On Lisp

Paul Graham

To my family, and to Jackie.

$\lambda \ldots$

前書き

この本は Lisp プログラマとして成長したいと思っている全ての人に向けて書かれた．読者が既に Lisp に親しんでいることを前提としたが，長いプログラミング経験が必ずしも要る訳ではない．始めの数章はかなりの量の復習を含んでいる．これらの章はきっと熟練 Lisp プログラマにも面白く思ってもらえるだろう．そこでは見慣れた話題に新しい光を当ててみたからだ．

あるプログラミング言語のエッセンスを一文で伝えるのは難しいが，John Foderaro の言葉はかなりそれに近い：

Lisp はプログラム可能なプログラミング言語である．

Lisp はそれだけのものではないが，Lisp を自分の意図に従わせる能力は Lisp エキスパートと初心者との大きな違いだ．熟練 Lisp プログラマは，プログラミング言語に従ってプログラムを書くのと同じように自分の書くプログラムに向けてプログラミング言語を構築していく．この本では，Lisp に元々向いているボトムアップ・スタイルでのプログラミング方法を教える．

-1.1 ボトムアップ・デザイン

ボトムアップ・デザインは，ソフトウェアがどんどん複雑化していく中で一層重要になってきている．現代のプログラムは，とてつもなく複雑だったり，ときには解釈が定まらないような仕様を満たさなければならない．そのような状況では伝統的なトップダウン方式が破綻してしまうことがある．そこで，ほとんどの計算機科学系学科で現在教えられているのとは大変異なったプログラミング手法が発達した：ボトムアップ・スタイルだ．そこではプログラムは一連の層として記述され，それぞれの層がその 1 段上に対して一種のプログラミング言語の役割を果たす．そうして書かれたプログラムの例は X Windows と TEX だ．

この本のテーマは二つある．Lisp がボトムアップ・スタイルで書かれるプログラムに対して自然なプログラミング言語であるということと，ボトムアップ・スタイルが Lisp プログラムを書くのに自然な方法であるということだ．だから *On Lisp* は 2 種類の読者にとって興味深いものになるだろう．拡張可能なプログラムを書くことに興味のある人には，この本は適切なプログラミング言語があれば何ができるのかを示す．Lisp プログラマには，この本は Lisp を最大限に活用するための実用的な説明を提供する．

この本のタイトルは Lisp でのボトムアップ・プログラミングの重要性を強調するつもりで付けられた．プログラムをただ Lisp で書くだけでなく，独自の言語を ***Lisp* を基に** *(on Lisp)* 書くことが可能であり，プログラムはその言語で書くことができる．

どのプログラミング言語でもボトムアップでプログラムを書くことはできるが，このスタイルのプログラミングには Lisp は最も自然な器だ．Lisp では，ボトムアップ・デザインは異常に大規模だったり複雑なプログラム専用の，特別なテクニックではない．しっかりしたプログラムはみな，部分的にはこのスタイルで書かれることになるだろう．Lisp は開発当初から拡張可能なプログラミング言語となるよう設計されていた．言語そのものの大半が Lisp の関数の集合体であり，それらはユーザ自身の定義によるものと何も違わない．それどころか Lisp の関数はリストとして表現できるが，リストは Lisp のデータ構造なのだ．このことは，ユーザが Lisp コードを生成する Lisp 関数を書けるということだ．

いい Lisp プログラマはこの可能性を利用する方法を知らなければならない．そのための方法は普通，**マクロ**と呼ばれる一種のオペレータを定義してみることだ．マクロの理解は，正しい Lisp プログラムを書くことから美しい Lisp プログラムを書くことへ移行する際の最も重要なステップの一つだ．入門的な Lisp 本に載っているのはマクロの大雑把な外観程度：マクロとは何かの説明と，ユーザがマクロを使えば可能になる奇妙で素晴らしいことのヒントとなる幾つかの例だ．ここではそういった奇妙で素晴らしい事柄に特に注意してみる．この本の狙いの一つは，そういったユーザ達がマクロに関する経験から今に至るまでに学ばなければならなかったことを一箇所にまとめることだ．

無理もないことだが，入門的な Lisp 本は Lisp と他のプログラミング言語との違いを強調しない．そういった本は（大抵）Pascal 用語でプログラムを考えるよう教育されてきた生徒達にメッセージを伝えなければならない．「Lisp の defun は Pascal の手続き定義に似ているが，実際は関数オブジェクトを作るコードを生成するプログラムを書くプログ

ラムであり，第 1 の引数として与えられたシンボルを名札に使うのだ.」と説明しても，混乱を招くばかりだろう．

この本の目的の一つは，Lisp と他のプログラミング言語との違いは何かを説明することだ．私は当初から，他の点がみな同じだったら C や Pascal や Fortran ではなく Lisp でプログラムを書いた方がずっといいということは分かっていた．これは単なる好みの問題ではないことも分かっていた．しかし実際に Lisp が何らかの点でよりいいプログラミング言語だと主張するなら，それがなぜかを説明できるようにした方がいいことに気が付いたのだ．

誰かが Louis Armstrong にジャズとは何かと尋ねたとき，彼はこう答えた．「ジャズとは何かが人に聞かないと分からないんだったら，分かるようにはならないだろうね.」しかし彼はある方法で確かにその質問に答えた：彼は人々にジャズとは何かを**示した**のだった．それが Lisp の力を説明する唯一の方法だ——他のプログラミング言語では困難だったり，不可能な技をやってみせることだ．プログラミングに関する本の大半が——Lisp プログラミングの本さえ——どんなプログラミング言語でも書けるような題材を扱っている．*On Lisp* は，Lisp でのみ書けるような類のプログラムを主に扱う．拡張性，ボトムアップ・プログラミング，インタラクティブな開発，ソースコードの変換，埋め込み言語——Lisp の長所が現れるのはそういった所だ．

もちろん原則的には，Turing 機械と等価なプログラミング言語はみな他のプログラミング言語と同じ作業が可能だ．しかしそういった能力は，プログラミング言語が目的とするものではない．原則的には，プログラミング言語でできることはみな Turing 機械でもできる．しかし実際には，Turing 機械のプログラミングは手間をかけるに値することではない．

だから私が「これは他のプログラミング言語では不可能なことのやり方についての本だ」と言うとき，数学的な意味で「不可能」だと言うのではなく，プログラミング言語に関わる意味でそう言っているのだ．つまり，この本の中のプログラムの幾つかを C で書かなければならないなら，始めに C で Lisp コンパイラを書くことで目的を達しても同じなのだから．例えば C で Prolog を埋め込み言語にしてみる——それにかかる手間は想像がつくだろうか？第 24 章では，180 行の Lisp コードを使ってその方法を示す．

しかし，ただ Lisp の能力のデモンストレーションをする以外にもっとやりたいことがある．**なぜ** Lisp は他と違っているのかも説明したい．これは実は微妙な問いだ——非常に微妙な問いなので「記号演算 (Symbolic computation)」などという言葉を答えにすることはできない．私がこれまでに学んだことはできる限り明快に説明するよう試みた．

-1.2　この本の方針

関数は Lisp プログラムの基礎なので，この本の最初の幾つかの章では関数を扱う．第 2 章では，Lisp の関数とは何かということと，関数のもたらす可能性を説明する．第 3 章では，Lisp プログラムでの主要なプログラミング・スタイルである関数プログラミングの長所について議論する．第 4 章では，Lisp の拡張のための関数の使い方を示す．そして第 5 章では，別の関数を返す関数を使って定義できる，新たな種類の抽象化を示唆する．最後に第 6 章では，伝統的なデータ構造の代わりに関数を使う方法を示す．

この本の残りの章では，関数よりもマクロの方を多く扱う．マクロにはより多くの注意を払った．一つにはマクロにはもっと語るべきことがあるからで，また書籍では今まで十分に説明されていなかったからでもある．第 7–10 章では，マクロに関するテクニックについての完全な手引きを与える．そこまでで，熟練 Lisp プログラマがマクロについて知っていること——作用の仕組み，定義方法，テスト，デバッグ，使うべきときとそうでないときの区別，マクロの主要な種類，マクロ展開を生成するプログラムの書き方，マクロのスタイルと Lisp のスタイルとの一般的な違い，マクロに影響するそれぞれの独特な問題を発見し，修正する方法——は全て理解できるだろう．

この手引きに続き，第 11–18 章ではマクロによって可能になる強力な抽象化を幾つか示す．第 11 章では，コンテキストを形成したり，繰り返しや条件分岐を実現する古典的なマクロの書き方を示す．第 12 章では，一般化された変数への作用におけるマクロの役割を説明する．第 13 章では，マクロを使い，行うべき計算をコンパイル時にやってしまうことでプログラムを速く走らせる方法を示す．第 14 章ではアナフォリック（anaphoric）マクロを導入する．これはプログラム内で代名詞を使うことを可能にする．第 15 章では，マクロを使って第 5 章で定義された関数ビルダに便利なインタフェイスを提供する方法を示す．第 16 章では，Lisp にユーザのプログラムをユーザのために書かせるために使われる，マクロを定義するマクロの使い方を示す．第 17 章ではリードマクロ（read macro）について，第 18 章では構造化代入（destructuring）のためのマクロについて議論する．

第 19 章からはこの本の第 4 部——埋め込み言語のために割かれた部分が始まる．第 19 章では，2 種類の同じ働きの

プログラムを示すことでその話題を導入する．それはデータベースに対する問い合わせ（クエリ）に答えるものだが，最初はインタプリタによって実装されたものを，そして次には本物の埋め込み言語として実装されたものを示す．第 20 章では，Common Lisp プログラムに，計算の残りを表すオブジェクトである継続（continuation）の概念を導入する方法を示す．継続は非常に強力なツールで，複数プロセスと非決定的選択の両方の実装に使える．これらの制御構造を Lisp に埋め込む方法は，それぞれ第 21, 22 章で議論する．非決定性を使えば予知能力を持っているかのようなプログラムが書けるようになるが，これは不思議な能力の概略のように思えていることだろう．第 23, 24 章では，非決定性がその触れ込みに違わないことを示す埋め込み言語を 2 個取り上げる：完全な ATN パーサと埋め込み Prolog だが，コードは全部で 200 行余り† になる．

これらのプログラムが短いという事実は，それ自身の中身は何もないということを表している．不可解なコードを書くことに頼ったなら，200 行で可能なことについては何も分からない．これらのコードが短く済むのは，プログラミング上のトリックに頼ったからではなく，Lisp が本来意図していた使い方に従って書かれているからなのだ．それこそが重要な点だ．第 23, 24 章の中核は，ATN パーサを 1 ページ分のコードで実装したり，Prolog を 2 ページ分のコードで実装する方法ではない．最も自然な Lisp による実装によれば，それらのプログラムは単にその短さで済むという事実だ．その後の章で使われる埋め込み言語は，最初のポイント二つ——Lisp はボトムアップ・デザインのための自然な言語であることと，ボトムアップ・デザインは Lisp の自然な使用方法であること——の例に基づく証明を与える．

この本のまとめは，オブジェクト指向プログラミングの議論と，特に CLOS (the Common Lisp Object System) だ．この話題を最後に取っておくことで，オブジェクト指向プログラミングが Lisp 内に既にある考え方の延長に位置するということを一層はっきりと理解できる．それは *Lisp* **で**構築できる数多くの抽象化構造のうちの一つなのだ．

注釈は 230ページから始まり，参考文献案内，追加または代替用のコードや，その場には直接関係のない Lisp の側面の説明も含む．注釈は † を肩に乗せて示した by 訳者．またパッケージに関する付録（230ページ）もある．

ニューヨークのツアーが世界の文化の大半のツアーになりうるのと同様，プログラム可能なプログラミング言語としての Lisp の学習は Lisp テクニックの大部分のスケッチになる．ここで説明されたテクニックの大半は Lisp コミュニティに知られているが，今までどこにも書かれていなかったことも多い．またマクロの適切な役割や変数キャプチャの本質といった話題は，多数の熟練 Lisp プログラマたちもおぼろげにしか理解していなかった．

-1.3 例

Lisp はプログラミング言語の一族だ．そのうち Common Lisp はこの先ずっと広く使われる方言 (dialect) だろうから，この本の例の大半は Common Lisp で書かれている．この言語は元々 1984 年に Guy Steele の本 *Common Lisp: the Language* (CLtL1) によって定義されたものだ．この定義は 1990 年に第 2 版 (CLtL2) の出版によって改訂された†．今度は将来の ANSI 標準がその役目を引き継ぐことだろう[*1]．

この本には，式一個から正しく動作する Prolog の実装まで，数百の例が載っている．この本のコードのうち，可能なものはみな全てのヴァージョンの Common Lisp で動作するように書かれている．CLtL1 の実装にない機能を必要とするわずかな例は，本文中にその旨を明記してある．後半の章には Scheme による例も幾つかある．これらもはっきりと分かるようにしてある．

コードは `endor.harvard.edu` の anonymous FTP の `pub/onlisp` ディレクトリから手に入る．質問やコメントは `onlisp@das.harvard.edu` に送ってほしい．

-1.4 謝辞

While writing this book I have been particularly thankful for the help of Robert Morris. I went to him constantly for advice and was always glad I did. Several of the examples in this book are derived from code he originally wrote, including the version of for on page 127, the version of aand on page 191, match on page 239, the breadth-first true-choose on page 304, and the Prolog interpreter in Section 24.2. In fact, the whole book reflects (sometimes, indeed, transcribes) conversations I've had with Robert during the past seven years. (Thanks, rtm!)

I would also like to give special thanks to David Moon, who read large parts of the manuscript with great care, and gave me very useful comments. Chapter 12 was completely rewritten at his suggestion, and the example of variable capture

[*1] 訳注：1996 年に ANSI Common Lisp（アメリカ公式標準）制定．GCL，Clisp，CMUCL，SBCL などフリーの実装あり．

on page 119 is one that he provided.

I was fortunate to have David Touretzky and Skona Brittain as the technical reviewers for the book. Several sections were added or rewritten at their sugges-tion. The alternative true nondeterministic choice operator on page 397 is based on a suggestion by David Toureztky.

Several other people consented to read all or part of the manuscript, including Tom Cheatham, Richard Draves (who also rewrote alambda and propmacro back in 1985), John Foderaro, David Hendler, George Luger, Robert Muller, Mark Nitzberg, and Guy Steele.

I'm grateful to Professor Cheatham, and Harvard generally, for providing the facilities used to write this book. Thanks also to the staff at Aiken Lab, including Tony Hartman, Janusz Juda, Harry Bochner, and Joanne Klys.

The people at Prentice Hall did a great job. I feel fortunate to have worked with Alan Apt, a good editor and a good guy. Thanks also to Mona Pompili, Shirley Michaels, and Shirley McGuire for their organization and good humor.

The incomparable Gino Lee of the Bow and Arrow Press, Cambridge, did the cover. The tree on the cover alludes specifically to the point made on page 27.

This book was typeset using LaTeX, a language written by Leslie Lamport atop Donald Knuth's TeX, with additional macros by L. A. Carr, Van Jacobson, and Guy Steele. The diagrams were done with Idraw, by John Vlissides and Scott Stanton. The whole was previewed with Ghostview, by Tim Theisen, which is built on Ghostscript, by L. Peter Deutsch. Gary Bisbee of Chiron Inc. produced the camera-ready copy.

I owe thanks to many others, including Paul Becker, Phil Chapnick, Alice Hartley, Glenn Holloway, Meichun Hsu, Krzysztof Lenk, Arman Maghbouleh, Howard Mullings, Nancy Parmet, Robert Penny, Gary Sabot, Patrick Slaney, Steve Strassman, Dave Watkins, the Weickers, and Bill Woods.

Most of all, I'd like to thank my parents, for their example and encouragement; and Jackie, who taught me what I might have learned if I had listened to them.

この本を読むのはきっと楽しいと思う．私は，知っているプログラミング言語のうちで Lisp が一番好きだ．それはただ Lisp が一番美しいからだ．この本は，最高に Lisp っぽい Lisp (Lisp at its lispiest) についての本だ．この本を書くのは楽しかった．その感じが行間に流れていたら嬉しい．

Paul Graham

目次

0　拡張可能なプログラミング言語

遠くない昔，Lisp は何のためのプログラミング言語かと尋ねれば，多くの人が「人工知能 (AI) 用のプログラミング言語」と答えただろう．実際は Lisp と AI との関係は歴史の偶然に過ぎない．Lisp の開発者は John McCarthy で，彼は「人工知能」という言葉の提唱者でもある．彼の生徒と同僚達は彼らのプログラムを Lisp で書いた．それが Lisp が AI 用のプログラミング言語と言われ出したきっかけだ．このつながりは広く取り上げられ，1980 年代の短い AI ブームの間に大変な程繰り返されたので，ほとんど迷信のようになってしまった．

幸運なことに，AI だけが Lisp の目的でないとの言葉が広まり始めた．最近のハードウェアとソフトウェアの進歩のおかげで，Lisp は商業的にも成功し始めた：今では最高の UNIX 系テキストエディタ GNU Emacs，業界標準のデスクトップ CAD ソフト Autocad，そして先駆的ハイエンド出版ソフト Interleaf で使われている．これらのプログラムでの Lisp の使われ方は AI とは何も関係ない．

Lisp が AI 用プログラミング言語でないのなら，いったい何なのか？ Lisp を取り巻く仲間でそれを判断するのでなく，言語そのものを見てみよう．他のプログラミング言語でできないことのうち，Lisp には何ができるか？ Lisp の最も特徴的な性質の一つは，Lisp によって書かれているプログラムに合わせて Lisp を仕立てることができる点だ．Lisp そのものが Lisp のプログラムの一つであり，Lisp のプログラムはリストとして表現できるが，リストは Lisp のデータ構造なのだ．これら 2 個の原則が相俟って，どのユーザも組み込みのものと区別の付かないオペレータを Lisp に追加できることになる．

0.1　進化によるデザイン

Lisp ではユーザに独自のオペレータを定義する自由があるので，Lisp を必要なプログラミング言語にきっちり仕立てることができる．ユーザがテキストエディタのプログラムを書いているのなら，Lisp をテキストエディタを書くための言語に変えることができる．また CAD ソフトのプログラムを書いているのなら，Lisp を CAD ソフトを書くための言語に変えることもできる．そしてどんなプログラムを書くかまだ確かでないなら，Lisp で書いておくのが安全な賭けだ．それがどんな種類のプログラムになったとしても，それを書いている間に，Lisp は**その種類の**プログラムを書くためのプログラミング言語に進化していることだろう．

どんなプログラムを書くかまだ確かでないなら？人によってはこの文は奇妙に響くだろう．その人は (1) これからやることを注意深く計画し，そして (2) それを実行する，というモデルとの不愉快な対照を目にしたことになる．このモデルによれば，プログラムのすべき動作を決める前にプログラムを書くことを Lisp が勧めるとすれば，それは的外れな考え方を勧めているに過ぎないことになる．

さて，全くそんなことはない．「計画－実装」方式もダム建設や侵略作戦の決行にはいい方法だったが，人々の経験によればプログラムを書くのにいい方法だったかどうかは定かでない．なぜか？ きっとそれは，コンピュータは大変正確だからだ．きっとプログラムにはダムや侵略作戦よりもヴァリエーションが多いからだ．または「余分」ということに関する古い概念とソフトウェア開発に類似点がないせいで，古い方式が機能しないことによるのだろう：ダムが 30% 余分なコンクリートを使っていても，それは間違いなのかどうかぎりぎりの所だ．しかしプログラムが 30% 余分な動作をしていたら，それは**間違い**だ．

古い方式が失敗に終わるのがなぜかを言うのは難しいかもしれないが，それは確かに失敗し，結果は誰の目にも明らかだ．ソフトウェアが期日通りに完成したことがあるだろうか？熟練プログラマは，どれ程慎重にプログラムの計画を立てても，プログラムを書き始めると，必ず計画にどこか不完全な点が見つかることを知っている．計画が望みのない程まで間違っていることもある．しかし「計画－実装」方式の犠牲者のほとんどはその基礎の健全さを疑おうとはしない．代わりに彼らは人間の失敗を責める：「もう少しいい展望の下に計画を立ててさえいたら，こんな問題は全て避けられただろう．」最高レベルのプログラマでさえ実装となれば問題に突き当たるのだから，人々が必ずそれだけの展望を持つことを望むのはどうやら無茶のようだ．おそらく「計画－実装」方式は，我々の持つ限界により適した別のアプローチで置き換えることができるだろう．

適切なツールがあるならば，プログラミングへの別のアプローチが可能だ．なぜ実装の前に計画を立てるのか？プロジェクトの立案にどっぷり浸かり込むことの大きな危険性は，自らを困難に追い込んでしまう可能性がある点だ．もっ

と柔軟なプログラミング言語があれば，この心配を減らせるのでは？まさにその通りだ．Lisp の柔軟性は全く新しいプログラミングのスタイルを生み出した．Lisp においては，計画の大部分を立てるのはプログラムを書きながらでいい．

なぜ後知恵が浮かぶのを待つのか？ Montaigne が気付いたように，考えを明確にするにはそれを書き下ろそうとすることが一番だ．自らを困難に追い込んでしまうとの心配からひとたび解放されれば，この可能性を最大限に活用できる．プログラムを書きながら計画を立てる能力には二つの重要な結果につながる：まず，プログラムを書くのにかかる時間が短くなる．それは計画を立てると同時にプログラムを書くと，注意を集中することで本当のプログラムが出来上がるからだ．そしてその方法で出来たプログラムはよりいいものであると分かるだろう．それはプログラムの最終デザインは必ず進化の産物だからだ．プログラムの目的を探す間，間違っていた部分を，明らかになったその場で必ず書き直すという原則を守る限り，最後に出来上がったものは，あらかじめ計画に数週間を費やした場合よりも優美なプログラムになるだろう．

Lisp が多目的なプログラミング言語だからこそ，この種のプログラミングが実用的な代替手段になる．事実，Lisp の持つ最大の危険は Lisp がユーザに悪影響を与えるかもしれないことだ．一度 Lisp をしばらく使うと，プログラミング言語とアプリケーションとの相性に敏感になりすぎ，元々使っていたプログラミング言語に戻っても，これでは必要な柔軟性が手に入らないという思いに常に囚われるようになりかねない．

0.2 ボトムアップ・プログラミング

プログラムの機能的要素は余り大きくなるべきでないというのが，プログラミング・スタイルの長年の原則だ．プログラムの構成要素が，読めば理解できる状態を超えて肥大化するなら，それは複雑さの固まりに成り果て，（大都市が流れ者を隠すように）簡単にエラーを覆い隠す．そういうソフトウェアは読み辛く，テストし辛く，デバッグし辛い．

この原則に従い，大規模なプログラムは部品へと分割しなければいけない．またプログラムが大規模であればある程，さらに分割しなければいけない．プログラムを分割する方法とは何か？伝統的アプローチは**トップダウン・デザイン**と呼ばれる：「このプログラムの目的はこれらの 7 個だから，プログラムを 7 個の主要サブルーチンに分割することにする．1 個目のサブルーチンはこれら 4 個のことを行わなければいけないから，それ自身のサブルーチンが今度は 4 個になり，...」といった具合だ．このプロセスはプログラム全体が適切なバラバラ状態——全ての部分が意味のある仕事を行えるだけの大きさでありながら，単一のユニットとして理解できる位小さくなった状態——になるまで続く．

熟練 Lisp プログラマがプログラムを分割する方法は違っている．トップダウン・デザインと同様，彼らには従う原則があり，それは**ボトムアップ・デザイン**——プログラミング言語を問題に適するように変えていく——と呼ばれる．Lisp では，プログラムをただプログラミング言語に従って書くことはしない．プログラミング言語を自分の書くプログラムに向けて構築するのだ．プログラムを書いているとき，「Lisp に○△のオペレータがあればなあ.」と思うことがあるかもしれない．そうしたらそれを書けばいい．後で，新しいオペレータの使用がプログラムの別の部分のデザインを簡潔にまとめることにつながったと気付くだろう．そういった感じでプログラミング言語とプログラムは共に進化する．交戦中の 2 国間の国境のように，二つの境界は何度も書き直される．それが落ち着くのは，山や川（ここではユーザの問題の持つ自然な境界）に辿り着いたときだ．最終的には，ユーザのプログラムではプログラミング言語がそのために設計されたかのような見掛けになる．そしてプログラミング言語とプログラムが相性良く落ち着いたとき，ユーザは明快で小規模で効率的なコードを手にする．

ボトムアップ・デザインは，同じプログラムをただ別の順番で書くだけのことではないということは，強調する価値がある．ボトムアップで作業すると，大抵別のプログラムが出来上がる．単一の一体となったプログラムではなく，抽象的なオペレータを持つ大規模なプログラミング言語と，それで書かれた小規模なプログラムが出来るのだ．ユーザは敷石ではなく，高いアーチを手にする．

典型的なコードでは，単なる簿記に過ぎない部分を一度抽象化すると，残るものはずっと短い．プログラミング言語を高く構築すればする程，頂点からそこに至るまでの道のりは短くなる．これは幾つかの長所をもたらす：

1. プログラミング言語に仕事を任せることで，ボトムアップ・デザインでは小規模で機敏なプログラムが生まれる．短いプログラムは多数の構成要素に分割する必要がない．そして構成要素が少ないということは，プログラムが読み易く修正し易いものだということだ．また構成要素が少ないということは，構成要素同士の連結も少ないということなので，そこでエラーが起きる可能性も少ない．工業デザイナが機械の可動部分を減らそうと努力

するのと同様に，熟練 Lisp プログラマはボトムアップ・デザインを使ってプログラムの大きさと複雑さを減らそうとする．

2. ボトムアップ・デザインはコードの再利用を促進する．プログラムを二つ以上書くとき，最初のプログラムのために書いたユーティリティの多くは他のプログラムでも有用になる．ユーティリティの大規模な基盤を手にしてしまえば，新しいプログラムを書く手間は，Lisp そのもので書かなければならないときにかかる手間の数分の一に過ぎない．
3. ボトムアップデザインはプログラムを読み易くする．この種の抽象化のインスタンスは，読む人に汎用オペレータを理解するよう要求する．機能的抽象化のインスタンスは，読む人に特殊目的のサブルーチンを理解するよう要求する[*2]．
4. ユーザにコード内のパターンに常に関心を払うようにさせる．ボトムアップでの作業はプログラムのデザインに関するアイディアを明確にするのに役立つ．一つのプログラムの中で種類の違う 2 個の構成要素が形の上で似ているなら，類似性に気付くよう促されるし，おそらく単純な方法でプログラムをデザインし直すよう促されるだろう．

ある程度までは，ボトムアップ・デザインは Lisp 以外のプログラミング言語でも可能だ．ライブラリ関数を見ればそこには必ずボトムアップ・デザインの例がある．しかし Lisp はこの点に関してずっと幅広い力を与え，プログラミング言語にそれに比例して Lisp のスタイルの中で大きな役割を演じるよう促す——その役割が余り大きいので，Lisp はただのプログラミング言語の一つではなく，全く違ったプログラミング方法になっている．

このスタイルの開発は，幾つかの小規模グループによって書ける程度のプログラムに適しているというのは確かに真実だ．しかし同時に，このスタイルは小規模グループのなし得ることの限界を拡張する．Frederick Brooks が「人月の神話」(*The Mythical Man-Month*)[†] の中で示唆したのは，プログラマのグループの生産性はその規模に対して線形には増大しないということだった．グループのサイズが増大するにつれ，個々のプログラマの生産性は低下してゆく．Lisp プログラミングの経験はこの法則を表現するためのより素敵な方法を提案する：グループのサイズが減少するにつれ，個々のプログラマの生産性は向上してゆく．相対的に小規模なグループが勝利を収める．理由は単純，小さいから．小規模グループが Lisp によって可能になるテクニックを活用するとき，完全な勝利が待っている．

0.3 拡張可能なソフトウェア

Lisp スタイルのプログラミングは，ソフトウェアが複雑さを増すにつれ重要性を増してきた．今時のユーザからのソフトウェアへの要求は余りに多く，プログラマにはほとんど予想しきれない．ユーザ自身も要求の全てを予想することができなくなっている．しかしユーザの望みを完璧に叶えるソフトウェアをプログラマが供給できなくとも，プログラマには拡張できるソフトウェアを供給することはできる．プログラマは自分のソフトウェアをただのプログラムからプログラミング言語に転換し，技術の高いユーザは必要な付加的機能をその上に作り上げるようになる．

ボトムアップ・デザインは拡張可能なプログラムに自然につながっていく．ボトムアップ・プログラムの最も単純なものは，2 層から成る：プログラミング言語とプログラムだ．複雑なプログラムも一連の層として書き上げることができる．それぞれの層は 1 段下の層に対してプログラミング言語として機能する．この方針が最上層まで貫かれれば，その層がユーザの使うプログラミング言語となる．そのようなプログラムではあらゆるレベルで拡張が可能になっていて，伝統的なブラックボックスとして書かれ，後から拡張性を付け加えたシステムよりも遥かにいいプログラミング言語が出来上がることが多い．

X Windows と TEX はこの原則に基づくプログラムの古い例だ．1980 年代には高性能のハードウェアによって Lisp を自らの拡張言語とする新世代のプログラムが可能になった．その最初は有名な UNIX のテキストエディタ，GNU Emacs だ．次は Autocad で，これは Lisp を拡張言語とする大規模商用製品としては初になる．1991 年には Interleaf の新ヴァージョンがリリースされたが，それは Lisp を拡張言語に使っていただけではなく，かなりの部分が Lisp で実装されていた．

Lisp は拡張可能なプログラムを書くためには素晴らしくいいプログラミング言語だ．それは Lisp 自身が拡張可能な

[*2] 「でもあんたのユーティリティを全部理解しないことには，プログラムが読めなくなるじゃないか.」そういった言葉は大抵誤りだ．なぜかについては，第 4.8 節を参照.

プログラムであるからだ．Lisp の拡張性をユーザにも渡すような Lisp プログラムを書けば，実質的には何もせずに拡張言語が出来たことになる．そして Lisp で Lisp プログラムを拡張することと，伝統的なプログラミング言語でそれを行うこととの違いは，誰かに直接会うことと手紙でやりとりすることとの違いのようなものだ．外部プログラムへのアクセス手段を提供することだけで拡張可能になったプログラムでは，既定のチャンネルを通じて連絡し合う二つのブラックボックスがせいぜいだ．Lisp では，拡張機能は背後にあるプログラム全体に直接アクセスできる．これはユーザにプログラムのあらゆる部分へのアクセスを与えなければならないということではなく，ただアクセスを与えるか与えないかの選択肢があるというだけのことだ．

このような部分的アクセスがインタラクティブな環境と結びつくと，最高の拡張性が得られる．自分のプログラムの拡張機能の基礎として使えるようなプログラムは，どれもかなり大規模になりがちだ．おそらく規模が大きすぎて全体的なイメージが掴めない程だろう．何か不確かな点があるときどうなるだろう？そのとき元のプログラムが Lisp で書かれていれば，それをインタラクティブに調べることができる：データ構造を調査できる．関数を呼び出せる．ソースコードを読むことさえできる．この種のフィードバックにより，確かな自信を持ってプログラムを製作できる．思い切った拡張機能を書けるし，しかもそれが素早くできる．インタラクティブな環境は常にプログラミングを容易なものにしてくれるが，拡張機能を書いているとき程それが有り難いときはない．

拡張可能なプログラムは諸刃の剣だが，最近の経験によればユーザはただの剣より諸刃の剣の方を好む．どんな危険性が備わっていようとも，拡張可能なプログラムの方が勝るようだ．

0.4 Lisp の拡張

Lisp に新しいオペレータを加えるには 2 通りの方法がある: 関数とマクロだ．Lisp ではユーザの定義した関数は組み込み関数と同じ地位を占める．もし `mapcar` の変種が新しく必要になったら，それを自分で定義し，`mapcar` を使うのと同じようにそれを使うことができる．例えば，ある関数が `1` から `10` までの整数に適用されたときにそれが返す値のリストが必要なら，新しいリストを作ってそれを `mapcar` に渡すことができる：

```
(mapcar fn
        (do* ((x 1 (1+ x))
              (result (list x) (push x result)))
          ((= x 10) (nreverse result))))
```

しかしこの方法は格好悪いし非効率だ[*3]. 代わりに新しい対応関数 `map1-n`（54 ページ参照）を定義し，それを次のように呼び出せばいい：

```
(map1-n fn 10)
```

関数を定義するのは比較的真っ当な方法だ．マクロは新しいオペレータのさらに一般的な（しかし余り理解されていない）定義方法を提供する．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．そしてその探求はこの本の主目的の一つなのだ．

マクロを注意して使えば驚異的に明確でエレガントなプログラムができる．これらの宝石のようなプログラムは，何もせずに得られたわけではない．最後にはマクロは世界で一番自然なものに思えるだろうが，最初は理解が難しい．これはマクロが関数よりも一般的で，書くときに注意すべきことが多いせいもある．しかしマクロの理解が難しい大きな理由は，それが全く異質な (*foreign*) ものだからだ．Lisp のマクロのようなものを持つ言語は他にない．だからマクロについて学ぶと，他のプログラミング言語からうっかり引き継いだ先入観を振り捨てることにつながるかもしれない．その中の主要なものは，死後硬直に悩まされるものとしてのプログラムという概念だ．なぜデータ構造が流動的で変更可能であるべきなのに，プログラムはそうでないのだろうか？ Lisp ではプログラムがデータ**である**のだ．しかしこの事実の持つ意味をモノにするには暫く時間がかかる．

マクロに慣れるのに暫く時間がかかっても，それは努力に見合うことだ．繰り返しのようなありふれた用途でも，マクロはプログラムを目覚ましく小型できれいなものに変える．ここで，あるプログラムがコード本体を `x` について `a` から `b` まで繰り返さなければならないとしよう．Lisp 組み込みの `do` はもっと一般的な目的のためのものだ．それを単純な繰り返しに使うと一番読み易いコードにはならない：

[*3] これは新しい Common Lisp の数列マクロを使ってもっとエレガントに書くこともできる．しかしこれらのマクロは Lisp そのものの拡張となっているので，結局は同じことだ．

```
(do ((x a (+ 1 x)))
    ((> x b))
  (print x))
```

代わりにこうするだけでいい：

```
(for (x a b)
  (print x))
```

マクロがこれを可能にする．6 行のコードで（154 ページ参照）`for` 文をこのプログラミング言語に追加できる．そしてそれは初めからあった構文のように機能する．そして後の章で見るように，`for` の実装はマクロでできることの手始めでしかない．

Lisp の拡張で一度に使えるのは関数やマクロ 1 個ずつに限られるわけではない．必要とあればあるプログラミング言語全体を Lisp の上に構築し，それを使ってプログラミングすることもできる．Lisp はコンパイラやインタプリタを書くのに最適のプログラミング言語だが，新しいプログラミング言語を定義する別の方法を提供する．その方がしばしばエレガントだし，労力が少なく済むのは確かだ：それは Lisp を修正して新しいプログラミング言語を定義する方法だ．そうすれば Lisp の機能のうち新しいプログラミング言語でも変更せずに使える部分（例えば算術演算や I/O）はそのまま利用でき，異なっている部分（例えば制御構造）だけ実装すればいい．このように実装されたプログラミング言語を**埋め込み言語**と呼ぶ．

埋め込み言語はボトムアップ・プログラミングの自然な帰結だ．Common Lisp には既に幾つか例がある．一番有名な CLOS については後の章で議論する．しかし自分だけの埋め込み言語を定義することもできる．自分のプログラムに適した埋め込み言語を作ることができるが，それが Lisp とかけ離れたようなものになってもいい．

0.5 なぜ（またはいつ）Lisp か

これらの新たな可能性は魔法の要素たった 1 個から生まれる訳ではない．この点を見れば，Lisp はアーチのようなものだ．楔形の石 (voussoirs) のうち，どれがアーチを支えているのか？これは質問そのものが誤っている：どの石もアーチを支えているのだ．アーチと同様，Lisp は組み合わさった機能の集合体だ．それらの幾つかをここで挙げることができる——動的メモリ割り当てとガーベジ・コレクション，実行時型指定，オブジェクトとしての関数，リストを生成する組み込みパーサ，リストとして表現されたプログラムを受け付けるコンパイラ，インタラクティブな環境等々——しかしどの一つをとっても Lisp の持つ力の理由にはならない．Lisp プログラミングを Lisp プログラミングたらしめているのは，そのコンビネーションだ．

過去 20 年でプログラミングの方法は変化した．インタラクティブな環境，動的リンク，オブジェクト指向プログラミングまで——これらの変化の多くは，Lisp の柔軟性を幾分かでも他のプログラミング言語に与えようとする細切れの試みだった．アーチの喩えは，それらがどれ程成功を収めたかを示唆している．

Lisp と Fortran が現存するプログラミング言語のうち最も古いものだということは，よく知られている．おそらくもっと意味のある事実は，それらはプログラミング言語のデザインの思想のうち対極にあるものを代表しているということだ．Fortran はアセンブリ言語からの進歩として開発された．Lisp はアルゴリズムを表現するプログラミング言語として開発された．そういった異なる意図は，大きく異なるプログラミング言語を生んだ．Fortran はコンパイラ開発者の人生を楽にしてくれる．Lisp はプログラマの人生を楽にしてくれる．それ以来，大抵のプログラミング言語は二つの極の間のどこかに位置するものだった．そして Fortran と Lisp は真ん中に向かって歩み寄ってきた．Fortran は今ではずいぶん Algol に似てきたし，Lisp は若かりし頃の無駄な習慣を幾つか諦めた．

最初の Fortran と Lisp は一種の戦場のようなものを定義した．片方では鬨の声は「効率〜！（実装は辛すぎる）」で，もう片方では鬨の声は「抽象化〜！（商用ソフトではあり得ない）」だ．古代ギリシャの戦争の結果を神々が高みから決めたように，この戦の結果はハードウェアによって決められるようになっている．年を追うごとに情勢は Lisp 側に有利になってきているようだ．今では Lisp に対する議論は，1970 年代初頭にアセンブリ言語プログラマ達が高水準プログラミング言語に対して仕掛けた議論に非常に似てきた．今や問いは**なぜ** *Lisp* **か？**ではなく**いつ** *Lisp* **か？**になっている．

1 関数

Lisp は Lisp プログラムを作り上げるブロックであり，Lisp を作り上げるブロックでもある．大抵のプログラミング言語では，オペレータ + はユーザの定義した関数と全く違ったものになっている．しかし Lisp は関数の適用という，プログラムによって行われる全ての計算を説明する単一のモデルを持っている．Lisp のオペレータ + は 1 個の関数であり，ユーザ自身が定義できる関数と全く同じだ．

事実，特殊オペレータと呼ばれる少数のオペレータを除き，Lisp の中核は Lisp の関数の集合だ．この集合に追加するのをためらう必要があるだろうか？そんなことはない：Lisp がこれをしてくれたら，と思うことがあればそれを自分で書くことができ，その新しい関数は組み込みのものと全く同じように扱われるだろう．

この事実は Lisp プログラマにとって重要な結論をもたらす．これは新しい関数のどれもが，Lisp への追加とも，特定のアプリケーションの一部とも考えられるということだ．熟練 Lisp プログラマの典型的なやり方は，両方を幾らかずつ書き，プログラミング言語とアプリケーションが完璧に噛み合うまで二つの境界を調整することだ．この本はプログラミング言語とアプリケーションを相性よくまとめる方法について書かれたものだ．この目標に向かって行うことはみな関数に依存するのだから，関数から始めるのが自然なやり方だ．

1.1 データとしての関数

Lisp の関数を他から際だたせているのは 2 個の特徴だ．1 個目は，上で述べたように Lisp そのものが関数の集合だということ．このことは Lisp に自分で作った新しいオペレータを追加できるということだ．関数に関して知るべきもう一つの大事なことは，関数は Lisp のオブジェクトだということだ．

Lisp は，他のプログラミング言語で目にするようなデータ型をほとんど提供する．整数も浮動小数点数もあれば，文字列，配列，構造体等もある．しかし Lisp はあるデータ型を提供するが，これは最初は衝撃的かもしれない：関数だ．ほとんど全てのプログラミング言語が何らかの形で関数や手続きを提供する．Lisp がそれらをデータ型として提供するとはいったいどのような意味なのだろう？それは Lisp では関数を使って，他の親しみ深いデータ型（整数等）でやろうと思うことが全てできるということだ：実行時に新しくオブジェクトを作り，変数や構造体の中に保持し，他の関数に引数として渡し，結果として返す，等．

実行時に関数を生成し，それを返す能力は特に便利だ．これは「あるコンピュータで走る自己修正的機械語プログラム」のように眉唾物の長所に思えるかもしれない．しかし実行時に新しい関数を生成することは，Lisp プログラミングの定石テクニックということが分かるだろう．

1.2 関数の定義

大抵，最初に `defun` で関数を定義する方法を習う筈だ．次の式は引数の 2 倍を返す `double` という関数を定義する．

```
> (defun double (x) (* x 2))
DOUBLE
```

