apply-op1 に入るたびにゴールがスタックにつまれていくが，計算中はこのスタックを各関数呼び出し時には持ちまわっている．これで実行すると次のように表示される．

```
gps(33): (GPS1 '((on ORU PORT1) (latched ORU) (has NOTHING)
                 (at HAND HOME) (aligned-to HAND SELF))
               '((unlatched ORU) (has ORU))
               *space-robot-ops*)

(applying unlatch-ORU)
(applying grasp-ORU)
(applying attach-HAND-to-ORU)
(applying detach-HAND-from-ORU)
(applying attach-HAND-to-ORU)
Break: call to the `break' function.
```

attach-HAND-to-OUR が繰り返し実行されてしまっている．GPSは最初に (unlatched ORU) を達成しようとするがそのためには (has ORU) でなければならず，grasp-ORU を達成しようとする．そのためには attach-HAND-to-ORU しなければならない．ところが，元々のプログラムでは grasp-ORU を達成しようとしたとしてもその情報はどこにも記録されないし，attach-HAND-to-ORU を達成しようとしたときにもその情報はどこにも記録されない．つまりシステムは自分の行動については何も記憶していない．本来であれば，探索の経路情報を何らかの形で記録しつつ，探索すべきである．あとの章では**時間的バックトラッキング**と**依存に基づくバックトラッキング**について説明するが，ここではとりあえず簡便な方法として，探索の繰り返しが起こったらその achieve の結果はその場で失敗ということにする．

```
(defun achieve1 (goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (cond ((member goal *state* :test #'equal))
        ((member goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op1 op (cons goal goal-stack)))
             (remove-if-not #'(lambda (op) (appropriate-p1 goal op)) *ops*)))))
```

これで計画できるか試してみよう．

```
gps(39): (GPS1 '((on ORU PORT1) (latched ORU) (has NOTHING)
                 (at HAND HOME) (aligned-to HAND SELF))
               '((unlatched ORU) (has ORU))
               *space-robot-ops*)
```

```
(applying unlatch-ORU)
(applying grasp-ORU)
(applying attach-HAND-to-ORU)
(applying detach-HAND-from-ORU)
(applying attach-HAND-to-ORU)
(applying approach-HAND-to-ORU-from-HOME)
(executing approach-HAND-to-ORU-from-HOME)
(applying align-HAND-to-ORU-from-SELF)
(executing align-HAND-to-ORU-from-SELF)
(executing attach-HAND-to-ORU)
(executing grasp-ORU)
(executing unlatch-ORU)
solved
```

いいらしい．"(applying ...)"のメッセージは作用子のpreconsが成立するまえに，成立検討段階で印刷されていることに注意しよう．肝心なのは"(executing ...)"メッセージである．ロボットはホーム位置からORU位置までアプローチして，ハンドの姿勢をORUに合わせて，ORUに接触させてから把持し，それからラッチをはずす．それではより本格的にポート 1 にある ORU を取り出してポート2に装着できるかやってみよう．

```
gps(40): (GPS1 '((on ORU PORT1) (latched ORU) (has NOTHING)
                 (at HAND HOME) (aligned-to HAND SELF))
               '((on ORU PORT2) (latched ORU) (has NOTHING)
                 (at HAND HOME) (aligned-to HAND SELF))
               *space-robot-ops*)

(applying attach-ORU-to-PORT2)
(applying grasp-ORU)
(applying attach-HAND-to-ORU)
(applying detach-HAND-from-ORU)
(applying attach-HAND-to-ORU)
(applying approach-HAND-to-ORU-from-HOME)
(executing approach-HAND-to-ORU-from-HOME)
(applying align-HAND-to-ORU-from-SELF)
(executing align-HAND-to-ORU-from-SELF)
(executing attach-HAND-to-ORU)
(executing grasp-ORU)
(applying detach-ORU-from-PORT2)
(applying unlatch-ORU)
(executing unlatch-ORU)
(applying attach-ORU-to-PORT2)
(applying approach-ORU-to-PORT2-from-PORT1)
(applying detach-ORU-from-PORT1)
(executing detach-ORU-from-PORT1)
(executing approach-ORU-to-PORT2-from-PORT1)
(executing attach-ORU-to-PORT2)
(applying latch-ORU)
(executing latch-ORU)
(applying ungrasp-ORU)
(executing ungrasp-ORU)
(applying approach-HAND-to-HOME-from-ORU)
(applying detach-HAND-from-ORU)
(executing detach-HAND-from-ORU)
(executing approach-HAND-to-HOME-from-ORU)
(applying align-HAND-to-SELF-from-ORU)
```

```
(executing align-HAND-to-SELF-from-ORU)
solved
```

実際にこれを自律的に実行するには，センサー類が必要になることは言うまでもない．手元に装着されたカメラ映像からハンドの位置や姿勢をロボットが自動的に修正できれば，地上からトップレベルのコマンドを発行することで ORU 交換が可能となる．

問題23.1 世界記述と作用子の記述が不十分だと，作用子のリストや preconds のリストの順序を変えただけで解が違ったり得られなくなることがある．宇宙ロボットでそれらの順番を変更してどんな挙動になるか観察せよ．

問題23.2 Monkey & Banana の例は簡単すぎる．この例では有効な作用子はいつも一つしかない．いろいろな発展が考えられる．もし天井からつるしたバナナと床にもバナナが置いてあったら，どうしたらよいだろうか．もし猿が手に持っているボールをバナナにあてて落として，それを食べることができるとするとどうだろうか．やってみればすぐ分かるが，世界記述と作用子をもっと詳細に記述しないとうまくいかない．自分で色々な状況を設定して，Monkey & Banana の定義を書き直し，きちんと計画できるかやってみよう．

問題23.3 天井からつるしたバナナよりも床にあるバナナを選択できるようにできるだろうか．また椅子の上に登ったほうがよい場合と，ボールを投げたほうがよい場合を区別できるだろうか．条件を追加して適切に猿が行動するようにせよ．

23.5 プログラムの改善（GPS 第2版）

PAIP における GPS 実行例では，親が子供を学校に車で送る例が示されている．Common Lisp システムを立ち上げ直したあと，githubからのプログラムを変更なしにもう一度ロードして，以下のように実行する．

```
cl-user(1): (in-package :gps)
#<The gps package>
gps(2): (GPS1 '(son-at-home car-works)
              '(son-at-school)
              *school-ops*)

(executing drive-son-to-school)
solved
```

自動車がちゃんと動く場合にはこれでよい．しかし，バッテリーの交換が必要な場合には，お店に電話をかけてバッテリーを持ってきてもらい交換しなければならない．

```
gps(3): (GPS1 '(son-at-home car-needs-battery have-money have-phone-book)
              '(son-at-school)
              *school-ops*)

(executing look-up-number)
(executing telephone-shop)
(executing tell-shop-problem)
(executing give-shop-money)
(executing shop-installs-battery)
(executing drive-son-to-school)
solved
```

ここで作用子 give-shop-money は次のようなものである.

```
   (make-op :action 'give-shop-money
            :preconds '(have-money)
            :add-list '(shop-has-money)
            :del-list '(have-money))
```

だから give-shop-money の行為によってお金は親の手元からお店の手に移るので，ゴール状態で親はお金を持っていないはずである(すなわち *state* には has-money はない）．ところが，次のようなゴールで GPS1 は誤った解を出してしまう.

```
gps(5): (GPS1 '(son-at-home car-needs-battery have-money have-phone-book)
              '(have-money son-at-school)
              *school-ops*)

(executing look-up-number)
(executing telephone-shop)
(executing tell-shop-problem)
(executing give-shop-money)
(executing shop-installs-battery)
(executing drive-son-to-school)
solved
```

単純な，あちらを立てればこちらが立たず問題

これに対応して改善したものが関数 GPS である.

```
gps(13): (GPS '(son-at-home car-needs-battery have-money have-phone-book)
              '(son-at-school)
              *school-ops*)
((start)
 (executing look-up-number)
 (executing telephone-shop)
 (executing tell-shop-problem)
 (executing give-shop-money)
 (executing shop-installs-battery)
 (executing drive-son-to-school))
gps(14): (GPS '(son-at-home car-needs-battery have-money have-phone-book)
              '(have-money son-at-school)
              *school-ops*)
nil
```

関数 GPS では，GPS1 と同様にすべてのゴール達成を行うとともに，その最後の状態 current-state で最初に与えられたゴールが満足されているかを確認する.

```
(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if #'atom (achieve-all (cons '(start) state) goals nil)))

(defun achieve-all (state goals goal-stack)
  "Achieve each goal, and make sure they still hold at the end."
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                        (setf current-state
                              (achieve current-state g goal-stack)))
```

```
                    goals)
                (subsetp goals current-state :test #'equal))
            current-state)))

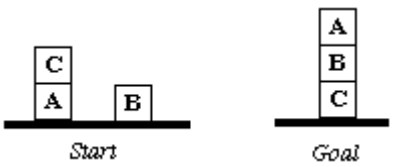
(defun achieve (state goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (dbg-indent :gps (length goal-stack) "Goal: ~a" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op state goal op goal-stack))
                 (find-all goal *ops* :test #'appropriate-p)))))

(defun member-equal (item list)
  (member item list :test #'equal))
```

この場合，バッテリーを購入することでお金がなくなってしまうわけだが，このような問題を**あちらを立てればこちらが立たず問題**(prerequisite clobbers sibling goal)と言うらしい．しかし上記例はまだ比較的単純な場合で，誰が考えても解がないことは容易に分かるし，同様に機械にも分かるが，単純でありながら人間なら簡単に解けるのに，機械が解くのが驚くほど難しい問題がある．

Sassman アノマリ

積み木の問題は人工知能の初期に，計画問題や自然言語対話など色々な分野で利用されてきた例題であるが，Gerald Sussman が博士論文を研究中に発見されたことから，次のような積み木問題をSassman Anomalyと呼ぶようになった．



初期状態では，積み木AとBはテーブルの上にあり `(A on table)`, `(B on table)`，積み木Cは積み木Aの上にある `(C on A)`．積み木Cと積み木Bの上には何もない `(space on C)`, `(space on B)`．ゴールは下からC，B，Aと積みあがった状態である `(C on table) (B on C) (A on B) (space on A)`．任意の積み木が置けるように，テーブルの上にはスペースがあるとする `(space on table)`.

さて，ロボットハンドが 1 個だけあるが，そのハンドでテーブルや上には何もない積み木を別の積み木やテーブルの上におく行為を考える．たとえば，(move A from B to C)という行為は次のような関数 move-op で生成され，その中の add-list と del-list も次のような関数 move-ons で生成されるとする．

```
(defun move-op (a b c)
  "Make an operator to move A from B to C."
  (op `(move ,a from ,b to ,c)
      :preconds `((space on ,a) (space on ,c) (,a on ,b))
      :add-list (move-ons a b c)
      :del-list (move-ons a c b)))
```

```
(defun move-ons (a b c)
  (if (eq b 'table)
      `((,a on ,c))
      `((,a on ,c) (space on ,b))))
```

これらの関数は make-block-ops に任意個の積み木を与えることで呼び出され，それを大域変数 *ops* にセットするのは関数 use である．

```
(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
      (dolist (b blocks)
        (unless (equal a b)
          (dolist (c blocks)
            (unless (or (equal c a) (equal c b))
              (push (move-op a b c) ops)))
          (push (move-op a 'table b) ops)
          (push (move-op a b 'table) ops))))
    ops))

(defun use (oplist)
  "Use oplist as the default list of operators."
  ;; Return something useful, but not too verbose:
  ;; the number of operators.
  (length (setf *ops* oplist)))
```

三つの積み木についてすべての行為を次のように生成してみる．

```
gps(15): (use (make-block-ops '(A B C)))
18
gps(16): *ops*
(#S(op :action (move C from B to table)
       :preconds ((space on C) (space on table) (C on B))
       :add-list ((executing (move C from B to table)) (C on table) (space on B))
       :del-list ((C on B)))
 #S(op :action (move C from table to B)
       :preconds ((space on C) (space on B) (C on table))
       :add-list ((executing (move C from table to B)) (C on B))
       :del-list ((C on table) (space on B)))
 #S(op :action (move C from B to A)
       :preconds ((space on C) (space on A) (C on B))
       :add-list ((executing (move C from B to A)) (C on A) (space on B))
       :del-list ((C on B) (space on A)))
 #S(op :action (move C from A to table)
       :preconds ((space on C) (space on table) (C on A))
       :add-list ((executing (move C from A to table)) (C on table) (space on A))
       :del-list ((C on A)))
 #S(op :action (move C from table to A)
       :preconds ((space on C) (space on A) (C on table))
       :add-list ((executing (move C from table to A)) (C on A))
       :del-list ((C on table) (space on A)))
 #S(op :action (move C from A to B)
       :preconds ((space on C) (space on B) (C on A))
       :add-list ((executing (move C from A to B)) (C on B) (space on A))
       :del-list ((C on A) (space on B)))
 #S(op :action (move B from C to table)
       :preconds ((space on B) (space on table) (B on C))
       :add-list ((executing (move B from C to table)) (B on table) (space on C))
```

```
      :del-list ((B on C)))
 #S(op :action (move B from table to C)
      :preconds ((space on B) (space on C) (B on table))
      :add-list ((executing (move B from table to C)) (B on C))
      :del-list ((B on table) (space on C)))
 #S(op :action (move B from C to A)
      :preconds ((space on B) (space on A) (B on C))
      :add-list ((executing (move B from C to A)) (B on A) (space on C))
      :del-list ((B on C) (space on A)))
 #S(op :action (move B from A to table)
      :preconds ((space on B) (space on table) (B on A))
      :add-list ((executing (move B from A to table)) (B on table) (space on A))
      :del-list ((B on A)))
 ...)
```

テーブル上に下から順に C，B，A と並んでいる積み木を逆順に並び替えるのは簡単である．

```
gps(17): (GPS '((A on B) (B on C) (C on table) (space on A) (space on table))
              '((B on A) (C on B)))
((start)
 (space on table)
 (executing (move A from B to table))
 (A on table)
 (executing (move B from C to A))
 (B on A)
 (space on C)
 (executing (move C from table to B))
 (C on B))
gps(18): (GPS* '((A on B) (B on C) (C on table) (space on A) (space on table))
               '((B on A) (C on B)))
((start)
 (executing (move A from B to table))
 (executing (move B from C to A))
 (executing (move C from table to B)))
```

このプログラムでは，convert-op によって，add-list に (executing 《行為》) が追加されるようになっている．GPS* は結果として返される値を (executing 《行為》) だけにするように修正されたものである．find-all-if はgpsaux.lispファイル中にあるライブラリであるが，その関数本体は remove-if-not に同じである．

```
(defun convert-op (op)
  "Make op conform to the (EXECUTING op) convention."
  (unless (some #'executing-p (op-add-list op))
    (push (list 'executing (op-action op)) (op-add-list op)))
  op)

(defun op (action &key preconds add-list del-list)
  "Make a new operator that obeys the (EXECUTING op) convention."
  (convert-op
    (make-op :action action :preconds preconds
             :add-list add-list :del-list del-list)))

(defun GPS* (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (find-all-if #'action-p
               (achieve-all (cons '(start) state) goals nil)))
```

```
(defun action-p (x)
  "Is x something that is (start) or (executing ...)?"
  (or (equal x '(start)) (executing-p x)))

(defun executing-p (x)
  "Is x of the form: (executing ...) ?"
  (starts-with x 'executing))

(defun starts-with (list x)
  "Is this a list whose first element is x?"
  (and (consp list) (eql (first list) x)))
```

Sassman アノマリでは容易には答えが得られない.

```
gps(19): (setq state '((C on A) (A on table) (B on table) (space on A) (space on B) (space on
table)))
((C on A) (A on table) (B on table) (space on A) (space on B) (space on table))
gps(20): (GPS* state '((A on B) (B on C)))
nil
gps(21): (GPS* state '((B on C) (A on B)))
nil
```

なぜこんなことになるかというと，`(A on B)` というゴールと `(B on C)` というゴールの間に干渉があるにもかかわらず，GPS はその二つのゴールを一つずつ達成しようとして，両方のことを同時に考慮できないからである．このようにゴール間に干渉のある計画問題のために半順序計画(Partial-Order Planning)アルゴリズムが開発されている.

23.6 GPS とはなんであったか？

GPS は人工知能の歴史の中で初期のものであり今日の目から見れば批判される点も多いが，そこにはその後の人工知能の色々な分野の要素が含まれていた．それをここでは，認知心理学，論理プログラミング，計画システムの観点から眺めてみる.

人間の問題解決行動

GPS オリジナル論文のタイトルは “GPS，人間の思考をシミュレートするプログラム” である．ファイゲンバウムとフェルドマンによるアンソロジー “コンピュータと思考” においても，この論文は第2部 “認知過程のシミュレーション” に収められている．ニューウェルとサイモンにとっては計画プログラムを作るのがGPS開発の目的であったのではない．人間の問題解決能力について観察し，問題領域によらない一般的な人間の知的な行動をシミュレーションすることによって，人間の知的能力の一端を明らかにしようとしたのである.

> 人間がするのと同じ仕事を機械におこなわせようとすることと，人間が実際に仕事をする過程をシミュレートしようとすることとは区別しなければならない，としばしば議論されている．この報告で述べるプログラム (GPS, General Problem Solver) は，この2つの接近法を最大限に混同させるものである．しかしそうすることによって相互に利益がえられる．以前には，GPS は問題解決プログラムを作る1つの試みとして述べられてきた（Newell, Shaw and Simon, 1959a,

> 1960a). [...] 同じ頃，GPS の変形されたプログラムによって，人間行動のシミュレーションが得られた(Newell and Simon, 1961a). ここでもっぱら注意を集中したいのは後者，すなわち GPS を人間の問題解決の理論として使うということである. [Newell and Simon (阿部，横山監訳）]

そのために彼らはこの論文の序文で，感覚と概念の関係に着目する連合主義，内観を重視するヴェルツブルク派，行動主義，ゲシュタルト心理学などについて触れてさえいる. そしてそれらの心理学の当時の現状に対して GPS を次のように位置づけた.

> 今日の心理学は，行動主義とゲシュタルト心理学という2つの極の間の比較的安定した張力均衡の状態にある. われわれは両者の主要な教訓をとり入れた. すなわち，われわれの実験における主観的な要素を懐疑的にとりあつかい，結局はすべての概念を行動上の尺度によってオペレーショナルにしなければならないという考えに同意する. また，われわれは，人間はきわめて複雑な組織化されたシステムであり，現代の行動主義心理学の簡単な体系は，これらすべてを反映するのがむずかしいように思われるということを認めている. [*ibid.*]

GPS によって開始された人間の知的行動のシミュレーションという立場は，その後，SOAR プロジェクトに引き継がれ，今ではACT-R プロジェクトに引き継がれたといえる. 興味を持たれたかたは SOAR ホームページからプログラムをダウンロードできるが，残念なことにそれは Java で書かれたものである. もちろん最初は Lisp であった. Lisp 版 SOAR はカーネギーメロン大学のリポジトリから入手可能である. また，ACT-R についてもそのホームページからLispプログラムをダウンロードできる.

論理プログラミングからの視点

GPS オリジナル論文で取り上げられた問題は命題論理の式を解く学生の行動である. 彼らは例題のプロトコル分析の結果を，手段・目的解析に照らして跡付けている. しかしこの論文からは彼らの論理プログラミングへの関心は見られない. 1981年発行のウィンストンの *LISP* にユニフィケーション(単一化，本書第29章参照)はないが，Common Lisp で書かれた第3版ではユニフィケーションがあることから類推すれば，1981年以前にはユニフィケーションというテクニックが一般的にはなっていなかったということが考えられる. AIMAにも見られるように，一時は人工知能技術の定式化に論理が全盛であった. アカデミックなにおいがするのと，厳密な議論ができることからそうなったのであろう. 著者自身は決して論理万能とは思っていないが，ユニフィケーションはElizaで示されたパターンマッチを論理推論の実現手段として仕上げたものである. GPS においても変数が扱えればと思うことはしばしばある. 本章で示した宇宙ロボットの例題では，ポート1とポート2についてそれぞれ似ているが厳密には異なる作用子を書かざるを得なかったが，変数を導入できればそのようなことはなくなる. 本書では第29章においてユニフィケーションを導入し，第31章のプラニングでは変数を可能としたプラニングシステムを紹介する.

計画プログラムからの視点

人工知能システムと一般のシステムとのプログラムコード上での明確な違いは，一般のシステムでは領域の知識がプログラムと一体になってプログラム中に埋め込まれているのに対して，人工知能システムでは必ず知識部分が手続き部分から独立分離されているということである. 人間の一般的な問題解決能力をシミュ

レートするという目的で開発された GPS は図らずもその人工知能の原則に先鞭をつけたことになる．以後，エキスパートシステムその他においてもこの原則は貫徹されている．最近の話題では，ジェパディで優勝した IBM のワトソンが医療診断分野にも応用できるというのも，原理的にはこの推論部と知識部の分離にその理由がある．

今では GPS は問題解決器としてよりも，計画プログラムとしての位置づけのほうが強い．最初の典型的な計画システムである STRIPS においても，その著者であるファイクスとニルソン自身が “STRIPS uses a GPS-like means-end analysis strategy” と述べているように，GPSは計画システムの先駆として大きな影響を与えた．STRIPS は論理証明の力を借りて GPS を新しく作り直したものと言えるが，今日ではこの方向での計画システムの定式化は，Ghallab, Nau, Traverso の “Automated Planning” に纏められている．本書第31章ではGhallab, Nau, Traverso の定式化による，Lisp の継続を利用したプラニングシステムを提示する．

第24章 推論と探索

> 推論の方向がうまくて目標の勝つ状態にいたることができればよいが，誤った方向に推論していく場合もある．推論の過程は本質的に試行錯誤プロセスであるといえる．ある段階まで推論していったとき，それが正しい方向の推論であるか，あるいは誤った方向のもので，そこで中断して別の方向を探るべきかどうかを決定することが，この種のもっとも重要で難しい問題である．
>
> [岩波講座 ソフトウエア科学〈〔知識〕14〉知識と推論，長尾 真，1988]

24.1 前向き推論と後向き推論

推論とは一般的には，前提と呼ばれる命題の集合から，別の命題を結論として導くこととされている．しかし一口に推論(inference あるいは reasoning)と言っても，演繹的推論(reduction)，帰納的推論(induction)，仮説推論(abduction)という推論分類から，推論対象や推論方法に着目した常識推論 (commonsense reasoning)，定性推論(qualitative reasoning)，アナロジーによる推論(anological reasoning)，事例ベース推論(case-based reasoning)など，多種多様な技術があり，その全体像を一口で述べるのは不可能に近い．それにもかかわらず，多くの推論方式において，モーダス・ポーネンスと呼ばれる最も基本的な規則が推論のベースとして用いられる．すなわち，事実としてある言明 A を真と認めることができ，同時に IF A THEN B という知識があったとき，B も真となると見做すのが，モーダス・ポーネンスである．IF ～ THEN ～ で表現された知識の確実性あるいは可能性について議論されることはあっても，モーダス・ポーネンスそのものの推論の妥当性について疑義を持つ人はいない．この節ではモーダス・ポーネンスを用いた二つの推論方式，すなわち，前向き推論と後向き推論について述べる．

推論知識として IF ～ THEN ～ という表現を利用するシステムは，ルールベース・システムとかプロダクション・システムと呼ばれる．ここで IF 部に書かれるものは前件部(前提条件)，THEN 部に書かれるものは後件部(帰結部)と呼ばれるが， IF A THEN B と IF B THEN C の二つのルールがあったとき，A が事実として分かれば，B が帰結し，さらに C を帰結とすることができる．このようにIF部からTHEN部に向かって複数のルールが働いて推論の連鎖が一歩づつ進む方式を**前向き推論**(forward reasoning)という．その逆に，Cを帰結として確証するためには IF B THEN C のルールからBが確証されればよいと考え，IF A THEN B のルールからAを調べるというように，THEN部からIF部に向かって推論が進む方式を**後向き推論**(backward reasoning)と呼ぶ．

前向き推論，後向き推論の例題問題として，簡単な動物当てクイズを取り上げる．GitHubからanimalをダウンロードしたのち，任意のCommon Lisp処理系で以下のように，実施してみてほしい．

まずはダウンロード先のディレクトリに移動してから，animal.lispというファイルをロードし，psパッケージの中に入る．

```
cl-user(7): :cd
C:\Users\誠二\
cl-user(8): :cd Documents/GitHub/animal/
```

```
C:\Users\誠二\Documents\GitHub\animal\
cl-user(9): (load "animal.lisp")
; Loading C:\Users\誠二\documents\GitHub\animal\animal.lisp
t
cl-user(10): (in-package :ps)
#<The ps package>
```

ここには次のような15個のルールがある.

```
ps(11): (pprint *rules*)

((rule id1 (if (animal has hair))
           (then (animal is mammal)))
 (rule id2 (if (animal gives milk))
           (then (animal is mammal)))
 (rule id3 (if (animal has feathers))
           (then (animal is bird)))
 (rule id4 (if (animal flies) (animal lays eggs))
           (then (animal is bird)))
 (rule id5 (if (animal eats meat))
           (then (animal is carnivore)))
 (rule id6 (if (animal has pointed teeth)
               (animal has claws)
               (animal has forward eyes))
           (then (animal is carnivore)))
 (rule id7 (if (animal is mammal) (animal has hoofs))
           (then (animal is ungulate)))
 (rule id8 (if (animal is mammal) (animal chews cud))
           (then (animal is ungulate) (even toed)))
 (rule id9 (if (animal is mammal)
               (animal is carnivore)
               (animal has tawny color)
               (animal has dark spots))
           (then (animal is cheetah)))
 (rule id10 (if (animal is mammal)
                (animal is carnivore)
                (animal has tawny color)
                (animal has black stripes))
            (then (animal is tiger)))
 (rule id11 (if (animal is ungulate)
                (animal has long neck)
                (animal has long legs)
                (animal has dark spots))
            (then (animal is giraffe)))
 (rule id12 (if (animal is ungulate) (animal has black stripes))
            (then (animal is zebra)))
 (rule id13 (if (animal is bird)
                (animal does not fly)
                (animal has long neck)
                (animal has long legs)
                (animal is black and white))
            (then (animal is ostrich)))
 (rule id14 (if (animal is bird)
                (animal does not fly)
                (animal swims)
                (animal is black and white))
            (then (animal is penguin)))
 (rule id15 (if (animal is bird) (animal flies well))
            (then (animal is albatross))))
```

たとえば，動物に毛皮があればそれは哺乳類であるとか，羽があればそれは鳥であるとかの知識が記述してある．一つのルールは次のような形式になっており，それから if-part-of や if-then-of でIF部やTHEN部を取りだすことができる．

```
      (rule <ID> (if {<IFデータ>}* )
                 (then {<thenデータ>}* ))
```

```
(defun if-part-of (rule)
  (cdaddr rule))

(defun then-part-of (rule)
  (cdr (cadddr rule)))

(defun id-of (rule)
  (cadr rule))
```

事実データとして，動物には体毛があり，鋭い歯があって，鋭い爪があって，目が真っすぐ前を向いていて，毛皮は茶色で黒い斑点があるということが分かっており，*facts* という大域変数にそれが記録されているとする．

```
ps(12): *facts*
nil
ps(13): (setq *facts*
      '((animal has hair)
        (animal has pointed teeth)
        (animal has claws)
        (animal has forward eyes)
        (animal has tawny color)
        (animal has dark spots)))
((animal has hair) (animal has pointed teeth) (animal has claws) (animal has forward eyes)
(animal has tawny color) (animal has dark spots))
```

すると，関数 deduce はこの事実データと推論ルールを使って，その動物は哺乳類であり，食肉獣であり，チータであるということを推論する．

```
ps(14): (deduce)

Rule id1 deduces (animal is mammal).
Rule id6 deduces (animal is carnivore).
Rule id9 deduces (animal is cheetah).
nil
```

前向き推論は事実から帰結に向かって推論する

前向き推論では，ルール集合に加えて**事実データが必要**であり，事実データとルールIF部とを照合し，マッチしたルールのTHEN部を帰結とする．たとえば，ルール1は「(if (animal has hair)) (then (animal is mammal))」というルールを持っており，そのIF部「(animal has hair)」が事実データである(animal has hair)とマッチするので，帰結として(animal is mammal)が導かれる．この帰結が得られることにより，今度はそれまでまったく関係が無かった(animal is mammal)をIF部に持つルール7，8，9，10が新たに推論に加わってくる．

関数 deduce は推論が有効に続く限り，関数 stepforward を実行し続ける．stepforward は，いつもすべてのルールについて順に tryrule で調べていき，真が返されたらすぐさま真を持って一旦帰るが，deduce の中で再び実行されることに注目されたい．ループの中でstepforwardが再実行されるときには，実は事実データは前の状態とはことなり，新たな帰結が加わっていることを注意しておく．こうしてすべての可能性が尽くされるまで，なんどでも stepforward が実行される．

```
(defun deduce ()
  "Top level command for forward-chaining."
  (loop while (stepforward)))

(defun stepforward ()
  "proceeds one step of inference from IF-part to THEN-part of rules."
  (loop for rule in *rules*
      when (tryrule rule)
      return t))

(defun tryrule (rule)
  "tests IF-part of <rule>> and establishes THEN-part with one
   step inference."
  (when (testif rule) (usethen rule)))
```

このようにIF部が成立してルールが成り立つことを，**ルールの発火**というが，それを調べるのは関数 testif である．この関数は引数として与えられたルールのIF部を取りだして，そのすべてが成り立つかどうかを recall で調べる．recall は調べたい事実を引数にとり，*facts* の中にマッチするものがあれば，その fact を返す．

```
(defun testif (rule)
  "returns true if every IF-part of <rule> is satisfied with *facts*."
  (every #'recall (if-part-of rule)))

(defun recall (fact)
  "returns <fact> if it is known as fact in *facts*."
  (when (member fact *facts* :test #'equal)
    fact))
```

一般的なプロダクション・システムでは，ルールが発火するときに実施されることは，帰結を新たなデータとして追加したり，何か具体的なアクションを実行したり，その実態はアプリケーションによって様々である．ここでは関数 usethen によって新たな帰結が記憶されるが，具体的にすることは関数 remenber を呼び出して，もし新たな知識だったらそれを *facts* に追加することである．

```
(defun usethen (rule)
  "establishes every THEN-part of <rule>. When a new one established,
   a message is printed."
  (let ((success nil))
    (loop for then in (then-part-of rule)
        when (remember then)
        do (format t "~%Rule ~S deduces ~S." (id-of rule) then)
          (setq success t))
    success))

(defun remember (new)
  "push <new> into *facts* if <new> is a new fact."
```

```
  (cond ((member new *facts* :test #'equal) nil)
        (t (push new *facts*)
           new)))
```

後向き推論では仮説から事実に向かって推論する

後向き推論では最初に確証したい帰結が仮説集合として定義済みである．この例では15個のルールに合わせて，７匹の動物が仮説として設定されている．

```
(defparameter *hypotheses*
  '((animal is albatross)
    (animal is penguin)
    (animal is ostrich)
    (animal is zebra)
    (animal is giraffe)
    (animal is tiger)
    (animal is cheetah))
  "This includes several animal hypotheses.")
```

大域変数 *asked* は，システムがユーザに同じ質問をしないようにするためのものであり，後述の verify 関数の中で，質問の回答が「Yes」であれば，その質問内容は真として *facts* に入れられるが，「No」であれば *asked* に入れられる．関数 diagnose は後向き推論におけるトップの関数であり，その中では仮説の一つ一つについて関数 verify でもって推論を行うが，仮説集合のうちの一つでも確証が得られれば，それでよしとする．

```
(defvar *asked* nil "list of asked assertions.")

(defun diagnose ()
  "Top level command for backward-chaining inference."
  (setq *asked* nil)
  (let ((verified (some #'verify *hypotheses*)))
    (when verified
      (format t "~%Hypothesis ~S is true." verified)
      verified)))
```

今，チータが目の前にいるとして，たとえば以下のように実行してみよう．たまたま食事時にぶつかれば，(animal eats meat) にYesと答えることができるが，ここではそれにたいしてはNoと答えている．同様に，寝ているだけの姿からはそれがするどい爪(claws)を持っているかどうかは分かりにくいが，ここではYesと答えている．

```
cl-user(3): (load "animal.lisp")
; Loading C:\Users\誠二\Documents\GitHub\animal\animal.lisp
t
cl-user(4): (in-package :ps)
#<The ps package>
ps(5): (diagnose)

Is this true: (animal has feathers)? n

Is this true: (animal flies)? n

Is this true: (animal has hair)? y
```

```
Rule id1 deduces (animal is mammal).

Is this true: (animal has hoofs)? n

Is this true: (animal chews cud)? n

Is this true: (animal eats meat)? n

Is this true: (animal has pointed teeth)? y

Is this true: (animal has claws)? y

Is this true: (animal has forward eyes)? y

Rule id6 deduces (animal is carnivore).

Is this true: (animal has tawny color)? y

Is this true: (animal has black stripes)? n

Is this true: (animal has dark spots)? y

Rule id9 deduces (animal is cheetah).
Hypothesis (animal is cheetah) is true.
(animal is cheetah)
```

さて，関数 verify はその引数に事実候補あるいは仮説を引き受けるが，1) もしその引数が事実として知られているのなら(recall)，そのままその事実を返す．さもなければ IF ～ THEN ～ ルールを用いた推論を行う．すなわち，2) 引数である仮説を THEN部に持つようなルールを集めてくる(inthen)けれども，引数が質問済み事実であれば，nil を持ってすぐ帰るが，そうでなければ，3) 最初に再帰なしの tryrule でルールを調べて，ルールが成立すればそこで成立した仮説を返す．さもなければ 4) tryrule を用いて，さらに推論を行う．

```
(defun verify (fact)
  "verifies <fact> with backward inference in the following steps.
1) if <fact> is directly known as fact, then returns true.
2) if no rules relevant to <fact>, question on <fact> is made and asked to users.
   Then, yes/no answer establishes <fact> or not.
3) first, relevant rules are applied to try establish <fact> in one step inference.
   If it is so, immediately returns with established <fact>.
4) otherwise, relevant rules are applied to try establish <fact> in multi-step inference.
   Note that invoking as verify -> tryrule+ -> testif+ -> verify with recursion"
  (if (recall fact) fact
    (let* ((relevant1 (inthen fact))
           (relevant2 relevant1))
      (unless relevant1
        (cond ((member fact *asked* :test #'equal) (return-from verify nil))
              ((y-or-n-p "~%Is this true: ~S? " fact)
               (remember fact)
               (return-from verify fact))
              (t (push fact *asked*)
                 (return-from verify nil))))
      (loop for rel in relevant1
          when (tryrule rel)
          do (return-from verify fact))
      (loop for rel in relevant2
```

```
          when (tryrule+ rel)
          do (return-from verify fact))
       nil)))

(defun inthen (fact)
  "collects rules whose THEN-part is relevant to <fact>."
  (remove-if-not #'(lambda (r) (then-p fact r)) *rules*))

(defun then-p (fact rule)
  "returns true if <fact> is in THEN-part of <rule>."
  (member fact (then-part-of rule) :test #'equal))
```

関数 tryrule+ は tryrule とよく似ているが，その中では testif ではなく，testif+ を呼び出す．関数 testif+ はその引数で与えられたルールの IF部を取りだして，それを新たな仮説として verify で再帰的に調べに入る．

```
(defun tryrule+ (rule)
  "tests IF-part of <rule> and establishes THEN-part with multi-step inference."
  (when (testif+ rule) (usethen rule)))

(defun testif+ (rule)
  "returns true if every IF-part of <rule> is verified as truth with inference by rules."
  (every #'verify (if-part-of rule)))
```

前向き推論と後向き推論の特徴を挙げるとつぎのとおりである．

- 前向き推論は事実データから出発する．推論が到達すべきゴールは知らなくてもよい．したがって，データ駆動型と言ってもよい．
- 後向き推論では仮説（ゴール）が分かっていなければならない．前向き推論と異なり，ゴール駆動型と言ってもよい．
- 前向き推論はバッチ処理的に処理するのに向いているが，後向き推論は対話的処理を導入するのに向いている．

故障診断のように，あらかじめゴールの候補を列挙できるような問題には，後向き推論が使えるが，設計問題のように，ゴールそのものが解であるような問題には使えない．前向き推論では，派生的に多くの帰結が得られて推論が発散していく可能性が否定できないが，後向き推論は目標指向的であって，必要とするメモリ空間も広がることはない．

問題 24.1 チータ以外の動物用に事実データを用意して，前向き推論を実行せよ．

問題 24.2 チータ以外の動物を想定して，後向き推論を実行せよ．

問題 24.3 七ひきの動物に加えて，他の動物を想定して，ルールと仮説を修正し，推論を実行せよ．うまく動物を識別できたであろうか？

24.2 AND/ORグラフと推論

IF ～ THEN ～ ルールのIF部が複数あったとき，それらはすべて満たされなければならない．つまりそれらはAND結合である．一方，同じ帰結を含むルールが複数あったとき，それらのいずれかが成立すればよい．つまりそれらはOR結合である．したがって，ルールベースシステムにおいて，事実や仮説をノードとする

AND/ORグラフをルール集合から作ることができる．動物のルールはつぎのようなAND/ORグラフになる．ここで矢印は IF ～ THEN ～ の方向を示し，○の結合点はAND結合とOR結合を示している．

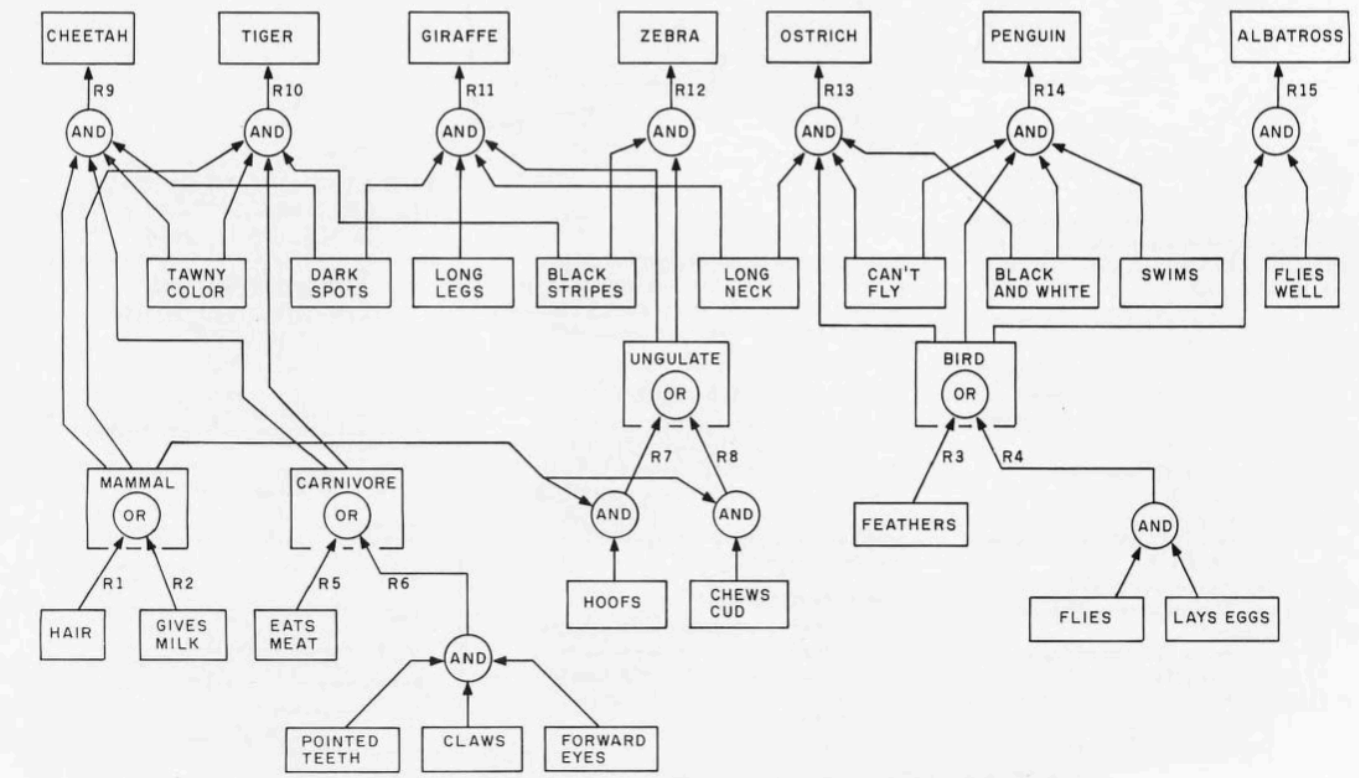


Figure 5: *A sample network for a simple rule-based system. Boxes represent assertions, circles represent ways of combining assertions, and the labels R1, R2, etc, identify rules. The diagram corresponds to the rules given in lines 3880 through 4080 of listing 1.*

そこで，今チータに注目して，チータが成立するための条件をAND結合とOR結合によりリスト表記すると以下のようになる．ただし，これらの並び順は，ルール集合におけるルールの並び順とIF部の並び順を反映したものである．

```
(and (or (animal has hair) (animal gives milk)) ; instead of mammal
     (or (animal eats meat)                     ; these lines are
         (and (animal has pointed teeth)        ; instead of
              (animal has claws)                ; carnivore
              (animal has forward eyes)))       ;
     (animal has tawny color)
     (animal has dark spots))
```

この条件式はチータに関するルールのIF部を集めたものであるが，ルールをコンパイルしたものに相当する．

問題 24.4 上記条件式を持つような関数 Is-this-cheetah? を定義せよ．そしてそれが働くようにするにはどうしたらよいか，考えよ．

問題 24.5 動物のクイズを対象とした，ルールコンパイラを開発せよ．

24.3 米国飛行機旅行経路探索

自動車のナビのような経路探索問題は探索の典型的な例題である．今では自動車のナビはコモディティ化されてAIでも何でもなくなってしまったが，PAIP では探索の例題として米国の都市を航空機で旅行する課題が取り上げられた．PAIPでは全米で22都市が緯度経度つきでデータ化されているが，その緯度経度は都市の中心部を指しているようだ．ここでは都市の代表的な空港の位置をウィキペディアで調べて，以下のようなデータに修正した．

```
(defparameter *cities*
  '((Atlanta      ATL 33.636667  -84.428056) (Los-Angeles   LAX 33.9425   -118.408056)
    (Boston       BOS 42.363056  -71.006389) (Memphis       MEM 35.0425    -89.976667)
    (Chicago      ORD 41.978611  -87.904722) (New-York      LGA 40.77725   -73.872611)
    (Denver       DEN 39.861656 -104.673178) (Oklahoma-City OKC 35.393056  -97.600833)
    (Eugene       EUG 44.123056 -123.218611) (Pittsburgh    PIT 40.491389  -80.232778)
    (Flagstaff    FLG 35.140278 -111.669167) (Quebec        YQB 46.791111  -71.393333)
    (Grand-Jct    GJT 39.1225   -108.526667) (Reno          RNO 39.499167 -119.768056)
    (Houston      IAH 29.984444  -95.341389) (San-Francisco SFO 37.618889 -122.375)
    (Indianapolis IND 39.717222  -86.294444) (Tampa         TPA 27.975556  -82.533333)
    (Jacksonville JAX 30.494167  -81.687778) (Victoria      YYJ 48.647222 -123.425833)
    (Kansas-City  MCI 39.2975    -94.713889) (Wilmington    ILM 34.270556  -77.9025)))
```

PAIPの記述では緯度経度の列挙の順番が今日の常識とは異なっていたり，PAIPでは記載がなかった空港の略称（IATA)を追加したので，ここでの構造体 city の定義は次のようになる．

```
(defstruct (city (:type list)) name airport lat long)
```

この構造体の定義は，上記のようなリスト構造から任意の都市の緯度経度情報を取りだすためのものであり，関数cityをこのように定義して，都市のデータからcity-latで緯度をcity-longで経度を得ることができる．一般にはリストの等価性を調べるにはequalでなければならないが，このcityを使う限りにおいては assoc の結果を使うので eql や eq でもよいことを指摘しておく．

```
(defun city (name)
  "Find the city with this name."
  (assoc name *cities*))
```

```
cl-user(7): (city-airport (city 'New-York))
LGA
cl-user(8): (city-long (city 'New-York))
-73.87261
```

地球上の2点間の大円距離は，2点の緯度経度が分かれば計算できる．

```
(defconstant earth-diameter 12742.0
  "Diameter of planet earth in kilometers.")

(defun air-distance (city1 city2)
  "The great circle distance between two cities."
  (let ((d (distance (xyz-coords city1) (xyz-coords city2))))
    ;; d is the straight-line chord between the two cities,
    ;; The length of the subtending arc is given by:
    (* earth-diameter (asin (/ d 2)))))

(defun xyz-coords (city)
  "Returns the x,y,z coordinates of a point on a sphere.
  The center is (0 0 0) and the north pole is (0 0 1)."
```

```
  (let ((psi (deg->radians (city-lat city)))
        (phi (deg->radians (city-long city))))
    (list (* (cos psi) (cos phi))
          (* (cos psi) (sin phi))
          (sin psi))))

(defun distance (point1 point2)
  "The Euclidean distance between two points.
  The points are coordinates in n-dimensional space."
  (sqrt (reduce #'+ (mapcar #'(lambda (a b) (expt (- a b) 2))
                            point1 point2))))

(defun deg->radians (deg)
  "Convert degrees to radians."
  (* deg pi 1/180))
```