Lisp にこれを与えれば，`double` は他の関数内でも，トップレベルでも呼び出せる：

```
> (double 1)
2
```

Lisp コードのファイルは主にそういった `defun` から成り立っていて，C や Pascal のようなプログラミング言語での手続き定義のファイルに似ている．しかし大きな違いが明らかになる．この `defun` は手続き定義ではなく，Lisp の呼び出しなのだ．この区別は `defun` の背後で何が起きているのかを見れば明らかになるだろう．

関数はそれ自身が全くのオブジェクトだ．実際に `defun` が果たす機能は関数を作り，1 個目の引数として与えられた名前でそれを保持することだ．だから `double` を呼び出すのと同様，その機能を実装する関数を得ることができる．これには普通 `#'`（シャープ・クォート）オペレータを使う．このオペレータは名前を実際の関数オブジェクトに対応させるものと理解すればいい．これを `double` の名前の前に置くことで上の定義で作られた実際のオブジェクトが得られる：

```
> #'double
#<Interpreted-Function C66ACE>
```

表示された表現は Lisp 処理系の実装ごとに違うけれど，Common Lisp の関数は基本的オブジェクトで，数や文字列といった親しみ深いオブジェクトと同じ役割を持っている．だからこの関数を引数として渡したり，返り値として返したり，データ構造のなかに保存したり，色々なことができる：

```
> (eq #'double (car (list #'double)))
T
```

関数を作るのには defun も要らない．Lisp のオブジェクトのほとんどと同じように，それを文字通りに参照することができる．整数を参照したいときは，ただ整数そのものを使う．文字列を表現したいときは，ダブルクォートで囲まれた一連の文字を使う．関数を表現するには，λ **式**と呼ばれるものを使う．λ 式はリストで，3 部分から成る：lambda シンボル，仮引数のリスト，0 個以上の式から成る本体だ．次の λ 式は double と等価な関数を示している：

```
(lambda (x) (* x 2))
```

これは 1 個の引数 x を取り，$2x$ を返す関数を規定している．

λ 式は関数名とも考えられる．double が「ミケランジェロ」のような固有名だとすれば，(lambda (x) (* x 2)) は「システィーナ礼拝堂の天井画を描いた男」のような，定義となる説明だ．シャープクォートを λ 式の前に置くことで対応する関数が得られる：

```
> #'(lambda (x) (* x 2))
#<Interpreted-Function C674CE>
```

この関数は double と全く同じ働きを持つが，二つは異なったオブジェクトだ．

関数呼び出しでは関数名が先頭に来て，引数が続く：

```
> (double 3)
6
```

λ 式は関数名でもあるから，関数呼び出しの先頭に来ることもできる：

```
> ((lambda (x) (* x 2)) 3)
6
```

Common Lisp では double という関数と double という変数を同時に持つことができる．

```
> (setq double 2)
2
> (double double)
4
```

名前が関数呼び出しの先頭かシャープクォートの次に来ると関数への参照と見なされ，それ以外では変数名と見なされる．

だから「Common Lisp には変数と関数に異なった**名前空間**がある．」と言える．foo という変数と foo という関数を両方持つことができ，それらは別々である必要もない．この状況はややこしく，コードがかなり格好悪くなってしまうことにつながるが，これは Common Lisp プログラマが付き合っていかなければならないことだ．

必要に応じ，Common Lisp ではシンボルをその表現する値に対応させたり，その表現する関数に対応させる関数が使える．関数 symbol-value はシンボルを引数に取り，対応するスペシャル変数の値を返す：

```
> (symbol-value 'double)
2
```

また symbol-function はグローバル関数について同じことをする．

```
> (symbol-function 'double)
#<Interpreted-Function C66ACE>
```

関数は普通のデータ・オブジェクトなので，変数が値として関数を持てることに注意：

```
> (setq x #'append)
#<Compiled-Function 46B4BE>
> (eq (symbol-value 'x) (symbol-function 'append))
T
```

背後では，defun はその 1 個目の引数の symbol-function を，残りの引数で組み立てられる関数に設定しているのだ．次の 2 個の式は大体同じことをしている：

```
(defun double (x) (* x 2))

(setf (symbol-function 'double)
#'(lambda (x) (* x 2)))
```

だから defun は他のプログラミング言語での手続き定義と同じ役割を持つ．つまり名前をコードの一部に関連づけることだ．しかし背後の仕組みまで同じではない．関数を作るのに defun は必要ではなく，関数は何かのシンボルの値として保存されなくてもいい．他のプログラミング言語の手続き定義に似た defun の背後には，もっと一般的な仕組みが隠れている：関数を作ることと，それをある名前に関連づけることは別々の働きだ．Lisp の関数の概念の一般性全体までは必要ないとき，defun はもっと制限の強いプログラミング言語と同じ位単純に関数定義を行う．

1.3 関数を引数にする

関数がデータ・オブジェクトであるということは，何より関数を他の関数の引数として渡せるということだ．この可能性は Lisp におけるボトムアップ・プログラミングの重要性を支えるものの一つなのだ．

関数がデータ・オブジェクトとなりうるプログラミング言語では，オブジェクトはそれを呼び出す方法も何か提供しなければならない．Lisp ではその関数は apply だ．一般的に apply は 2 個の引数で呼ばれる：関数と，そのための引数のリストだ．以下の 4 個の式はみな同じ働きをする：

```
(+ 1 2)
(apply #'+ '(1 2))
(apply (symbol-function '+) '(1 2))
(apply #'(lambda (x y) (+ x y)) '(1 2))
```

Common Lisp では apply には幾つの引数を渡してもよく，1 番目に渡された関数が適用されるのは，残りの引数を最後のリストにコンシングすることで得られるリストだ．だからこの式

```
(apply #'+ 1 '(2))
```

は上記の 4 個の式と等価だ．引数をリストとして渡すのが不便だと思ったら funcall を使えばいい．これはその点のみが apply と違っている．この式

```
(funcall #'+ 1 2)
```

も上記の式と同じ働きを持つ．

Common Lisp の組み込み関数の多くが関数を引数に取れる．中でも最もよく使われるものの一つが対応付け (mapping) 関数だ．例えば mapcar は 2 個以上の引数（関数と 1 個以上のリスト，その関数の取る引数毎にリストが 1 個必要）を取り，その関数をそれぞれのリストの要素に先頭から適用していく．

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)
> (mapcar #'+
          '(1 2 3)
          '(10 100 1000))
(11 102 1003)
```

Lisp プログラマにはリストの要素それぞれに対して何か操作を行い，結果のリストを返してほしいと思うことがよくある．上の例の 1 番目はそれを行う古典的な例を示している：してほしい操作を行う関数を作り，リストに渡って mapcar を使う．

関数をデータとして扱えることがどれ程便利なことかは既に見てきた．多くのプログラミング言語では，mapcar のようなものに関数を引数として渡せるようになっていたとしても，それはソースファイルのどこかで予め定義された関数でなければならない．ただコードの一部でリストの要素それぞれに 10 を加えたいときでも，plus_ten とかいう名の関数をそのためだけに定義しなければならないだろう．λ 式を使えば関数そのものを直接参照できる．

Common Lisp とそれ以前の方言との大きな違いの一つは，関数を引数に取れる関数が多数組み込まれていることだ．あらゆるところで目にする mapcar の次に多く使われるのは，sort と remove-if だ．前者は多目的の整列関数で，リストと述語を取り，要素のペアをその述語に渡して整列したリストを返す．

```
> (sort '(1 4 2 5 6 7 3) #'<)
(1 2 3 4 5 6 7)
```

sort の動作を覚えるには，重複のないリストを < で整列させ，その結果に < を適用したら真が返る，と覚えておけばよい．

さてもし remove-if が Common Lisp に含まれていなかったら，これは一番最初に書き加えたいユーティリティだろう．これは関数とリストを取り，そのリストの要素のうち関数が偽を返すものを全て返す．

```
> (remove-if #'evenp '(1 2 3 4 5 6 7))
(1 3 5 7)
```

関数を引数に取る関数の例として，ここに remove-if の機能限定版の定義を示す：

```
(defun our-remove-if (fn lst)
  (if (null lst)
      nil
      (if (funcall fn (car lst))
          (our-remove-if fn (cdr lst))
          (cons (car lst) (our-remove-if fn (cdr lst))))))
```

この定義の中で fn にシャープクォートが付いていないことに注意．関数はデータ・オブジェクトなので，変数は関数を値として普通に保持することができる．上でやっていることはそれだ．シャープクォートはシンボルで表される関数（普通 defun 等でグローバルに定義されたもの）を参照するときだけしか使わない．

第 4 章で見るように，関数を引数に取るユーティリティを新しく書くのはボトムアップ・プログラミングの重要な要素だ．Common Lisp にはユーティリティがたくさん組み込まれているので，必要だと思ったものは既に存在しているかもしれない．しかし sort 等の組み込みユーティリティを使うにせよ，自分で書き加えるにせよ，原則に違いはない．関数を連結せずに，関数を引数に渡せばいい訳だ．

1.4 属性としての関数

関数が Lisp オブジェクトであるせいで，新しい事態に対処できるように実行中に拡張できるようなプログラムが書ける．動物の種類を取り，適切に振る舞う関数が書きたいとしよう．大抵のプログラミング言語では，これを実現する方法は case 文だろう．Lisp でも次のようにして実現できる：

```
(defun behave (animal)
  (case animal
    (dog (wag-tail)
      (bark))
    (rat (scurry)
      (squeak))
    (cat (rub-legs)
      (scratch-carpet))))
```

新しい動物を追加したいときにはどうしようか？新しい動物の追加を計画しているなら，動物の振る舞いを以下のように定義する方がいいだろう：

```
(defun behave (animal)
  (funcall (get animal 'behavior)))
```

そして個々の動物の振る舞いは，例えばその名前の属性リストに保存された関数として定義する：

```
(setf (get 'dog 'behavior)
  #'(lambda ()
      (wag-tail)
        (bark)))
```

この方法では，新しい動物を追加するには新しい属性を定義しさえすればいい．関数の書き直しは必要なくなる．

2 番目の方法は柔軟だが，動作が遅いように思われる．事実その通りだ．もし速度が最重要だというのなら属性リストの代わりに構造体を使い，また特に関数をコンパイルしておくだろう．（この方法は第 foo 章で説明する．）構造体とコンパイル済み関数を使えば，柔軟なコードでも case 文を使った方と同等かそれを凌ぐ速度が出る．

関数のこのような使い方はオブジェクト指向プログラミングでのメソッドの概念に対応する．一般的に言って，メソッドとはオブジェクトのプロパティである関数であり，それは正にここで実現したものだ．このモデルに継承を加えればオブジェクト指向プログラミングの要素がみな揃う．第 foo 章で見るように，これは驚く程短いコードで実現できる．

オブジェクト指向プログラミングの最大のセールス・ポイントの一つは，それによってプログラムが拡張可能になることだ．この見込みは Lisp の世界ではそれ程興奮を誘うものではない．ここでは拡張性はいつも当然のものと見なされているからだ．必要な種類の拡張性が余り継承に頼らずに済むものなら，素の Lisp でも十分だろう．

1.5 スコープ

Common Lisp はレキシカルスコープを持つ Lisp だ．Scheme はレキシカルスコープを持つ Lisp 方言のうち最も古いもので，Scheme 以前にはダイナミックスコープは Lisp の特徴的な機能とされていた．レキシカルスコープとダイナミックスコープとの違いは，処理系の実装がどのようにフリー変数を扱うかという点だ．仮引数として現れるか，変数を束縛する機能を持つ `let` や `do` 等のオペレータによるかして，変数として作られたシンボルは，式に束縛されている．束縛されていないシンボルはフリーであると言われる．次の例では，スコープが動作に絡んでくる：

```
(let ((y 7))
  (defun scope-test (x)
    (list x y)))
```

この `defun` 式の中では，`x` が束縛されていて `y` はフリーだ．フリー変数は，あるべき値が明らかでないので興味深い．束縛変数の値に関しては曖昧なことは何もない——`scope-test` が呼ばれたとき，`x` の値はとにかく引数で渡されたものになる．しかし `y` の値はどうあるべきだろう？この問の答えはその Lisp 方言のスコープの扱いによる．

ダイナミックスコープを持つ Lisp では，`scope-test` の実行時のフリー変数の値を見つけるには，それを呼び出した関数の連鎖を遡る必要がある．そして `y` が束縛されている環境が見つかれば，その束縛が `scope-test` で使われているものだ．見つからなかったら，`y` のグローバルな値が使われる．だからダイナミックスコープを持つ Lisp では，`y` の値は呼び出し側の式の中での値になる：

```
> (let ((y 5))
    (scope-test 3))
      (3 5)
```

ダイナミックスコープでは，`scope-test` が定義された時点で `y` が 7 に束縛されていたことには何の意味もない．影響があるのは，`scope-test` が呼ばれたときに `y` の値が 5 だったということだ．

レキシカルスコープを持つ Lisp では，関数呼び出しの連鎖を遡る代わりに，関数が定義された時点での周りの環境を遡って調べる．レキシカルスコープを持つ Lisp では，上の例の `y` は `scope-test` が定義された時点での束縛を持つだろう．また Common Lisp でもそうなる：

```
> (let ((y 5))
    (scope-test 3))
      (3 7)
```

ここで，呼び出し時に `y` を 5 に束縛したことは返り値に何の影響も持たない．

変数をスペシャル宣言することでダイナミックスコープを持たせることができるが，Common Lisp では普通はレキシカルスコープになる．一般的に言って，Lisp コミュニティはダイナミックスコープの廃止をほとんど残念に思っていないようだ．一つには，それが恐ろしく捉えづらいバグを招いたことがある．しかしレキシカルスコープはバグ回避の方法程度のものではない．次章で見るように，それによって幾つかの新しいプログラミング技法が可能になる．

1.6 クロージャ

Common Lisp がレキシカルスコープを持つせいで，フリー変数を中に含む関数を定義したときには，処理系はその関数が定義された時点でのそれらの変数の束縛をコピーして保存しなければならない．関数と変数束縛の一式のそのような組み合わせは**クロージャ**と呼ばれる．クロージャは多方面の応用において便利なものだと分かるだろう．

クロージャは Common Lisp の中に広く浸透しているので，それとは意識することさえなく使うことができる．`mapcar` にフリー変数を含む λ 式をシャープクォートして与える度，ユーザはクロージャを使っているのだ．例えば，数のリストを取り，ある数をそれぞれに加える関数を書きたいとしよう．この関数 `list+`

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

が望みの動作をしてくれる：

```
> (list+ '(1 2 3) 10)
(11 12 13)
```

list+ の中で mapcar に渡される関数をよく見れば，それが実際クロージャだと分かる．n のインスタンスはフリーで，その束縛は周りの環境から来ている．レキシカルスコープの下では，そのような対応付け関数を使う度にクロージャが作られている[*4].

Abelson と Sussman の古典 *Structure and Interpretation of Computer Programs*[†] によって奨められたプログラミング・スタイルでは，クロージャはさらに顕著な役を担っている．クロージャはローカルな状態を持った関数であるとされる．この状態を使う例の最も単純なものは次のような状況だ：

```
(let ((counter 0))
  (defun new-id () (incf counter))
  (defun reset-id () (setq counter 0)))
```

これら 2 個の関数はカウンタとして働く変数を共有している．1 個目はカウンタを 1 増やしてその値を返し，2 個目はカウンタを 0 にリセットする．カウンタをグローバル変数とすることで同じことが実現できるが，この方法ではカウンタは意図しない参照から守られている．ローカルな状態を持つ関数を返せるのは便利でもある．例えば，関数 make-adder

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

は数を取り，「呼ばれると引数にその数を加えるクロージャ」を返す．その足し算関数のインスタンスは幾らでも作ることができる：

```
> (setq add2 (make-adder 2)
        add10 (make-adder 10))
#<Interpreted-Function BF162E>
> (funcall add2 5)
7
> (funcall add10 3)
13
```

make-adder に返されたクロージャでは内部状態が固定されているが，状態を変化させられるクロージャを作ることもできる．

```
(defun make-adderb (n)
  #'(lambda (x &optional change)
      (if change
          (setq n x)
          (+ x n))))
```

上の新ヴァージョンの make-adder が返すクロージャは，1 個の引数で呼ばれたときには古いものと全く同じ動作をする：

```
> (setq addx (make-adderb 1))
#<Interpreted-Function BF1C66>
> (funcall addx 3)
4
```

しかし新しい足し算関数が非 nil の第 2 引数と共に呼ばれると，内部にある n のコピーは第 1 引数として渡された値にリセットされる：

```
> (funcall addx 100 t)
100
> (funcall addx 3)
103
```

同じデータ・オブジェクトを共有するクロージャのグループを返すことさえ可能だ．第 1図には初歩的データベースを作る関数を示した．それは連想リスト (db) を取り，それぞれエントリのクエリ，追加，削除を行う 3 個のクロージャのリストを返す．make-dbms を呼び出す度，連想リストの自分専用のコピーを内部で共有する新しい関数の組が返される．

```
> (setq cities (make-dbms '((boston . us) (paris . france))))
(#<Interpreted-Function 8022E7>
 #<Interpreted-Function 802317>
 #<Interpreted-Function 802347>)
```

```
(defun make-dbms (db)
  (list
    #'(lambda (key)
        (cdr (assoc key db)))
    #'(lambda (key val)
        (push (cons key val) db)
        key)
    #'(lambda (key)
        (setf db (delete key db :key #'car))
        key)))
```

図 1　3 個のクロージャがリストを共有する例.

データベース内の実際の連想リストは，外からは見えない．それどころか，それが連想リストなのかどうかも分からない．しかしそれは cities のコンポーネントである関数を通じて到達できる：

```
> (funcall (car cities) 'boston)
US
> (funcall (second cities) 'london 'england)
LONDON
> (funcall (car cities) 'london)
ENGLAND
```

リストの car を呼ぶのは少し格好悪い．実際のプログラムでは，アクセス関数は構造体のエントリでもいいかもしれない．それらを使うとさらに簡潔になるだろう．データベースは次のような関数

```
(defun lookup (key db)
  (funcall (car db) key))
```

から間接的にアクセスできるようになる．しかし，クロージャの基本的な動作はそういったチューニングとは無関係だ．実際のプログラム使われるクロージャとデータ構造は，make-adder や make-dbms で見たものより手の込んだものだろう．共有されている変数は 1 個だったが，幾つに増やしてもよく，それぞれはどのようなデータ構造にも束縛できる．クロージャは Lisp の長所のうち顕著で明白なものの一つだ．Lisp プログラムの中には，努力すれば力の弱いプログラミング言語に翻訳できるものもあるかもしれない．しかし上のようなクロージャを使うプログラムを翻訳しようとしてみれば，この抽象化構造がどれ程労力の節約になっているか明らかになるだろう．後の章ではクロージャをさらに詳細に扱う．第 5 章では，それらを使って複合的な関数を作る方法を示す．第 6 章では，伝統的データ構造の代わりとしての使用法を見る．

1.7　ローカル関数

λ 式で関数を定義したとき，defun では分からなかった制限に直面する：λ 式で定義した関数には名前がなく，そのためにそれ自身を参照する方法がない．このことは，Common Lisp では再帰関数を定義するのに λ 式が使えないということだ．ある関数をリストの要素全てに適用したいときは，Lisp の慣用法のうち最も親しみ深いものを使う：

```
> (mapcar #'(lambda (x) (+ 2 x))
          '(2 5 7 3))
(4 7 9 5)
```

mapcar の第 1 引数に再帰関数を与えたいときにはどうすればいいのだろうか？その関数が defun で定義されていたら，ただ関数の名前によってそれを参照すればいい：

```
> (mapcar #'copy-tree '((a b) (c d e)))
((A B) (C D E))
```

しかし使う関数が，mapcar の存在する環境から幾つかの束縛を引き継ぐようなクロージャでなくてはいけないとしよう．例に挙げた list+

```
(defun list+ (lst n)
  (mapcar #'(lambda (x) (+ x n))
          lst))
```

[*4] ダイナミックスコープの下でも，mapcar の仮引数がどれも x という名を持たない限り同じ例文が機能するが，その理由は異なる.

では mapcar への第 1 引数 #'(lambda (x) (+ x n)) は list+ 内で定義されていなければいけない．それは n の束縛を捉えなければいけないからだ．そこまではいいが，ローカルな束縛を必要とし，かつ再帰的な関数を mapcar に与えたいとしたらどうだろう？ローカルな束縛が必要だから，別の場所で defun で定義された関数は使えない．また λ 式で再帰関数を定義することはできない．その関数には自分自身を参照する方法がないからだ．このジレンマを解決するため，Common Lisp には labels が用意されている．1 点の重要な留保を除き，labels は関数に対する一種の let として説明できる．labels 式の中の束縛指定部は，それぞれ次のような形になる：

(⟨ 名前 ⟩ ⟨ 仮引数 ⟩ . ⟨ 本体 ⟩)

labels 式の中では，name は次のような関数と等価な関数を参照する：

(lambda ⟨ 仮引数 ⟩ . ⟨ 本体 ⟩)

例を挙げよう：

```
> (labels ((inc (x) (1+ x)))
          (inc 3))
4
```

しかし let と labels には重要な違いがある．let 式の中では，ある変数の値は同じ let で作られた別の変数に依存することはできない．つまり

```
(let ((x 10) (y x))    ;; 誤り
  y)
```

として，新しい y の値が新しい x の値を反映することを期待しても駄目なのだ．それに対し，labels 式の中で定義された関数 f の本体では，そこで定義されたどの関数を参照してもいい．それは f 自身でもよく，おかげで再帰関数の定義が可能になっている．labels を使って list+ に似た関数を書けるが，こちらでは mapcar への第 1 引数が再帰関数になっている：

```
(defun count-instances (obj lsts)
  (labels ((instances-in (lst)
             (if (consp lst)
                 (+ (if (eq (car lst) obj) 1 0)
                    (instances-in (cdr lst)))
                 0)))
    (mapcar #'instances-in lsts)))
```

この関数は 1 個のオブジェクトと 1 個のリストを取り，そのリストの要素内にオブジェクトが何個あったかの回数のリストを返す：

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

1.8 末尾再帰

再帰関数とは自分自身を呼び出す関数だ．そして関数呼び出しの後に行うべき作業が残っていなければ，その呼び出しは**末尾再帰**だ．次の関数は末尾再帰でない.

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

再帰呼び出しから戻った後，結果を 1+ に渡さなければいけないからだ．しかし次の関数は末尾再帰だ.

```
(defun our-find-if (fn lst)
  (if (funcall fn (car lst))
      (car lst)
      (our-find-if fn (cdr lst))))
```

再帰呼び出しの値が即座に返されているからだ．多くの Common Lisp コンパイラが末尾再帰の関数をループに変換できるので，末尾再帰の方がいい．そのようなコンパイラでは，優美な再帰をソースコード内で使っても実行時に関数呼び出しのオーヴァヘッドは必要ない．末尾再帰にすると速度は大抵かなり向上するので，Lisp プログラマはどんどん

関数を末尾再帰にしようとする．末尾再帰でない関数もしばしば末尾再帰に変換できることがある．それには総和変数を使うローカルな関数を埋め込んでやればいい．ここで総和変数とはそれまでに計算された値を保持するパラメータを指す．例えば our-length は次のように変換できる：

```
(defun our-length (lst)
  (labels ((rec (lst acc)
            (if (null lst)
                acc
                (rec (cdr lst) (1+ acc)))))
    (rec lst 0)))
```

ここでそれまでに読み取ったリストの要素の数は，第 2 仮引数 acc に保持されている．再帰がリスト末尾まで達したとき acc の値がリスト全体の長さになっているので，それをそのまま返せばいい．呼び出しのツリーから戻る途中で値を求めていくのでなく，呼び出しのツリーを下るにつれ値を累積していくことで，rec を末尾再帰に変えることができる．

Common Lisp コンパイラの多くが末尾再帰の最適化を行うことができるが，それら全てが始めからそうする設定になっている訳ではない．だから自分の関数を末尾再帰に書き換えたら，ファイルの先頭に

```
(proclaim '(optimize speed))
```

と付け足し，努力の成果をコンパイラが確かに活用できるようにするといいかもしれない[*5]．

末尾再帰形式と型宣言を与えられると，現在の Common Lisp コンパイラは C と同等か，C より速いコードを生成できる．Richard Gabriel[†] による下の例の関数は，1 から n までの整数の和を返す：

```
(defun triangle (n)
  (labels ((tri (c n)
                (declare (type fixnum n c))
                (if (zerop n)
                    c
                    (tri (the fixnum (+ n c))
                         (the fixnum (- n 1))))))
    (tri 0 n)))
```

高速な Common Lisp コードはこういう形になる．始めは関数をこのように書くのは不自然に思えるかもしれない．始めは関数をとにかく一番自然に思える形で書き，その後必要があれば等価な末尾再帰形式に書き換えるのがいい．

1.9 コンパイル

Lisp の関数は 1 個 1 個でもファイル単位でもコンパイルできる．トップレベルに defun 式を打ち込むだけだと

```
> (defun foo (x) (1+ x))
FOO
```

多くの処理系が作るのは（コンパイルされていない）インタプリタに読み込まれた関数だ．ある関数がコンパイルされているかどうかは，それを compiled-function-p に渡せば分かる：

```
> (compiled-function-p #'foo)
NIL
```

foo をコンパイルするにはその名前を compile に与える：

```
> (compile 'foo)
FOO
```

これは foo の定義をコンパイルし，評価された関数をコンパイル済みのものに置き換える関数だ[†]．

```
> (compiled-function-p #'foo)
T
```

コンパイル済みの関数と評価された関数は共に Lisp のオブジェクトで，違うのは compiled-function-p を適用したときの結果だけだ．関数を文字通りに表記したものもコンパイルできる：compile は第 1 引数がコンパイルする関数の名前であることを期待するが，第 1 引数に nil を与えると，第 2 引数で与えられた λ 式をコンパイルする．

[*5] 宣言 (optimize speed) は (optimize (speed 3)) の省略形の筈だが，ある Common Lisp 処理系は前者では末尾再帰の最適化をするのに，後者ではしないことが分かっている．

```
> (compile nil '(lambda (x) (+ x 2)))
#<Compiled-Function BF55BE>
```

名前と関数を両方与えると，compile は defun をコンパイルするような形になる：

```
> (progn (compile 'bar '(lambda (x) (* x 3)))
(compiled-function-p #'bar))
T
```

プログラミング言語の中にコンパイルの命令があるということは，プログラムが実行中に新しい関数を作ってコンパイルできるということだ．しかし compile を陽に呼ぶのは，eval を呼ぶのに匹敵する程過激な方法で，同じくらい疑ってかかるべきだ[*6]．第 2.1 節で実行時に関数を作るのは頻繁に使われるプログラミング技法だと言ったが，それは make-adder に作られた新しいクロージャ等を指しており，生のリストに対して compile を呼んで作った関数のことではない．compile を呼ぶのは頻繁に使われる技法ではない——その方法は物凄く希だ．だから不必要に使うのは避けること．Lisp の上に別のプログラミング言語を実装するのでもない限り（そしてその場合でさえ大抵は），必要なことはマクロを使えば可能だろう．

2 種類の関数は compile に引数として与えることができない．CLtL2 によると（vox ページ），「空でないレキシカル環境においてインタプリタ的に定義された関数」だ．つまりトップレベルで let を使って foo を定義すると

```
> (let ((y 2))
(defun foo (x) (+ x y)))
```

(compile 'foo) には意味がない[*7]．また既にコンパイル済みの関数で compile を呼ぶこともできない．この状況に関し CLtL2 は不吉な示唆をしている．曰く，「そのときの結果は…不定である．」

Lisp コードをコンパイルする方法は，普通，関数を個別に compile でコンパイルするのでなく，ファイル全体を compile-file でコンパイルすることだ．この関数はファイル名を取り，ソースファイルのコンパイル済みのものを造り出す．ファイル名本体は同じで拡張子が違う，というのが典型的なものだ．コンパイル済みファイルが読み込まれると，compiled-function-p はそのファイルで定義された全ての関数に真を返す筈だ．後の章では，コンパイルの別の効果を使っている：ある関数が別の関数内で作られ，外側の関数がコンパイルされると，内側の関数もコンパイルされる．CLtL2 ではこうなるとは明記していないようだが，ちゃんとした処理系ではこうなる筈と考えていい．

内部の関数のコンパイルの問題は，関数を返す関数を使うと明らかになる．make-adder（foo ページ）がコンパイルされると，それが返す関数もコンパイルされている：

```
> (compile 'make-adder)
MAKE-ADDER
> (compiled-function-p (make-adder 2))
T
```

後の章で見るように，この事実は埋め込み言語の実装において大変重要だ．新しいプログラミング言語がコード変換によって実装されているとき，変換を司るコードがコンパイルされると，それはコンパイル済みコードを生成する．そして新しいプログラミング言語のためのコンパイラでその問題が明らかになる．（簡単な例を baz ページで説明した．）

かなり小さい関数を定義したときは，それをインラインでコンパイルしてほしいことがある．そうしないと呼び出し機構に関数自体よりも多くの手間がかかってしまう．関数を定義して

```
(defun 50th (lst) (nth 49 lst))
```

インライン宣言をすると

```
(proclaim '(inline 50th))
```

コンパイル済み関数内での 50th への参照には実際の関数呼び出しが要らなくなる．このとき 50th を呼び出す関数を定義してコンパイルすると

```
(defun foo (lst)
  (+ (50th lst) 1))
```

50th のコードはその中にそのまま組み入れられる．すると最初の所で次のように書いたかのような結果になる：

```
(defun foo (lst)
  (+ (nth 49 lst) 1))
```

[*6] 陽に eval を呼ぶのがなぜいけないかは，foo ページで説明する．

[*7] このコードをファイルに保存し，ファイルをコンパイルするのは問題ない．制限は実装上の理由から「インタプリタ的に読み込まれたコード」という点に対するもので，異なるレキシカル環境で関数を定義することに何も問題はない．

```
(defun bad-reverse (lst)
  (let* ((len (length lst))
         (ilimit (truncate (/ len 2))))
    (do ((i 0 (1+ i))
         (j (1- len) (1- j)))
        ((>= i ilimit))
      (rotatef (nth i lst) (nth j lst)))))
```

図 2　リストの要素の順を逆転させる関数.

インライン関数の欠点は，50th を定義し直したときには foo を再コンパイルしなくてはいけないことだ．そうしないと foo は 50th の古い定義を反映したままになってしまう．インライン関数に対する制限は，マクロに対するものと基本的に同じだ（第 foo 章を参照）†.

1.10　リストから作られる関数

Lisp の初期の方言では，関数がリストとして表現されているものがあった．このことは Lisp プログラムに自分自身で Lisp プログラムを書き，実行するという注目すべき能力をもたらした．Common Lisp では，関数はもうリストから作られてはいない——よくできた処理系は関数をネイティブな機械語にコンパイルする．しかしプログラムを書くプログラムをユーザが書くことは依然として可能だ．それはリストがコンパイラに対する入力形式だからだ.

Lisp プログラムが Lisp プログラムを書けるということは，どんなに強調しても強調し過ぎにはならない．特に，この事実はしばしば軽く見られがちだからだ．熟練 Lisp ユーザでさえ，Lisp のこの機能から得られる利点を理解していることは少ない．例えば Lisp のマクロがあれ程強力な理由はこれだ．この本で説明されている技法のほとんどが，Lisp の式を操作できるプログラムを書ける能力によるものだ.

2　関数的プログラミング

前の章では，Lisp と Lisp のプログラムが共に一つの材料から成り立っている様子を説明した．どの建材でも同様だが，材料の性質は建物の性質とそれを建てる方法に影響を与える.

この章では，Lisp の世界で広く使われている建築技法の種類を説明する．これらの方法に精通すれば，もっと思い切った種類のプログラムを書こうという気になる．次の章では，Lisp で可能になる，特に重要な種類のプログラムを説明する：古い「計画－実装」方式で開発されるのでなく，進化していくプログラムだ.

2.1　関数的デザイン

オブジェクトの性格はそれを作る要素に影響を受ける．例えば木造の建物は石造りの建物とは違った見掛けになる．木や石そのものを見る機会が全くない人でも，建物の全体的な形からそれが何でできているかは分かるだろう．Lisp の関数の性格は Lisp のプログラムの構造に対して似たような影響を及ぼしている.

関数的プログラミングとは，副作用ではなく，値を返すことで動作するプログラムを書くことだ．副作用とはオブジェクトの破壊的な変更（rplaca の使用等）や変数への代入（setq の使用等）を含む．副作用を使う数が少なく，その影響範囲もローカルなものであれば，プログラムの読み取り，テスト，デバッグは簡単になる．Lisp のプログラムが必ずこの方法で書かれてきた訳ではないが，時を追って Lisp と関数的プログラミングは次第に分かち難いものになってきた.

関数的プログラミングが別のプログラミング言語でのプログラミングとどれ程違うのかは，例を見れば分かるだろう．何かの理由であるリストが逆順になったものの要素が欲しいとしよう．そのときリストを逆順にする関数を書くのではなく，リストを引数とし，同じ要素で逆順のリストを返す関数を書く.

第 2図はリストを逆順にする関数だ．これはリストを配列として扱い，要素の場所を使って逆転させる．返り値に意味はない：

```
> (setq lst '(a b c))
```

```
(defun good-reverse (lst)
  (labels ((rev (lst acc)
             (if (null lst)
                 acc
                 (rev (cdr lst) (cons (car lst) acc)))))
    (rev lst nil)))
```

図 3　要素の順が逆転したリストを返す関数.

```
(A B C)
> (bad-reverse lst)
NIL
> lst
(C B A)
```

名前が示すように，bad-reverse は Lisp でのよいスタイルとは程遠い．それどころか伝染病のような醜さを持っている：つまりこれが副作用によって働き，また呼び出し側に対しても関数的プログラミングの理想を妨げるということだ．悪者役にされてしまったが bad-reverse には長所が 1 個ある．これは 2 個の値を入れ替えるときの Common Lisp での慣用法を示している．rotatef マクロは任意の数の汎変数（generalized variable，setf の第 1 引数として与えることができる式）の値を逆に並び替える．そして引数が 2 個だけの時にはそれらを交換する．対照的に，第 2図は逆順のリストを返す関数を示している．good-reverse では返り値として逆順のリストが得られ，元のリストはそのままだ．

```
> (setq lst '(a b c))
(A B C)
> (good-reverse lst)
(C B A)
> lst
(A B C)
```

かつては人の容貌を見ればその人の性格が分かると思われていた．これが人間に対して正しかろうとそうでなかろうと，Lisp プログラムに対しては一般的に正しい．関数的プログラムは命令的プログラムと異なった容貌を持っている．関数的プログラムの構造は全て式内部の引数の構成によって決まり，また引数がインデントされているので，そのインデント方法は多岐に渡る．関数的プログラムのコードはページ内を流れていくように見える[*8]．命令的プログラムのコードは固定的でブロックに別れていて，ちょうど Basic に似ている．

眺めるだけでも，bad-reverse と good-reverse のどちらがいいかが伝わってくる．そして短いだけでなく good-reverse は効率的でもある：$O(n^2)$ ではなく $O(n)$ なのだ．

Common Lisp には組み込みで reverse があるので，それを書く手間は要らない．この関数はしばしば関数的プログラミングに対する表面的な誤解をもたらすので，軽く見ておく価値はある．good-reverse と同様，組み込みの reverse も値を返すことで働く——その引数は手付かずだ．しかし Lisp の学習途中の人は，それが bad-reverse のように副作用によって働いていると思うかもしれない．プログラムのどこかでリスト lst を逆順に変えたいとき

```
(reverse lst)
```

と書いて，どうして呼び出しの効果が出ないのだろう，と思うかもしれない．実際は，そういった関数の副作用が欲しいときには呼び出しコードの中で副作用が起きるようにしてやらなければいけない．つまり，代わりに

```
(setq lst (reverse lst))
```

と書く必要があるということだ．reverse 等のオペレータは，副作用でなく返り値のために呼ばれるよう意図されている．自分のプログラムもこのスタイルで書く価値がある——Lisp の生まれ持った長所のためばかりでなく，そうしないと Lisp に逆らってプログラムを書くことになるからだ．

bad-reverse と good-reverse との比較で無視していた点の一つは，bad-reverse はコンシングをしない点だ．新しい構造を作るのでなく，元のリストに対して動作している．これは危険となりうる——そのリストはプログラムのどこかで必要になっているかもしれない．しかし効率のためにはそれが必要なときもある．そんな場合のために，Common Lisp には $O(n)$ オーダーの破壊的な逆転関数 nreverse がある[†]．

[*8] 特徴的な例については bump ページを参照.

破壊的関数とは渡された引数に変更を及ぼせる関数だ．しかし破壊的関数さえも普通は返り値のために使われる：nreverse は引数として与えられたリストを使い回すと思わなければいけないが，それがリストを逆順に変えてくれると思ってはいけない．これまでのものと同様，逆順になったリストは返り値によって得られる．関数の途中に

```
(nreverse lst)
```

と書いても，それ以後 lst が逆順になっていると思ってはいけない．大抵の処理系では次のようなことが起きる：

```
> (setq lst '(a b c))
(A B C)
> (nreverse lst)
(C B A)
> lst
(A)
```

lst を逆順に変えるためには，普通の reverse でやるのと同じように lst に返り値を代入しなければいけない．

ある関数が破壊的だと書かれていても，その関数が副作用のために呼ばれる筈の関数だということではない．危ない点は，幾つかの破壊的関数がそういった印象を与えることだ．例えば

```
(nconc x y)
```

と

```
(setq x (nconc x y))
```

とはほぼ同じ効果を持つ．前者の慣用法によるコードを書くと，正しく働くように思えることもあるかもしれない．しかし x が nil のときには思った通りの動作をしないだろう．

Lisp オペレータのうち，副作用のために呼ばれるよう意図されているものはほんの僅かだ．一般的に言って，組み込みオペレータは返り値のために呼ばれるよう意図されている．sort，remove や substitute 等の名前に惑わされてはいけない．副作用が必要なら，返り値を setq で代入すること．

まさにこのルールが，副作用を不可避なものにしている．関数的プログラミングを理想とするというのは，プログラムが決して副作用を使ってはいけないということではない．ただ必要以上に使うべきでないということだ．

この習慣を育てるには時間がかかるかもしれない．一つの方法は，以下のオペレータは税金がかかっているつもりで扱うことだ：

```
set setq setf psetf psetq incf decf push pop pushnew
rplaca rplacd rotatef shiftf remf remprop remhash
```

あと let*もそうだ．この中に命令的プログラムが潜んでいることがしばしばある．これらのオペレータに税金がかかっているつもりになるのは，よい Lisp のプログラミング・スタイルへ向かう手助けとして勧めただけで，それがよいスタイルの基準なのではない．しかし，それだけでもずいぶん進歩できるだろう．

他のプログラミング言語では，副作用を使う理由で最も大きいものは，多値を返す関数が必要になることだ．関数が 1 個の値しか返せなければ，他の値はパラメータに変更を加えることで「返す」しかない．幸運なことに，Common Lisp ではその必要はない．どの関数も多値を返せるからだ．

例えば組み込み関数 truncate は 2 個の値（切り捨て結果の整数と切り捨てられた部分）を返す．典型的な処理系では，トップレベルで truncate を呼ぶと両方を表示する：

```
> (truncate 26.21875)
26
0.21875
```

呼び出し側のコードが値を 1 個しか取らないときは，1 番目が使われる：

```
> (= (truncate 26.21875) 26)
T
```

multiple-value-bind を使うことで，呼び出し側コードは両方の返り値を捉えることができる．このオペレータは変数のリスト，関数呼び出し，コード本体を引数に取る．本体が評価されるときには，関数呼び出しからの返り値は各々が変数に代入されている：

```
> (multiple-value-bind (int frac) (truncate 26.21875)
    (list int frac))
(26 0.21875)
```

最後に，多値を返すには values オペレータを使う：

```
> (defun powers (x)
    (values x (sqrt x) (expt x 2)))
POWERS
> (multiple-value-bind (base root square) (powers 4)
    (list base root square))
(4 2.0 16)
```

全体的に見て関数的プログラミングはいい方法だ．それは Lisp で使うには特にいい．と言うのも Lisp は関数的プログラミングを支援するために進化してきたからだ．reverse や nreverse 等の組み込みオペレータはこのように使われることを意図している．values や multiple-value-bind 等のオペレータは，関数的プログラミングを簡単にするために特に用意されたものだ．

2.2 命令的プログラミングの裏返し

関数的プログラミングの狙いは，もっと普通のアプローチ，命令的プログラミングの狙いと対比させるとはっきり見えてくるかもしれない．関数的プログラムは，それが欲しがるものを求める．命令的プログラムは，何をすべきかの指示を求める．関数的プログラムの「a と，x の第 1 要素の 2 乗から成るリストを返せ.」：

```
(defun fun (x)
  (list 'a (expt (car x) 2)))
```

命令的プログラミングではこうだ.「x の第 1 要素を求め，それを 2 乗せよ．そして a と，先程 2 乗した値から成るリストを返せ.」：

```
(defun imp (x)
  (let (y sqr)
    (setq y (car x))
    (setq sqr (expt y 2))
    (list 'a sqr)))
```

このプログラムを両方の方法で書ける Lisp ユーザは幸運だ．命令的プログラミングだけに適したプログラミング言語もある——Basic と，ほとんどのマシン語だ．実際，imp の定義はほとんどの Lisp コンパイラが fun に対して生成するマシン語と構成が似ている．

コンパイラが代わりにやってくれるのに，どうしてそんなコードを書くのだろう？多くのプログラマには，この疑問は浮かびすらしない．プログラミング言語はそのパターンを私達の考えに焼き付ける：命令的プログラミングに慣れてしまった人は，プログラムを命令的プログラミング用語で考えるようになってしまった結果，実際に関数的プログラミングより命令的プログラミングの方が易しく思えるのかもしれない．この精神的習慣は，乗り越える価値のあるものだ——そうさせてくれるプログラミング言語を持っているなら．

他のプログラミング言語の卒業生にとっては，Lisp を使い始めるのは始めてスケートリンクの上に立つのと似ているかもしれない．氷の上で動き回るのは陸上で動き回るのより実際はずっと簡単だ——スケート靴を使えば．そうするまではこのスポーツが何だというのかと一人ぽつんと悩むことになるだろう．

スケートと氷の関係は，関数的プログラミングと Lisp との関係と同じだ．それらを組み合わせれば，優雅に，しかも苦労せずにあちこち回れるようになる．しかし別の種類の移動方法に慣れてしまっていると，始めは何やら感じが掴めないだろう．第二言語として Lisp を学ぶときにはっきり理解しにくい点の一つは，関数的プログラミングのスタイルを学ぶことだろう．

幸運なことに，命令的プログラムを関数的プログラムに変換するうまい方法がある．この方法は完成したコードに適用することから始めるといい．すぐに勘が働くようになり，コードを書きながら変換できるようになるだろう．そうすればあっという間に，始めから関数的プログラミングの用語でプログラムを考えるようになるだろう．

その方法は，命令的プログラムは関数的プログラムを裏返しにしたものと思うことだ．関数的プログラムが命令的プログラムの中に隠れているのを見つけるには，ただ裏返しにすればいい．この方法を imp で試してみよう．

最初に気付くのは，先頭の let 内で y と sqr を作っていることだ．これはよくないことが続く前触れだ．実行時の eval の呼び出しと同様，初期化されていない変数は滅多に使われないので，一般的に言ってプログラム内で変なことをした現れと見なしていい．そういった変数はしばしばプログラムを留める画鋲のようなもので，プログラムが自然な形になろうとするのを妨げている．

しかしここではそれは暫く無視し，関数の終わりまでまっすぐ進んでみる．命令的プログラムで最後に起きることは，関数的プログラムでは一番最初に起きる．だから最初の一歩は最後に行われる list の呼び出しを掴み，プログラムの残りをその中に詰め込むことだ——シャツを裏返しにするように．そしてシャツを袖からカフスにかけて次第に裏返していくように，同じ変換方法を繰り返し適用し続けていく．

後ろから見ていって，sqr を (expt y 2) で置き換え

```
(list 'a (expt y 2)))
```

が得られる．次に y を (car x) で置き換えると

```
(list 'a (expt (car x) 2))
```

となる．こうしてコードの残りは最後の式の中に詰め込まれたので，それを捨て去ることができるようになった．ここに至るまでに変数 y と sqr の必要性はなくなったので，let も削ることができる．

最終結果は始めのものより短く，理解も容易だ．元のコードでは最後に (list 'a sqr) という式が現れたが，これでは sqr の値がどこから来るのかすぐには明らかにならなかった．今は返り値の元になるものは道路地図のように目の前に広がっている．

この章での例は短いものだったが，この方法のスケールは次第に拡大する．実際，大規模な関数に適用すればそれだけ価値が出てくる．副作用を起こす関数でさえ，副作用を起こさないパーツへときれいに分解できる．

2.3 関数的インタフェイス

副作用が起こす問題には大きいものと小さいものがある．例えば次の関数では nconc を呼んでいるが，参照透明性は保たれている[*9]．

```
(defun qualify (expr)
  (nconc (copy-list expr) (list 'maybe)))
```

これを任意の引数で呼んでも，引数が同じなら必ず同じ（equal を満たす）値を返す．呼び出し側から見れば qualify は純粋な関数的コードであるも同然だ．そのことは，実際に引数を書き換えてしまう bad-reverse（kaboom ページ）には当てはまらない．

全ての副作用を一括りに悪者にする代わりに，問題のある場合とない場合を区別する方法があれば便利だろう．大雑把に言うと，関数が，他の誰のものでもないオブジェクトを書き換えるのは無害だ．例えば qualify 内の nconc は無害だと言える．それは第 1 引数として与えられたリストは新たにコンシングされるからだ．他のどの関数もそれに関わることはない．

一般的な場合では，オブジェクトをどの関数が支配しているかではなく，どの関数呼び出しが支配しているかについて考えなければいけない．次の例では変数 x を支配しているものは他に何もないが，呼び出しの作用は，次の呼び出し時に明らかになる．

```
(let ((x 0))
  (defun total (y)
    (incf x y)))
```

だからルールはこうあるべきだ：任意の関数呼び出しが，自分だけが支配するオブジェクトを安全に書き換えられるようにする．

何が引数と返り値を支配するのだろう？関数呼び出しは返り値として受け取るオブジェクトを支配するが，引数として渡されるオブジェクトは支配しない，というのが Lisp の慣習のようだ．引数に変更を加える関数は「破壊的」との呼び名で区別されるが，返ってくるオブジェクトに変更を加える関数には特に呼び名がない．

例えば次の関数は慣習に従っている：

```
(defun ok (x)
  (nconc (list 'a x) (list 'c)))
```

これは慣習に従わない nconc を呼んでいるが，nconc が切り張りするリストは新しく作られたもので，ok に引数として渡されるリストとは違う．だから ok そのものに問題はない．

しかしこれが僅かに違って

[*9] 参照透明性の定義については pooh ページを参照.