PAIPオリジナルデータでは緯度経度は <度>.<分> のデータであったが，これは十進データと紛らわしい．上記の緯度経度データはすべて度の十進データであるので，関数 deg->radians はPAIPオリジナルのコードとは異なることを注意しておく．また，厳密には地球は扁平であるが，平均して直径は12,742kmほどである．PAIPでは12,765kmになっているが，この数値がどこから出てきているのか不明である．ここでは通例用いられる12,742kmに従った．

```
cl-user(27): (* pi earth-diameter)
40030.173592041145d0
cl-user(28): (air-distance (city 'San-Francisco) (city 'New-York))
4141.0044079325d0
```

ここで，ある都市から1000km以内にある都市をリストアップする関数 neighbors と現在地点から1000km以内の都市かどうかを調べる関数 reachable-p を次のように定義しておく．

```
(defun reachable-p (city1 city2)
  (and (not (eq city1 city2))
       (< (air-distance city1 city2) 1000.0)))

(defun neighbors (city)
  "Find all cities within 1000 kilometers."
  (remove-if-not #'(lambda (c) (reachable-p c city))
                 *cities*))
```

この関数を使って，例えば，ニューヨーク近傍の都市は以下の四つがあると分かる．1000km以上の距離を飛ぶことができない小型飛行機の場合には，ニューヨーク・ラガーディア空港から飛び立つと，次に到着可能な空港はこの四つに絞られる．

```
cl-user(31): (neighbors (city 'New-York))
((Boston BOS 42.363056 -71.006386) (Pittsburgh PIT 40.49139 -80.23278)
 (Quebec YQB 46.79111 -71.39333) (Wilmington ILM 34.270557 -77.9025))
```

逆にボストンからはニューヨークも近傍だが，ウィルミントンはそうではない．

```
cl-user(32): (neighbors (city 'Boston))
((New-York LGA 40.77725 -73.87261) (Pittsburgh PIT 40.49139 -80.23278)
 (Quebec YQB 46.79111 -71.39333))
```

さて，ボストン・ローガン空港から，サンフランシスコ国際空港までの，小型飛行機による旅行を考えてみよう．ボストンから1000km以内の空港は，ニューヨーク・ラガーディア空港，ピッツバーグ国際空港，ケベック・ジャン・ルサージ国際空港の三つだが，サンフランシスコに行くのにそれらのうちのどこに飛ぶのがよいのだろうか．神の目ならぬ状況であればここで探索が必要になるが，探索にも可能な解をすべて計算しようとして，１個でも得られればよしとするもの，コストを考慮して効率の良い解を計算するもの，最適解ではないかも知れないが探索空間を抑えてそこそこの解が得られればよいとするものなど，色々なものがある．

深さ優先探索

深さ優先探索は，現在地点から到達可能地点のうちの一つを選び，それを現在地点にしてさらに目的地まで再帰的に探索する手法である．多くの場合，解が１個でも得られればそれでよしとする．再帰的に探索するので，探索の前線は一度足を踏み出した方向に（めくらめっぽうであっても）どんどん進んでいくので深さ優先と言われる．以下の深さ優先探索コードでは，graph-search の combiner に append が与えられていることに注目されたい．states は city 実現体のリストであるが，それは常に探索の前線を表現している．そして combiner が実際に働くところでは，常に states の先頭がその展開された結果と置き換えられる．

```
(defconstant fail nil)

(defun isa (value &key (key #'identity) (test #'eql))
  "Return a predicate that tests for a given value."
  #'(lambda (x) (funcall test value (funcall key x))))

(defun trip1 (start dest)
  (depth-first-search start (isa dest) #'neighbors))

(defun depth-first-search (start goal-p successors)
  "Search new states first until goal is reached."
  (graph-search (list start) goal-p successors #'append))

(defun graph-search (states goal-p successors combiner &optional (state= #'eql) trail)
  "Find a state that satisfies goal-p.  Start with states,
  and search according to successors and combiner.
  Don't try the same state twice."
  (dbg :search "~&;; Search: ~a" states)
  (cond ((null states) fail)
        ((find-if goal-p states)
         (reverse (cons (find-if goal-p states) trail)))
        (t (let ((new-states
                  (funcall combiner
                           (new-states (first states) states successors state= trail)
                           (rest states))))
             (graph-search (remove-if-not #'(lambda (c) (reachable-p c (first states)))
                                          new-states)
                           goal-p successors combiner state=
                           (cons (first states) trail))))))

(defun new-states (first states successors state= trail)
  "Generate successor states that have not been seen before."
  (remove-if
   #'(lambda (state)
```

```
        (or (member state states :test state=)  ; remove self
            (member state trail :test state=))) ; avoid revisit
    (funcall successors first)))
```

このコードでは，graph-searchはそれまでの探索の軌跡 trail を記録する．そして successors (実態は関数 neighbors) による展開結果中にすでに訪れた地点があればそれを削除するし，states と重複する地点があればそれも削除する．そうしないと経路がループしてしまうためである．関数 dbg はプログラムにデバッグ機能を提供するもので，そのコードは 21.2節の debug を参照されたい．

ボストンからサンフランシスコまでの探索経路は以下のようになる．

```
cl-user(5): (debug-id :search)
(:search)
cl-user(6): (trip1 (city 'Boston) (city 'San-Francisco))
;; Search: ((Boston BOS 42.363056 -71.006386))
;; Search: ((New-York LGA 40.77725 -73.87261) (Pittsburgh PIT 40.49139 -80.23278)
            (Quebec YQB 46.79111 -71.39333))
;; Search: ((Wilmington ILM 34.270557 -77.9025) (Pittsburgh PIT 40.49139 -80.23278)
            (Quebec YQB 46.79111 -71.39333))
;; Search: ((Atlanta ATL 33.636665 -84.428055) (Indianapolis IND 39.71722 -86.29444)
            (Tampa TPA 27.975555 -82.53333) (Jacksonville JAX 30.494167 -81.687775)
            (Pittsburgh PIT 40.49139 -80.23278))
;; Search: ((Memphis MEM 35.0425 -89.97667) (Chicago ORD 41.97861 -87.904724)
            (Indianapolis IND 39.71722 -86.29444) (Tampa TPA 27.975555 -82.53333)
            (Jacksonville JAX 30.494167 -81.687775) (Pittsburgh PIT 40.49139 -80.23278))
;; Search: ((Oklahoma-City OKC 35.393055 -97.60083) (Houston IAH 29.984444 -95.34139)
            (Kansas-City MCI 39.2975 -94.71389) (Chicago ORD 41.97861 -87.904724)
            (Indianapolis IND 39.71722 -86.29444) (Jacksonville JAX 30.494167 -81.687775))
;; Search: ((Denver DEN 39.861656 -104.67318) (Houston IAH 29.984444 -95.34139)
            (Kansas-City MCI 39.2975 -94.71389))
;; Search: ((Flagstaff FLG 35.140278 -111.66917) (Grand-Jct GJT 39.1225 -108.526665)
            (Kansas-City MCI 39.2975 -94.71389))
;; Search: ((Los-Angeles LAX 33.9425 -118.40806) (Reno RNO 39.49917 -119.76806)
            (San-Francisco SFO 37.61889 -122.375) (Grand-Jct GJT 39.1225 -108.526665))
((Boston BOS 42.363056 -71.006386) (New-York LGA 40.77725 -73.87261)
 (Wilmington ILM 34.270557 -77.9025) (Atlanta ATL 33.636665 -84.428055)
 (Memphis MEM 35.0425 -89.97667) (Oklahoma-City OKC 35.393055 -97.60083)
 (Denver DEN 39.861656 -104.67318) (Flagstaff FLG 35.140278 -111.66917)
 (San-Francisco SFO 37.61889 -122.375))
```

結果の経路はボストン，ニューヨーク，ウィルミントン，アトランタ，メンフィス，オクラホマシティ，デンバー，フラグスタッフ，サンフランシスコである．このルートを図24.1に示す．

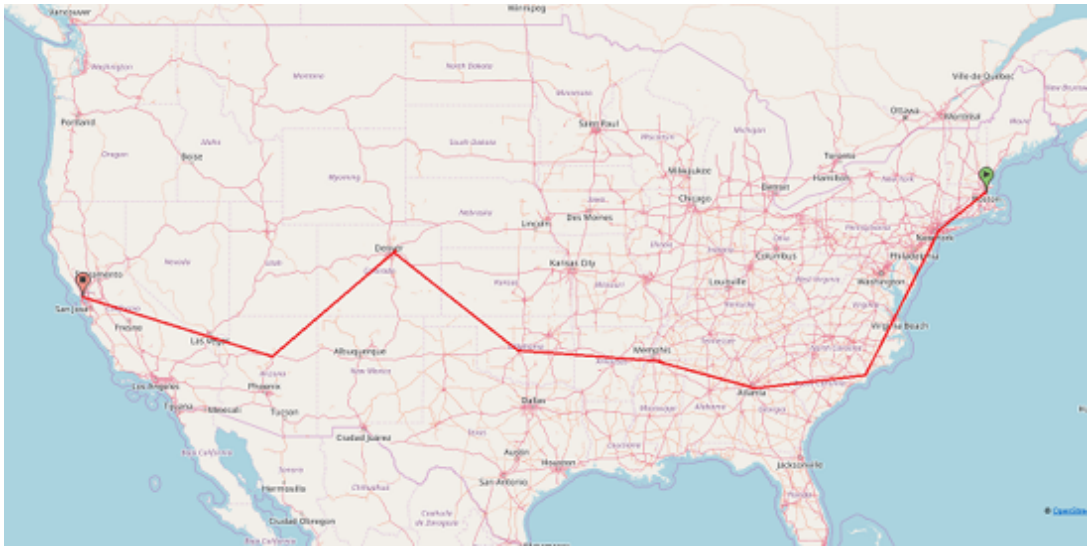


図24.1　深さ優先探索による解の例

ボストンからピッツバーグは到達可能であるにもかかわらず，ゴールからは遠いニューヨークに行っている．また，ニューヨークからピッツバーグに行く方がゴールに近いにもかかわらず，わざわざ遠く離れたウィルミントンに行っている．これらの理由はどちらが目的地に近いとかの情報を探索で利用していないからである．あとで最良優先探索でこのような問題の解決を行う．

問題24.6 上記コードは，常に関数 new-states の結果の先頭が展開される．new-states の結果を reverse したものを combiner に与えることにしたらどうなるであろうか．

幅優先探索

幅優先探索では，深さ優先探索においては (first states) を展開した結果を (rest states) の先頭においたところを，先頭ではなく最後に置くだけである．append のかわりに prepend が combiner として与えられていることに注意されたい．これで展開結果を追求するのを後回しにして，まだ展開されていない地点を優先的に展開することになる．

```
(defun prepend (x y)
  "Prepend y to start of x"
  (append y x))

(defun breadth-first-search (start goal-p successors)
  "Search old states first until goal is reached."
  (graph-search (list start) goal-p successors #'prepend nil))

(defun trip2 (start dest)
  (breadth-first-search start (isa dest) #'neighbors))
```

```
cl-user(7): (debug-id :search)
(:search)
cl-user(8): (trip2 (city 'Boston) (city 'San-Francisco))
;; Search: ((Boston BOS 42.363056 -71.006386))
;; Search: ((New-York LGA 40.77725 -73.87261) (Pittsburgh PIT 40.49139 -80.23278)
```

```
            (Quebec YQB 46.79111 -71.39333))
;; Search: ((Pittsburgh PIT 40.49139 -80.23278) (Quebec YQB 46.79111 -71.39333)
            (Wilmington ILM 34.270557 -77.9025))
;; Search: ((Quebec YQB 46.79111 -71.39333) (Wilmington ILM 34.270557 -77.9025)
            (Atlanta ATL 33.636665 -84.428055) (Chicago ORD 41.97861 -87.904724)
            (Indianapolis IND 39.71722 -86.29444))
;; Search: nil
nil
```

上記実施例では，ニューヨークに行ってからピッツバーグに行き，ビッツバーグからアトランタでもインディアナポリスでも行けるにも関わらず，ケベックに行ってしまったため，答えが得られなかった．この経路を図24.2に示す．答えが得られなかった理由は，ケベックを展開するとボストン，ニューヨーク，ピッツバーグだが，これらはいずれも既に訪問済みであるため，探索候補からは削除されてしまうからである．

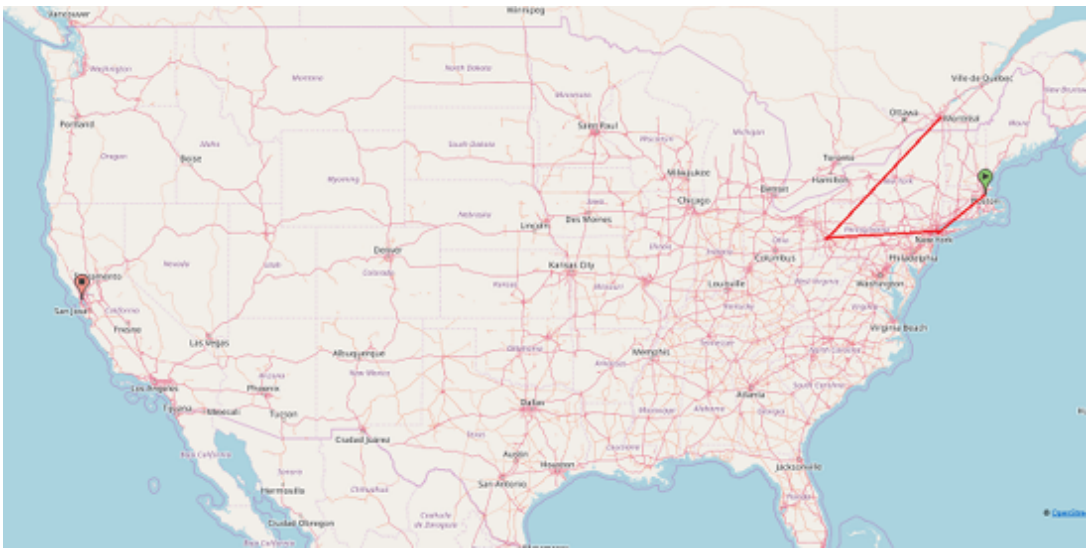


図24.2　幅優先探索の例

問題24.7 前の問題と同様に，もし関数 new-states の reverse したものを combiner に与えたらどうなるであろうか．

いつも必ず幅優先探索よりも深さ優先探索の方がよい，ということではない．問題によっては幅優先探索の方がたまたま早くゴールに到達する場合もある．ただし，対話型のような場合には，幅優先探索ではシステムからの質問が脈絡なく飛ぶので，ユーザから見てシステムの振る舞いが分かりにくくなるということは言える．実は24.1節における動物の後向き推論は，常にルールの IF 部を取りだして，それについて再帰的にverifyするので深さ優先探索になっている．

本来ならば，幅優先探索はもしゴールがあるのなら，理論的にはそれが必ず求まることが分かっている．なぜならば，通常は出発点（ルートの節点）からの第１レベルがすべて展開されて探索されるし，第１レベルの節点もすべて展開されて第２レベルの節点となって探索されるし，以下同様だからである．その代わりに，探索に必要なメモリ空間は急速に指数関数的に大きくなる．ただしこの例題では展開されて得られる節点は1000kmまでの場所なので，探索空間が広がりすぎるということはないが，上記のように再訪しないこと

にすると場合によっては解が求まらないこともある．一方，深さ優先探索では探索に必要なメモリ空間は小さく抑えられるが，問題によっては深さが無限に近く展開されると，実用的な意味ではゴールにたどり着かないという場合もある．

最良優先探索

これまでの探索方法ではその領域の知識を用いてこなかった．ドメインの知識を用いれば，より効率のよい探索ができる．その最も簡単な方法として，最良優先探索を紹介する．最良というのが何であるかは，問題に依存する．次の展開すべき候補を選び出すときに，なるべく早くゴールに到達する可能性の高いものを選びたい．この例題では探索途中地点からゴールへの直線距離が一番短いものを選ぶことにする．だから正確に言えば，最良**予測**優先探索であるが，通常これを最良優先探索と呼んでいる．評価関数は，ある都市からゴールまでの直線距離であるとすると，探索は評価関数（実はコスト関数）の小さいものを優先的に探索すればよいことになる．

```
(defun distance-to-goal (goal)
  #'(lambda (c) (air-distance c goal)))

(defun sorter (cost-fn)
  "Return a combiner function that sorts according to cost-fn."
  #'(lambda (new old)
      (sort (append new old) #'< :key cost-fn)))

(defun best-first-search (start goal-p successors cost-fn)
  "Search lowest cost states first until goal is reached."
  (graph-search (list start) goal-p successors (sorter cost-fn)))

(defun trip3 (start dest)
  (best-first-search start (isa dest) #'neighbors (distance-to-goal dest)))
```

このコードでは combiner として，(sorter cost-fn) が与えられている．cost-fn は (distance-to-goal dest) である．すると次のように実行できる．

```
cl-user(13): (trip3 (city 'Boston) (city 'San-Francisco))
;; Search: ((Boston BOS 42.363056 -71.006386))
;; Search: ((Pittsburgh PIT 40.49139 -80.23278) (New-York LGA 40.77725 -73.87261)
            (Quebec YQB 46.79111 -71.39333))
;; Search: ((Chicago ORD 41.97861 -87.904724) (Indianapolis IND 39.71722 -86.29444)
            (Atlanta ATL 33.636665 -84.428055) (Wilmington ILM 34.270557 -77.9025)
            (New-York LGA 40.77725 -73.87261) (Quebec YQB 46.79111 -71.39333))
;; Search: ((Kansas-City MCI 39.2975 -94.71389) (Memphis MEM 35.0425 -89.97667)
            (Indianapolis IND 39.71722 -86.29444) (Atlanta ATL 33.636665 -84.428055))
;; Search: ((Denver DEN 39.861656 -104.67318) (Oklahoma-City OKC 35.393055 -97.60083)
            (Memphis MEM 35.0425 -89.97667) (Indianapolis IND 39.71722 -86.29444))
;; Search: ((Flagstaff FLG 35.140278 -111.66917) (Grand-Jct GJT 39.1225 -108.526665)
            (Oklahoma-City OKC 35.393055 -97.60083))
;; Search: ((San-Francisco SFO 37.61889 -122.375) (Reno RNO 39.49917 -119.76806)
            (Los-Angeles LAX 33.9425 -118.40806) (Grand-Jct GJT 39.1225 -108.526665))
((Boston BOS 42.363056 -71.006386) (Pittsburgh PIT 40.49139 -80.23278)
 (Chicago ORD 41.97861 -87.904724) (Kansas-City MCI 39.2975 -94.71389)
```

```
(Denver DEN 39.861656 -104.67318) (Flagstaff FLG 35.140278 -111.66917)
(San-Francisco SFO 37.61889 -122.375))
```

得られた経路は，ボストン，ピッツバーグ，シカゴ，カンザスシティ，デンバー，フラグスタッフ，サンフランシスコである．この経路を図24.3に示す．

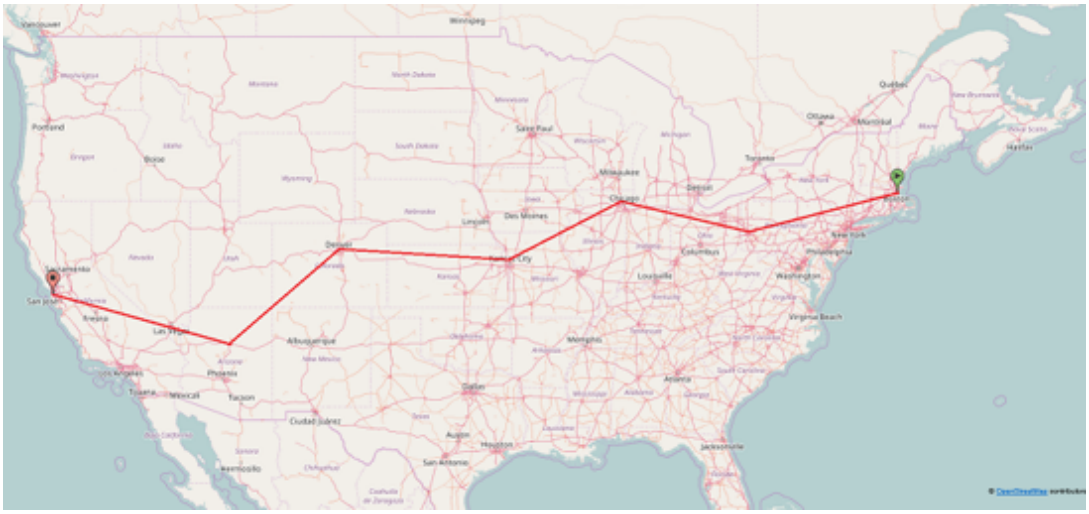


図24.3 最良優先探索の例

これでずいぶんよい経路にはなったが，実はこれが最短経路というわけではない．旅行のコストをどう捉えるかが問題であるが，今，飛行場への着陸・離陸の回数は無視して，単純に全行程の距離が短いほどよいとすると，今度は途中途中でのそれまでの行程距離を評価に加えなければならない．スタート地点から探索経路の途中までのコストと，探索途中からゴールまでの予測コストの両方を考慮する探索方法はA*アルゴリズムと呼ばれている．

A* アルゴリズム

A* アルゴリズムでは，都市ではなく，出発点から途中途中の都市への経路の行程 (journey) のリストを新たな探索空間にする．探索空間を行程にすると，同じ場所にたどり着くのに，複数の行程が出現することもあり得る．

行程 (journey) の定義は次のとおり．

```
(defstruct (journey (:print-function print-journey))
  state (previous nil) (cost-so-far 0) (total-cost 0))

(defun print-journey (journey &optional (stream t) depth)
  (declare (ignore depth))
  (format stream "#<Journey to ~A const ~,1f>"
    (journey-state journey) (journey-total-cost journey)))
```

ここで，state は従来通り city 実現体であるが，previous というのはその名のとおり，現在の journey の一歩手前の journey 実現体，cost-so-far は出発点からこれまでの行程距離，total-cost は cost-so-far に加えて，目的地までの予測コスト（この場合は目的地までの直線距離）も加味したものである．印刷時には都市の名前がその行程のその時点でのトータルコストと共に出力される．

以下のコードでは，常に journeys の先頭が展開されるが，journeys は常にこれまでの行程cost とこれからの予測cost の合計である全cost が小さい順に並ぶようになっている．しかも，もし展開されて新しく得られた都市(state)が同一で異なる行程(journey)が得られた場合には，両方を比較してコストの小さい方を残すようになっている．同一都市の場合にはゴールまでの予測コストは同一であり，違うのはそれまでの行程の経路であることを注意しておく．

```
(defun a*-search (journeys goal-p successors cost-fn cost-left-fn
                          &optional (state= #'eql) old-journeys)
  "Find a journey whose state satisfies goal-p.  Start with journeys,
  and expand successors, exploring least cost first.
  When there are duplicate states, keep the one with the
  lower cost and discard the other."
  (dbg :search ";; Search: ~a" journeys)
  (cond
   ((null journeys) fail)
   ((find-if goal-p journeys)
    (reverse (journey-states (find-if goal-p journeys)))
    )
   (t (let* ((journey (pop journeys))
             (state (journey-state journey)))
        ;; Update states and old-journeys to reflect
        ;; the new successors of STATE:
        (setf old-journeys (insert-journey journey old-journeys))
        (dolist (state2 (funcall successors state))
          (let* ((cost (+ (journey-cost-so-far journey)
                          (funcall cost-fn state2 state)))
                 (cost2 (funcall cost-left-fn state2))
                 (journey2 (make-journey
                            :state state2
                            :previous journey
                            :cost-so-far cost
                            :total-cost (+ cost cost2)))
                 (old nil))
            ;; Place the new journey, journey2, in the right list:
            (cond
             ((setf old (find-journey journey2 journeys state=))
              (when (better-journey journey2 old)
                (setf journeys (insert-journey
                                journey2 (delete old journeys)))))
             ((setf old (find-journey journey2 old-journeys state=))
              (when (better-journey journey2 old)
                (setf journeys (insert-journey journey2 journeys))
                (setf old-journeys (delete old old-journeys))))
             (t (setf journeys (insert-journey journey2 journeys))))))
        ;; Finally, call A* again with the updated journey lists:
        (a*-search journeys goal-p successors cost-fn cost-left-fn
                   state= old-journeys)))))

(defun find-journey (journey journeys state=)
  "Find the journey with this journey's state among a list of journeys."
  (find (journey-state journey) journeys :key #'journey-state :test state=))

(defun better-journey (journey1 journey2)
  "Is journey1 cheaper than journey2?"
  (< (journey-total-cost journey1) (journey-total-cost journey2)))

(defun insert-journey (journey journeys)
```

```
  "Put journey into the right position, sorted by total cost."
  ;; MERGE is a built-in function
  (merge 'list (list journey) journeys #'< :key #'journey-total-cost))

(defun journey-states (journey)
  "Collect the states along this journey."
  (if (null journey)
      nil
      (cons (journey-state journey)
            (journey-states (journey-previous journey)))))

(defun trip5 (start dest)
  (a*-search (list (make-journey :state start))
             (isa dest :key #'journey-state)
             #'neighbors
             #'air-distance
             (distance-to-goal dest)
             ))
```

実行結果は次のようになる.

```
cl-user(15): (trip5 (city 'Boston) (city 'San-Francisco))
;; Search: (#<Journey to (Boston BOS 42.363056 -71.006386) cost 0.0>)
;; Search: (#<Journey to (Pittsburgh PIT 40.49139 -80.23278) cost 4414.7>
            #<Journey to (New-York LGA 40.77725 -73.87261) cost 4437.5>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>)
;; Search: (#<Journey to (Chicago ORD 41.97861 -87.904724) cost 4422.6>
            #<Journey to (New-York LGA 40.77725 -73.87261) cost 4437.5>
            #<Journey to (Indianapolis IND 39.71722 -86.29444) cost 4439.0>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5501.6>)
;; Search: (#<Journey to (New-York LGA 40.77725 -73.87261) cost 4437.5>
            #<Journey to (Indianapolis IND 39.71722 -86.29444) cost 4439.0>
            #<Journey to (Kansas-City MCI 39.2975 -94.71389) cost 4512.0>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 5152.6>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5501.6>)
;; Search: (#<Journey to (Indianapolis IND 39.71722 -86.29444) cost 4439.0>
            #<Journey to (Kansas-City MCI 39.2975 -94.71389) cost 4512.0>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5085.9>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 5152.6>)
;; Search: (#<Journey to (Kansas-City MCI 39.2975 -94.71389) cost 4448.9>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 4834.0>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5085.9>)
;; Search: (#<Journey to (Denver DEN 39.861656 -104.67318) cost 4450.7>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Oklahoma-City OKC 35.393055 -97.60083) cost 4767.6>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 4834.0>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5085.9>)
;; Search: (#<Journey to (Grand-Jct GJT 39.1225 -108.526665) cost 4456.2>
            #<Journey to (Flagstaff FLG 35.140278 -111.66917) cost 4704.3>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
```

```
            #<Journey to (Oklahoma-City OKC 35.393055 -97.60083) cost 4767.6>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 4834.0>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5085.9>)
;; Search: (#<Journey to (Reno RNO 39.49917 -119.76806) cost 4514.4>
            #<Journey to (Flagstaff FLG 35.140278 -111.66917) cost 4704.3>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Oklahoma-City OKC 35.393055 -97.60083) cost 4767.6>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 4834.0>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5085.9>)
;; Search: (#<Journey to (San-Francisco SFO 37.61889 -122.375) cost 4514.4>
            #<Journey to (Flagstaff FLG 35.140278 -111.66917) cost 4704.3>
            #<Journey to (Quebec YQB 46.79111 -71.39333) cost 4736.2>
            #<Journey to (Oklahoma-City OKC 35.393055 -97.60083) cost 4767.6>
            #<Journey to (Memphis MEM 35.0425 -89.97667) cost 4834.0>
            #<Journey to (Atlanta ATL 33.636665 -84.428055) cost 5079.1>
            #<Journey to (Wilmington ILM 34.270557 -77.9025) cost 5085.9>
            #<Journey to (Los-Angeles LAX 33.9425 -118.40806) cost 5378.9>
            #<Journey to (Eugene EUG 44.123055 -123.21861) cost 5521.0>)
((Boston BOS 42.363056 -71.006386) (Pittsburgh PIT 40.49139 -80.23278)
 (Indianapolis IND 39.71722 -86.29444) (Kansas-City MCI 39.2975 -94.71389)
 (Denver DEN 39.861656 -104.67318) (Grand-Jct GJT 39.1225 -108.526665)
 (Reno RNO 39.49917 -119.76806) (San-Francisco SFO 37.61889 -122.375))
```

結果は，ボストン，ピッツバーグ，インディアナポリス，カンザスシティ，デンバー，グランドジャンクション，リノ，サンフランシスコである．図24.5 にA*アルゴリズムによる探索の結果を示す．

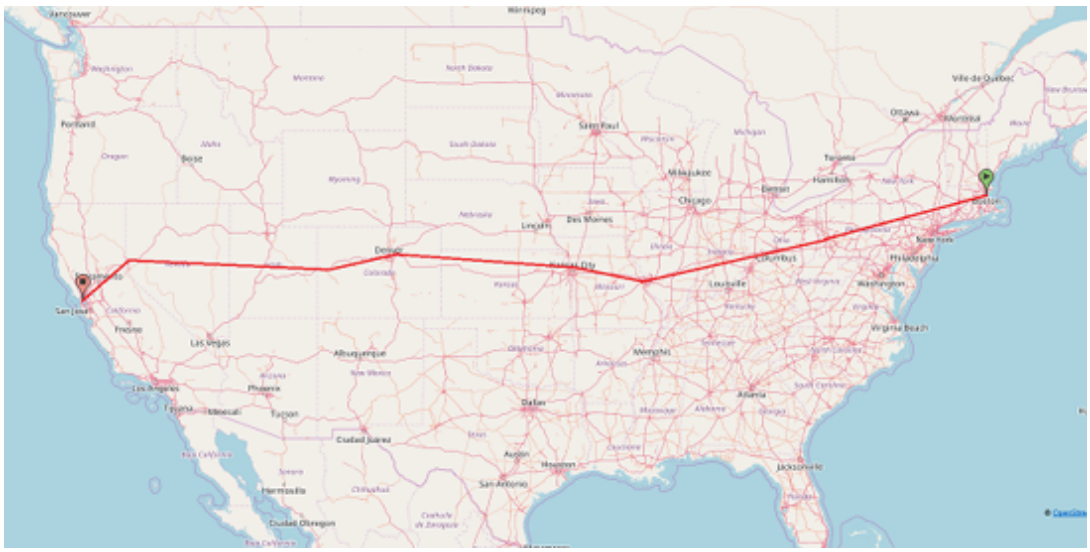


図24.5 A*アルゴリズムによる探索結果

24.4 推論にまつわる話題

人工知能とは，一つには人間の知能とは何なのか，その実態やメカニズムはどうなっているのかを考える科学であり，もう一つには従来人間しかできなかった知的作業をコンピュータに行わせるための技術のことであるが，推論はその主要な基礎的一分野となっている．確かに，推論は知的とよばれるに相応しい機能であるが，我々が日常生活で行う推論と，計算機に行わせるための推論は似て非なるものと考えたほうが良い．たとえば，

ソクラテスは人間である．人間は死すべきものである．ゆえにソクラテスは死ぬ．

は正しいモーダス・ポーネンスの運用であるが，単純な IF ～ THEN ～ ルールの連鎖では誤った結論に導くようなルール集合を作ることは容易であるし，そもそも日常我々が行う推論は，計算機で行わせるようなものではない．

もし加山が謝罪するなら，彼の無礼を忘れてやる

という文章は，日常生活においてはその裏に，（もし加山が謝罪しないなら，彼の無礼を忘れてやらない）という意味を含んでいるが，論理的にはこれはもちろん正しくない．加山が謝罪しなくても，彼の無礼を忘れてしまう可能性が否定されていないからである．ところが（もし加山の無礼を忘れていないのなら，彼は謝罪していない）は現実的にも論理的にも先の文章から正しいが，日常生活においては普通こんな言い方はしない．我々は論理的に正しいことに興味があるわけではないのである．

別の例をあげてみよう．前述のように，

ソクラテスは人間である．人間は死すべきものである．ゆえにソクラテスは死ぬ．

は正しいモーダス・ポーネンスの運用である．ここには IF A THEN B と IF B THEN C から IF A THEN C となる推移律が働いている．しかし次の推移律は正しくない．

次郎は勉強すれば，しかられない．次郎はしかられなければ，勉強しない．
-> 次郎は勉強すれば，勉強しない．

日常生活では，我々は常に生きていくのに必要な「常識的な」推論をしているが，それをいついかなるときでも正しい「論理的な」推論にするためには特別な配慮が必要である．計算機に常に推論を正しく行わせるための形式論理については，第30章で説明するが，第26章のエキスパートシステムでは，得られる結論が常に正しいということは論理的には証明されない．IF ～ THEN ～ ルールの連鎖を正しく行わせるのはルール作成者の役割なのである．

なお，本節で示した例は「日常言語の推論」（板原）より取られた．

第25章 フレームシステム

> 理論の本質はつぎのとおりである．新しい状況(situation)に直面したとき（あるいは，現在の問題に対する見方を実質的に変更した時），人は記憶の中からフレーム(frame)と呼ばれる基本的構造を選び出す．これは，人間が記憶している枠組(framework)で，その細部を必要に応じて変更することにより，現実の世界に適合させるものである．
>
> [Minsky, 1975](白井・杉原訳)

25.1 知識表現の枠組み

フレームという知識表現のアイデアは，Marvin Minsky によって1974年に発表された．彼がフレームを考え付いた動機は人間の認知メカニズムについての洞察にある．

> 人工知能および心理学におけるほとんどの理論で扱う基本単位は，概してあまりにも小さく局所的で非構造的なため，人間の常識的思考の効率のよさを説明する[...]ための役には立たないように思われる．精神的活動の能力とその速さを説明するためには，推論，言語，記憶，“知覚”の“まとまり(chunk)”が，もっと大きく，もっと構造化されていなければならず，また，事実に関する内容と手続に関する内容とが，もっと密接に結びついていなければならない．(白井・杉原訳)

つまり，人間の認知の仕組みをフレームという構造で説明するのが目的であった．実際にはフレームシステムは知識表現の構造化に利用され，その後，フレームシステムの意味論を論理的に明らかにすることを目的に，記述論理 (description logics) が考案された．

フレームは内部構造に複数のスロットを持つが，その中には次のような特殊な意味を持つスロットや機能があるのが特徴となっている．

1. **isa** と呼ぶ特別なスロットがあり，フレームの上位下位関係を記述できる．
2. 上位フレームにおけるスロットを下位フレームは**継承**する．
3. 何も値がないときの**default値**を記述できる．
4. スロット値を取り出すときに動く局所的手続き(デーモン)がある．通常 **if-needed** と呼ばれる．
5. スロット値を与えると動く局所的手続き(デーモン)がある．通常 **if-added** と呼ばれる．

フレームとオブジェクトシステムはよく似ている．オブジェクトシステムでは，すべてのオブジェクトに何らかの形の上位下位関係があって，下位オブジェクトの局所変数は上位オブジェクトのその値を継承する．しかし，if-needed および if-added のデーモンはフレーム特有の特徴である．

著者が最初にフレームを学んだのは，Winston の Lisp 第1版であった．これは当時の MacLisp で書かれていた．ところがどういうわけか，Common Lisp で書かれている第2版にあったフレームのプログラムが第3版にはない．そして PAIP にもフレームシステムのプログラムがない．そこで今回は，今まで歴史的に数多く登場したフレームシステムの中から，比較的コンパクトでかつ上記フレームの特徴をそなえている XRL について解説する(Charniak，Riesbeck, McDermott，Meehan の Artificial Intelligence Programming 第2版).

25.2 XRL システムのロードと実行

本書用 github サイトの FRAME リポジトリからファイルをダウンロードし，それをロードして，たとえば以下のように実行してみる．

```
cl-user(11): (in-package :xrl)
#<The xrl package>
xrl(12): (frame :name thing)
#<frame: thing>
xrl(13): (frame :name dwelling :isa thing)
#<frame: dwelling>
xrl(14): (frame :name apartment :isa dwelling)
#<frame: apartment>
xrl(15): (frame :name apt-at-100-york
                :isa apartment
                :with ((street-name = 'york)
                       (street-number = 100)
                       (wall-color = 'white)
                       (floor-surface = 'wood)))
#<frame: apt-at-100-york>
xrl(16): (frame :name apt1
                :isa apt-at-100-york
                :with ((number-of-rooms = 3)))
#<frame: apt1>
xrl(17): (fget (get-frame 'apt1) 'wall-color)
white
xrl(18): (fget (get-frame 'apt1) 'number-of-rooms)
3
xrl(19): (defun get-street-noise (frame)
           (cond ((not (eq (fget frame 'street-name) 'york))
                  nil)
                 ((> (fget frame 'street-number) 200)
                  'high)
                 (t 'very-high)))
get-street-noise
xrl(20): (frame :name dwelling
                :isa thing
                :with ((street-noise
                           :if-needed #'get-street-noise)))
#<frame: dwelling>
xrl(21): (fget (get-frame 'apt1) 'street-noise)
very-high
```

frame はフレームを定義するときに用いられる次の呼び出し形式を持つマクロである．ここで，:isa には一つのフレーム名しかこれないことに注意せよ．つまり，単一継承である．なお，第27章の事例ベース推論では複数の上位フレームを許す，より複雑なコードになっている．

```
(frame :name 《name》
       :isa 《super-name》
       :with ((《slot》 《facet》 《value》)*))
```

《name》 や 《super-name》 にはフレームの名前をLispシンボルで与えるが，frame をマクロにすれば関数のように引数をクォートする必要はない．キーワードパラメータ with には (《slot》 《facet》 《value》) という書式のリストをクォートなしで与えるが，《facet》 には以下の種類とそれぞれの役割がある．

- = はそのフレームのそのスロットに値をセットする．
- :if-needed はそのフレームのそのスロットにif-neededデーモンとなる関数をセットする．
- :if-added はそのフレームのそのスロットにif-addedデーモンとなる関数をセットする．

関数 get-frame はシンボルからその名前を持つフレーム実体を取り出す．関数 fget はフレームとスロット名を引数として，その値を取り出す．

上記実施例12～16では上位フレームから順にフレームを定義している．:isa でそのフレームの直接の上位フレームの名前を与えている．実施例18では apt1 の部屋数を聞いていて，実施例17では壁の色を聞いているが，壁紙の色はフレーム apt1 では定義されていなくて，その上位フレーム apt-at-100-york で与えられている．このように該当フレームのスロットに値がない場合には，その上位フレームのスロット値がその値となる．これをスロットの**継承**という．

実施例19で定義された関数が，実施例20で上位フレームの if-needed ファセットの値として与えられているが，その影響で実施例21でフレーム apt1 の street-noise を取りに行ったとき，この関数が働き，その結果が値として返される．これが **if-needed** のデーモンである．

25.3 XRL のプログラムコード

XRL のフレームは次のような Lisp の構造体として実現される．

```
(defstruct (frame
            (:print-function
             (lambda (frame stream depth)
               (declare (ignore depth))
               (format stream "#<frame: ~S>"
                 (frame-name frame)))))
  name
  (isa nil)
  (slot-table
   (make-hash-table :size *slot-table-size*)))

(defvar *slot-table-size* 10
  "expected average number of slots")
```

この構造体定義では，構造体のインスタンスが作られるときに，name スロットと isa スロット以外に slot-table スロットに個別のフレーム用のハッシュテーブルを作っておく．

フレーム名をキーとしてフレーム実体を辞書引きするためのハッシュテーブルが大域変数 *frame-table* に置かれる．フレーム名からフレーム実体を引き出すには get-frame が用いられ，フレーム実体を作ってフレーム名のキーバリューとして保存するには set-frame が用いられる．ここでは，もし既にその名前のフレーム名があれば set-frame はただ単にそれを取りだすだけであり，さもなければ，新しいフレーム実体を作って保存することに注意されたい．なお，このときスロット情報は何もないが，スロットテーブルは用意されることにも注意されたい．

```
(defvar *frame-table* (make-hash-table))

(defun get-frame (frame-name)
  "return the frame stored under frame-name,
   if any.  If none, signal an error."
  (multiple-value-bind (frame found-p)
      (gethash frame-name *frame-table*)
    (if found-p
        frame
      (error "No such frame as ~S.~%" frame-name))))

(defun set-frame (frame-name)
  "return the frame stored under <frame-name>,
   if any.  If none, make one."
  (multiple-value-bind (frame found-p)
      (gethash frame-name *frame-table*)
    (if found-p
        frame
      (setf (gethash frame-name *frame-table*)
        (make-frame :name frame-name)))))
```

スロットテーブルの中には，今度はファセットをキーとするファセットテーブルがあるが，ここでslotというのはスロットテーブルのキーとなるLispシンボルであり，get-facet で frame の slot の facet の値が取り出される．facet はファセットテーブルのキーとなるLispシンボルであり，ファセットテーブルはそのスロットに最初に facet の値を保存するときに作られて保存され，以後はそのファセットテーブルが用いられる．

```
(defun get-facet (frame slot facet)
  "return the value of <facet> of <slot> of <frame>."
  (assert (typep frame 'frame) ()
          "The first argument frame must be an instance of frame.")
  (let ((slot-table (frame-slot-table frame)))
    ;; the slot-table is always defined when a frame is made.
    (let ((facet-table (gethash slot slot-table)))
      (when (hash-table-p facet-table)
        ;; returns only a primary value
        (values (gethash facet facet-table))))))

(defun set-facet (frame slot facet value)
  "set the value of the <facet> of the <slot> of the
   <frame>."
  (let ((slot-table (frame-slot-table frame)))
    (setf (gethash facet
                   (force-entry slot slot-table
                                #'(lambda ()
                                    (make-hash-table
                                     :size *facet-table-size*))))
      value)))

(defsetf get-facet set-facet)

(defun force-entry (index table maker)
  (multiple-value-bind (entry found-p)
      (gethash index table)
    (if found-p
        entry
```

```
      (setf (gethash index table)
        (funcall maker)))))
```

explicit-frame-slots はそのフレームに明示的に定義してあるスロットをリストアップする.

```
(defun explicit-frame-slots (frame)
  "returns a list of the slot names stored with
   frame."
  (let ((slots '()))
    (maphash #'(lambda (slot facet-table)
                 (declare (ignore facet-table))
                 (push slot slots))
             (frame-slot-table frame))
    slots))
```

実例は以下のとおり.

```
xrl(17): (explicit-frame-slots (get-frame 'apt1))
(number-of-rooms)
xrl(18): (explicit-frame-slots (get-frame 'apt-at-100-york))
(street-name street-number floor-suface wall-color)
xrl(19):
```

マクロである frame の使い方は前節に示したが，そのプログラムコードは次のようになっている.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defmacro frame (&key (name (gensym "frame")) isa with)
    "(frame :name name
            :isa super-name
            :with ((<slot> <facet> <value>)*))
     Macro that expands into a call to <store-frame>,
     generating a value for <name>, if missing,
     and quoting everything except the <value>s."
    `(store-frame ',name ',isa
                  (list ,@(loop for spec in with
                                collect `(list ',(first spec)
                                               ',(second spec)
                                               ,(third spec))))))

  (defun store-frame (frame-name isa with)
    "return a frame whose name is frame-name, with an
     isa link to the frame named by super-name, if
     given, and whose slots and facets are set by
     spec-list."
    (let ((frame (set-frame frame-name)))
      (when isa (setf (frame-isa frame) (set-frame isa)))
      (loop for spec in with
          do (apply #'set-facet frame spec))
      frame))
  )
```

最後に継承関係も含んで，フレームのスロットの値をとってくる機能であるが，ここでの実装は非常に原始的で，関数 fget は該当フレームの slot のファセットが = の値を get-facet で取り出すが，もしそれがなければ，isa 関係をたどって，上位の同一のslot の値を探索する．その探索に用いられるのが some-isa で，値が見つかればそれを返すが，なければ isa 関係にあるフレームを取りだして再帰する．探索は自分自身か

ら行われることを注意しておく．探索対象となる上位フレームが尽きれば，見つからなかったということで nil を返す．ここで，該当フレームに値がないとき，上位に探索に行く前に，if-needed-valが起動され，該当フレームの :if-needed ファセットの値が取りだされる．ここに関数があれば，その関数が引数をフレームとして実行され，それから値が得られれば，その値が返される．前節のデモンストレーションで，関数 get-street-noise が if-needed 関数であり，実施例19および20の処置を施すことにより，21でこれが作用していることを確かめられたい．

```
(defun fget (frame slot)
  (flet ((if-needed-val (an-isa slot frame)
                        (let ((fn (get-facet an-isa slot
                                             :if-needed)))
                          (if fn (funcall fn frame) nil))))
    (some-isa
     #'(lambda (an-isa)
         (or (get-facet an-isa slot '=)
             (if-needed-val an-isa slot frame)))
     frame)))

(defun some-isa (fn frame)
  (cond ((null frame) nil)
        ((funcall fn frame))
        (t (some-isa fn (frame-isa frame)))))

(defun set-fget (frame slot value)
  (set-facet frame slot '= value))
(defsetf fget set-fget)

(defun frame-slots (frame)
  (let ((slots '()))
    (some-isa
     #'(lambda (an-isa)
         (setq slots
               (union slots
                      (explicit-frame-slots an-isa)))
         nil)
     frame)
    slots))
```

問題25.1 24.1節で取り上げた動物のエキスパートシステムの動物をフレームで記述せよ．スロット名として has と can を用いるといいだろう．

問題25.2 *facts* と *hypotheses* を定義して，24.1節と同様な推論をフレームベースで行う deduce と verify をプログラムせよ．

問題25.3 上記のコードには if-added がない．これに if-added を追加せよ．そして適当な例題を考えよ．

25.4 フレームとはなんであったか？

Minsky がフレームのアイデアを発表したのは1974年であったが，オブジェクト指向言語である Smalltalk80がSmalltak-72, -76と開発されてきたことから見ると，多くのフレームシステム開発とオブジェクト指向言語の開発は，それぞれ独立して並行して行われたと見るのが妥当であろう．Smalltalkがプログラ

ミングにおける新たなパラダイム提供を目的としていたのに対して，フレーム理論は人間の認知メカニズムについての考察から着想を得て，記憶機構と記憶のまとまり（チャンキング）を問題意識としていた．

歴史的に見ると，1974年のMinskyの論文以後フレームシステムは，MITによるFRL (1977)，BobrowとWinogradによるKRL (1977)，CMUのSRL (1985)，BrackmanとSchmolzeによる KL-ONE (1985)など，多くのシステムが開発されてきたが，第２次ＡＩブームの影響もあって，これらは当時意味ネットワークに並ぶ知識表現言語という位置づけであった．フレームシステムの最大の特徴は，推移律を有する is-a 階層または ako (a kind of)階層を特別な属性として，その他の属性の継承機能を is-a 関係によって実現したことである．この機能は今ではオブジェクト指向プログラミングにおいても当然の機能となっているが，フレームシステム以前に属性の継承機能を有するシステムは無かった．これによって，概念の包摂関係を踏まえて世界に関する知識をどう記述するかが知識表現問題と意識されるようになった．特にKL-ONEにおいては，フレーム実現体のタイプ付け(classifier)問題が明確に意識されており，後の記述論理(Description Logics)にまで影響を与えた．フレームシステムでは，まず最初に is-a 階層ありきであるが，逆にどのように整合的な is-a 階層を定義したらよいかが問題となる．第27章の事例ベース推論においては属性と値の束からどのように is-a 階層を形成したらよいかが議論される．

一方，if-needed，if-addedのような局所的な手続きのフレームへの付与(local procedural attachment)もKL-ONEを除くフレームシステムに共通の特徴であったが，これがその後の知識表現に大きな影響を与えたとは思われない．事例ベース推論では if-needed や if-added に限らないより一般的な局所的手続き活性化の機能が開発されたが，そのような局所的手続きは形式化の欠如のためかオントロジーやプログラミングパラダイムへの影響は皆無といってよく，その潜在的可能性をもっと研究開発する必要があると思われる．

第26章 ルールベースシステム

> 問題解決システムの能力を議論するにあたっては，一般性と専門性とを区別しなければならない．一般性では問題解決器がどのくらい広範囲な問題領域に働くのかが問われるし，専門性では合理的なコストでどのくらいよい答えに到達できるのかが問われる．ものによっては一般性が非常に有益だが，しばしば優れた性能が伴わない．専門家は，普通はスペシャリストである．
>
> [Feigenbaum, Buchanan, Lederberg, 1971]

> 専門技術は特定の領域についての知識と，その領域内での問題の理解と，それらの問題のいくつかを解く技術(skill)からなっている．[...] 専門家は一般的に，公表される文献の中にはけっして書かれることのない私的な知識を持っている．この私的知識の多くはヒューリスティックスと呼ばれることになった経験法則 (rule of thumb) からなっている．ヒューリスティックスは専門家が必要とあれば教養のある推測をやり，問題に対する有望なアプローチを認識し，エラーの多い不完全なデータを効果的に扱うことを可能にする．そのような知識を明らかにし再構成することがエキスパート・システム構築における中心的な課題である．
>
> [Hayes-Roth, Waterman, Lenat (eds.), Building Expert Systems, 1983]
> (エキスパートシステム， AIUEO訳)

26.1 ルールベースシステムとは

24.1 節の前向き推論と後ろ向き推論で述べたように，ルールベースシステムとは IF ～ THEN ～ という形式を知識の記述形式として，推論エンジンと組み合わせて何かを実行するシステムのことである．本書で登場するようなAIシステムの範囲では，書こうと思えばどんな知識でもルールで記述することができるが，IF ～ THEN ～ という形式が専門家の持つ高度な専門知識を記述するのに便利と考えられて，一時期専門家の仕事を代替するエキスパートシステムを実装するのにルールベースシステムが盛んに用いられた．

26.2 MYCIN のロードと実行

Mycin は感染症の患者について，その原因となる細菌の同定を行うエキスパートシステムである．最初にGitHubリポジトリからMycinをダウンロードして，そのダウンロードしたディレクトリに移動し，次のように実行してみよう．ただし，sbcl の方は入力時に小文字ではなく大文字入力が要求される（femailではなくFEMAIL）ことに注意されたい．

```
cl-user(1): :cd
C:\Users\誠二\
cl-user(2): :cd Documents/GitHub/mycin
C:\Users\誠二\Documents\GitHub\mycin\
cl-user(3): (require :asdf)
; Fast loading C:\acl90-smp.64\code\asdf.005
;;; Installing asdf patch, version 5.
t
cl-user(4): :ld mycin.asd
; Loading C:\Users\誠二\Documents\GitHub\mycin\mycin.asd
cl-user(5): (asdf:load-system :mycin)
```

```
...
...

t
cl-user(6): (mycin)
------ PATIENT-1 ------
Patient's name:  Jane Doe
Sex: femail
Illegal reply.  Type ? to see legal ones.
Sex: female
Age: 27
------ CULTURE-1 ------
From what site was the specimen for CULTURE-1 taken? blld
Illegal reply.  Type ? to see legal ones.
From what site was the specimen for CULTURE-1 taken? blood
How many days ago was this culture (CULTURE-1) obtained? 3
------ ORGANISM-1 ------
Enter the identity (genus) of ORGANISM-1: unknown
The gram stain of ORGANISM-1: ?
A gram must be of type (member acid-fast pos neg)
The gram stain of ORGANISM-1: neg
Is ORGANISM-1 a rod or coccus (etc.): rod
What is the aerobicity of ORGANISM-1? why
[Why is the value of aerobicity being asked for?]
It is known that:
    1) the gram of the ORGANISM is neg
    2) the morphology of the ORGANISM is rod
Therefore,
Rule 107:
  If
    1) the aerobicity of the ORGANISM is aerobic
  Then there is suggestive evidence (0.8) that
    1) the identity of the ORGANISM is enterobacteriaceae
What is the aerobicity of ORGANISM-1? aerobic
Is Jane Doe a compromised host? yes
Is Jane Doe a burn patient?  If so, mild or serious? why
[Why is the value of burn being asked for?]
It is known that:
    1) the site of the CULTURE is blood
    2) the gram of the ORGANISM is neg
    3) the morphology of the ORGANISM is rod
Therefore,
Rule 52:
  If
    1) the burn of the PATIENT is serious
  Then there is weakly suggestive evidence (0.4) that
    1) the identity of the ORGANISM is pseudomonas
Is Jane Doe a burn patient?  If so, mild or serious? serious
Findings for ORGANISM-1:
 identity: enterobacteriaceae (0.800)   pseudomonas (0.760)
Is there another ORGANISM? no
Is there another CULTURE? no
Is there another PATIENT? no
nil
cl-user(7):
```

ここでは結論として腸内細菌科(Enteobacteriaceae)の細菌（確信度 0.8），またはシュードモナス属(Pseudomonas)の細菌（確信度 0.76）とされた．エンテロバクターはグラム陰性 (gram negatic)，嫌気

性 (anaerobic) の桿菌 (rod) で，シュードモナスはグラム陰性，好気性 (aerobic) の桿菌である．この例ではユーザはグラム染色の結果についてはグラム陰性，かつ菌の形態について桿菌，と答えたため，好気性か嫌気性かがわからないが，ルール52の知識からひどいやけどの患者であればシュードモナスの可能性も弱く支持されるということで，答えとしてこのように二つそれぞれの確信度つきで示された．compromised host というのは易感染宿主のことである．

Mycinからの質問に対して，Mycin では期待される入力候補が質問ごとに決められており（実はパラメータとその取りうる値），ミスタイプした場合には再入力が促されるし，何を入力したらよいか分からないときは，「?」と打ち込むことで入力すべきものを知ることができる．また「why」で質問する理由を確かめることもできる．

Mycin コンテキスト

Mycin では論理変数を使わずに，患者(patient)，培養された細菌(culture)，そして同定のターゲットとなる細菌(organism)というカテゴリーをプログラム中で特別に扱う．これらの三つのカテゴリーを合わせて，Mycinではコンテキストと呼んでいるが，言ってみれば patient，culture，organizm は推論対象カテゴリーに相当し，ユーザとの質問応答により，これらの三種類のカテゴリーの詳細を決めていくことが，Mycin における推論過程である．三つのコンテキストの実際は複数あり得て，プログラム中でそれぞれ PATIENT-1とか，CULTURE-2とか，ORGANIZM-3とか呼ばれるが，一つのPATIENTにつき複数のCULTURE，一つのCULTUREにつき複数のORGANIZMがあり得て，PATIENT，CULTURE，ORGANIZMの順に推論が進められる．このとき決定されるコンテキスト実現の相互関係はコンテキストツリーと呼ばれている．つまり推論開始前には PATIENT – CULTURE – ORGANIZM の一直線のカテゴリー・ツリーだったものが，推論によって実際のデータを有するコンテキストのツリー構造に変わっていくとみなすことができる．コンテキストツリーの例はこちらのFIGURE5-1(pp.84)を参照されたい．

以下にコンテキストに関するプログラムコードを示す．

```
(defstruct context
  "A context is a sub-domain, a type."
  name (number 0) initial-data goals)

(defmacro defcontext (name &optional initial-data goals)
  "Define a context."
  `(make-context :name ',name :initial-data ',initial-data
                 :goals ',goals))
```

メインとなる関数 mycin を実行すると，最初に三つのコンテキスト実現体の定義が行われるが，その中身は何もない．ここで，initial-data として与えられる name，sex，age，site，days-old，identity などはパラメータと呼ばれているが，これらパラメータの値を決めていくのが本システムの仕事である．

```
(defun mycin ()
  "Determine what organism is infecting a patient."
  (emycin
    (list (defcontext PATIENT  (name sex age)  ())
```

```
(defcontext CULTURE  (site days-old) ())
(defcontext ORGANISM ()              (identity)))))
```

Mycin 確信度

エキスパートの経験的知識では，絶対的な真偽値を言うことができるのはまれであって，「～と確信する」とか，「おおよそ～と思う」とか，「～かも知れない」とかといった，曖昧な表現がよく用いられる．Mycinではこれを確信度 (ceatainty factor)と呼んで －1から+1 までの数値で表す．Mycinにおけるルールには，ルールごとに確信度を表す数値が付いており，推論による IF ～ THEN ～ の連鎖の結果として結論にも確信度が付与される．問題はどのようにして各ルールの確信度を結果に反映させるのかであるが，これをMycinでは次のような数式で行っている．同じ結論を支持する二つのルールがあったとき，ルールの確信度AとBのOR結合の合成は以下の式になる．

```
combine(A, B) =
  A + B - AB;                 A,B > 0 のとき
  A + B + AB;                 A,B < 0 のとき
  A + B / (1 - min(|A|,|B|)); それ以外
```

実際のリスプコードは以下の通り

```
(defun cf-or (a b)
  "Combine the certainty factors for the formula (A or B).
  This is used when two rules support the same conclusion."
  (cond ((and (> a 0) (> b 0))
         (+ a b (* -1 a b)))
        ((and (< a 0) (< b 0))
         (+ a b (* a b)))
        (t (/ (+ a b)
              (- 1 (min (abs a) (abs b)))))))
```

この計算式の特徴は，AもしくはBが 0 のときは計算結果はBもしくはAの値となり，AとBが－1から+1までのときに．結果が1以上や-1以下になることはなく，AもしくはBが正の場合は結果はBもしくはAの値を強化して，負の場合には弱化するということである．論理的にはこの計算式は or の場合に用いられ，and の場合にはどちらかの最小値が選択される．

```
(defun cf-and (a b)
  "Combine the certainty factors for the formula (A and B)."
  (min a b))
```

問題26.1 Bをパラメータとして-1から0.2ごとに+1まで振ったときの，横軸をA，縦軸をcombine(A, B)の値としたときのグラフを描け．

Mycin キャッシング

Mycin ではフレームのような構造化された記憶装置はないが，推論過程における重要なデータはデータベースにキャッシュされ，無駄な質問をしたりしないようになっている．データベースのキーインデックスと

なるのは，任意のシンボルやリストである．以下にキャッシュ用データベースのコードを示す．ハッシュテーブルの同等性比較に equal が用いられていることに注意されたい．

```
(let ((db (make-hash-table :test #'equal)))
  (defun get-db (key) (gethash key db))
  (defun put-db (key val) (setf (gethash key db) val))
  (defun clear-db () (clrhash db)))
```

トップレベルでいきなり let 変数を導入しているが，その理由や詳細は7.2節の「エクステントとは何か？」を参照されたい．以下に put-db をトレースしたときの最初の部分を示す．下線を引いた部分がデータベースのキーインデックスに相当する．

```
 0[5]: (put-db PATIENT "PATIENT-1")
0[5]: returned "PATIENT-1"
0[5]: (put-db current-instance "PATIENT-1")
0[5]: returned "PATIENT-1"
0[5]: (put-db current-rule initial)
0[5]: returned initial
0[5]: (put-db (asked name "PATIENT-1") t)
0[5]: returned t
```

Mycin パラメータ

前述のように，Mycin コンテキストにはそれぞれ固有のパラメータが属性のようにあり，それぞれのパラメータの値が入力や推論によって決められる．パラメータに関するコードを以下に示す．

```
(defstruct (parm (:constructor
                  new-parm (name &optional context type-restriction
                                 prompt ask-first reader)))
  name (context nil) (prompt "~&What is the ~*~a of ~2:*~a?")
  (ask-first nil) (type-restriction t) (reader 'read))

(defmacro defparm (parm &rest args)
  "Define a parameter."
  `(setf (get ',parm 'parm) (apply #'new-parm ',parm ',args)))

(defun parm-type (parm-name)
  "What type is expected for a value of this parameter?"
  (parm-type-restriction (get-parm parm-name)))

(defun get-parm (parm-name)
  "Look up the parameter structure with this name."
  ;; If there is none, make one
  (or (get parm-name 'parm)
      (setf (get parm-name 'parm) (new-parm parm-name))))
```

たとえば，PATIENT には全部で5個のパラメータがあり，name パラメータはどんな値でも値でもよいが．sex パラメータ値は male か female である．それぞれプロンプトのあと，name パラメータは read-line で読み込まれて文字列となるが，sex パラメータ値は規定値としての read 関数で読み込まれてシンボルになる．

```
;;; Parameters for patient:
(defparm name PATIENT t "Patient's name: " t read-line)
(defparm sex PATIENT (member male female) "Sex:" t)
(defparm age PATIENT number "Age:" t)
(defparm burn PATIENT (member no mild serious)
  "Is ~a a burn patient?  If so, mild or serious?" t)
(defparm compromised-host PATIENT yes/no
  "Is ~a a compromised host?")

;;; Parameters for culture:
(defparm site CULTURE (member blood)
  "From what site was the specimen for ~a taken?" t)
(defparm days-old CULTURE number
  "How many days ago was this culture (~a) obtained?" t)

;;; Parameters for organism:
(defparm identity ORGANISM
  (member pseudomonas klebsiella enterobacteriaceae
          staphylococcus bacteroides streptococcus)
  "Enter the identity (genus) of ~a:" t)
(defparm gram ORGANISM (member acid-fast pos neg)
  "The gram stain of ~a:" t)
(defparm morphology ORGANISM (member rod coccus)
  "Is ~a a rod or coccus (etc.):")
(defparm aerobicity ORGANISM (member aerobic anaerobic))
(defparm growth-conformation ORGANISM
  (member chains pairs clumps))
```

Mycin ルール

今まで述べてきたように，Mycin のルールは確信度つきである．ルールの記述形式は以下のようになっている．

```
(defrule <ID> if <前件部並び> then <cf> <後件部>)
```

驚いたことに，このPAIPコードではルールはたった 6 個しかない．

```
(defrule 52
  if (site CULTURE is blood)
     (gram ORGANISM is neg)
     (morphology ORGANISM is rod)
     (burn PATIENT is serious)
  then .4
     (identity ORGANISM is pseudomonas))

(defrule 71
  if (gram ORGANISM is pos)
     (morphology ORGANISM is coccus)
     (growth-conformation ORGANISM is clumps)
  then .7
     (identity ORGANISM is staphylococcus))

(defrule 73
  if (site CULTURE is blood)
     (gram ORGANISM is neg)
     (morphology ORGANISM is rod)
     (aerobicity ORGANISM is anaerobic)
```

```
  then .9
     (identity ORGANISM is bacteroides))

(defrule 75
  if (gram ORGANISM is neg)
     (morphology ORGANISM is rod)
     (compromised-host PATIENT is yes)
  then .6
     (identity ORGANISM is pseudomonas))

(defrule 107
  if (gram ORGANISM is neg)
     (morphology ORGANISM is rod)
     (aerobicity ORGANISM is aerobic)
  then .8
     (identity ORGANISM is enterobacteriaceae))

(defrule 165
  if (gram ORGANISM is pos)
     (morphology ORGANISM is coccus)
     (growth-conformation ORGANISM is chains)
  then .7
     (identity ORGANISM is streptococcus))
```

マクロ defrule は呼び出されるたびに構造体 rule の実現を生成し，ルール・データベースに追加していく

```
(defmacro defrule (number &body body)
  "Define a rule with conditions, a certainty factor, and
  conclusions.  Example: (defrule R001 if ... then .9 ...)"
  (assert (eq (first body) 'if))
  (let* ((then-part (member 'then body))
         (premises (ldiff (rest body) then-part))
         (conclusions (rest2 then-part))
         (cf (second then-part)))
    ;; Do some error checking:
    (check-conditions number premises 'premise)
    (check-conditions number conclusions 'conclusion)
    (when (not (cf-p cf))
      (warn "Rule ~a: Illegal certainty factor: ~a" number cf))
    ;; Now build the rule:
    (put-rule
       (make-rule :number ',number :cf ,cf :premises ',premises
                  :conclusions ',conclusions))))

(defstruct (rule (:print-function print-rule))
  number premises conclusions cf)
```

ルール・データベースは，キャッシュ用データベースと同様な作り方になっている．

```
(let ((rules (make-hash-table)))

  (defun put-rule (rule)
    "Put the rule in a table, indexed under each
    parm in the conclusion."
    (dolist (concl (rule-conclusions rule))
      (push rule (gethash (first concl) rules)))
    rule)
```

```
  (defun get-rules (parm)
    "A list of rules that help determine this parameter."
    (gethash parm rules))

  (defun clear-rules () (clrhash rules)))
```

通常のプログラム感覚では，ルールベースにルールを登録するときのキーにはルールIDが用いられそうだが，ここではそうではなく，後件部そのものが用いられている．かくして，後件部をキーとしてそのルールを取りだすことができる．

ちなみに，Mycin をロード直後に，`(get-rules 'identity)` を実施すると次のようになる．

```
cl-user(6): (get-rules 'identity)
(
Rule 165:
  If
    1) the gram of the ORGANISM is pos
    2) the morphology of the ORGANISM is coccus
    3) the growth-conformation of the ORGANISM is chains
  Then there is suggestive evidence (0.7) that
    1) the identity of the ORGANISM is streptococcus
Rule 107:
  If
    1) the gram of the ORGANISM is neg
    2) the morphology of the ORGANISM is rod
    3) the aerobicity of the ORGANISM is aerobic
  Then there is suggestive evidence (0.8) that
    1) the identity of the ORGANISM is enterobacteriaceae
Rule 75:
  If
    1) the gram of the ORGANISM is neg
    2) the morphology of the ORGANISM is rod
    3) the PATIENT is compromised-host
  Then there is suggestive evidence (0.6) that
    1) the identity of the ORGANISM is pseudomonas
Rule 73:
  If
    1) the site of the CULTURE is blood
    2) the gram of the ORGANISM is neg
    3) the morphology of the ORGANISM is rod
    4) the aerobicity of the ORGANISM is anaerobic
  Then there is strongly suggestive evidence (0.9) that
    1) the identity of the ORGANISM is bacteroides
Rule 71:
  If
    1) the gram of the ORGANISM is pos
    2) the morphology of the ORGANISM is coccus
    3) the growth-conformation of the ORGANISM is clumps
  Then there is suggestive evidence (0.7) that
    1) the identity of the ORGANISM is staphylococcus
Rule 52:
  If
    1) the site of the CULTURE is blood
    2) the gram of the ORGANISM is neg
    3) the morphology of the ORGANISM is rod
    4) the burn of the PATIENT is serious
  Then there is weakly suggestive evidence (0.4) that
```

```
    1) the identity of the ORGANISM is pseudomonas)
t
```

Mycin 後向き推論

Mycin における推論は後向き推論に相当する．トップレベルで関数 mycin を呼び出すと，mycin は PATIENT，CULTURE，ORGANISM の順に中身が空のコンテキストを作って，その順番のリストを引数として emycin を呼び出す．関数 emycin では最初にキャッシュ用データベースをクリアしたのち，get-context-data を呼び出す．

```
(defun emycin (contexts)
  "An Expert System Shell.  Accumulate data for instances of each
  context, and solve for goals.  Then report the findings."
  (clear-db)
  (get-context-data contexts))
```

get-context-data は引数としてもらったコンテキストについて，頭から順番に推論をひとつづつ進めていくが，いつも最後にそのコンテキストについてもう一つ同種のコンテキストがないか確認している．推論は最初にコンテキストの初期値(initial-data)を取りだして入力を行い，次に goals について後向き推論を行うようになっているが，推論のかぎは find-out にある．find-out ではキャッシュされた値がなければ，(ask-vals parm inst) あるいは (use-rules parm) によりパラメータの値を取りだし，それを (known <パラメータ> <コンテキスト実現体>) をキーとする値として保存する．parm-ask-first でそのパラメータの ask-first スロットの値が得られることを注意しておく．

```
(defun get-context-data (contexts)
  "For each context, create an instance and try to find out
  required data.  Then go on to other contexts, depth first,
  and finally ask if there are other instances of this context."
  (unless (null contexts)
    (let* ((context (first contexts))
           (inst (new-instance context)))
      (put-db 'current-rule 'initial)
      (mapc #'find-out (context-initial-data context))
      (put-db 'current-rule 'goal)
      (mapc #'find-out (context-goals context))
      (report-findings context inst)
      (get-context-data (rest contexts))
      (when (y-or-n-p "Is there another ~a? "
                      (context-name context))
        (get-context-data contexts)))))

(defun find-out (parm &optional (inst (get-db 'current-instance)))
  "Find the value(s) of this parameter for this instance,
  unless the values are already known.
  Some parameters we ask first; others we use rules first."
  (or (get-db `(known ,parm ,inst))
      (put-db `(known ,parm ,inst)
              (if (parm-ask-first (get-parm parm))
                  (or (ask-vals parm inst) (use-rules parm))
                  (or (use-rules parm) (ask-vals parm inst))))))
```

PATIENT コンテキストと CULTURE コンテキストには initial-data しかなく，goals はない．PATIENT コンテキストの initial-data は name，sex，age のパラメータで，CULTURE コンテキストのパラメータは site と days-old である．だからこれらのデータは find-out の中の ask-vals でユーザからの入力が順番に行われる．一方，ORGANISM コンテキストの番になると，今度は goals しかなく，そのパラメータは identity である．そこで初めて，実際の推論が use-rules で行われる．

use-rules では get-rules で受け取った parm に関連するルールを取りだしてその一つ一つに順番に use-rule を適応し，一つでも真だと確信が持てるものがあればそれを返す．この場合，`(get-rules 'identity)` で得られるルール集合は，たまたま全ルール集合である．use-rule は一つのルールの前件部に一つでも reject されたものがあれば何もしないが，そうでなければ，satisfy-premises を用いて確信度を計算し，真という確信が得られれば即座に後件部を確信度つきでconcludeを使って成立させる．

```
(defun use-rules (parm)
  "Try every rule associated with this parameter.
  Return true if one of the rules returns true."
  (some #'true-p (mapcar #'use-rule (get-rules parm))))

(defun use-rule (rule)
  "Apply a rule to the current situation."
  ;; Keep track of the rule for the explanation system:
  (put-db 'current-rule rule)
  ;; If any premise is known false, give up.
  ;; If every premise can be proved true,  then
  ;; draw conclusions (weighted with the certainty factor).
  (unless (some #'reject-premise (rule-premises rule))
    (let ((cf (satisfy-premises (rule-premises rule) true)))
      (when (true-p cf)
        (dolist (conclusion (rule-conclusions rule))
          (conclude conclusion (* cf (rule-cf rule))))
        cf))))
```

satisfy-premises では，引数にもらった前件部すべてを評価して確信度を集積していくが，最初は+1.0から出発して true-p ではないと評価されたところで偽を返して終わるが，最後まで評価されたときは一つのルールの前件部による確信度集積結果を返す．

```
(defconstant true   +1.0)
(defconstant false  -1.0)
(defconstant unknown 0.0)

(defun satisfy-premises (premises cf-so-far)
  "A list of premises is satisfied if they are all true.
  A combined cf is returned."
  ;; cf-so-far is an accumulator of certainty factors
  (cond ((null premises) cf-so-far)
        ((not (true-p cf-so-far)) false)
        (t (satisfy-premises
             (rest premises)
             (cf-and cf-so-far
                     (eval-condition (first premises)))))))

(defun eval-condition (condition &optional (find-out-p t))
```

```
  "See if this condition is true, optionally using FIND-OUT
  to determine unknown parameters."
  (multiple-value-bind (parm inst op val)
      (parse-condition condition)
    (when find-out-p
      (find-out parm inst))
    ;; Add up all the (val cf) pairs that satisfy the test
    (loop for pair in (get-vals parm inst)
          when (funcall op (first pair) val)
          sum (second pair))))
```

ここで condition として渡されるものは，ルールの前件部における一つの条件文，たとえば，(mophology ORGANISM is coccus) といった，コンテキストは抽象であるものだが，parse-condition はこれを (morphology ORGANISM-1 is coccus) のように，現実の現コンテキストに変更したものを多値として返す．だから，eval-condition の中では評価し始めたコンテキストと前件部について関連するパラメータの情報を find-out で再帰的に調べまくる．これにより，あるルール中でまだ調べられていない前件部の条件があればそれが調べられるようになる．

```
(defun parse-condition (condition)
  "A condition is of the form (parm inst op val).
  So for (age patient is 21), we would return 4 values:
  (age patient-1 is 21), where patient-1 is the current patient."
  (values (first condition)
          (get-db (second condition))
          (third condition)
          (fourth condition)))

(defun is (a b) (equal a b))

(defun get-vals (parm inst)
  "Return a list of (val cf) pairs for this (parm inst)."
  (get-db (list parm inst)))
```

ここでは，op にわたってくるのは関数 is しかない．すなわち，get-vals で返される値は パラメータとコンテキストの対であるが，eval-condition の中では最後にデータベース中にキャッシュされた値を使って値を返している．eval-condition で返された値は，satisfy-premises の中で cf-and を使って集積される．つまり一つのルール中の前件部はAND結合として計算される．これは常識的である．

一方，use-rule の中で前件部が成立すると後件部と確信度（前件部確信度とルール確信度を乗じたもの）が conclude に渡されるが，そこでは update-cf によって確信度が積算される．update-cf では cf-or でそれまでにキャッシュされている確信度と引数でもらった確信度をOR結合として計算される．すなわち，ルール間では同一の結論を導くルールはOR結合となる．これも常識的である．

```
(defun conclude (conclusion cf)
  "Add a conclusion (with specified certainty factor) to DB."
  (multiple-value-bind (parm inst op val)
      (parse-condition conclusion)
    (declare (ignore op))
    (update-cf parm inst val cf)))

(defun update-cf (parm inst val cf)
```

```
  "Change the certianty factor for (parm inst is val),
  by combining the given cf with the old."
  (let ((new-cf (cf-or cf (get-cf parm inst val))))
    (put-db (list parm inst)
            (cons (list val new-cf)
                  (remove val (get-db (list parm inst))
                          :key #'first)))))
```

Mycin 対話インタフェース

24.1節でも述べたように，後向き推論は対話型インタフェース機能を実現するのに適している．Mycin ではさらにコンテキストに対して想定されるパラメータを定義することで，誤入力を防ぐ機能がある．さらに why と入力すると推論に使用中のルールを使って質問の意図を説明する機能もある．それらを一手に引き受ける関数が ask-vals である．

```
(defun ask-vals (parm inst)
  "Ask the user for the value(s) of inst's parm parameter,
  unless this has already been asked.  Keep asking until the
  user types UNKNOWN (return nil) or a valid reply (return t)."
  (unless (get-db `(asked ,parm ,inst))
    (put-db `(asked ,parm ,inst) t)
    (loop
      (let ((ans (prompt-and-read-vals parm inst)))
        (case ans
          (help (format t +help-string+))
          (why  (print-why (get-db 'current-rule) parm))
          (rule (princ (get-db 'current-rule)))
          ((unk unknown) (return nil))
          (?    (format t "~&A ~a must be of type ~a"
                        parm (parm-type parm)) nil)
          (t    (if (check-reply ans parm inst)
                    (return t)
                    (format t "~&Illegal reply.  ~
                             Type ? to see legal ones."))))))))
```

PATIENT の名前以外の場所では，help，why，rule，unknown，？ などをタイプして，ヘルプを示したり，質問の意図を聞いたり，関連ルールを示したりできるし，unk または unknown を入れることで入力しないこともできる．

一方，出力においてはルール出力時に単に確信度を示すだけでなく，「確かな証拠がある」とか「強く示唆する証拠がある」などという言葉をつけてルール出力が行われる．本来は否定的な表現も結論を示すときに言えるのだが，このPAIPのコードでは使う場面がないようである．

```
(defun cf->english (cf)
  "Convert a certainy factor to an English phrase."
  (cond ((= cf  1.0) "there is certain evidence")
        ((> cf   .8) "there is strongly suggestive evidence")
        ((> cf   .5) "there is suggestive evidence")
        ((> cf  0.0) "there is weakly suggestive evidence")
        ((= cf  0.0) "there is NO evidence either way")
        ((< cf  0.0) (concatenate 'string (cf->english (- cf))
                                  " AGAINST the conclusion"))))
```

```
(defun print-rule (rule &optional (stream t) depth)
  (declare (ignore depth))
  (format stream "~&Rule ~a:~&  If" (rule-number rule))
  (print-conditions (rule-premises rule) stream)
  (format stream "~&  Then ~a (~a) that"
          (cf->english (rule-cf rule)) (rule-cf rule))
  (print-conditions (rule-conclusions rule) stream))

(defun report-findings (context inst)
  "Print findings on each goal for this instance."
  (when (context-goals context)
    (format t "~&Findings for ~a:" (inst-name inst))
    (dolist (goal (context-goals context))
      (let ((values (get-vals goal inst)))
        ;; If there are any values for this goal,
        ;; print them sorted by certainty factor.
        (if values
            (format t "~& ~a:~{~{ ~a (~,3f)  ~}~}" goal
                    (sort (copy-list values) #'> :key #'second))
            (format t "~& ~a: unknown" goal))))))
```

問題26.2 find-outやeval-conditionやget-valsにトレースを掛けて，推論の挙動を観察せよ．

問題26.3 感染症の知識は素人にはわからない．自分の専門分野や好きな分野で，エキスパートシステムを作ってみよう．Mycinを使う場合には，エキスパートシステムのタスクがMycinに合致しているかどうかを検討しなければならない．分類問題や診断問題のように，あらかじめゴールやパラメータが列挙できるものであれば，可能性は高い．

問題26.4 Mycin システムでは，コンテキストは推論制御と知識表現の二つの役割がある．推論の順番は初期入力の次に，後向き推論となっている．また，複数の可能性を許容するためにコンテキスト（カテゴリー）と推論の結果としてのコンテキストツリー（実現）となっている．個々のコンテキストカテゴリーはCLOSで言えばクラスに相当するし，その実現はCLOSで言えば実現体に相当する．Mycinのコンテキストを CLOSを使って書き換えることができるだろうか？あるいはフレームを使って書き換えることができるであろうか？フレーム＋ルールベースは知識表現のマルチパラダイムとして一時期もてはやされたものである．

26.3 エキスパートシステムとはなんであったか？

世の中の仕事を概観したとき，それらの多くがアルゴリズム的な手続きで処理されているとか，論理的に処理されているとは思えない．現実にはある課題についてそのすべての関連知識や周辺知識が得られるというものではない．そのような情報が不完全，知識が不完全，解くべき手法があいまいな場合でも，経営者や専門家は最善の手を打つように意思決定しているはずである．すなわち，我々はほとんどヒューリスティックに問題解決をしている．

1980年前後はエキスパートシステム全盛であったが，今日では宇宙ロケット打ち上げ作業への応用のような特殊な例はあるが，産業界においてエキスパートシステムがそれほど用いられていない．その理由はルールベースシステムの保守性の悪さにあった．一般にプログラミングの要諦は，プログラム中の制御構造にある．ルールベースシステムではプログラム制御に相当するものはルール発火の順番であるが，この見通し

の悪さが保守性を阻害している．筆者の経験では，制御用ファジールールは別として，ルールベースシステムはルール数が300程度でなんとか実用化が見えてくる．そしてこれくらいまでなら人手で保守可能であるが，これが1000以上になると，問題領域のすべての問題について正しい結論が得られるように，ルール集合全体を整合性よく保守していくのが困難になる．プログラムのデバッグは1万行を超えると大変困難になるが，同様にルールのデバッグは千行を超えると困難になる．そしてエキスパートシステム作成にかかわったエキスパートが退職や配置転換などでいなくなると，新しい知識の追加などができなくなり，しだいにエキスパートシステムの有効性が下がっていって，ついには使われなくなってしまう．特に故障診断のように普段はあまり使われないようなシステムの場合にこれが著しい．

エキスパートシステムの研究は知識表現問題を進化させて，人工知能研究者は，属人的ではないより本質的な知識を問題とするようになり，今では世界の本質的な知識を記述するオントロジーという一分野を形成することになった．しかし，IF ～ THEN ～ という知識表現形式は，IT専門家ではない一般の技術者にも，読むだけで理解可能という利点があり，計算機の計算性能も昔とは比較にならないほど向上している．AI専門家ではない現場の技術者が比較的小規模のルールベースシステムを，現場のPC上で自家薬籠中の物として使う場面がもっとあってもよいと思われる．

第27章 事例ベース推論

> 事例ベース推論は人間の推論がどのように働くのかの本質なのだ.
> [Riesbeck/Schank, 1989] (Inside Case-based Reasoning)

> 人間の脳は蓄積した記憶を使って，見たり，聞いたり，触れたりするものすべてを，絶えず予測しているのだ.
> [Haukins/Blakeslee, 2004] (On Intelligence，(伊藤訳))

> 私の考えでは，視覚に取って最も重要な認知構造は，予期図式とも呼ばれるもので，それは，他の情報に比べてある特定の情報を選択的に受け入れ，それによって見る活動をコントロールする，いわば準備状態を意味している.
> [Neisser, 1976] (Cognition and Reality，(古崎・村瀬共訳))

27.1 事例とは何か？

事例ベース推論とは「過去の類似問題の解法に基づいて類推して新たな問題を解く推論の手法」(Wikipedia, 事例ベース推論)とされている．この説明が間違いというわけではないが，事例とは何かが説明されないと本当には分かった気になれない．Schankらの事例ベース推論では，**事例とは属性とその値の束**のことである．属性と言ってもいろいろあるが，属性・値ペアの集合が何種類か複数あったとき，異なる事例はその属性・値ペアの違いにより識別される．属性値が具体的な事実データ（すなわち個物）の集合体は，具体的な事例を表現する．個々の具体的な事例はその抽象概念を持つが，その事例の抽象概念の属性値は，具体的な属性値の抽象概念であるのが普通である．また，抽象的な事例自身もさらにより抽象的な概念の特殊化されたものであり，抽象的概念の属性値と特殊な概念の属性値を比較すれば，抽象的概念の属性値の方が特殊な概念の属性値よりも抽象的であるのが通例である．すなわち，抽象的な属性値は抽象的な事例（概念）を構成し，特殊な属性値は特殊な事例（概念）を構成する．事例とその事例を構成する属性値のそれぞれの抽象・特殊・具体の関係は並行的な階層構造を形成する.

例えばポチが溺れている飼い主太郎を助けるという具体的な事例があったとき，その事例の「行為者」は「ポチ」であり、「行為」は「溺れている太郎を助ける」であり，「対象」は「太郎」であるが，その抽象概念では「行為者」は「飼い犬」であり，「行為」は「溺れている飼い主を助ける」であり，「対象」は「飼い主」である．さらにその抽象概念では「行為者」は「ペット動物」，「行為」は「溺れている人を助ける」であり，「対象」は「人間」でありえる.

抽象的属性・値ペアの集合は，抽象的な事例を表現するが，少ない属性・値ペアの集合は多い属性・値ペアより抽象的である．たとえば、抽象事例「救助」では「行為者」＝「エージェント」，「行為」＝「対象を行為者が助ける」，「対象」＝「エージェント」であるが，「水難救助」では「行為者」＝「エージェント」，「行為」＝「水に溺れた対象を行為者が助ける」，「対象」＝「エージェント」であって，さらに「場所」として「海 or 川 or 池」という属性が付加されているであろう．すなわち，特殊であればあるほど，属性は多くなり，属性値は特殊になるのが通常である.

27.2 メモリ機構

事例ベース推論システムの具体的な実現方法は様々であるが，Schankらの研究ではすべての事例ベース推論でMemory Organization Package (MOP)がその実現の基盤として用いられている．MOP機構では，属性名(ロール)とその値(フィラー)のペアをスロットと呼び，ある事例はスロット集合を有する一つのフレーム（これもmopと呼ばれる）として表現される．ただし，MOP機構においては原則としてフィラーもmopである．したがって，フィラーにも階層構造がありえる．

メモリ中に存在する mop に関する概念階層構造は，mop の直接上位概念(direct abstraction)と直接下位概念(direct special)で記述される．ある mop の直接上位概念の mop もまたその直接上位概念 mop を有しているが，もとの mop はその直接上位概念 mop を経由して，間接的にそのより上位の概念 mop につながっている．このことは下位概念の方向についても同様である．ある mop のすべての抽象概念 (all abstractions) とは，直接間接を問わず，上位概念のリンクをたどってすべての上位概念を集めたものである．これは下位概念についても同様であり，mop の特殊概念は直接間接を問わず，すべての下位概念を集めたものである．

スロットを有しないが概念階層は有するような mop を原子 mop (atomic mop) と呼ぶことにする．概念階層に加えてスロットもあって，フィラーもフレームmopであるようなmopを複合mop (composed mop) と呼ぶことにする．複合mopのスロットのフィラーは原子mopでも複合mopでもよいことを注意しておく．事例ベース推論において，概念階層がなくてスロット情報だけがあるようなmopは，メモリ中に保存されるべき場所を探している状態である．

MOPシステムではメモリ中の概念構造が全体的に整合的であることが重要である．すなわち，ある複合mopのすべての上位概念mopにおいて，それら上位概念のフィラーmopがあったとき，そのフィラーmopは下位複合mopの同属性のフィラーmopに等しいか抽象であるとき，その複合mopの階層構造は整合的であると言い，メモリ中のすべての複合mopが整合的であるときに，そのメモリは整合的であると言う．

事例ベースシステムにおけるメモリ機構の仕事は，事例を既存の階層構造に対して正しく蓄えたり取り出したりすることであるが，MOPシステムの最大の特徴は，スロット集合を入力したときそれを有する新しい事例mopを自動生成し，メモリ中の抽象・特殊の階層関係中の適切な位置にそのmopを位置づけることである．すなわち，メモリ中にフィラーmopの概念階層が定義されていて，スロット中のフィラーmopの概念階層に整合的な事例が抽象的あるいは具体的な過去の事例として存在している状態で，新たな事例がスロットの集合としてシステムに入力されたとき，システムは入力されたスロット集合と同様なスロットを持つmopをメモリ中に探索し，発見された類似のmopを返したり，さもなければメモリ中の全体階層関係のなかで，新しいmopを適切な場所に位置づけることができる．事例においてその属性と値のペアはその事例の内包に相当するが，一方で事例の階層関係は事例の外延関係を示す．MOPでは**内包が外延を決定する**.

MOPシステム

Riesbeck and Schank の *Inside Case-based Reasoning* にて説明されるMOPシステムでは，古いLispシステムとの互換性を重視して Lisp のごく基本的な機能のみで実装され，シンボルの属性リスト(property list)を利用してkey-valueのテーブルとしたものである．

Inside Case-based Reasoning 出版時の1989年にはすでに *CLtL1* は出版済みであったが(1984)，著者らはきっと前書である *Inside Computer Understanding* 出版の成功を見て，前書と同様に当時のパソコンでも動くことを狙ってそのようにしたのであろう．ここではマクロのプログラム典型例としての紹介を兼ねて，オリジナルのコードを示す．

しかし，その後に出版された Schank, Kass, and Riesbeck *Inside Case-Based Explanation* では，Lisp構造体を用いてフレームとしてmopが実装されたが，今ならきっと CLOS を用いて実装されるであろう．本書で提供するMOPシステムでは，古いRiesbeck and Schank のコードを利用した．すなわち，一つのmop実装は以下のデータを有している．ここで，mopの `absts` や `slots` は，CLOSにおけるsuper-classやスロットと類似のものであるが，実装は全く別物であることを注意しておく．

シンボルの属性リストを利用したmopの実現

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (defmacro define-table (fn vars place)
    (let ((key (car vars))
          (set-fn (gentemp "set-fn."))
          (val (gentemp "val.")))
      `(progn (defun ,fn (,key) (getf ,place ,key))
         (defun ,set-fn (,key ,val)
           (setf (getf ,place ,key) ,val))
         (defsetf ,fn ,set-fn)
         ',fn)))
  )

(defun delete-key (table key)
  (remf table key) table)

(defun table-keys (table)
  (and table
       (cons (car table)
             (table-keys (cddr table)))))

(defvar *mop-tables* nil)
(define-table mop-table (table-name) *mop-tables*)

(define-table mop-absts (mop) (mop-table 'absts))
(define-table mop-all-absts (mop) (mop-table 'all-absts))
(define-table mop-specs (mop) (mop-table 'specs))
(define-table mop-slots (mop) (mop-table 'slots))
(define-table mop-type (mop) (mop-table 'type))
```

テーブルの意味は次のとおり．

1. absts　　直接抽象mopリスト，属性継承関係で直接上位 mop（名）のリスト
2. all-absts　すべての抽象mopのリスト，属性継承関係で直接・間接に上位 mop（名）のリスト

3. specs　　直接特殊mopリスト，属性継承関係で直接下位 mop（名）のリスト
4. slots　　スロットリスト，スロット（属性名とその値のペア）が並んだリスト
5. type　　抽象 mop か具体 mop かの情報を置く

上記のコードだけから，mop-absts，mop-all-absts，mop-specs，mop-slots，mop-typeという関数が次のように定義される．

mop-absts *mop* [関数]

mop の直接上位 mop を取り出したり，setf でセットしたりする．

mop-all-absts *mop* [関数]

mop のすべての上位 mop を取り出す．

mop-specs *mop* [関数]

mop の直接下位 mop を取り出す．

mop-slots *mop* [関数]