```
(defun not-ok (x)
  (nconc (list 'a) x (list 'c)))
```

と書かれていたら，nconc の呼び出しは not-ok に渡された引数に変更を加えるだろう．多くの Lisp プログラムはこの慣習を（少なくともローカルな範囲では）破っている．しかし ok で見たように，ローカルな範囲で慣習を破っても，関数呼び出しが慣習を破ることにはならない．そして上の条件を満たす関数は，純粋に関数的なコードの長所を多く保つだろう．

純粋に関数的なものと全く区別の付かないプログラムを書くには，もう一つ条件を付け加えなければいけない．関数は，このルールに従わない他のコードとオブジェクトを共有してはいけない．例えば次の関数は副作用を持たない．

```
(defun anything (x)
  (+ x *anything*))
```

この関数の返り値はグローバル変数 *anything* に依存する．だから他の関数がこの変数の値を変えることがあれば，anything の返す値はどうとでもなりうる．

呼び出しても自分だけが関わるものにしか変更を加えないように書かれたコードは，純粋に関数的なコードに引けを取らない．上記の条件を全て満たす関数は，少なくとも外部に関数的インタフェイスを提供している：それを同じ引数で 2 回呼び出せば，同じ結果が得られる筈だ．そして，次の章で見るように，これがボトムアップ・プログラミングに必須のポイントだ．

破壊的な操作の問題点の一つは，グローバル変数と同様，それがプログラムのローカル性を損ないうる点だ．関数的なコードを書いているときは，焦点を絞ることができる：書いている関数から呼び出されたり，書いている関数を呼び出す関数だけを考えればいい．何かに破壊的な変化を加えようとすると，この長所は失われてしまう．その関数はどこで使ってもいい筈だったのに．

上記の条件は，純粋な関数的コードで得られる完全なローカル性を保証する訳ではないが，状況は幾分なりとも改善される．例えば次のように f が g を呼び出すものとしよう：

```
(defun f (x)
  (let ((val (g x)))
  ; ここで val を書き換えていいものか？
  ))
```

f が val に対して nconc で何かを連結するのは安全だろうか？ g が identity のときはそうではない：そのときは，元々は f そのものに引数として渡されたものを書き換えてしまうだろう．

だから Lisp の慣習を確かに守っているプログラムでも，f 内で何かを書き換えたいならその向こうを見通さなければいけないだろう．と言っても，それ程遠くを見通す必要はない：プログラム全体について心配しなくても，f の下の部分ツリー（subtree）だけを考慮すればいい．

上の慣習から導かれるのは，関数は安全に書き換えられないものを返してはいけないということだ．だから返り値にクォート付きオブジェクトを含むような関数を書くのは避けるべきだ．ここで exclaim を，その返り値がクォート付きリストを含むように定義し

```
(defun exclaim (expression)
  (append expression '(oh my)))
```

呼び出し後に返り値に破壊的な操作をすると

```
> (exclaim '(lions and tigers and bears))
(LIONS AND TIGERS AND BEARS OH MY)
> (nconc * '(goodness))
(LIONS AND TIGERS AND BEARS OH MY GOODNESS)
```

関数内のリストに変更を及ぼすことになりうる：

```
> (exclaim '(fixnums and bignums and floats))
(FIXNUMS AND BIGNUMS AND FLOATS OH MY GOODNESS)
```

exclaim をそういった問題に対して堅固なものにするには，こう書くべきだ：

```
(defun exclaim (expression)
  (append expression (list 'oh 'my)))
```

関数がクォート付きリストを返すべきでないというルールには，大きな例外が 1 個ある：マクロを生成する関数だ．マクロ展開関数は，生成するマクロ展開がコンパイラにそのまま受け取られるなら，その中にクォート付きリストを安全に含むことができる．

それ以外では，一般的に言ってクォート付きリストは疑っていい．それらの多くは大抵 `in`（boom ページ）等のマクロで実現すべきものだ．

2.4 インタラクティブ・プログラミング

前の章ではプログラムの構成のよい方法として関数的なプログラミング・スタイルを提示した．しかし関数的プログラミング・スタイルはそれだけのものではない．Lisp プログラマは美的な理由だけで関数的プログラミング・スタイルを取っているのではない．作業が簡単になるからそうするのだ．Lisp の動的な環境では，関数的プログラムは並外れた速さで書くことができ，同時に並外れた信頼性が高い．

Lisp ではプログラムのデバッグは比較的簡単だ．たくさんの情報が実行時に利用可能で，エラーの原因をトレースするのに役立つ．しかしもっと重要なのは，プログラムをテストするときの簡単さだ．プログラムをコンパイルして全体を一度にテストする必要がない．関数はトップレベル・ループから個々に呼び出してテストできる．

テストを随時行うことには大変価値があるので，Lisp のプログラミング・スタイルはその長所を取り入れて進化してきた．関数的プログラミング・スタイルで書かれたプログラムは関数 1 個毎に理解できるが，読む方の側からはこれが大きな長所だ．しかし関数的プログラミング・スタイルは，テストを随時行うことにも完全に適応している：このスタイルで書かれたプログラムは関数 1 個毎にテストできる．ある関数が外部を参照もせず，変更もしないなら，どんなバグもすぐに明らかになる．そういった関数は返り値を通してのみ外部に影響を及ぼすのだ．返り値が予想通りである限り，それを返したコードは信頼できる．

実際，熟練 Lisp プログラマはテストしやすいようにプログラムをデザインする：

1. 彼らは副作用を使う部分を幾つかの関数に隔離し，プログラムの大部分は純粋に関数的なプログラミング・スタイルで書けるようにする．
2. 関数が副作用を使うのを避けられないなら，彼らは少なくともそこに関数的インタフェイスを盛り込もうとする．
3. 彼らは関数に明確な目的を一つだけ与える．

関数を書くと，それを代表的な状況のどれかでテストし，済んだら次の状況に移る．煉瓦がそれぞれ予想通りの仕事をすれば，壁はしっかり立つだろう．

Lisp では，壁自体もうまくデザインすることができる．1 分間の転送の遅れがある中で，遠く離れた誰かと会話するのを想像して欲しい．次にとなりの部屋の誰かと会話するのを想像して欲しい．同じ会話が速く済むだけではなく，会話の種類が変わるだろう．Lisp では，ソフトウェア開発は面と向かって話をするようなものだ．コードを書くにつれてテストができる．そして簡単に方針転換できることは，それが会話に及ぼすのと同じくらい劇的な影響をソフトウェア開発に及ぼす．そのとき，同じプログラムを速く書いているだけではない．別の種類のプログラムを書いているのだ．

それはどうやって？テストが素早く済めば，もっと頻繁にテストができる．Lisp では（他のどのプログラミング言語でもそうだが），ソフトウェア開発はコード書きとテストのサイクルから成る．しかし Lisp ではサイクルがとても短い：関数 1 個毎，いや関数の部分毎でもいい．あらゆる部分を書くにつれてテストすれば，エラーが起きたときにどこを見ればいいか分かるだろう：最後に書いた部分だ．単純に聞こえるが，この原則はボトムアップ・プログラミングを堅固なものにしている要因の大きな割合を占めている．それは更なる自信をもたらし， Lisp プログラマは（少なくともある一時）古い「計画－実装」スタイルのソフトウェア開発からフリーになれるのだ．

第 1.1 節では，ボトムアップ・デザインが進化するプロセスであることを強調した．プログラミング言語を構築しつつ，それでプログラムを書くことができる．このアプローチが有効なのは下部のコードを信頼できるときの話だ．その層を本当にプログラミング言語として使いたいなら，遭遇したバグはプログラミング言語そのものではなく自分のアプリケーション内にある，と（他のプログラミング言語を使ったときと同じように）確信できるようでなくてはいけない．

だから新しく作った抽象的構造はこの重い責任を背負わなければならず，しかもそれらは必要に応じて作られるようでなければいけない．結構な話だ．Lisp では両方が実現できる．プログラムを関数的スタイルで書き，それを随時テス

トしていけば，即座に目的を達せるような柔軟性と，普通なら注意深く立てた計画と結びつけて考えられるような信頼性が得られる．

3 ユーティリティ関数

Common Lisp のオペレータは 3 種類に分かれる：関数にマクロ（ユーザが作れるもの）と，特殊オペレータ（ユーザには作れない）だ．この章では，Lisp を新しい関数で拡張するテクニックを説明する．しかしここで言う「テクニック」は普通の意味のものではない．そういった関数について知るべき重要な点は，それらをどうやって書くかということではなく，それらがどこから来たのかということだ．Lisp の拡張には，他の関数を書くときと大体同じテクニックが使われることになる．そういった拡張を書くとき難しいのは，どうやって書くかを決めることではなく，何を書くかを決めることだ．

3.1 ユーティリティの誕生

一番単純な場合では，誰がユーザの Lisp をデザインしようと，ボトムアップ・プログラミングとは後から付いた名前に過ぎない．また同時に，ユーザはプログラムを書きながら，プログラムを書き易くする新しいオペレータを Lisp に追加していく．これらの新しく作られたオペレータはユーティリティと呼ばれる．

「ユーティリティ」という言葉に正確な定義などない．ある短いコードがユーティリティと呼ばれるのは，それが短すぎて独立のアプリケーションとは見なせず，しかも用途が一般的過ぎてあるプログラムの一部とも見なせない場合だ．例えば，データベース・プログラムはユーティリティにはならないが，リストに単一の操作を行う関数はユーティリティになりうる．大部分のユーティリティは Lisp が既に備えている関数やマクロに似ている．実際，Common Lisp の組み込みオペレータの多くはユーティリティとして生まれたものだ．関数 `remove-if-not` はリストの要素のうちある述語を満たすものを全て集めるが，Common Lisp の一部になるまでの何年もの間，個々のプログラマによって使われていた．

ユーティリティの書き方は，書くときのテクニックとするよりも心得とした方が上手く説明できる．ボトムアップ・プログラミングとは，プログラムとプログラミング言語を同時に書いていくことだ．これを上手にこなすには，プログラムにどのオペレータが欠けているかを鋭く感じる感覚を鍛えないといけない．プログラムを見て，「ああ，あなたが本当に言いたいのはこれでしょう.」と言えるようにならなくてはいけない．

例えば，ここで `nicknames` が名前を引数に取り，それから派生しうる愛称の全てをリストにまとめる関数だとしよう．この関数が与えられたとき，名前のリストから得られる愛称を全て集めるにはどうすればいいだろう？ Lisp を学習中の誰かが次のようなコードを書くかもしれない：

```
(defun all-nicknames (names)
  (if (null names)
      nil
      (nconc (nicknames (car names))
        (all-nicknames (cdr names)))))
```

経験を積んだ Lisp プログラマは，こんな関数を見て「ああ，本当に欲しいのは `mapcan` だろ.」と言う．それならある人々の愛称を全て探す新しい関数を定義して呼び出さなくても，1 個の式で済む：

```
(mapcan #'nicknames people)
```

`all-nicknames` は車輪の再発明だ．だが，それがまずいのはその点だけではない：汎用オペレータで可能なことを特定の関数に埋もれさせてしまっている．

この場合オペレータ `mapcan` は既に存在していた．`mapcan` を知っていた人は `all-nicknames` を見ると少し変な気がしただろう．ボトムアップ・プログラミングに上達するには，使われていないオペレータがまだ書かれていないものだったとき，同じような不快感を感じられるようにならなくてはいけない．「本当に欲しいのは `x` だろ.」と言えなくてはならず，同時に `x` がどういうものであるべきか分かっていなければならない．Lisp プログラミングは，何よりも必要に応じて新ユーティリティを生み出すことを必然的に伴う．この章の狙いは，そういったユーティリティがどのように生まれるかを示すことだ．

ここで，towns は近隣の街を近い方から順に並べたリストで，bookshops は市内の全ての書店のリストを返す関数だとしよう．書店がある一番近い街と，その中の書店を探したいとき，まず次のものから始めよう：

```
(let ((town (find-if #'bookshops towns)))
  (values town (bookshops town)))
```

しかしこれは少し格好悪い：find-if が「bookshops が非 nil の値を返すような要素」を見つけたときもその値は捨てられ，find-if から戻った直後に再びその値を求めるようになっている．bookshops の呼び出しのコストが大きいとすると，慣用法に従ったこの方法は格好悪いばかりか非効率にもなるだろう．不必要な処理を避けるため，代わりに次の関数を使う：

```
(defun find-books (towns)
  (if (null towns)
      nil
      (let ((shops (bookshops (car towns))))
        (if shops
            (values (car towns) shops)
            (find-books (cdr towns))))))
```

(find-books towns) の呼び出しでは，少なくとも必要最低限の計算で望みの結果が出る．しかしちょっと待て——いつか将来，再び似たような検索をしたくなることはないだろうか？ここで本当に必要なのは，find-if と，発見された要素と比較関数の返り値の両方を返す何かの関数を結合するユーティリティだ．そのようなユーティリティは

```
(defun find2 (fn lst)
  (if (null lst)
      nil
      (let ((val (funcall fn (car lst))))
        (if val
            (values (car lst) val)
            (find2 fn (cdr lst))))))
```

として定義できる．find-books と find2 が似ていることに注意しよう．実際，後者は前者の全体構造として表せる．今，新しいユーティリティを使えば，最初の目的を 1 個の式で達成できる：

```
(find2 #'bookshops towns)
```

Lisp プログラミング独特の特徴の一つは，引数としての関数の重要性だ．これは Lisp がボトムアップ・プログラミングに適している理由の一部だ．関数の骨格を抽象化するのは，引数に関数を使うことで肉付けができるときには比較的簡単だ．

プログラミングの入門課程では，最初に「抽象化は二度手間の回避につながる」と教える．その中の初歩の一つに「動作を重複させてはいけない」とある．例えば，1〜2 個の定数に従って同じことをする関数を定数別に定義するのではなく，1 個の関数を定義し，それに定数を引数として与えればいい．

Lisp では関数全体を引数として渡せるので，この考えをさらに深めることができる．前述の例の両方で，特定の関数から始めて，関数を引数に取る一般的な関数に進んだ．1 番目の例では既に定義されていた mapcan を使い，2 番目の例では新しいユーティリティ find2 を書いたが，全体的な原則は同じだ：一般部分と個別部分を混ぜ合わせるのでなく，一般部分を定義して個別部分を引数として渡すこと．

注意深く適用すれば，この原則は目覚ましい程エレガントなプログラムをもたらす．ボトムアップ・プログラミングを支える力はそれだけではないが，これは主要なものだ．この章で定義した 32 個のユーティリティのうち，18 個が関数を引数に取ることができる．

3.2 抽象化への投資

簡潔さがウィットの魂ならば，それはまた，効率と並んで，いいソフトウェアの真髄でもある．プログラムを書いたり保守するのにかかるコストは，プログラムが長くなるにつれて高まる[†]．他の条件が同じなら，短いプログラムの方がいい．

この視点からは，ユーティリティの作成は主要な労力配分の対象と見なされるべきだ．find-books をユーティリティ find2 で置き換えた結果，コードの行数は変わらなかった．しかしある意味ではプログラムを短くしたことになった．なぜならユーティリティの長さは現在製作中のプログラムの長さに加えなくてもいいからだ．

Lisp の拡張を主要な労力配分の対象として扱うのは，単なる数字上のごまかしではない．ユーティリティは独立したファイルに収めることもできる．プログラム製作に従事しているときには，それらは私達の目を煩わせないし，後でそのプログラムの何かを変更しないといけなくなったときにも，それらの中身が関わってくることはまずないだろう．

しかし，主要な労力配分の対象として，ユーティリティには余分に注意を払わないといけない．それは上手く書けていることが特に重要だ．繰り返し使われるものだから，間違いや非効率な動作も繰り返されるだろう．そのデザインにも余分な注意が必要だ：新しいユーティリティは，目前の問題だけではなく一般的な状況に対して書かなければいけない．最後に，その他の主要な労力配分の対象と同様，慌てて書いてはいけない．新しい何かのオペレータを生み出そうと考えているが，別のときに必要になるかどうかよく分からないときは，それを書いてしまうこと．しかし，それを使う特定のプログラム内にまだ置いておくのだ．その後，他のプログラムでその新オペレータを使うことがあれば，それをサブルーチンからユーティリティへ昇格させて広く利用可能にすればいい．

ユーティリティ `find2` はよい投資だったようだ．7 行の投資をすることで即座に 7 行が節約できた．ユーティリティというものは，1 回使うだけで元が取れる．Guy Steele の言葉に，プログラミング言語は「私達の持つ簡潔さへの自然な傾向と協調」しなければならない，とある．

> ...プログラミング構造のコストはプログラマの強いられた苦労の量に比例する，と信じがちだ（ここで「信じる」という言葉は，熱烈な確信ではなく無意識の傾向という意味で使った）．実際，プログラミング言語デザイナがこの精神的原則を心に留めておくのは悪いことではない[†]．加算操作をコストの低いものと思う理由の一つは，それを 1 文字 “+” で表記できることだ．あるプログラミング構造に高いコストがかかるとは思っても，それがコード書きの労力を半分にしてくれるなら，しばしばコストの低いものよりそちらが好まれる．

どのプログラミング言語でも「簡潔さへの傾向」は，それを新しいユーティリティに適応させられる柔軟性がない限り，問題の種になる．一番短い慣用法が一番効率的なことは滅多にない．あるリストが別のリストより長いかどうか知りたいとき，素のままの Lisp では

```
(> (length x) (length y))
```

と書きたくなる．ある関数を複数のリストの要素に適用したいときも，同様にリストを次のように連結したくなる：

```
(mapcar fn (append x y z))
```

こうした例から，ユーティリティはそれなしでは非効率的になってしまう状況に対して書くことが大事だと分かる．適切なユーティリティによって補強されたプログラミング言語があれば，一層抽象的なプログラムが書けるだろう．更にそれらのユーティリティは，適切に定義されていれば，効率的なプログラムを書けるよう導いてくれる．

ユーティリティ集は確実にプログラミングを簡単にしてくれる．しかしユーティリティ集の役割はそれだけではない：いいプログラムが書けるようにもしてくれる．芸術の神は，コックと同じで，材料を目にした時点で行動に移る．だから芸術家はアトリエ内にたくさんの道具や画材を持っておきたがるのだ．彼らは，必要なものを手元にしっかり準備しておけば，新しいことを始め易いことを知っている．同じ現象はボトムアップ・スタイルで書かれるプログラムでも現れる．新しいユーティリティを書くと，それを予想より多く使うことに気付くだろう．次の章では，数種類のユーティリティ関数を説明する．それらは，どのような意味においても，あなたが Lisp に追加したいと思うような関数の全ての種類を代表しているわけではない．しかし，例示されたユーティリティはどれも現場で価値を証明されたものばかりだ．

3.3 リストに対する操作

元々，リストは Lisp の主要なデータ構造だった．事実，“Lisp” という名前は “LISt Processing” から来ている．しかし Lisp がこの歴史上の事実のせいで誤解されては困る．ポロシャツがポロ専用でないのと同じく，Lisp はリスト操作用に生まれたのではない．高度な最適化を施された Common Lisp プログラムでは，リストを全く見かけないだろう．

しかし，Lisp プログラムはコンパイル時にはやはりリストだ．高度に洗練された Lisp プログラムは実行時には余りリストを使わないが，逆にその分，コンパイル時のマクロ展開を生成するときには多くのリストを使う．だから現代の Lisp 方言ではリストの役割は減少したが，リストに対する操作は依然として Lisp プログラムの大きな部分を占めることがある．

```
(proclaim '(inline last1 single append1 conc1 mklist))

(defun last1 (lst)
  (car (last lst)))

(defun single (lst)
  (and (consp lst) (not (cdr lst))))

(defun append1 (lst obj)
  (append lst (list obj)))

(defun conc1 (lst obj)
  (nconc lst (list obj)))

(defun mklist (obj)
  (if (listp obj) obj (list obj)))
```

図 4　リストに作用する小さな関数

第 4図と第 5図には，リストを生成したりリストに関する条件判断を行う関数を幾つか示した．第 4図のものは，定義する価値のあるユーティリティの中でも最小規模のものだ．効率性のため，それらにはみなインライン宣言をするべきだ．

1 番目の last1 はリストの最後の要素を返す．組み込み関数 last が返すのはリストの最後のコンスで，最後の要素ではない．last を使うのは大抵 (car (last ...)) として最後の要素を得るためだ．そんな場合に新しいユーティリティを書く価値があるだろうか？そう，それが実質的に組み込み関数の代わりになるようなときは書く価値がある．last1 はエラー・チェックをしないことに注意すること．一般的には，この本で定義されたコードはどれもエラー・チェックをしない．これは例を簡潔にするためもあるが，短いユーティリティ内ではとにかくエラー・チェックを一切しない方が理に適っている．もし次のようにすると

```
> (last1 "blub")
>>Error: "blub" is not a list.
Broken at LAST...
```

エラーは last 自身に捉えられる．ユーティリティが小さいときは，それが形成する抽象化の層は大変薄く，ほとんど透明だ．薄い氷の向こうが透けて見えるように，last1 のようなユーティリティの向こうは透けて見えるので，その背後の関数で起きたエラーは解釈することができる．

関数 single は引数が 1 個の要素を持つリストかどうかを調べる．Lisp のプログラムではこれをかなり頻繁に調べなくてはならない．最初はつい人間語からの自然な翻訳を使いがちだ：

```
(= (length lst) 1)
```

こうして調べるのはとても非効率的だ．第 1 要素を見終わった直後にはもう必要な情報は分かっているからだ．

次は append1 と conc1 だ．両方ともリストの末尾に新しい要素を付け加えるが，後者は破壊的だ．これらの関数は小さいが，大変頻繁に使われるので定義しておく価値はある．実際，append1 は以前の Lisp 方言では組み込み関数だった．

mklist も（少なくとも）Interlisp では組み込み関数だった．これは引数がリストであるかどうかを確かめる．Lisp の関数の多くは，単一の値と複数の値のリストのどちらかを返すようになっている．lookup がそんな関数で，data というリストの要素全てについてそれを呼び出した結果を集めたいとしよう．それは次のようにすれば可能だ：

```
(mapcan #'(lambda (d) (mklist (lookup d)))
        data)
```

第 5図にはリスト用ユーティリティの長めの例が示されている．1 番目は長いが，抽象性もさることながら効率の面から見ても有益なものだ．それは 2 個の連続構造 (sequence) を比較し，1 番目の方が長いときのみ真を返す．2 個のリストの長さを比較したいとき，つい次のようにしたくなる：

```
(> (length x) (length y))
```

```
(defun longer (x y)
  (labels ((compare (x y)
                    (and (consp x)
                         (or (null y)
                             (compare (cdr x) (cdr y))))))
    (if (and (listp x) (listp y))
        (compare x y)
        (> (length x) (length y)))))

(defun filter (fn lst)
  (let ((acc nil))
    (dolist (x lst)
      (let ((val (funcall fn x)))
        (if val (push val acc))))
    (nreverse acc)))

(defun group (source n)
  (if (zerop n) (error "zero length"))
  (labels ((rec (source acc)
                (let ((rest (nthcdr n source)))
                  (if (consp rest)
                      (rec rest (cons (subseq source 0 n) acc))
                      (nreverse (cons source acc))))))
    (if source (rec source nil) nil)))
```

図5　リストに作用する関数の大規模なもの

この慣用法は，両方のリスト全体を探索しているので非効率だ．もし片方の方がずっと長ければ，その差の部分を探索した手間は無駄になってしまう．longer のように 2 個のリストを並行的に探索する方が速い．

longer の中には 2 個のリストの長さを比べる再帰関数が埋め込まれている．longer は長さを比べるためのものだから，その再帰関数は length の引数に与えられるもの全てに対して機能するべきだ．しかし並行的に長さを比較することができるのはリストに適用されたときだけなので，内部の再帰関数は引数が両方リストだったときのみ呼び出される．

次の関数 filter の性質は，find-if に対する remove-if-not の性質と似ている．

組み込み関数 remove-if-not の返り値は，find-if に関数を渡してリストの連続した cdr 部に適用したときの返り値全体と同じだ．それと似たように，filter の返り値は，ある関数がリストの連続した cdr 部に対して返すものと同じだ：

```
> (filter #'(lambda (x) (if (numberp x) (1+ x)))
    '(a 1 2 b 3 c d 4))
(2 3 4 5)
```

filter は関数と 1 個のリストを取り，その関数がリストに適用されたときに非 nil 値が返されるような要素全てをリストにして返す．

filter は，第 2.8 節の末尾再帰関数と同様な総和変数を使っている点に注意して欲しい．実際，末尾再帰関数を書くことの狙いは，コンパイラに filter と同じ構造のコードを生成させることにある．filter に対しては，反復構造による直接的な定義の方が末尾再帰によるものより単純だ．filter の定義コード内での push と nreverse との組み合わせは，Lisp でリストの総和を求める際の一般的な慣用法だ．

第 5図の最後の関数は，リストを新しいリストの部分リストとしてまとめるものだ．group にリスト l と数 n を与えると新しいリストが返され，その中では l の要素は長さ n の部分リストにまとめられている．余りは最後の部分リストに入る．だから第 2 引数に 2 を与えると

```
> (group '(a b c d e f g) 2)
((A B) (C D) (E F) (G))
```

という連想リストが得られる．この関数は，末尾再帰形式（第 2.8 節）にするためにかなり入り組んだ方法で書かれている．ラピッド・プロトタイピングの原則は，プログラム全体と同様に個々の関数にも適用できる．flatten のような

```
(defun flatten (x)
  (labels ((rec (x acc)
                (cond ((null x) acc)
                      ((atom x) (cons x acc))
                      (t (rec (car x) (rec (cdr x) acc))))))
    (rec x nil)))

(defun prune (test tree)
  (labels ((rec (tree acc)
                (cond ((null tree) (nreverse acc))
                      ((consp (car tree))
                       (rec (cdr tree)
                            (cons (rec (car tree) nil) acc)))
                      (t (rec (cdr tree)
                              (if (funcall test (car tree))
                                acc
                                (cons (car tree) acc)))))))
    (rec tree nil)))
```

図 6　2 重再帰を使ったリスト・ユーティリティ

関数を書くとき，可能な限り単純な実装方法から始めるのがいいだろう．そして単純なものが正しく動作すれば，必要に応じて効率的な末尾再帰版や反復版に変えることができる．最初のものは（十分短ければ）コメントとして残せば，交換後の関数の動作の説明になる．（group や第 4図と第 5図内の他の関数の単純な実装は，第 vaom ページ内の note に書かれている．）

group の定義は最低でも 1 種類のエラーチェックをする点で他と異なっている：第 2 引数が 0 であるかどうかだ．その場合，チェックをしないと無限の再帰に陥ってしまう．

この本の中の例は，ある 1 点で通常の Lisp の慣習から外れている：それぞれの章を独立させるため，コード例は可能な限り素の Lisp で書かれている点だ．しかし group は例外だ．これはマクロ定義に大変便利なので，この後の章でも数回再登場する．

第 5図の関数は，みなリストのトップレベル構造に沿って動作する．第 6図には入れ子になったリスト内へ下っていく関数の例を 2 個示した．1 番目の flatten も Interlisp では組み込み関数になっている．これは引数のリストの要素であるか，要素の要素であるか，そのまた要素の...といった全てのアトムのリストを返す．

```
> (flatten '(a (b c) ((d e) f)))
(A B C D E F)
```

第 6図の 2 番目の関数 prune と istoremove-if との関係は copy-tree と copy-list との関係と似ている．つまり，再帰的に部分リスト内へ下っていく：

```
> (prune #'evenp '(1 2 (3 (4 5) 6) 7 8 (9)))
(1 (3 (5)) 7 (9))
```

引数として与えた関数が真を返す葉は全て除かれる．

3.4　検索

この章では，リストを検索する関数の例を幾つか与える．Common Lisp にはそのための組み込みオペレータが豊富にあるが，それでも困難な——少なくとも効率的に実行するのは困難な——課題はある．このことは第 pump ページで説明した仮想的な状況の中で既に見た†．第 7図の 1 番目の関数 find2 は，それへの回答として作られたものだ．

次のユーティリティ before は似た意図を持って書かれた．これはあるオブジェクトがリスト内で別のオブジェクトよりも先に現れるかどうかを調べる：

```
> (before 'a 'b '(a b c d))
(B C D)
```

これを素の Lisp を使って愚直に調べることも十分簡単だ：

```
(< (position 'a '(a b c d)) (position 'b '(a b c d)))
```

```
(defun find2 (fn lst)
  (if (null lst)
    nil
    (let ((val (funcall fn (car lst))))
      (if val
          (values (car lst) val)
          (find2 fn (cdr lst))))))

(defun before (x y lst &key (test #'eql))
  (and lst
       (let ((first (car lst)))
         (cond ((funcall test y first) nil)
               ((funcall test x first) lst)
               (t (before x y (cdr lst) :test test))))))

(defun after (x y lst &key (test #'eql))
  (let ((rest (before y x lst :test test)))
    (and rest (member x rest :test test))))

(defun duplicate (obj lst &key (test #'eql))
  (member obj (cdr (member obj lst :test test))
          :test test))

(defun split-if (fn lst)
  (let ((acc nil))
    (do ((src lst (cdr src)))
      ((or (null src) (funcall fn (car src)))
       (values (nreverse acc) src))
      (push (car src) acc))))
```

図 7　リストを検索する関数

しかしこの慣用法は非効率だし，エラーにつながり易い．非効率な理由は，この関数は本当は両方のオブジェクトを見つける必要はなく，先に現れるものだけを見つければいいからだ．またエラーにつながるというのは，オブジェクトがどちらもリスト内にないとき，nil が < に引数として渡されてしまうからだ．before を使えば両方の問題が解決する．

before の精神は集合への所属関係を調べることに似ているので，組み込み関数 member に似せて書かれた．member と同様に比較関数（デフォルトでは eql）をオプショナル引数として取る．また，ただ t を返すだけでなく，第 1 引数で与えられたオブジェクトで始まる cdr 部という，有益な情報を含んだ値を返す：

before が真を返すのは，第 2 引数を見つける前に第 1 引数を見つけたときだということに注意．そのため，第 2 引数がリスト内に存在しないときには真が返される：

```
> (before 'a 'b '(a))
(A)
```

もっと精密に調べるには，after（引数が両方リスト内に存在しないと真を返さない）を使う：

```
> (after 'a 'b '(b a d))
(A D)
> (after 'a 'b '(a))
NIL
```

(member o l) がリスト l 内に o を見つけたとき，ただの t でなく o で始まる l の cdr 部を返す．この返り値は，例えば重複して存在するオブジェクトを調べるために使える．l 内に o が重複して存在していると，それは member の返したリストの cdr 部内にも存在する．この慣用法は次のユーティリティ duplicate 内に埋め込まれている：

```
> (duplicate 'a '(a b c a d))
(A D)
```

重複を調べるユーティリティは同じ原則に基づいて他にも定義できる．潔癖性のプログラミング言語デザイナは，Common Lisp では「偽」と「空リスト」の両方を nil で表すことに衝撃を受ける．確かにそれが問題を起こすこともある（第 14.2 章を参照）が，duplicate 等の関数では便利だ．連続構造の包含関係を問うとき，偽であることを空の連

```
(defun most (fn lst)
  (if (null lst)
      (values nil nil)
      (let* ((wins (car lst))
             (max (funcall fn wins)))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (when (> score max)
              (setq wins obj
                    max score))))
        (values wins max))))

(defun best (fn lst)
  (if (null lst)
      nil
      (let ((wins (car lst)))
        (dolist (obj (cdr lst))
          (if (funcall fn obj wins)
              (setq wins obj)))
        wins)))

(defun mostn (fn lst)
  (if (null lst)
      (values nil nil)
      (let ((result (list (car lst)))
            (max (funcall fn (car lst))))
        (dolist (obj (cdr lst))
          (let ((score (funcall fn obj)))
            (cond ((> score max)
                   (setq max score
                         result (list obj)))
                  ((= score max)
                   (push obj result)))))
        (values (nreverse result) max))))
```

図 8　要素を比較する検索関数

続構造として表現するのは自然に思える．第 7図の最後の関数も，member の一般化の一種だ．member は，それが発見した要素から始まるリストの cdr 部を返すのに対し，split-if は元のリストの前半分もあわせて返す．このユーティリティは，主に何かの順番で整列されたリストに対して使われる：

```
> (split-if #'(lambda (x) (> x 4))
            '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4)
(5 6 7 8 9 10)
```

第 8には別の種類の検索関数が載っている：要素同士を比較する関数だ．最初の most は要素を 1 度に 1 個ずつ調べる．これはリストと点数付け関数を引数に取り，最高点を与える要素を返す．等しい点を与える要素があったときは，リスト内で最初に現れたものを返す．

```
> (most #'length '((a b) (a b c) (a) (e f g)))
(A B C)
3
```

most は返した要素の与えた（最高）点も返す（その方が便利だ）．

もっと一般的な種類の検索には best を使う．このユーティリティも関数とリストを引数に取るが，これに使う関数は 2 個の引数を取る述語でなければいけない．返り値は 1 個の要素で，述語がその他の要素全てに対して勝っていると判断するものだ．

```
> (best #'> '(1 2 3 4 5))
5
```

best は sort の結果の car 部と見ることもできるが，こちらの方がずっと効率がいい．リスト内の要素に完全な順位を定義する述語を提供するかどうかは呼び出し側次第だ．そうでなければ要素の並び順は結果に影響し，most と同様，等しい点が出たときには最初の要素が返される．

最後の mostn は点数付け関数とリストを引数に取り，関数が最高点を付ける要素全てから成るリスト（と最高点）を返す：

```
> (mostn #'length '((a b) (a b c) (a) (e f g)))
((A B C) (E F G))
3
```

3.5 対応付け

広く使われる Lisp の関数の種類には，他に対応付け関数——ある関数を複数の引数に適用するもの——がある．第 9, 10図には新しい対応付け関数の例を示した．最初の 3 個は関数をある範囲の数に（それらの数を含むリストをコンシングせずに）適用するためのものだ．最初の 2 個，map0-n と map1-n は，正の整数の範囲で動作する：

```
> (map0-n #'1+ 5)
(1 2 3 4 5 6)
```

両方ともさらに一般的な mapa-b（任意の範囲の数に対して動作する）を使って定義されている．

```
> (mapa-b #'1+ -2 0 .5)
(-1 -0.5 0.0 0.5 1.0)
```

mapa-b の次はさらに一般的な map->で，これは任意のオブジェクトの連続構造に対して機能する．連続構造は第 2 引数で与えられたオブジェクトから始まり，第 3 引数で与えられたオブジェクトで終わって，その間のオブジェクトは第 4 引数で与えられた関数によって生成される．map->を使うと，数の連続への操作と同様に，任意のデータ構造を扱うことが可能だ．mapa-b は map->を使うと次のように定義できる：

```
(defun mapa-b (fn a b &optional (step 1))
  (map-> fn
         a
         #'(lambda (x) (> x b))
         #'(lambda (x) (+ x step))))
```

組み込み関数 mapcan は効率のために破壊的関数として作られている．それは次のようにして定義できる：

```
(defun our-mapcan (fn &rest lsts)
  (apply #'nconc (apply #'mapcar fn lsts)))
```

mapcan は nconc を使ってリストをつなぎ合わせているので，第 1 引数によって返されるリストは新しく作られたものの方がいい．そうしないと次にそのリストを使うときには変更を受けていることだろう．nicknames（dove ページ）が，愛称の「リストを生成する」関数として定義されたのはそういった理由による．それがただどこかに保持されたリストを返すだけなら，mapcan を使うのは安全でなくなるだろう．代わりに，返されたリストをつなぎ合わせるのには append を使わなければならなくなる．そのような場合のため，mappend は「非破壊的な mapcan」を提供する．

次のユーティリティ mapcars が使われるのは，mapcar で関数を適用したいリストが複数あるときだ．ここで数のリストが 2 個あり，両方のリストの要素の平方根から成る 1 個のリストが欲しいとしよう．素の Lisp を使って

```
(mapcar #'sqrt (append list1 list2))
```

とすれば実現できるが，そうすると不必要なコンシングが行われている．list1 と list2 とをつなぎ合わせても，その結果はすぐに捨てられてしまう．mapcars を使うと

```
(mapcars #'sqrt list1 list2)
```

によって同じ結果が得られ，不必要なコンシングは行われていない．

第 10図の最後の関数は，mapcar のツリー対応版だ．rmapcar という名前は “recursive（再帰的） mapcar” の省略で，これは mapcar が単層リストに対して行う操作をツリーに対して行う：

```
> (rmapcar #'princ '(1 2 (3 4 (5) 6) 7 (8 9)))
123456789
(1 2 (3 4 (5)6)7 (8 9))
```

これは mapcar と同様，複数のリストを引数に取れる．

```
(defun map0-n (fn n)
  (mapa-b fn 0 n))

(defun map1-n (fn n)
(mapa-b fn 1 n))

(defun mapa-b (fn a b &optional (step 1))
  (do ((i a (+ i step))
       (result nil))
    ((> i b) (nreverse result))
    (push (funcall fn i) result)))

(defun map-> (fn start test-fn succ-fn)
  (do ((i start (funcall succ-fn i))
       (result nil))
    ((funcall test-fn i) (nreverse result))
    (push (funcall fn i) result)))
```

図 9　対応付け関数 1

```
(defun mappend (fn &rest lsts)
  (apply #'append (apply #'mapcar fn lsts)))

(defun mapcars (fn &rest lsts)
  (let ((result nil))
    (dolist (lst lsts)
      (dolist (obj lst)
        (push (funcall fn obj) result)))
    (nreverse result)))

(defun rmapcar (fn &rest args)
  (if (some #'atom args)
      (apply fn args)
      (apply #'mapcar
             #'(lambda (&rest args)
                 (apply #'rmapcar fn args))
             args)))
```

図 10　対応付け関数 2

```
> (rmapcar #'+ '(1 (2 (3) 4)) '(10 (20 (30) 40)))
(11 (22 (33) 44))
```

この後で登場する関数のうち幾つか（第 prep ページの rep 等）は，本当は rmapcar を使うべきものだ．

伝統的なリストの対応付け関数は，CLtL2 で導入された数列マクロにより，時代遅れなものとしてある程度置き換えられた．例えば

```
(mapa-b #'fn a b c)
```

は次のように変えることができる：

```
(collect (#Mfn (scan-range :from a :upto b :by c)))
```

しかし対応付け関数への需要はまだある．場合によっては対応付け関数の方が明確でエレガントなことがある．map-> で表現できるものも数列では表現し辛いこともある．最後に，対応付け関数は引数として渡すことができる（関数としての性質）．

3.6　入出力

第 11図では I/O ユーティリティの例を 3 個示した．この種のユーティリティへの需要はプログラム毎にまちまちだ．第 11図のものは代表的な例に過ぎない．1 番目のものはプログラムのユーザに括弧なしで式を入力させたいときに使

```
(defun readlist (&rest args)
  (values (read-from-string
            (concatenate 'string "("
                         (apply #'read-line args)
                         ")"))))

(defun prompt (&rest args)
  (apply #'format *query-io* args)
  (read *query-io*))

(defun break-loop (fn quit &rest args)
  (format *query-io* "Entering break-loop.~%")
  (loop
    (let ((in (apply #'prompt args)))
      (if (funcall quit in)
          (return)
          (format *query-io* "~A~%" (funcall fn in))))))
```

図 11　入出力関数

う．これは 1 行分の入力を読み取り，それをリストとして返す：

```
> (readlist)
Call me "Ed"
(CALL ME "Ed")
```

values を呼び出すことで返り値を 1 個に限定している (read-from-string 自身，ここでは無駄な第 2 値を返す)．

関数 prompt は質問の表示と回答の読み取りを統合したものだ．これは format に対する引数と同じもの（ただしストリーム指定用の第 1 引数以外）を引数に取る．

```
> (prompt "Enter a number between ~A and ~A.~%>> " 1 10)
Enter a number between 1 and 10.
>> 3
3
```

最後の break-loop を使うのは Lisp のトップレベルを真似たいときだ．これは引数に 2 個の関数とレスト引数（プロンプトとして繰り返し渡される）を取る．2 番目の関数が入力に対して偽を返す限り，1 番目の関数が入力に適用される．だから，例えば実際の Lisp のトップレベルは次のように模倣できる：

```
> (break-loop #'eval #'(lambda (x) (eq x :q)) ">> ")
```

すると break-loop の中に入る．

```
>> (+ 2 3)
5
>> :q
:Q
```

ところで，Common Lisp のヴェンダが一般的に実行時ライセンスを主張するのはこれが理由だ．実行時に eval を呼べば，任意の Lisp プログラムが Lisp 自身を包含できる．

3.7　シンボルとストリング

シンボルとストリングは大変近い関係にある．表示関数と読み取り関数の手段として，二つの表現は混用できる．第12図にはこの境界で動作するユーティリティの例を示した．最初の mkstr は任意の数の引数を取り，それらの印字表現を連結してストリングとして返す：

```
> (mkstr pi " pieces of " 'pi)
"3.141592653589793 pieces of PI"
```

symb はそれに基づいて作られたもので，主にシンボルの生成に使われる．これは 1 個以上の引数を取り，その印字名を連結したものを印字名とするシンボルを（必要ならば新しく生成して）返す．これは印字表現を持つ任意のオブジェクト——シンボル，ストリング，数，さらにリストまで——を引数に取れる．

```
(defun mkstr (&rest args)
  (with-output-to-string (s)
    (dolist (a args) (princ a s))))

(defun symb (&rest args)
  (values (intern (apply #'mkstr args))))

(defun reread (&rest args)
  (values (read-from-string (apply #'mkstr args))))

(defun explode (sym)
  (map 'list #'(lambda (c)
                 (intern (make-string 1
                                      :initial-element c)))
       (symbol-name sym)))
```

図 12　シンボルとストリングに作用する関数

```
> (symb 'ar "Madi" #\L #\L 0)
|ARMadiLL0|
```

mkstr を呼んで引数をみな 1 個のストリングに連結した後，symb はそれを intern に送る．これは Lisp の伝統的なシンボル生成用関数で，ストリングを引数に取り，そのストリングで表示されるシンボルを返すか，新しく生成する．任意のストリングが（小文字や，括弧などのマクロ文字を含むものでも）シンボルの印字名として使える．シンボルの名前にそういった変わった文字が使われているときは，名前は上のように縦棒に挟まれて表示される．ソースコードでは，そういったシンボルは縦棒の間にあるか，“¥” が文字の前に付いていなければいけない：

```
> (let ((s (symb '(a b))))
(and (eq s '|(A B)|) (eq s '\(A\ B\))))
T
```

次の関数 reread は，symb の一般形だ．オブジェクトの列を引数に取り，それらを表示し，再び読み込む．これは symb と同様にシンボルを返せるが，他にも read が返せるものなら何でも返せる．リードマクロはシンボルの名前の一部としては扱われず，対応する関数が呼び出される．また a:b はカレント・パッケージの|a:b|というシンボルではなく，パッケージ a 内のシンボル b として読み込まれる[*10]．一般的な関数は几帳面でもある：reread は，引数が正しい Lisp 文法に従っていないとエラーを発生する．

第 12図の最後の関数は，昔の幾つかの方言では組み込み関数だった：explode はシンボルを引数に取り，そのシンボルの名前に使われている文字から作られるシンボルから成るリストを返す．

```
> (explode 'bomb)
(B O M B)
```

この関数が Common Lisp に含まれなかったのは偶々という訳ではない．シンボルを分解したいようなときには，おそらく非効率的な作業をしているのだ．しかし製品にするソフトウェアでなくてその原型の中なら，この類のユーティリティを使う余地はある．

3.8　密度

コード内に多くの新ユーティリティを使うと，理解し辛いと言ってくる人がいる．Lisp をまだ使いこなせない人は素の Lisp しか読みとれない．そういう人は拡張可能なプログラミング言語という考えが全く身に付いていないのだ．そういう人がユーティリティをふんだんに使ったプログラムを目にすると，その作者は全く変人で，プログラムを一種の個人用プログラミング言語で書くことに決めてしまったと思うことだろう．

それらの新オペレータは，どれも（議論の余地はあるが）プログラムを読み辛くしてしまう．プログラムを読み取れるようになる前に，それらのユーティリティを全て理解しなければいけない．こういった言明がなぜ誤解されるのかについては，pop ページで説明した例（一番近い書店を探した例）のことを考えてみて欲しい．そのプログラムを find2

[*10] パッケージの紹介は，pow ページから始まる付章を参照.

を使って書けば,「プログラムを読み取れるようになる前に，この新ユーティリティの定義を理解しなければいけないじゃないか.」と不満を言う人が出てくる．それでは，find2 を使わなかったとしてみよう．すると find2 の定義は理解しなくてもいいが，find-books の定義を理解しなければいけない．その中では find2 の仕事が「書店を見つける」という個別の課題と混ざっている．find2 を理解するのはせいぜい find-books と同じくらい難しいだけだ．また，ここでは新ユーティリティは 1 回しか使っていない．ユーティリティは繰り返し使うよう意図されたものだ．実際のプログラムでは，find2 を理解しなければいけないか，または 3, 4 個の特定目的の検索ルーチンを理解しなければいけないかの，どちらかの選択だろう．前者の方が確実に簡単だ．

そう，ボトムアップ形式のプログラムを読み取るには，作者の定義した新ユーティリティを全て理解しなければいけない．しかしこれにかかる労力は，ユーティリティなしの場合に必要なコードを理解しなければならないときの労力よりは，ほとんど常に少ないだろう．ユーティリティを使うとコードが読み辛くなると言う人がいたとすれば，その人達はユーティリティを使わないとコードがどんなものになるか理解していないのだろう．ボトムアップ・プログラミングは，それなしでは巨大なプログラムになった筈のものを，小さくて単純なプログラムに見えるように変えてくれる．こうするとそのプログラムは大した量の処理を行っていないとの印象を与え，そのため理解が簡単になる筈だ．Lisp に未熟な人がコードをよく見て，実際はかなりの処理を行っていることに気付くと，狼狽するのだ．

同じ現象は別の分野でも発見できる：上手に設計された機械の部品は少ないが，より複雑に見える．それは部品が狭い空間に詰め込まれているからだ．ボトムアップ形式のプログラムは概念の密度が高い．それは読み取るのに労力が必要かもしれないが，それはプログラムがボトムアップ形式で書かれていなかったときに必要になった筈の労力程ではない．慎重に考えた上でユーティリティの使用を避けた方がいい場合が一つある：コードの大部分とは独立して配布する小さなプログラムを書かなければいけないときだ．ユーティリティというものは通常 2, 3 回使って始めて元が取れるものだが，小規模なプログラムではユーティリティを含める意味がある程には使えないことがある．

4　返り値としての関数

前章では，関数を引数として渡せることが抽象化への可能性をどれ程大きくするかを見た．関数に対して行える操作が豊かな程，その可能性を深く利用できる．新しい関数を生成して返す関数を定義することで，関数を引数に取るユーティリティの効果を増幅できる．

この章で示すユーティリティは関数に対して動作する．多くのユーティリティを式に対して動作するように書く方が（少なくとも Common Lisp では）自然だろう．つまり，マクロとして書くのだ．第 15 章では，これらのオペレータの幾つかにマクロの層が挿入される．しかしそれらの関数が結局はマクロを通じてのみ呼び出されるとしても，機能のどの部分が関数で実現できるのかを知ることは重要なことだ．

4.1　Common Lisp は進化する

Common Lisp には元々コンプリメント関数 (complement function) の対が幾つかある．関数 remove-if と remove-if-not もそういった対の一つだ．pred が引数を 1 個取る述語だとすると

```
(remove-if-not #'pred lst)
```

と

```
(remove-if #'(lambda (x) (not (pred x))) lst)
```

とは等価だ.

引数として渡された関数をそうした関数に変えることで，もう片方の機能を複製できる．そのとき，どうして両方の必要があるだろう？ CLtL2 はそのような状況を意図した関数を新しく含んでいる：complement は述語 p を取り，常に反対の値を返す関数を返す．p が真を返すとき，コンプリメント関数は偽を返す．逆も同じだ．すると complement を使えば

```
(remove-if-not #'pred lst)
```

は等価な

```
(remove-if (complement #'pred) lst)
```

で置き換えられる．-if-not の類の関数を使い続けることを正当化する理由はほとんどない[*11]．実際 CLtL2 (p. 391) は，それらの使用は現在では推奨されないと記している．それらが Common Lisp に残されるとしたら，その理由は互換性を保つために過ぎないだろう．

新しいオペレータ complement は重要な主題——関数を返す関数——の氷山の一角だ．それは Scheme の慣用法の中では長らく重要な位置を占めていた．Scheme は関数をレキシカルクロージャにした最初の Lisp で，返り値に関数を使うことを興味深いものにしたのもそれだ．

ダイナミックスコープの Lisp では関数を返せない訳ではない．次の関数はダイナミックスコープの Lisp でもレキシカルスコープの Lisp でも同じように動作するだろう：

```
(defun joiner (obj)
  (typecase obj
    (cons #'append)
    (number #'+)))
```

これはオブジェクトを引数に取り，その型に応じてそれらのオブジェクトを加え合わせる関数を返す．これは数やリストに対して働く多態的な (polymorphic) 連結関数の定義に使える：

```
(defun join (&rest args)
  (apply (joiner (car args)) args))
```

しかし予め決めた中から選んで関数を返す程度が，ダイナミックスコープの下でできることの限界だ．実行時に関数を生成することは（上手には）できない．joiner は 2 個の関数の中どちらかを返せるが，返せる 2 個は固定されている．

これとは別に prop ページで関数を返す関数を見た．それはレキシカルスコープによるものだ：

```
(defun make-adder (n)
  #'(lambda (x) (+ x n)))
```

make-adder を呼ぶとクロージャが生成されるが，その振る舞いは元々引数として与えられた値に依存する．

```
> (setq add3 (make-adder 3))
#<Interpreted-Function BF1356>
> (funcall add3 2)
5
```

レキシカルスコープの下では，選択肢の中からどれかの関数を選ぶだけでなく，実行時にクロージャを生成することができる．ダイナミックスコープの下ではその技法は不可能だ[*12]．complement がどのように書かれているかを考えれば，それがクロージャを返す他ないということが分かる：

```
(defun complement (fn)
  #'(lambda (&rest args) (not (apply fn args))))
```

complement の返した関数は，complement が呼ばれた時点でのパラメータ fn の値を使っている．だから固定された選択肢の中から関数を選ぶだけでなく，complement はどの関数のコンプリメント関数でも求めに応じて生成できる：

```
> (remove-if (complement #'oddp) '(1 2 3 4 5 6))
(1 3 5)
```

関数を引数として渡せることは抽象化のための強力な道具だ．関数を返す関数が書けることで，それを最大限に利用できるようになる．残りの節では関数を返すユーティリティの例を幾つか挙げる．

4.2 直交性

直交的 (orthogonal) なプログラミング言語とは，少数のオペレータを多数の様々な方法で結合させることで，多様な意味が表現できるもののことだ．おもちゃのブロックは極めて直交的だが，プラモデルはほとんど直交的でない．complement の主な長所は，プログラミング言語を一層直交的にしていることだ．complement の登場以前，Common Lisp には remove-if と remove-if-not，subst-if と subst-if-not 等の関数の組があった．complement があれば，それらの片方で事は足りる．

[*11] remove-if-not は別だろう．これは remove-if よりも使われている．

[*12] ダイナミックスコープの下でも make-adder のようなものを作れるが，それはまともに動作しないだろう．返された関数が最終的に呼び出された環境に n の束縛が左右され，制御することはできないだろう．

```
(defvar *!equivs* (make-hash-table))

(defun ! (fn)
  (or (gethash fn *!equivs*) fn))

(defun def! (fn fn!)
  (setf (gethash fn *!equivs*) fn!))
```

図 13　等価で破壊的な関数を返す.

マクロ setf も Lisp の直交性を高めている．Lisp の方言の古いものでは，データを読む関数と書く関数の組があったりしたものだ．すると例えば属性リストがあれば，属性を設定する関数が 1 個，求める関数がもう 1 個必要だろう．Common Lisp には後者に当たる get しかない．属性を設定するには get を setf と組み合わせて使う：

```
(setf (get 'ball 'color) 'red)
```

Common Lisp の体系そのものを小さくすることは難しいが，代わりにいい方法がある．Common Lisp の小規模な部分集合だけを使うことだ．私達をこのゴールへ導いてくれる，complement や setf 等の新しいオペレータを定義できないものだろうか？関数を組にまとめられる方法が少なくとも一つある．多くの関数には等価で破壊的な別の関数がある．remove-if と delete-if，reverse と nreverse，append と nconc 等だ．ある関数と等価で破壊的な関数を返すオペレータを定義することで，破壊的な関数を明示する必要がなくなる．

第 13図には「等価で破壊的な関数」の考えを補助するコードを示した．グローバルなハッシュ表 *!equivs* は，ある関数とそれに等価で破壊的な関数との対応付けを行う．def! によって等価で破壊的な関数を設定し，! はそれを返す．オペレータ !（びっくりマーク）は，Scheme で副作用のある関数の名前に ! を付け加える慣習から取った．

```
(def! #'remove-if #'delete-if)
```

さて上のように定義してしまえば

```
(delete-if #'oddp lst)
```

とする代わりに

```
(funcall (! #'remove-if) #'oddp lst)
```

とすればいい．ここでは Common Lisp の造りがまずいせいでこの考えの元々の美点が隠れてしまっているが，Scheme ではもっとはっきり見える：

```
((! remove-if) oddp lst)
```

直交性の向上もさることながら，オペレータ ! は他にも幾つかの利点をもたらす．(! #'foo) は foo と等価で破壊的な関数だと見てすぐ分かるので，プログラムがさらに簡潔になる．また，破壊的な操作がソースコード内で目立って認識しやすい形を取るようになる．それらはバグを探すときに特別な注意を払うべきものだから，これはいいことだ．

ある関数とそれに等価で破壊的な関数との関係は，普通は実行する前に明らかになるので，! はマクロとして定義するのが一番効率的だ．そのためのリードマクロを定義してもいいだろう．

4.3　関数の値のメモワイズ

計算コストの高い関数を同じ引数で複数回呼び出したいときがあるなら，値をメモワイズ (memoize) しておくのが得だ．以前の返り値をみな保管しておき，関数が呼び出される度にまず保管場所を見て，値が既に得られていないか調べる．

第 14図には一般的なメモワイズユーティリティを示した．memoize に関数を渡すと，等価なメモワイズ機能付き関数が返る．それは以前の返り値を蓄えるハッシュ表を持ったクロージャだ．

```
> (setq slowid (memoize #'(lambda (x) (sleep 5) x)))
#<Interpreted-Function C38346>
> (time (funcall slowid 1))
Elapsed Time = 5.15 seconds
1
> (time (funcall slowid 1))
Elapsed Time = 0.00 seconds
1
```

```
(defun memoize (fn)
  (let ((cache (make-hash-table :test #'equal)))
    #'(lambda (&rest args)
        (multiple-value-bind (val win) (gethash args cache)
          (if win
              val
              (setf (gethash args cache)
                    (apply fn args)))))))
```

図 14　メモワイズユーティリティ.

```
(defun compose (&rest fns)
  (if fns
      (let ((fn1 (car (last fns)))
            (fns (butlast fns)))
        #'(lambda (&rest args)
                  (reduce #'funcall fns
                          :from-end t
                          :initial-value (apply fn1 args))))
      #'identity))
```

図 15　合成関数のためのオペレータ.

メモワイズ機能付き関数では，呼び出しの繰り返しはハッシュ表の検索に過ぎない．もちろん新しい値で呼んだときにも検索してしまうという余分な負荷はあるが，十分計算コストの高い関数だけにメモワイズ機能を付けるのだから，その負荷は比較すれば無視できると想定していいだろう．さて上の `memoize` の実装方法は大抵の場合には適しているが，幾つか制限もある．呼び出し方を同じと見なすのは引数リストが `equal` のときだが，これは関数がキーワード引数を取るときには厳しすぎるだろう．また返り値が 1 個の関数のみを想定していて，多値を蓄えたり返したりはできない．

4.4　関数を合成する

関数 f のコンプリメント関数は $\sim f$ と表記される．第 5.1 節では $\sim$ を Lisp の関数として定義するクロージャを示した．関数に対してよく使われる操作には他に合成があり，$\circ$ で表記される．f と g が関数ならば $f \circ g$ も関数で，$f \circ g(x) = f(g(x))$ である．これもクロージャで Lisp の関数として定義できる．

第 15図では関数 `compose` を定義した．これは任意の数の関数を引数に取り，それらの合成を返す．例えば

```
(compose #'list #'1+)
```

が返すのは

```
#'(lambda (x) (list (1+ x)))
```

と等価な関数だ．`compose` の引数に渡された関数は，最後以外はみな引数が 1 個の関数でなければならない[†]．最後の関数には制限は何もなく，それがどんな引数を取っても，`compose` の返す関数も同じ引数を取る．

```
> (funcall (compose #'1+ #'find-if) #'oddp '(2 3 4))
4
```

`not` が Lisp の関数なので，`complement` は `compose` の特殊形と言える．それは

```
(defun complement (pred)
  (compose #'not pred))
```

として定義できる．

関数の組み合わせ方は合成だけではない．例えば

```
(mapcar #'(lambda (x)
            (if (slave x)
                (owner x)
                (employer x)))
        people)
```

```
(defun fif (if then &optional else)
  #'(lambda (x)
      (if (funcall if x)
          (funcall then x)
          (if else (funcall else x)))))

(defun fint (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fint fns)))
        #'(lambda (x)
            (and (funcall fn x) (funcall chain x))))))

(defun fun (fn &rest fns)
  (if (null fns)
      fn
      (let ((chain (apply #'fun fns)))
        #'(lambda (x)
            (or (funcall fn x) (funcall chain x))))))
```

図 16　関数生成方法の更なる例.

のような式をよく見かける．このような関数を自動的に生成するオペレータが定義できる．第 16図の `fif` を使い

```
(mapcar (fif #'slave #'owner #'employer)
        people)
```

とすれば同じ結果になる．

第 16図にはよく使われる種類の関数を生成する関数を幾つか示した．2 番目の `fint` を使うのは次のようなときだ：

```
(find-if #'(lambda (x)
             (and (signed x) (sealed x) (delivered x)))
         docs)
```

`find-if` の第 2 引数に与えられた述語が，その内部で呼ばれている 3 個の述語の共通部分を定義している．`fint`（“function intersection” 関数の共通部分）を使えば

```
(find-if (fint #'signed #'sealed #'delivered) docs)
```

と書ける．同様に，関数集合の合併を返すオペレータも定義できる．関数 `fun` は `fint` と似ているが，`and` でなく `or` を使っている．

4.5　Cdr 部での再帰

Lisp のプログラムでは再帰関数は極めて重要なので，それらを生成するユーティリティを定義しておいて損はない．この章と次の章では，よく使われる 2 種類の再帰関数を生成する関数について説明する．ただ Common Lisp では，これらの関数を使うのはまずい方法だ．話題がマクロまで進めば，この仕組みにさらにこなれた外見を与える方法を知ることになるだろう．再帰関数を生成するマクロについては第 15.2, 15.3 章で議論する．

プログラム内に繰り返した形が現れるのは，より高いレベルの抽象的手法が使えることの印だ．そういう形は Lisp プログラムの中では次のような関数より頻繁に見られる：

```
(defun our-length (lst)
  (if (null lst)
      0
      (1+ (our-length (cdr lst)))))
```

または

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

```
(defun lrec (rec &optional base)
  (labels ((self (lst)
                 (if (null lst)
                     (if (functionp base)
                         (funcall base)
                         base)
                     (funcall rec (car lst)
                              #'(lambda ()
                                  (self (cdr lst)))))))
    #'self))
```

図 17　平坦なリストに対する再帰関数を定義する関数.

```
; copy-list
(lrec #'(lambda (x f) (cons x (funcall f))))
; remove-duplicates
(lrec #'(lambda (x f) (adjoin x (funcall f))))
; find-if, for some function fn
(lrec #'(lambda (x f) (if (fn x) x (funcall f))))
; some, for some function fn
(lrec #'(lambda (x f) (or (fn x) (funcall f))))
```

図 18　lrec で表現された関数.

これら 2 個は構造がかなり共通している．共にリストの連続した cdr 部に再帰的に作用し，毎回同じ式を評価している．ベース・ケースでは別で，他とは違った値を返す．この形は Lisp のプログラム内で大変頻繁に現れるので，経験を積んだプログラマは思考を中断せずに読みこなしたり新しいのを書いたりできる．実際，そういった形を新しい抽象化構造に組み込む方法はすぐに理解できる．

しかし形そのものはみな同じだ．これらの関数を手で書く代わりに，自分のためにそれらを生成してくれる関数を書けるようになっておくべきだ．第 17図には `lrec` (“list recurser”) という関数生成関数を示した．これはリストの連続した cdr 部に再帰的に作用する関数のほとんどを生成してくれる筈だ．`lrec` の第 1 引数は，その時点でのリストの car 部と，再帰を続けるために呼ばれる関数という 2 個の引数を取る関数でなければならない．`lrec` を使えば，`our-length` は

```
(lrec #'(lambda (x f) (1+ (funcall f))) 0)
```

として表現できる．リストの長さを得るには要素を調べたり途中で止まる必要はないので，オブジェクト `x` は常に無視してよく，常に関数 `f` を呼び出せばよい．しかし `our-every` を表現する可能性の利点は両方とも活用する必要がある．例えば `oddp` だ[*13]：

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f))) t)
```

`lrec` の定義では，`self` というローカルな再帰関数を生成するために `labels` を使った．再帰途中に関数 `rec` に与えられる引数は，その時点でのリストの cdr 部と再帰呼び出しを形成する関数の 2 個だ．`our-every` 等の再帰呼び出しを行う部分が最後に来る関数には，第 1 引数が偽を返したその時点で止まって欲しい．これは再帰呼び出しに渡された引数は値でなく，必要があれば呼び出して値を求められる関数でなければならないということだ．

第 18図に示したのは，既存の Common Lisp の関数を `lrec` を使って定義したものだ[*14]．

`lrec` を使っても求める関数の最も効率のよい実装方法に必ず行き着く訳ではない．実際，この章で定義されたような `lrec` を始めとする再帰関数生成関数は，末尾再帰による方法に一歩及ばないことが多い．このためこれらはプログラム開発の初期のうちや，速度が重要でない箇所で使うのが一番だ．

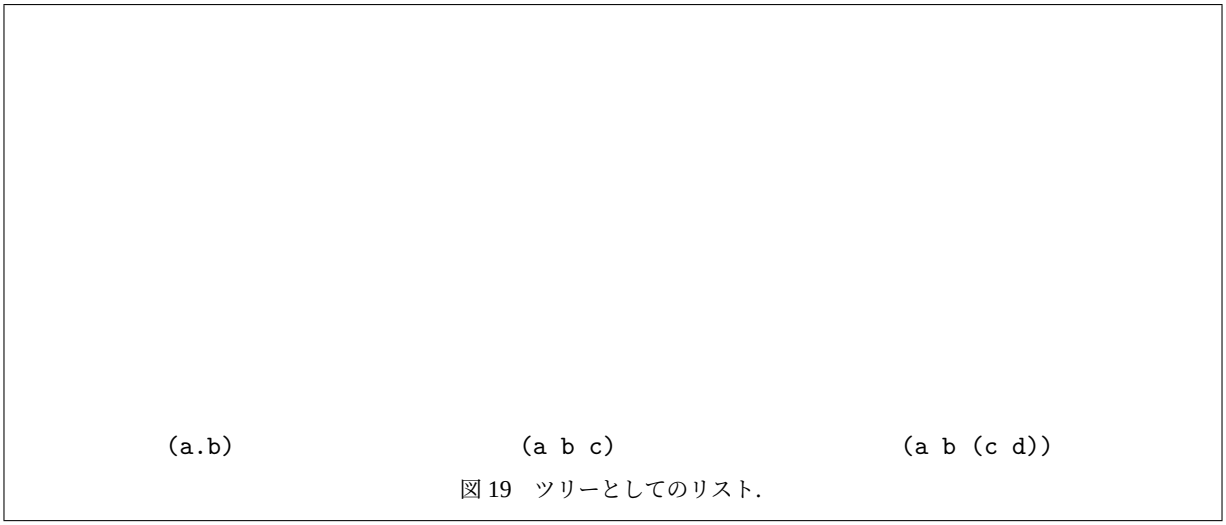

図 19　ツリーとしてのリスト.

4.6　部分ツリーでの再帰

Lisp プログラムによく見られる再帰の形には他のものもある：部分ツリーに対する再帰だ．この形は（おそらく入れ子になった）リストについて，その car 部と cdr 部の両方を再帰的に下っていきたい場合に見られる．

Lisp のリストは多目的な構造体で，連続構造，集合，対応，配列，ツリーを含む色々なものを表現できる．リストをツリーとして解釈する方法は幾つかある．一番よく使われるのは，リストを二分ツリー（binary tree）に，car 部と cdr 部をそれぞれ左右の枝に見なす方法だ．（実際，リストの内部表現は大抵そうだ.）第 19図にはリストとそれが表現するツリーの例を 3 個示した．そういうツリーの内部節点はドット対（dotted-pair）表現したリストの点に対応するので，リストをその形式で考えればツリーの構造は解釈しやすいだろう：

```
(a b c) = (a . (b . (c . nil)))
(a b (c d)) = (a . (b . ((c . (d . nil)) . nil)))
```

どのリストも二分ツリーとして解釈できる．そのため `copy-list` と `copy-tree` 等の Common Lisp の関数の組がある．前者はリストを連続構造としてコピーする．リストが部分リストを含むとき，それは連続構造の要素に過ぎないのでコピーされない：

```
> (setq x '(a b)
        listx (list x 1))
((A B) 1)
> (eq x (car (copy-list listx)))
T
```

それとは対照的に，`copy-tree` はリストをツリーとしてコピーする．部分リストは部分ツリーなので，やはりコピーされる：

```
> (eq x (car (copy-tree listx)))
NIL
```

`copy-tree` は次のように定義できる：

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

この定義は実は広く使われる形の一つだということが分かる．（この後出てくる関数は形をはっきりさせるために少し妙な方法で書かれている.）例えばツリーの葉を数えるユーティリティを考えてみよう：

[*13] 広く使われているある Common Lisp 処理系では，`functionp` は誤って `t` と `nil` に真を返す．その処理系では `lrec` の第 2 引数に渡しても機能しないだろう.

[*14] 処理系によっては，これらの関数を表示する前に `*print-circle*` を `t` に設定しなければならないかもしれない.

```
(defun ttrav (rec &optional (base #'identity))
  (labels ((self (tree)
                 (if (atom tree)
                     (if (functionp base)
                         (funcall base tree)
                         base)
                     (funcall rec (self (car tree))
                              (if (cdr tree)
                                  (self (cdr tree)))))))
    #'self))
```

図 20　ツリーに対する再帰関数.

```
(defun count-leaves (tree)
  (if (atom tree)
      1
      (+ (count-leaves (car tree))
         (or (if (cdr tree) (count-leaves (cdr tree)))
             1))))
```

ツリーの持つ葉の数は，リストとして表現されたときに見えるアトムの数より多い：

```
> (count-leaves '((a b (c d)) (e) f))
10
```

ツリーの持つ葉とは，ツリーをドット対表現で見たときに見えるアトムの全てだ．ドット対表現では ((a b (c d)) (e) f) は 4 個の nil を含むが，リスト表現ではそれらは見えない（括弧 1 組につき 1 個あるのだが）．だから count-leaves は 10 を返す.

前の章では，ツリーに作用するユーティリティを幾つか定義した．例えば flatten（joo ページ）はツリーを引数に取り，その中の全てのアトムをリストに括って返す．つまり flatten に入れ子になったリストを渡すと，一番外側以外の括弧が無くなっただけのようなリストが返る：

```
> (flatten '((a b (c d)) (e) f ()))
(A B C D E F)
```

この関数は（かなり非効率的だが）次のようにも定義できる：

```
(defun flatten (tree)
  (if (atom tree)
      (mklist tree)
      (nconc (flatten (car tree))
             (if (cdr tree) (flatten (cdr tree))))))
```

最後に find-if の再帰版であり，平坦なリストだけでなくツリーにも作用できる rfind-if について考えよう：

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (if (cdr tree) (rfind-if fn (cdr tree))))))
```

find-if をツリーにも作用するように一般化するには，葉のみを検索するのか，それとも部分ツリー全体を検索するのかを決めなければならない．ここでの rfind-if は前者を採ることにする．つまり呼び出し側は第 1 引数として渡された関数はアトムにのみ適用されると考えてよい：

```
> (rfind-if (fint #'numberp #'addp) '(2 (3 4) 5))
3
```

copy-tree，count-leaves，flatten に rfind-if といった 4 個の関数の構造は，何と似ていることだろう！実際，それらはみな部分ツリーに作用する関数の典型的な例だ．cdr 部での再帰もそうだが，こういうものの原型を曖昧なままに放っておく事はない——そのインスタンスを生成する関数を書くことができる.

原型そのものを掴むため，それらの関数をよく見て何が定型でないのかに注目しよう．our-copy-tree は本質的には 2 個の事実を表す：

```
; our-copy-tree
(ttrav #'cons)
; count-leaves
(ttrav #'(lambda (l r) (+ l (or r 1))) 1)
; flatten
(ttrav #'nconc #'mklist)
```

図 21　ttrav で表現された関数.

```
(defun trec (rec &optional (base #'identity))
  (labels
    ((self (tree)
           (if (atom tree)
               (if (functionp base)
                   (funcall base tree)
                   base)
               (funcall rec tree
                        #'(lambda ()
                            (self (car tree)))
                        #'(lambda ()
                            (if (cdr tree)
                                (self (cdr tree))))))))
    #'self))
```

図 22　ツリーに対して再帰を行うための関数.

1. 再帰の基底では引数をそのまま返す.
2. 再帰の途中では，左の部分ツリー（car 部）での再帰結果と右の部分ツリー（cdr 部）での再帰結果に cons を適用する.

そのため，これは 2 個の引数を取る関数生成関数の呼び出しとして表現できる：

```
(ttrav #'cons #'identity)
```

ttrav (“tree traverser”) の定義は第 20図に示した．再帰呼び出しのときは渡す値は 1 個でなく 2 個で，それぞれ左の部分ツリーと右の部分ツリーに対して再帰を行う．再帰の基底が関数のときは，その関数はその時点での葉に対して呼ばれる．平坦リストに対する再帰では基底の値は必ず nil だが，ツリーに対する再帰では基底の値は有用であることがあり，それを使いたくなることもあるかもしれない.

ttrav を使うと，これまで説明した関数は rfind-if 以外みな表現できる．（第 21図に示した.）rfind-if を定義するには，いつ，どの場合に再帰呼び出しが行われるかを制御させてくれるような，さらに一般的なツリー用再帰関数生成関数が要る．ttrav の第 1 引数には，再帰呼び出しの結果を取る関数を与えた．一般的な状況においては，代わりに，再帰呼び出しそのものを代表する 2 個のクロージャを取る関数を使う．するとツリーの任意の深さまでのみを探索する再帰関数が書ける.

ttrav によって生成された関数は必ずツリー全体を探索する．count-leaves や flatten はとにかくツリー全体を探索せざるを得ないのでそれでいいが，rfind-if には探す対象が見つかった時点で探索を止めて欲しい．それにはさらに一般的な trec（第 20図）を使わなければならない．trec の第 2 引数は関数で，再帰途中の時点でのオブジェクトと 2 個の再帰関数という 3 個の引数を取る．それらの再帰関数はそれぞれ左右の部分ツリーに対する再帰を代表するクロージャだ．trec を使うと flatten はこのように定義できる：

```
(trec #'(lambda (o l r) (nconc (funcall l) (funcall r)))
      #'mklist)
```

以上より rfind-if は oddp によって以下のように表現できる：

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

4.7 いつ関数を作るべきか

残念なことに関数を表現するのに関数生成関数を呼ぶのは，シャープクォート付きのλ式を呼ぶのに比べ，実行時に不必要な負荷がかかる．シャープクォートの付いたλ式は固定されたデータだが，生成関数の呼び出しは実行時に評価される．本当に実行時に生成関数を呼ばなければならないなら，それを使ってもよいことはない．しかし，少なくともある場合には生成関数を予め呼ぶことができる．リードマクロ `#.`（シャープ・ドット，シャープ点）を使うと，ソース読み込み時に新しい関数を生成できる．その式が読み込まれたときに `compose` とその引数が定義されている限り，例えば

```
(find-if #.(compose #'oddp #'truncate) lst)
```

とすることができる．これなら `compose` の呼び出しは Lisp リーダによって評価され，生成された関数は固定的データとしてコード内に挿入される．`oddp` と `truncate` は共に組み込み関数なので，`compose` の定義が既に読み込まれている限り，`compose` の呼び出しは読み込み時に評価されると思っていいだろう．

一般的に言って，関数の合成と組み合わせはマクロを使えば一層簡単かつ効率的に実現できる．関数の名前空間が分離している Common Lisp では特に効果が顕著だ．マクロの導入後，第 15 章で，ここでの話題の大半をもっと豪華な器を使って扱う．

5 表現としての関数

一般的に言って，データ構造は何かを表現するために使われる．配列は幾何学的変換を表現できる．ツリーは命令のヒエラルキーを表現できる．グラフは鉄道ネットワークを表現できる．Lisp ではクロージャが表現手段に使われることがある．クロージャ内では変数束縛は情報を蓄えることができるし，複雑なデータ構造を構築するときのポインタの役目を果たすこともできる．束縛を共有するかまたは互いを参照できるクロージャのグループを作ることで，データ構造とプログラムの長所を併せ持ったハイブリッド・オブジェクトを創れる．

その内部では共有された束縛がポインタになっている．クロージャにはポインタを高度に抽象化して扱う利点がある．静的なデータ構造で表現する筈の物を表現するクロージャを使うことで，しばしば優雅さと効率が大きく向上するのを期待できる．

5.1 ネットワーク

クロージャには 3 個の便利な性質がある．アクティブで，ローカルな状態を保持していて，複数のインスタンスを造れることだ．アクティブでローカルな状態を持ったオブジェクトの複数のコピーを使うべき所とはどこだろう？何よりもネットワークに関わるアプリケーションが挙げられる．多くの場合ネットワークの節点はクロージャとして表現できる．クロージャはそれ自身のローカルな状態を保持できるだけでなく，別のクロージャを参照できる．だからネットワーク内の節点を表現するクロージャは，出力を送るべき幾つかの節点（クロージャ）の状態を理解できる．これはある種のネットワークならそのままコードに変換できるということだ．

この章と次章では，ネットワークを探索する方法を二つ検討する．まず，構造体として定義された節点と，ネットワークを探索する別のコードを使う伝統的な方法を追ってみる．そして次章で，同じプログラムを単一の抽象化手法で作る方法を示す．

例として，可能な中で最も単純な応用方法を使う．「20 の質問」を行う類のあれだ．ここで使うネットワークは二分ツリーで，葉でない節点は二択式の質問を保持し，質問への答えに応じて右か左の部分ツリーへ下って探索して行く．葉の節点は返り値を保持する．探索が葉に到達すると，その値は探索結果として返される．このプログラムの実行結果は第 23図に示したようになる．

伝統的な方法は節点を表現するための何らかのデータ構造を定義することだろう．節点は幾つかの情報を持たなければならない：それが葉であるかどうか．そうなら返す値は何か．そうでないなら質問は何で，答えに応じてどちらへ進むか．それを実現するのに十分なデータ構造を第 24図に示した．それは最小限の構造を持っている．`contents` 欄は質問または返り値を保持する．節点が葉でないなら，`yes` 欄と `no` 欄で答えに応じてどの節点に進むかを指定する．葉であるなら，それらが空でなのでそのことが分かる．`*nodes*` はグローバルなハッシュ表で，これに節点名の一覧を載

```
> (run-node 'people)
Is the person a man?
>> yes
Is he living?
>> no
Was he American?
>> yes
Is he on a coin?
>> yes
Is the coin a penny?
>> yes
LINCOLN
```

図 23　「20 の質問」の実行例.