*mop*のスロットを取り出したり，setf でセットしたりする．

mop-type *mop* [関数]

mop を抽象 mop (mop) あるいは具体 mop (instance) とする．

第25章で示したフレームシステムの例を，もしmopとして記述するならば，次のように書くことはできる．

```
(progn
(setf (mop-type 'thing) 'mop)
(setf (mop-type 'dwelling) 'mop)
(setf (mop-absts 'dwelling) 'thing)
(setf (mop-type 'apartment) 'mop)
(setf (mop-absts 'apartment)'dwelling)
(setf (mop-type 'apt-at-100-york) 'mop)
(setf (mop-absts 'apt-at-100-york) 'apartment)
(setf (mop-slots 'apt-at-100-york) '((street-name york)))
(setf (mop-slots 'apt-at-100-york) '((street-number 100)))
(setf (mop-slots 'apt-at-100-york) '((wall-color white)))
(setf (mop-slots 'apt-at-100-york) '((floor-surface wood)))
(setf (mop-type 'apt1) 'instance)
(setf (mop-absts 'apt1) 'apt-at-100-york)
(setf (mop-slots 'apt1) '((number-of-rooms 3)))
)
```

mop-absts には直接上位の抽象 mop を定義するが，すると後々システムが mop-specs と mop-all-absts の面倒を見てくれる．

ここで，mop-typeには mop か instance しか書いてはいけないことに注意されたい．mop-type の値が mop の場合は抽象 mop であり，クラスに相当するが，instance の場合は具体 mop である．しかし，CLOS の場合とは異なってクラスとインスタンスの実装の構造には何の違いもない．正確に言えば，define-table の実態は構造体やCLOSオブジェクトではなく，ただのシンボルの属性リストだが，それをフレームか CLOS のように見せかけているだけなのだ．なお，数は具体mopとして扱う．

クラスベース型のオブジェクトシステムであるCLOSでは，クラスとインスタンスは明確に分離され，CLOSインスタンスを定義するためには，あらかじめそのインスタンスのクラスが定義済みでなければならない．しかし，このMOPシステムでは，抽象mop定義なしで具体mopを定義できる．本システムは全体的にはプロトタイプ型のオブジェクトシステムに近いが，抽象クラスとしての mop とインスタンスとしての mop は認識され区別される．このシステムでは mop フレーム中の type 属性に値として instance があれば具体 mop，そうでなければ抽象 mop としている．なお，数は具体 mop の一種として扱われることを注意しておく．ただし，nil は mop ではない．

```
(defun mopp (x)
  (or (numberp x) (and (symbolp x) (mop-type x))))

(defun instance-mopp (x)
  (and (mopp x)
       (or (numberp x)
           (eql (mop-type x) 'instance))))

(defun abst-mopp (x)
  (and (mopp x)
       (eql (mop-type x) 'mop)))
```

一つのスロットは以下に示すように，属性を著すロール（シンボル）とその値（フィラー）がペアになったリストであり，関数 role-filler によってロールを指定してそのロールの値を取り出し，関数 add-role-filler によってmopのロールに値（フィラー）をセットする．ただし，この実装ではすでに該当するロールについてフィラーが定義されていた時にはエラーとなる．すなわち，add-role-filler で同一のロールについて複数のフィラーを持つことはできない．本MOPシステムではフィラーの個数は一個だけとしている．複数のフィラーを与える時には，後述の group mop や sequence mop を用いる．

```
(defun slot-role (slot) (car slot))
(defun slot-filler (slot) (cadr slot))
(defun make-slot (role mop) (list role mop))

(defun role-slot (role x)
  (assert (or (mopp x) (listp x)) ()
          "ROLE-SLOT: slot ~S must be a list or a mop." x)
  (assoc role
         (cond ((mopp x) (mop-slots x))
               (t x))))

(defun role-filler (role x)
  (slot-filler (role-slot role x)))
```

```
(defun add-role-filler (role mop filler)
  (assert (mopp mop) () "ADD-ROLE-FILLER: ~S given as mop." mop)
  (assert (null (role-filler role mop)) ()
    "ADD-ROLE-FILLER: mop ~S has already a filler of role ~S."
          mop role)
  (format t "~&~S:~S <= ~S" mop role filler)
  (setf (mop-slots mop)
    (cons (make-slot role filler)
          (mop-slots mop)))
  filler)
```

role-filler *role* ***x*** [関数]

*x*は mop あるいはスロット集合．この関数は *x* から *role* の属性値を取り出す．

add-role-filler *role* ***mop*** ***filler*** [関数]

mop の *role* に属性値として *filler* をセットする．

mop-all-absts に値をセットするのは calc-all-absts と redo-all-absts の働きによる．関数 new-mop（後述）で新しくmop を作るたびに link-abst でその直接上位 mop へのリンクが張られるが，そのたびに redo-all-absts が自動的に呼ばれるので，ユーザが直接これらの関数を使うことはない．

```
(defun calc-all-absts (mop)
  "calculates a list of all the abstractions for mop.
   We assume that all the abstractions are a correct for
   the immediate abstractions of mop, so all we have to
   do is get those abstractions, with mop-all-absts,
   add mop itself to the front of the list, and
   remove duplicates."
  (remove-duplicates
   (cons mop
         (loop for abst in (mop-absts mop)
             append (mop-all-absts abst)))))

(defun redo-all-absts (mop)
  (setf (mop-all-absts mop) (calc-all-absts mop))
  (loop for spec in (mop-specs mop)
      do (redo-all-absts spec)))

(defun link-abst (spec abst)
  (assert (abst-mopp abst) ()
          "LINK-ABST: abst ~S must be an abstraction." abst)
  (assert (mopp spec) ()
          "LINK-ABST: spec ~S must be a mop." spec)
  (assert (not (abstp spec abst)) ()
          "LINK-ABST: ~S is already linked to abst ~S."
          spec abst)
  (cond ((not (abstp abst spec))
         (setf (mop-absts spec)
           (cons abst (mop-absts spec)))
         (setf (mop-specs abst)
           (cons spec (mop-specs abst)))
         (redo-all-absts spec)))
  spec)
```

```
(defun unlink-abst (spec abst)
  "removes the abstraction link between the two mops, if any,
   updates the abstraction hierarchy, and returns spec."
  (cond ((abstp abst spec)
         (setf (mop-absts spec)
           (remove abst (mop-absts spec)))
         (setf (mop-specs abst)
           (remove spec (mop-specs abst)))
         (redo-all-absts spec)))
  spec)
```

link-abst は与えられた引数の spec と abst の間で，spec の mop-absts に abst を追加し，abst の mop-specs に spec を追加する．これで引数の spec と abst の間で，リンクが確立されるが，さらに spec の all-absts を再設定する．

unlink-abst は与えられた引数の spec と abst の間で，もしリンクがあればそれを削除し，さらに spec の all-absts を再設定する．

抽象関係を判定する

MOPシステムで最も頻繁に呼び出される基礎的な関数は，二つの mop の抽象関係を判定する abstp であろう．この関数は下位の mop フレームの all-absts リストの中に上位の mop が存在すれば，真を返す．ただし，abstp は mop 以外のシンボルも引き受けることができて，両者が同じであれば真を返す．後述の自然言語意味解析システムDMAPでは，解析対象になるテキスト中の単語（実はLispシンボル）が mop と区別なく扱われるため，このようになっている．

しかし，明示的なmop抽象関係だけからではなく，二つの mop のスロットの構成から抽象関係を判定する機能もある．ここでスロット集合について抽象関係を判定する slots-abstp についてその詳細を記述する．slots-abstp は、あるスロット集合について、引数に与えられた抽象mopをそのスロット集合の抽象とするにふさわしいかどうかを判別するために用いられるものである．すなわち，与えられた抽象mopのスロットがスロット集合の制約となり，その抽象mopのすべての制約がスロット集合中のスロットによって満たされ(satisfiedp)なければならない．すなわち，抽象mopのフィラーがスロット集合の該当スロットフィラーの抽象であるか，あるいはさらに再帰的に slots-abstp を満足するかである．つまり，抽象側のすべてのフィラーが制約となって，特殊側のフィラーの抽象でなければならない．ただし，もし抽象mop側のフィラーがインスタンスであれば，スロット集合側に該当スロットフィラーが**存在してはならない**.

最後に，パターンmopとは，m-pattern を抽象にもつような mop のことであるが，もし slots-abstp の引数に与えられた抽象mopがパターンmopのスロットを有するならば，抽象関係を判定する手続きがあたえられているということであり，スロット集合の該当スロットフィラーを判定するために，関数 satisfiedp はそのパターンmopから abst-fn 関数を取りだして，それを稼働する．これは一種のデーモンである．結果が真であるときのみ制約が満たされ slots-abstp も真となる．なお slots-abstp では slots 集合側のスロットに制約と関係のない余分のスロットがあってもよいことを注意しておく．

```
(defun abstp (abst spec)
  (or (eql abst spec)
      (member abst (mop-all-absts spec))))

(defun slots-abstp (mop slots)
  (and (abst-mopp mop)
       (not (null (mop-slots mop)))
       (loop for slot in (mop-slots mop)
           always (satisfiedp (slot-filler slot)
                              (get-filler (slot-role slot)
                                          slots)
                              slots))))

(defun satisfiedp (constraint filler slots)
  (cond ((null constraint))
        ((patternp constraint)
         (funcall (inherit-filler 'abst-fn constraint)
                  constraint filler slots))
        ((abstp constraint filler))
        ((instance-mopp constraint) (null filler))
        (filler (slots-abstp constraint filler))
        (t nil)))

(defun patternp (x) (abstp 'm-pattern x))
```

mop 中の all-absts には，その mop の直接上位概念mopから上位をたどって，到達するすべての抽象mopが置かれなければならない（そのmop自身も含められる）．関数 calc-all-absts がすべての上位抽象mopを拾い集めてくるのに用いられるが，直接上位のすべての抽象mopについて all-absts が正しく計算されていれば直接上位のそれらを集めるだけでよい．しかし結果的に上位抽象mopがどのように all-absts として並ぶのかについては，曖昧さが残されている．CLOSのように直接上位のクラスの間でも上下関係があるとは考えない方がよさそうだ．CLOSではトポロジカル・ソートの採用によってそれが可能となっている．

新しい mop を生成する

引数として与えられた直接上位概念 absts とスロット集合 slots で，とりあえず新しい mop を生成するのが関数 new-mop である．ただし，得られた mop は，その後，さらにそのときの概念階層の中で詳細化(refine)されたり，すでに全く同一の情報を持つ mop がある場合には作られたmop が削除されたりすることを注意しておく．名前は指定しなくてもよく，その場合にはシステムが自動的に唯一の名前を生成してつける．また抽象mop か具体mop かも指定しなくてもよく，その場合にはシステムが抽象mop か具体mop かを判別しタイプ付けする．

原子mopを積極的にインスタンスと定義する場合を除いて，mop定義時にそれを抽象mopとすべきかそれともインスタンスmopとすべきかは，absts と slots の内容から calc-type によって自動的に計算される．すなわち，absts の中に一つでもパターンmopがあれば，それは抽象であり，あるいは slots のなかに一つでも抽象mopがあれば，それでも抽象となる．すなわち，抽象mopをフィラーに持つ複合mopは抽象である．また，absts にパターンmopがなく，スロットがない（すなわち原子mop）場合も，それはデフォールトと

して，抽象となる．それ以外の場合には，インスタンスである．すなわち，具体mopは具体的なデータを豊富に持つであろうが，抽象mopはそうでない場合が多いということだ．

名前が指定されなかった場合には，spec-nameが名前を計算する．すなわち，抽象mopであれば，上位抽象リストのうちの最初のものの名前を使って，その末尾に数字を付けた名前にする．具体mopならさらに頭に"I-"を追加する．

```
(defun calc-type (absts slots)
  (or (loop for abst in absts
          when (patternp abst)
          return 'mop)
      (and (null slots) 'mop)
      (loop for slot in slots
          when (not (instance-mopp (slot-filler slot)))
          return 'mop)
      'instance))

(defun spec-name (absts type)
  (gentemp (format nil (cond ((eql type 'mop) "~S.")
                             (t "I-~S."))
             (car absts))))

(defun new-mop (name absts type slots)
  (assert (symbolp name) ()
          "NEW-MOP: name ~S must be a lisp symbol." name)
  (assert (loop for abst in absts
            always
            (mopp abst)) ()
            "NEW-MOP: one of absts is not a mop.")
  (or type (setf type (calc-type absts slots)))
  (or name (setf name (spec-name absts type)))
  (setf (mop-type name) type)
  (and slots (setf (mop-slots name) slots))
  (loop for abst in absts do (link-abst name abst))
  name)
```

absts のリストの各要素は抽象mop名でなくてはならない．もし absts にインスタンスmop あるいは数字が含まれていれば，ただちにエラーとなる．もしmop名を指定して，同名のmopが定義済みであれば，この関数は強制的に再定義してしまう．最後に直接上位概念mopとのリンクを確立し，さらにすべての上位概念mopを計算して結果をmop-all-absts に保存する．

Slotの束からmopを作る

実際に slot の束から mop を作るのは，slots->mop である．この関数は new-mop を用いて mop を作りだしたのち， install-instance あるいは install-abstraction により，メモリ中の適切な場所にそれをインストールする．引数 slots はスロット書式のリストであるが，ただし先頭の第 1 要素がアトムの場合にはそれはシンボル instance か mop でなければならず，これは new-mop でmopのタイプを指定するために用いられる．

slots->mop における一つのスロットは (<ロール>　<x>) という書式であるが，forms->slots という関数ではそのようなスロット形式以外に (<ロール>　<抽象mop名>　{<スロット書式>}$^+$) という書式も引き受けることができて，それから (<ロール> <インラインmop>)というスロットを作りだす．このインラインmop の作成に，与えられた<抽象mop名>と<スロット書式>のリストが用いられる．このように一つのスロット書式にはその中で別のスロット書式があってよいが，これはこのようなカスケードな形でフィラーmopをトップレベルではなくインラインで生成するのに便利な書式である．ここで<ロール>はLispシンボル，<x>はmop名（シンボル），または数字であり，数字の場合にはすぐに (<ロール> <x>) が返される．

```
(defun slots->mop (slots absts must-work)
  (assert (and (not (null absts))
               (loop for abst in absts always (mopp abst)))
          () "SLOTS->MOP: Illegal absts given ~S." absts)
  (or (and (null slots) (null (cdr absts)) (car absts))
      (let ((type (and slots (atom (car slots)) (car slots))))
        (and type (setf slots (cdr slots)))
        (let ((mop (new-mop nil absts type slots)))
          (let ((result
                 (cond ((instance-mopp mop)
                        (install-instance mop))
                       (t (install-abstraction mop)))))
            (assert (or result (null must-work)) ()
     "SLOTS->MOP: null result cannot be accepted in this case.")
            (unless result
             (warn "Null result is obtained in SLOTS->MOP."))
            result)))))

(defun forms->slots (slot-forms)
  (loop for slot-form in slot-forms
      collect
        (cond ((atom slot-form) slot-form)
              (t (make-slot (slot-role slot-form)
                            (let ((abst (cadr slot-form)))
                              (assert (atom abst) () "FORMS->SLOTS: illegal abst ~S." abst)
                              (and abst
                                   (slots->mop
                                    (forms->slots (cddr slot-form))
                                    (list abst)
                                    t)))))))
```

mop をインストールするとはどういうことか？

forms->slots や slots->mop を用いた場合，そこで生成されるmopは，install-instance や install-abstraction によってメモリ中にインストールされる．そのアルゴリズムは具体mopと抽象mopで異なっている．

具体mopの場合には，それは合法的な抽象mopにリンクされなければならない．合法的な抽象mopとは，そこに具体mopがリンクされることを許すようなもので，この実装においては，それは原子mopではなくかつ特殊mopを持たないような抽象mopである．すなわち，具体mopはスロットのある概念階層の末端抽象mopにしか付かない．一方，抽象mopのインストールではそのような制限はなく，new-mop で指定された

absts の下位に直接インストールされる．ただし，その場合でも兄弟(直接上位概念を共有するmop)に全く同一のスロット集合を持つようなもの(これを双子と呼ぶ)がたまたまあれば，双子の古いほうが優先され，新しく生成されたmopは捨てられる．

関数 get-twin は抽象・具体を問わず，あるmopの双子があればそれを返すが，同一性の判定には二つのmopのスロット集合の包含関係を調べる mop-includesp が用いられる．get-filler は継承関係も含めて mop のロールのフィラーを取りだす関数であるが，その詳細は後述する．

```
(defun mop-includesp (mop1 mop2)
  (and (eql (mop-type mop1) (mop-type mop2))
       (loop for slot in (mop-slots mop2)
           always (eql (slot-filler slot)
                       (get-filler (slot-role slot)
                                   mop1)))
       mop1))

(defun mop-equalp (mop1 mop2)
  (and (mop-includesp mop2 mop1)
       (mop-includesp mop1 mop2)))

(defun get-twin (mop)
  (loop for abst in (mop-absts mop)
      thereis (loop for spec in (mop-specs abst)
                  thereis (when (not (eql spec mop))
                            (mop-equalp spec mop)))))
```

install-instance では生成時に指定された absts がそのまま使われるのでなく，refine-instance によって，生成時の absts を出発点として，抽象概念階層の下のほうへ下っていき，最後に落ち着いた場所でその具体mopをインストールしようとする．その場所で双子が見つかれば，新しい方を破棄して古い双子を返す．さもなければ，その場所が合法的かどうかが調べられ，合法的でなければそれは破棄されて nil が返される．合法的であるかどうかは has-legal-absts-p で調べられるが，その中で非合法な abst が見つかれば，それへのリンクが外されることに注意されたい．

```
(defun legal-abstp (abst instance)
  (declare (ignore instance))
  (and (mop-slots abst)
       (loop for spec in (mop-specs abst)
           always (instance-mopp spec))))

(defun has-legal-absts-p (instance)
  (loop for abst in (mop-absts instance)
      when (not (legal-abstp abst instance))
      do (unlink-abst instance abst))
  (mop-absts instance))

(defun install-instance (instance)
  (refine-instance instance)
  (let ((twin (get-twin instance)))
    (cond (twin (remove-mop instance) twin)
          ((has-legal-absts-p instance) instance)
          (t (remove-mop instance) nil))))
```

一方，install-abstraction では refine されることなく，その場にインストールしようとする．双子があれば，新しい方を削除して古い双子を返す．さもなければ，その概念階層の指定場所に新しいmopがインストールされるが，その結果，兄弟に具体mopがあって，かつ新しいmop の下にきてよい場合には reindex-siblings の中でそれらの具体mopを新しいmopの下につけ直す．

```
(defun reindex-siblings (mop)
  (loop for abst in (mop-absts mop)
      do (loop for spec in (mop-specs abst)
             when (and (instance-mopp spec)
                       (slots-abstp mop spec))
             do (unlink-abst spec abst)
               (link-abst spec mop)))
  mop)

(defun install-abstraction (mop)
  (let ((twin (get-twin mop)))
    (cond (twin (remove-mop mop) twin)
          (t (reindex-siblings mop)))))
```

refine-instance はその absts を先頭から順にmops-abstpで調べて，その特殊mop（すなわち兄弟）に付けられるのがあればリンクをそこに付け替える．そして再度自己再帰するが，ここは深さ優先探索をしている．また深さ優先探索が終われば次のabstについて調べられるので，結局，深さ優先ではあるが幅方向にもすべてチェックされることも分かる．実際のリンクは mops-abstp の中で実行され，unlink は mops-abstp の外で実行されることに注意されたい．mops-abstp の中では先に述べた slots-abstp が用いられる．

```
(defun mops-abstp (mops instance)
  (not (null (loop for mop in mops
                when (slots-abstp mop instance)
                collect (link-abst instance mop)))))

(defun refine-instance (instance)
  (loop for abst in (mop-absts instance)
      thereis
        (when (mops-abstp (mop-specs abst) instance)
          (unlink-abst instance abst)
          (refine-instance instance))))
```

mop 定義マクロ

プログラムコードとして mop 定義をするには defmop を用いる．特に具体mopと分かっているときには，definstance を用いる．defmop の呼び出し形式は以下のとおりである．

```
(defmop <mop名> ({<抽象mop名>}*) [ mop | instance ]
  { <スロット書式> }*
  )
```

三番目の引数にはシンボル「mop」か「instance」を指定するが，前者の場合は抽象mop定義を，後者の場合は具体mop定義をしたことになる．省略した場合には，システムが抽象か具体かを計算する．defmop 本体にはスロット書式が並ぶがこれは次のいずれかの形式である．

```
<スロット書式> := (<ロール名> [ <mop名> | <数> ])
                | (<ロール名> <抽象mop名> {<スロット書式>}+ )
```

スロットフィラーには，mop名（シンボル）か，数が来るが，さらにmopをシリアライズしたような，フィラーmopを構成する抽象mop名とスロット書式の並びが来てもよい．その場合には，その抽象mopとスロット書式並びから，フィラーmopが生成される．このとき defmop の場合と異なって，インラインでは抽象mopを一つしか指定できないことに注意されたい．

```
(defmacro defmop (name absts &rest args)
  (let ((type (and args (atom (car args)) (car args))))
    (let ((slot-forms (cond (type (cdr args))
                            (t args))))
      `(new-mop ',name ',absts ',type
                (forms->slots ',slot-forms)))))

(defun remove-mop (name)
  (loop for abst in (mop-absts name)
      do (unlink-abst name abst))
  (loop for table-name in (table-keys *mop-tables*)
      do (setf (mop-table table-name)
           (delete-key (mop-table table-name)
                       name))))
```

definstance は特に上記 defmop + instance で具体mopを定義するかわりに，definstance 一言で具体mopを定義するためのものである．

```
(defun add-instance (name absts slots)
  "creates an instance mop with the given <name>,
  <abstractions>, and <slots>.  This returns <name>."
  (new-mop name absts 'instance slots))

(defmacro definstance (name absts &rest args)
  "DEFINSTANCE <name> <abstractions> [ (<role> <filler>) ... ]
   creates an instance mop with the given <name>, direct
   <abstractions>, and slots which are a list of pairs of
   <role> and <filler>. <filler> is an atom, a list of slot
   forms, or a filler form which is a list containing an
   abstract mop, an optional mop type (mop or instance),
   and zero or more slot forms. A slot form is a list of a
   role and a filler form."
  `(progn
     (assert (loop for abst in ',absts always
                   (abst-mopp abst)) ()
       "DEFINSTANCE: some of absts ~S is not an abstraction."
       ',absts)
     (add-instance ',name ',absts ',args)
     ',name))
```

属性とその値の継承

CLOSの場合と同様に，MOPシステムでも下位抽象mopは上位抽象mopに定義された属性とその値を引き継ぐことができる．それを行うのが inherit-filler である．ところが，ここでもその意味論はCLOSの場合と大いに異なる．MOPシステムでは標準的にはフィラー自身もmopであることを前提として，事例mopの上位に

向かって継承する属性値の全探索が行われる．ただし，get-filler では role-filler ですぐに属性値が見つかった場合にはそれを返し，継承属性値の探索は行われないことを注意しておく．なお，get-filler ではフレームの if-needed デーモンと同じものがある．すなわち，継承で得られたフィラーが m-function の特殊で calc-fun 属性値があればそれは if-needed と同様に値を計算する手続きであり，それを m-function の特殊mop（フィラー）ともともとの mop を引数にして呼び出す，得られた値は add-role-filler にて記憶される．

```
(defun inherit-filler (role mop)
  (loop for abst in (mop-all-absts mop)
      thereis (role-filler role abst)))

(defun get-filler (role mop)
  (or (role-filler role mop)
      (let ((filler (inherit-filler role mop)))
        (and filler
             (or (and (instance-mopp filler) filler)
                 (and (abstp 'm-function filler) filler)
                 (let ((fn (get-filler 'calc-fn filler)))
                   (and fn
                        (let ((new-filler
                               (funcall fn filler mop)))
                          (and new-filler
                               (add-role-filler
                                 role mop
                                 new-filler)))))))))
```

get-fillerで働くデーモン機能をプログラムするためには，以下のようにする． m-patternの特殊である抽象パターンmopを定義し，そこに(:calc-fn 関数名)というスロットを用意する．その関数名で，第一引数に上記で定義したパターンMOPを，第二引数にget-fillerに与えられたMOPを引き受けるような関数をプログラムする．これがデーモンプログラムとなる．同時に同じ関数名をmop名としてm-functionの特殊な抽象関数mopを定義する。 デーモンを働かせたいスロットを持つ抽象mopを定義して，上記で定義したパターンmopを該当スロットフィラーに持つように定義する．あるいは明示的にパターンmopを定義することなく，直接スロット記述として以下のように書くこともできる．

```
(defmop <mop名> ([<抽象mop名>*]) (<role名> m-pattern (:calc-fn <関数名>)))
```

この場合，このmopが定義されるときにシステムが自動的にパターンmopを定義する．

ロール・チェーン探索

フィラーも mop であることを前提に，ある mop から出発してロールを指定してフィラーを次々とたどることができる．それを行うのが path-filler である．

```
(defun path-filler (path mop)
  (and (loop for role in path
          always (setf mop (get-filler role mop)))
       mop))
```

パターン mop とは何か

上記 path-filler で使っているのは，inherit-filler ではなくて，get-filler である．get-filler ではスロットの継承だけでなく，値が見つからなかったときには関数を起動して値を取ってくるあるいは作りだすということができる．これはフレームシステムにおける if-needed と同様の機能だが，それをもっと一般化したのがパターンmopである．パターンmopの特殊では，何か有益な仕事をさせることができるような関数を設定できる．abst-fn スロットや calc-fn スロットはその例であるが，ほかにも，抽象関係で兄弟にあたるmopをとってくるパターンもある．

すべてのパターンmop のスロットの値にはm-functionの特殊が与えられなければならない．m-functionの特殊はMOPシステムにおける関数であるが，mop名と同じ名前のLisp関数が定義されていなければいけない．例えば abst-fn のパターンであれば，それは引数リストが<制約>，<フィラー>，<スロットリスト>であるような関数の名前である．abst-fn に目的に合致した制約を満たすような関数を定義して，与えればよい．

27.3 カーネルシステムの組込みmopと局所手続き

MOPシステムのビルド過程では，ごく基本的なカーネルの mop が定義される．m-root と呼ばれる mop は，最も抽象的な mop であり，抽象階層の最上位に位置している． m-root の直接抽象は nil である．カーネルには root 以外にも，MOPシステム機能を実現するために設けられたいくつかの基本的な mop がある。これらの基本 mop の抽象階層を以下に示す．

```
(m-root
  (m-group (m-empty-group (I-m-empty-group)))
  (m-sequence (m-empty-sequence (I-empty-sequence)))
  (m-failed-solution)
  (m-case)
  (m-role)
  (m-pattern (m-not)
             (m-pattern.0))
  (m-function (get-sibling)
              (constraint-fn (not-constraint)))
  (m-actor)
  (m-act)
  (m-state)
  (m-event))
```

m-failed-solutionはそこに失敗する解が置かれるmopである．

グループとシーケンス

このMOPシステムではスロットのフィラーは 1 個でなければならない．あるロールに複数のフィラーを与えたいときにはそれらを m-group の，あるいは m-sequence の実現とするしかない．

```
(defmop m-group (m-root))
(defmop m-empty-group (m-group))
(definstance I-m-empty-group (m-empty-group))
```

```
(defun groupp (x) (abstp (mop-of 'm-group) x))
```

例えば，m-motive-group が m-group の特殊だとして，そのとき以下のように数字をロールに使って，複数のフィラーをグループとして定義する．

```
(definstance I-m-group.70 (m-motive-group)
 (1 I-m-retaliation)
 (2 I-m-retaliation)
 (3 I-m-retaliation)
 (4 I-m-retaliation))
```

グループでは構成要素に順序関係はないが，シーケンスには順序関係があり，リストをmopシーケンスにしたり，その逆をしたりする関数が用意されている．

```
(defmop m-sequence (m-root))
(defmop m-empty-sequence (m-sequence))
(definstance I-m-empty-sequence (m-empty-sequence))

(defun sequencep (x)
  "returns true if <mop> is a special mop of m-sequence.
   Sequence mops are used to hold time sequences of MOPs."
  (abstp 'm-sequence x))
```

m-sequenceとインスタンスI-empty-sequenceの間にm-empty-sequenceが定義されているが，それは本システムでは抽象mopと具体mopを兄弟にすることはできないため，ダミーの抽象mop，m-empty-sequenceを置いて、その下に具体mopであるempty-sequenceを置いたものである．これはグループについても同様である．

list->sequence はLispのリストからmopシーケンスを作りだす関数である．逆に，シーケンスmopからその中身のシーケンスをリストとして取りだすには sequence->list を用いる．

```
(defun list->sequence (l)
  "returns a sequence mop with members from <list>.
   The first element of <list> fills the first role,
   the second fills the second role, and so on. If the list
   is empty, the instance mop EMPTY-SEQUENCE is returned."
  (if (null l) 'm-empty-sequence
      (slots->mop
       (loop for x in l
             for i from 1 to (length l)
             collect (make-slot i x))
       '(m-sequence)
       t)))

(defun sequence-size (x)
  "returns the size of the sequence."
  (and (sequencep x) (length (mop-slots x))))

(defun sequence->list (sequence)
  "returns a list of the members of the sequence, or the
   filler of first role, of second role, and so on."
```

```
  (and sequence
       (progn (assert (sequencep sequence) ()
                      "SEQUENCE->LIST: illegal MOP.")
              t)
       (loop for index from 1 to (sequence-size sequence)
             when (role-filler index sequence)
             collect it)))

(defun sequence-member (mop sequence)
  "returns true if <mop> is a member of <sequence>."
  (and (sequencep sequence)
       (loop for slot in (mop-slots sequence)
            thereis (eql (slot-filler slot) mop))))

(defun sequence-splice (new old sequence)
  "returns a new sequence mop with all the elements of
   <sequence>, except that <mop> is replaced with the
   elements of <mop-list>.  Note that a NIL <mop-list>
   returns a sequence with <mop> removed."
  (list->sequence
   (loop for mop in (sequence->list sequence)
        append (cond ((eql mop old) new)
                     (t (list mop))))))

(defun sequence-insert (mop sequence)
  "returns a new sequence mop with all the elements of <sequence> plus <mop>,
   added at the end."
  (cond ((null mop) sequence)
        ((sequence-member mop sequence) sequence)
        (t (list->sequence (append (sequence->list sequence) (list mop))))))
```

制約パターン

m-not はネガティブな制約のためのパターンmopであり、not-constraint はその制約関数である。

```
(defmop m-function (m-root))
(defmop constraint-fn (m-function))

(defmop m-pattern (m-root)
  (abst-fn constraint-fn))
(defmop not-constraint (constraint-fn))

(defun constraint-fn (constraint filler slots)
  (declare (ignore constraint filler slots))
  "always returns true, that is, there is no constraint on
   <filler>."
  t)

(defun not-constraint (constraint filler slots)
  "is true if <filler> does not satisfy the constraint found
   in the <object> slot of <constraint>."
  (assert (not (null filler)) ()
     "NOT-CONSTRAINT: The filler is null.")
  (not (satisfiedp (get-filler 'object constraint)
                   filler slots)))
```

ケース

m-caseの特殊であるmopにはoldスロットがあるが，このoldスロットのフィラーの値として，兄弟mopを取りだすことができる．実際には m-case の特殊はすべて、必要に応じて get-sibling 関数によってoldスロットを継承する．

m-roleは「actor」や「object」のようなロールのための mop である．多くのパターン mop は mop の一部分を参照するためにロールを用いる。厳密に言うとすべてのロールをロール mop として定義すべきであるが、実際には簡単のためにパターン mop 中のフィラーとして用いられるものしか、定義しないであろう．

```
(defmop m-role (m-root))
(defmop m-case (m-root)
  (old m-pattern (calc-fn get-sibling)))

(defmop m-not (m-pattern)
  (abst-fn not-constraint))
(defmop m-failed-solution (m-root))

(defmop get-sibling (m-function))
(defun get-sibling (pattern mop)
  (declare (ignore pattern))
  "finds a sibling of <mop>. This function don't look for instance mops,
   and instances of <m-failed-solution>."
  (loop for abst in (mop-absts mop)
        thereis
        (loop for spec in (mop-specs abst)
              thereis
              (when (and (instance-mopp spec)
                         (not (eql spec mop))
                         (not (abstp 'm-failed-solution spec)))
                spec))))
```

メモリ管理とユーティリティ

以下はメモリを管理するための関数群である．メモリ中にはすべての mop の最高位の抽象として　m-root が存在しなければならない．clear-memory はメモリ中の情報をすべて消去するが、すぐにこの m-root をメモリ中にインストールする．前述のように本システムではいくつかの基本的 mop が存在するが、clear-memory はそれらの mop を定義するために、すべての操作の最初に呼び出される．

```
(defun clear-memory ()
  (setf *mop-tables* nil)
  (new-mop 'm-root nil 'mop nil)
  (setf (mop-all-absts 'm-root)
    (calc-all-absts 'm-root))
  'm-root)

(defun all-mops ()
  (table-keys (mop-table 'type)))

(defun remove-mop (name)
  (loop for abst in (mop-absts name)
      do (unlink-abst name abst))
```

```
  (loop for table-name in (table-keys *mop-tables*)
      do (setf (mop-table table-name)
           (delete-key (mop-table table-name)
                       name))))
```

clear-memory [関数]

メモリ中からすべての mop を消去し、m-root を再インストールする.

remove-mop *mop* [関数]

メモリ中からmopを消去する. *mop*はMOP名でなければならない。該当*mop*のすべての抽象についてリンクをはずし、メモリ中からこの*mop*を削除する.

dah *mop* [関数]

mop の下の特殊をプリントする. この名前は「display abstraction hierarchy」を短くしたものである.

dph *mop* [関数]

mop のスロットをプリントし、ロール・チェーンのようにそのフィラーのスロットをプリントし、等々である。この名前は「display packaging hierarchy」を短くしたものである.

```
(defun dah (mop)
  (pprint (tree->list mop #'specs->list nil)))

(defun dph (mop)
  (pprint (tree->list mop #'slots->forms nil)))

(defun specs->list (mop visited)
  (loop for spec in (mop-specs mop)
      collect (tree->list spec #'specs->list visited)))

(defun slots->forms (mop visited)
  (loop for slot in (mop-slots mop)
      collect (cons (slot-role slot)
                    (mop->form (slot-filler slot)
                               visited))))

(defun mop->form (mop visited)
  (tree->list mop #'slots->forms visited))

(defun tree->list (mop fn visited)
  (cond ((member mop visited) (list mop))
        (t (setf visited (cons mop visited))
           `(,mop ,@(funcall fn mop visited)))))
```

CBRプログラミング例

上位抽象からの情報の継承

関数 get-filler では上位抽象からスロットのフィラーを継承することができる. 以下は Inside Case-Based Reasoning からの料理レシピの例である.

```
cbr(5): (defmop m-cook-act (m-act))
m-cook-act
cbr(6): (definstance I-m-chop (m-cook-act))
I-m-chop
cbr(7): (defmop m-step (m-event) (action m-act))
m-step
cbr(8): (defmop m-cook-step (m-step) (action m-cook-act))
```

```
m-cook-step
cbr(9): (defmop m-ingred (m-root))
m-ingred
cbr(10): (defmop m-large-ingred (m-ingred))
m-large-ingred
cbr(11): (defmop m-chop-step  (m-cook-step)
           (action I-m-chop)
           (object m-large-ingred))
m-chop-step
cbr(12): (definstance I-m-beef (m-large-ingred))
I-m-beef
cbr(13): (slots->mop '((object I-m-beef)) '(m-chop-step) t)
I-m-chop-step.16
cbr(14): (get-filler 'action 'I-m-chop-step.16)
I-m-chop
```

このように，m-root に近い上位の抽象から徐々に特殊に向かって mop 定義をする．m-chop-step には行為スロットと対象スロットがあるが，13行目で m-chop-step の下で (object I-m-beef) というスロットのみの事例を与えたら，それは具体 mop となって，その action スロットの値としては上位抽象にある I-m-chop を継承した．

スロットマッチングによる分類

27.2節で述べたように、関数 slots->mop は与えられたスロット集合を持つ mop を、やはり引数に与えられた抽象 mop の下に作ったあと、スロットマッチングによりこれを階層構造のなるべく下にさげて詳細化する．すなわち、作られた mop の直接抽象をできるかぎりその抽象 mop の特殊に切り換える．ただし，それは事例の結果が具体 mop となる場合だけである．このスロットマッチングの機能を用いて、与えられた色々なスロット集合の具体事例を，様々な特殊 mop の下に分類することができる． たとえば、料理する行動には骨を取ったり、刻んだり、煮たり、揚げたりと色々なものがあるが、料理の手順もそれらの何か一つの行為を行うものとして定義する．スロット集合で与えられたある料理の手順が何の手順に分類されるかは、システムが自動的に行うことができる．

マッチング制御のための制約記述

たとえば、あるスロットの値がある範囲内の時にのみスロットのマッチングを成功させるような機能が欲しいときがある．その場合，次のようなプログラムをする．

```
(defmop range-constraint (constraint-fn)) ; range-constraint is a
(defmop m-range (m-pattern)               ; constraint about m-range.
  (abst-fn range-constraint))

(defun range-constraint (constraint filler slots)
  (declare (ignore slots))
  (and (numberp filler)
       (let ((below (role-filler 'below constraint))
             (above (role-filler 'above constraint)))
         (and (or (null below) (< filler below))
              (or (null above) (< above filler))))))
```

すると、ある数値をとるスロットにおいて次のように抽象 mop を定義しておけば、

```
(defmop mop (abst) (role m-range (below number)))
```

以後、そのスロットがこの制約を満足するものしか、マッチングが行われない．傷害事件を過去の裁判事例から判定するCBR、JUDGEでは、傷害行為の軽重を次のようにして、自動的に分類している．

```
(defmop m-hurt-act (m-fight-act) (severity m-range (below 5)))

(defmop m-wound-act (m-fight-act) (severity m-range (above 4)))
```

これで severity が５以上の傷害行為は m-hurt-act に、４以下の傷害行為は m-wound-act に自動的に分類される．

27.4 判例ベース・システム Judge

裁判の訴訟や判例は，英語でcaseと言うが，英米は過去の判例を重視する判例主義である．ここで紹介するJudgeは過去の判例データをベースに，問題となる裁判の量刑をするシステムである．取り上げられる問題は殺人罪である．

```
(defmop m-crime-type (m-root))            ; There is a m-crime-type.
(definstance I-m-homicide (m-crime-type)) ; homicide is a kind of m-crime-type.
```

登場するのは６人の係争者である．行為者 (m-actor) は MOP システムのビルド時に定義されている．

```
(definstance I-m-Al (m-actor))     ; Al is a m-actor.
(definstance I-m-Chuck (m-actor))  ; Chuck is a m-actor.
(definstance I-m-David (m-actor))  ; David is a m-actor.
(definstance I-m-Randy (m-actor))  ; Randy is a m-actor.
(definstance I-m-Ted (m-actor))    ; Ted is a m-actor.
(definstance I-m-Tim (m-actor))    ; Tim is a m-actor.
```

事件の動機には自己防衛 (self-defence)，報復 (retaliation)，挑発されないのに行われた攻撃 (unprovoked) の三種類があるが，これらのうち自己防衛のみが正当化されるもの (justified)，であるがその他の行為は正当化されないもの (unjustified) である．

```
(defmop m-motive (m-root))               ; There is a m-motive.
(defmop m-justified (m-motive))          ; m-justified is a kind of m-motive.
(defmop m-unjustified (m-motive))        ; m-unjustified is a kind of m-motive.
(definstance I-m-self-defence (m-justified)) ; self-defence is a m-justified.
(definstance I-m-retaliation (m-unjustified)); retaliation is a m-unjustified.
(definstance I-m-unprovoked (m-unjustified)) ; unprovoked is a m-unjustified.
```

障害行為には，深刻さの度合いを示す数値が付与されている．sevirity の１から４までなら，軽い障害行為 (hurt) だが，sevirity の5は重い障害行為 (wound) である．

```
(defmop m-fight-act (m-act) (severity nil))
(defmop m-hurt-act (m-fight-act) (severity m-range (below 5)))
(definstance I-m-slap (m-hurt-act) (severity 1))
(definstance I-m-hit (m-hurt-act) (severity 1))
(definstance I-m-strike (m-hurt-act) (severity 2))
```

```
(definstance I-m-knock-down (m-hurt-act) (severity 3))
(definstance I-m-slash (m-hurt-act) (severity 4))

(defmop m-wound-act (m-fight-act) (severity m-range (above 4)))
(definstance I-m-stab (m-wound-act) (severity 5))
(definstance I-m-shoot (m-wound-act) (severity 5))
(definstance I-m-break-skull (m-wound-act) (severity 5))
```

障害行為の結果の状態にも深刻度がある．すなわち，ナイフで突き刺しても(stab)，切られただけ(cut)の場合もあれば，死んでしまう(dead)場合もある．

```
(defmop m-state (m-root))
(defmop m-phys-state (m-state) (severity nil))
(definstance I-m-bruised (m-phys-state) (severity 1))
(definstance I-m-knocked-down (m-phys-state) (severity 2))
(definstance I-m-cut (m-phys-state) (severity 3))
(definstance I-m-dead (m-phys-state) (severity 5))
```

一連の行為と結果の状態が問題とされるが，そのために動機(motive)，けんかイベント(m-fight-event)，けんか行為の結果(m-fight-outcome)，および拡大（escalation)が時系列的なグループとして組織化される．

```
(defmop m-event-group (m-group) (1 m-event))
(defmop m-outcome-group (m-group) (1 m-outcome))
(defmop m-escalation-group (m-group) (1 m-range))
(defmop m-motive-group (m-group) (1 m-motive))
```

個々のけんかイベントの概念は m-event の特殊であるが，けんか行為(m-fight-act)を属性に持つ．

```
(defmop m-fight-event (m-event)
  (action m-fight-act))
```

個々のけんか行為結果の概念は m-outcome の特殊であるが，上記 m-phys-state と行為者(m-acotr)を属性に持つ．

```
(defmop m-outcome (m-root))
(defmop m-fight-outcome (m-outcome)
  (state m-phys-state)
  (actor m-actor))
```

最初の事例は次のような殺人事件である．

- 被告人: Ted
- 被害者: Al
- 経緯:
  1. Ted が Al を切りつける（１回）-> Al は切られた
  2. Al が Ted を切りつける（１回）-> Ted は切られた
  3. Ted が Al を刺す（繰り返し）　-> Al は死亡した
- 判決: 40年

```
(defparameter *case1*
  '((crime-type I-m-homicide)
    (defendant I-m-Ted) (victim I-m-Al)
    (events m-group
            (1 m-fight-event
```

```
             (action I-m-slash)
             (actor I-m-Ted) (object I-m-Al)
             (freq I-m-once))
            (2 m-fight-event
             (action I-m-slash)
             (actor I-m-Al) (object I-m-Ted)
             (freq I-m-once))
            (3 m-fight-event
             (action I-m-stab)
             (actor I-m-Ted) (object I-m-Al)
             (freq I-m-repeatedly)))
    (outcomes m-group
              (1 m-fight-outcome
               (state I-m-cut) (actor I-m-Al))
              (2 m-fight-outcome
               (state I-m-cut) (actor I-m-Ted))
              (3 m-fight-outcome
               (state I-m-dead) (actor I-m-Al)))
    (sentence 40)))
```

二番目の事例も殺人事件であるが，判決がない．

- 被告人: Randy
- 被害者: Chuck
- 経緯:
  1. Randy が Chuck をぶった（繰り返し）-> Chuck は打撲を負った
  2. Chuck が Randy をぶった（繰り返し）-> Randy は打撲を負った
  3. Randy が Chuck を切りつけた（一回）　-> Chuck は切られた
  4. Chuck が Randy を切りつけた（一回）　-> Randy は切られた
  5. Randy が Chuck を突き刺した（繰り返し）-> Chuck は死んだ

```
(defparameter *case2*
  '((crime-type I-m-homicide)
    (defendant I-m-Randy) (victim I-m-Chuck)
    (events m-group
            (1 m-fight-event
             (action I-m-strike)
             (actor I-m-Randy) (object I-m-Chuck)
             (freq I-m-repeatedly))
            (2 m-fight-event
             (action I-m-strike)
             (actor I-m-Chuck) (object I-m-Randy)
             (freq I-m-repeatedly))
            (3 m-fight-event
             (action I-m-slash)
             (actor I-m-Randy) (object I-m-Chuck)
             (freq I-m-once))
            (4 m-fight-event
             (action I-m-slash)
             (actor I-m-Chuck) (object I-m-Randy)
             (freq I-m-once))
            (5 m-fight-event
             (action I-m-stab)
             (actor I-m-Randy) (object I-m-Chuck)
             (freq I-m-repeatedly)))
    (outcomes m-group
              (1 m-fight-outcome
```

```
                (state I-m-bruised) (actor I-m-Chuck))
               (2 m-fight-outcome
                (state I-m-bruised) (actor I-m-Randy))
               (3 m-fight-outcome
                (state I-m-cut) (actor I-m-Chuck))
               (4 m-fight-outcome
                (state I-m-cut) (actor I-m-Randy))
               (5 m-fight-outcome
                (state I-m-dead) (actor I-m-Chuck)))))
```

三番目の事例も殺人事件であるが，これも判決がない.