```
(defstruct node contents yes no)

(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (make-node :contents conts
                   :yes yes
                   :no no)))
```

図 24　節点の表現と定義.

```
(defnode 'people "Is the person a man?" 'male 'female)
(defnode 'male "Is he living?" 'liveman 'deadman)
(defnode 'deadman "Was he American?" 'us 'them)
(defnode 'us "Is he on a coin?" 'coin 'cidence)
(defnode 'coin "Is the coin a penny?" 'penny 'coins)
(defnode 'penny 'lincoln)
```

図 25　ネットワークの例.

```
(defun run-node (name)
  (let ((n (gethash name *nodes*)))
    (cond ((node-yes n)
           (format t "~A~%>> " (node-contents n))
           (case (read)
             (yes (run-node (node-yes n)))
             (t (run-node (node-no n)))))
          (t (node-contents n)))))
```

図 26　ネットワーク探索のための関数.

せる．最後に defnode は新しい節点（葉であってもそうでなくてもいい）を創り，それを *nodes* に蓄える．これらの素材を使ってツリーの最初の節点が定義できる：

```
(defnode 'people "Is the person a man?"
         'male 'female)
```

第 25図には，第 23図のやりとりを実現するのに必要なだけのネットワークを示した.

そうなれば必要なのはこのネットワークを探索し，質問を表示し，示された経路を辿る関数を書くことだけだ．その関数 run-node は第 26図に示した．名前が与えられると，対応する節点を探す．それが葉でないなら contents 欄の質問を行い，答えに応じて二つの可能な行き先の中から選んで探索を続ける．節点が葉であるなら，run-node は contents 欄の内容を返すだけだ．この関数は第 25図で定義されたネットワークを使い，第 23図に示した出力を行う.

```
(defvar *nodes* (make-hash-table))

(defun defnode (name conts &optional yes no)
  (setf (gethash name *nodes*)
        (if yes
            #'(lambda ()
                (format t "~A~%>> " conts)
                (case (read)
                  (yes (funcall (gethash yes *nodes*)))
                  (t (funcall (gethash no *nodes*)))))
            #'(lambda () conts))))
```

図 27　クロージャへとコンパイルされるネットワーク.

5.2　ネットワークのコンパイル

前章で書いたネットワーク・プログラムは，どのプログラミング言語でも書けるようなものだった．あのプログラムは全く単純なもので，他の書き方があるなどとはなかなか思えない．しかしそれがあるのだ——実際，遥かに単純に書き直せる．

第 27図に示したコードはその点を説明している．これがあのネットワークを機能させるのに必要な全てだ．節点をデータ構造として表し，それを探索する別の関数を創る代わりに，ここでは節点をクロージャとして表現した．前の例では構造体の中に蓄えられていたデータはクロージャ内の変数束縛に蓄えられている．節点自身の中に run-node の機能が組み込まれているので，もうそれは必要ない．探索を始めるには，始めたい節点で funcall を呼ぶだけだ：

```
(funcall (gethash 'people *nodes*))
Is the person a man?
>>
```

それから先の実行結果は前の実装法と全く変わりない.

節点をクロージャとして表現することで，「20 の質問」で使うネットワーク全体をコードに変換できた訳だ．見た通り，上のコードでは実行時に名前によって節点のクロージャを探さなければならない．しかしネットワークが実行中に再定義されることがないと分かっているなら，更に拡張を加えられる．節点のクロージャがハッシュ表を介さず，子を直接呼ぶようにすることができる.

第 28図には新しい定義を示した．ここでは *nodes* はハッシュ表ではなく捨ててもいいリストだ．節点は前と同じく defnode で定義するが，この時点ではクロージャは生成されない．全ての節点の定義後，compile-net を呼んでネットワーク全体を一度にコンパイルする．この関数は再帰的にツリーの葉まで下り，戻りながら各節点で二つの部分ツリーそれぞれに対応する節点／クロージャを返す[*15]．だから全ての節点は二つの子の名前を把握しているだけなのではなく，それらを直に制御できる．compile-net の元の呼び出しが返ると，ネットワークのコンパイルした部分に対応する関数が得られる.

```
> (setq n (compile-net 'people))
#<Compiled-Function BF3C06>
> (funcall n)
Is the person a man?
>>
```

compile-net は二つの意味で「コンパイル」を行うことに注意しよう．まずネットワークの抽象的表現をコードに変換するという一般的な意味のコンパイルを行う．さらに compile-net 自身が処理系にコンパイルされていれば，返される関数もコンパイルされたものだ.（wowow ページを参照.）

ネットワークをコンパイルした後は defnode の作ったリストはもう要らないので，捨ててしまって（例えば *nodes* に nil を代入する）ガーベジ・コレクタに任せればよい.

[*15] この実装法ではネットワークがツリーだと仮定している．この章での実装法ではそうでなければならない.

```
(defvar *nodes* nil)

(defun defnode (&rest args)
  (push args *nodes*)
  args)

(defun compile-net (root)
  (let ((node (assoc root *nodes*)))
    (if (null node)
        nil
        (let ((conts (second node))
              (yes (third node))
              (no (fourth node)))
          (if yes
              (let ((yes-fn (compile-net yes))
                    (no-fn (compile-net no)))
                #'(lambda ()
                    (format t "~A~%>> " conts)
                    (funcall (if (eq (read) 'yes)
                                 yes-fn
                                 no-fn))))
              #'(lambda () conts))))))
```

図 28　静的参照によるコンパイル.

5.3　前を向く

ネットワークに関わる多くのプログラムが，節点をクロージャにコンパイルすることで実装できる．クロージャはデータ・オブジェクトであり，構造体が表現できるものを表現するのに使える．そうするには見かけたことのない思考法が幾らか要るが，見返りには高速で優雅なプログラムが得られる．

マクロはクロージャをデータ表現に使うときに大きな助けになる．「クロージャで表現する」と言うのは「コンパイルする」を言い直したに過ぎない．マクロはコンパイル時に仕事をこなすのだから，この技法にとって全く自然な手段なのだ．マクロの導入後，第 23, 24 章で，ここで使った戦略に基づいたずっと大規模なプログラムを提示する．

6　マクロ

Lisp のマクロ機能を使うと，コードの変換によってオペレータを実装することができる．マクロ定義とは実質的に Lisp コードを生成する関数だ——つまりプログラムを書くプログラムだ．この一歩から始めて，大きな可能性と予期せぬ危険に足を踏み入れていく．第 7 章から第 10 章はマクロの基礎講座だ．この章ではマクロがどのように動作するのかを説明し，マクロを書いたり動作を確かめるときのテクニックを示し，マクロを書くスタイルについての話題に進む.

6.1　マクロはどのように動作するか

マクロは呼び出せるし値を返せるので，関数と一緒くたにされがちだ．マクロ定義は関数定義に似ていることもあるし，実際はマクロである do を普段「組み込み関数」と呼ぶ人も多い．しかしこの喩えを突き詰めすぎると混乱の元になる．マクロの動作は普通の関数とは違っている．マクロはどのように，そしてなぜ違うのかを知ることは，マクロを正しく使うための鍵だ．関数は**結果**を生むが，マクロは**式**を生む——そしてこの式が評価されると結果を生む.

習い始めるのに一番良い方法は，実例に当たることだ．ここでは引数を nil に設定するマクロ `nil!`を書きたいとしよう．(`nil!  x`) としたとき (`setq x nil`) と同じ効果が欲しいのだ．それには `nil!`を，ある形を持った式を別の形の式に変えるマクロとして定義する.

```
> (defmacro nil! (var)
    (list 'setq var nil))
NIL!
```

人間語に直せば，この定義は Lisp に「(nil! var) の形の式を見たら，必ず評価前に (setq var nil) の形に変えること.」と命じている.

マクロの生成した式は元のマクロ呼び出しの場所で評価される．マクロ呼び出しとは第 1 要素がマクロの名前であるリストだ．マクロ呼び出し (nil! x) をトップレベルに打ち込んだら何が起きるだろうか？ Lisp は nil!がマクロの名前であることに気付き，

1. 与えた定義の指定した式を生成し，
2. その式を元のマクロ呼び出しの場所で評価する．

新しい式を生成する工程はマクロ展開と呼ばれる．Lisp は nil!の定義を探すが，そこにはマクロ呼び出しを置き換える式の生成法が載っている．nil!の定義が，マクロ呼び出しの引数として与えられた式に対して関数のように適用される．それは 3 個の要素（setq，マクロに引数として与えられた式，nil）から成るリストを返す．この場合 nil!の引数は x だから，マクロ展開の結果は (setq x nil) だ.

マクロ展開の後，第 2 工程すなわち評価が行われる．Lisp は展開結果 (setq x nil) を，始めの場所に直接書いてあったかのように評価する．必ずしもトップレベルのときのように評価が展開直後に行われる訳ではない．関数定義内のマクロ呼び出しは関数がコンパイルされたときに展開されるが，展開形（またはそれに基づくオブジェクト・コード）はその関数が呼び出されるまで評価されない.

マクロに関して出会う問題の多くは，マクロ展開と評価をきちんと区別することで避けられる．マクロを書くときは，展開時と評価時に，それぞれどのような動作をするのかしっかり理解すること．なぜなら二つの工程は一般的に別の種類のオブジェクトに対して作用するからだ：マクロ展開は式を扱い，評価はその値を扱う.

マクロ展開は nil!の場合より複雑になることがある．nil!の展開形は組み込み特殊オペレータの呼び出しだったが，展開形が別のマクロ呼び出しになることもある．ちょうどロシア人形の中に別の人形が入っているようなものだ．そのようなとき，マクロ展開はマクロ呼び出しでない式に到達するまで続く．この過程が最終的に終わるまでには，任意の長さの段階を要する.

多くのプログラミング言語は何らかの種類のマクロを備えているが，Lisp のマクロは並外れて強力だ．Lisp コードのファイルがコンパイルされるとき，パーサがソースを読んで出力をコンパイラに送る．大事なのは次だ．パーサの出力は Lisp のオブジェクトのリストから成る．マクロを使えば，プログラムがパーサとコンパイラの間の中間形式のときに操作することができる．必要ならばその操作で Lisp の拡張も行える．展開形を生み出すマクロは Lisp の力の全てを握っている．それはマクロが生まれ持った性質だ．実際，マクロは Lisp の関数だ——たまたま式を返すだけのことだ．nil!の定義では list が使われただけだったが，別のマクロでは展開時に大きなサブルーチンを呼び出すこともある.

コンパイラが読む情報を書き換えられるというのはコンパイラを書き換えられるのとほとんど同じだ．既存のデータ構造を変形することで定義できるようなデータ構造なら何でも Lisp に追加できる.

6.2 逆クォート

逆クォート（バッククォート，backquote）は特別なクォートで，Lisp の式の雛形を作るのに使える．それが一番よく使われるのはマクロ定義の中だ.

逆クォート ‘ は，普通のクォート ’ を逆さにした形なのでそう呼ばれる．逆クォートが式の前に付いているだけのときは，働きはクォートと変わらない：

‘(a b c) は ’(a b c) と等価である.

逆クォートが便利なのは，カンマ , やカンマ・アット ,@ と組み合わせて使うときだ．逆クォートがリストの雛形になるとき，カンマはその中に値を挿入する．逆クォート付きリストは，各要素にクォートを付けて list を呼ぶのと等価だ．つまり

‘(a b c) は (list ’a ’b ’c) と等価である.

逆クォートのスコープ内では，カンマは Lisp に「評価の保護を止めろ」と命じる．カンマがリストのある要素の前に

付くと，そこに仮想的に付けられているクォートの効果を打ち消す．だから

‘(a ,b c ,d) は (list ’a b ’c d) と等価である．

シンボル b の代わりにその値が挿入されたリストが作られる．どれ程深く入れ子になったリスト内でもカンマは機能する：

```
> (setq a 1 b 2 c 3)
3
> ‘(a ,b c)
(A 2 C)
> ‘(a (,b c))
(A (2 C))
```

またカンマの前にクォートが付いてもよいし，クォート付きの部分リスト内で使ってもよい：

```
> ‘(a b ,c (’,(+ a b c)) (+ a b) ’c ’((,a ,b)))
(A B 3 (’6) (+ A B) ’C ’((1 2)))
```

1 個のカンマは 1 個の逆クォートの効果を打ち消すのだから，カンマは逆クォートと対応が取れなければならない．ある特定のオペレータがカンマの前に付くか，カンマを含む式の前に付いているとき，「カンマはそのオペレータに囲まれている」と言うことにしよう．例えば‘(,a ,(b ‘,c))) では，最後のカンマはカンマ 1 個と逆クォート 2 個に囲まれている．一般的に言うと：n 個のカンマに囲まれたカンマは最低 $n+1$ 個の逆クォートに囲まれていなければならない．それから明らかに分かるのは，カンマは逆クォートの付いた式の外側にあってはならないということだ．逆クォートとカンマは，上の規則に従う限り入れ子になってもよい．次の式はトップレベルに打ち込んだらどれもエラーになる：

```
,x   ‘(a ,,b c)   ‘(a ,(b ,c) d)   ‘(,,‘a)
```

逆クォートの中の逆クォートは，マクロを定義するマクロ位でしか使われない．どちらの点についても第 16 章で議論する．

逆クォートは大抵リストを作るのに使われる[*16]．逆クォートの作るリストは list と普通のクォートを使って作ることもできる．逆クォートを使うことの利点は，逆クォート付きの式はそれの生成する式と似ているので式を読み易くできることだ．前章で nil!を次のように定義した：

```
(defmacro nil! (var)
  (list ’setq var nil))
```

逆クォートを使うと，同じマクロが次のように定義できる：

```
(defmacro nil! (var)
  ‘(setq ,var nil))
```

この場合では違いはあまりないが，マクロ定義が長くなれば逆クォートを使う重要性は増す．第 29には，数の符号について if 分岐を行うマクロ nif の定義の候補を二つ示した[*17]．

第 1 引数は評価されると数にならなければならない．そしてその数か正か 0 か負かに応じてそれぞれ第 2，3，4 引数が評価される：

```
> (mapcar #’(lambda (x)
              (nif x ’p ’z ’n))
          ’(0 2.5 -8))
(Z P N)
```

第 29図の二つのマクロ定義は同じマクロを定義している．ただ 1 番目は逆クォートを使っているが，2 番目では list を陽に呼ぶことで展開形を作っている．最初の定義からは，例えば (nif x ’p ’z ’n) が

```
(case (truncate (signum x))
  (1 ’p)
  (0 ’z)
  (-1 ’n))
```

[*16] 逆クォートはベクタを作るのにも使えるが，マクロ定義内では滅多にそういうことはしない．

[*17] このマクロ定義は gensym の使用を避けるために変なやり方をしている．よい定義方法は zip ページで示す．

逆クォートを使う：

```
(defmacro nif (expr pos zero neg)
  `(case (truncate (signum ,expr))
     (1 ,pos)
     (0 ,zero)
     (-1 ,neg)))
```

逆クォートを使わない：

```
(defmacro nif (expr pos zero neg)
  (list 'case
        (list 'truncate (list 'signum expr))
        (list 1 pos)
        (list 0 zero)
        (list -1 neg)))
```

図 29　マクロ定義に逆クォートを使った例と使わない例.

に展開されるのはすぐ分かる．マクロ定義本体の形が，生成される展開形にそっくりだからだ．逆クォートのない 2 番目の定義を理解するには，展開の過程を頭の中で追っていかなければならない.

カンマ・アット ,@ はカンマの変種で，機能はカンマと同じだが違いが 1 点ある：次に続く式の値をカンマのようにそのまま挿入するのでなく，カンマ・アットは切り張り操作を行う．つまり 1 番外側の括弧を取り除いて挿入する：

```
> (setq b '(1 2 3))
(1 2 3)
> `(a ,b c)
(A (1 2 3) C)
> `(a ,@b c)
(A 1 2 3 C)
```

カンマはリスト (1 2 3) を b の所に挿入するが，カンマ・アットはそこにリストの要素を挿入する．カンマ・アットを使うには幾つか制限がある：

1. 引数に切り張り操作を行うので，カンマ・アットは連続構造内になければならない．`,@b 等とするのは誤りだ．b の値を張り付ける先がないからだ.
2. 切り張りされるオブジェクトは，最後に来ない限りリストでなければならない．式 `(a ,@1) が評価されると (a . 1) となるが，`(a ,@1 b) 等として，アトムをリストの中に切り張りしようとするとエラーになる.

カンマ・アットが使われるのは，不定個の引数を取り，やはり不定個の引数を取る関数やマクロに渡すマクロの中が多い．普通この状況は暗黙のブロックを実装するときに現れる．Common Lisp には block，tagbody，progn 等，コードをブロックにまとめるオペレータが幾つかある．これらのオペレータはソース・コード内には滅多に出て来ない．それらは暗黙に使われる——つまり，マクロに隠されている.

暗黙のブロックは式を実行本体として持てる組み込みマクロでは必ず使われている．例えば let と cond には暗黙の progn が使われている．そんな中で最も単純な組み込みマクロは when だろう：

```
(when (eligible obj)
  (do-this)
  (do-that)
  obj)
```

(eligible obj) が真を返すとき残りの式が評価され，when が式として返す値はそのうち最後の式の値だ．コンマ・アットを使った例として，ここに when の定義方法の一例を示す：

```
(defmacro our-when (test &body body)
  `(if ,test
       (progn
         ,@body)))
```

この定義は &body 仮引数を使っているが，これは表示整形 (pretty-printing) の効果を除いて &rest と同じで，任意個の引数が取れるようにしている．またそれらを式 progn の中に切り張りするためにカンマ・アットを使っている．上の呼び出しでの展開形では，本体の 3 個の式は 1 個の progn の中に現れる：

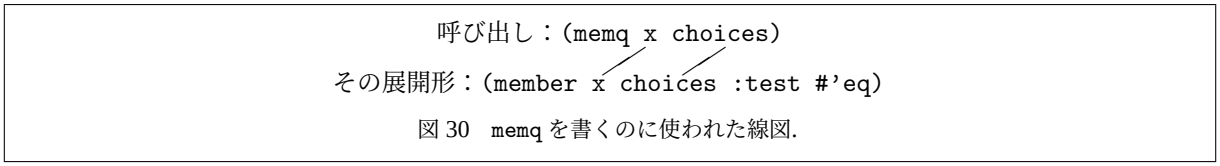

図 30　memq を書くのに使われた線図.

```
(if (eligible obj)
    (progn (do-this)
           (do-that)
           obj))
```

繰り返しを行うマクロの大半は似た方法で引数を切り張りする.

カンマ・アットの効果は逆クォートを使わなくても実現できる．例えば式 `‘(a ,@b c)` は `(cons 'a (append b (list 'c)))` と等価だ．カンマ・アットの存在理由は上のように式を生成する式を読み易くすることだけだ.

マクロ定義は（普通）リストを生成する．マクロ展開は関数リストを使っても作れるが，逆クォートを使ったリストの雛形は仕事を随分楽にしてくれる．defmacro と逆クォートで定義したマクロは，表面的には defun で定義した関数に似ている．この類似性にミスリードされない限り，逆クォートによってマクロを書くのも読むのも楽になる.

逆クォートはマクロ定義の中で非常によく使われるので，defmacro の一部だと思っている人もいる．逆クォートについて忘れてはいけないことの最後は，それがマクロ内での役割とは別に自分自身の生き方があるということだ．連続構造を生成する必要のあるときには，いつでも逆クォートが使える：

```
(defun greet (name)
  `(hello ,name))
```

6.3　単純なマクロの定義

プログラミングの学習では，できるだけ早く実験することが一番の方法であることが多い．理論的に完全に理解するのは後でもいい．それに従って，この章ではマクロをすぐに書き始める方法を提示する．その方法が使える範囲は広くはないが，可能なところでは機械的に適用できる.（もうマクロを書いた経験があるのなら，この章は飛ばしてもよい.）

例として Common Lisp の組み込み関数 member の変種を書く方法について考える．普通 member は 2 個のオブジェクトが等しいかどうか調べるのに eql を使う．eq を使って所属関係を調べたいときには，わざわざ指定しなければならない：

```
(member x choices :test #'eq)
```

これを何度も繰り返すことがあるなら，必ず eq を使う member の変種を書いてもいいだろう．幾つかの昔の Lisp 方言にはそんな関数があり，memq と呼ばれていた：

```
(memq x choices)
```

普通 memq はインライン関数として定義されるが，マクロの例を示すため，ここではマクロとして復活させる.

さて方法とは：まず定義したいマクロの典型的な呼び出し方から始める．メモ用紙にそれを書き，その下にそれが展開されてできる筈の式を書く．第 30図にそのような 2 個の式の例を示した．マクロ呼び出しに従い，これから書くマクロの仮引数リストを作る．引数それぞれについて適当な仮引数名を考える．この場合引数が 2 個あるので仮引数を 2 個使う．それらを obj と lst と呼ぼう：

```
(defmacro memq (obj lst)
```

そして先程書いた 2 個の式に戻る．マクロ呼び出しの中の引数それぞれから，下の展開形の中で使われている場所まで線を引く．第 30図には平行な 2 本の線が引いてある．マクロ本体を書くには展開形に注意を向けなければいけない．まず逆クォートから本体を書き始める．次に展開形を式毎に区切って読んでいく．マクロ呼び出しの引数の一部でない括弧を見つけたら，マクロ定義内にも括弧を書く．すると逆クォートの次には左括弧が来る．そうしたら展開形の式それぞれについて

1. マクロ呼び出しと線でつながってなければ，その式そのものを書く.
2. マクロ呼び出しの引数の一つとつながっていれば，マクロの仮引数リスト内で対応するシンボルを書き，その前にカンマを付ける.

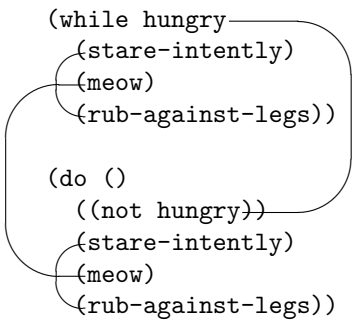

図 31　while を書くのに使われた線図.

第 1 要素 member はどこにもつながっていないので，member そのものを使う：

```
(defmacro memq (obj lst)
  `(member
```

しかし x からは元の式の第 1 引数に線が引いてあるので，マクロ本体の中にカンマを付けた第 1 引数を置く：

```
(defmacro memq (obj lst)
  `(member ,obj
```

この要領で続けていくと次のようなマクロ定義が完成する：

```
(defmacro memq (obj lst)
  `(member ,obj ,lst :test #'eq))
```

これまでのところ，決まった数の引数を取るマクロだけを書いてきた．さて今書きたいのは while だとしよう．これはテスト式と本体となる幾らかの式を引数に取り，テスト式が真を返す限りその式を繰り返し実行する．第 31図に猫の振る舞いを描写した while ループの例を示した.

そのようなマクロを書くには，先程の方法を少し修正する必要がある．まず前と同じようにマクロ呼び出しの例を書くことから始める．それからマクロの仮引数リストを書くのだが，任意個の引数を取りたいところでは &rest または &body の次に仮引数を置く：

```
(defmacro while (test &body body)
```

さて展開される筈の形をマクロ呼び出しの下に書き，前と同じようにマクロ呼び出しの引数を展開形内の場所までつなぐ線を引こう．しかし幾つかの引数が単一の&rest または&body 仮引数として詰め込まれるときは，それらを塊として扱い，引く線はそれら全体に対して 1 本にする．結果の線図は第 31図に示した.

そしてマクロ定義の本体を書くには展開形を参考に前と同じことをするが，先程の 2 個に加えもう 1 個の規則が必要だ：

3. 展開形内の幾つかの式からマクロ呼び出し内の幾つかの引数へつながる線があれば，対応する &rest（または &body）仮引数を書き，その前にカンマ・アットを置く.

結局マクロ定義は以下のようになる：

```
(defmacro while (test &body body)
  `(do ()
     ((not ,test))
     ,@body))
```

式を本体として持つマクロを書くには，幾つかの仮引数が漏斗として働く必要がある．ここではマクロ呼び出し内の複数の引数が本体にまとめられ，本体が展開形の中に切り張りされるときに再び分解されている.

この章で説明した手法を使えば，最も単純なマクロ——仮引数を並び替えるだけのもの——を書くことができる．しかしマクロの力はそんなものではない．第 7.7 節では，展開形がただの逆クォート付きリストでは表現できないような例を示す．そういうマクロを展開するにはマクロ自身がプログラムでなければならない.

```
> (defmacro while (test &body body)
    `(do ()
       ((not ,test))
       ,@body))
WHILE
> (pprint (macroexpand '(while (able) (laugh))))
(BLOCK NIL
       (LET NIL
            (TAGBODY
              #:G61
              (IF (NOT (ABLE)) (RETURN NIL))
              (LAUGH)
              (GO #:G61))))
T
> (pprint (macroexpand-1 '(while (able) (laugh))))
(DO NIL
    ((NOT (ABLE)))
    (LAUGH))
T
```

図 32　1 個のマクロとその 2 段階の展開.

```
(defmacro mac (expr)
  `(pprint (macroexpand-1 ',expr)))
```

図 33　マクロ展開確認用のマクロ.

6.4　マクロ展開の確認

マクロを書くのが終わったら，どうやってその動作を確かめよう？ memq のようなマクロは単純だから，ちょっと見ればその動作が分かる．更に複雑なマクロを書くときは，それが正しく展開されるのか確認できなければならない．

第 32図には，1 個のマクロ定義とその 2 通りの展開形を示した．組み込み関数 macroexpand は指揮を引数に取り，それをマクロ展開したものを返す．マクロ呼び出しを macroexpand に渡すと，それが評価前に最終的にどのように展開されるのかが分かる．しかし完全な展開形はマクロの確認では必ずしも望ましいものではない．問題のマクロが別のマクロを使っていると展開が進みすぎてしまうので，完全な展開形は読みづらいことがある．

第 32図の 1 番目の式からは，while が意図した通りに展開されたかどうかは判断し辛い．組み込みマクロの do が prog に展開され，さらにそれも展開されているからだ．ここで必要なのは 1 段階だけ展開した結果を見る方法だ．それが 2 番目の例に示した組み込み関数 macroexpand-1 の目的だ．macroexpand-1 は 1 段階だけマクロを展開すると，それが依然としてマクロ呼び出しであっても動作を止める．

マクロ呼び出しの展開形を見たいとき，いつも

```
(pprint (macroexpand-1 '(or x y)))
```

と打ち込むのは馬鹿らしい．第 33図の新しいマクロを使うなら，代わりにこうすればよい：

```
(mac (or x y))
```

デバッグを進めるときの定石は，関数は呼び出すこと，マクロは展開することだ．しかしマクロ呼び出しは 2 層に渡る命令が関わるので，誤りの起きる場所も 2 箇所ある．マクロの動作がおかしいときは，大抵は展開形を見るだけで間違いが見つかる．しかし，希に展開形には問題ないように見えて，問題がどこで起きるのかを見るためにそれを評価したいときもある．展開形にフリー変数が含まれるときは，先にそれらを設定したいこともあるだろう．処理系によっては，展開形をコピーしてトップレベルに貼り付けたり，展開形を選択してメニューから eval を選んで評価させることができる．どうしても問題が掴めないときは，適当な変数を macroexpand-1 の返すリストに設定し，それに対して eval を呼ぶこと：

```
> (setq exp (macroexpand-1 '(memq 'a '(a b c))))
(MEMBER (QUOTE A) (QUOTE (A B C)) :TEST (FUNCTION EQ))
```

```
> (eval exp)
(A B C)
```

最後に，マクロ展開は単なるデバッグの補助手段ではなく，マクロの書き方の勉強手段でもあることを言っておきたい．Common Lisp には 100 以上の組み込みマクロがあり，中には大変複雑なものもある．そんなマクロの展開形を見ることで，それらがどう書かれたのかが分かることも多い．

6.5 引数リストの構造化代入

構造化代入 (destructuring) とは，関数呼び出しによって行われる一種の代入[*18]の一般化だ．幾つかの引数を取る関数を定義したとき

```
(defun foo (x y z)
  (+ x y z))
```

その関数が呼ばれると

```
(foo 1 2 3)
```

関数の仮引数には順番に従って引数が代入される：x には 1，y には 2，そして z には 3 という風に．構造化代入とは，このような引数の場所に従って行われる代入が，任意のリスト構造に対して (x y z) と同様に行われることだ．

Common Lisp のマクロ destructuring-bind（CLtL2 で新しく導入された）は，パターン，リストに評価される引数及び本体となる式を引数に取り，パターン内の仮引数をリスト内の対応する要素に束縛した状態で式を評価する：

```
> (destructuring-bind (x (y) . z) '(a (b) c d)
    (list x y z))
(A B (C D))
```

この新オペレータとその仲間が第 18 章の主題になる．

構造化代入はマクロの仮引数リストでも使える．Common Lisp の defmacro では仮引数リストは任意のリスト構造であってよい．マクロ呼び出しが展開されたとき，マクロ呼び出しの components はマクロの仮引数に destructuring-bind と同様に代入される．組み込みマクロ dolist は仮引数リストの構造化代入を活用している．次のようなマクロ呼び出しでは：

```
(dolist (x '(a b c))
  (print x))
```

展開されて生まれる関数は，第 1 引数として与えられたリストの中から x と'(a b c) を取り出さなければならない．それは dolist に適切な仮引数リストを与えることで暗黙の中に実現された[*19]：

```
(defmacro our-dolist ((var list &optional result) &body body)
  `(progn
     (mapc #'(lambda (,var) ,@body)
           ,list)
     (let ((,var nil))
       ,result)))
```

Common Lisp では，普通 dolist のようなマクロは実行本体でない引数をリスト内にまとめ込んでいる．dolist はそれが返す結果になる付加的な引数を取るので，何にせよ 1 種類目の引数を独立したリストにまとめなれけばならない．しかし余分なリスト構造が必要でなくても，それによって dolist の呼び出しが読み易くなる．ここで when に似たマクロ when-bind を定義したいとしよう．ただしこれは調べる式の返す値に何かの変数を束縛するものとする．このマクロは入れ子になった仮引数リストを使うと一番上手く実装できるだろう：

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))
```

これは次のように呼び出せばよく，

[*18] 構造化代入は大抵，代入を行うのでなく束縛を作るオペレータで見られる．しかし概念的には構造化代入は値の代入の一手法であって，新しい変数だけでなく既存の変数に対しても機能する．つまり構造化代入型 setq を書いて悪いことはない．

[*19] 後に導入する gensym の使用を避けるため，ここではおかしな方法で定義している．

```
(defmacro our-expander (name) `(get ,name 'expander))

(defmacro our-defmacro (name parms &body body)
  (let ((g (gensym)))
    `(progn
       (setf (our-expander ',name)
             #'(lambda (,g)
                 (block ,name
                        (destructuring-bind ,parms (cdr ,g)
                          ,@body))))
       ',name)))

(defun our-macroexpand-1 (expr)
  (if (and (consp expr) (our-expander (car expr)))
      (funcall (our-expander (car expr)) expr)
      expr))
```

図 34　defmacro の概形.

```
(when-bind (input (get-user-input))
           (process input))
```

こうしなくてもいい：

```
(let ((input (get-user-input)))
  (when input
    (process input)))
```

Used sparingly, 仮引数リストの構造化代入は明確なコードにつながる．少なくとも，2 個以上の引数と本体となる式を取る when-bind や dolist 等のマクロで使える．

6.6　マクロのモデル

マクロが何をするのかを形式的に説明しても，長く，ややこしくなるだけだろう．経験を積んだ Lisp プログラマの頭の中にもそんな説明がある訳ではない．defmacro の定義方法を想像することで，それが何をするのかを覚えた方が便利だ．

Lisp にはそのような説明を使う長い伝統がある．1962 年初版の *The Lisp 1.5 Programmer's Manual*† も，参考のために Lisp で書かれた eval の定義を載せている．defmacro 自身がマクロなので，同じ扱いができる．定義は第 34図に示した．そこではまだ扱っていない技法を幾つか使っているので，参考にするのは後にしたいと思われる方もいるだろう．

第 34図に示した定義は，マクロが何をするのかについてかなり正確な感覚を与えてくれるが，飽くまでも概形なので不完全な所もある．これは &whole キーワードを適切に扱えない．また defmacro が第 1 引数にマクロ関数として実際に保持させるものは 2 引数の関数で，マクロ呼び出しと，それが呼び出されたレキシカル環境を引数に取る．しかしそれらの性質が使われるのは最高度に難解なマクロだけだ．マクロが第 34図のように実装されていると思っていれば，間違うことはまず無いだろう．例えばこの本の中で定義されたマクロはそれを使っても全て正しく機能する．

第 34図の定義が生み出す展開形の関数は，シャープクォート付きの λ 式だ．つまりそれはクロージャである筈だ．マクロ定義内のどんなフリーシンボルも defmacro の呼び出された環境内の変数を参照できる．だから次のようにすることができる筈だ：

```
(let ((op 'setq))
  (defmacro our-setq (var val)
    (list op var val)))
```

CLtL2 では，確かに可能だ．しかし CLtL1 では，マクロ展開関数は空レキシカル環境で定義されていたので[*20]，古い処理系では上の our-setq は動作しないかもしれない．

[*20] この区別が問題になるマクロの例については，第 pom ページの注を参照.

```
(do ((w 3)
     (x 1 (1+ x))
     (y 2 (1+ y))
     (z))
  ((> x 10) (princ z) y)
  (princ x)
  (princ y))
```

これは次のようなものに展開されて欲しい：

```
(prog ((w 3) (x 1) (y 2) (z nil))
      foo
      (if (> x 10)
          (return (progn (princ z) y)))
      (princ x)
      (princ y)
      (psetq x (1+ x) y (1+ y))
      (go foo))
```

図 35　do の望ましい展開方法.

6.7　プログラムとしてのマクロ

マクロ定義は必ずしも単なる逆クォート付きリストでなくてもよい．マクロはある種の式を別の形に変形する関数だ．その関数は結果を生成するために list を呼んでもいいが，数百行のコードから成る副プログラムを丸ごと呼んでもいい.

第 7.3 節では，マクロを簡単に書く方法を示した．その方法によって，展開形がマクロ呼び出しの中と同じ式を含むマクロを書くことができる．しかし残念なことに，その条件を満たすのは一番単純な種類のマクロだけだ．複雑なマクロの例として，組み込みマクロ do について考えてみよう．do は引数を並び替えるだけのマクロとして定義することは不可能だ．その展開時には，マクロ呼び出しには決して出て来ない複雑な式を生成しなければならない.

マクロを書くときの更に一般的な手法とは，使えるようにしたい式の種類と，それがどのように展開されるかについて考え，使いたい式を展開形に変形するプログラムを書くことだ．マクロの例を手で展開してみて，ある形が変形されるときには何が起きているのかを見ると良い．例に倣うことで，自分の求めるマクロには何が必要なのかを掴むことができる.

第 35図に示したのは，do の使用例と，それが展開されて生まれる筈の式だ．自分の手で展開してみるのはマクロがどのように働くべきかを明確にするのに良い方法だ．例えば，展開形を手で書かないと，ローカル変数の更新に psetq を使わなければならないことは明らかではないだろう.

組み込みマクロ psetq（“parallel setq” つまり「並列 setq」）の動作は setq と似ているが，全ての（偶数番目の）引数がどの代入よりも前に評価される点が違う．普通の setq が 2 個より多い引数を取るとき，第 1 引数の新しい値は第 4 引数の評価中に見えている：

```
> (let ((a 1))
    (setq a 2 b a)
    (list a b))
(2 2)
```

ここでは a が最初に設定されたので，b は a の新しい値の 2 を得た．psetq はその引数が並列的に代入されたかのように機能するものだ：

```
> (let ((a 1))
(psetq a 2 b a)
(list a b))
(2 1)
```

そのため，ここでは b は a の古い値を得ている．マクロ psetq は，do 等，引数の幾つかを並列的に評価する必要のあるマクロの補助のために特に提供されたものだ．（setq を使っていたら，do* の定義になっていた.）

展開形を見れば，foo をループ用ラベルに使うのは本当はまずいことも明らかだ．foo が do の実行本体のループ用

```
(defmacro our-do (bindforms (test &rest result) &body body)
  (let ((label (gensym)))
    `(prog ,(make-initforms bindforms)
           ,label
           (if ,test
               (return (progn ,@result)))
           ,@body
           (psetq ,@(make-stepforms bindforms))
           (go ,label))))

(defun make-initforms (bindforms)
  (mapcar #'(lambda (b)
              (if (consp b)
                  (list (car b) (cadr b))
                  (list b nil)))
          bindforms))

(defun make-stepforms (bindforms)
  (mapcan #'(lambda (b)
              (if (and (consp b) (third b))
                  (list (car b) (third b))
                  nil))
          bindforms))
```

図 36　do の実装例.

ラベルに使われていたらどうするのか？第 9 章で，この問題を詳細に取り扱う：今の所は，マクロ展開では foo ではなく，関数 gensym の返す特別な無名シンボルを使わなければならない，と覚えておけばよい.

do を書くには，第 35図の上の式を下の形に変形するには何が必要かを考えることになる．そのような変形を行うには，逆クォート付きリストの正しい位置にマクロの仮引数を入れるだけでは済まない．先頭の prog の次のリストはシンボルとその初期値から成るが，それらは do の第 2 引数の中から抽出しなければならない．第 36図の make-initforms はそんなリストを返す．それから psetq のために引数のリストを作らなければならないが，更新すべきシンボルは全部ではないので，こちらの処理は複雑になる．第 36図の make-stepforms は，psetq に渡す引数を返す．これら 2 個の関数を使うと，残りの定義には込み入った技は要らない.

第 36図のコードは，実際の処理系で do に使われている定義ではない．展開中に行われる処理を強調するため，make-initforms と make-stepforms は独立した関数として分けておいた．いずれそんなコードも大抵は defmacro 内に残すようになるだろう.

このマクロの定義により，マクロに何ができるのかが見えてきた．マクロは Lisp の式を作る能力を最大限に生かしている．マクロの展開形を作るコードはそれ自身がプログラムであっても良いのだ.

6.8　マクロのスタイル

良いスタイルはマクロについては意味が違ってくる．スタイルが問題になるのはコードが人間に読まれるか，Lisp に評価されるときだ．マクロの場合，その両方が普通とは少し違った状況で行われる.

マクロ定義に関わるコードには種類の異なる 2 個がある：展開コード，つまりマクロが展開形を生成するために使うコードと，被展開コード，つまり展開形そのものに現れるコードだ．スタイルの原則はそれぞれ違っている．一般的に言って，プログラムの良いスタイルとは明確で効率的であることだ．これらの原則は 2 種類のマクロのコードにおいては別々の方向に向かって強調される：展開コードは効率より明確さを優先し，被展開コードは明確さより効率を優先する.

効率性が一番問題になるのはコンパイル済みコードである訳だが，コンパイル済みコードの中ではマクロ呼び出しは既に展開済みだ．展開コードが効率的ならコンパイルが幾分早く済むだろうが，プログラムの動作の効率には一切違いはない．マクロ呼び出しの展開はコンパイラの仕事のうち僅かの部分に過ぎないことが多いので，効率的に展開される

```
(defmacro our-and (&rest args)
  (case (length args)
    (0 t)
    (1 (car args))
    (t `(if ,(car args)
            (our-and ,@(cdr args))))))

(defmacro our-andb (&rest args)
  (if (null args)
      t
      (labels ((expander (rest)
                         (if (cdr rest)
                             `(if ,(car rest)
                                  ,(expander (cdr rest)))
                             (car rest))))
        (expander args))))
```

図 37　and と等価なマクロの例 2 個.

マクロというのも普通はコンパイル速度にも大した影響を及ぼさない．だから大抵，展開コードはプログラムの初期ヴァージョンを手早く書く感じで書くことができる．展開コードが不必要な処理をしたりコンシングを多く起こしたからといって，何だと言うのか？プログラマの時間はプログラムの別の場所を改良するのに費やした方が賢い．展開コードにおいて明確さと効率のどちらかを選ぶことがあったら，明確さを優先すべきだ．マクロ定義は一般的に関数定義よりも読み下しづらいが，それは別々の二つの時点で評価される式が混じっているからだ．展開コードの効率性の代わりにこの混乱が減らせるのなら，お得な取引と言える．

例えば一種の and をマクロとして定義したいとしよう．(and a b c) は (if a (if b c)) と等価だから，第 37図の上の定義のように，and は if を利用して書ける．普通のコードを審査する基準からすれば，our-and の定義はまずい．展開コードが再帰的で，再帰のそれぞれの段階で同じリストの連続した cdr 部の長さを求めている．このコードが実行時に評価されるなら，このマクロは同じ展開を一切無駄なく行う our-andb のように定義する方が良いだろう．しかしマクロ定義としては our-and は（勝っているとは言えないが）同程度の出来だ．再帰のそれぞれの段階で length を呼ぶのは非効率的だろうが，その構成は展開形が条件の数に依存する様子をはっきり表している．

何でもそうだが，例外もある．Lisp ではコンパイル時間と実行時間の区別は恣意的なもので，その区別に基づく法則はどれもやはり恣意的なものだ．プログラムによっては，コンパイル時間が実行時間なのだ．プログラムの主目的が何らかの形式の変換で，マクロをそのために利用しているなら，話は全く違ってくる：展開コードがプログラムの中心に，展開形がその出力になる，もちろんそんな状況では展開コードは効率性を念頭に置いて書かれるべきだ．しかし大抵の展開コードは（ア）コンパイル速度にのみ影響し，（イ）その影響も大したことはない——つまり明確さがほぼ必ず一番重要だ．

被展開コードについては，全く逆になる．マクロの展開形は（特に他人には）読まれることはまず無いので，明確さは大した問題ではない．禁断の goto も展開形では必ずしも禁じられておらず，評判の悪い setq もそれ程嫌がられない．

構造化プログラミングの推進者達が goto を憎むのは，それがソース・コードに及ぼした影響のせいだ．彼らが有害だと見なすのはマシン語のジャンプ命令ではない——ソース・コード内で抽象的構造に隠されている限りは問題にならないのだ．goto が Lisp で非難の対象になっている理由は，楽に隠蔽できるからに他ならない：代わりに do を使えるし，do が備わっていなくとも自分で書くことができる．もちろん新しい抽象的構造を goto の上に構築するなら，どこかに goto が存在せざるを得ない．だから新しいマクロの定義に go を使うのは，既存のマクロが代わりに使えないときなら，必ずしも悪いことではない．

似たように setq も，変数がどこで値を得たのかが判り辛くなるので白い目で見られる．しかしマクロ展開は多くの人が読むものではないので，普通はマクロ展開の中で創られた変数に setq を使うことに害は少ない．幾つかの組み込みマクロの展開形を見てみれば，setq が山程あるだろう．

状況によっては展開形コードでの明確さが一層大事になることがある．複雑なマクロを書くときは（少なくともデ

バッグ中に）結局展開形を読む羽目になる．また単純なマクロでも，展開コードと被展開コードの区別は単なる逆クォートなので，そんなマクロが格好悪い展開形を生成するのなら，ソース・コード内でも格好悪さがはっきり見えてしまうだろう．しかし被展開コードの明確さが問題になるときでも，効率性が依然として優先するべきだ．プログラム実行時に評価される大部分のコードの中では，効率性が重要なのだ．マクロ展開では二つの理由からそれが際立つ：マクロがどこででも使われるせいと，マクロが不可視なせいだ．マクロはしばしば一般用途のユーティリティを実装するのに使われるが，それらはプログラムのあらゆる所で呼び出されるものだ．それ程頻繁に使われるものは非効率的なままで放って置く訳にはいかない．無害で小さなマクロに見えるものが，全て展開された後には，プログラム内で無視できない分量を占める．そんなマクロには，長さから判断して必要と思えるよりも多くの注意が必要だ．特にコンシングを避けること．不必要なコンシングを起こすユーティリティは，それさえなければ効率的なプログラムの動作を鈍化させることにもなりかねない．

展開形コードの効率性に目を向ける理由のもう一つは，それが非常に見え辛い点だ．関数の実装方法が下手でも，その関数がプログラマの目に入る度にその事実は意識される．しかしマクロはそうではない．マクロ定義からは展開形の効率の悪さは明らかでないこともある．展開形コードの非効率性は明らかにならないかも知れないので，一層注意して見るべきなのだ．

6.9 マクロへの依存

関数定義を書き直すと，それを呼び出す関数は自動的に更新後の関数を使うようになる[*21]．ただしマクロについては必ずしも同じようには行かない．関数定義内にあるマクロ呼び出しは関数がコンパイルされるときに展開形に置換される．マクロを呼び出す関数がコンパイルされた後にそのマクロを再定義したらどうなるだろうか？元々のマクロ呼び出しの形跡は残っていないので，関数内の展開形は更新されない．関数の動作は古いマクロ定義を反映し続けることになる：

```
> (defmacro mac (x) `(1+ ,x))
MAC
> (setq fn (compile nil `(lambda (y) (mac y))))
#<Compiled-Function BF7E7E>
> (defmacro mac (x) `(+ ,x 100))
MAC
> (funcall fn 1)
2
```

マクロを呼ぶコードがマクロそのものの定義前にコンパイルされると似た問題が起こる．CLtL2 には「マクロ定義は，マクロの最初の使用よりも先にコンパイラに読まれていなければならない」とある．幸運なことにどちらの問題も避けるのは容易だ．以下の二つの原則を忘れなければ，古かったり存在しないマクロ定義について思い悩むことはない：

1. マクロはそれを呼び出す関数（またはマクロ）の前で定義する．
2. マクロが再定義されたときは，それを（直接または間接的に）呼び出している関数とマクロを全て再コンパイルする．

マクロ定義が確かに最初にコンパイルされるようにするため，一つのプログラム内で使われるマクロを全て独立したファイルに分ける方法がこれまで提案されてきた．しかしそれはやり過ぎだ．`while` のような汎用マクロを独立したファイルに分けるのは理に適っているが，汎用ユーティリティは関数であれマクロであれ，プログラムの他の部分とは分けておく方がよい．

マクロにはプログラムのある特定の部分のためだけに書かれたものもあるので，それらは使う側のコードと一緒に定義されるべきだ．マクロ定義が呼び出しのどれよりも前に現れる限りコンパイルは上手く行く．マクロは関数と違うからというだけの理由で全て一緒にしておいても，コードが読み辛くなるだけのことだ．

[*21] インラインでコンパイルされた関数は別で，再定義に関してマクロと同様の制限を負っている．

6.10 関数からマクロへ

この章では関数をマクロに翻訳する方法を扱う．最初の 1 歩は，本当にその必要があるかどうか自問することだ．関数をインライン宣言するだけで済むのでないだろうか (p. 26)？

しかし関数をマクロに翻訳する方法を考えることにはしっかりした理由がある．マクロを書きはじめると，関数を書いているような気がしてくることがある．普通このアプローチでは完璧なマクロは書けないが，少なくともこの感じは手がかりになる．マクロと関数の関係を理解しておきたい理由には，他にも，それらの違いを理解したいというのもある．最後に，Lisp プログラマにはとにかく関数をマクロに書き換えたいことが時々あるものなのだ．

関数をマクロに翻訳することの難しさは，その関数の持つ性質の数による．最も翻訳が簡単な部類の関数は

1. 本体が単一の式から成っていて
2. 仮引数リストには仮引数の名前のみが含まれ
3. （仮引数以外に）新しい変数を創らず
4. 再帰的でなく（また相互再帰的な関数のグループにも含まれず）
5. 本体内で複数回使われる仮引数を持たず
6. 仮引数リスト内で先に現れる仮引数の値より，後に現れる仮引数の値の方が本体内で先に使われることがなく
7. フリー変数を含まない

ものだ．これらの原則を満たす関数の例は Common Lisp の組み込み関数 `second` で，これはリストの第 2 要素を返す．その関数はこう定義すればよい：

```
(defun second (x) (cadr x))
```

この場合関数定義は上の条件を全て満たしているので，等価なマクロ定義に翻訳するのは簡単だ．逆クォートを関数本体の前に付け，仮引数リストに含まれていたシンボルが出て来る度にコンマを付けるだけでよい：

```
(defmacro second (x) `(cadr ,x))
```

もちろんこのマクロは関数とまったく同じ条件で呼び出すことはできない．`apply` や `funcall` の第 1 引数としては渡せないし，呼出側の関数が新たなローカルな束縛を生むような環境では使うべきでない．それでも普通のインライン呼び出しでは，マクロ `second` は関数 `second` と同じ動作をするはずだ．

マクロは単一の式に展開されなければならないので，本体に複数の式が含まれるときには少し違う方法を使う．条件 1 が満たされないときには `progn` を追加すればよい．つまり関数 `noisy-second` は：

```
(defun noisy-second (x)
  (princ "Someone is taking a cadr!")
  (cadr x))
```

次のマクロ定義として翻訳できる：

```
(defmacro noisy-second (x)
  `(progn
     (princ "Someone is taking a cadr!")
     (cadr ,x)))
```

関数が`&rest` または`&body` 仮引数を使っていて条件 2 が満たされないときは，仮引数に関してだけ方法を変える．ただコンマを前に付けるのではなく，仮引数の中身が `list` の呼び出しの中に切り貼りされなければならない．よって

```
(defun sum (&rest args)
  (apply #'+ args))
```

は

```
(defmacro sum (&rest args)
  `(apply #'+ (list ,@args)))
```

となるが，この場合は次のようにした方がよい：

```
(defmacro sum (&rest args)
  `(+ ,@args))
```

条件 3 が満たされないとき——関数本体内で新しい変数が創られるとき——は，コンマを付ける方法を修正しなければならない．仮引数リストに含まれるシンボル全ての前にコンマを付けるのではなく，仮引数を参照しているものにのみコンマを付ける．例えば次の例では

```
(defun foo (x y z)
  (list x (let ((x y))
            (list x z))))
```

x の内後ろの二つは仮引数の x を参照していない．2 番目の x は一切評価の対象にならず，3 番目は let の創り出した新しい変数を参照している．だから 1 番目にだけコンマを付ける：

```
(defmacro foo (x y z)
  `(list ,x (let ((x ,y))
              (list x ,z))))
```

条件 4〜6 が満たされなくとも関数をマクロに翻訳できることもあるが，これらの内容は後で独立して扱う．マクロにおける再帰については第 10.4 章で，評価の重複と評価順の混乱の危険性についてはそれぞれ第 10.1 章と 10.2 章で扱う．

条件 7 については，xien ページで触れたエラーに似た技を使ってクロージャをマクロで真似ることができる．しかしこれは程度の低い技であり，この本の基調に馴染まないので，詳しく扱うことはしない．

6.11 シンボル・マクロ

CLtL2 で Common Lisp に新種のマクロが導入された——シンボル・マクロだ．普通のマクロ呼び出しの見掛けが関数呼び出しに似ているのに対して，シンボル・マクロの「呼び出し」の見掛けはシンボルに似ている．

シンボル・マクロはローカルにのみ定義できる．スペシャル式 symbol-macrolet を使うと，その本体内では唯のシンボルが式のように振る舞うようになる：

```
> (symbol-macrolet ((hi (progn (print "Howdy")
                               1)))
                   (+ hi 2))
"Howdy"
3
```

symbol-macrolet の本体は引数の場所にある hi が全て (progn (print "Howdy") 1) で置き換えられたかのように評価される．

概念的には，シンボル・マクロは引数をとらないマクロに似ている．引数がなければ，マクロは見掛け上の省略形に過ぎない．しかしシンボル・マクロが無益だと言いたいのではない．それは第 15 章と第 18 章で使われているが，後者では必要不可欠な役目を持っている．

7 いつマクロを使うべきか

ある関数が，本当にマクロであるよりも関数であった方が良いというのは，どうしたら分かるだろうか？マクロが必要な場合とそうでない場合の間には，大抵明確な区別がある．基本的には関数を使うべきだ：関数で間に合う所にマクロを使うのはエレガントでない．マクロが何か特定の利益を与えるときのみそれを使うべきなのだ．

マクロが利益を与えるときとはいつだろうか？それがこの章の主題だ．普通この疑問は，改良を目指すときでなく必要に迫られた上で発せられる．マクロで実現できることの大部分は，関数ではできない．第 8.1 節ではマクロとしてのみ実装できるオペレータの種類を列挙する．だが境界線上に立つような場合もわずかにある（しかしこれは興味深い）．つまりオペレータを関数で書いてもマクロで書いてもよいような状況だ．そのような状況について，第 8.2 節ではマクロについての賛否両論を挙げる．最後に，マクロに何ができるのかを考慮した上で，第 8.3 節では関連した疑問について考える．人はマクロでどんなことを行うものなのだろうか？

7.1 他に何も関係しないとき

プログラム内の数個所に似通ったコードが表れているとき，サブルーチンを書き，それらをサブルーチン呼出しで置き換えるのは，よいデザインのための一般原則だ．この原則を Lisp プログラムに適用するとき，「サブルーチン」を関数にすべきかマクロにすべきか決めなければならない．

関数ではなくマクロを書こうと用意に決心できることがある．マクロのみが必要なことを実現できるときだ．1+ のような関数は，関数としてもマクロとしても書けるかもしれない．

```
(defun 1+ (x) (+ 1 x))

(defmacro 1+ (x) `(+ 1 ,x))
```

しかし第 7.3 節の `while` は，マクロでないと定義できない：

```
(defmacro while (test &body body)
  `(do ()
       ((not ,test))
     ,@body))
```

このマクロの動作を関数で真似る方法はない．`while` の定義では，本体として渡された式を `do` の本体内に張り込むようになっているが，これはテスト式が `nil` を返したときのみ評価される．どんな関数にもそれは無理だ：関数呼び出しでは，全ての引数は関数の呼び出しよりも前に評価されるからだ．

確かにマクロが必要というとき，マクロに何を求めているのだろうか？マクロにできて関数にできないことは 2 つある：マクロは引数の評価を制御（または抑制）でき，また呼び出し側のソースコードそのものへと展開される．結局，マクロが必要なアプリケーションには，これらの性質の片方または両方を必要なのだ．

簡単に説明するときの「マクロは引数を評価しない」というのは少し間違っていて，こう言うと正確になる．マクロはマクロ呼び出し内の引数の評価を制御する．評価回数は引数がマクロの展開コードのどこに置かれるかによるが，1 回でも複数回でも良いし，全く評価しないこともある．マクロでのこの制御の使われ方は主に次の 4 通りだ：

1. 変形．Common Lisp のマクロ `setf` は引数を評価前に取り出す類のものだ．組み込みのアクセス関数には，しばしば逆の動作を行う関数がある．その目的はアクセス関数が取ってきたものを設定することだ．`car` については `rplaca`，`ofcdr`，`rplacd` 等が当てはまる。`setf` を使うと，そういったアクセス関数を設定されるべき変数と同じように呼び出せる．例えば `(setf (car x) 'a)` は `(progn (rplaca x 'a) 'a)` に展開され得る．
   この技を実現するには，`setf` はその第 1 引数の内部を見なければならない．上の場合で `rplaca` が必要だと知るためには，`setf` には第 1 引数が `car` で始まる式であることが見えなければならない．よって `setf` を始めとする，引数の変形を行うオペレータは，どれもマクロとして書かなければならない．
2. 変数束縛．レキシカル変数はソースコード内に直接書かれなければならない．例えば `setq` の第 1 引数は評価されないので、`setq` の上に構築されたものは全て `setq` へと展開されるマクロでなければならない．`setq` を呼び出す関数ではだめだ．それは引数が λ 式の仮引数として現れる `let` などのオペレータや，`lets` へと展開される `do` 等のマクロについても同様だ。引数のレキシカルな束縛を変更するための新オペレータは，いずれもマクロとして書かれなければならない。
3. 条件分岐による評価．関数の引数は全て評価される．`when` のような制御構文では，ある引数は特定の条件下でのみ評価される．そのような柔軟性はマクロでのみ得られる．
4. 複数回の評価．関数の引数は全て評価されるだけでなく，きっかり 1 回評価される．マクロは `do` のような制御構文の定義に使われるが，この場合，ある引数は繰り返し評価されなければならない．

マクロのインライン展開を活用する方法もいくつかある。そこで展開形は例えばマクロ呼び出しのレキシカルコンテキスト内に現れることを強調したい．なぜなら 3 通りのうち 2 通りのマクロがその事実に依拠しているからだ．すなわち、

5. 呼び出し側環境を利用する．マクロは，呼び出し側のコンテキストに束縛された変数を含む展開形を生成できる．下のマクロの動作は，`foo` が呼ばれた場所での `y` の束縛に依存する．

   ```
   (defmacro foo (x)
     `(+ ,x y))
   ```

   普通この類のレキシカルな結び付きは喜びの源というよりも伝染病の感染源として見なされる．通常，そのようなマクロを書くのは悪いスタイルだ．関数プログラミングの理想はマクロにも同様に適用される：マクロは引数を通じて相互作用することが望ましい．実際，呼出側環境を利用することは滅多に必要にならず，そのような状況が起きたとしても，それは大抵誤りによるものだ（第 9 章を参照）．この本の中の全てのマクロのうち，継続を渡すマクロ（第 20 章）と ATN コンパイラ（第 23 章）のみがその方法で呼び出し側環境を利用している．

6. 新しい環境を包み込む．マクロは引数を新たなレキシカル環境で評価させることもできる．古典的な例は `let` で，`lambda` の上にマクロを被せることで実装できる（penpen ページ）．`(let ((y 2)) (+ x y))` 等の式の実行部本体の中では，`y` は新しい変数を指すことになる．
7. 関数呼び出しを節約する．マクロの展開結果をインライン挿入することの 3 つ目の結果は，コンパイル後のコードではマクロ呼び出しに関わるオーヴァヘッドが一切ないことだ．実行時には，マクロ呼び出しは展開形に置き換え済みになっている．(原則的にはインライン宣言された関数でも同じことだが)

驚いたことに第 5 番目と第 6 番目の用法は，意図せずに使われたときは，変数捕捉の問題を引き起こす．これはマクロを書くときに恐らく一番恐ろしいことだ．変数捕捉については第 9 章で論じる．

マクロの利用法には 7 通りあると言う代わりに，6 通り半あると言った方がよいだろう．理想の世界では全ての Common Lisp コンパイラがインライン宣言に従い，関数呼び出しの節約はマクロではなくインライン関数の仕事になっていることだろう．理想の世界は読者への課題として残しておく．

7.2 マクロと関数どちらがよい？

前節では判断の容易なケースを扱った。引数に、評価前にアクセスするオペレータは全てマクロで書かなければならない。他に選択肢はないからだ。それではマクロでも関数でも書けるようなオペレータはどうだろうか？例えば引数の平均値を返すオペレータ `avg` を考えてみよう．次のように関数として書くことができる．

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))
```

しかし次のようにマクロとして書いた方がよい．

```
(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

関数版 `avg` は呼ばれる度に `length` を呼ぶが，それは不必要だ．コンパイル時には引数の値までは不明かもしれないが，それが幾つあるかは分かる．よって `length` の呼び出しはそのときに済ませた方がよい．似たような選択を迫られたときに考慮すべき点は以下の通りだ．

マクロの長所

1. コンパイル時の計算．マクロ呼び出しには，2 つの時点での処理が関わってくる．マクロの展開時と，展開形の評価時だ．Lisp プログラムでのマクロ展開は全てプログラムがコンパイルされたときに行われ，コンパイル時に実行できるどのような計算も，実行時にプログラムを遅くすることはない．オペレータが仕事の一部をマクロ展開の段階で済ませるように書けるなら，マクロとして書く方が効率がよいだろう．賢いコンパイラでもできないようなことがあれば，結局実行時に関数がこなすことになるのだから．第 13 章では `avg` のように展開時に仕事を行うマクロについて説明する．
2. Lisp との緊密な連携．関数ではなくマクロを使うと，プログラムを Lisp と緊密に連携させられることがある．ある問題を解くプログラムを書く代わりに，マクロを使ってその問題を Lisp が既に解法を知っているような問題へ変形できるときがある．このアプローチが可能なときは，大抵プログラムは小さくかつ効率的になる：Lisp が処理を代わりに行ってくれるので小さくなり，Lisp 処理系の製品は一般的にユーザのプログラムより無駄を削いであるから効率がよくなる，という訳だ．この長所は主に埋め込み言語で明らかになるが，それについての説明は第 19 章から始まる．
3. 関数呼び出しの節約．マクロ呼び出しは，書かれた所に直接展開される．だからよく使われるコードのまとまりをマクロとして定義しておけば，それが使われる度に関数を呼び出さなくともよい．Lisp の古い方言では，プログラマは実行時の関数呼び出しを節約するために，マクロのこの性質を活用していた．Common Lisp では，この仕事はインライン宣言された関数が代わりに行うものとされている．
   関数をインライン宣言することで，コンパイルされたら呼出側コードの中に埋め込まれるよう指示できる．マクロと全く同じだ．しかし，ここでは理想と現実のギャップが出てくる．CLtL2 (p. 229) には「コンパイラがこの宣言を無視するのはフリーである」とあり，幾つかの Common Lisp コンパイラは確かに無視している．そのようなコンパイラを使わざるを得ないのなら，関数呼び出しの節約にマクロを使うのも認められるだろう．

効率と Lisp との緊密な連携との長所が相俟って，マクロを使おうという考えが強い説得力を持つ場合がある．第 19 章のクエリ・コンパイラでは，コンパイル時に移せる計算の量がとても大きいので，プログラム全体を 1 つの巨大なマクロに変えてしまってよいといえる．この変換はスピードのために行われたのだが，プログラムを Lisp と緊密にすることにもつながっている．新しいヴァージョンではクエリの中で数値評価などの Lisp の式を簡単に使えるのだ．

マクロの短所

1. 関数はデータだが，マクロはコンパイラへの指示に近い．関数は（例えば `apply` に）引数として渡すことも，関数から返すことも，データ構造内に格納することもできる．マクロではそれらはどれも不可能だ．これらはマクロ呼び出しを λ 式で包むことで実現できる場合がある．この方法は例えばマクロに `apply` や `funcall` を適用したいときに使える．

   ```
   > (funcall #'(lambda (x y) (avg x y)) 1 3)
   2
   ```

   しかしこの方法は不便だし，必ず使えるというものではない．`avg` のように，マクロに`&rest` パラメータがあっても，それに可変個の引数を渡す方法はない．
2. ソースコードの明確さ．マクロ定義は，同等な関数の定義より読み辛くなりがちだ．だから何かをマクロとして書いてもプログラムが大して進歩する訳でもないなら，代わりに関数を使った方がよい．
3. 実行時の明確さ．マクロは関数よりデバッグし辛いことがある．多数のマクロ呼び出しを含むコードで実行時エラーに出くわしても，`backtrace` で見えるコードはそれら全てのマクロ呼び出しの展開形からなり，元々書いたコードとは似ても似つかないかもしれない．

   マクロは展開時に消えてしまうので，実行時には動作の説明がつかない．普通は `trace` でマクロが呼び出される様子を見ることはできない．たとえ使っても，`trace` はマクロ呼び出しそのものではなく，マクロを展開する関数の呼び出しを表示するだろう．
4. 再帰．マクロ内で再帰を使うのは，関数のときほど簡単でない．マクロ展開の結果の関数は再帰的でもよいが，展開そのものは再帰的であってはならないからだ．第 10.4 章でマクロにおける再帰を扱う．

いつマクロを使うかを決める際には，これらの考慮のバランスを取らなければならない．どちらが有利かを教えてくれるのは経験のみだ．しかしこの後の章のマクロの例は，マクロが便利な状況を大体カヴァーしている．考え中のマクロがここでの例に似ているなら，そのように書いても大丈夫だろう．

最後に言いたいのは，実行時の明確さ（ポイント 6）はほとんど問題にならないということだ．大量のマクロを含むコードのデバッグは，思った程難しくないだろう．マクロ定義が数百行に及ぶのなら，実行時にその展開形をデバッグするのは嫌になることだ．しかし少なくともユーティリティは，小さく信頼できる層として書かれることが多い．一般的には定義は 15 行以下になる．だからもし `backtrace` を通してコードとにらめっこする羽目になっても，そのようなマクロが目をひどく曇らせることはないだろう．

7.3 マクロの応用例

マクロで何ができるのかを考えた上で，次に浮かぶ質問はこうだ：マクロをどのような応用に使えばよいのだろう？ マクロの利用についての一般的な説明に一番近いのは，マクロは主に構文変換に使われると言うことだろう．マクロの適用範囲が制限されていると言いたいのではない．Lisp プログラムはリストから作られるが[*22]，リストは Lisp の持つデータ構造であり，実際「構文変換」は大変奥が深い．第 19-24 章には，構文変換が目的であると言ってよいプログラムの全体を載せた．そしてそれは，結局は全てマクロである．

マクロの応用は，`while` のような小さな汎用マクロと，この後の章で定義する大規模で特別な用途を持ったマクロとの間の連続体を形成する．端点の片方はユーティリティ，つまりどの Lisp 処理系も組み込みで持っているマクロの類似品だ．それは普通小さく，一般的で，独立して書かれている．しかし特定の種類のプログラムのためのユーティリティを書くこともできる．例えば，グラフィックス・プログラムの中で使うようなマクロが充実してきたときには，そ

[*22] リストがコンパイラへの入力であるという意味で「作られる」ということだ．昔の方言の幾つかとは違い，現在では関数はリストから作られてはいない．

れらはグラフィックスのためのプログラミング言語に似てくるだろう．連続体のもう片方の端点は，Lisp とは明らかに異なった言語でプログラム全体を書けるようにしてくれるマクロだ．この方法で使われるマクロは，埋め込み言語を実装していると言われる．

ユーティリティは，ボトムアップ・スタイルからの帰結の第 1 番目である．多層構造を作るにはプログラムが小さ過ぎるような場合ですら，最下層である Lisp の上に層を加えておくことには利点があるかもしれない．ユーティリティ nil!は引数に nil を代入するものだが，マクロ以外では定義できない．

```
(defmacro nil! (x)
  `(setf ,x nil))
```

nil!を見ると，「何もしてないじゃないか，打ち込む文字を減らしてるだけだ」と言いたくなる．それは本当だが，マクロのやっていることと言ったら，打ち込む文字の節約が全てだ．似たような考え方をしたいなら，コンパイラの仕事はマシン語を打ち込む作業を省くことだと言える．ユーティリティの効果は累積していくので，その価値は見過ごせない．単純なマクロを複数の層に重ねることで，エレガントなプログラムと理解しがたいプログラムとの違いが分かれる．

ほとんどのユーティリティは埋め込まれたパターンだ．自分のコードの中のパターンに気付いたら，それをユーティリティに変えることを考えよう．パターンとはまさにコンピュータの得意分野だ．代わりに仕事をしてくれるプログラムが手に入るというのに，どうして頭を悩ます必要があるのだろうか？ 何かのプログラムを書いているとき，色々な場所で同じ構造を持った do ループを使っていたことに気付いたとしよう．

```
(do ()
  ((not ⟨条件⟩ ))
  ⟨コード本体⟩ )
```

コード内で繰り返されるパターンに気付いたとき，そのパターンにはしばしば名前が付けられる．この場合のパターンは while という名前だ．それを新しいユーティリティ内で提供したいときには，条件評価と複数回の評価が必要なので，マクロを使わなければならない．rurirurl ページにある下の定義を使って while を定義すると，

```
(defmacro while (test &body body)
  `(do ()
     ((not ,test))
     ,@body))
```

上のパターン例は次のように書き換えられる．

```
(while ⟨ 条件 ⟩
  ⟨ コード本体 ⟩ )
```

そうすることでコードは短くなり，更に動作内容がはっきりと示される．

引数を変形できる能力のおかげで，マクロはインタフェイスを書くのに便利だ．適切なマクロを使えば，長く込み入った式が必要な筈のときでも短く簡潔な式が書ける．GUI のせいでエンド・ユーザのためにそのようなマクロを書く必要は減ってきたが，プログラマがこの類のマクロを使うことは減らない．一番馴染深い例は defun だろう．これは関数を束縛するもので，見掛け上は Pascal や C 等の言語の関数定義に似ている．第 2 章では以下の 2 つの式はほぼ同じ効果を持つことに触れた．

```
(defun foo (x) (* x 2))

(setf (symbol-function 'foo)
      #'(lambda (x) (* x 2)))
```

よって defun は前者を後者へ変換するマクロとして実装できる．defun は次のように書かれていると想像できる[*23]．

```
(defmacro our-defun (name parms &body body)
  `(progn
     (setf (symbol-function ',name)
           #'(lambda ,parms (block ,name ,@body)))
     ',name))
```

while や nil!等のマクロは，汎用ユーティリティであると言える．それらはどんな Lisp プログラムで使ってもよい．しかし特定の領域のためのユーティリティを書いてもよい．基盤の Lisp が，拡張のための言語が利用できる唯一のレ

[*23] 明確さのために，この例では defun が行わなければならない細々とした仕事を全て無視している．

```
(defun move-objs (objs dx dy)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (incf (obj-x o) dx)
      (incf (obj-y o) dy))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))

(defun scale-objs (objs factor)
  (multiple-value-bind (x0 y0 x1 y1) (bounds objs)
    (dolist (o objs)
      (setf (obj-dx o) (* (obj-dx o) factor)
            (obj-dy o) (* (obj-dy o) factor)))
    (multiple-value-bind (xa ya xb yb) (bounds objs)
      (redraw (min x0 xa) (min y0 ya)
              (max x1 xb) (max y1 yb)))))
```

図 38　元々の `move` と `scale`.

ベルだと考える理由はない．例えば CAD プログラムを書いているのなら，2 層に分けて書くことが一番の結果を生むだろう．CAD プログラムのための言語（お好みならツールキットという穏やかな言葉でもいいが）と，その層の中で書かれた，ある特定のアプリケーションだ．

他のプログラミング言語では当り前だと思われているあれこれの区別は，Lisp ではぼやけている．他のプログラミング言語では，コンパイル時と実行時，プログラムとデータ，プログラミング言語とプログラムの間には，概念的な区別が確かに存在する．Lisp では，これらの区別は言葉の上での慣習として存在するに過ぎない．例えば，プログラミング言語とプログラムを区切る線は存在しないのだ．問題に適した所に，好きなように手書きの線を引いてよい．背後のコードの層をツールキットと呼ぶかプログラミング言語と呼ぶかは，全く用語の問題に過ぎない．プログラミング言語とみなすことの長所の一つは，Lisp でやっているのと同じように，それをユーティリティで拡張する気になれることだ．

インタラクティブな 2D ドロー・プログラムを書いているとしよう．話を簡単にするため，そのプログラムが扱うオブジェクトは線分のみとする．線分は始点 `x`，`y` とベクタ `dx`，`dy` で表現される．その種のプログラムが対応しなければならない操作に，オブジェクトのグループの平行移動がある．それが第 39図の関数 `move-objs` の役目だ．効率を考え，操作毎にスクリーン全体を再描画することは避けたい——変更のあった部分だけを再描画したい．そのため 2 個ある `bounds` の呼び出しは，オブジェクトのグループを囲む長方形を表す 4 個の座標 $(\min \mathtt{x}, \min \mathtt{y}, \max \mathtt{x}, \max \mathtt{y})$ を返す．`move-objs` の動作部分は，移動前と移動後の長方形をそれぞれ見つけるための `bounds` の呼び出し 2 個に挟まれており，影響を受けた領域全体を再描画する．

関数 `scale-objs` は，オブジェクトのグループの大きさを変えるためのものだ．拡大率に応じてグループを囲む領域は広がったり狭まったりするので，この関数も `bounds` の 2 個の呼び出しの間で動作しなければならない．プログラムを書き進めていけば，このパターンが更に現れるだろう．回転，裏返し，転置等だ．

マクロによってこれらの関数が共通して持つコードを抽象化することができる．第 39図のマクロ `with-redraw` は第 38の関数が共有している骨格を与えている[*24]．その結果，それらの関数はそれぞれ 4 行の長さで定義できた．これら 2 つの関数に利用できたことでその新マクロはすでに簡潔さの点で元が取れたことになる．そしてこれらの関数は，スクリーン再描画の詳細が抽象化されたら，何と明確になったことだろう．

`with-redraw` は，インタラクティブ・ドロー・プログラムを書くためのプログラミング言語の一構造として捉えることができる．そのようなマクロを更に整備していけば，それらは名前の上だけでなく，実際にプログラミング言語に似てくるだろう．そして書いているアプリケーション自身も，その特定の用途のために定義されたプログラミング言語で書けばこうなるだろうか，と思える程エレガントなものに変わっていく．

マクロの主要な利用法のもう一つは，埋め込み言語の実装だ．Lisp はプログラミング言語を書くのに際立って優れ

[*24] このマクロの定義には gensym が使われているので，次章の知識が必要になる．その目的はまもなく説明する．

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    `(let ((,gob ,objs))
       (multiple-value-bind (,x0 ,y0 ,x1 ,y1) (bounds ,gob)
         (dolist (,var ,gob) ,@body)
         (multiple-value-bind (xa ya xb yb) (bounds ,gob)
           (redraw (min ,x0 xa) (min ,y0 ya)
                   (max ,x1 xb) (max ,y1 yb)))))))

(defun move-objs (objs dx dy)
  (with-redraw (o objs)
               (incf (obj-x o) dx)
               (incf (obj-y o) dy)))

(defun scale-objs (objs factor)
  (with-redraw (o objs)
               (setf (obj-dx o) (* (obj-dx o) factor)
                     (obj-dy o) (* (obj-dy o) factor))))
```

図 39　filleted された `move` と `scale`.

たプログラミング言語だ．それは Lisp プログラムはリストとして表現できるが，Lisp にはそのように表現されたプログラムに対する組み込みパーサ（`read`）とコンパイラ（`compile`）があるからだ．大抵は `compile` を呼ぶ必要もない．埋め込み言語は，変換を行うコードをコンパイルすることで自動的にコンパイルさせられる（pnk ページ）．

埋め込み言語とは Lisp の上に書かれたというよりは混じり合って書かれたものである．そのため，文法は Lisp とそのプログラミング言語に特有の構造との混合になる．埋め込み言語を実装する素朴な方法は，Lisp でそのインタプリタを書くことだ．もし可能ならばもっと良いアプローチがある．変換によってそのプログラミング言語を実装することだ．すなわち，評価させたいときには Lisp インタプリタが動作するような Lisp コードに個々の式を変換するのだ．そこにマクロの活躍の場がある．マクロの仕事は正にある種の式を別の式に変換することだから，埋め込み言語を書く際にはマクロは自然な選択だ．

一般的には，埋め込み言語はなるべく変換によって実装できた方がよい．第 1 に，労力が少なくて済む．例えば新しいプログラミング言語に算術演算機能があれば，数値を表現し，操作することの面倒に一切関わらずに済む．Lisp の算術能力が目的に取って十分なものなら，新しい算術式を等価な Lisp の算術式に変換し，残りは Lisp に任せるだけで済む．

普通，変換を使うと，埋め込み言語を速くすることにもなる．インタプリタは本質的に速度に関して不利である．例えばあるコードがループ内にあるとしよう．コンパイルされたコードでは 1 回で済む動作を，インタプリタでは繰り返しの度にしなれけばならないことがしばしばある．そのため自前のインタプリタによる埋め込み言語は，そのインタプリタそのものがコンパイルされたときでも遅い．しかし埋め込み言語の式が Lisp の式に変換されていれば，そのコードは Lisp コンパイラでコンパイルできる．そのように実装された埋め込み言語では，実行時のインタプリテーションのオーヴァヘッドに一切悩まされずに済む．自分のプログラミング言語のための本当のコンパイラを書くのでなければ，マクロによって一番の性能が得られるだろう．実際，埋め込み言語を変換するマクロは，埋め込み言語のコンパイラと見なせる——その動作のほとんどを既存の Lisp コンパイラに依存しているだけのことだ．

第 19-25 章は全てその話題にあてられているので，ここでは埋め込み言語の例について考えることはしない．第 19 章では特に埋め込み言語のインタプリティングと変換の違いを扱い，その 2 通りで同じプログラミング言語を実装して見せる．

Common Lisp についてのある本では，マクロの適用領域が限られていると警告している．その根拠として，CLtL1 で定義されているオペレータのうち，マクロは 10% にも満たないことを引用している．これは家が煉瓦で作られているから，家具も煉瓦で作ろうと言うようなものだ．Common Lisp プログラム内のマクロの比率は，その用途に完全に依存する．マクロを一切使わないプログラムもあれば，全てマクロから成るプログラムもある．

8　変数捕捉

マクロの弱点は，変数捕捉と呼ばれる問題だ．変数捕捉はマクロ展開が名前の衝突を起こしたときに生じる．つまりあるシンボルが別のコンテキストの変数を参照してしまったときだ．うっかりして起きた変数捕捉が，ものすごく些細なバグにつながることもあり得る．この章では，どのようにそれを予測し，避けるかを扱う．しかしながら意図的な変数捕捉は便利なプログラミング技法であり，第 14 章はそれを利用したマクロがほとんどだ．

8.1　マクロ引数の捕捉

意図しない変数捕捉を弱点に持つマクロとは，バグのあるマクロと言ってよい．そのようなマクロを書くのを防ぐには，変数捕捉がいつ起きるのかを正確に知らなければならない．実際の変数捕捉の元を辿ると，以下の 2 通りの状況に行き着く．マクロ引数の捕捉とフリーシンボルの捕捉だ．マクロ引数の捕捉とは，マクロ呼び出しの引数として渡されたシンボルが，不注意によってマクロ展開そのものによって生成された変数を参照してしまうことである．マクロ `for` の，以下の定義を考えてみよう．これは Pascal の `for` 構文のように，本体となる式の実行を繰り返すものだ．

```
(defmacro for ((var start stop) &body body)          ; 誤り
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

このマクロは初見では正しいように見える．それどころかちゃんと動作までするようだ．

```
> (for (x 1 5)
       (princ x))
12345
NIL
```

実際，誤りは微妙なもので，上の形のマクロを数百回使っても全て完璧に動作するかも知れない．しかし，下のように呼び出さなければ，の話だ．

```
(for (limit 1 5)
  (princ limit))
```

この式も上のものと同じ効果を持つと思える．しかしこれは何も表示しない．エラーを起こすのだ．理由を知るために，その展開形を見てみよう．

```
(do ((limit 1 (1+ limit))
         (limit 5))
        ((> limit limit))
  (princ limit))
```

これで何が誤りを引き起こしたのかは明らかだ．マクロの展開形についてローカルなシンボルと，マクロに引数として渡されたシンボルとの間に，名前の衝突があったのだ．展開形が `limit` を捕捉していた．結局それが同じ `do` の中で 2 回現れることになり，エラーになったのだ．

変数捕捉によるエラーは稀だが，頻度が少ない分，質の悪いものだ．先程の変数捕捉は比較的大人しい——少なくとも，ここではエラーが起きた．しかし大抵は，変数捕捉を起こすマクロは，何かがおかしいという兆候を何も示さずに誤った結果をもたらす．下の場合では，

```
> (let ((limit 5))
    (for (i 1 10)
         (when (> i limit)
           (princ i))))
NIL
```

結果のコードはエラーを出さず，しかし何の動作もしない．

8.2　フリーシンボルの捕捉

起こる頻度は少ないが，不注意からマクロ定義そのものが，マクロが展開された環境内の束縛を参照することがある．あるプログラムで，問題が起きたときユーザに警告を発する代わりに，後ほど検討するために警告をリストに蓄え

たいとしよう．ある人はマクロ gripe を書いた．これは警告メッセージを取り，それをグローバルなリスト w に付け加えるものだ．

```
(defvar w nil)

(defmacro gripe (warning)                          ; 誤り
  `(progn (setq w (nconc w (list ,warning)))
          nil))
```

別のある人は関数 sample-ratio を書こうと思った．それは 2 つのリストの長さの比を返すものだ．また，どちらかのリストが 1 個以下の要素しか持たなければ，代わりに nil を返し，同時に「統計的に意味の見出せない状況だ」と警告を発する．(実際の警告はもっと丁寧なのだろうが，警告内容はこの例にとってはどうでもよい．)

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
      (gripe "sample < 2")
      (/ vn wn))))
```

sample-ratio が w = (b) という状態で呼ばれたら，「引数の一つが要素を 1 個しか持っていない，統計的に無意味だ」と警告を発しようとするだろう．しかし gripe の呼び出しが展開されると，sample-ratio が下のように定義されたかのような結果になる．

```
(defun sample-ratio (v w)
  (let ((vn (length v)) (wn (length w)))
    (if (or (< vn 2) (< wn 2))
      (progn (setq w (nconc w (list "sample < 2")))
             nil)
      (/ vn wn))))
```

ここでの問題は，gripe の使われたコンテキストでは w にはそれ自身のローカルな束縛があるということだ．警告はグローバルな警告リストに保存されずに，sample-ratio の仮引数の 1 つの末尾に nconc されてしまう．警告が失われるだけではない．リスト (b) は恐らくプログラムのどこかでデータとして使われるのだろうが，後ろに余計な文字列がくっついてしまう．

```
> (let ((lst '(b)))
    (sample-ratio nil lst)
    lst)
(B "sample < 2")
> w
NIL
```

8.3 捕捉はいつ起きるのか

マクロ定義を見て次の 2 種類の捕捉から起き得る問題を予測できるようになりたいと，多くのマクロ作者が願い続けてきた．捕捉は微妙な問題で，捕捉され得るシンボルがどのようにプログラムに障害をばら撒くかを全て予測できるようになるには，幾らかの経験が要る．幸運なことに，捕捉がプログラムにどのような悪影響を及ぼしているのかを考えなくとも，マクロ定義内の捕捉の起き得るシンボルを判別し，取り除くことができる．この節では，捕捉可能なシンボルを判別するための直截的な方法を示す．この章の残りの節では，そのようなシンボルを取り除く方法を説明する．

捕捉可能な変数を定義する規則は幾つかの耳慣れない概念に依存しており，それらを先に定義しておかなければならない．

> フリー：シンボル s が式の中でフリーなままで現れるというのは，その式の中で変数として使われているが，式の中でその変数が束縛されていないときだ．

以下の式の中では，w，x や z はどれも式 list 内でフリーに現れており，束縛を作っていない．

```
(let ((x y) (z 10))
  (list w x z))
```

しかし外側の式 let は x と z に束縛を与えているので，let の中全体では，y と w のみがフリーなまま現れている．下の式の中では，

```
(let ((x x))
  x)
```

2 個目に現れる x はフリーである——x に対して作られた新しい束縛のスコープ内にはない.

> 骨格：マクロ展開の骨格は，展開形そのものからマクロ呼び出しの引数の一部だったものを全て取り除いたものだ.

以下のように foo が定義されていて,

```
(defmacro foo (x y)
  `(/ (+ ,x 1) ,y))
```

下のように呼ばれていたら,

```
(foo (- 5 2) 6)
```

これは下のような展開形を与える.