- 被告人: Tim
- 被害者: David
- 経緯:
  1. David が Tim をひっぱたいた（何回か）-> Tim はあざができた
  2. Tim が David をぶった（何回か）-> David は打撲を負った
  3. David が Tim を殴り倒した（一回） -> Tim は倒れた
  4. Tim が David を突き刺した（何回か） -> David は死んだ

```
(defparameter *case3*
  '((crime-type I-m-homicide)
    (defendant I-m-Tim) (victim I-m-David)
    (events m-group
            (1 m-fight-event
             (action I-m-slap)
             (actor I-m-David) (object I-m-Tim)
             (freq I-m-several-times))
            (2 m-fight-event
             (action I-m-strike)
             (actor I-m-Tim) (object I-m-David)
             (freq I-m-several-times))
            (3 m-fight-event
             (action I-m-knock-down)
             (actor I-m-David) (object I-m-Tim)
             (freq I-m-once))
            (4 m-fight-event
             (action I-m-stab)
             (actor I-m-Tim) (object I-m-David)
             (freq I-m-several-times)))
    (outcomes m-group
              (1 m-fight-outcome
               (state I-m-bruised) (actor I-m-Tim))
              (2 m-fight-outcome
               (state I-m-bruised) (actor I-m-David))
              (3 m-fight-outcome
               (state I-m-knocked-down) (actor I-m-Tim))
              (4 m-fight-outcome
               (state I-m-dead) (actor I-m-David)))))
```

このシステムがすることは，やられたらやり返すというこの流れにおいて，自己防衛的ではなく，エスカレーションの度合いを測って，量刑を決めることだけであり，殺人の動機とか反省の度合いなどは考慮しない.

`calc-escalaion` は `this-severity` と `prev-severity` を比較して `escalation` の度合いを計算する.

```
(defun calc-escalations (pattern mop)
  "takes the sequence of events in <mop> and calculates a sequence of
   corresponding escalations"
  (declare (ignore pattern))
  (format t "~&---------------")
  (format t "~&Calculating escalations in ~S" mop)
  (list->group
   (let ((prev-severity 0))
     (loop for event in (group->list (role-filler 'events mop))
       collect (let ((this-severity (path-filler '(action severity) event)))
                 (let ((result (- this-severity prev-severity)))
                   (setf prev-severity this-severity)
                   result))))))

(defun calc-motives (pattern mop)
  "takes the sequence of events in <mop> and calculates a sequence of
   corresponding motives."
  (declare (ignore pattern))
  (format t "~&---------------")
  (format t "~&Calculating motives in ~S" mop)
  (list->group
   (let ((prev-motive 0))
     (loop for escalation in (group->list (get-filler 'escalations mop))
       collect (setf prev-motive
                     (mop-calc `((role motive)
                                 (escalation ,escalation)
                                 (prev-motive ,prev-motive))))))))

;; slots like ((role sentence) (index ...) (old-sentence ...)
;;             (old-action ...) (old-motive ...) (old-severity ...)
;;             (this-action ..) (this-motive ..) (this-severity ..))
;; is input in the adaptation.
(defun mop-calc (slots)
  "finds the specialization of m-calc with the given slots and
   returns its value filler."
  (let ((instance (slots->mop slots '(m-calc) nil)))
    (and instance (get-filler 'value instance))))

(definstance motive (m-role))

(defmop m-calc (m-root))

(defmop m-calc-motive (m-calc)
  (role motive)
  (value nil))
```

前回がどうあろうと，今回エスカレーションしていれば，それは報復的とされる．

```
;; If the role is motive, and the escalation is above 0,
;; then it is m-calc-escalation-motive, and its value is I-m-retaliation.
(defmop m-calc-escalation-motive (m-calc-motive)
  (escalation m-range (above 0))
  (value I-m-retaliation))
```

もしその前の motive が正当化されなくて，今回エスカレーションしていなければ，それは自己防衛とされる．

```
;; If the role is motive, and the escalation is below 1,
;; and prev-motive is unjustified,
;; then it is self-defence-motive, and its value is I-m-self-defence.
(defmop m-calc-self-defence-motive (m-calc-motive)
  (escalation m-range (below 1))
  (prev-motive m-unjustified)
  (value I-m-self-defence))
```

もしその前の motive が正当化されていたら，今回エスカレーションしていなくても，それは報復的とされる．

```
;; If the role is motive, and the escalation is below 1,
;; and prev-motive is justified,
;; then it is m-calc-retaliation-motive, and its value is I-m-retaliation.
(defmop m-calc-retaliation-motive (m-calc-motive)
  (escalation m-range (below 1))
  (prev-motive m-justified)
  (value I-m-retaliation))
```

過去の判決をもとに今回の量刑を計算するのが，adjust-sentenceである．

```
(defun adjust-sentence (pattern mop)
  "is called by the sentence calculation MOPs to adjust the sentence
   appropriately. The slots in <mop> contain the old sentence, a
   weighing factor (between 0.00 and 0.50) that says how critical
   the difference is, an index (between 0 and the number of events
   in the shortest fight) that says how close the difference was to
   the final act, and a direction (1 or -1) that says which case is
   worse, the old one(-1) or the new one(1)."
  (declare (ignore pattern))
  (format t "~&---------------")
  (format t "~&~S applied, ~S events from the end" mop (get-filler 'index mop))
  (adjust-fn
   (get-filler 'old-sentence mop)
   (get-filler 'weight mop)
   (get-filler 'index mop)
   (get-filler 'direction mop)))

(defun adjust-fn (x y index direction)
  "ADJUST-FN <sentence> <weight> <index> <direction>
   determines the value of the new sentence, based on the old sentence,
   using the formula
       sentence + (sentence x (weight + closeness) x direction)
   where closeness is 0.25 if <index> is 0 or 1, i.e., the last or
   sencond last events differed, or 0.00 if <index> >= 1."
  (+ x  (* x
           (+ y (cond ((< index 2) 0.25)
                      (t 0.0)))
           direction)))
```

Judgeの以下のdemoプログラムを実施すると次のような結果が得られる．

```
(defun judge-demo ()
  (run-judge *case1* '*case1*)
  (run-judge *case2* '*case2*)
```

```
  (run-judge *case3* '*case3*))

(defun run-judge (case case-name)
  (format t "~&---------------")
  (format t "~&Sentencing ~S in ~S"
            (role-filler 'defendant case) case-name)
  (let ((instance (judge (forms->slots case))))
    (assert (not (null instance)) () "RUN-JUDGE: null instance.")
    (format t "~&Sentence in ~S is ~S years"
              instance (role-filler 'sentence instance))
    instance))

(defun judge (slots)
  "JUDGE <slot-list>
   finds or creates a case under m-crime with the given slots
   and returns it."
  (let ((instance (slots->mop slots '(m-crime) t)))
    (and (get-filler 'sentence instance)
         instance)))
```

```
cg-user(1): (in-package :cbr)
#<The cbr package>
cbr(2): (judge-demo)
---------------
Sentencing I-m-Ted in *case1*
Sentence in I-m-crime.24 is 40 years
---------------
Sentencing I-m-Randy in *case2*
I-m-crime.37:old <= I-m-crime.24
ADAPT-SENTENCE: old-mop I-m-crime.24
--------------
Adapting the sentence in I-m-crime.24
---------------
Calculating motives in I-m-crime.24
---------------
Calculating escalations in I-m-crime.24
I-m-crime.24:escalations <= I-m-group.38
I-m-crime.24:motives <= I-m-group.42
---------------
Calculating motives in I-m-crime.37
---------------
Calculating escalations in I-m-crime.37
I-m-crime.37:escalations <= I-m-group.43
I-m-crime.37:motives <= I-m-group.49
--------------------
No major difference found
Using old sentence
I-m-crime.37:sentence <= 40
Sentence in I-m-crime.37 is 40 years
---------------
Sentencing I-m-Tim in *case3*
I-m-crime.63:old <= I-m-crime.37
ADAPT-SENTENCE: old-mop I-m-crime.37
--------------
Adapting the sentence in I-m-crime.37
---------------
Calculating motives in I-m-crime.63
---------------
Calculating escalations in I-m-crime.63
```

```
I-m-crime.63:escalations <= I-m-group.64
I-m-crime.63:motives <= I-m-group.69
---------------
I-m-calc.71 applied, 1 events from the end
I-m-calc.71:value <= 30.0
I-m-crime.63:sentence <= 30.0
Sentence in I-m-crime.63 is 30.0 years
I-m-crime.63
```

27.5 Direct Memory Access Parser (DMAP)

Direct Memory Access Parser (DMAP) は，Memory Organization Package (MOP) 上に構築された，事例に基づくParserである．この自然言語パーザは他のパーザと異なり，形態素解析／統語解析／意味解析といった，通常の自然言語処理手続きを踏まずに，フレーズの出現と概念階層知識を直接参照して，過去のフレーズ事例から予期(prediction)を作りだし，フレーズ中の単語と予期中の概念とのマッチングをとりつつ，予期を１単語ずつ消費していく．

たとえば，"Milton Friedman"というフレーズは，具体mopとして定義されている I-m-Friedman を活性化させ，"(actor) says (info) period"というフレーズは，actorとinfoという二つのロール名をスロットに持つ記憶の中のm-mtrans抽象mopを活性化させる．

だからパージングの際には，解析される文中にある概念も個物（インスタンス）も mop として事前に登録済みでなければならない．DMAPデモの例では以下のような mop がまずは定義される．

```
(defmop m-actor (m-root))
(defmop m-economist (m-actor))
(defmop m-monetarist (m-economist))
(definstance I-m-Friedman (m-monetarist))
(defmop m-supply-sider (m-economist))
(definstance I-m-Laffer (m-supply-sider))

(defmop m-act (m-root))
(definstance I-m-mtrans-act (m-act))

(defmop m-variable (m-root))
(defmop m-bad-var (m-variable))
(defmop m-neutral-var (m-variable))
(definstance I-m-interest-rates (m-bad-var))
(definstance I-m-money-supply (m-neutral-var))

(defmop m-behavior (m-root))
(definstance I-m-increase (m-behavior))
(definstance I-m-decrease (m-behavior))

(defmop m-state-change (m-root))

(defmop m-vtrans (m-state-change)
  (var m-variable) (beh m-behavior))

(defmop m-good-vtrans (m-vtrans)
  (var m-variable))
(defmop m-bad-vtrans (m-vtrans)
  (var m-variable))
```

```
(defmop m-neutral-vtrans (m-vtrans)
  (var m-neutral-var))

(defmop m-bad-down (m-good-vtrans)
  (var m-bad-var) (beh I-m-decrease))
(defmop m-bad-up (m-bad-vtrans)
  (var m-bad-var) (beh I-m-increase))

(defmop I-m-ms-up (m-neutral-vtrans)
  (var I-m-money-supply))

(defmop m-causal (m-root)
  (ante m-state-change) (cnsq m-state-change))
(defmop m-causal/econ (m-causal)
  (ante m-vtrans) (cnsq m-vtrans))
(defmop m-causal/mon (m-causal/econ)
  (ante I-m-ms-up) (cnsq m-bad-vtrans))

(defmop m-mobject (m-root)
  (content nil))
(defmop m-mobject/state-change (m-mobject)
  (content m-state-change))
(defmop m-arg (m-mobject)
  (content m-causal))
(defmop m-arg/econ (m-arg)
  (content m-causal/econ))
(defmop m-arg/mon (m-arg/econ)
  (content m-causal/mon))

(defmop m-mtrans (m-event)
  (action I-m-mtrans-act) (actor m-actor)
  (info m-mobject))

(defmop m-mtrans/state-change (m-mtrans)
  (info m-mobject/state-change))

(defmop m-mtrans/econ (m-mtrans)
  (actor m-economist)
  (info m-arg (content m-causal/econ)))
```

すなわち，I-m-Friedmanは"Milton Friedman"に対応すべき具体 mop であり，それはマネタリストであり，I-m-Lafferは"Arthur Laffer"で活性化されるべき具体 mop であり，それはサプライサイドの経済学者である．

ロジャー・シャンクは概念依存理論を創出したことで有名であるが，ここでもそれが用いられている．概念依存理論では動詞をごく基本的ないくつかの抽象的なカテゴリーに分類する．PTRANSはgoのようなオブジェクトの物理的な場所の変化を伴う移動を表し，ATRANSはgiveやtakeのような抽象的な関係の移動を表し，MTRANSはtellやsayのようなメンタルな行為を表す．上記では，`m-mtrans/state-change` は状態変化のMTRANSであり，`m-mtrans/econ` は経済に関するMTRANSである．

27.6 事例ベース推論とは何であったか？

Schank らによるMOPは，オリジナルではリスプシンボルとシンボル属性を利用して，構造体やフレームのように見せかけたものである．その後に出版された本では，実際フレームと称する構造体で作られたMOPのバージョンも存在した．また，IS-A階層のような抽象・特殊の上下関係や属性の継承などフレームと同様な機能も作られた．しかし，MOPにはそれ以上の機能もある．

エキスパートシステムではドメインの知識をルール形式に表現し問題解決を行う．すなわち，問題解決に必要十分な知識がなければならないが，事例ベース推論では過去の事例が豊富になければならない．例えば設計業務を考えたとき，あるいは与信評価を考えたとき，自動化のためのアルゴリズムがあってシステム化できていれば，毎回スクラッチでゼロから計算すればよいが，一般にそのようなシステム開発は大変である．かと言って業務に必要な知識をきちんと集めることも大変である．しかし過去の事例として，失敗事例，成功事例が豊富にあれば，それを糧にして，事例ベースのシステムを開発することができる．ただし，その場合でも，何も知識が要らないというわけではなく，属性とその値の抽出や選択，事例を分類するための階層化などは必要である．しかし，スクラッチでエキスパートシステムを開発するよりは実現性は高いであろう．過去の事例というのは細部の知識が捨象された結果である知識の塊であり，IF～THENルールの結果であると見ることもできる．実際，設計分野への応用についてコンファレンスが持たれたこともあったが，その後類似の話を聞かない．今ではディープラーニング全盛であるが，機械学習の知識だけでドメインの知識なしに，設計や与信評価のようなアプリ開発がうまくいくとは思えない．知識全盛とデータ全盛の間を埋めるものとして，事例ベースにもう一度光が当たってもいいと思われる．

第28章 命題論理

> あらゆる分野において，基礎は重要であると言われながらも，日本人の特質からか，概して基礎よりも応用に力がはいりがちである．記号論理学が日本でこれまでほとんど十分な研究がなされなかったのも，そういったことの現われかもしれないが，これからはそうでないことを祈りたい．ある時点で"意味はあるが有用ではない"と思われていたものが，いつかの時点でその意義と有用性が広く認められ，それが欠くべからざるものになることは多い．情報科学にとって記号論理学はそういったものの一つであろう．
>
> [長尾 真, 淵 一博, 1983]

28.1 命題論理とは何か

第24章と第26章では，推論とルールベースシステムについて記述したが，その記述がいつも正しいとは限らない．たとえば，"Colorless green ideas sleep furiously"（無色の緑色の考えが猛烈に眠る）という文は，文法的には完全に正しいが現実に照らして意味を成さない．計算機は単語の意味を知らなければ，この文は意味を成さないということが分からない．ルール文においても，現実にナンセンスなルールはいくらでも作ることができる．以下はその一例である．

- IF 青色が緑虫を食べる THEN その青色は夕日を産む

世界に関する知識をどう表現するかという問題は，**知識表現**と呼ばれるが，知識をどうテキストとしてある文法に基づいて一貫性のある記述をするかという問題は**統語論** (syntax)と呼ばれ，表現された知識が世界に照らして正しいかどうかという問題は**意味論** (semantics)と呼ばれる．分かりやすく算数の例で言えば，"1+2=5"という文は統語論的に正しいが意味論的には正しくない．一方，"+1=2 5"という文はまず統語論的に受け付けることができない．同じ知識をルール表現で表したり，フレームとして表したりするが，それは知識表現問題であり，計算機が (意味とは関係なく) 扱える表現体系にするのは統語論的問題であり，表現された知識が現実的に正しいかどうかは意味論的問題である．

有意味か無意味かはどのように判断されるのだろうか．今ではそれはTarskiの提唱した意味論(Tarskian semantics)が主流となっている．図28.1 を参照されたい．文の集合は世界における事実の集合に対応する．元の文の集合から他の文が導かれるとき，その導かれた文も世界のある事実に対応しなければならない．元の文集合からもともとそこに含まれるべき他の文を導くことは**伴意**と呼ばれる．ある伴意規則を定めたとき，文の集合から導かれるすべての文が世界に照らして対応があるときに，元の文集合と伴意規則は正しいと考える．文の表現と世界の事実との関係を考えるのは**解釈**(interpret)と呼ばれる．これも分かりやすく算数の例で言えば，文字として書かれた"1+2=3"を見たときに，我々は"1"や"2"や"3"を書かれた文字としてでなく，それらが意味する数として解釈するのが普通である．数字の世界ではなく現実世界の知識表現問題としては，"青色"という言葉が何を意味するか，"緑虫"という言葉が何を意味するのかを解釈して，現実世界に照らして文の真偽を判定する．

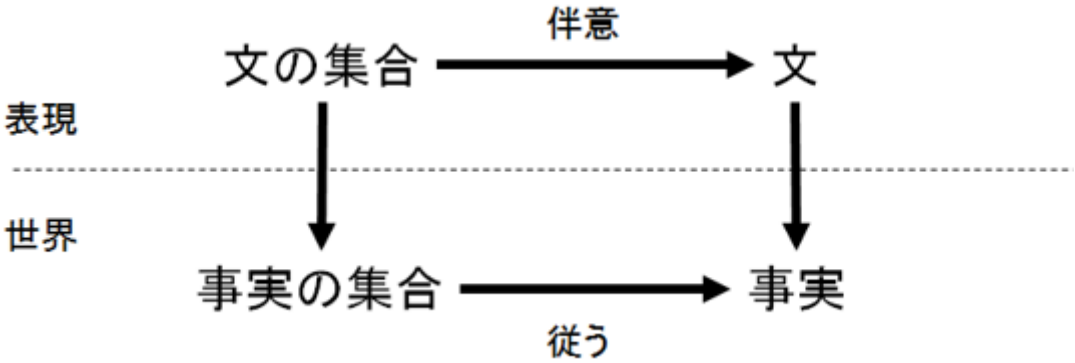


図28.1 文と事実の関係 (AIMAより)

命題 (**proposition**) とは世界に関する事実を表す文 (sentence) のことであるが，**命題論理**では，このような世界についての言明 (statement) が正しいかどうかを問題とする．世界の事実に照らして(解釈して)，ある命題が正しければその命題は「真」である．正しくなければその命題は「偽」である．ここで，「かつ」や「または」などにより論理結合されていない文(原子文，atomic sentence)は，事実に照らして真偽が即座に判定できるものとする．実際には，にわかには真偽は判定できるものではないが，古典的な命題論理では，事実に対応する単純な文は「わからない」とか「未知である」ということはないものとしている．また，時や場合によって真偽が変わることもないものとする．しかも命題論理では変項（変数）はない．命題論理ではそのような前提条件の下で，複数の文の組み合わせの真偽を取り上げるが，その組み合わせの種類は次のものに限られている．

- **連言** (**conjunction**) : たとえば (**and** (animal is bird) (animal flies well))
- **選言** (**disjunction**) : たとえば (**or** (animal has hair) (animal gives milk))
- **含意** (**implication**) : たとえば (**imply** (animal has hair) (animal is mammal))
- **同値** (**equivalence**) : たとえば (**equal** (animal is mammal) その動物は哺乳類である)
- **否定** (**negation**) : たとえば (**not** (animal is mammal))

28.2 命題論理の演算

命題論理では上記の論理結合子(connective)，連言(∧で表す)，選言(∨で表す)，含意(⇒で表す)，同値(⇔で表す)，否定(¬で表す)による複合文の真偽を取り上げる．たとえば上記例で言えば次のようになる．

- 連言 (conjunction) : たとえば (animal is bird) ∧ (animal flies well)
- 選言 (disjunction) : たとえば (animal has hair) ∨ (animal gives milk)
- 含意 (implication) : たとえば (animal has hair) ⇒ (animal is mammal)
- 同値 (equivalence) : たとえば (animal is mammal) ⇔ その動物は哺乳類である
- 否定 (negation) : たとえば ¬ (animal is mammal)

ここで，丸括弧は単に論理結合子以外の文のまとまりを表現するためだけに導入されたものであり，他に意味はない．論理結合子のない文は原子文(atomic sentence)と呼ばれる．命題論理演算の一般的性質を述べ

るのにいちいち"(animal is mammal)"とか「その動物は哺乳類である」と表記するのは不都合なのでこれらの原子文を「P」とか「Q」といったアルファベットの大文字で表記することにする．原子文は当然のことながら，真偽が判定できるものでなければならない．ただの color とか bird というのは真偽が判定できないので命題論理の原子文ではない．ここで，論理的な「真」を「⊤」で，「偽」を「⊥」で表記する．

原子文に真偽を当てはめたとき，論理結合子で組み合わされた複合文の真偽を判定することを考える．複合文の真偽はそれを構成する原子文から計算される．これは次のような真理値表から簡単に計算できる．

conjunction

P \ Q	⊤	⊥
⊤	⊤	⊥
⊥	⊥	⊥

disjunction

P \ Q	⊤	⊥
⊤	⊤	⊤
⊥	⊤	⊥

negation

P	⊤	⊥
$\neg P$	⊥	⊤

連言は「かつ」と読まれてよく，$P \wedge Q$ は P と Q がともに真のときのみ真となる．選言は「または」と読まれてよく，$P \vee Q$ は P と Q のどちらかが真であれば真となる．自然言語では「明日は晴れまたは雨」というと，晴れでなければ雨のように聞こえるが，論理の世界では「または」は晴れと雨が両方とも真であってもよい．P の否定は真理値を反転すればよい．

equivalence

P \ Q	⊤	⊥
⊤	⊤	⊥
⊥	⊥	⊤

implication

P \ Q	⊤	⊥
⊤	⊤	⊥
⊥	⊤	⊤

一方，同値と含意は注意が必要である．同値 $P \Leftrightarrow Q$ は $(P \Rightarrow Q) \wedge (Q \Rightarrow P)$の言い換えであるが，これらは第24章や第26章で説明したルールベースとは意味が異なる．ルールベースシステムでは，ルールは因果関係のようなものを表しているが，命題論理ではそうではなく，因果関係に関係なく上記の関係が成り立てばよい．そのような例として，AIMA 第１版では「５が奇数ならば東京は日本の首都である」を挙げている．因果関係とは関係なく，命題論理ではこの文は真である．特に含意では，if P then Q において P が偽ならば Q の真偽にかかわらず，if P then Q は真と評価されることに注意されたい．たとえば「５が偶数ならばSamは利口である」という文は，Samが利口か否かにかかわらず命題論理では真である．

問題 28.1 シンボルを原子文とし，その値 (symbol value) を t あるいは nil とすると，上記 conjunction, disjunction, negation は Common Lisp の eval と同じである．(and P Q), (or P Q), (not P), (imply P Q),

(equiv P Q) を評価して正しく値を返す関数 interp を定義せよ.

妥当性と世界モデル

「唯一神はいる」という言明はキリスト教徒やイスラム教徒にとっては真であるが，神道の信者や無神論者にとっては偽である．一方，「神はいない」という言明は無神論者にとっては真だが，そうでないものにとっては偽である．「神はいるまたは神はいない」という言明は，どんな世界やどんな人にとっても真であるが，「神はいるかつ神はいない」という言明は，どんな世界やどんな人にとっても偽である．ある解釈によってある言明が真となる世界があるとき，その言明は**充足可能** (satisfiable)と呼ばれ，その世界は解釈と一緒にして**モデル**と呼ばれる．可能などんな世界においてもある言明が真であるとき，それは**恒真的** (valid)と呼ばれ，どんな世界においても偽であるとき，それは**恒偽式**あるいは**充足不能** (unsatisfiable)と呼ばれる.

恒真的な論理式は時に**トートロジー**と呼ばれるが，大体においてトートロジーとなる論理式は証明を仕事とする論理学者にとっては重要であるが，知識工学やオントロジーの観点では，それ自体では何か新しい知識をつけ加えるわけでもなく，あまり面白くないことがほとんどである．例えば，「雨が降っているかまたは降っていないかである」と聞いたところで，それ自体は常に真（恒真）であるが，何か天気について新しい知識を得たわけではない（論理哲学論考，4-461，ウィトゲンシュタイン）．一方，**伴意**とはある文集合で充足可能なすべてのモデルにおいて，別の文も同モデルで必ず充足可能であることを言う．たとえば，「ＡはＢの下位概念であるとき，ＡもＢも概念である」という伴意ルールがあったとき，「哺乳類は動物の下位概念である」という言明があれば，両者から「哺乳類は概念である」と「動物は概念である」という文が伴意される．トートロジーは得てしてつまらないが，伴意は重要な機能である．あるモデルにおいて，甲と言ったら乙と言ったことにもなるのだから，つまらなくはない.

28.3 命題論理式とその計算

論理定項 true, false と 原子文と論理結合子，すなわち連言，選言，含意，同値，否定において括弧で結合された式を命題論理式という.

命題論理式の統語論

1. 論理定項 true と false は論理式である
2. 原子文は論理式である
3. 問題28.1 に示すように，論理式を and，or，not，imply，equiv を用いて括弧でくくった式は論理式である

命題論理式の計算は，原子文の真偽値が決まっていれば簡単である．原子文に真偽値を割り当てたのち，上記の演算ルールを使って各複合文の真偽値を計算すればよい．原子文の数が有限で，論理式も有限の長さなので，解は必ず得られる．ただし，ある命題論理式が恒真式か恒偽式か，あるいは充足可能か充足不能かを計算するためには真偽値の割り当てについてすべての組み合わせを考えなければならない．原子文の数が

多くなるとその真偽値の割り当ての組み合わせは2のn乗で指数関数的に増大する．このように取り扱う要素数の大きさに応じて指数関数的に計算時間が増大する論理式は，**至難** (**intractable**)と呼ばれる．一方，論理学者にとってはある論理式が恒真式か恒偽式かが興味の対象となるが，恒真式／恒偽式を計算する問題は**決定問題**と言われる．一般の命題論理式は**決定可能**(**decidable**)であるが**至難**(**intractable**)である．

28.4 命題論理式のプログラム

教科書などで書かれる論理式は通常，(animal is bird) ∧ (animal flies well) とか (animal has hair) ⇒ (animal is mammal) といった中置き (infix) 形式で記述される．しかしLispの場合は前置形式が便利である．問題28.1で示したような前置形式の命題論理式を解釈する関数 interp は，例えば次のようにプログラムされる．

```
(defun make-exp (op &rest args) (cons op args))
(defun op (exp) "Operator of an expression" (if (listp exp) (first exp) exp))
(defun args (exp) "Arguments of an expression" (if (listp exp) (rest exp) nil))
(defun arg1 (exp) "First argument" (first (args exp)))
(defun arg2 (exp) "Second argument" (second (args exp)))

(defsetf args (exp) (new-value)
  `(setf (cdr ,exp) ,new-value))

(defun interp (sentence &optional interpretation)
  "Evaluate the truth of the sentence under an interpretation.
  The interpretation is a list of (proposition . truth-value) pairs,
  where a truth-value is t or nil, e.g., ((P . t) (Q . nil)).
  It is an error if there are any propositional atomics in the sentence
  that are not given a value in the interpretation."
  (cond (interpretation (interp (sublis interpretation sentence :test #'equal) nil))
        ((eq sentence 't) t)
        ((eq sentence 'nil) nil)
        ((atom sentence) (error "No interpretation for ~A." sentence))
        (t (case (op sentence)
             (or  (some #'interp (args sentence)))
             (and (every #'interp (args sentence)))
             (not (not (interp (arg1 sentence))))
             (imply (or (interp (arg2 sentence))
                        (not (interp (arg1 sentence)))))
             (equiv (eq (interp (arg1 sentence))
                        (interp (arg2 sentence))))
             (otherwise (error "Unknown connective ~A in ~A"
                               (op sentence) sentence))))))
```

ここで sentence に与えられる論理式の原子文の一つ一つに対して，真偽値の割り当てを関数 interp の第2引数に与える．たとえば，次の様である．

```
cl-user(21): (interp '(imply (animal has hair) (animal is mammal))
                     '(((animal has hair) . t) ((animal is mammal) . t)))
t
cl-user(22): (interp '(imply (animal has hair) (animal is mammal))
                     '(((animal has hair) . t) ((animal is mammal) . nil)))
nil
cl-user(23): (interp '(imply (animal has hair) (animal is mammal))
```

```
                    '(((animal has hair) . nil) ((animal is mammal) . t)))
t
```

さて命題論理の場合には，考えうる世界は無限にあるわけではない．つまりintractableであっても，各原子文への真偽値の割り当ての組み合わせは有限であり，それを行うのが次の compute-truth-entries である．

```
(defun compute-truth-entries (atomics sentences)
  "Compute the truth of each sentence under all interpretations of atomics."
  (mapcar #'(lambda (interpretation)
              (mapcar #'(lambda (sentence)
                          (interp sentence interpretation))
                      sentences))
          (all-truth-interpretations atomics)))

(defun all-truth-interpretations (atomics)
  "Return all 2^n interpretations for a list of n atomics."
  (if (null atomics)
      (list nil)
      (let ((atomic (first atomics)))
        (mapcan #'(lambda (sub-rest)
                    `(((,atomic . t) . ,sub-rest)
                      ((,atomic . nil) . ,sub-rest)))
                (all-truth-interpretations (rest atomics))))))
```

関数 all-truth-interpretations は与えられた原子文のリストから，その各々の原子文についての真偽値割り当てのすべての組み合わせを作る．関数 compute-truth-entries は，作られたすべての真偽値割り当てを使って，引数に与えられた原子文リストからその一つ一つについて sentences 中の命題論理文について解釈を実行して，その結果を返す．得られる結果は，sentences 中の各命題論理文ごとの結果がリストになったものである．たとえば，P と Q を原子文としたときの連言と選言については次のような結果になる．

```
cl-user(26): (compute-truth-entries '(P Q) '(P Q (and P Q) (or P Q)))
((t t t t) (nil t nil t) (t nil nil t) (nil nil nil nil))
```

与えられた命題論理式について，その中の原子文について真偽値のすべての組み合わせを考えて真偽値表 (truth table) を作ることを考える．

```
(defstruct truth-table
  atomics              ; The propositional atomics
  sentences            ; Sentences that head the columns
  rows                 ; Lists of t or nil truth values
  )

(defun build-truth-table (sentence &key short)
  "Build a truth table whose last column is the sentence.  If SHORT is true,
  then that is the only column. If SHORT is false, all the sub-sentences
  are also included as columns (except constants)."
  (let* ((atomics (prop-atomics-in sentence))
         (sentences (if short
                        (list sentence)
                        (append atomics (complex-sentences-in sentence)))))
    (make-truth-table :atomics atomics
```

```
                    :sentences sentences
                    :rows (compute-truth-entries atomics sentences))))
```

prop-atomics-in は与えられた命題論理式中から原子文を抜き出す関数であり，complex-sentences-in は原子文以外の論理式を抜き出す関数である．

```
(defun atomic-p (sentence)
  (or (eq sentence 't)
      (eq sentence 'nil)
      (atom sentence)
      (not (member (op sentence) '(or and not imply equiv)))))

(defun prop-atomics-in (sentence)
  "Return a list of all the atomic sentences in <sentence>."
  (cond ((member sentence '(t nil)) nil)
        ((atom sentence) (list sentence))
        ((atomic-p sentence) (list sentence))
        (t (delete-duplicates (mapcan #'prop-atomics-in (args sentence))
                              :test #'equal
                              :from-end t))))

(defun complex-sentences-in (sentence)
  "Return a list of all non-atom sub-sentences of sentence."
  (cond ((atomic-p sentence) nil)
        (t (delete-duplicates
            (nconc (mapcan #'complex-sentences-in (args sentence))
                   (list sentence))))))
```

関数 truth-table は与えられた命題論理式について実際に真偽値表を作りだして，印刷する．

```
(defun truth-table (sentence)
  "Build and print a truth table for this sentence, with columns for all the
  propositions and all the non-trivial component sentences.  If the sentence
  is valid, the last column will have all T's.
  Example: (truth-table '(equiv P (not (not P))))."
  (print-truth-table (build-truth-table sentence)))

(defun print-truth-table (table &optional (stream t))
  "Print a truth table."
  (let* ((headers (mapcar #'sentence-output-form
                          (truth-table-sentences table)))
         (width (+ (* 2 (length headers))
                   (sum headers #'length))))
    ;; Each sentence is printed as a column header, surrounded by 2 spaces
    (print-dashes width stream t)
    (format stream "~{ ~A ~}~%" headers)
    (print-dashes width stream t)
    (dolist (row (truth-table-rows table))
      (mapcar #'(lambda (entry header)
                  (print-centered (if entry "T" "F")
                                  (+ 2 (length header))
                                  stream))
              row
              headers)
      (format stream "~%"))
    (print-dashes width stream t)))

(defun print-dashes (width &optional (stream t) separate-line)
```

```
  "Print a line of dashes WIDTH wide."
  (when separate-line (format stream "~&"))
  (print-repeated "-" width stream)
  (when separate-line (format stream "~%")))

(defun print-centered (string width &optional (stream t))
  "Print STRING centered in a field WIDTH wide."
  (let ((blanks (- width (length (stringify string)))))
    (print-repeated " " (floor blanks 2) stream)
    (format stream "~A" string)
    (print-repeated " " (ceiling blanks 2) stream)))

(defun print-repeated (string n &optional (stream t))
  "Print the string n times."
  (dotimes (i n)
    (format stream "~A" string)))

(defun stringify (exp)
  "Coerce argument to a string."
  (cond ((stringp exp) exp)
        ((symbolp exp) (symbol-name exp))
        (t (format nil "~A" exp))))

(defun sentence-output-form (sentence)
  "Convert a prefix sentence back into an infix notation with brief operators."
  (format nil "~{~A~^ ~}"
    (mklist (sublis '((and . "∧") (not . "~") (or . "V") (equiv . "⇔") (imply . "⇒"))
                    (prefix->infix sentence)))))

(defun mklist (x)
  "If x is a list, return it; otherwise return a singleton list, (x)."
  (if (listp x) x (list x)))

(defun sum (numbers &optional (key #'identity))
  "Add up all the numbers; if KEY is given, apply it to each number first."
  (if (null numbers)
      0
      (+ (funcall key (first numbers)) (sum (rest numbers) key))))
```

以下はその実施例である．T は真，F は偽を表している．印刷の都合上，not は ～ で表されている．

```
cl-user(27): (truth-table '(or P (not P)))
------------------
 P  ~ P  P V (~ P)
------------------
 T   F       T
 F   T       T
------------------
nil
cl-user(28): (truth-table '(and P (not P)))
------------------
 P  ~ P  P ∧ (~ P)
------------------
 T   F       F
 F   T       F
------------------
nil
cl-user(29): (truth-table '(equiv P (not (not P))))
```

```
--------------------------------
 P  ~ P  ~ (~ P)  P ⇔ (~ (~ P))
--------------------------------
 T   F      T           T
 F   T      F           T
--------------------------------
nil
cl-user(30): (truth-table '(equiv P (not (not Q))))
-----------------------------------
 P  Q  ~ Q  ~ (~ Q)  P ⇔ (~ (~ Q))
-----------------------------------
 T  T   F      T           T
 F  T   F      T           F
 T  F   T      F           F
 F  F   T      F           T
-----------------------------------
nil
```

実施例27や29のように，どんな真偽値の割り当てでも常に真が得られる式が恒真式であり，実施例28のように，どんな真偽値の割り当てでも常に偽が得られる式が恒偽式であり，実施例30のように割り当てによって真偽が変わるが真が得られる割り当てがあるときは充足可能である．

それでは次はどうだろうか

```
cg-user(31): (truth-table '(equiv P Q))
-------------
 P  Q  P ⇔ Q
-------------
 T  T    T
 F  T    F
 T  F    F
 F  F    T
-------------
nil
```

PもQも真のときは P ⇔ Q は当然真だが，PもQがともに偽のときでも P ⇔ Q は真だ．これはP ⇔ Qという言明についての真偽値だから素直に納得できる．PもQがともに偽のときには P ⇔ Q も偽であるべきだと考える人はいないだろう．ところがこれが imply になると我々の常識的な感覚と異なってくる．

```
cg-user(32): (truth-table '(imply P Q))
-------------
 P  Q  P ⇒ Q
-------------
 T  T    T
 F  T    T
 T  F    F
 F  F    T
-------------
nil
```

すなわち，これが前出の「『５が偶数ならばSamは利口である』という文は，Samが利口か否かにかかわらず命題論理では真である．」に該当する．

第29章 ユニフィケーション

29.1 ユニフィケーション（単一化）とは？

ユニフィケーション(単一化)とは，二つの記号表現があるときに，その二つの記号表現が同一であるかを調べたり(変項がない場合)，変項があるときには，二つのパターンを同一視するような変項の割り当てを求めることである．たとえば，$s = f(a, x)$ と $t = f(y, b)$ がある場合，$y \approx a$， $x \approx b$ と割り当てることができれば，この二つの式は同じことになって，$s \approx t$ と同一視することができる．ここで， x や y は論理変項であるが，一つの変項を二つの異なる記号に割り当てることはできないことを注意しておく．したがって，$s' = f(a, x)$ と $t' = f(x, b)$ を単一化することはできない．このような単一化の操作は，Prolog やプラニングプログラムの基礎であり，**融合**(**resolution**)など論理操作の基礎であり，人工知能のいたるところで出てくる．本書籍でも，第22章のElizaプログラムにおいてパターンマッチング関数であるpat-machを紹介したが，それをもっと高機能化したものがユニフィケーション・プログラムである．

一般に，単一化の答えは複数ときには無限にありえる．たとえば，$f(x, y)$ と $f(y, x)$ は，$x \approx a$，$y \approx a$ とすれば単一化できるが，その場合 a, b, c, ... と無限個の答えが出てくることは明らかである．しかし，我々がほしいのは $x \approx y$ という割り当てだけでよく，これを**最汎単一化** (most general unifier, MGU) という．

ここで，ユニフィケーションを記述するための最小限の用語を説明する．論理表現の**論理定項**，**論理変項**，および**関数表現**は，**項**(term)と呼ばれ，何らかのオブジェクトを指示(denote)する．論理表現中の定項は**論理定項**，変項は**論理変項**とも呼ばれる．$Love(John, Mary)$ あるいはLispコードであれば `(Love John Mary)` を論理表現と捉えたとき，$John$ や$Mary$ は論理定項である．$Love(John, x)$ あるいは `(Love John ?x)` を論理表現と捉えたとき， x や `?x` は論理変項である．どんな記号を論理変項とするかは，テキストの最初に記述されるのが普通であるが，通常 x，y，z などが論理変項として用いられ，Lisp の S 式では第1文字が ? や $ であるシンボルが変項としてよく用いられる．一方，上記において $Love$ は**述語**と呼ばれるもので，述語とは論理表現において複数オブジェクトの間の関係を記述するものである．$Married(FatherOf(Richard), MotherOf(John))$ あるいは S 式 `(Married (FatherOf Richard) (MotherOf John))` において，$Married$ は述語であるが， $FatherOf$ や $MotherOf$ は関数である．論理表現における関数表現は，その引数オブジェクトから一意に定まるオブジェクトを指示するものである．

今，二つの論理表現 $Knows(John, x)$ と $Knows(John, Jane)$ があるとき，これらを同一にするには，前者の変項 x を $Jane$ に置き換えればよい．これを**置換** (**substitution**)と呼ぶが，一般に一つの論理表現には複数の変項があって，それらの変項を置換する操作は，その論理表現の変項を束縛したものと考えることができる．そのような変項束縛の集合を，**単一化代入**(unifier)と呼び，$\{ x_1 \mapsto s_1, \ldots, x_n \mapsto s_n \}$ と表記する．本節以後のLispにおける実現ではこれを assoc リストとして表現する．上記例では，$\{ x \mapsto Jane \}$ あるいは `((?x . Jane))` が単一化代入となる．

世の中にはLispに限っても様々なユニフィケーション・コードがあるが，ここでは Novig の PAIP にある非破壊的単一化プログラムと破壊的単一化プログラムを紹介したあと，後述のプラニングの章で用いられる構造体を対象とした単一化プログラムと，型を考慮した単一化プログラムを紹介する．

29.2 非破壊的単一化プログラム

関数 unify は二つの論理表現と一つの変項束縛を引数にとり，もらった変項束縛を前提に二つの論理表現の間でユニフィケーションを行って，変項束縛に新しい束縛を加えて，その新しく得た変項束縛を返す．

```
(defun unify (x y &optional (bindings +no-bindings+))
  "See if x and y match with given bindings.  If they do,
   return a new binding list that would make them equal."
  (cond ((eq bindings +fail+) +fail+)
        ((eql x y) bindings)
        ((variable? x) (unify-variable x y bindings))
        ((variable? y) (unify-variable y x bindings))
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
                (unify (first x) (first y) bindings)))
        (t +fail+)))
```

+no-bindings+ は文字通り単一化のための束縛がないことを示すものである．assoc リストの場面で紛らわしくないものであればどんなものでもよいが，今ここでは以下のようになっている．+fail+ は単一化の失敗を示すものであるが，当然のことながらその実態は nil である．

```
(defconstant +fail+ nil "Indicates unification failure")

(defconstant +no-bindings+ '((t . t))
  "Indicates unification success, with no variables.")
```

もっとも一般的な場合では，x も y もリストであり，まず両者の第 1 要素について単一化した結果を束縛情報として，後続のリスト要素について単一化する．x や y が変項であれば，それについて unify-variable する．変項でなければ両者が等しいかどうかを直接調べればよい．両者がまったく同一であれば，束縛情報に変化はなく bindings をそのまま返せばよい．

肝心なところは，unify-variable 以下になるが，unify-variable は変項についてその束縛があれば束縛結果について unify で調べ，さもなければ，引数である var と x を新しい束縛として従来の束縛に追加する．

```
(defun unify-variable (var x bindings)
  "Unify var with x, using (and maybe extending) bindings."
  (cond ((get-binding var bindings)
         (unify (lookup var bindings) x bindings))
        ((and (variable? x) (get-binding x bindings))
         (unify var (lookup x bindings) bindings))
        ((occurs-in? var x bindings) +fail+)
        (t (extend-bindings var x bindings))))

(defun variable? (x)
  "Is x a variable (a symbol starting with ?)?"
```

```
  (and (symbolp x) (eql (char (symbol-name x) 0) #\?)))

(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun binding-var (binding)
  "Get the variable part of a single binding."
  (car binding))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun make-binding (var val) (cons var val))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))

(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (make-binding var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy +no-bindings+
        (if (eq bindings +no-bindings+)
            nil
          bindings)))

(defun occurs-in? (var x bindings)
  "Does var occur anywhere inside x?"
  (cond ((eq var x) t)
        ((and (variable? x) (get-binding x bindings))
         (occurs-in? var (lookup x bindings) bindings))
        ((consp x)
         (or (occurs-in? var (first x) bindings)
             (occurs-in? var (rest x) bindings)))
        (t nil)))
```

さて，以下は簡単な実施例である．

```
cl-user(12): (unify '(Love John Jane) '(Love John Jane))
((t . t))
cl-user(13): (unify '(Love John ?x) '(Love John Jane))
((?x . Jane))
cl-user(14): (unify '(Love John ?x) '(Love ?y Jane))
((?x . Jane) (?y . John))
cl-user(15): (unify '(Married (FatherOf Mary) (MotherOf ?x))
                    '(Married (FatherOf ?y) (MotherOf Mary)))
((?x . Mary) (?y . Mary))
cl-user(16): (unify '(Married ?x (MotherOf Mary))
                    '(Married (FatherOf ?y) (MotherOf Mary)))
((?x FatherOf ?y))
cl-user(17): (unify '(Married ?x            (MotherOf Mary))
                    '(Married (FatherOf ?x) (MotherOf Mary)))
nil
cg-user(18): (unify '(Love John . ?x) '(Love John . Jane))
((?x . Jane))
cg-user(19): (unify '(Love John . ?x) '(Love John . (Jane1 Jane2)))
```

```
((?x Jane1 Jane2))
cg-user(20): (unify '(Love John . ?x) '(Love John Jane1 Jane2))
((?x Jane1 Jane2))
cg-user(21): (unify '(Love John ?x) '(Love John Jane1 Jane2))
nil
```