```
(/ (+ (- 5 2) 1) 6)
```

この展開形の骨格は，上の式の仮引数 x や y が入っていた所に穴を開けたものだ.

```
(/ (+                  1) )
```

これら 2 つの概念が定義されていれば，捕捉され得るシンボルを判別するための便利な規則を述べることができる.

> 捕捉可能：何らかのマクロの展開形内でシンボルが捕捉可能だというのは，以下の条件のどちらかが満たされるときである．(1) マクロ展開の骨格内でフリーなまま現れているか，(2) マクロに渡された引数が束縛または評価される骨格の一部に束縛されている.

幾つかの例からこの規則の表すものを見てみよう．下のような一番単純な場合では,

```
(defmacro cap1 ()
  '(+ x 1))
```

x は骨格内でフリーなまま現れるだろうから，捕捉可能だ．これが gripe 内のバグを引き起こしたものだ．下のマクロでは,

```
(defmacro cap2 (var)
  `(let ((x ...)
             (,var ...))
         ...))
```

x は捕捉可能だ．それは，x が束縛されているのと同じ式にマクロ呼び出しの引数も束縛されるからだ．(それが for の失敗を引き起こした．) 同様に以下の 2 つのマクロでは,

```
(defmacro cap3 (var)
  `(let ((x ...))
        (let ((,var ...))
           ...)))

(defmacro cap4 (var)
  `(let ((,var ...))
        (let ((x ...))
           ...)))
```

どちらでも x は捕捉可能だ．しかし例えば下の場合のように，x の束縛と引数として渡された変数とがともに可視であるようなコンテキストがなければ,

```
(defmacro safe1 (var)
  `(progn (let ((x 1))
            (print x))
          (let ((,var 1))
            (print ,var))))
```

x は捕捉可能にはならない．骨格に束縛されている変数が全て危険な訳ではない．しかしマクロ呼び出しの引数が骨格によって作られた束縛の中で評価されたら,

```
(defmacro cap5 (&body body)
  `(let ((x ...))
        ,@body))
```

そのように束縛された変数には捕捉の危険がある．cap5 の中では，x は捕捉可能だ．しかし次の場合では，

```
(defmacro safe2 (expr)
  `(let ((x ,expr))
        (cons x 1)))
```

x は捕捉可能ではない．expr に渡された引数が評価される時点では，x の新たな束縛は可視でないからだ．心配する必要があるのは骨格内の変数の束縛だけだということも気を付けて欲しい．下のマクロでは，

```
(defmacro safe3 (var &body body)
  `(let ((,var ...))
        ,@body))
```

どのシンボルにも不注意による捕捉の危険はない（第 1 引数は束縛されるものと分かっているとして）．

捕捉可能なシンボルを同定する規則を新しく手にした所で，for の元の定義を見てみよう．

```
(defmacro for ((var start stop) &body body)          ; 誤り
  `(do ((,var ,start (1+ ,var))
            (limit ,stop))
           ((> ,var limit))
         ,@body))
```

この for の定義は，2 つの点で捕捉を生み出し得ることが分かる．まず元の例のように limit が for の第 1 引数として渡され得る点だ．

```
(for (limit 1 5)
  (princ limit))
```

しかし limit がループ本体内に現れたときも同じ位危険だ．

```
(let ((limit 0))
  (for (x 1 10)
       (incf limit x))
  limit)
```

for をこのように使う人は自分の limit の束縛がループ内で 1 ずつ増やされ，式全体も 55 を返すものと思っているだろう．しかし実際には，展開形の骨格に生成された limit の束縛だけが 1 ずつ増やされる．

```
(do ((x 1 (1+ x))
     (limit 10))
  ((> x limit))
  (incf limit x))
```

そしてそれが反復を制御する役目を持っているので，反復は終了さえしない．

この章で提示された規則は，あくまでも目安として意図されているという留保の元で使って欲しい．それらは形式に則って述べられてさえいないし，ましてや形式上正しくもない．捕捉が問題になるのは元々の意図によることなので，定義は曖昧だ．例えば下のような式では，

```
(let ((x 1)) (list x))
```

(list x) が評価されたときに x が新しい変数を参照することは，エラーとは見なさない．let にはそのような動作が期待されている．変数捕捉を判別する規則もやはり厳密ではない．上の 3 個のテストを満たしながら，なお意図しない変数捕捉を引き起こし得るようなマクロが書ける．例えば下のようなマクロだ．

```
(defmacro pathological (&body body)                ; 誤り
  (let* ((syms (remove-if (complement #'symbolp)
                          (flatten body)))
         (var (nth (random (length syms))
                   syms)))
    `(let ((,var 99))
       ,@body)))
```

このマクロが呼ばれると，本体内の式は progn の中にあるかのように評価される——しかし本体内のランダムなどれか 1 個の変数が違った値を持つかもしれない．これは明らかに変数捕捉だが，その変数は骨格内にはないので，上の規則を満たしている．しかし実践の場では，上の規則はほぼ必ず機能するだろう．上の例のようなマクロを書きたいときは（あったとしても）めったにない．

捕捉を起こしやすい：

```
(defmacro before (x y seq)
  `(let ((seq ,seq))
     (< (position ,x seq)
       (position ,y seq))))
```

正しいヴァージョン：

```
(defmacro before (x y seq)
   `(let ((xval ,x) (yval ,y) (seq ,seq))
      (< (position xval seq)
        (position yval seq))))
```

図 40　let による変数捕捉の回避方法.

8.4　適切な名前によって捕捉を避ける

第 1, 2 節では変数捕捉の実例を 2 種類に分けた：引数の捕捉，これは引数内で使われたシンボルがマクロの骨格に生成された束縛に捕まるものだ．そしてフリー変数の捕捉，これはマクロの展開形内のフリーなシンボルが，マクロが展開された場所で効力を持つ束縛に捕捉されるものだ．普通，後者は単にグローバル変数に区別の付くような名前を与えることで解決される．Common Lisp のグローバル変数には，先頭と末尾にアスタリスクが付く名前を付けるのが伝統だ．例えばカレント・パッケージを定義する変数は*package*と名付けられる．(このような名前は，普通の変数でないことを強調するために「スター・パッケージ・スター」と発音されることがある．)

だからただの w ではなく，*warnings*等の名前の変数に警告を蓄えるようにすることは，完全に gripe の作者の責任だった．sample-ratio の作者が*warnings*を仮引数の名前に使っていたら，現れるバグというバグはみな自分の責任だろう．しかし仮引数を w という名前にしても安全だろうと考えた点は責められることではない.

8.5　優先評価によって捕捉を避ける

危険のある引数をマクロ展開によって作られる束縛よりも外で評価することだけで，引数の捕捉が回避できることがある．一番単純なものはマクロを式 let から始めることで回避できる．第 40図にはマクロ before の 2 通りの定義を示したが，これは 2 個のオブジェクトと 1 個のシーケンスを引数に取り，そのシーケンス内で 1 個目のオブジェクトが 2 個目より前に現れるときに真を返すものだ[*25].

1 個目の定義は不適切だ．最初の let は seq に渡された式が確かに 1 回だけ評価されるようにしているが、これでは次のような問題を防ぐのに不十分だ。

```
> (before (progn (setq seq '(b a)) 'a)
          'b
          '(a b))
NIL
```

これは結局「(a b) の中で a は b より前にあるだろうか？」と尋ねているわけだ．適切に作られた before ならば真を返すところだ。マクロ展開を見ると実際に何が起きるかが分かる．<の第 1 引数の評価が第 2 引数内でも使われるリストを並べ替えてしまっている.

```
(let ((seq '(a b)))
  (< (position (progn (setq seq '(b a)) 'a)
               seq)
     (position 'b seq)))
```

この問題を回避するには，引数を全て 1 個の大きな let で最初に評価してしまえば十分だ．そのため第 40図内の 2 番目の定義には捕捉の危険はない.

残念なことに，let を使う技が通用する状況は多くない.

[*25] このマクロは例として使われているに過ぎない．本当はマクロとして実装すべきでもないし，こんな非効率的なアルゴリズムを使うべきでもない．適切な定義は fang ページを参照.

捕捉を起こしやすい：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
     ,@body))
```

正しいヴァージョン：

```
(defmacro for ((var start stop) &body body)
  `(do ((b #'(lambda (,var) ,@body))
        (count ,start (1+ count))
        (limit ,stop))
       ((> count limit))
     (funcall b count)))
```

図 41　クロージャによる変数捕捉の回避方法.

1. 捕捉の危険のある引数はきっかり 1 回だけ評価されるべきで,
2. マクロ骨格に生成された束縛のスコープ内ではどの引数も評価の必要はない,

というマクロでしか使えない．この制限は多数のマクロを切り捨ててしまう．前に提案されたマクロ for はどちらの条件も満たさない．しかしこの手法の変形を使うことで，for のようなマクロから捕捉の危険を取り除くことができる．ローカルに生成されたどの束縛よりも外側の λ 式の中に実行本体となる式を包み込んでしまうのだ.

反復用マクロを含む幾つかのマクロは，マクロ呼出しに現れる式が新しく生成された束縛の中で評価されるような式を生成する．例えば for の定義内では，繰り返しの本体はマクロの作った do の中で評価されなければならない．そのため，繰り返し本体内に出て来る変数は do に生成された束縛による捕捉の危険がある.

繰り返し本体内の変数をそのような捕捉から保護するためには，本体をクロージャで包み，さらに繰り返しを行う際に，式そのものを挿入せずにクロージャを funcall で呼べばよい.

第 41図にはこの技法を用いたヴァージョンの for を示した．クロージャは for の展開形の中で最初に作られるので，繰り返し本体内に現れるフリーシンボルは全てマクロを呼んでいる環境内の変数を参照する．これなら do はクロージャの仮引数を通じてその本体と関わり合うことになる．全てのクロージャは現在が繰り返しの何回目かを do から伝えられる必要があるので，ただ 1 個の仮引数（マクロ呼び出しでインデックスに指定したシンボル）を持つ.

式を λ 式で包む技法は普遍的な対処法ではない．これはコード本体の保護には使えるが，例えば（最初の不適切な for のように）同じ変数が同じ let や do に 2 回束縛される危険があるような場合には，クロージャは何の役にも立たない．幸運なことにこの場合は，本体をクロージャ内に包むように for を書き直したことで，do が引数 var のために束縛を生成する必要がなくなっている．元の for の引数 var はクロージャの仮引数になっており，do 内では実際のシンボル count で置き換えてもよい．よって第 9.3 節のテストでも分かるように，for の新しい定義は変数捕捉の危険を完全に克服している.

クロージャを使うことの短所は，やや非効率的になるかも知れないことだ．関数呼び出しを 1 回余計に増やしたことになったかも知れない．事に依るともっと不都合なことに，コンパイラがクロージャにダイナミックエクステントを与えていなければ，そのためのスペースは実行時にヒープ領域に割り当てる必要が出るかもしれない.

8.6　Gensym によって捕捉を避ける

マクロの変数捕捉を避けるには，確実な方法が 1 つある．捕捉され得るシンボルを gensym で置き換えてしまうのだ．for の元のヴァージョンでは，問題は 2 個のシンボルが不注意から同じ名前を持ってしまったときに起きた．マクロ骨格が呼出側コードでも使われている名前を含むという可能性を回避したいなら，マクロ定義内では変な名前のシンボルだけを使うことで対処が望めるかも知れない：

```
(defmacro for ((var start stop) &body body)              ; 誤り
  `(do ((,var ,start (1+ ,var))
```

捕捉を起こしやすい：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var))
        (limit ,stop))
       ((> ,var limit))
       ,@body))
```

正しいヴァージョン：

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
         ,@body)))
```

図 42　Gensym による変数捕捉の回避方法.

```
    (xsf2jsh ,stop))
  ((> ,var xsf2jsh))
  ,@body))
```

しかしこれは解決策とはとても言えない．これはバグを取り除いたのではなく，表面化しにくいようにしただけだ．それも大して表面化しにくいわけではない——同じマクロを入れ子にして使ったときに起きる衝突がやはり想定できる．

シンボルが一意的であることを保証する方法が必要だ．Common Lisp の関数 gensym は，まさにこのために存在する．この関数は gensym と呼ばれるシンボルを返すが，これはコードに打ち込まれたりプログラムに生成されたどのシンボルとも eq ではないことが保証されている．

Lisp システムはどうやってそれを保証するのだろうか？ Common Lisp の全てのパッケージは，その中で認識されている全てのシンボルのリストを保持している．(パッケージへの導入に付いては，chuan ページを参照．) そのリストに載っているシンボルはパッケージにインターンされていると言われる．gensym を呼び出す度に，一意でインターンされていないシンボルが返される．そして read に読み取られるシンボルは全てインターンされるので，gensym と等しいものを打ち込むことはできない．そのため，次のように始まる式は，

```
(eq (gensym) ...
```

何を続けても真を返すようにすることはできない．

gensym にシンボルを生成させることは，奇妙な名前のシンボルを選ぶ手法を一歩進めたようなものだ——gensym は，電話帳を探しても載っていないような名前のシンボルを返す．Lisp が gensym を表示しなければならないときは，次のようにする．

```
> (gensym)
#:G47
```

表示されるものは Lisp にとって「名無の権兵衛」程のものに過ぎない．これは任意の名前で，名前が意味を持つことがないように作られたものだ．そしてこの表示について余計な想像を一切引き起こさないように，gensyms はシャープ・コロンに続いて表示される．これは特殊なリードマクロで，表示された gensym を再び読み込もうとしたときエラーを起こすためだけに存在している．

CLtL2 に従う Common Lisp では，gensym の印字表現に現れる数は*gensym-counter*から来ている．これは常に整数に束縛されているグローバル変数だ．このカウンタを手動で設定することで 2 個の gensym を同じように表示させることができる．

```
> (setq x (gensym))
#:G48
> (setq *gensym-counter* 48 y (gensym))
#:G48
>(eqxy)
NIL
```

しかしこれらは同一ではない．

第 42図には，gensym を使った for の正しい定義を載せた．マクロに渡された式の中のシンボルと衝突を起こしていた limit はもうない．それはマクロ展開の時点で生成されたシンボルに置き換わってしまう．マクロが展開される度，limit の場所には展開時に生成された一意な名前のシンボルが代わりに置かれる．

for の正しい定義は一発で書き上げるには複雑過ぎる．完成品のコードは，完成した数学定理のように，しばしば多くの試行錯誤を覆い隠している．だからあるマクロを何段階にも分けて書かなくてはならなくても心配しないことだ．for のようなマクロを書き始めるには，最初のヴァージョンは変数捕捉について考えずに書き，そうしたら前に戻って変数捕捉に関わるシンボルを gensym で置き換えるのがいいかも知れない．

8.7 パッケージによって捕捉を避ける

ある程度までは，マクロを独自のパッケージに入れることで変数捕捉を避けることができる．パッケージ macros を作ってその中で for を定義すれば，最初に挙げた定義

```
(defmacro for ((var start stop) &body body)
     `(do ((,var ,start (1+ ,var))
             (limit ,stop))
            ((> ,var limit))
         ,@body))
```

を使っても他のパッケージからは安全に呼び出せる．for を別のパッケージ（例えば mycode）から呼ぶと，第 1 引数に limit を使っても，それは mycode::limit になる——これはマクロ骨格に現れる macros::limit とは別物だ．

しかしパッケージは変数捕捉の問題の一般的な解決策にはならない．第 1 には，マクロはあるプログラムに統合された一部分であって，それらを独自のパッケージ内に分けておかなければならないのでは不便だ．第 2 には，パッケージ macros 内の別のコードに対する保護には全くなっていない．

8.8 異なる名前空間での捕捉

これまでの章ではあたかも捕捉が変数のみに悪さをする問題かのように扱ってきた．確かにほとんどの捕捉は変数捕捉だが，Common Lisp の他の名前空間においても同様に問題が起き得る．

関数もローカルな束縛を持つことができ，そして関数の束縛も同様に不用意な捕捉を引き起こす可能性がある．例：

```
> (defun fn (x) (+ x 1))
FN
> (defmacro mac (x) `(fn ,x))
MAC
> (mac 10)
11
> (labels ((fn (y) (- y 1)))
    (mac 10))
9
```

捕捉を見つける規則から予測できるように，mac の骨格内でフリーなまま使われている fn には捕捉の危険がある．fn がローカルに再束縛されると，mac の返す値は普通とは違ったものになる．

これにはどう対処すればよいだろうか？捕捉の危険のあるシンボルが組込みの関数やマクロの名前ならば，何もしないでおくのが理に適っている．CLtL2 によれば (p. 260)，組込みのものの名前がローカル関数またはマクロに束縛されたとき，「結果は定義されない」．だから書いたマクロの動作の問題ではない．組込み関数を再束縛したら，自分のマクロだけでなく様々な問題に悩まされることになるだろう．

そうでないとき，変数名を保護するのと同じ方法で関数名をマクロ引数の捕捉から保護できる．マクロ骨格によってローカルに定義されたどのような関数にも，gensym を関数名に使うことだ．この場合，フリーなシンボルの捕捉を避けることは少し難しくなる．変数をフリーシンボル捕捉から保護するには，はっきり区別できるグローバル変数用の名前を付ければよかった：例えば w でなく*warnings*を使えばよい．しかしこの解決策は関数には有効ではない．グローバル関数を区別するための命名法には慣習が無いからだ——何しろ大部分の関数がグローバルなのだから．マクロが，必要な関数がローカルに再定義されているかもしれない環境内で呼ばれることが心配なら，最良の解決策は，おそらくコードを独立したパッケージに入れることだ．

ブロック名，すなわち go や throw で使われるタグからもやはり捕捉が起こり得る．マクロにそのような名前のシンボルが必要なら，our-do の定義と同様に gensym を使うべきだ．

また do 等のオペレータは暗黙のうちに nil という名のブロックに囲まれることも忘れてはいけない．そのため do の中の return や return-from nil は，do の中の式ではなく do そのものから制御を戻すことになる．

```
> (block nil
         (list 'a
               (do ((x 1 (1+ x)))
                 (nil)
                 (if (> x 5)
                   (return-from nil x)
                   (princ x)))))
12345
(A 6)
```

do が nil という名前のブロックを作っていなかったら，この例は (A 6) でなくただの 6 を返していたはずだ．

do の暗黙のブロックは問題とは言えない．do はそのように振る舞うことになっているからだ．しかし展開形が do を含むようなマクロを書いたときは，それらがブロック名 nil を捕捉してしまうことは覚えておくべきだ．for のようなマクロでは，return や return-from nil は，外側のブロックではなく for から制御を戻すことになる．

8.9 変数捕捉にこだわる理由

これまで示した例には極めて病的と言っていいものもあった．それらを見て，こう言いたくなった人もいるかもしれない．「変数捕捉なんて滅多に起きるもんじゃない，なんで気に病まなきゃいけないんだ？」この問には 2 通りの答え方がある．1 つ目は別の質問を返すことだ：バグのないプログラムが書けるのに，どうして少々のバグを持ったプログラムを書くのですか？

長い答え方は，実際の応用では書いたコードの用途について予測を立てるのは危険だと指摘することだ．全ての Lisp プログラムは，現在「オープン・アーキテクチャ」と呼ばれる構造を持つ．他人の使うコードを書いているのなら，その人達は予測もつかない方法でそれを使うかも知れない．また，心配しなければならないのは他人だけではない．プログラムもプログラムを書くのだ．

```
(before (progn (setq seq '(b a)) 'a)
        'b
        '(a b))
```

このようなコードを書く人間はいないかもしれないが，プログラムに生成されたコードはしばしばこのような形をしている．個々のマクロが単純で理に適った外見の展開形を生成するときでさえ，マクロ呼び出しにマクロ呼び出しを渡したりすれば，途端に展開形はいかにも人が書いたものには見えない大きなプログラムになり得る．そのような場合は，マクロが誤って展開されてしまう状況に対しては，それがどんなにありそうもないものでも，防御策を取っておいて損はない．

最後に言いたいのは，捕捉を避けるのはそれ程難しくはないということだ．少し経てば習慣になってしまう．Common Lisp の古典的な defmacro は料理人の包丁のようなものだ．エレガントな道具で危険にも思えるが，達人は確信を持って使っている．

9 マクロのその他の落し穴

マクロを書くときには注意が余分に必要だ．関数は自分のレキシカルな領域に孤立しているが，マクロは，呼出側のコード内に展開されるので，注意深く書かないと不愉快な驚きをもたらすことがある．第 9 章で扱った変数捕捉はその最たるものだ．この章ではマクロを定義する際に避けるべき問題をさらに議論する．

9.1 評価の回数

前の章では for の不適切な定義例をいくつか示した．第 43図にはさらに 2 例を，比較用の適切な定義と共に示した．

捕捉を起こすことはないが，2 番目にはバグがある．これの展開形は stop として渡された式を反復の度に評価して

適切なヴァージョン：

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
       ,@body)))
```

複数回の評価を起こし得る：

```
(defmacro for ((var start stop) &body body)
  `(do ((,var ,start (1+ ,var)))
       ((> ,var ,stop))
     ,@body))
```

評価の順番が間違っている：

```
(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,gstop ,stop)
          (,var ,start (1+ ,var)))
         ((> ,var ,gstop))
       ,@body)))
```

図 43　引数評価の制御

しまう．一番運のよかった場合でも，この種のマクロは非効率で，1 回行えばよいことを繰り返している．stop に副作用があれば，このマクロが実際に間違った結果をもたらすかも知れない．例えば次のループはいつまでも終了しない．ゴールが反復の度に遠のいてゆくからだ．

```
> (let ((x 2))
    (for (i 1 (incf x))
       (princ i)))
12345678910111213...
```

for のようなマクロを書くときには，マクロの引数は式であって値ではないことを忘れてはいけない．それらが展開形内に現れる場所によっては複数回評価されることもある．上の場合，解決策は式 stop が返した値に変数を束縛し，繰り返しの間にはその変数を参照することだ．

マクロは，明確に反復を意図していなければ，式がきっかりマクロ呼び出しに出て来る回数だけ評価されるようにしておくべきだ．この規則の当てはまらない明らかな例がある：Common Lisp の or は，全ての引数を必ず評価するようになったらずっと不便になってしまうだろう（それでは Pascal の or 構文だ）．しかしこのような場合では何回評価が行われるかは分かっている．上の 2 番目の for ではそうではない．式 stop が複数回評価されると想定するような理由はないし，実際そうであるべき理由もない．2 番目の for のようにして書かれたマクロは過誤の産物であることが一番多い．

意図せぬ複数回の評価は setf の上に構築されたマクロにとっては特に難しい問題だ．Common Lisp はそのようなマクロを書き易くするための幾つかのユーティリティを提供している．問題と解決策は第 12 章で扱う．

9.2　評価の順番

式が評価される順番は式が評価される回数程重要ではないが，時々問題になる．Common Lisp の関数呼び出しでは，引数は左から右の順で評価される．

```
> (setq x 10)
10
> (+ (setq x 3) x)
6
```

マクロについてもそうするのがよい習慣だ．マクロでは常に確かに式がマクロ呼び出し内と同じ順で評価されるようにするべきだ．

第 43図では，3 番目の for にも微妙なバグがある．仮引数 stop は start より前に評価される．マクロ呼び出しでは逆の順番になっているというのに．

```
> (let ((x 1))
    (for (i x (setq x 13))
         (princ i)))
13
NIL
```

このマクロは，時間が逆戻りしたかのような disconcerting 印象を与える．字面の上では式 start が最初に出て来るにも関わらず，式 stop の評価が式 start の返す値に影響しているのだ．

適切に定義された for では，引数は確かに出て来る順で評価されるようになっている．

```
> (let ((x 1))
    (for (i x (setq x 13))
         (princ i)))
12345678910111213
NIL
```

これなら x を式 stop 内で設定しても，前の引数が返す値には何の影響もない．

上に示した例はこのために作ったものだが，この種の問題が顕在化する状況は存在し，そしてこのようなバグは発見が極めて難しい．マクロのある引数の評価が別の引数が返す値に影響するようなコードを書く人はまずいないだろう．しかし人は意図的には決してやらないことでも誤ってやってしまうことがあるものだ．ユーティリティは，意図された通りに使われれば正しく機能するだけでなく，バグを隠蔽してもいけない．上の例のようなコードを書く人がいたら，それはきっと間違って書いたのだろう．しかし適切に定義された for は容易に誤りに気付かせてくれる．

9.3 関数によらないマクロ展開

Lisp は，マクロ展開を生成するコードは第 3 章で論じた意味で純粋に関数的であるものと予期している．展開を行うコードは引数として渡された式にのみ依存すべきで，値を返す他には周囲の世界に影響しようとすべきではない．

CLtL2 にもあるように (p. 685)，コンパイル済みコード内でのマクロ呼び出しは実行時に再展開されることはないと思っておくのが安全だ．そうでなければ，Common Lisp はいつの時点で，またはどれほどの頻度でマクロ呼び出しが展開されるのか保証できなくなってしまう．マクロの展開結果がそのどちらかによって変わることがあれば，それは誤りと見なされる．例えば，あるマクロが使われた回数を数えたいとしよう．単にソースに検索をかければよい訳ではない．そのマクロがプログラムの生成したコード内で呼ばれているかも知れないからだ．だから次のようなマクロを定義したくなる．

```
(defmacro nil! (x)                    ; 誤り
  (incf *nil!s*)
  `(setf ,x nil))
```

この定義ではグローバル変数*nil!s*が nil!が展開される度に 1 だけ増加する．しかしこの変数の値から nil!の呼ばれた回数が分かると思ったら間違いだ．あるマクロ呼び出しは複数回展開され得る（実際しばしばそうだ）．例えばソースコードの変換を行うプリプロセッサは，式内のマクロ呼び出しを，それを変換すべきかどうか判断する前に展開しなければならないかもしれない．

一般則としては，展開を行うコードは引数以外の何にも依存すべきでない．だから例えば展開形を文字列から作り出すようなどのマクロも，展開時に何のパッケージ内にいるかについて勝手に予想しないよう注意すべきだ．次の簡潔だがかなり病的な例は，

```
(defmacro string-call (opstring &rest args)              ; 誤り
     `(,(intern opstring) ,@args))
```

オペレータの印字名をとってそのオペレータへの呼び出しを展開するマクロを定義している．

```
> (defun our+ (x y) (+ x y))
OUR+
> (string-call "OUR+" 2 3)
5
```

intern は文字列を取り，対応するシンボルを返す．しかしオプショナル引数のパッケージを省くと，カレント・パッケージが使われる．よって展開形はそれが生成されるときのパッケージに依存することになる．our+ がそのパッケージで可視でない限り，展開形は未定義関数への呼び出しになる．

Miller と Benson の *Lisp Style and Design* は展開コードの副作用から起きる問題の特別に汚い例に言及している．Common Lisp では CLtL2 にもあるように (p. 78)，&rest 引数に束縛されたリストが新しく生成されたものである保証はない．それはプログラム内のどこかのリストと構造を共有しているかも知れない．その結果，&rest 引数は破壊的に操作してはいけないことになる．同時に他の何に変更を加えているか分からないからだ．

この可能性は関数とマクロ両方に影響する．関数については apply を使うと問題が顕在化する．Common Lisp の規格に沿った処理系の実装では次の問題が起こり得るのだ．引数のリストの末尾に et al を付け加える関数 et-al を定義したいとしよう．

```
(defun et-al (&rest args)
  (nconc args (list 'et 'al)))
```

この関数は普通に呼ぶ限りでは適切に動作するように思える：

```
> (et-al 'smith 'jones)
(SMITH JONES ET AL)
```

しかしそれを apply 経由で呼び出すと，既存のデータ構造が変更される可能性がある：

```
> (setq greats '(leonardo michelangelo))
(LEONARDO MICHELANGELO)
> (apply #'et-al greats)
(LEONARDO MICHELANGELO ET AL)
> greats
(LEONARDO MICHELANGELO ET AL)
```

少なくとも Common Lisp の規格に沿った処理系ではこうなる可能性がある．今のところ実際にはそのような処理系はないようだが．

マクロでは危険度が大きい．&rest 引数に変更を加えるマクロはそれによってマクロ呼び出しに変更を加える可能性がある．言い替えれば，うっかり自己修正的なプログラムができてしまいかねないということだ．現実的な危険性もある——これは既存の処理系で起こったことなのだ．何かを&rest 引数に nconc するマクロを書き[*26]，

```
(defmacro echo (&rest args)
  `',(nconc args (list 'amen)))
```

次にそれを呼び出す関数を書く：

```
(defun foo () (echo x))
```

広く使われている Common Lisp 処理系では，次のような結果になる．

```
> (foo)
(X AMEN AMEN)
> (foo)
(X AMEN AMEN AMEN)
```

foo は誤った結果を返すだけでなく，実行する度に結果は異なる．マクロが展開される度に foo の定義が変更されているからだ．

あるマクロ呼び出しが複数回展開されることについて先に論じた点についても，この例から分かることがある．この特定の処理系では，foo の 1 回目の呼び出しは 2 個の amen から成るリストを返す．何かの理由でこの処理系はマクロ呼び出しを foo の展開時に 1 回展開し，そしてその後呼び出される度にも 1 回ずつ展開したのだ．

echo は次のように定義した方が安全だ．

```
(defmacro echo (&rest args)
  `',(,@args amen))
```

これはコンマ・アットは nconc でなく append と等価だからだ．このマクロを再定義した後は，foo はコンパイルされていなくとも再定義しなければならない．前のヴァージョンの echo がそれを書き換えてしまったからだ．??? ここから??END までの行が挿入か削除されたようです

[*26] `',(foo) は `(quote ,(foo)) と等価な表現だ．

正しく動作するもの：

```
(defun ntha (n lst)
  (if (= n 0)
      (car lst)
      (ntha (- n 1) (cdr lst))))
```

コンパイルできないもの：

```
(defmacro nthb (n lst)
  `(if (= ,n 0)
       (car ,lst)
       (nthb (- ,n 1) (cdr ,lst))))
```

図 44　誤って再帰関数と同じように捉えてしまった例.

マクロでは，同様の危険があるのは&rest 引数だけではない．リストであるマクロの引数はいずれも変更しないでおくべきだ．引数のどれかに変更を加えるマクロとそれを呼ぶ関数を定義すると，

```
(defmacro crazy (expr) (nconc expr (list t)))

(defun foo () (crazy (list)))
```

呼出側関数のソースコードは変更されてしまう可能性がある．ある処理系では 1 回目に呼び出したときにそうなった．

```
> (foo)
(T T)
```

これはコンパイラとインタプリタ両方で起きたことだ.

要するに，引数のリスト構造を破壊的に変更することでコンシングを避けようとしないことだ．出来上がったプログラムは，例え動作したとしても可搬性を持たないだろう．可変個の引数を取る関数内でコンシングを避けたいなら，解決策の一つはマクロを使い，コンシングをコンパイル時にずらす方法だ．マクロをこのように使うことについては，第 13 章を参照すること.

またマクロ展開を行うコードの返した式がクォート付きリストを含むときは，それに破壊的操作を行うことも避けるべきだ．これはマクロの本質的な制限ではなく，第 3.3 節で大まかに触れた原則の一例だ.

9.4　再帰

関数を再帰的に定義するのが自然なときがある．次のような関数は，本質的に再帰的な性質を持っている.

```
(defun our-length (x)
  (if (null x)
    0(1+ (our-length (cdr x)))))
```

上の定義は，等価な反復版よりも（おそらく遅いだろうが）自然に思える.

```
(defun our-length (x)
  (do ((len 0 (1+ len))
       (y x (cdr y)))
    ((null y) len)))
```

再帰的でもなく，相互再帰的な関数の集合の一部でもない関数は，第 7.10 章で述べた簡単な方法によりマクロに変換できる．しかしただ逆クォートとコンマを挿入するだけでは再帰的関数は変換できない．組込み関数 nth を例に取ってみよう．(話を簡単にするため，この nth はエラーチェックをしないものとする．) 第 44図には nth をマクロとして定義しようとして間違えた例を示した．表面的には nthb は ntha と等価に見えるが，nthb を使ったコードはコンパイルに通らない．これはマクロ呼び出しの展開が終了しないためだ.

一般的に，マクロが他のマクロへの参照を含むことは，展開がどこかで終了する限りは問題ない．nthb の問題は，nthb のどの展開形も nthb 自身への参照を含む点だ．関数版の ntha は終了する．それは n の値について再帰的なのだが，この値は再帰の度に減少するからだ．しかしマクロ展開は式にしかアクセスがなく，その値までは分からない．コンパイラが，例えば (nthb x y) を展開しようとすると，最初の展開では

```
(defmacro nthd (n lst)
  `(nth-fn ,n ,lst))

(defun nth-fn (n lst)
  (if (= n 0)
    (car lst)
    (nth-fn (- n 1) (cdr lst))))

(defmacro nthe (n lst)
  `(labels ((nth-fn (n lst)
              (if (= n 0)
                  (car lst)
                  (nth-fn (- n 1) (cdr lst)))))
     (nth-fn ,n ,lst)))
```

図 45　問題の 2 通りの修正方法.

```
(if (= x 0)
  (car y)
  (nthb (- x 1) (cdr y)))
```

が作られ，これは更に次のようになる．

```
(if (= x 0)
  (car y)
  (if (= (- x 1) 0)
    (car (cdr y))
    (nthb (- (- x 1) 1) (cdr (cdr y)))))
```

こうして無限ループに陥ってしまうのだ．マクロが自分自身の呼び出しを展開することに問題はないが，それはいつまでも続かないときの話だ．

nthb のような再帰的マクロの危険な点は，インタプリタでは適切に機能することだ．とうとう動作するようになったプログラムをコンパイルしようとしたとき，それがコンパイルを通りすらしないことになる．それだけでなく，問題が再帰的マクロによることは普通分からない．コンパイラはただ無限ループに陥り，そこで何が悪いのか考える羽目になる．

上の場合，ntha は末尾再帰的だ．末尾再帰的関数は容易に反復形に変換でき，それはマクロのモデルに使える．nthb のようなマクロはこうして書ける．

```
(defmacro nthc (n lst)
  `(do ((n2 ,n (1- n2))
        (lst2 ,lst (cdr lst2)))
     ((= n2 0) (car lst2))))
```

原則的にはマクロで再帰関数を複製することは不可能ではない．しかし複雑な再帰関数の変換は難しかったり，不可能ですらあるときもある．

マクロの用途によっては，代わりにマクロと関数の組合せを使うことで十分なときもある．第 45には再帰的マクロのようなものを作る方法を 2 通り示した．nthd で示される 1 番目の戦略は，単にマクロを再帰関数の呼び出しへ展開させることだ．例えばマクロが引数にクォートを付ける手間を省くためだけに使われているのなら，この手法で十分だろう．

マクロが必要な理由が，その展開形を丸ごと呼出側のレキシカルな環境に挿入したいということなら，nthe に例示した方法の方ががよいだろう．組込みの特殊式 labels（第 2.7 節を参照）がローカルな関数定義を作っている．nthc の展開形はいずれもグローバル関数 nth-fn を呼んでいたが，nthe の展開形はそれぞれの内部に同様な関数を持っている．

再帰関数を直接マクロに変換できなくても，展開形が再帰的に生成されるマクロを書くことはできる．マクロを展開する関数は Lisp の通常の関数であって，もちろん再帰的になり得る．例えば組込みの or を独自に定義するときには，再帰的な展開関数を使うことになるだろう．

```
(defmacro ora (&rest args)
  (or-expand args))

(defun or-expand (args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               ,(or-expand (cdr args)))))))

(defmacro orb (&rest args)
  (if (null args)
      nil
      (let ((sym (gensym)))
        `(let ((,sym ,(car args)))
           (if ,sym
               ,sym
               (orb ,@(cdr args)))))))
```

図 46　再帰的展開関数.

第 46図には，or のための再帰的な展開関数を定義する方法を 2 通り示した．マクロ ora は展開形を作るために再帰関数 or-expand を呼び出す．これは適切に動作する．また等価な orb も同様だ．orb は再帰的だが，マクロの引数そのものについて再帰的であり（これは展開時にアクセスできる），その値について再帰的なのではない（こちらはアクセスできない）．展開形が orb そのものへの参照を含むようにも見えるが，ある 1 段階のマクロ展開で生成された orb の呼び出しは次の段階で let に置き換えられ，展開の最終形では入れ子になった let のスタックしか残らない．(orb x y) は次のコードと等価なコードに展開される．

```
(let ((g2 x))
  (if g2
      g2
      (let ((g3 y))
        (if g3 g3 nil))))
```

実際 ora と orb は等価であり，どちらを使うかは個人的な好みの問題に過ぎない.

10　古典的なマクロ

この章では，一番よく使われる種類のマクロを定義する方法を示す．それらは——かなり重複が生じるが——3 種類に分けられる．1 種類目はコンテキストを作るマクロだ．オペレータが引数を新しいコンテキスト内で評価するものなら，おそらくマクロとして定義されなければならないだろう．始めの 2 節は基本的な 2 種類のコンテキストについて説明し，それぞれに対してマクロを定義する方法を示す.

続く 3 節は条件付き評価と反復評価のためのマクロについて説明する．オペレータが引数を 1 回より少なく，または複数回評価するものなら，やはりマクロとして定義されなければならない．条件付き評価のためのマクロと反復評価のためのマクロとの間には明確な区別はない．この章で示した例の幾つかは両方を兼ねる（束縛と同様に）．最後の節は条件付き評価のためのマクロと反復評価のためのマクロとの間のもう 1 つの類似性について説明する．場合によっては，どちらも関数によって実現できるのだ.

10.1　コンテキストの生成

コンテキストには 2 つの意味がある．1 種類目のコンテキストとはレキシカルな環境だ．特殊式 let は新しいレキシカル環境を作る．let の本体内の式は新しい変数を含んでいるかも知れないような環境内で評価される．x がトップレベル内で a に設定されたとする．しかし次の式は (b) を返す.

```
(defmacro our-let (binds &body body)
  `((lambda ,(mapcar #'(lambda (x)
                         (if (consp x) (car x) x))
                     binds)
      ,@body)
    ,@(mapcar #'(lambda (x)
                  (if (consp x) (cadr x) nil))
              binds)))
```

図 47　let のマクロによる実装.

```
(defmacro when-bind ((var expr) &body body)
  `(let ((,var ,expr))
     (when ,var
       ,@body)))

(defmacro when-bind* (binds &body body)
  (if (null binds)
      `(progn ,@body)
      `(let (,(car binds))
         (if ,(caar binds)
             (when-bind* ,(cdr binds) ,@body)))))

(defmacro with-gensyms (syms &body body)
  `(let ,(mapcar #'(lambda (s)
                     `(,s (gensym)))
                 syms)
     ,@body))
```

図 48　変数を束縛するマクロの例.

```
(let ((x 'b)) (list x))
```

これは値が b である新しい変数 x を含む環境内で list が呼ばれたからだ.

式を実行本体として取るオペレータは普通はマクロとして定義されなければならない．prog1 や progn などの場合を除けば，そのようなオペレータの目的は普通は本体部を何か新しいコンテキスト内で評価させることだろう．コンテキストを生成するコードで本体部の外を覆うためには，コンテキストが新しいレキシカル変数を含まないときであっても，マクロが必要になる.

第 47図には let がどのように lambda 上のマクロとして定義できるかを示した．our-let は関数アプリケーションに展開される――

```
(our-let ((x 1) (y 2))
  (+ x y))
```

これは次のようになる.