(?x FatherOf ?y) は (?x . (FatherOf ?y)) という意味である．17の例では (?x . (FatherOf ?x)) となるべきところであるが，このように変項のマッチすべきパターンの中に再びその変項が現れるような場合にどう扱うべきかが問題となる．簡単にはこのような巡回を許さないとして，そのような束縛パターンが現れたときには単一化失敗とする．それをするのが occurs-in?である．

unifier は，二つの記号表現の間の単一化代入を実際に実施して，両者を同一にした結果を返す．

```
(defun unifier (x y)
 "Return something that unifies with both x and y (or fail)."
 (subst-bindings (unify x y) x))

(defun subst-bindings (bindings x)
  "Substitute the value of variables in bindings into x,
  taking recursively bound variables into account."
  (cond ((eq bindings +fail+) +fail+)
        ((eq bindings +no-bindings+) x)
        ((and (variable? x) (get-binding x bindings))
         (subst-bindings bindings (lookup x bindings)))
        ((atom x) x)
        (t (reuse-cons (subst-bindings bindings (car x))
                       (subst-bindings bindings (cdr x))
                       x))))

(defun reuse-cons (x y x-y)
  "Return (cons x y), or reuse x-y if it is equal to (cons x y)"
  (if (and (eql x (car x-y)) (eql y (cdr x-y)))
      x-y
    (cons x y)))
```

reuse-cons は 21.2節の説明を参照されたい．

以下に unifier の実施例を示す．

```
cl-user(26): (unifier '(Love John ?x) '(Love ?y Jane))
(Love John Jane)
cl-user(27): (unifier '(Married (FatherOf Mary) (MotherOf ?x))
                      '(Married (FatherOf ?y) (MotherOf Mary)))
(Married (FatherOf Mary) (MotherOf Mary))
cl-user(28): (unifier '(Married ?x (MotherOf Mary))
                      '(Married (FatherOf ?y) (MotherOf Mary)))
(Married (FatherOf ?y) (MotherOf Mary))
```

29.3 破壊的単一化プログラム

先に述べたユニフィケーションでは，変項のユニフィケーションはLisp の連想リスト(assoc list) を使ったものだった．連想リストは昔々のLisp で，実際にリスプシンボルの束縛の実現に利用されたものだ．しかし

すぐ分かるように，あるシンボルの束縛を得るには，連想リストを線形に走査する必要があり，束縛の量が膨大になるとその効率が問題となる．ここで紹介する破壊的単一化では，変項の束縛は var という構造体を使ってその binding スロットを取り出すという方式で行われる．それでも変項の binding がまた変項のときにはその値を取り出すということを延々とすることになるが，連想リストを無駄に走査する処置はなくなる．

変項は次のような構造体のインスタンスだ．文字列 "Unbound" は予約語になるが，まさかそれを使いたい場合は皆無であろう．

```
(defconstant +unbound+ "Unbound")

(defvar *var-counter* 0)

(defstruct (var (:constructor ? ())
                (:print-function print-var))
  (name (incf *var-counter*))
  (binding +unbound+))

(defun print-var (var stream depth)
  (if (or (and *print-level* (>= depth *print-level*))
          (var-p (deref var)))
      (format stream "?~a" (var-name var))
    (write var :stream stream)))
```

変項の束縛値は binding スロットに入れられる．defstruct のオプションである :constructor では，その構造体のコンストラクタの名前を指定できる．ここでコンストラクタは関数?となる．構造体varの実現が起こるたびに，そのnameには新しい数字が入れられる．変項に束縛されている値を取り出すのはマクロ deref だ．これはもちろん dereference の意味である．

```
(defun bound-p (var)
  (not (eq (var-binding var) +unbound+)))

(defmacro deref (exp)
  "Follow pointers for bound variables."
  `(progn
     (loop while (and (var-p ,exp) (bound-p ,exp))
         do (setf ,exp (var-binding ,exp)))
     ,exp))
```

これで何が行われるのか理解するために，ちょっと実行してみよう．構造体 var インスタンスの印刷には print-var を用いるように指定されているので，見かけは ?1 とか ?2 のように印刷される．

```
cl-user(2): (setq foo (?))
?1
cl-user(3): (setq bar (?))
?2
cl-user(4): (describe foo)
?1 is a structure of type var. It has these slots:
  name 1
  binding "Unbound"
```

破壊的ユニフィケーションを実行するのは unify ではなくて，unify! だ．その形はほとんど unify と同じだが，set-bindings! では value を変項の binding スロットにセットすることに注意されたい．そして x と y の両者がリストだったら，先にそれらの car をユニファイして，そのあと cdr をユニファイしていることも注意されたい．だから car のユニファイに失敗すれば cdr のユニファイは実行されないし，ユニファイに成功すれば，その影響は deref を介して cdr のユニファイに影響することになる．

```
(defun unify! (x y)
  "Destructively unify two expressions"
  (cond ((eql (deref x) (deref y)) t)
        ((var-p x) (set-binding! x y))
        ((var-p y) (set-binding! y x))
        ((and (consp x) (consp y))
         (and (unify! (first x) (first y))
              (unify! (rest x) (rest y))))
        (t nil)))

(defvar *trail* (make-array 200 :fill-pointer 0 :adjustable t))

(defun set-binding! (var value)
  "Set var's binding to value, after saving the variable
  in the trail.  Always returns t."
  (unless (eq var value)
    (vector-push-extend var *trail*)
    (setf (var-binding var) value))
  t)
```

変項の束縛がセットされるたびにその変項が *trail* にプッシュされる．これでユニフィケーションの経緯がわかる．とりあえず *trail* のサイズは 200 になっているが，足らなくなったらいくらでも拡張される．12.3節のフィルポインタのところを参照されたい．

最後に一番大事なことは，Prolog などでバックトラックしたときにはそれまでのユニフィケーションの効果を元に戻すことだ．それをするのが，undo-binding! だ．

```
(defun undo-bindings! (old-trail)
  "Undo all bindings back to a given point in the trail."
  (loop until (= (fill-pointer *trail*) old-trail)
      do (setf (var-binding (vector-pop *trail*)) +unbound+)))
```

undo-bindig! は trail のバックトラックする時点でのフィルポインタをもらったら，そこまで一つずつポップしながら，ポップした変項を unbind する．

29.4 複合物単一化プログラム

これまでのところでは，論理表現をリスプのS式で表し，リストのCAR部が述語，CDR部がその引数として論理表現を考えた．述語を論理変項にはしなかったことに気が付かれたであろうか．それは述語まで変項にすることを許すと，高階論理となってしまうからである．リスト形式以外のリスプのデータ型，たとえば構造体を，論理表現に持ち込む場合も，リスト表現が構造体表現になったものと考えればよい．ただし，その場合でもスロットのスロット名は論理変項としてはならないものとする．

一般のデータ型を許すように拡張したunifyは次のようなものである．

```
(defun unify (x y &optional (bindings +no-bindings+))
  "See if x and y match with given bindings.  If they do,
  return a binding list that would make them equal [p 303]."
  (cond ((eq bindings +fail+) +fail+)
        ((eql x y) bindings)
        ((variable? x) (unify-variable x y bindings))
        ((variable? y) (unify-variable y x bindings))
        ((compound-unify? x y) (compound-unify x y bindings))   ;***
        ((and (consp x) (consp y))
         (unify (rest x) (rest y)
                (unify (first x) (first y) bindings)))
        (t +fail+)))
```

ここで，構造体であろうがCLOSオブジェクトであろうが，同種のデータを引き受けて，単一化を行うメソッドをそれぞれの型ごとにメソッド定義すればよい．以下では最も一般的な型指定がない場合のみを記述している．

```
(defgeneric compound-equal (exp1 exp2) (:documentation "Are these expressions equal?"))
(defmethod compound-equal (exp1 exp2)
  (equal exp1 exp2))

(defgeneric compound-unify? (exp1 exp2) (:documentation "Is this expression unifiable?"))
(defmethod compound-unify? (exp1 exp2)
  (declare (ignore exp1 exp2))
  nil)

(defgeneric compound-unify (exp1 exp2 bindings) (:documentation "unify compounds"))
(defmethod compound-unify (exp1 exp2 bindings)
  +fail+)
```

ここで compound-unify? と compound-unify は個別の問題向きに各アプリプログラムにおいて，メソッド定義されるものとしている．ちょっと先走りして第31章のプラニングにおいての例を見てみよう．

状態空間プラニングではプラニングであつかう世界は**状態**であり，これを構造体 statevar のリストとして表現する．一つの statevar は symbol と args と val からなっている．symbol は状態名であり，これが言わば述語に相当する．正規化したあとでは val には t か nil しかこない．論理変項でありえるのは args だ．だから compound-unify はこんな風になる．

```
(defstruct (statevar (:print-function print-statevar))
  symbol args val)

(defun statevar-unify (statevar state bindings)
  "returns a list of possible bindings of statevar against <state>.
   This function collects all possibilities for <state>."
  (loop for st in state with new-bindings
      unless (eq (setq new-bindings (compound-unify statevar st bindings)) +fail+)
      collect new-bindings))

(defmethod compound-unify? ((x statevar) (y statevar))
  "trivially returns true because <x> and <y> are instances of statevar."
  t)
```

```
(defmethod compound-unify ((x statevar) (y statevar) bindings)
  "returns bindings as a result of unification of <x> and <y>. Statevars in
   unification must be normalized."
  (cond ((eql (statevar-symbol x) (statevar-symbol y))
         ;; x and y must be normal statevar form.
         (setq x (normalize-statevar x))
         (setq y (normalize-statevar y))
         (cond ((eql (statevar-val x) (statevar-val y))
                ;; if xval = yval with t&t or nil&nil
                ;; args must be unified
                (unify (statevar-args x) (statevar-args y) bindings))
               (t ;; else t&nil or nil&t
                +fail+
                )))
        (t +fail+)))
```

29.5 型を考慮した単一化プログラム

第30章 述語論理

物理学を専攻する学生が数学に精通する必要があるのと同様に，人工知能を選考する学生は論理式で作業する能力を磨かなければならない．
[Stuart Russell，PeterNorvig, エージェントアプローチ人工知能，古川監訳]

30.1 述語論理とは何か

第28章命題論理では，and や or のような論理結合子で結合された文の真偽値の演算について述べたが，そこには論理変項がなく，文としての記述能力には限界があった．たとえば (animal has hair) ⇒ (animal is mammal) という命題論理文では， animal とか hair のような単語がなんであるかには一切関わりなく，(animal has hair) という文の真偽だけを問題とした．また，命題論理では「すべての」とか「ある」といった限量子がなく，たとえば「だれでもだれかを愛している」とか「だれかを憎んでいる人がいる」というような文を作っても両者の文の関係は一切ない．述語論理では命題論理に加えて，論理変項と「すべての」を意味する**全称限量子** ∀ と「ある」を意味する**存在限量子** ヨ を導入し，世界についてのより豊富な記述を可能にする．第29章ユニフィケーションにて述べたように，論理における述語とは，複数オブジェクトの関係を記述するものである．以下に述語論理の例をその解釈とともに掲げる．

- $\forall x\ \{Animal(x) \wedge Has(x, hair) \Rightarrow Mammal(x)\}$：すべての x について，それが動物でかつ毛があるなら，それは哺乳類である．
- $Eat(blueColor, greenWorm) \Rightarrow Bear(blueColor, sunSet)$：青色が緑虫を食べるとその青色は夕日を産む
- $\forall x \exists y\ \{Person(x) \wedge Person(y) \wedge Love(x, y)\}$：誰でも誰かを愛している．
- $\exists x \exists y\ \{Person(x) \wedge Person(y) \wedge Hate(x, y)\}$：誰かを憎んでいる人がいる．

上記の 2 番目の式は論理変項を含まないが，Eatとか$Bear$が述語と呼ばれるものである．

述語論理のなかでも，特に述語を変項としない次のような定義に従うものを一階述語論理式という．

- 世界に対して解釈可能なシンボルを**オブジェクト**または**個体定項**(論理定項)と呼ぶ．
- 個体定項を指す論理変項を**個体変項**と呼ぶ．
- 任意個の個体定項の組から 1 個の個体定項を導くものを**論理関数**と呼ぶ．
- **述語**とは複数の個体間の関係を主張するものである．

論理関数では通常の関数と異なり，実際に計算するということは問題としない．たとえば，$father\text{-}of(Kennedy)$というのはKennedyという個体のお父さんを指すだけのことである．

1. 個体定項および個体変項は**項**(**term**)である．
2. $t_1,...,t_n$が項であるとき，n変数の関数記号を f として作られる $f(t_1,...,t_n)$は項である．
3. $t_1,...,t_n$が項であり，P をn変数の述語記号とするとき，$P(t_1,...,t_n)$は**基本論理式**と呼ばれる．
4. P，Q が論理式ならば，$\neg P$，$P \Rightarrow Q$も論理式である．
5. P が論理式であり，x が個体変項のとき，$\forall x P$ は論理式である．
6. P が論理式であり，x が個体変項のとき，$\exists x P$ は論理式である．

命題論理(第28章)でいう，原子文は上記基本論理式に相当するので，以後基本論理式のことを原子文ともいうことにする．命題論理では原子文をさらに分解することはなかったが，述語論理では基本論理式は述語とその引数の構造となっている．上記第4項では否定(negation)と含意(implication)しか記述していないが，連言(conjunction)，選言(disjunction)，および同値(equivalence)はあとで示すように，否定と含意から導かれるので，これでよい．

述語論理式において，そこに含まれるすべての変数に全称限量子あるいは存在限量子がついていたとき，その論理式は**閉論理式** (closure) と呼ばれる．それに対して，一つでも限量子のない論理変項があれば，それは**開論理式**と呼ばれる．厳密に言えば，開論理式ではそれを解釈することはできないが，通常は，限量されていない変数は全称限量されているものとして解釈される．項とは，真，偽，論理定項，論理変項，そして関数表現のことである．述語がとる項の数は**アリティ**(arity)と呼ばれる．特別に，項数 1 は**ユナリ**(unary)，項数2は**バイナリ**(binary)と呼ばれる．たとえば， $Mammal(x)$ や $Has(x, hair)$ は原子文であり，前者はユナリ，後者はバイナリである．先に，述語とは複数オブジェクトの関係を示すものと説明したが，ユナリの述語はそのオブジェクトの属性を記述する．論理定項は**リテラル**とも呼ばれるが，リテラルばかりで論理変項を含まない項は**基底項** (ground term)と呼ばれる．基底項ばかりで論理変項を含まない文は**基底文** (ground sentence) と呼ばれる．

原子基底文の真偽は中に含まれるリテラルが世界に照らして何を指示 (denote) しているかという**解釈**(図28.1参照）によって決められる．原子文の真偽値が決まれば，複合文の真偽値は命題論理と同様の演算で決めることができる．たとえば，$Animal(Bo) \wedge Has(Bo, hair) \Rightarrow Mammal(Bo)$という複合文は， 論理定項 Bo が元オバマ大統領の家族の犬のことを指し，$hair$が体毛のことを指すとすれば，事実に照らして $Mammal(Bo)$ は真でありこの論理式自体も真となる．論理定項 Bo がボー・ジャクソンのことと解釈しても同様である．もしhairが映画のことであれば，犬は映画を持っていないし，ボー・ジャクソンも多分持っていないだろうから，事実に照らして前件部は偽となるが（それは別の証拠から$Mammal(Bo)$ が真であることを妨げないし，偽となることも妨げない），命題論理の場合と同様にこの論理式自体は真である．Bo をボリビアの国名コードと解釈すると，$Mammal(Bo)$ は偽であるが．それでも相変わらずこの論理式としては真である．一方，$\exists x \{Animal(x) \wedge Has(x, hair) \Rightarrow Mammal(x)\}$ という論理式は,世界にそういう x が（一つでも）いるということを主張しており，他方， $\forall x \{Animal(x) \wedge Has(x, hair) \Rightarrow Mammal(x)\}$ では哺乳類のみならず，動物以外の x についても世界のすべてのオブジェクトについてそれが動物でかつ体毛があるなら哺乳類であると主張している．ここでも動物以外においても論理式自体は真となることを注意しておく．こうして，述語論理式の真偽はリテラルあるいは論理変項が世界の何を指示しているかで決められるが，複数の論理式全体は世界はかく配置されているということの記述であり，主張でもある．

論理変項または論理定項が世界の何を指示するにしても，その論理式が常に真となるものは**恒真式**または**トートロジー**と呼ばれ，常に偽となるものは**恒偽式**または**充足不能**と呼ばる．指示するものによって真となったり偽となったりするものは**充足可能**と呼ばれる．

30.2 一階述語論理の実際

述語論理プログラミングは人工知能の歴史の中でも比較的新しい分野である．PAIP では Prolog はあっても述語論理のプログラム提示はない．本章では最初にAIMAのLispコードの中から一階述語に関連する部分のコードを紹介し，次に、現在最も一般的で実用に近い述語論理プログラムと思われる SRI が長年開発してきた SNARK について紹介する．

30.2.1 CLIF の Lisp 構文

CLIF(Common Logic Interchage Format) というのはISO国際標準である Common Logic (ISO/IEC 24707:2007 および ISO/IEC 24707:2018) に定められた，一階述語論理のための記述構文である。同類のものとしてはそのほかに，CGIF (Conceptual Graph Interchange Format) と XCL (eXtended Common Logic Markup Language) がある．Common Logic の目的はかつて「知識交換フォーマット」として提案されたKIF(Knowledge Interchange Format)とは異なり，構文的に定義されたインターリングアではなく，単一の共通の意味的枠組みを提供することとされているが，実際に国際標準となっているのは、CLIFでもその他の記法でも一階述語論理の書き方であり、その意味論ではない．

実際にトップオントロジーの国際標準となっているBFO(Basic Formal Ontology, ISO/IEC 21838-2:2021)ではオントロジーの意味記述には，自然言語か Common Logic を用いることとなっていて、述語論理によるあいまいさのない記述が期待されてはいるが，その推論器は規定されず、外部の任意の推論システムを（勝手に）用いてよいことになっている．

30.2.2 AIMAのLispコード

ここで紹介するのはテキストにあるような論理式をLispで解釈可能な式に変換するプログラムである．すなわち，下図の Infix のような論理表記の文字列から Prefix のようなLisp論理表現を生成する．

本項では国際標準となっている Common Logic のうちLispのS式と非常に近しい CLIF 構文について説明する。

```
Infix          Prefix              Meaning              Alternate Infix Notation
==========     ======              ===========          ========================
~P             (not P)             negation             not P
P ^ Q          (and P Q)           conjunction          P and Q
P | Q          (or P Q)            disjunction          P or Q
P => Q         (=> P Q)            implication
P <=> Q        (<=> P Q)           logical equivalence
P(x)           (P x)               predicate
Q(x,y)         (Q x y)             predicate with multiple arguments
f(x)           (f x)               function
f(x)=3         (= (f x) 3)         equality
forall(x){P(x)}(forall(x) (P x)) universal quantification
```

```
exists(x,P(x) (exists (x) (P x)) existential quantification
[a,b]         (listof a b)       list of elements
{a,b}         (setof a b)        mathematical set of elements
true          true               the true logical constant
false         false              the false logical constant
```

すると $\forall x\{Animal(x) \wedge Has(x, hair) \Rightarrow Mammal(x)\}$ という論理式は Infix 文字列表記では "forall x(Animal(x) ^ Has(x,Hair) => Mammal(x)"であるが，Lisp 向きの prefix 表記では (forall(x)(=> (and (Animalx) (Has x Hair)) (Mammal x))

30.2.3 SNARKのダウンロードとインストール

SNARK - SRI's New Automated Reasoning KitのCurrent versionからSNARK最新版をダウンロードして解凍し，READMEを見て以下のように実行する．

ダウンロードし，解凍したフォルダに行き， SBCL を起動する．snark-system.lisp をロードしたのち，make-snark-system を実行する．テスト用にいくつかのプログラムが用意されているので，それらのうちの任意のものを実行する．

```
C:\Users\Seiji\Downloads\snark>sbcl
This is SBCL 2.0.0, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.

WARNING: the Windows port is fragile, particularly for multithreaded
code.  Unfortunately, the development team currently lacks the time
and resources this platform demands.
CL-USER(1): :ld snark-system.lisp
loading "snark-system.lisp"
CL-USER(2): (make-snark-system)
STYLE-WARNING:
    .......
    <省略>
    .......

; file: C:/Users/Seiji/Downloads/snark/src/unify-bag.lisp
    .......
    <省略>
    .......

; file: C:/Users/Seiji/Downloads/snark/src/eval.lisp
    .......
    <省略>
    .......

; file: C:/Users/Seiji/Downloads/snark/src/eval.lisp
    .......
    <省略>
    .......

; file: C:/Users/Seiji/Downloads/snark/src/eval.lisp
```

```
    .......
    <省略>
    .......

; file: C:/Users/Seiji/Downloads/snark/src/eval.lisp
    .......
    <省略>
    .......

; file: C:/Users/Seiji/Downloads/snark/src/eval.lisp
    .......
    <省略>
    .......
;
; compilation unit finished
;   caught 8 STYLE-WARNING conditions

; Running SNARK from C:/Users/Seiji/Downloads/snark/snark-system.lisp in SBCL 2.0.0 (64-bit)
on HP-ELITEDESK at 2021-04-28T12:25:59
NIL
SNARK-USER(3): (steamroller-example)

; Running SNARK from C:/Users/Seiji/Downloads/snark/snark-system.lisp in SBCL 2.0.0 (64-bit)
on HP-ELITEDESK at 2021-04-28T12:26:18
; The current SNARK option values are
;       (USE-RESOLUTION NIL)
;    (USE-HYPERRESOLUTION T)
;       (USE-NEGATIVE-HYPERRESOLUTION NIL)
;       (USE-UR-RESOLUTION NIL)
;       (USE-UR-PTTP NIL)
;       (USE-PARAMODULATION NIL)
;       (USE-FACTORING NIL)
;       (USE-EQUALITY-FACTORING NIL)
;       (USE-CONDENSING T)
;       (USE-RESOLVE-CODE NIL)
;       (USE-UNIT-RESTRICTION NIL)
;       (USE-INPUT-RESTRICTION NIL)
;       (USE-LITERAL-ORDERING-WITH-RESOLUTION NIL)
;       (USE-LITERAL-ORDERING-WITH-HYPERRESOLUTION NIL)
;       (USE-LITERAL-ORDERING-WITH-NEGATIVE-HYPERRESOLUTION NIL)
;       (USE-LITERAL-ORDERING-WITH-UR-RESOLUTION NIL)
;       (USE-LITERAL-ORDERING-WITH-PARAMODULATION NIL)
;       (USE-SUBSUMPTION T)
;       (USE-SUBSUMPTION-BY-FALSE :FALSE)
;       (USE-SIMPLIFICATION-BY-UNITS T)
;       (USE-SIMPLIFICATION-BY-EQUALITIES T)
;       (USE-TERM-ORDERING :RPO)
;       (USE-DEFAULT-ORDERING T)
;       (1-ARY-FUNCTIONS>2-ARY-FUNCTIONS-IN-DEFAULT-ORDERING NIL)
;       (ORDERING-FUNCTIONS>CONSTANTS NIL)
;       (RPO-STATUS :MULTISET)
;       (KBO-STATUS :LEFT-TO-RIGHT)
;       (USE-INDEFINITE-ANSWERS NIL)
;       (USE-CONDITIONAL-ANSWER-CREATION NIL)
;       (USE-CONSTRAINT-SOLVER-IN-SUBSUMPTION NIL)
;       (ALLOW-SKOLEM-SYMBOLS-IN-ANSWERS T)
;       (REWRITE-ANSWERS NIL)
;       (USE-CONSTRAINT-PURIFICATION NIL)
;       (USE-FUNCTION-CREATION NIL)
```

```
;       (USE-REPLACEMENT-RESOLUTION-WITH-X=X NIL)
;       (USE-PARAMODULATION-ONLY-INTO-UNITS NIL)
;       (USE-PARAMODULATION-ONLY-FROM-UNITS NIL)
;       (USE-SINGLE-REPLACEMENT-PARAMODULATION NIL)
;       (ASSERT-CONTEXT :ROOT)
;       (ASSERT-SUPPORTED T)
;       (ASSUME-SUPPORTED T)
;       (PROVE-SUPPORTED T)
;       (ASSERT-SEQUENTIAL NIL)
;       (ASSUME-SEQUENTIAL NIL)
;       (PROVE-SEQUENTIAL NIL)
;       (NUMBER-OF-GIVEN-ROWS-LIMIT NIL)
;       (NUMBER-OF-ROWS-LIMIT NIL)
;       (AGENDA-LENGTH-BEFORE-SIMPLIFICATION-LIMIT 10000)
;       (AGENDA-LENGTH-LIMIT 3000)
;       (RUN-TIME-LIMIT NIL)
;       (ROW-WEIGHT-LIMIT NIL)
;       (ROW-WEIGHT-BEFORE-SIMPLIFICATION-LIMIT NIL)
;       (LEVEL-PREF-FOR-GIVING NIL)
;       (AGENDA-ORDERING-FUNCTION ROW-PRIORITY)
;       (PRUNING-TESTS (ROW-WEIGHT-LIMIT-EXCEEDED))
;       (PRUNING-TESTS-BEFORE-SIMPLIFICATION (ROW-WEIGHT-BEFORE-SIMPLIFICATION-LIMIT-
EXCEEDED))
;       (USE-CLAUSIFICATION T)
;       (USE-EQUALITY-ELIMINATION NIL)
;       (USE-MAGIC-TRANSFORMATION NIL)
;       (USE-AC-CONNECTIVES T)
;       (USE-PURITY-TEST NIL)
;       (USE-RELEVANCE-TEST NIL)

(Row SNAILS-ARE-SMALLER-THAN-BIRDS
   (M ?X.SNAIL ?Y.BIRD)
   ASSERTION)
(Row BIRDS-ARE-SMALLER-THAN-FOXES
   (M ?X.BIRD ?Y.FOX)
   ASSERTION)
(Row FOXES-ARE-SMALLER-THAN-WOLVES
   (M ?X.FOX ?Y.WOLF)
   ASSERTION)
(Row WOLVES-DONT-EAT-FOXES
   (NOT (E ?X.WOLF ?Y.FOX))
   ASSERTION)
(Row WOLVES-DONT-EAT-GRAIN
   (NOT (E ?X.WOLF ?Y.GRAIN))
   ASSERTION)
(Row BIRDS-EAT-CATERPILLARS
   (E ?X.BIRD ?Y.CATERPILLAR)
   ASSERTION)
(Row BIRDS-DONT-EAT-SNAILS
   (NOT (E ?X.BIRD ?Y.SNAIL))
   ASSERTION)
(Row CATERPILLARS-EAT-SOME-PLANTS
   (E ?X.CATERPILLAR (SKOLEMAJKP1.PLANT ?X.CATERPILLAR))
   ASSERTION)
(Row SNAILS-EAT-SOME-PLANTS
   (E ?X.SNAIL (SKOLEMAJKP2.PLANT ?X.SNAIL))
   ASSERTION)
(Row 10
   (OR (E ?X.ANIMAL ?Y.ANIMAL) (E ?X.ANIMAL ?Z.PLANT)
```

```
       (NOT (M ?Y.ANIMAL ?X.ANIMAL)) (NOT (E ?Y.ANIMAL ?U.PLANT)))
   ASSERTION)
(Row 11
   (OR (NOT (E ?X.ANIMAL ?Y.ANIMAL)) (NOT (E ?Y.ANIMAL ?Z.GRAIN)))
   NEGATED_CONJECTURE
   Answer (VALUES ?X.ANIMAL ?Y.ANIMAL ?Z.GRAIN))
(Row 12
   (E ?X.BIRD ?Y.PLANT)
   (REWRITE (HYPERRESOLVE 10 SNAILS-ARE-SMALLER-THAN-BIRDS SNAILS-EAT-SOME-PLANTS) BIRDS-
DONT-EAT-SNAILS))
(Row 13
   (OR (E ?X.FOX ?Y.BIRD) (E ?X.FOX ?Z.PLANT))
   (HYPERRESOLVE 10 12 BIRDS-ARE-SMALLER-THAN-FOXES))
(Row 14
   (E ?X.FOX ?Y.PLANT)
   (HYPERRESOLVE 11 13 12)
   Answer (VALUES ?X.FOX ?Z.BIRD ?U.GRAIN))
(Row 15
   (OR (E ?X.FOX ?Y.BIRD) (E ?Z.WOLF ?U.PLANT))
   (REWRITE (HYPERRESOLVE 10 13 FOXES-ARE-SMALLER-THAN-WOLVES) WOLVES-DONT-EAT-FOXES))
(Row 16
   (E ?X.WOLF ?Y.PLANT)
   (REWRITE (HYPERRESOLVE 10 14 FOXES-ARE-SMALLER-THAN-WOLVES) WOLVES-DONT-EAT-FOXES)
   Answer (VALUES ?Z.FOX ?U.BIRD ?V.GRAIN))
(Row 17
   FALSE
   (HYPERRESOLVE WOLVES-DONT-EAT-GRAIN 16)
   Answer (VALUES ?X.FOX ?Y.BIRD ?Z.GRAIN))

(Refutation
(Row SNAILS-ARE-SMALLER-THAN-BIRDS
   (M ?X.SNAIL ?Y.BIRD)
   ASSERTION)
(Row BIRDS-ARE-SMALLER-THAN-FOXES
   (M ?X.BIRD ?Y.FOX)
   ASSERTION)
(Row FOXES-ARE-SMALLER-THAN-WOLVES
   (M ?X.FOX ?Y.WOLF)
   ASSERTION)
(Row WOLVES-DONT-EAT-FOXES
   (NOT (E ?X.WOLF ?Y.FOX))
   ASSERTION)
(Row WOLVES-DONT-EAT-GRAIN
   (NOT (E ?X.WOLF ?Y.GRAIN))
   ASSERTION)
(Row BIRDS-DONT-EAT-SNAILS
   (NOT (E ?X.BIRD ?Y.SNAIL))
   ASSERTION)
(Row SNAILS-EAT-SOME-PLANTS
   (E ?X.SNAIL (SKOLEMAJKP2.PLANT ?X.SNAIL))
   ASSERTION)
(Row 10
   (OR (E ?X.ANIMAL ?Y.ANIMAL) (E ?X.ANIMAL ?Z.PLANT)
       (NOT (M ?Y.ANIMAL ?X.ANIMAL)) (NOT (E ?Y.ANIMAL ?U.PLANT)))
   ASSERTION)
(Row 11
   (OR (NOT (E ?X.ANIMAL ?Y.ANIMAL)) (NOT (E ?Y.ANIMAL ?Z.GRAIN)))
   NEGATED_CONJECTURE
   Answer (VALUES ?X.ANIMAL ?Y.ANIMAL ?Z.GRAIN))
```

```
(Row 12
   (E ?X.BIRD ?Y.PLANT)
   (REWRITE (HYPERRESOLVE 10 SNAILS-ARE-SMALLER-THAN-BIRDS SNAILS-EAT-SOME-PLANTS) BIRDS-
DONT-EAT-SNAILS))
(Row 13
   (OR (E ?X.FOX ?Y.BIRD) (E ?X.FOX ?Z.PLANT))
   (HYPERRESOLVE 10 12 BIRDS-ARE-SMALLER-THAN-FOXES))
(Row 14
   (E ?X.FOX ?Y.PLANT)
   (HYPERRESOLVE 11 13 12)
   Answer (VALUES ?X.FOX ?Z.BIRD ?U.GRAIN))
(Row 16
   (E ?X.WOLF ?Y.PLANT)
   (REWRITE (HYPERRESOLVE 10 14 FOXES-ARE-SMALLER-THAN-WOLVES) WOLVES-DONT-EAT-FOXES)
   Answer (VALUES ?Z.FOX ?U.BIRD ?V.GRAIN))
(Row 17
   FALSE
   (HYPERRESOLVE WOLVES-DONT-EAT-GRAIN 16)
   Answer (VALUES ?X.FOX ?Y.BIRD ?Z.GRAIN))
)

; Summary of computation:
;        17 formulas have been input or derived (from 16 formulas).
;        17 (100%) were retained.  Of these,
;           17 (100%) are still being kept.
;
; Run time in seconds:
;     0.000   0%   Assert                       (11 calls)
;     0.000   0%   Process new row              (17 calls)
;     0.000   0%   Resolution                   (15 calls)
;     0.000   0%   Condensing                   (3 calls)
;     0.000   0%   Forward subsumption          (7 calls)
;     0.000   0%   Backward subsumption         (7 calls)
;     0.000   0%   Clause clause subsumption    (6 calls)
;     0.000   0%   Forward simplification       (17 calls)
;     0.000   0%   Backward simplification      (17 calls)
;     0.000   0%   Sortal reasoning             (450 calls)
;     0.000   0%   Other
;     0.000        Total
;     0.068        Real time
;
; Term-hash-array has 26 terms in all.
; Feature-vector-row-index has 7 entries (7 at peak, 7 added, 0 deleted).
; Feature-vector-row-index has 6 nodes (6 at peak, 6 added, 0 deleted).
;  Retrieved 3 possibly forward subsuming rows in 6 calls.
;  Retrieved 9 possibly backward subsumed rows in 6 calls.
; Path-index has 22 entries (22 at peak, 22 added, 0 deleted).
; Path-index has 19 nodes (19 at peak, 19 added, 0 deleted).
; Trie-index has 22 entries (22 at peak, 22 added, 0 deleted).
; Trie-index has 15 nodes (15 at peak, 15 added, 0 deleted).
; Retrieved 85 generalization terms in 23 calls.
; Retrieved 29 instance terms in 10 calls.
; Retrieved 149 unifiable terms in 29 calls.
;
; The agenda of rows to give has 1 entry:
;     1 with value (4 11)
:PROOF-FOUND
SNARK-USER(4):
```

当初はこれまでの他の章と同様に，コードウォークスルーをしようと思ったが，SNARKの使用法を学ぶだけでも大変であり，コードの説明などはとても不可能であることがわかった．しかし，SNARKは等号（equality）を考慮した定理証明システムであり，デモジュレーション・パラモジュレーション（AIMA第３版，pp354参照）を含む反駁法（reftaion）を学ぶには格好の素材を提供してくれている．

第31章 プラニング

> 行動についての推論は知的行為に必須の要素である．人はほとんど対話することなく，とった行動が周りの世界にどんな影響を及ぼすかを考えずにお店に行ったりできる．しかし，未知の状況では人々はしばしば顕著な努力を傾けて，ことのなりゆきについてよく考える．人工知能と関連分野における数十年にわたる研究において，この特定領域の人間の持つ能力を形式化することは非常に難しかった．しかしながら，行動についての推論は知的行為の本質的なコンポーネントであり，未来の知的計算機システムに必須の部分である．
>
> [David E. Wilkins, Practical Planning, 1988]

> 目的的行為というのはプランで決定されるというものだが，合理的行為者の正しいモデルとして，西欧の人間科学に深く根づいている．今日，認知科学と呼ばれていることに貢献する領域に対して，プランの分析と統合が行為の研究を効果的に構成しているかぎりでは，行為の計算論的モデルを構築するために，プランの論理的な形式は魅力的なものとなっている．しかしながら，私の主張は，人間の行為の研究者として，私たちはトラック島の航海者を無視することで危険を冒しているということである．ヨーロッパの航海者がいかに航海するかということがわかったにせよ，それがいかにプランに従ったものであろうとプランによらないものであろうと，行為の本質的特徴は状況に埋め込まれたものなのである．
>
> [Lucy A. Suchman, Plans and Situated Actions, 1987](プランと状況的行為, 佐伯監訳)

プラニングは人工知能の歴史の初期から非常に重要な応用分野の一つであったが，宇宙や軍事以外の現実の社会において大いに成功したという例は，スケジューリングは別としてまだない．それは複雑なオープンワールドにおけるプラニング研究がまだ進んでいないということの反映である．PAIPにはプラニングの記述がなく，AIMAでも第１版には古典的プラニングの疑似コードはあってもLispの実コードはなく，第3版においてPlanning and Acting in the Real Worldという章が新しく立てられたという具合である．第23章で紹介したように，プラニングの歴史は一般問題解決器(GPS)とSTRIPSに始まるが，今日ではプラニングは状況計算によるしっかりとした論理的ベースの上にその形式化がなされている．本書では今日最も優れた形式的かつ包括的なプランニングの教科書と思われるAutomated Planning theory and practice の内容に従った古典的プラニングのCommon Lispコードを示す．

31.1 状態空間プランニング

31.1.1 前向き探索

Malik Ghallab とDana Nau と Paolo Traverso によるAutomated Planning theory and practiceの最初は状態空間についてのプラニング記述である．状態空間法では23章と同様に，世界の状態を探索空間として，動作ごとの状態変化を追跡する．前向き探索では，初期状態から出発し，STRIPS と同様にプレコンディションを見て実施可能なアクションが見つかればそれを実行する．アクション実行により世界の状態が変わるのでこれを繰り返して，ゴールに到達すれば探索成功，すなわちプラニング成功である．図31.1はGhallab, Nau，Traversoによる前向き探索のアルゴリズムである．

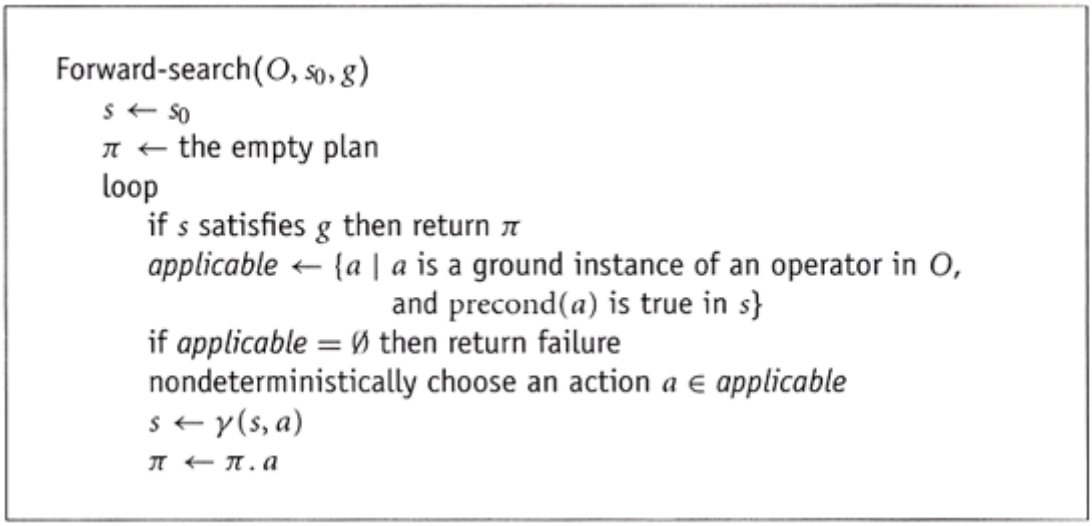


図31.1 前向き探索アルゴリズム

ここで，O はオペレーション集合，s_0 は初期状態，g はゴールである．s が状態であり，π はアクションの履歴である．γ は状態 s にアクション a を実施した時の状態変化の計算である．

これをそのままLispのコードに起こしたものが以下である．

```
(=defun forward-search (operators state goal plan)
  "This program searches <state> that satisfies <goal>, and returns an
   accumulated sequence of actions or <plan> that achieves <goal>.
   Starting at initial <state>, finding actions that are applicable to
   <state>, and this continuable function nondeterministically chooses one of
   applicable actions, then progresses <state> with the application of the
   action, and recursively calls with new <state> and partial <plan>.
   Here, <state> is a set of ground statevars but <goal> may include
   variables. If there is no applicable action, then the search is failed and
   the program control backtracks up to the next choice of applicable
   actions. Exactly, ''forward-search'' is a macro name and
   ''=forward-search'' is actually a function name. Both together pretends to
   support the continuation like Scheme. See callcc module in detail."
  (if (satisfy-p state goal) (=values (reverse plan))
    (let ((applicable
            (remove-if-not #'(lambda (a) (applicable-action-p a state))
                           (mappend #'(lambda (o)
                                        (operator-unify-for-progression
                                          o state +no-bindings+))
                                    operators))))
      (setq applicable (remove-action-occured applicable plan))
      (if (null applicable) (fail)
        (choose-bind action applicable
                     (format t "~&Action taken ~S" (action-name action))
                     (forward-search operators
                                     (progress-action action state)
                                     goal
                                     (cons action plan)))))))
```

ここで，継続関数 =defunと=valuesを使っていることに注意されたい．choose-bindが図31.1の“nondeterministically chooses an action”に相当する部分である．バックトラックを含んだ非決定的プログラミングの機能のおかげで，アルゴリズムの姿をほとんど保ったままコード化することができた．

satisfy-p はゴールにある正リテラルがすべて状態 s にあり，負リテラルが状態 s に正リテラルとしてはなく，ゴールにある正リテラルが状態 s には負リテラルとしてないとき，そのゴールはその状態に満足される．局所関数 satisfy-p-with-bindings-accumulation の中で compound-unify を使っているが，これは 29.2節で説明した複合物の単一化プログラムである．状態空間にたいするそれはこのあとの状態空間のプログラムで説明する．

```
(defun satisfy-p (state goal)
  "tests <state> satisfies <goal>, and returns unifiable bindings if so,
   otherwise returns false. Note that <goal> may include variables any
   time, <state> does not include any variables in forward search but may
   include in backward search."
  (let ((bindings +no-bindings+)
        (positive-goal (remove-if-not #'statevar-val goal))
        (negative-goal (remove-if     #'statevar-val goal))
        (positive-state (remove-if-not #'statevar-val state))
        (negative-state (remove-if     #'statevar-val state)))
    (flet ((satisfy-p-with-bindings-accumulation (gl st)
             ;; if unifiable updates bindings, otherwise no effects and returns nil
             (let ((result (compound-unify gl st bindings)))
               (unless (equal result +fail+)
                 (setq bindings result)))))
      (when (and (subsetp positive-goal positive-state
                          :test #'satisfy-p-with-bindings-accumulation)
                 (null (intersection negative-goal positive-state
                                     :test #'satisfy-p-with-bindings-accumulation))
                 (null (intersection positive-goal negative-state
                                     :test #'satisfy-p-with-bindings-accumulation)))
        bindings))))
```

operator-unify-for-progression は，引数にオペレータと状態を取り，オペレータのプレコンディションが状態と単一化できれば，単一化代入されたオペレータのアクションを返す．

```
(defun operator-unify-for-progression (operator state bindings)
  "computes multiple substitutions between <operator>'s precond and <state>,
   then instantiates <operator> with computed substitutions. Note that this
   function computes possible multi-substitutions for <operator> and <state>,
   and such multi-substitutions generate possible multi-instantiations in
   pallarel, so this function returns a set of completely instantiated
   <operator>s."
  (mapcar #'(lambda (bindings) (make-action-from operator bindings))
    (precond-unify (operator-precond operator) state bindings)))
```

precond-unify は内部でプレコンディションの正コンディションについて単一化を行う statevar-unify と，負コンディションについて単一化を行う opposite-statevar-unify を呼び出す．

```
(defun precond-unify (precond state bindings)
  "returns a list of possible new bindings between <precond> and <state>.
   Every positive <precond> must meet <state> with unification, and every
   negative <precond> must not meet <state> with binding constraint.
   So, positive <precond> produces possible bindings and negative <precond>
   filters unsatisfiable bindings."
  (let ((positive-precond (remove-if-not #'statevar-val precond))
        (negative-precond (remove-if     #'statevar-val precond))
        (bindings-list (list bindings)))
```

```
    (loop for svar+ in positive-precond
        do (setq bindings-list
                 (mappend #'(lambda (bindings)
                              (statevar-unify svar+ state bindings))
                          bindings-list)))
    (setq bindings-list
          (remove-if #'(lambda (bindings)
                         (loop for svar- in negative-precond
                             when (setq bindings
                                        (opposite-statevar-unify
                                           svar- state bindings))
                               do (return t)))
                     bindings-list))))

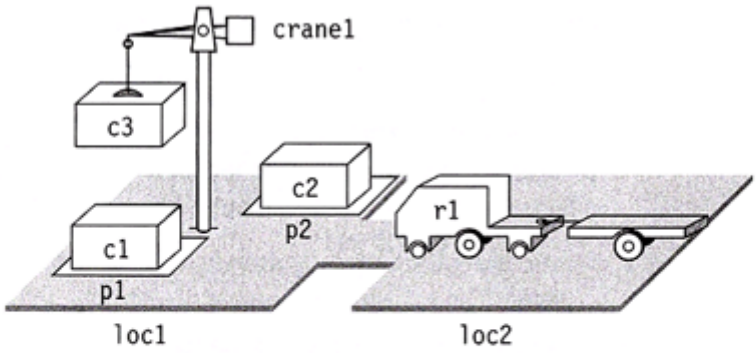
(defun statevar-unify (statevar state bindings)
  "returns a list of possible bindings of statevar against <state>.
   This function collects all possibilities for <state>."
  (loop for st in state with new-bindings
      unless (eq (setq new-bindings (compound-unify statevar st bindings))
                 +fail+)
      collect new-bindings))

(defun opposite-statevar-unify (statevar state bindings)
  "returns a list of possible bindings of positive(negative) <statevar>
   against negative(positive) <state>. If there is no unifiable element
   in <state> with <bindings> constraint, returns false."
  (loop for st in state with new-bindings
      unless (eq (setq new-bindings
                       (opposite-compound-unify statevar st bindings))
                 +fail+)
      collect new-bindings))
```

31.1.2 状態空間の論理記述

例として，図31.2にしめすような初期状態を考える．この状態を論理記述したのが以下の表現である．

attach(p1,loc1)：場所 1 はロケーション 1 にある
in(c1,p1)：コンテナ 1 は場所1の中にある
top(c1,p1)：
on(c1,pallet)：コンテナ 1 はパレットに載っている
attach(p2,loc1)：場所 2 はロケーション 1 にある
in(c2,p2)：コンテナ 2 は場所 2 の中にある
top(c2,p2)：
on(c2,pallet)：：コンテナ 2 はパレットに載っている
belong(crane1,loc1)：クレーン 1 は場所 1 に属している
holding(crane1,c3)：クレーン 1 はコンテナ 3 を持っている
adjacent(loc1,loc2)：場所 1 は場所 2 の隣に接している
adjacent(loc2,loc1)：場所 2 は場所 1 の隣に接している
at(r1,loc2)：ロボット1は場所 2 にいる
occupied(loc2)：場所 2 に置き場所はない
unloaded(r1)：ロボット1には荷物がない



The DWR state s_5 = {attached(p1,loc1), in(c1,p1), top(c1,p1), on(c1,pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), holding(crane1,c3), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)}.

図31.2 例題初期状態

ゴールはロボット1が場所 1 にあって `at(r1,loc1)`，コンテナ3を積んだ状態 `loaded(r1,c3)` であるとする．第30章述語論理では関数 `logic` によって，"load(r1)=c1" とか "~load(r1)=c1" という中置の論理表現を `(= (load r1) c1)` とか `(not (= (load r1) c1))` といったLispのS式に変換した．同様に状態空間における "at(r1,loc1)" という論理表現をLispでも扱いやすくすることを考える．

"at(r1,loc1)" という状態は時間 *t* におけるある状態において真であったり，偽であったりの論理値を持つと考える．そういう意味で以下ではこれを状態変数 statevar と呼ぶこととする．これを以下のような構造体で表す．ここで `symbol` は "at" のような述語であり，`args` がその引数リスト，`val` がその時々で真であったり偽であったりする論理値である．関数 `statevar` は文字列として"at(r1,loc1)" というような論理表現を引き受けて，構造体 `statevar` のインスタンスを作り出す．

ちょっと，やってみよう．

```
cg-user(1): (in-package :plan)
#<The plan package>
plan(2): (setq foo (statevar "at(r1,loc1)"))
at(r1 loc1)
plan(3): (describe foo)
at(r1 loc1) is a structure of type statevar.  It has these slots:
  symbol at
  args   (r1 loc1)
  val    t
plan(4): (setq bar (statevar "~at(r1,loc1)"))
~at(r1 loc1)
plan(5): (describe bar)
~at(r1 loc1) is a structure of type statevar.  It has these slots:
  symbol at
  args   (r1 loc1)
  val    nil
```

行番号2の実行結果で statevar のインスタンスが作られるが，その印刷結果をこのように論理表現に似せているのが，この構造体の印刷関数 print-statevar だ．それが本当に statevar のインスタンスだということは行番号3の結果を見ればわかる．これは正のリテラルだが負のリテラルの場合が，行番号4以下のものだ．さらに， "at(r1,loc1)" を "at(r1)=loc1" とも表記できるようにして，その意味するところは同じとする．

```
plan(5): (statevar-equalp (statevar "at(r1,loc1)") (statevar "at(r1)=loc1"))
t
plan(6): (describe (statevar "at(r1)=loc1"))
at(r1)=loc1 is a structure of type statevar.  It has these slots:
  symbol at
  args   (r1)
  val    loc1
```

```
(defstruct (statevar (:print-function print-statevar))
  symbol args val)

(defun print-statevar (x stream depth)
  "prints statevar <x> in infix notation."
  (declare (ignore depth))
  (case (statevar-val x)
    ((t) (format stream "~S~:[~;~:*~S~]"
                        (statevar-symbol x) (statevar-args x)))
    ((nil) (format stream "~~~S~:[~;~:*~S~]"
                          (statevar-symbol x) (statevar-args x)))
    (otherwise (format stream "~S~:[~;~:*~S~]=~S"
                       (statevar-symbol x) (statevar-args x) (statevar-val x)))))

(defun statevar (exp)
  "makes a statevar instance. If <exp> is a string, the infix expression is
   converted to a prefix expression and recursively called."
  (if (stringp exp) (statevar (logic exp))
    (case (op exp)
      (not (case (op (arg1 exp))
             (=                          ; (not (= x y))
              (make-statevar :symbol (op (arg1 (arg1 exp)))
                             :args (append (args (arg1 (arg1 exp)))
                                           (list (arg2 (arg1 exp))))
                             :val nil))
             (otherwise (make-statevar :symbol (op (arg1 exp))
                                       :args (args (arg1 exp))
                                       :val nil))))
      (= (let ((val (arg2 exp)))
           (when (eq val 'true) (setq val t))
           (when (eq val 'false) (setq val nil))
           (make-statevar :symbol (op (arg1 exp))
                          :args (args (arg1 exp))
                          :val val)))
      (otherwise (make-statevar :symbol (op exp)
                                :args (args exp)
                                :val t)))))

(defun negative-statevar-p (statevar)
  "returns true if <statevar> is negative."
  (null (statevar-val statevar)))
```

```
(defun positive-statevar-p (statevar)
  "returns true if <statevar> is positive."
  (statevar-val statevar))

(defun single-var-p (statevar)
  "returns true if <statevar> has only one state variable, or
   if <statevar> has two arguments in unnormalized expression.
   See <normalize-statevar> and <unnormalize-statevar>."
  (length=1 (statevar-args (unnormalize-statevar statevar))))
```

波括弧でくくられた状態変数の並びをLisp表現に変換するのが関数 state である.

```
plan(7): (state "{at(r1)=l1,~load(r1),cpos(c1)=l1,cpos(c2)=l2}")
(at(r1)=l1 ~load(r1) cpos(c1)=l1 cpos(c2)=l2)
```

```
(defun state (exp)
  "makes and returns a list of statevar instances. The <exp> should be either
   a string in infix notation or a list in prefix notation. A sequence of literals
   in infix notation must be enveloped by braces."
  (if (stringp exp)
      (state (logic exp))
    (mapcar #'statevar exp)))
```

このシステムの論理表現ではアルファベット1文字のシンボルは論理変数としており，Lisp表現に変換されたときには頭に"$"記号を付けたシンボルに変換される．論理変数を含む状態変数は，関数 statevar-unify を使って，状態中の変数のないリテラルと単一化できる.

```
plan(8): (setq foo (statevar "at(r1,l)"))
at(r1 $l)
plan(9): (describe foo)
at(r1 $l) is a structure of type statevar.  It has these slots:
  symbol at
  args   (r1 $l)
  val    t
plan(10): (statevar-unify (statevar "at(r,l)") (state "{at(r1,l1),at(r2,l2)}")
                                    +no-bindings+)
((($l . l1) ($r . r1)) (($l . l2) ($r . r2)))
```

```
(defun statevar-unify (statevar state bindings)
  "returns a list of possible bindings of statevar against <state>.
   This function collects all possibilities for <state>."
  (loop for st in state with new-bindings
      unless (eq (setq new-bindings (compound-unify statevar st bindings)) +fail+)
      collect new-bindings))

(defun opposite-statevar-unify (statevar state bindings)
  "returns a list of possible bindings of positive(negative) <statevar> against
   negative(positive) <state>.
   If there is no unifiable element in <state> with <bindings> constraint,
   returns false."
  (loop for st in state with new-bindings
      unless (eq (setq new-bindings (opposite-compound-unify statevar st bindings))
                 +fail+)
      collect new-bindings))

(defun precond-unify (precond state bindings)
  "returns a list of possible new bindings between <precond> and <state>.
   Every positive <precond> must meet <state> with unification, and every
```

```
   negative <precond> must not meet <state> with binding constraint.
   So, positive <precond> produces possible bindings and negative <precond>
   filters unsatisfiable bindings."
  (let ((positive-precond (remove-if-not #'statevar-val precond))
        (negative-precond (remove-if     #'statevar-val precond))
        (bindings-list (list bindings)))
    (loop for svar+ in positive-precond
        do (setq bindings-list
                 (mappend #'(lambda (bindings)
                              (statevar-unify svar+ state bindings))
                          bindings-list)))
    (setq bindings-list
          (remove-if #'(lambda (bindings)
                         (loop for svar- in negative-precond
                             when (setq bindings
                                        (opposite-statevar-unify
                                           svar- state bindings))
                             do (return t)))
                     bindings-list))))
```

31.1.3 オペレータの定義

オペレータは名前とコメントとプレコンディションとエフェクトを有する構造体であるが，プラニング推論で重要なのはオペレーション実行の前提条件となるプレコンディションとそのオペレーション実行による状態変化エフェクトである．関数 operator は引数にオペレーション名，プレコンディション，エフェクトを引き受けて，構造体 operator のインスタンスを作り出す．

```
(defstruct (operator (:print-function print-operator))
                     name comment precond effects)
(defun print-operator (x stream depth)
  "prints <operator> such as ''(operator <name> <comment>
   (:precond <precond>) (:effects <effects>)''."
  (declare (ignore depth))
  (cond ((eq (operator-symbol x) 'start)
         (format stream "~<(start~;~:1I~{ ~W~:_~}~;)~:>"
                 (list (operator-effects x))))
        ((eq (operator-symbol x) 'finish)
         (format stream "~<(finish~;~:1I~{ ~W~:_~}~;)~:>"
                 (list (operator-precond x))))
        ((operator-comment x)
         (format stream "~<(operator ~;~W ~_~:I~W ~_~W ~_~W~;)~:>"
           (list (operator-name x) (operator-comment x)
                 (cons :precond (operator-precond x))
                 (cons :effects (operator-effects x)))))
        (t (format stream "~<(operator ~;~W ~_~:I~W ~_~W~;)~:>"
                   (list (operator-name x)
                         (cons :precond (operator-precond x))
                         (cons :effects (operator-effects x)))))))

(defun operator-symbol (operator)
  "returns <ope-name-symbol> of <operator>'s name."
  (ope-name-symbol (operator-name operator)))
(defun operator-variables (operator)
  "returns <ope-name-parms> of <operator>'s name."
  (ope-name-parms (operator-name operator)))
```

```
(defun operator (ope-name &key precond effects comment)
  "makes an instance of <operator> with <ope-name>, <precond>,
   <effects>, and <comment>, and returns it. Each parameter in <precond>
   and <effects> may be in prefix notation in lisp and in infix notation in string.
   <precond> and <effects> do not require to envelop it with braces.
   This function does not add the result into *operators*."
  (make-operator :name (ope-name ope-name)
                 :precond (mapcar #'normalize-statevar
                                  (mapcar #'statevar
                                    (if (stringp precond)
                                        (logic (with-brace precond))
                                      precond)))
                 :effects (mapcar #'normalize-statevar
                                  (mapcar #'statevar
                                    (if (stringp effects)
                                        (logic (with-brace effects))
                                      effects)))
                 :comment comment))

(defun operator-equal-p (operator1 operator2)
  "returns true if <operator1> and <operator2> are equal in the operator definition."
  (or (equalp operator1 operator2)
      (and (operator-p operator1) (operator-p operator2)
           (ope-name-equal-p (operator-name operator1) (operator-name operator2))
           (subsetp (operator-precond operator1) (operator-precond operator2)
                    :test #'statevar-equalp)
           (subsetp (operator-precond operator2) (operator-precond operator1)
                    :test #'statevar-equalp)
           (subsetp (operator-effects operator1) (operator-effects operator2)
                    :test #'statevar-equalp)
           (subsetp (operator-effects operator2) (operator-effects operator1)
                    :test #'statevar-equalp))))
```

マクロ defoperator は内部で関数 operator を呼び出すが，ユーザがREPLで使いやすくすると同時に大域変数 *operators* に定義したオペレータを保存する．

```
(defvar *operators* () "A global var that stores operators defined.")

(defmacro defoperator (ope-name &rest initargs)
  "defines an operator and pushes the result into *operators*."
  `(car
    (setq *operators*
          (cons (operator ,ope-name
                          :comment ,(if (stringp (car initargs)) (car initargs))
                          :precond ,(if (stringp (car initargs))
                                        (getf (cdr initargs) :precond)
                                      (getf initargs :precond))
                          :effects ,(if (stringp (car initargs))
                                        (getf (cdr initargs) :effects)
                                      (getf initargs :effects)))
                *operators*))))

(defun find-operator (symbol)
  "finds the operator of which operator symbol is <symbol> and returns it."
  (find symbol *operators* :test #'eq :key #'operator-symbol))

(defun positive-precond (precond)
  "retrieves positive literals from <precond> and returns it."
```

```
  (remove-if-not #'statevar-val precond))
(defun negative-precond (precond)
  "retrieves negative literals from <precond> and returns it."
  (remove-if #'statevar-val precond))

(defun positive-effects (effects precond)
  "retrieves positive literals from <effects> and returns it.
   <precond> is ignored."
  (declare (ignore precond))
  (remove-if-not #'statevar-val effects))

(defun negative-effects (effects precond)
  "retrieves negative literals from <effects> and returns it."
  (declare (ignore precond))
  (remove-if #'statevar-val effects))
```

それでは図31.2の例題について，前向き推論で計画してみよう．

```
plan(11): (setq *state*
       (state
        "{attach(p1,loc1),in(c1,p1),top(c1,p1),on(c1,pallet),
          attach(p2,loc1),in(c2,p2),top(c2,p2),on(c2,pallet),
          belong(crane1,loc1),holding(crane1,c3),adjacent(loc1,loc2),
          adjacent(loc2,loc1),at(r1,loc2),occupied(loc2),unloaded(r1)}"))
(attach(p1 loc1) in(c1 p1) top(c1 p1) on(c1 pallet) attach(p2 loc1) in(c2 p2) top(c2 p2)
on(c2 pallet) belong(crane1 loc1) holding(crane1 c3) ...)
plan(12): (defoperator "move(r,l,m)"
     "robot <r> moves from location <l> to an adjacent location <m>"
   :precond "adjacent(l,m),at(r,l),~occupied(m)"
   :effects "at(r,m),occupied(m),~occupied(l),~at(r,l)")
(operator move($r $l $m)
          "robot <r> moves from location <l> to an adjacent location <m>"
          (:precond adjacent($l $m) at($r $l) ~occupied($m))
          (:effects at($r $m) occupied($m) ~occupied($l) ~at($r $l)))
plan(13): (defoperator "load(k,l,c,r)"
     "crane <k> at location <l> loads container <c> onto robot <r>"
   :precond "belong(k,l),holding(k,c),at(r,l),unloaded(r)"
   :effects "empty(k),~holding(k,c),loaded(r,c),~unloaded(r)")
(operator load($k $l $c $r)
          "crane <k> at location <l> loads container <c> onto robot <r>"
          (:precond belong($k $l) holding($k $c) at($r $l) unloaded($r))
          (:effects empty($k) ~holding($k $c) loaded($r $c) ~unloaded($r)))
plan(14): (setq *goal* (state "{at(r1,loc1),loaded(r1,c3)}"))
(at(r1 loc1) loaded(r1 c3))
plan(15): (forward-search *operators* *state* *goal* ())
Action taken move(r1 loc2 loc1)
Action taken load(crane1 loc1 c3 r1)
((operator move(r1 loc2 loc1)
           "robot <r> moves from location <l> to an adjacent location <m>"
           (:precond adjacent(loc2 loc1) at(r1 loc2) ~occupied(loc1))
           (:effects at(r1 loc1) occupied(loc1) ~occupied(loc2) ~at(r1 loc2)))
 (operator load(crane1 loc1 c3 r1)
           "crane <k> at location <l> loads container <c> onto robot <r>"
           (:precond belong(crane1 loc1) holding(crane1 c3) at(r1 loc1) unloaded(r1))
           (:effects empty(crane1) ~holding(crane1 c3) loaded(r1 c3) ~unloaded(r1))))
```

第32章 Lispで作るProlog

> われわれの経験では，初級のプログラマは Prolog のプログラムが従来の言語によるプログラムより理解し易いと感じる．この人々は従来の言語が計算機の資源に対して置く制限を評価しない傾向がある．一方，従来の言語で経験をつんだプログラマは変数や流れのような抽象的概念を扱う準備ができている．しかし，このような経験にもかかわらず，この人々は Prolog に慣れるのは難しいと感じ，Prolog が有益なプログラミングの手段であると信じるまでには何回も納得させることが必要な場合がある．
>
> [W.F. Clocksin, C.S. Mellish, 中村訳，Prologプログラミング, 1983]

PAIPの第11章は Logic Programming となっているが，そこで書かれているのは Lisp で実装する Prolog である． そしてPAIP第12章はそのコンパイルについて書かれている．第13章でちょっと論理とは離れてオブジェクト指向プログラミングになるけれども， PAIP第14章のタイトルこそ知識表現と推論となっているが，実のところ11章の拡張の話になっている． そして第19章の自然言語，20章のUnification Grammar，21章のGrammar of English はすべて，11章のPrologの応用と位置付けることができる．やはりLispとPrologは人工知能用プログラム言語の双璧なのである．自然言語までやるかどうかは，AIの中でも専門とする分野に分かれてくるところではあるが，AI に取り組む人は論理とPrologプログラミングは一通りは押さえておかなければならない基本である．

本書第30章では一般の一階述語論理について説明したが，Prologはそれにさらに以下のような制限を設けて，実用的な速度の論理計算を可能にした．

- 含意(implication)の帰結部(Prologではheadという)には複合文はこれない．

32.1 Prolog を実行する

cl-aip/Prologからプログラムをダウンロードしてダウンロードして Common Lisp環境にロードする．そしてトップレベルで次のように入力する

```
cl-user(1): (<- (hasHusband Marie_Curie Pierre_Curie))
hasHusband
cl-user(2): (<- (hasHusband Irene_Joliot-Curie Jean_Frederic_Joliot-Curie))
hasHusband
cl-user(3): (<- (hasMother Irene_Joliot-Curie Marie_Curie))
hasMother
cl-user(4): (<- (hasFather Irene_Joliot-Curie Pierre_Curie))
hasFather
cl-user(5): (<- (hasWife ?x ?y) (hasHusband ?y ?x))
hasWife
cl-user(6): (<- (loves ?x ?y) (hasWife ?x ?y))
loves
cl-user(7):  (<- (loves ?x ?y) (hasHusband ?x ?y))
loves
```

ここで "<-" に始まる一つの入力形式は Prolog では**節**(clause)と呼ばれる．上記例の1から4までの節は中に一つのリストしかなく，5から７までは二つのリストがあることに注意されたい．節の最初のリストは**頭部**(head)とよばれ，残りのリストは**本体**(body)と呼ばれるが，IF～THEN 形式のルール表現でかけば，それは次のような意味である．

```
IF body THEN head
```

つまり，Prologでは前件部には複数の書式を書くことができるが，帰結部には一つの書式しか書くことができない．上記のPrologの頭部が帰結部と解釈され，本体が前件部と解釈される．そして，頭部だけの書式は事実の宣言に相当する．だから，上記例の1から4まではマリー・キュリーとピエール・キュリーの夫婦関係と，ジョリオ・キュリー夫妻の関係と，その親子関係を言ったことになるのだ．上記5から７までのルールは次のような意味である．

`(<- (hasWife ?x ?y) (hasHusband ?y ?x))`：　ある y が夫 x を持っていれば，その夫 x はその y を妻としている．
`(<- (loves ?x ?y) (hasWife ?x ?y))`　　：　x が妻 y を持てば x はその y を愛している
`(<- (loves ?x ?y) (hasHusband ?x ?y))`　：　x が夫 y を持てば x はその y を愛している

ここで，hasHusbandやhasWifeは述語と呼ばれるが，述語は人間が見たらいかにも意味があるかのように思えるが，Prolog はもちろん人間と同じように意味を知っているわけではない．しかし，最初のルールは hasWife と hasHusband は逆の関係にあることをいい，2番目と3番目のルールは hasWife あるいは hasHusband の関係があれば loves の関係が導けることを表している．Prologはオブジェクト間の関係を記述する言語なのだ．また計算機は Marie_Curie というシンボルが実世界の何をさすのかということも一切わかっていないということを注意しておく．

これまで何回も色々なところで出ている動物のルールは，Prologでは次のように書くことができる．

```
cl-user(8): (<- (mammal ?animal) (has ?animal hair))
mammal
cl-user(9): (<- (mammal ?animal) (gives ?animal milk))
mammal
cl-user(10): (<- (bird ?animal) (has ?animal feathers))
bird
cl-user(11): (<- (bird ?animal) (flies ?animal) (lays ?animal eggs))
bird
cl-user(12): (<- (carnivore ?animal) (eats ?animal meat))
carnivore
cl-user(13): (<- (carnivore ?animal)
                 (has ?animal pointed-teeth)
                 (has ?animal claws)
                 (has ?animal forward-eyes))
carnivore
cl-user(14): (<- (ungulate ?animal) (mammal ?animal) (has ?animal hoofs))
ungulate
cl-user(15): (<- (ungulate ?animal) (mammal ?animal) (chews ?animal cud))
ungulate
cl-user(16): (<- (cheetah ?animal)
                 (mammal ?animal)
                 (carnivore ?animal)
                 (has ?animal tawny-color)
                 (has ?animal dark-spots))
cheetah
cl-user(17): (<- (tiger ?animal)
                 (mammal ?animal)
                 (carnivore ?animal)
                 (has ?animal tawny-color)
                 (has ?animal black-stripes))
tiger
cl-user(18): (<- (giraffe ?animal)
                 (ungulate ?animal)
                 (has ?animal long-neck)
                 (has ?animal long-legs)
                 (has ?animal dark-spots))
giraffe
```

bodyに複数の書式が来た場合には，それらの節は and 結合となる．一方，上記(ungulate ?animal)のように，同じ head が別の節で複数宣言された場合には， それらは or 結合となる．

ここまでの入力でいくつかの事実とルールが入力された．さてここで，入力された事実への問い合わせをする．この Prolog では問い合わせには記号 ?- を使う．問い合わせに変数を用いなければ，データベースに単にそういう事実があるかどうかの問い合わせになる．以下では yes の次にピリオドをタイプ入力していることを注意しておく．

```
prolog(19): (?- (hasHusband Marie_Curie Pierre_Curie))
Yes .
End of prove.
prolog(20): (?- (hasMother Irene_Joliot-Curie Marie_Curie))
Yes .
End of prove.
```

さて次はどうだろうか？

```
prolog(21): (?- (loves Marie_Curie Pierre_Curie))
Yes .
End of prove.
prolog(22): (?- (loves Marie_Curie Irene_Joliot-Curie))
End of prove.
```

この場合入力されたルールを使って，(hasHusband Marie_Curie Pierre_Curie) ⇒ (loves Marie_Curie Pierre_Curie) から回答が得られたが，(hasMother Irene_Joliot-Curie Marie_Curie) から得られるものがないため，答えが得られなかった．正確に言うと，そういう問い合わせを肯定することは与えられた事実とルールではできない，ということだ．Prolog では自分の知っていることがすべてで， そのほかの可能性があるということは一切考慮されない．これを閉世界仮説(Closed World Assumption, CWA)という．しかもPrologでは，証明に失敗したらそれは否定されたということとされる．これを証明失敗としての否定(Negation as Failure, NaF)という．上記22行では，何も答えが得られず End of Prove のみが印刷されているが，普通はよくこれを No, と印刷する．ここでは End of prove と印刷しているが Prolog を使うときには，否定（No）は厳密にはそれが証明できなかったと理解されなければならない．それでは，もう二つルールを追加して肯定的回答が得られるようにしてみよう．

```
prolog(23): (<- (hasChild ?x ?y) (hasMother ?y ?x))
hasChild
prolog(24): (<- (loves ?x ?y) (hasChild ?x ?y))
loves
prolog(25): (?- (loves Marie_Curie Irene_Joliot-Curie))
Yes .
End of prove.
```

問い合わせに変数を用いた場合には，その変数にマッチする結果が返される．

```
prolog(26): (?- (loves Marie_Curie ?x))
?x = Pierre_Curie ;
?x = Irene_Joliot-Curie ;
End of prove.
```

ここで，最初の答え Pierre_Curie の次にセミコロンをタイプ入力していることを注意しておく．このように，答えが複数あるときには，対話型モードではひとつづつ答えが出力されるので，次も見たいときにはセミコロンを入力すればよい．途中で表示を打ち切りたいときはピリオドを入力する．

Prologが他の手続き型プログラムによる実行や通常のデータベースへのクエリと異なるのは，次のような芸当もできることだ．

```
prolog(27): (?- (loves ?x Marie_Curie))
?x = Pierre_Curie ;
End of prove.
```

これまで，マリーキュリーが誰かに愛されているなどという事実はどこにもなかったことに注意されたい．マリーキュリーはピエールを夫にしているが，ピエールはマリーを妻にしており，夫は妻を愛しているから，ピエールはマリーを愛していると推論されたのだ．

それでは，これが最後のおまけの問い合わせだ．

```
prolog(28): (?- (hasHusband ?x ?y))
?x = Marie_Curie
?y = Pierre_Curie ;
?x = Irene_Joliot-Curie
?y = Jean_Frederic_Joliot-Curie ;
End of prove.
prolog(29): (?- (hasWife ?x ?y))
?x = Pierre_Curie
?y = Marie_Curie ;
?x = Jean_Frederic_Joliot-Curie
?y = Irene_Joliot-Curie ;
End of prove.
```

32.2 prolog.lisp （インタープリタ版）のコードウォークスルー

prolog.lisp のソースコードの最初にはこんなコードが書いてある．

```
;; Clauses are represented as (head . body) cons cells
(defun clause-head (clause) (first clause))
(defun clause-body (clause) (rest clause))

;; Clauses are stored on the predicate's plist
(defun get-clauses (pred) (get pred 'clauses))
(defun predicate (relation) (first relation))

(defvar *db-predicates* nil
  "a list of all predicates stored in the database.")

(defun add-clause (clause)
  "Add a clause to the data base, indexed by head's predicate."
  ;; The predicate must be a non-variable symbol.
  (let ((pred (predicate (clause-head clause))))
    (assert (and (symbolp pred) (not (variable-p pred))))
    (pushnew pred *db-predicates*)
    (setf (get pred 'clauses)
          (nconc (get-clauses pred) (list clause)))
    pred))

(defmacro <- (&rest clause)
  "add a clause to the data base."
  `(add-clause ',(replace-?-vars clause)))
```

すなわち，マクロ <- の引数には複数のリスト並びが来るが，これが 1 個の節（clause）となる．節（clause）の第一要素が head であり，その他の残りが body であるが，各々の節は head の述語の clauses 属性リストに保存される．

マクロ <- は，事実の宣言にもルールの宣言にも使いられる．そしてこのマクロがすることは add-clause の関数呼び出し形式に直すことだけである．一階述語論理では述語に変数は許されないので，それを assert 文で確かめている． このコードではデータベースに相当するものは単にグローバル変数 *db-predicates* に登録された述語と，その述語の clauses 属性リストで作られる． pushnew はそれがまだ push されていない

ときだけスタックに push する．get はシンボルの属性リストからある属性を取り出す関数だが， pred の clauses 属性リストを取り出して，その最後に今もらった clause を追加している．

データベースへの問い合わせマクロ ?- は単に top-level-prove への関数呼び出し形式にするだけだ．

```
(defmacro ?- (&rest goals) `(top-level-prove ',(replace-?-vars goals)))

(defun top-level-prove (goals)
  (prove-all `(,@goals (show-prolog-vars ,@(variables-in goals)))
             no-bindings)
  (format t "~&End of prove.")
  (values))
```

Prolog では証明すべき書式を特別に**ゴール**と呼ぶ．クエリ ?- の頭部はゴールであるが，そのゴールは頭部に関する本体が証明されれば成立するので，本体の一つ一つは**サブゴール**と呼ばれる．クエリは複数のゴールを受け取って，それをすべて満たすような条件を求めようとする． その詳細はすべて prove-all の中身になる． prove-all ではゴール，サブゴールの並びのリストを頭から順番に prove で証明しようとする．ゴールの述語に関連する節をデータベースから取り出して，それらを順番にゴールと単一化しようとする．単一化に成功すればその bindings リストが prove-all 再帰呼び出しの bindings として使われる．単一化に失敗すれば bindings は fail であるが，その場合には prove-all も失敗となる．かくして，prove-all の各ゴールが単一化に成功する限り，証明は続き，単一化の結果はこの再帰構造の過程で蓄積されていく．一般に，解は 1 セットとは限らない． bindings リスト中の一つが問い合わせを満足する 1 セットになる．しかし，ここで説明しなければならないトリックがある．prove-all に与えられるゴール中には show-prolog-vars もあるのだ．それをするのが上記の top-level-prove である．以下のコードの最後に show-prolog-vars を述語のように clauses 属性リストに自分自身を与えているのを見られたい．ただし，節形式にはなっていない．そして，関数 prove の中では節がリストなら通常の節として prove-all の呼び出しを行うが，アトムだったらそれは関数として関数実行する．こんなトリックで対話形式での解の印刷の継続か停止かの制御を実現している．

```
(defun prove-all (goals bindings)
  "Find a solution to the conjunction of goals."
  (cond ((eq bindings fail) fail)
        ((null goals) bindings)
        (t (prove (first goals) bindings (rest goals)))))

(defun prove (goal bindings other-goals)
  "Return a list of possible solutions to goal."
  (let ((clauses (get-clauses (predicate goal))))
    (if (listp clauses)
        (some
          #'(lambda (clause)
              (let ((new-clause (rename-variables clause)))
                (prove-all
                  (append (clause-body new-clause) other-goals)
                  (unify goal (clause-head new-clause) bindings))))
          clauses)
        ;; The predicate's "clauses" can be an atom:
```

```
        ;; a primitive function to call
        (funcall clauses (rest goal) bindings
                 other-goals))))

(defun show-prolog-vars (vars bindings other-goals)
  "Print each variable with its binding.
  Then ask the user if more solutions are desired."
  (if (null vars)
      (format t "~&Yes ")
      (dolist (var vars)
        (format t "~&~a = ~a " var
                (subst-bindings bindings var))))
  (if (continue-p)
      fail
      (prove-all other-goals bindings)))

(setf (get 'show-prolog-vars 'clauses) 'show-prolog-vars)
```

動きをトレースして観察してみよう.

```
cl-user(17): (trace prove-all prove show-prolog-vars)
(show-prolog-vars prove prove-all)
cl-user(18): (?- (hasHusband ?x ?y))
 0[5]: (prove-all ((hasHusband ?x ?y) (show-prolog-vars ?x ?y)) ((t . t)))
   1[5]: (prove (hasHusband ?x ?y) ((t . t)) ((show-prolog-vars ?x ?y)))
     2[5]: (prove-all ((show-prolog-vars ?x ?y)) ((?y . Pierre_Curie) (?x . Marie_Curie)))
       3[5]: (prove (show-prolog-vars ?x ?y) ((?y . Pierre_Curie) (?x . Marie_Curie)) nil)
         4[5]: (show-prolog-vars (?x ?y) ((?y . Pierre_Curie) (?x . Marie_Curie)) nil)
?x = Marie_Curie
?y = Pierre_Curie ;
         4[5]: returned nil
       3[5]: returned nil
     2[5]: returned nil
     2[5]: (prove-all ((show-prolog-vars ?x ?y)) ((?y . Jean_Frederic_Joliot-Curie) (?x .
Irene_Joliot-Curie)))
       3[5]: (prove (show-prolog-vars ?x ?y) ((?y . Jean_Frederic_Joliot-Curie) (?x .
Irene_Joliot-Curie)) nil)
         4[5]: (show-prolog-vars (?x ?y) ((?y . Jean_Frederic_Joliot-Curie) (?x .
Irene_Joliot-Curie)) nil)
?x = Irene_Joliot-Curie
?y = Jean_Frederic_Joliot-Curie ;
         4[5]: returned nil
       3[5]: returned nil
     2[5]: returned nil
   1[5]: returned nil
 0[5]: returned nil
End of prove.
cl-user(19):
```

ここでは prove-all と prove が互いに互いを呼び合っている．いわば間接的な再帰だが，(hasHusband ?x ?y) の証明(prove)の次は (show-prolog-vars ?x ?y) の証明になっており，ここで（証明ではなく）関数呼び出しが起こり，そのときの関数の引数には，その前の証明時に得られた変数のbindingが与えられて，印刷が実行される.

prove は一個のゴールに対してそれを証明しようとする．ひとつのゴールに対しては，そのゴールの述語についてのすべての節を求めて(`(get-clauses (predicate goal))`)，どれか一つでも証明できればよいから `some` で実行する．節の head に対してユニフィケーションでマッチする条件を求め，それを糧にして今度はその節の body に対して，prove-all で証明に入る．だからこれは後ろ向き推論だ．注目してほしいのはゴールと head の単一化の結果がbody の証明に使われるというところだ．もし途中で単一化に失敗すれば，その節の証明の失敗になるし，もし最後までいって成功すれば，それはbodyの複数の要素に対してandで満足するような解になっているということだ．

ここで rename-variables について説明する．論理変数は混同さえしなければ，書き換えても意味はかわらない．たとえば，`?x` を全くどこにも使われていない `?thx1138` という変数名に書き変えても問題ない．逆にある節の中の変数名が，まったく別の節の変数名とたまたま一致してしまうと困ったことになる．一つのルールの内部では`?x` は同じものを指示しないといけないが，別のルールの`?x` はそれとはまったく関係はないからだ．論理においてはこのような操作はいたるところに出てきて**標準化分離**という言葉が与えられている．以下のコードではgensymが使われていることに注目されたい．

```
(defun rename-variables (x)
  "replace all variables in x with new ones."
  (sublis (mapcar #'(lambda (var) (cons var (gensym (string var))))
                  (variables-in x))
          x))
```

32.3 Prolog プログラミング

これまでのPrologによるデモの実例を見て，Lisp のような，あるいは一般の手続き型言語との違いが分かったであろうか．中にはこれはすごいと思われたかたもいるかもしれない． Prolog による証明過程を計算の手続きの過程と解釈しなおして，これでもって何でもプログラムしてやろうということが昔行われた．それが日本における第５世代コンピュータプロジェクトである．関数型言語とか手続き型言語との比較で，Prolog を関係型言語とか宣言型（叙述）言語と呼ぶ．そしてそのプログラミングを論理プログラミングと呼ぶ．「Prolog には，非決定性，並列性，パタンによる手続きの呼び出しなどの各種の"知的"プログラミング技術が実際的かつ効率的に実現されている」（Prolog プログラミング， Clocksin&Mellish）というわけである．ただし．私見では「実際的かつ効率的」というのは，機械にとってというよりもPrologプログラミングに慣れた人にとってということであろう．確かに，Javaによるプログラムの行数がLispでは1/5 から1/20になる．そしてLispによるプログラムの行数がPrologで書けば極端に減る場合がある．しかし，やってみればすぐ分かることだが，Prolog プログラムを理解するためには，その背後に隠れたプログラムとしての実行過程を追跡しなければならないし，論理プログラミングをするためには，自分のコードの実行過程を想像してプログラミングしなければならない．

以下では，そういう論理プログラミングについて，ちょっとだけ触ってみよう．

Lisp で実装された Prolog では，リストは Lisp のリストで実現されてよい．そして論理プログラミングで member は次のように書かれる．

```
cl-user(1): (<- (member ?item (?item . ?rest)))
member
cl-user(2): (<- (member ?item (?x . ?rest)) (member ?item ?rest))
member
```

member の定義は，もし item がリストの car にあったら自明だし（１行目），自明でなければ cdr 部について調べればよい(２行目)．これを実際に試してみよう．

```
cl-user(3): (?- (member 2 (1 2 3)))
Yes ;
End of prove.
cl-user(4): (?- (member 0 (1 2 3)))
End of prove.
cl-user(5): (?- (member 2 (1 2 3 2 1)))
Yes ;
Yes ;
End of prove.
cl-user(6): (?- (member ?x (1 2 3)))
?x = 1 ;
?x = 2 ;
?x = 3 ;
End of prove.
cl-user(7):
```

素晴らしい．たった2行で member を実現できた．しかし，マジックではない．中で行われたことはこうだ．

```
1  ゴールリスト ((member 2 (1 2 3)))
2  ゴール　　　 (member 2 (1 2 3))
3  マッチ失敗　 (member 2 (1 2 3)) vs. (member ?item (?item . ?rest))
4  マッチ成功　 (member 2 (1 2 3)) vs. (member ?item (?x . ?rest))
5  新ゴール　　 (member 2 (2 3))
6  マッチ成功　 (member 2 (2 3))   vs. (member ?item (?item . ?rest))
7  bindings    ((?item . 2))
8  新ゴール　　 (member 2 (3))
9  マッチ失敗　 (member 2 (3))     vs. (member ?item (?item . ?rest))
10 マッチ成功　 (member 2 (3))     vs. (member ?item (?x . ?rest))
11 新ゴール　　 (member 2 ())
12 マッチ成功　 (member 2 ())      vs. (member ?item (?x . ?rest))
13 新ゴール　　 (member nil ())
14 bindings=nil で11に戻り，8に戻り，bindings=((?item . 2)) で5に戻り
15 答えは((?item . 2))
```

次の例は二つのリストをつなぎ会わせる操作，concatである．

```
cl-user(1): (<- (concat () ?l ?l))
concat
cl-user(2): (<- (concat (?x . ?alist) ?blist (?x . ?ablist)) (concat ?alist ?blist ?ablist))
concat
cl-user(3): (?- (concat (prolog programming) (is hard) ?y))
?y = (prolog programming is hard) ;
End of prove.
```

空リストとリストの連結はリストそのもの（1行目），aリストとbリストの連結がabリストならば，aリストに何かをコンスしたものはabリストに同じものをコンスしたものになるではあろう（2行目）．しかし，これで確かに答えが得られるとは，従来の手続き型プログラム言語に慣れた目にはやはりマジックとしか思えない．

問題32.1 concat の働きを member と同様に追跡せよ．

問題32.2 reverse のあらましは次のようなコードである．下線部に適切な表記を挿入せよ．

```
(<- (rev () ()))
(<- (rev (?first . ?rest) ?reversed) (rev ?rest ?reversedRest) (concat ?reversedRest _______
?reversed)
```

最初に掲げた member のコードは lisper にとっても分かりやすいが，concat を使った prolog らしいより宣言的なコードがある．次の例を見られたい．

```
cl-user(4): (<- (member* ?x ?lst) (concat ?before (?x . ?after) ?lst))
member*
cl-user(5): (?- (member* 2 (1 2 3 2 1)))
Yes ;
Yes ;
End of prove.
cl-user(6): (?- (member* ?x (1 2 3)))
?x = 1 ;
?x = 2 ;
```

```
?x = 3 ;
End of prove.
```

なるほど，確かに宣言的な定義をするだけで，プログラムとして実行できる．それでは，こんなプログラムではどうだろうか．

```
cl-user(7): (<- (person ?x) (person ?y) (mother ?x ?y))
person
```

人間の母は人間という知識のつもり．これを（無限ループにおちいるのを覚悟で）実行してみる．

```
cl-user(8): (<- (mother Marie_Curie Irene_Joliot-Curie))
mother
cl-user(9): (<- (person Irene_Joliot-Curie))
person
cl-user(10): (?- (person ?x))
Error: Recursive error while printing or signalling an error.  This is sometimes caused by an
error in a print-object method.

Restart actions (select using :continue):
```

ご覧のとおりだ．もう一つの例をあげよう．

```
cl-user(12): (<- (hasHusband Marie_Curie Pierre_Curie))
hasHusband
cl-user(13): (<- (hasWife ?x ?y) (hasHusband ?y ?x))
hasWife
cl-user(14): (<- (hasHusband ?x ?y) (hasWife ?y ?x))
hasHusband
cl-user(15): (?- (hasWife Pierre_Curie ?x))
?x = Marie_Curie ;
?x = Marie_Curie ;
?x = Marie_Curie ;
?x = Marie_Curie ;
?x = Marie_Curie ;
?x = Marie_Curie ;
?x = Marie_Curie .
End of prove.
```

x が y の夫なら，y は x の妻．逆に x が y の妻なら，y は x の夫，という知識のつもりだが，御覧のように同じ答えを無限に繰り返すだけなので，最後はピリオドを入力した．何故無限ループにハマってしまうのか，お分かりであろう．ゴールを確立させようとして，また似たようなゴールを設定してしまうのだ．bindings は無限に伸びていくが，いつまでたっても停止にはたどりつかない．こんな簡単なプログラムで無限に陥るのなら，複雑なプログラムだったらそれこそ簡単に無限地獄に陥ってしまうのではないか．

きちんと書かれた Prolog コードだったら，コンパクトでとてもメンテしやすいかも知れない．でも人の書いたコードを内容を理解して手を入れるというのはとても大変だ．せいぜい数百行ぐらいのコード量であれば，いいのかも知れない．Prolog をきちんともう少し勉強しようとされる人は，今なら SWI-Prolog をダウンロードして勉強されたい．これはとんでもなく早い．一体どうやって，このスピードを実現しているのかとびっくりしてしまう．

32.4 Prolog の自動バックトラッキング

第24章では色々な探索戦略を実装した．この Prolog では探索の実装が prove と prove-all の関数呼び出しスタックになっている． prove-all の bindings に fail をもらった時は単一化失敗だから fail で次々と関数呼び出しからの戻りが行われる．これを探索の世界ではバックトラックと言う．失敗の理由を問わず，関数呼び出しスタックを巻き戻すような，探索経路をただ戻すようなバックトラックはクロノロジカル・バックトラックと言う．それに対して，失敗の原因を調べて，一気にその時点まで戻すようなものを依存に基づくバックトラック（dependency-directed backtracking）と言う．もちろん，クロノロジカル・バックトラックは関数の呼び出しを戻るだけだから簡単だが，依存に基づくバックトラックの実装は難しい．

では，最後に次のようなデモをしてみよう．

```
cl-user(1): (<- (likes Kim Robin))
likes
cl-user(2): (<- (likes Sandy Lee))
likes
cl-user(3): (<- (likes Sandy Kim))
likes
cl-user(4): (<- (likes Robin cats))
likes
cl-user(5): (<- (likes Sandy ?x) (likes ?x cats))
likes
cl-user(6): (<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
likes
cl-user(7): (<- (likes ?x ?x))
likes
cl-user(8): (?- (likes Sandy ?who))
?who = Lee ;
?who = Kim ;
?who = Robin ;
?who = Sandy ;
?who = cats ;
?who = Sandy ;
End of prove.
cl-user(9): (?- (likes ?who Sandy))
?who = Sandy ;
?who = Kim ;
?who = Sandy ;
End of prove.
cl-user(10): (?- (likes Robin Lee))
End of prove.
cl-user(11): (?- (likes ?x ?y) (likes ?y ?x))
?y = Kim
?x = Sandy ;
?y = Sandy
?x = Sandy ;
?y = Sandy
?x = Sandy ;
?y = Sandy
?x = Kim ;
?y = Sandy
?x = Sandy ;
?y = ?x1463
```

```
?x = ?x1463 ;
End of prove.
cl-user(12):
```

32.5 Prologc(コンパイラ版)のコードウォークスルー

Prologのコンパイラ版では複数の節の全体がLisp関数群に変換され，それらがコンパイルされる．述語の名前がLisp関数名となるのなら，その述語の引数が Lisp 関数の引数となるべきだが，その引数の数は一般的には固定ではない．同一の述語でも arity が異なることがあり得る．実際に作られるLisp関数名は引数の数(arity)と組み合わせた "‹predicate›/‹arity›" というようなものである．たとえば次のような例では，三つの節があっても述語は likes しかなく，arity は2なので作られる関数名は likes/2 一つでよく，その中身のコードは三つの節の内容をまとめたものとなる．その変換を実行するのが compile-predicate であり，この例の場合には以下に示すコードが出来あがる．

```
(<- (likes Robin cats))
(<- (likes Sandy ?x) (likes ?x cats))
(<- (likes Kim ?x) (likes ?x Lee) (likes ?x Kim))
```

```
(defun compile-predicate (symbol arity clauses)
  "Compile all the clauses for a given symbol/arity
  into a single LISP function."
  (let ((*predicate* (make-predicate symbol arity))    ;***
        (parameters (make-parameters arity)))
    (compile
     (eval
      `(defun ,*predicate* (,@parameters cont)
         .,(maybe-add-undo-bindings
            (mapcar #'(lambda (clause)
                        (compile-clause parameters clause 'cont))
              clauses)))))))
```

```
(defun likes/2 (?arg1 ?arg2 cont)
  (let ((old-trail (fill-pointer *trail*)))
    (if (unify! ?arg1 'Robin) (if (unify! ?arg2 'cats) (funcall cont)))
    (undo-bindings! old-trail)
    (if (unify! ?arg1 'Sandy) (likes/2 ?arg2 'cats cont))
    (undo-bindings! old-trail)
    (if (unify! ?arg1 'Kim) (likes/2 ?arg2 'Lee #'(lambda nil (likes/2 ?arg2 'Kim cont))))))
```

見てすぐ分かるように，1番目の節が likes/2 の Lisp コードの1番目に対応し，2番目の節がコードの真ん中に対応し，三番目の節がコードの最後に対応する． unify! は29.3節における破壊的単一化であり，cont はクエリの最初の一つの項が成功した時，次に実行されるべき継続である．PAIPではこれを**成功継続**と読んでいる．上記三つの節はそれぞれ，1) head のみの場合，2) body の要素が一つの場合，3) body の要素が二つの場合だが，それぞれにおいてどう成功継続が作られ，使われているか観察されたい．

最初の節に相当する単一化に失敗した時にはバックトラックが起きなければならないが，それに相当するのが undobindings である．単一化の効果を元に戻してから，次の節に相当するコードに入る．成功継続まで含めて，クエリ節のすべての項が成功した時最後に待ち構えているのは show-prolog-vars に相当する show-prolog-vars/2 関数であり，bindings 結果を印刷したのちは，失敗した時と同様に継続から戻って，次の節に相当するコードに入る．

compile-predicate は prolog-compile によって明示的に呼び出される．引数の symbol が述語名であり，頭部の述語と arity と共通する節に相当する節が集められて compile-predicate に与えられる．最後に arity の異なる節を集めて prolog-compile が再帰される．

```
(defun prolog-compile (symbol &optional (clauses (get-clauses symbol)))
  "Compile a symbol; make a separate function for each arity."
  (unless (null clauses)
    (let ((arity (relation-arity (clause-head (first clauses)))))
      ;; Compile the clauses with this arity
      (compile-predicate
        symbol arity (clauses-with-arity clauses #'= arity))
      ;; Compile all the clauses with any other arity
      (prolog-compile
        symbol (clauses-with-arity clauses #'/= arity)))))
```

prolog-compile を呼び出すのは，prolog-compile-symbols だが，prolog-compile-symbols は run-prolog によって呼び出される．run-prolog を呼び出すのは，コンパイラ版における top-level-prove だ．

```
(defun top-level-prove (goals)
  "Prove the list of goals by compiling and calling it."
  ;; First redefine top-level-query
  (clear-predicate 'top-level-query)
  (let ((vars (delete '? (variables-in goals))))
    (add-clause `((top-level-query)
                  ,@goals
                  (show-prolog-vars ,(mapcar #'symbol-name vars)
                                    ,vars))))
  ;; Now run it
  (run-prolog 'top-level-query/0 #'ignores)
  (format t "~&End of prove.")
  (values))

(defun run-prolog (procedure cont)
  "Run a 0-ary prolog procedure with a given continuation."
  ;; First compile anything else that needs it
  (prolog-compile-symbols)
  ;; Reset the trail and the new variable counter
  (setf (fill-pointer *trail*) 0)
  (setf *var-counter* 0)
  ;; Finally, call the query
  (catch 'top-level-prove
    (funcall procedure cont)))

(defun prolog-compile-symbols (&optional (symbols *uncompiled*))
  "Compile a list of Prolog symbols.
  By default, the list is all symbols that need it."
  (mapc #'prolog-compile symbols)
  (setf *uncompiled* (set-difference *uncompiled* symbols)))
```

コンパイラ版のコードはまだまだあるが，きりがないので細部は各自で探求されたい．上記で説明した述語＋アリティに合致するLisp関数をあらかじめ定義しておくことで，組み込み述語が定義される．最後に次節でその組み込み述語のいくつかについて説明する．

32.6 Prolog の組み込み述語

組み込み述語で基礎的なものは，true，false，fail，not，=，call などであるが，PAIP の Prolog では，これらのうち，not と = と call しか提供されない，しかも = では SWI Prolog やその他の Prolog の挙動と異なるところがあり注意が必要である．以下にそんな例を紹介する．

PAIP Prolog の = は二つの引数が単一化可能かどうかを調べるものであり，== は二つの引数を単一化せずに等しいかどうかを調べるものである．両方ともアトムだけの場合もリストの場合も正しく動くが，変数同士の比較では注意が必要である．

```
prolog(16): (?- (== ?x ?x))
?x = ?1 ;
End of prove.
prolog(17): (?- (= ?x ?x))
Warning: Variable cont is never used.
End of prove.
prolog(18): (?- (== ?x ?y))
End of prove.
prolog(19): (?- (= ?x ?y))
?x = ?1
?y = ?1 ;
End of prove.
```

これを同様な SWI Prolog の場合と比べてみよう．

```
?- X = X.
true.

?- X = Y.
X = Y.
```

特に PAIP では (= ?x ?x) で 答えが得られないのが問題である．二つの引数が両方とも同じ論理変数ではあるが単一化の答えは得られなければならない．ちなみに，(unify '?x '?x)ー＞((t . t)) となる．また，(unify! '?x '?x) では答えとして t が得られる．=/2 は以下のような関数であるが，=/2 と unify! にトレースをかけてもトレースは得られず，下記のようになる．これを SWI Prolog のようにするには相当基本的なところで改善が必要になると思われる．

```
(defun =/2 (?arg1 ?arg2 cont)
  (if (unify! ?arg1 ?arg2)
      (funcall cont)))

prolog(20): (trace =/2 unify!)
(unify! =/2)
prolog(21): (?- (= ?x ?x))
Warning: Variable cont is never used.
End of prove.
prolog(22):
```

== の場合も芳しくない．(== ?x ?x) でtrueが欲しいところだが，標準化分離で得られる新たな変数が返されている．==/2 は以下のような関数であるが，トレースをかけて次のような実行結果が得られる．

```
(defun ==/2 (?arg1 ?arg2 cont)
  "Are the two arguments EQUAL with no unification,
  but with dereferencing?  If so, succeed."
  (if (deref-equal ?arg1 ?arg2)
      (funcall cont)))

(defun deref-equal (x y)
  "Are the two arguments EQUAL with no unification,
  but with dereferencing?"
  (or (eql (deref x) (deref y))
      (and (consp x)
           (consp y)
           (deref-equal (first x) (first y))
           (deref-equal (rest x) (rest y)))))
```

```
prolog(22): (trace ==/2 deref-equal)
(deref-equal ==/2)
prolog(23): (?- (== ?x ?x))
 0[5]: (==/2 #1=?1 #1# #<Closure (:internal top-level-query/0 0) @ #x204f7c072>)
   1[5]: (deref-equal #1=?1 #1#)
   1[5]: returned t
?x = ?1 ;
 0[5]: returned nil
End of prove.
```

終章 おわりに

この書籍は，著者のブログ，「セマンティックウェブ・ダイアリー」に掲載された記事のうち，Common Lispによる人工知能プログラミングの部分に加筆修正してまとめたものである．

英語でも日本語でも，Lispの教科書は世の中に数多くあるが，不思議なことに人工知能プログラミングについては，そうではない．この書籍をまとめるにあたって，改めて最近の人工知能教科書や人工知能プログラミングの教科書を調べてみたが，最新の情報が記載されていると思われる The Cambridge Handbook of Artificial Intelligence (2014年出版) においてすら，Further reading にあるのは Artificial Intelligence: A Modern Approach（この本にはLispではない疑似プログラムコードが記載されている）に並んで，実にこの原稿執筆時の23年前，1992年に発行の Paradigms of Artificial Intelligence Programming （頭文字をとって俗にPAIP，パイプと呼ばれる）なのである．すなわち，現在最も信頼できて，かつ包括的な（記号処理の）AIプログラミングのテキストといえるようなものは，23年前のPAIPしかないという状況である．

それではPAIP出版当時の1990年頃からAI研究は進歩していないのかといえばもちろんそんなことはなく，現在最先端の研究では人間の身体性，感性，倫理も含めて，全体的にかつ一層深く人間の本質的な存在に迫ろうとしているのは，前出のThe Cambridge Handbook of AI の目次や Artificial Intelligence: A Modern Approach，第3版のAI研究の歴史を見れば明らかである．しかし一方で，それら最先端の研究テーマについての便利な計算機コードは見当たらず，いわば過去の人工知能プログラミングはPAIPに纏められてはいるが，それを継ぐものはいまだ現れていない．

そういうAI 研究の最先端に対してこの書籍がどう位置づけられるかと言えば，これもまた基本的にはPAIPを超えるものではないということをお断りしておかなければならない．ただし，PAIP出版当時とは異なり，HTTPプロトコルによるウェブの時代（Yahoo創業が1994年）とGoogleに代表される検索技術による情報爆発の時代（Google創業は1998年）を迎えて，さらに現在のビッグデータ，オープンデータの勢いを反映して，その内容は大量データ処理，並列処理，日本語処理，検索技術などに配慮したものである．

一方，現今の第3次AIブームではディープラーニング全盛であり，そのため，今ではAIと言えばディープラーニングのことのみのようになってしまい，古い記号処理のAIについて市販のテキストを入手することさえ困難になってしまった．

ここで筆者の持論を述べれば，センシングに始まる知覚・認知の部分はディープラーニングが得意なところであるが，統合から推論・判断の部分は記号処理が得意の部分であって，どうしてこれを「A]と読んだかとか，どうしてこれを「猫」と認知したかという部分ではあまり根拠を問われることはないが，複数の異なるモーダルの入力から関連を発見して状況判断するとか，適切な方策を立案するというような課題では根拠を説明できる能力が必要になり，そこでは記号処理が有効である，ということだ．近い将来ニューラルネットワーク技術と記号処理技術の融合が当たり前に求められるようになると思われる．

付録８ 弁別木による知識ストア dtree

本文における「第24章 推論と探索」では，動物のルールベースについて説明し、前向き推論と後ろ向き推論を説明した．そこでは異なる動物のルール前件部に．同じような表現が多く含まれ、前件部の少しの違いが似た特徴を持つ動物の異なりを決定することに気が付かれたであろう．

付8.1 弁別木による知識ストア dtree

本書 32.2 節のインタープリタ版 Prolog では Prolog の節 clause は，その頭部の述語をキーとしてそのシンボル属性 clauses の値として節全体が追加保存された．以下にそのコードを再掲する．

```
(defun add-clause (clause)
  "Add a clause to the data base, indexed by head's predicate."
  ;; The predicate must be a non-variable symbol.
  (let ((pred (predicate (clause-head clause))))
    (assert (and (symbolp pred) (not (variable-p pred))))
    (pushnew pred *db-predicates*)
    (setf (get pred 'clauses)
          (nconc (get-clauses pred) (list clause)))
    pred))
```

しかし，このような保存方法では，少しだけ違うけど似たような節が大量にあるような場合には，効率が悪い．ここで，通常のルールのように頭部と本体を持つ節は従来通りとして，頭部しか持たないような節，すなわち事実を大量に持つ問題に適した弁別木によるデータベース dtree を導入する．データベースと言っても，Lisp 環境におけるin memoryの実現であり，ファイルベースのものとは異なることに注意されたい．

krep では，事実は次の add-fact 関数によりデータベースに保存される．

```
(defparameter *primitives* '(kr:and sub ind rel val))

(defun add-fact (fact)
  "Add the fact to the data base."
  (cond ((eq (predicate fact) 'kr:and)
         (mapc #'add-fact (args fact)))
        ((or (not (every #'atom (args fact)))
             (some #'variable-p (args fact))
             (not (member (predicate fact) *primitives*)))
         (error "Ill-formed fact: ~A" fact))
        ((not (fact-present-p fact))
         (index fact)
         (run-attached-fn fact)))
  t)
```

primitives と run-attached-fn の説明は今は置いておいて話を進めると，一般の事実であればここで (index fact) が実行される．この index が弁別木 dtree に事実を保存するものである．

```
(defun index (key)
  "Store key in a dtree node.  Key must be (predicate . args);
  it is stored in the predicate's dtree."
```

```
  (dtree-index key (rename-variables key)    ; store unique vars
               (get-dtree (predicate key))))
```

関数 index の引数 key には Prolog の頭部に相当する事実が渡される．頭部の predicate をルートにして弁別木が作られるが，既に過去に作られた弁別木があれば，まずはそれを取り出して dtree-index で新しい枝を追加していく．これが初めてならば，新しい弁別木の根元を作ってそれを predicate のシンボル属性 dtree に保存する．

弁別木 (Discrimination Tree) というのは，Lisp のS式を木構造に見立てたときに，複数のS式で共通する部分は共有するが異なる枝についてはそこから異なる枝を派生させて保持するようなものである．典型的なコードは charniak, riesbeck, meehan による artificial intelligence progamming の11章に詳しい説明があるが，ここで用いられるものは実装が異なっていて，リストの代わりにベクターを用いて木構造とし，リストの長さを情報に持つリスト nlist を用いたものである．

```
;; An nlist is implemented as a (count . elements) pair:
(defun make-empty-nlist ()
  "Create a new, empty nlist."
  (cons 0 nil))

(defun nlist-n (x) "The number of elements in an nlist." (car x))
(defun nlist-list (x) "The elements in an nlist." (cdr x))

(defun nlist-push (item nlist)
  "Add a new element to an nlist."
  (incf (car nlist))
  (push item (cdr nlist))
  nlist)
```

dtree のノードは first，rest に加えて，アトム用の atoms と変数用の var を持つベクターである．atoms にはアトムと key として与えられたS式の nlist の連想リスト(bindings)が保存される．S式の要素が変数の場合には atoms ではなく var に保存される．
dtree-index の引数 key と value は最初の index に与えられる key が変数を含まない場合には同一であることを注意しておく．変数を含む場合には，value の方は変数のみが標準化分離されたものになる．key と value がリストの場合には car と cdr について再帰されるが，そのとき dtree も first と rest をもって再帰する．すなわち，key で与えられたS式を dtree に相似形に移しとっていくことになる．

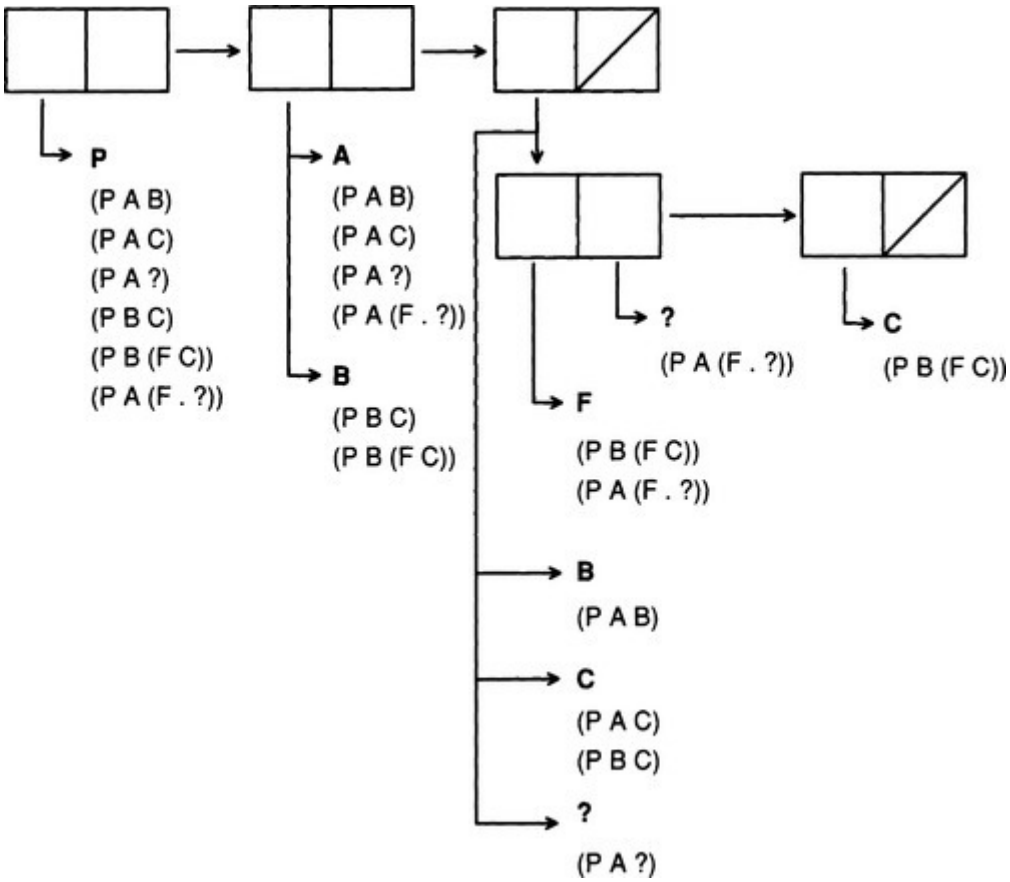
```
(defstruct (dtree (:type vector))
  (first nil) (rest nil) (atoms nil) (var (make-empty-nlist)))

(defun dtree-index (key value dtree)
  "Index value under all atoms of key in dtree."
  (cond
    ((consp key)               ; index on both first and rest
     (dtree-index (first key) value
                  (or (dtree-first dtree)
                      (setf (dtree-first dtree) (make-dtree))))
     (dtree-index (rest key) value
                  (or (dtree-rest dtree)
                      (setf (dtree-rest dtree) (make-dtree)))))
```

```
    ((null key))               ; don't index on nil
    ((variable-p key)          ; index a variable
     (nlist-push value (dtree-var dtree)))
    (t ;; Make sure there is an nlist for this atom, and add to it
     (nlist-push value (lookup-atom key dtree)))))

(defun get-dtree (predicate)
  "Fetch (or make) the dtree for this predicate."
  (cond ((get predicate 'dtree))
        (t (push predicate *predicates*)
           (setf (get predicate 'dtree) (make-dtree)))))
```

こうして，よく似ているが少しずつ異なるS式を保存した結果は例えば**付図4.1**のようになる．ここで各ノードは dtree のノードであり，P とか A,B などが atoms に保存されるアトムであり，(P A B) や (P A ?) が value である．



付図4.1　dtree の例

なおこの dtree のコードでは，全く同じS式を複数回与えた場合，同一の value が与えた回数だけ保存されてしまうことを注意しておく．それを避けるには，以下の index-new-fact を用いればよい．

```
(defun fact-present-p (fact)
  "Is this fact present in the data base?"
  (retrieve fact))

(defun index-new-fact (fact)
  "Index the fact in the data base unless it is already there."
  (unless (fact-present-p fact)
    (index fact)))
```

それでは，ここまでのところでキュリー夫妻とその家族の事実を入力してみよう．

```
cg-user(1): (in-package :krep)
#<The krep package>
kr(2): (index-new-fact '(hasHusband Marie_Curie Pierre_Curie))
t
kr(3): (index-new-fact '(hasHusband Irene_Joliot-Curie Jean_Frederic_Joliot-Curie))
t
kr(4): (index-new-fact '(hasMother Irene_Joliot-Curie Marie_Curie))
t
kr(5): (index-new-fact '(hasFather Irene_Joliot-Curie Pierre_Curie))
t
```

データベースを直接検索するには retieve を用いる．

```
(defun fetch (query)
  "Return a list of buckets potentially matching the query,
  which must be a relation of form (predicate . args)."
  (dtree-fetch query (get-dtree (predicate query))
               nil 0 nil most-positive-fixnum))

(defun mapc-retrieve (fn query &optional (bindings no-bindings))
  "For every fact that matches the query,
  apply the function to the binding list."
  (dolist (bucket (fetch query))
    (dolist (answer bucket)
      (let ((new-bindings (unify query answer bindings)))
        (unless (eq new-bindings fail)
          (funcall fn new-bindings))))))

(defun retrieve (query &optional (bindings no-bindings))
  "Find all facts that match query.  Return a list of bindings."
  (let ((answers nil))
    (mapc-retrieve #'(lambda (bindings) (push bindings answers))
                   query bindings)
    answers))


kr(7): (retrieve '(hasHusband ?x ?y))
(((?y . Pierre_Curie) (?x . Marie_Curie)) ((?y . Jean_Frederic_Joliot-Curie) (?x .
Irene_Joliot-Curie)))
kr(8): (retrieve '(hasMother ?x Marie_Curie))
(((?x . Irene_Joliot-Curie)))
kr(9): (retrieve '(hasFather Irene_Joliot-Curie ?y))
(((?y . Pierre_Curie)))
```

見てわかるように unification を使ってパターンマッチにより強力な検索が可能であるが，いわゆる Prolog の推論機能はない．

付8.2 カテゴリ，個体，関係，値域・領域制約

krep ではここでカテゴリ(category) と個体(indibidual) ，関係(relation) と値域・領域制約を導入する．ここで言うカテゴリとサブカテゴリはRDFやOWLでいうところのクラスや概念であり，曖昧な意味を有する我々が通常使う用語，いわゆる「カテゴリ」ではない．以下では krep でいうサブカテゴリ，スーパーカテ

ゴリという言葉を避けて，サブクラス，スーパークラスという言葉を使うことにする．組み込みの述語 sub は二つのクラスがサブクスとスーパークラスの関係にあることをいう．これはRDFで言えば rdfs:subClassOf に相当するものである．組み込み述語 ind はあるオブジェクトがあるクラスの個体(individual) であることをいう．これはRDFで言えば rdf:type のことである．また，組み込み述語 rel は個体についてのある関係について，1番目の個体は領域クラスに所属し，2番目の個体は値域クラスに所属していることを言う．これはRDFで言えばrdf:propertyOf とその領域制約(rdfs:domain) と値域制約(rdfs:rang) のことである．つまり，個体間の関係は組み込み述語 val で記述されるが，そのとき領域個体は領域制約クラスに所属しなければならないし，値域個体は値域制約クラスに所属しなければならない．

```
(sub サブクラス スーパークラス)
(ind 個体 クラス)
(rel 関係 領域制約クラス 値域制約クラス)
(val 関係 領域個体 値域個体)
```

これまで本文で何度も出てきた動物のルールについて，これらの組み込み述語を使って事実を記述すれば以下のようになる．

```
(cl:defpackage :animal
  (:shadow #:and)
  (:use :common-lisp :krep)
  (:export #:index)
  )

(in-package :animal)

(add-fact '(sub mammal animal))
(add-fact '(sub bird animal))
(add-fact '(sub carnivore mammal))
(add-fact '(sub ungulate mammal))
(add-fact '(sub cheetah carnivore))
(add-fact '(sub tiger carnivore))
(add-fact '(sub giraffe ungulate))
(add-fact '(sub zebra ungulate))
(add-fact '(sub ostrich bird))
(add-fact '(sub penguin bird))
(add-fact '(sub albatross bird))

(add-fact '(ind Skipper penguin))
(add-fact '(ind Kowalski penguin))
(add-fact '(ind Rico penguin))
(add-fact '(ind Clawhauser cheetah))
(add-fact '(ind Jeremy giraffe))
(add-fact '(ind Gia giraffe))
(add-fact '(ind Fragmeyer tiger))
(add-fact '(ind Jackson tiger))
(add-fact '(ind Yolanda zebra))
(add-fact '(ind Rolfina zebra))

(add-fact '(rel has mammal hair))
(add-fact '(rel gives mammal milk))
(add-fact '(rel has bird feathers))
(add-fact '(kr:and (rel can bird fly)
                   (rel lays bird eggs)))
```

```
(add-fact '(rel eats carnivore meats))
(add-fact '(kr:and (rel has carnivore pointed_teeth)
                   (rel has carnivore claws)
                   (rel has carnivore forward_eyes)))
(add-fact '(rel has ungulate hoofs))
(add-fact '(rel chews ungulate cud))
(add-fact '(kr:and (rel has cheetah tawny_color)
                   (rel has cheetah dark_spots)))
(add-fact '(kr:and (rel has tiger tawny_color)
                   (rel has tiger black_stripes)))
(add-fact '(kr:and (rel has giraffe long_neck)
                   (rel has giraffe long_legs)
                   (rel has giraffe dark_spots)))
(add-fact '(rel has zebra black_stripes))
(add-fact '(kr:and (rel cannot ostrich fly)
                   (rel has ostrich long_neck)
                   (rel has ostrich long_legs)
                   (rel isColored ostrich black_and_white)))
(add-fact '(kr:and (rel cannot penguin fly)
                   (rel can penguin swims)
                   (rel isColored penguin black_and_white)))
(add-fact '(rel can albatross flies_well))
```

ここで，述語 ind の1番目のパラメータは個体を指示し，2番目のパラメータはその個体が所属するクラスを指示する．また，述語 rel の次のパラメータは2番目と3番目のパラメータの関係を記述する述語である．上記では，has, gives, can, lays などであるが，ここにくる2番目と3番目のパラメータはクラスであることを重ねて注意しておく．すなわち，(has mammal hair) や (gives mammal milk) において，hair や milk もクラスである．ここまでのところで，個体について知識を記述すれば，以下のようになる．

```
(add-fact '(kr:and (val has Clawhauser tawny_color-1)
                   (val has Clawhauser dark_spots-1)))

(add-fact '(kr:and (val cannot Kowalski fly-1)
                   (val can Kowalski swims-1)
                   (val isColored Kowalski black_and_white-1)))
```

ここで，Kowalski はペンギンである個体であるが，fly-1 もクラス fly のインスタンスである．同様に，Clawhauser はチータである個体であるが，tawny_color-1 もクラス tawny_color のインスタンスである．ペンギンやチータは個体として違和感がないが，色や動作についてはどうしても違和感がぬぐえない．あえて言えば　Kowalski の動作のこと，Clawhauser の体色のことと思ってもらうしかない．

付録９ 知識表現言語 krep

ここでは本文における「第32章 Lisp で作る Prolog 」の応用編にあたる知識表現言語 krep を紹介する．わざわざこれを紹介するわけは，PAIP の 14.10 節に相当するこの部分に Semantic Web やオントロジーに重なる部分が多くあり，Semantic Web 用 Lisp (SWLisp) への橋渡しとしての価値があると考えるからである．ただし，PAIP ではそのような位置づけは皆無であるし，また唯一名仮説や閉世界仮説など，Prologの前提を一切超えるものではないということを一言お断りしておく．一方，セマンティックウェブでは非唯一名仮説であるし，開世界仮説となっている．つまり，名前が違っていても同じ個体を指示するかもしれないし，推論できないからといって世界のどこかにそれを支持する世界があるかもしれない．その詳細はまた，別の機会に．

付9.1 弁別木による知識ストア dtree

本書 32.2 節のインタープリタ版 Prolog では Prolog の節 clause は，その頭部の述語をキーとしてそのシンボル属性 clauses の値として節全体が追加保存された．以下にそのコードを再掲する．

```
(defun add-clause (clause)
  "Add a clause to the data base, indexed by head's predicate."
  ;; The predicate must be a non-variable symbol.
  (let ((pred (predicate (clause-head clause))))
    (assert (and (symbolp pred) (not (variable-p pred))))
    (pushnew pred *db-predicates*)
    (setf (get pred 'clauses)
          (nconc (get-clauses pred) (list clause)))
    pred))
```

しかし，このような保存方法では，少しだけ違うけど似たような節が大量にあるような場合には，効率が悪い．ここで，通常のルールのように頭部と本体を持つ節は従来通りとして，頭部しか持たないような節，すなわち事実を大量に持つ問題に適した弁別木によるデータベース dtree を導入する．データベースと言っても，Lisp 環境におけるin memoryの実現であり，ファイルベースのものとは異なることに注意されたい．

krep では，事実は次の add-fact 関数によりデータベースに保存される．

```
(defparameter *primitives* '(kr:and sub ind rel val))

(defun add-fact (fact)
  "Add the fact to the data base."
  (cond ((eq (predicate fact) 'kr:and)
         (mapc #'add-fact (args fact)))
        ((or (not (every #'atom (args fact)))
             (some #'variable-p (args fact))
             (not (member (predicate fact) *primitives*)))
         (error "Ill-formed fact: ~A" fact))
        ((not (fact-present-p fact))
         (index fact)
         (run-attached-fn fact)))
  t)
```

primitives と run-attached-fn の説明は今は置いておいて話を進めると，一般の事実であればここで (index fact) が実行される．この index が弁別木 dtree に事実を保存するものである．

```
(defun index (key)
  "Store key in a dtree node.  Key must be (predicate . args);
  it is stored in the predicate's dtree."
  (dtree-index key (rename-variables key)    ; store unique vars
               (get-dtree (predicate key))))
```

関数 index の引数 key には Prolog の頭部に相当する事実が渡される．頭部の predicate をルートにして弁別木が作られるが，既に過去に作られた弁別木があれば，まずはそれを取り出して dtree-index で新しい枝を追加していく．これが初めてならば，新しい弁別木の根元を作ってそれを predicate のシンボル属性 dtree に保存する．

弁別木 (Discrimination Tree) というのは，Lisp のS式を木構造に見立てたときに，複数のS式で共通する部分は共有するが異なる枝についてはそこから異なる枝を派生させて保持するようなものである．典型的なコードは charniak, riesbeck, meehan による artificial intelligence progamming の11章に詳しい説明があるが，ここで用いられるものは実装が異なっていて，リストの代わりにベクターを用いて木構造とし，リストの長さを情報に持つリスト nlist を用いたものである．

```
;; An nlist is implemented as a (count . elements) pair:
(defun make-empty-nlist ()
  "Create a new, empty nlist."
  (cons 0 nil))

(defun nlist-n (x) "The number of elements in an nlist." (car x))
(defun nlist-list (x) "The elements in an nlist." (cdr x))

(defun nlist-push (item nlist)
  "Add a new element to an nlist."
  (incf (car nlist))
  (push item (cdr nlist))
  nlist)
```

dtree のノードは first，rest に加えて，アトム用の atoms と変数用の var を持つベクターである．atoms にはアトムと key として与えられたS式の nlist の連想リスト(bindings)が保存される．S式の要素が変数の場合には atoms ではなく var に保存される．
dtree-index の引数 key と value は最初の index に与えられる key が変数を含まない場合には同一であることを注意しておく．変数を含む場合には，value の方は変数のみが標準化分離されたものになる．key と value がリストの場合には car と cdr について再帰されるが，そのとき dtree も first と rest をもって再帰する．すなわち，key で与えられたS式を dtree に相似形に移しとっていくことになる．

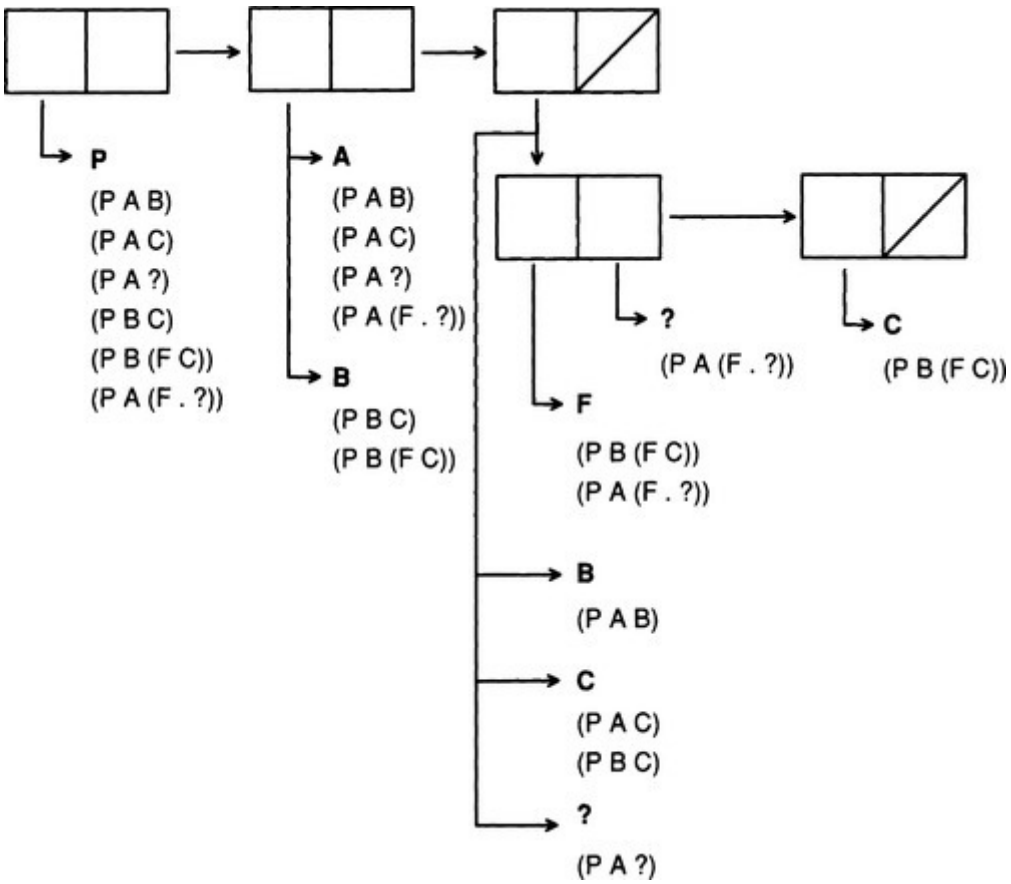
```
(defstruct (dtree (:type vector))
  (first nil) (rest nil) (atoms nil) (var (make-empty-nlist)))

(defun dtree-index (key value dtree)
  "Index value under all atoms of key in dtree."
  (cond
```

```
    ((consp key)              ; index on both first and rest
     (dtree-index (first key) value
                  (or (dtree-first dtree)
                      (setf (dtree-first dtree) (make-dtree))))
     (dtree-index (rest key) value
                  (or (dtree-rest dtree)
                      (setf (dtree-rest dtree) (make-dtree)))))
    ((null key))              ; don't index on nil
    ((variable-p key)         ; index a variable
     (nlist-push value (dtree-var dtree)))
    (t ;; Make sure there is an nlist for this atom, and add to it
     (nlist-push value (lookup-atom key dtree)))))

(defun get-dtree (predicate)
  "Fetch (or make) the dtree for this predicate."
  (cond ((get predicate 'dtree))
        (t (push predicate *predicates*)
           (setf (get predicate 'dtree) (make-dtree)))))
```

こうして，よく似ているが少しずつ異なるS式を保存した結果は例えば**付図4.1**のようになる．ここで各ノードは dtree のノードであり，P とか A,B などが atoms に保存されるアトムであり，(P A B) や (P A ?) が value である．



付図4.1　dtree の例

なおこの dtree のコードでは，全く同じS式を複数回与えた場合，同一の value が与えた回数だけ保存されてしまうことを注意しておく．それを避けるには，以下の index-new-fact を用いればよい．

```
(defun fact-present-p (fact)
  "Is this fact present in the data base?"
```

```
  (retrieve fact))

(defun index-new-fact (fact)
  "Index the fact in the data base unless it is already there."
  (unless (fact-present-p fact)
    (index fact)))
```

それでは，ここまでのところでキュリー夫妻とその家族の事実を入力してみよう．

```
cg-user(1): (in-package :krep)
#<The krep package>
kr(2): (index-new-fact '(hasHusband Marie_Curie Pierre_Curie))
t
kr(3): (index-new-fact '(hasHusband Irene_Joliot-Curie Jean_Frederic_Joliot-Curie))
t
kr(4): (index-new-fact '(hasMother Irene_Joliot-Curie Marie_Curie))
t
kr(5): (index-new-fact '(hasFather Irene_Joliot-Curie Pierre_Curie))
t
```

データベースを直接検索するには retieve を用いる．

```
(defun fetch (query)
  "Return a list of buckets potentially matching the query,
  which must be a relation of form (predicate . args)."
  (dtree-fetch query (get-dtree (predicate query))
               nil 0 nil most-positive-fixnum))

(defun mapc-retrieve (fn query &optional (bindings no-bindings))
  "For every fact that matches the query,
  apply the function to the binding list."
  (dolist (bucket (fetch query))
    (dolist (answer bucket)
      (let ((new-bindings (unify query answer bindings)))
        (unless (eq new-bindings fail)
          (funcall fn new-bindings))))))

(defun retrieve (query &optional (bindings no-bindings))
  "Find all facts that match query.  Return a list of bindings."
  (let ((answers nil))
    (mapc-retrieve #'(lambda (bindings) (push bindings answers))
                   query bindings)
    answers))


kr(7): (retrieve '(hasHusband ?x ?y))
(((?y . Pierre_Curie) (?x . Marie_Curie)) ((?y . Jean_Frederic_Joliot-Curie) (?x .
Irene_Joliot-Curie)))
kr(8): (retrieve '(hasMother ?x Marie_Curie))
(((?x . Irene_Joliot-Curie)))
kr(9): (retrieve '(hasFather Irene_Joliot-Curie ?y))
(((?y . Pierre_Curie)))
```

見てわかるように unification を使ってパターンマッチにより強力な検索が可能であるが，いわゆる Prolog の推論機能はない．

付9.2 カテゴリ，個体，関係，値域・領域制約

krep ではここでカテゴリ(category) と個体(indibidual) , 関係(relation) と値域・領域制約を導入する．ここで言うカテゴリとサブカテゴリはRDFやOWLでいうところのクラスや概念であり，曖昧な意味を有する我々が通常使う用語，いわゆる「カテゴリ」ではない．以下では krep でいうサブカテゴリ，スーパーカテゴリという言葉を避けて，サブクラス，スーパークラスという言葉を使うことにする．組み込みの述語 sub は二つのクラスがサブクスとスーパークラスの関係にあることをいう．これはRDFで言えば rdfs:subClassOf に相当するものである．組み込み述語 ind はあるオブジェクトがあるクラスの個体(individual) であることをいう．これはRDFで言えば rdf:type のことである．また，組み込み述語 rel は個体についてのある関係について，1番目の個体は領域クラスに所属し，2番目の個体は値域クラスに所属していることを言う．これはRDFで言えばrdf:propertyOf とその領域制約(rdfs:domain) と値域制約(rdfs:rang) のことである．つまり，個体間の関係は組み込み述語 val で記述されるが，そのとき領域個体は領域制約クラスに所属しなければならないし，値域個体は値域制約クラスに所属しなければならない．

```
(sub サブクラス スーパークラス)
(ind 個体 クラス)
(rel 関係 領域制約クラス 値域制約クラス)
(val 関係 領域個体 値域個体)
```

これまで本文で何度も出てきた動物のルールについて，これらの組み込み述語を使って事実を記述すれば以下のようになる．

```
(cl:defpackage :animal
  (:shadow #:and)
  (:use :common-lisp :krep)
  (:export #:index)
  )

(in-package :animal)

(add-fact '(sub mammal animal))
(add-fact '(sub bird animal))
(add-fact '(sub carnivore mammal))
(add-fact '(sub ungulate mammal))
(add-fact '(sub cheetah carnivore))
(add-fact '(sub tiger carnivore))
(add-fact '(sub giraffe ungulate))
(add-fact '(sub zebra ungulate))
(add-fact '(sub ostrich bird))
(add-fact '(sub penguin bird))
(add-fact '(sub albatross bird))

(add-fact '(ind Skipper penguin))
(add-fact '(ind Kowalski penguin))
(add-fact '(ind Rico penguin))
(add-fact '(ind Clawhauser cheetah))
(add-fact '(ind Jeremy giraffe))
(add-fact '(ind Gia giraffe))
(add-fact '(ind Fragmeyer tiger))
(add-fact '(ind Jackson tiger))
(add-fact '(ind Yolanda zebra))
(add-fact '(ind Rolfina zebra))
```

```
(add-fact '(rel has mammal hair))
(add-fact '(rel gives mammal milk))
(add-fact '(rel has bird feathers))
(add-fact '(kr:and (rel can bird fly)
                   (rel lays bird eggs)))
(add-fact '(rel eats carnivore meats))
(add-fact '(kr:and (rel has carnivore pointed_teeth)
                   (rel has carnivore claws)
                   (rel has carnivore forward_eyes)))
(add-fact '(rel has ungulate hoofs))
(add-fact '(rel chews ungulate cud))
(add-fact '(kr:and (rel has cheetah tawny_color)
                   (rel has cheetah dark_spots)))
(add-fact '(kr:and (rel has tiger tawny_color)
                   (rel has tiger black_stripes)))
(add-fact '(kr:and (rel has giraffe long_neck)
                   (rel has giraffe long_legs)
                   (rel has giraffe dark_spots)))
(add-fact '(rel has zebra black_stripes))
(add-fact '(kr:and (rel cannot ostrich fly)
                   (rel has ostrich long_neck)
                   (rel has ostrich long_legs)
                   (rel isColored ostrich black_and_white)))
(add-fact '(kr:and (rel cannot penguin fly)
                   (rel can penguin swims)
                   (rel isColored penguin black_and_white)))
(add-fact '(rel can albatross flies_well))
```

ここで，述語 ind の1番目のパラメータは個体を指示し，2番目のパラメータはその個体が所属するクラスを指示する．また，述語 rel の次のパラメータは2番目と3番目のパラメータの関係を記述する述語である．上記では，has, gives, can, lays などであるが，ここにくる2番目と3番目のパラメータはクラスであることを重ねて注意しておく．すなわち，(has mammal hair) や (gives mammal milk) において，hair や milk もクラスである．ここまでのところで，個体について知識を記述すれば，以下のようになる．

```
(add-fact '(kr:and (val has Clawhauser tawny_color-1)
                   (val has Clawhauser dark_spots-1)))

(add-fact '(kr:and (val cannot Kowalski fly-1)
                   (val can Kowalski swims-1)
                   (val isColored Kowalski black_and_white-1)))
```

ここで，Kowalski はペンギンである個体であるが，fly-1 もクラス fly のインスタンスである．同様に，Clawhauser はチータである個体であるが，tawny_color-1 もクラス tawny_color のインスタンスである．ペンギンやチータは個体として違和感がないが，色や動作についてはどうしても違和感がぬぐえない．あえて言えば　Kowalski の動作のこと，Clawhauser の体色のことと思ってもらうしかない．

参考文献（下）

Charniak, Eugene, Chistopher Riesbeck, Drew V. McDermott, and James R. Meehan: *Artificial Intelligence Programming*, 2nd Ed., LEA (1987)

Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J.: "On Generality and Problem Solving: A Case Study Using the DENDRAL Program," In Meltzer, B. and Michie, D. (Eds.), *Machine Intelligence* 6, pp. 165–190. Edinburgh University Press (1971) (Also Stanford Heuristic Programming Project Memo [30] HPP-70-5, and Computer Science Memo STAN-CS-176.)

Fikes, Richard E., and Nils J. Nilsson: "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence* 2, 189-208 (1971), Reprinted in *Computation & Intelligence*, (George F. Luger, ed.) (1995)

Ghallab, Malik, Dana Nau, Paolo Traverso: *Automated Planning Theory and Practice*, Morgan Kaufmann (2004)

Graham, Paul: *On Lisp*, Prentice Hall (1993), On Lisp(野田訳), オーム社 (2006)

Hawkins, Jeff, Sandra Blakeslee: *On Intelligence,* Times Books (2004), 考える脳考えるコンピューター（伊藤訳）, ランダムハウス講談社 (2005)

Hayes-Roth, Frderick, Donald A. Waterman, Douglas B. Lenat (eds.): *Building Expert Systems,* Addison-Weslay (1983), エキスパートシステム（AIUEO訳）, 産業図書 (1985)

Kiczales, Gregor, Jim des Rivierès, Daniel G. Bobrow: *The Art of the Metaobject Protocol*, MIT (1992)

McDermott, Drew: Artificial Intelligence Meets Natural Stupidity, SIGART Bull., 57, pp.4-9, ACM (April,1976)

Minsky, Marvin: "A Framework for Representing Knowledge", MIT-AI Laboratory Memo 306 (1974). Reprinted in *The Psychology of Computer Vision*, P. Winston (ed.), McGraw-Hill(1975). コンピュータビジョンの心理, 白井・杉原訳, 産業図書(1979). Shorter versions in J. Haugeland, (ed.), *Mind Design*, MIT Press(1981), and in Cognitive Science, Collins, Allan and Edward E. Smith (eds.) Morgan-Kaufmann (1992) ISBN 55860-013-2.

Neisser, Ulric: Cognition and Reality, W.H.Freeman (1976), 認知の構図, 古崎・村瀬共訳, サイエンス社 (1978), ISBN 4-7819-0004-6

Newell, Allen, and Herbert A. Simon: "GPS, A Program that Simulates Human Thought," Lernende Automaten, #191, Munchen (R. Oldenbourg, ed.) (1961), Reprinted in *Computers and Thought* (Feigenbaum and Feldman, eds.) (1963) , コンピュータと思考 (阿部,横山監訳) 好学社 (1969), Reprinted in *Computation & Intelligence*, (George F. Luger, ed.) (1995)

Nilsson, Nils J.: Principles of Artificial Intelligence, Tioga Publishing Co. (1980) , 人工知能の原理 (白井, 辻井, 佐藤訳) 日本コンピュータ協会 (1983)

Norvig, Peter: *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann (1992)

Riesbeck, K. Chirstopher, Roger C. Schank: *Inside Case-Based Reasoning*, LEA (1989)

Russell, Stuart J., Peter Norvig: *Artificial Intelligence A Modern Approach*, Pretice-Hall (1995), 古川(監訳), エージェントアプローチ人工知能, 共立出版 (1997)

Schank, Roger C., Alex Kass, Christopher K. Riesbeck: *Insiede Case-Based Explanation*, LEA (1944)

Shortliffe, Edward H.: 5 Details of the Consultation System, "*Rule-Based Expert Systems. The MYCIN Experiments of the Stanford Heuristic Programming Project*", Bruce G. Buchanan, Edward H. Shortliffe (eds.), Addison-Wesley (1985)

Suchman, Lucy A.: *Plans and Situated Actions The problem of human machine communication*, Cambridge (1987), 佐伯(監訳)，プランと状況的行為 人間-機械コミュニケーションの可能性，産業図書 (1999)

Weizenbaum, Joseph: *Comupter Power and Human Reason*, W. H. Freeman and Company (1976), 秋葉(訳), コンピュータパワー, サイマル出版会 (1979)

Wilkins, David E.: *Practical Planning*, Morgan Kaufmann (1988)

長尾 真, 淵 一博: 論理と意味, 岩波書店(1983)