```
((lambda (x y) (+ x y)) 1 2)
```

第 48図にはレキシカル環境を作り出す新マクロを 3 個載せた．第 7.5 節では when-bind を引数リストの構造化代入の例として扱ったので，このマクロはすでに ching ページに説明されている．更に一般的な when-bind*は (*symbol expression*) の形のペアから成るリストを取る――let の引数と同じ形式だ．どれかの式が nil を返したら，式 when-bind*全体の値として nil が返される．そうでなければそれぞれのシンボルが let*と同様に束縛された状態で本体部が評価される：

```
> (when-bind* ((x (find-if #'consp '(a (1 2) b)))
                (y (find-if #'oddp x)))
       (+ y 10))
11
```

最後に，マクロ with-gensyms はそれ自身がマクロを書くために使われる．多くのマクロ定義は gensym の生成から始まるが，それは時としてかなりの数に昇る．マクロ with-redraw (p. 115) では 5 個だった.

```
(defmacro condlet (clauses &body body)
  (let ((bodfn (gensym))
        (vars (mapcar #'(lambda (v) (cons v (gensym)))
                      (remove-duplicates
                        (mapcar #'car
                                (mappend #'cdr clauses))))))
    `(labels ((,bodfn ,(mapcar #'car vars)
                      ,@body))
       (cond ,@(mapcar #'(lambda (cl)
                           (condlet-clause vars cl bodfn))
                       clauses)))))

(defun condlet-clause (vars cl bodfn)
  `(,(car cl) (let ,(mapcar #'cdr vars)
                (let ,(condlet-binds vars cl)
                  (,bodfn ,@(mapcar #'cdr vars))))))


(defun condlet-binds (vars cl)
  (mapcar #'(lambda (bindform)
              (if (consp bindform)
                  (cons (cdr (assoc (car bindform) vars))
                        (cdr bindform))))
          (cdr cl)))
```

図 49　cond と let との組み合せ.

```
(defmacro with-redraw ((var objs) &body body)
  (let ((gob (gensym))
        (x0 (gensym)) (y0 (gensym))
        (x1 (gensym)) (y1 (gensym)))
    ...))
```

このようなマクロ定義は，with-gensyms によって簡潔になる．これはリスト内の変数を全て gensym に束縛するものだ．このマクロを使えば，ただ次のように書けばよい.

```
(defmacro with-redraw ((var objs) &body body)
  (with-gensyms (gob x0 y0 x1 y1)
    ...))
```

この新マクロは以降の章を通じて使われる.

幾つかの変数を束縛し，その後何かの条件に基づいて式の組の中の 1 つを評価したいとき，let 内でただ条件判断を使っていた.

```
(let ((sun-place 'park) (rain-place 'library))
  (if (sunny)
      (visit sun-place)
      (visit rain-place)))
```

残念なことに，逆の状況には便利な慣用法が存在しない．つまり実行したいコードは常に同じだが，束縛が何かの条件によって変化するときだ.

第 49図にはそのような状況のためのマクロを示した．名前から分かるように，condlet は cond と let との合の子のような働きをする．これは束縛指定の節と，その次の実行本体部を引数に取る．それぞれの束縛節はテスト式によって守られている．コード本体部はテスト式が真を返した最初の節に指定された束縛の下で評価される．一部の節にのみ含まれる変数は，真となった節が束縛を指定していなければ nil に束縛される.

```
> (condlet (((= 1 2) (x (princ 'a)) (y (princ 'b)))
            ((= 1 1) (y (princ 'c)) (x (princ 'd)))
            (t       (x (princ 'e)) (z (princ 'f))))
           (list x y z))
CD
(D C NIL)
```

condlet の定義は our-let の定義の一般化として捉えられる．後者は本体部を関数に変え，その関数が初期値を決める式の評価結果に適用されるようにする．condlet は labels を使ってローカルな関数を定義しているコードに展開される．その中では初期値を決める式のどの組が評価されて関数に渡されるかを cond 節が決めるようになっている．

展開コードは束縛指定節から変数名を抽出するために mapcan でなく mappend を使っていることに注意しよう．これは mapcan が破壊的であって，第 10.3 章で警告したように，引数のリスト構造に変更を加えることは危険だからだ．

10.2 with-系マクロ

レキシカル環境の他に，もう 1 種のコンテキストがある．そのコンテキストとは，広い意味で世界の状態のことを指す．これはスペシャル変数の値，データ構造体の内容，Lisp 外部の事物の状態を含む．この種のコンテキストを作り出すオペレータも，コード本体がクロージャとしてまとめられていない限り，やはりマクロとして定義されなければならない．

コンテキスト生成マクロの名前はしばしば with-で始まる．この種のマクロの中で一番よく使われるのは多分 with-open-file だろう．その本体は新しく開かれたファイルが指定の変数に束縛された状態で評価される．

```
(with-open-file (s "dump" :direction :output)
  (princ 99 s))
```

この式の評価後にはファイル「dump」は自動的に閉じられ，中身は 2 文字「99」となっているだろう．

このオペレータは s を束縛するので，明らかにマクロとして定義されなければならない．しかし式を新しいコンテキスト内で評価させるようなオペレータは，とにかくマクロとして定義されなければならないのだ．マクロ ignore-errors は CLtL2 で新たに追加されたものだが，これはその引数が progn 内にあるかのように評価されるようにする．いずれかの時点でエラーが発生すると，式 ignore-errors 全体が nil を返す．(これは，例えばユーザの入力したテキストを読み込む際などに便利だ．) ignore-errors は変数を作る訳ではないが，マクロとして定義されなければならない．その引数が新しいコンテキスト内で評価されるからだ．

一般的に言って，コンテキストを生成するマクロはコードのブロックに展開される．本体の前，後，または両方に式が付加されるかもしれない．そのコードが本体の前に来たときは，その目的はシステムを定常状態に保つことかも知れない——つまり何らかの後始末だ．例えば with-open-file は，自分が開いたファイルを閉じなければならない．そのような状況下では，unwind-protect に展開されるコンテキスト生成マクロを作るのが定石だ．

unwind-protect の目的は，プログラムの実行が割り込みされたときであっても，ある式が必ず評価されるようにすることだ．それは 1 個以上の引数を取り，それらが順に評価される．何事もなければ prog1 と同様に第 1 引数の値を返す．それとの違いは，第 1 引数の評価がエラーや throw によって中断されたときでさえも，残りの引数が評価される点だ．

```
> (setq x 'a)
A> (unwind-protect
        (progn (princ "What error?")
               (error "This error."))
        (setq x 'b))
What error?
>>Error: This error.
```

式 unwind-protect 全体としてはエラーを返す．しかしトップレベルに戻った後には，第 2 引数も評価されていたことに気が付く：

```
>x
B
```

これは with-open-file が unwind-protect に展開されたことで，本体の評価中にエラーが起きても普通なら開かれたファイルが閉じられるからだ．

コンテキスト生成マクロは，大抵は特定の用途のために作られる．例として，複数の遠隔データベースを扱うプログラムを書いているとしよう．プログラムはグローバル変数*db*で指定されるデータベース 1 個ずつに命令する．データベースの使用前には，他人が同時に使うことがないようにそれをロックしなければならない．また作業が済んだらロックを解除しなければならない．データベース db のクエリ q の値が欲しいときは，次のようにするだろう．

純粋なマクロ：

```
(defmacro with-db (db &body body)
  (let ((temp (gensym)))
    `(let ((,temp *db*))
       (unwind-protect
         (progn
           (setq *db* ,db)
           (lock *db*)
           ,@body)
         (progn
           (release *db*)
           (setq *db* ,temp))))))
```

マクロと関数との組合せ：

```
(defmacro with-db (db &body body)
  (let ((gbod (gensym)))
    `(let ((,gbod #'(lambda () ,@body)))
       (declare (dynamic-extent ,gbod))
       (with-db-fn *db* ,db ,gbod))))

(defun with-db-fn (old-db new-db body)
  (unwind-protect
    (progn
      (setq *db* new-db)
      (lock *db*)
      (funcall body))
    (progn
      (release *db*)
      (setq *db* old-db))))
```

図 50　典型的な with-系マクロ.

```
(let ((temp *db*))
  (setq *db* db)
  (lock *db*)
  (prog1 (eval-query q)
    (release *db*)
    (setq *db* temp)))
```

マクロを使えば煩わしい作業を全て隠蔽できる．第 50図では，高い抽象化レベルでデータベースを扱えるようにしてくれるマクロを定義した．with-db を使えば，ただこうするだけでよい．

```
(with-db db
  (eval-query q))
```

with-db は単なる prog1 でなく unwind-protect に展開されるので，安全に呼び出せる．

第 50図内の 2 つの with-db の定義は，この種のマクロの 2 通りの書き方の可能性を示している．1 番目は純粋なマクロで，2 番目は関数とマクロとの組み合わせだ．実現したい with-系マクロが複雑になるに連れ，2 番目の手法の方が実用的になる．

CLtL2 準拠の Common Lisp 処理系では，宣言 dynamic-extent によって本体部を括るクロージャが効率的に割り当てられるようにできる（CLtL1 準拠の処理系では無視される）．このクロージャは with-db-fn を呼んでいる間だけ必要なので，その通りのことを宣言すれば，必要なメモリ空間をコンパイラにスタック上に割り当てさせることができる．この空間は後でガーベジ・コレクタに回収されるのではなく，式から出るときに自動的に回収される．

10.3　条件付き評価

マクロ呼び出し内の引数を，ある条件が成立するときだけ評価したいことがある．これは，引数を必ず評価することになっている関数にはできないことだ．if，and や cond のような組込みオペレータは引数の幾つかを保護していて，

```
(defmacro if3 (test t-case nil-case ?-case)
  `(case ,test
     ((nil) ,nil-case)
     (?       ,?-case)
     (t       ,t-case)))

(defmacro nif (expr pos zero neg)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ((plusp ,g) ,pos)
             ((zerop ,g) ,zero)
             (t ,neg)))))
```

図 51　条件付き評価のためのマクロ.

他の引数がある値を返さない限り評価しない．例えば次の式では,

```
(if t 'phew
      (/ x 0))
```

第 3 引数は評価されると「ゼロ除算」エラーを引き起こす．しかし 1，2 番目の引数だけが評価されることになるので, if 全体としては常に安全に phew を返す.

このようなオペレータを新しく書くには，既存のこのようなオペレータに展開されるようなマクロを書けばよい．第51には，if の考え得る多くの変種のうち 2 つを示した．if3 の定義は，三値論理のための条件判断を定義する方法を示している．nil を偽，そのほか全てを真とするのではなく，このマクロは真理の 3 種類の度合を考慮する：真，偽，そして？で表される可能だ．これは 5 歳の子供を描写する際に次のように使えるかも知れない.

```
(while (not sick)
  (if3 (cake-permitted)
       (eat-cake)
       (throw 'tantrum nil)
       (plead-insistently)))
```

この新しい条件判断オペレータは case に展開される．(キー nil はリストに括らなければならない．nil 単独では意味があいまいになるからだ．) 最後の 3 つの引数は，第 1 引数の値に従ってどれか 1 つだけが評価される.

nif という名前は「numeric if（数値的 if)」から来ている．別の実装方法は bnm ページに示した．これは数値の式を第 1 引数に取り，その符号に従って残り 3 つの引数のどれかを評価する.

```
> (mapcar #'(lambda (x)
                  (nif x 'p 'z 'n))
          '(0 1 -1))
(ZPN)
```

第 52図には，条件付き評価の利点を生かしたマクロを更に幾つか示した．マクロ in は集合への所属関係を効率的に調べるものだ．あるオブジェクトが幾つかのオブジェクトの中のどれかと同じものか調べたいとき，その問は選言（訳注：いわゆる or）として表現できる.

```
(let ((x (foo)))
  (or (eql x (bar)) (eql x (baz))))
```

または同じ事を集合への所属関係としても表現できる.

```
(member (foo) (list (bar) (baz)))
```

後者の方が抽象的だが，非効率的だ．member を使った式には 2 つの理由で不必要なコストがかかる．これはオブジェクトの集合を member が調べるためにリストに括らなければならないので，コンシングを起こす．そしてオブジェクトの集合をリストに括るためにはそれらを全て評価しなければならないが，中には全く必要でない値もある．(foo) の値が (bar) の値に等しければ，(baz) を評価する必要はない．概念としてはどれ程優れていても，このように member を使うのはよい方法ではない．同様な抽象化はマクロによればより効率的に実現できる．in は member の抽象性と or の効率性を併せ持ったものだ．in を使った等価な式は member の式と同じ形をしている.

```
(defmacro in (obj &rest choices)
  (let ((insym (gensym)))
    `(let ((,insym ,obj))
       (or ,@(mapcar #'(lambda (c) `(eql ,insym ,c))
                     choices)))))

(defmacro inq (obj &rest args)
  `(in ,obj ,@(mapcar #'(lambda (a)
                          `',a)
                      args)))

(defmacro in-if (fn &rest choices)
  (let ((fnsym (gensym)))
    `(let ((,fnsym ,fn))
       (or ,@(mapcar #'(lambda (c)
                         `(funcall ,fnsym ,c))
                     choices)))))

(defmacro >case (expr &rest clauses)
  (let ((g (gensym)))
    `(let ((,g ,expr))
       (cond ,@(mapcar #'(lambda (cl) (>casex g cl))
                       clauses)))))

(defun >casex (g cl)
  (let ((key (car cl)) (rest (cdr cl)))
    (cond ((consp key) `((in ,g ,@key) ,@rest))
          ((inq key t otherwise) `(t ,@rest))
          (t (error "bad >case clause")))))
```

図 52　条件付き評価のためのマクロ.

`(in (foo) (bar) (baz))`

しかしこれは次のように展開される.

```
(let ((#:g25 (foo)))
  (or (eql #:g25 (bar))
      (eql #:g25 (baz))))
```

よくあることだが，慣用法の明確なものと効率的なものとの間の選択に直面したときは，前者を後者に変換するマクロを書くことでディレンマの角を抜けることができる（訳注：「どちらかの選択」に実は第 3 の道があること).

`inq` は “in queue” と発音するが，これはクォートを使う `in` の変種で，これを使うのは `set` の代わりに `setq` を使うようなものだ．次の式は,

`(inq operator + - *)`

次のように展開される.

`(in operator '+ '- '*)`

`member` のデフォルト動作と同様に，`in` と `inq` は同一性の判断に `eql` を使う．判断に他のオペレータを使いたいなら——1 個の引数を取る関数なら何でもよいが——更に一般的な `in-if` を使うとよい．`in-if` と `some` との関係はちょうど `in` と `member` との関係と同じだ．次の式は,

`(member x (list a b) :test #'equal)`

次のように書き換えられる.

`(in-if #'(lambda (y) (equal x y)) a b)`

同様に次の式は,

`(some #'oddp (list a b))`

次のようになる.

`(in-if #'oddp a b)`

```
(defmacro while (test &body body)
  `(do ()
     ((not ,test))
     ,@body))

(defmacro till (test &body body)
  `(do ()
     (,test)
     ,@body))

(defmacro for ((var start stop) &body body)
  (let ((gstop (gensym)))
    `(do ((,var ,start (1+ ,var))
          (,gstop ,stop))
         ((> ,var ,gstop))
         ,@body)))
```

図 53　単純な反復用マクロ.

cond と in を組み合わせることで，case の便利な変種が定義できる．Common Lisp のマクロ case のキーが定数であることを前提にしている．しかし case の動作が欲しいが，キーを評価して欲しいときもある．>case はそのような場合のために作られた．これは case と似ているが，それぞれの節を保護するキーが比較前に評価されるようになっている．(名前の「 > 」は矢印が評価を表すつもりで付けられた．) >case は in を使っているので，必要以上のキーは評価しない.

キーは Lisp の式であってもよいので，(x y) が関数やマクロの呼び出しなのか 2 つのキーから成るリストなのか判断が付かない．曖昧さをなくすため，(t 以外の）キーは，1 つしかない場合でも必ずリストに括られていなければならない．case の場合，曖昧であるために nil は節の car 部になってはいけなかった．>case では，nil は節の car 部として曖昧ではない．しかしそうすると節の残りが決して評価されないことになる.

明確さのため，それぞれの>case 節の展開形を生成するコードは独立した関数>casex として定義された．>casex 自身も inq を使っていることに注意しよう.

10.4　反復

関数にまつわる問題が，引数が必ず評価されることではなく，引数が 1 回しか評価されないことであるときもある．関数の引数はどれもきっかり 1 回だけ評価されるので，本体となる式を取ってそれを反復評価するオペレータを定義したければ，それをマクロとして定義するしかない.

一番単純な例は，引数を順番に永遠に評価し続けるマクロだろう.

```
(defmacro forever (&body body)
  `(do ()
     (nil)
     ,@body))
```

これはちょうど組込みマクロ loop にキーワードを与えなかったときの動作だ．永遠の繰り返しには大した将来性はないように思える（むしろ将来に渡って続き過ぎる)．しかし block や return-from と組み合わせれば，この種のマクロが，中断は緊急時のみであるようなループを表現するための一番自然な方法になる.

反復のためのマクロの単純な例を幾つか第 53に示した．while はすでに見たものだ (p. vom)．この本体部はテスト式が真を返す間中評価され続ける．till はそれと対をなすもので，テスト式が偽を返す間評価を繰り返す．最後の for もすでに見た (p. qwe) もので，ある範囲の数に対して反復を行う.

これらのマクロを do に展開するようにすることで，その本体部の中で go や return が使えるようになる．do の持つこの性質は block と tagbody から引き継いだものだから，while，till と for は do からやはり引き継ぐことになる．zxc ページで説明したように，do を囲む暗黙のブロックのタグ nil は第 53図のマクロに捕捉される．この点はバ

```
(defmacro do-tuples/o (parms source &body body)
  (if parms
      (let ((src (gensym)))
        `(prog ((,src ,source))
               (mapc #'(lambda ,parms ,@body)
                     ,@(map0-n #'(lambda (n)
                                   `(nthcdr ,n ,src))
                               (1- (length parms)))))))))

(defmacro do-tuples/c (parms source &body body)
  (if parms
      (with-gensyms (src rest bodfn)
                    (let ((len (length parms)))
                      `(let ((,src ,source))
                         (when (nthcdr ,(1- len) ,src)
                           (labels ((,bodfn ,parms ,@body))
                             (do ((,rest ,src (cdr ,rest)))
                               ((not (nthcdr ,(1- len) ,rest))
                                ,@(mapcar #'(lambda (args)
                                              `(,bodfn ,@args))
                                          (dt-args len rest src))
                                nil)
                               (,bodfn ,@(map1-n #'(lambda (n)
                                                     `(nth ,(1- n)
                                                           ,rest))
                                                 len)))))))))

(defun dt-args (len rest src)
  (map0-n #'(lambda (m)
              (map1-n #'(lambda (n)
                          (let ((x (+ m n)))
                            (if (>= x len)
                                `(nth ,(- x len) ,src)
                                `(nth ,(1- x) ,rest))))
                      len))
          (- len 2)))
```

図 54　部分リストに渡る再帰のためのマクロ.

グというより仕様だが，最低でも明記しておくべきだ.

さらに強力な反復構造を定義する必要があるときにはマクロは必要不可欠だ．第 54図には dolist の一般化を 2 つ示した．両方とも変数の組をリストの隣り合った部分リストに束縛した状態で本体部を評価する．例えば do-tuples/o は引数を 2 つ取り，リストの要素に渡って反復を行う.

```
> (do-tuples/o (x y) '(a b c d)
               (princ (list x y)))
(a b)(b c)(c d)
nil
```

同じ引数に対しては do-tuples/c も同じ動作をするが，こちらはリスト末尾で先頭につながる.

```
> (do-tuples/c (x y) '(a b c d)
               (princ (list x y)))
(a b)(b c)(c d)(d a)
nil
```

どちらのマクロも本体内で陽に return が使われない限り nil を返す.

この種の反復は経路の概念を何らかの形で扱うプログラムでしばしば必要になる．名前の末尾の/o と/c は，それぞれが開いた (open) 経路と閉じた (closed) 経路を探索することを表すつもりで付けられた．例えば points が点のリス

```
(do-tuples/c (x y z) '(a b c d)
             (princ (list x y z)))
```

は次のように展開される：

```
(let ((#:g2 '(a b c d)))
  (when (nthcdr 2 #:g2)
    (labels ((#:g4 (x y z)
               (princ (list x y z))))
      (do ((#:g3 #:g2 (cdr #:g3)))
        ((not (nthcdr 2 #:g3))
         (#:g4 (nth 0 #:g3)
               (nth 1 #:g3)
               (nth 0 #:g2))
         (#:g4 (nth 1 #:g3)
               (nth 0 #:g2)
               (nth 1 #:g2))
         nil)
        (#:g4 (nth 0 #:g3)
              (nth 1 #:g3)
              (nth 2 #:g3))))))
```

図 55　do-tuples/c の呼び出しの展開形.

トで，(drawline x y) が x と y との間に線分を描くものとしよう．最初の点から最後の点まで折れ線を描くにはこうすればよい．

```
(do-tuples/o (x y) points (drawline x y))
```

しかし points が多角形の頂点のリストだったら，その外周を描くにはこうすればよい．

```
(do-tuples/c (x y) points (drawline x y))
```

第 1 引数として渡された引数のリストは任意の長さでよく，反復はその長さと同数の変数の組に渡って行われる．引数が 1 個だけのときは，共に dolist に縮退する．

```
> (do-tuples/o (x) '(a b c) (princ x))
abc
NIL
> (do-tuples/c (x) '(a b c) (princ x))
ABC
NIL
```

do-tuples/c の定義は do-tuples/o より複雑になっている．これはリスト末尾に届いたときに先頭につながるようにしているせいだ．n 個の引数があれば，do-tuples/c は値を返す前に $n-1$ 回余計に反復を行わなければならない．

```
> (do-tuples/c (x y z) '(abcd)
               (princ (list x y z)))
(A B C)(B C D)(C D A)(D A B)
NIL

> (do-tuples/c (wxyz)'(abcd)
               (princ (list w x y z)))
(A B C D)(B C D A)(C D A B)(D A B C)
NIL
```

do-tuples/c の呼び出しの 1 つ目の展開形は第 55に示した．生成し辛いのは，リスト先頭へつながることを表す呼び出しの連続だ．これらの呼び出しは（この場合はそれらのうち 2 つ）dt-args によって生成される．

10.5　複数の値に渡る反復

組込みマクロ do の歴史は多値よりも古い．幸い do は新しい状況に適応するように進化することができる．Lisp の進化はプログラマの手の中にあるからだ．第 56には多値に対応した do*の変種を示した．mvdo*を使えば，始めの節は

```
(defmacro mvdo* (parm-cl test-cl &body body)
  (mvdo-gen parm-cl parm-cl test-cl body))

(defun mvdo-gen (binds rebinds test body)
  (if (null binds)
      (let ((label (gensym)))
        `(prog nil
               ,label
               (if ,(car test)
                   (return (progn ,@(cdr test))))
               ,@body
               ,@(mvdo-rebind-gen rebinds)
               (go ,label)))
      (let ((rec (mvdo-gen (cdr binds) rebinds test body)))
        (let ((var/s (caar binds)) (expr (cadar binds)))
          (if (atom var/s)
              `(let ((,var/s ,expr)) ,rec)
              `(multiple-value-bind ,var/s ,expr ,rec))))))

(defun mvdo-rebind-gen (rebinds)
  (cond ((null rebinds) nil)
        ((< (length (car rebinds)) 3)
         (mvdo-rebind-gen (cdr rebinds)))
        (t(cons (list (if (atom (caar rebinds))
                          'setq
                          'multiple-value-setq)
                      (caar rebinds)
                      (third (car rebinds)))
                (mvdo-rebind-gen (cdr rebinds))))))
```

図 56　多値に対応した do*.

複数個の変数を束縛できる.

```
> (mvdo* ((x 1 (1+ x))
          ((y z) (values 0 0) (values z x)))
         ((> x 5) (list x y z))
         (princ (list x y z)))
(1 0 0)(2 0 2)(3 2 3)(4 3 4)(5 4 5)
(656)
```

この種の反復は，例えばインタラクティブなグラフィックス・プログラムで便利になる．それにはしばしば座標や領域などの複数の数量を扱う必要があるからだ.

単純なインタラクティブ・ゲームを書きたいとしよう．目的は追ってくる 2 個の敵に挟まれるのを避けることだ．敵が同時にぶつかってきたら，ゲームは負け．それぞれ単独でぶつかると，勝ちになる．第 57図には，このゲームのメイン・ループを mvdo*を使って書く方法を示した.

またローカル変数を並列して束縛する mvdo を書くこともできる.

```
> (mvdo ((x 1 (1+ x))
         ((y z) (values 0 0) (values z x)))
        ((> x 5) (list x y z))
        (princ (list x y z)))
(1 0 0)(2 0 1)(3 1 2)(4 2 3)(5 3 4)
(645)
```

do の定義に psetq が必要な点については asd ページで説明した．mvdo を定義するには，多値に対応した psetq が必要だ．そういうものは Common Lisp にはないので，自分で書かなければならない．それを第 58に示した．この新しいマクロは次のように動作する.

```
> (let ((w 0) (x 1) (y 2) (z 3))
    (mvpsetq (w x) (values 'a 'b) (y z) (values w x))
```

```
(mvdo* (((px py) (pos player)   (move player mx my))
        ((x1 y1) (pos obj1)     (move obj1 (- px x1)
                                           (- py y1)))
        ((x2 y2) (pos obj2)     (move obj2 (- px x2)
                                           (- py y2)))
        ((mx my) (mouse-vector) (mouse-vector))
        (win     nil            (touch obj1 obj2))
        (lose    nil            (and (touch obj1 player)
                                     (touch obj2 player))))
       ((or win lose) (if win 'win 'lose))
       (clear)
       (draw obj1)
       (draw obj2)
       (draw player))
```

(pos obj) は 2 つの値 x と y を返す．これらは obj の位置を示す．最初は，3 つの物体はランダムな位置に置かれる．
(move obj dx dy) はオブジェクト obj の型とヴェクタ dx, dy に従って obj を移動させる．新しい位置を示す 2 つの値 x と y を返す．
(mouse-vector) はマウスの現在の位置を示す 2 つの値 dx と dy を返す．
(touch obj1 obj2) は，obj1 と obj2 が接触しているならば真を返す．
(clear) はゲーム領域を消去する．
(draw obj) は obj を現在位置に従って描画する．

図 57　挟みゲーム．

```
(defmacro mvpsetq (&rest args)
  (let* ((pairs (group args 2))
         (syms (mapcar #'(lambda (p)
                           (mapcar #'(lambda (x) (gensym))
                                   (mklist (car p))))
                       pairs)))
    (labels ((rec (ps ss)
                  (if (null ps)
                      `(setq
                         ,@(mapcan #'(lambda (p s)
                                       (shuffle (mklist (car p))
                                                s))
                                   pairs syms))
                      (let ((body (rec (cdr ps) (cdr ss))))
                        (let ((var/s (caar ps))
                              (expr (cadar ps)))
                          (if (consp var/s)
                              `(multiple-value-bind ,(car ss)
                                 ,expr
                                 ,body)
                              `(let ((,@(car ss) ,expr))
                                 ,body)))))))
      (rec pairs syms))))

(defun shuffle (x y)
  (cond ((null x) y)
        ((null y) x)
        (t (list* (car x) (car y)
                  (shuffle (cdr x) (cdr y))))))
```

図 58　多値に対応した psetq.

```
(defmacro mvdo (binds (test &rest result) &body body)
  (let ((label (gensym))
        (temps (mapcar #'(lambda (b)
                           (if (listp (car b))
                               (mapcar #'(lambda (x)
                                           (gensym))
                                       (car b))
                               (gensym)))
                       binds)))
    `(let ,(mappend #'mklist temps)
       (mvpsetq ,@(mapcan #'(lambda (b var)
                              (list var (cadr b)))
                          binds
                          temps))
       (prog ,(mapcar #'(lambda (b var) (list b var))
                      (mappend #'mklist (mapcar #'car binds))
                      (mappend #'mklist temps))
             ,label
             (if ,test
                 (return (progn ,@result)))
             ,@body
             (mvpsetq ,@(mapcan #'(lambda (b)
                                    (if (third b)
                                        (list (car b)
                                              (third b))))
                                binds))
             (go ,label)))))
```

図 59　多値の束縛に対応した do.

```
    (list wxyz))
(AB01)
```

mvpsetq の定義は 3 つのユーティリティ関数に依存している：mklist (p. sdf)，group (p. dfg) に，ここで定義した shuffle だ．それは 2 つのリストを交互に組み合わせる働きをする．

```
> (shuffle '(a b c) '(1 2 3 4))
(A1B2C34)
```

mvpsetq を使い，mvdo は第 59図のように書ける．condlet と同様，元のマクロ呼び出しに変更を加えるのを避けるため，このマクロは mapcar ではなく mappend を使っている．慣用法 mappend-mklist はツリーを 1 段階だけ平たくする．

```
> (mappend #'mklist '((a b c) d (e (f g) h) ((i)) j))
(ABCDE(FG)H(I)J)
```

このかなり大きなマクロの理解を助けるため，第 60図には展開形の例も示した．

10.6　マクロの必要性

引数を評価から保護する方法はマクロだけではない．引数をクロージャで括る方法もある．条件付き評価と反復評価は，どちらも本質的にマクロが必要な訳ではないので，似たようなものだ．例えば if を関数として書くことができる．

```
(defun fnif (test then &optional else)
  (if test
      (funcall then)
      (if else (funcall else))))
```

then 部と else 部に当たる引数はクロージャとして表現することで保護できる．だから次のようにはせず，

```
(if (rich) (go-sailing) (rob-bank))
```

代わりに次のようにすることになる．

```
(mvdo ((x 1 (1+ x))
       ((y z) (values 0 0) (values z x)))
      ((> x 5) (list x y z))
      (princ (list x y z)))
```

これは次のように展開される：

```
(let (#:g2 #:g3 #:g4)
  (mvpsetq #:g2 1
           (#:g3 #:g4) (values 0 0))
  (prog ((x #:g2) (y #:g3) (z #:g4))
        #:g1
        (if (> x 5)
            (return (progn (list x y z))))
        (princ (list x y z))
        (mvpsetq x (1+ x)
                 (y z) (values z x))
        (go #:g1)))
```

図 60　mvdo の呼び出しの展開形.

```
(fnif (rich)
      #'(lambda () (go-sailing))
      #'(lambda () (rob-bank)))
```

ただ条件付き評価だけを実現したいなら，マクロが絶対必要というわけではない．マクロはプログラムを明確にしてくれるだけだ．しかし引数の式の中身を切り分けたり，引数として渡された変数を束縛するためにはマクロが必要になる．

同じことが反復用のマクロにも当てはまる．マクロは本体部の式の前に反復構造を定義する唯一の方法だが，ループ本体が関数そのもので括られている限り，関数で反復を実現できる[*27]．例えば組込み関数 mapc は dolist の関数版だ．

```
(dolist (b bananas)
  (peel b)
  (eat b))
```

この式は次と同じ副作用を持つ．

```
(mapc #'(lambda (b)
          (peel b)
          (eat b))
      bananas)
```

(ただし前者は nil を，後者はリスト bananas を返すが．) 同様に forever も，式本体をクロージャとして渡すようにすれば関数として実装できる．

```
(defun forever (fn)
  (do ()
    (nil)
    (funcall fn)))
```

しかし forever でもそうだが，反復構造は普通は反復だけが目的なのではない．束縛と反復の組み合わせが目的なのが普通だ．関数を使うと，束縛の方の見込みは限られる．変数をリストの隣り合う要素にそれぞれ束縛したいときには，対応関数のどれかを使えばよい．しかし要求されることがずっと込み入ってくると，マクロを書かざるを得ないだろう．

[*27] 引数を関数に括る必要のない反復用関数を書くのは不可能ではない．引数として渡された式で eval を呼ぶ関数が定義できる．それが eval の使いかたとして良くないことの説明には，wer ページを参照.

11　汎変数

第 8 章では，マクロの利点の 1 つは引数を変形できる能力だということに触れた．その種のマクロの 1 つが setf だ．この章では，setf の持つ隠れた意味に視点を向け，その上に構築できるマクロの例を幾つか示す．

setf の上に適切なマクロを構築することは，驚く程難しい．この話題への導入として，最初の節では少々まずい所のある単純な例を示す．次の節ではそれの何がまずいのかを説明し，改善方法を示す．その後の 2 節では setf の上に構築されたユーティリティの例を提示し，最後の節で setf の逆に当たる独自のオペレータの定義方法を説明する．

11.1　概念

組込みマクロ setf は setq の一般化に当たる．setf の第 1 引数はただの変数でなく関数やマクロの呼び出しであってもよい．

```
> (setq lst '(a b c))
(A B C)
> (setf (car lst) 480)
480
> lst
(480 B C)
```

一般的には (setf x y) は「x が y に評価されるようにしておいてくれ」という指示として理解できる．setf はマクロなので，引数の内部を見てそのような指示を実現するには何をすればよいかを判断できる．(展開後の) 1 引数がシンボルだったら，setf は単に setq に展開される．しかし第 1 引数が別の変数を求めるための式だったときは，setf は対応する確定した命令 (assertion) に展開される．第 2 引数は定数なので，前の例は次のように展開されるかもしれない．

```
(progn (rplaca lst 480) 480)
```

このように変数を求める式から決定的な命令へ変換することはインヴァージョン (inversion) と呼ばれる．car，cdr，nth，aref，get，gethash 及び defstruct で定義されるアクセス関数等，Common Lisp で頻繁に使われる，オブジェクトにアクセスするための関数にはインヴァージョンに当たるものが予め定義されている．(完全な一覧は CLtL2 の 125 ページにある)

setf の第 1 引数として機能する式は汎変数（generalized variable）と呼ばれる．汎変数は強力な抽象化手段であることが分かっている．インヴァージョン可能な参照に展開されるマクロ呼び出しは全てそれ自体がインヴァージョン可能という点で，マクロ呼び出しは汎変数と似ている．

また setf の上にマクロも構築すれば，相乗効果によってかなり明確なプログラムができる．setf の上に構築できるマクロの 1 つは toggle だ[*28]．

```
(defmacro toggle (obj)                  ; 間違い
  `(setf ,obj (not ,obj)))
```

これは汎変数の値を切替える：

```
> (let ((lst '(a b c)))
    (toggle (car lst))
    lst)
(NIL B C)
```

さて次のアプリケーションを考えよう．誰かが——昼のドラマ劇場の脚本家，精力に溢れた社交界の人間，はたまた政党の事務員か——小さな町の住人の間の人間関係を全て記録したデータベースを運営したいとしよう．テーブルには住人の友人を記録する欄が必要だ．

```
(defvar *friends* (make-hash-table))
```

このハッシュ表のエントリはそれ自身がハッシュ表で，ここには友人の候補の名前が t または nil に結合されている．

```
(setf (gethash 'mary *friends*) (make-hash-table))
```

John を Mary の友人として登録するには，次のようにすることになる．

[*28] 以下の節で見るように，この定義は正しくない．

```
(setf (gethash 'john (gethash 'mary *friends*)) t)
```

町は 2 つのグループに分けられる．グループというものはいつでも互いに「仲間じゃないやつは敵だ」と言いたがるので，町の人間はどちらかのグループに属することを余儀なくされる．だから誰かがグループを移ると，元の友人は敵に，敵は友人に変わる．

組込みオペレータだけによって x が y の友人かどうかを切替えるには，次のようにすることになる．

```
(setf (gethash x (gethash y *friends*))
      (not (gethash x (gethash y *friends*))))
```

これはかなり複雑な式だ（もちろん setf なしではこの程度では済まなかっただろうが）．データベースに次のようなアクセス用マクロを定義していれば，

```
(defmacro friend-of (p q)
  `(gethash ,p (gethash ,q *friends*)))
```

このマクロと toggle とで，データベースの変更を扱うためにはかなり準備が整ったことになる．上の更新操作は簡潔に表現できるだろう．

```
(toggle (friend-of x y))
```

汎変数は美味しい健康食品のようなものだ．これによってできるプログラムは見事にモジュール化され，それでいて美しくエレガントだ．データ構造にマクロかインヴァージョン可能な関数を通じてアクセスを提供すれば，他のモジュールはデータ表現の詳細を知る必要のないまま setf でデータ構造を操作できる．

11.2 複数回の評価に関わる問題

前の節では toggle の最初の定義はまずい所があると警告した．

```
(defmacro toggle (obj)                  ; 間違い
  `(setf ,obj (not ,obj)))
```

これは第 10.1 節で説明した問題，つまり複数回の評価の元になる．問題は引数に副作用があるときに起きる．例えば lst がオブジェクトから成るリストで，それを使ってこう書いたとき，

```
(toggle (nth (incf i) lst))
```

$(i+1)$ 番目の要素の値を切替えているものと思うだろう．しかし toggle の現在の定義ではこのマクロ呼び出しは次のように展開される．

```
(setf (nth (incf i) lst)
      (not (nth (incf i) lst)))
```

これは i を 2 回増加させ，$(i+1)$ 番目の要素を $(i+2)$ 番目の要素の逆に設定する．よって次の例では，

```
> (let ((lst '(t nil t))
            (i -1))
        (toggle (nth (incf i) lst))
        lst)
(T NIL T)
```

toggle を呼んでも効果はないようだ．

toggle の引数として与えられた式を取り出し，setf の第 1 引数の場所に挿入するだけでは十分ではない．式の内部を見て，その動作を知らなければならないのだ．それが部分式を含むなら，それらが副作用を持つことを考慮し，分割して別個に評価しなければならない．一般的に言えば，これは複雑な仕事だ．

余計な手間を省くため，Common Lisp は setf の上に，ある限られた範囲のマクロを自動的に定義するマクロを提供している．このマクロは define-modify-macro という名前で，3 個の引数を取る．定義したいマクロの名前，（汎変数の後に）取るかもしれない付加的な引数，そして汎変数の新しい値を返す関数の名前だ[*29]．

define-modify-macro を使うと，define はこうして定義できる．

```
(define-modify-macro toggle () not)
```

[*29] これは一般的な意味での関数名だ：1+ または (lambda (x) (+ x 1)) のどちらでもよい．

```
(defmacro allf (val &rest args)
  (with-gensyms (gval)
    `(let ((,gval ,val))
       (setf ,@(mapcan #'(lambda (a) (list a gval))
                       args)))))

(defmacro nilf (&rest args) `(allf nil ,@args))

(defmacro tf (&rest args) `(allf t ,@args))

(defmacro toggle (&rest args)
  `(progn
     ,@(mapcar #'(lambda (a) `(toggle2 ,a))
               args)))

(define-modify-macro toggle2 () not)
```

図 61　汎変数に対して機能するマクロ.

換言すれば，これは「(toggle place) の形の式を評価するには，place の指定する位置を見つけ，そしてそこに蓄えられた値が val ならば，それを (not val) の値で置き換えること」と指示していることになる．次は新しいマクロを同じ例に使ったところだ.

```
> (let ((lst '(t nil t))
        (i -1))
    (toggle (nth (incf i) lst))
    lst)
(NIL NIL T)
```

この定義は正しい結果をもたらすが，更に一般化することもできる．setf と setq は任意の数の引数を取れるのだから，toggle もそうあるべきだ．第 61に示したように，この機能は define-modify-macro の上に別のマクロを定義することで追加できる.

11.3　新しいユーティリティ

この節では汎変数に対して機能する新しいユーティリティの例を幾つか示す．引数を setf にそのままの形で渡すためには，それらはマクロでなければならない.

第 61には，setf の上に定義した新しいマクロを 4 個示した．最初の allf は，複数の汎変数を同じ値に設定するためのものだ．さらにその上に nilf と tf が作られた．これらは引数をそれぞれ nil と t に設定する．これらは単純なマクロだが，プログラムに大きな違いを生む.

setq と同様，setf も複数の引数を取ることができる．すなわち他に設定したい変数と値の組だ.

```
(setf x 1 y 2)
```

これらのユーティリティも同様だが，半分の引数を省略することもできる．複数の変数を nil に設定したいときは,

```
(setf x nil y nil z nil)
```

ではなく，ただこうすればよい.

```
(nilf x y z)
```

最後のマクロ toggle は前の節で説明した．これは nilf と似ているが，それぞれの引数に逆の真理値を与える.

これら 4 個のマクロは代入用オペレータに関する重要な点をはっきり表している．オペレータを普通の変数のみに対して使うつもりでも，setq でなく setf に展開されるマクロを書いた方がよいということだ．第 1 引数がシンボルならば setf は結局 setq に展開されるのだ．setf の一般性をなんの引替もなく享受できるのだから，マクロ展開では setq を使う方が好ましいことはまずないだろう.

第 62図にはリスト末尾を破壊的に操作するマクロを 3 個示した．第 3.1 節では副作用を期待して

```
(nconc x y)
```

```
(define-modify-macro concf (obj) nconc)

(define-modify-macro conc1f (obj)
  (lambda (place obj)
    (nconc place (list obj))))

(define-modify-macro concnew (obj &rest args)
  (lambda (place obj &rest args)
    (unless (apply #'member obj place args)
      (nconc place (list obj)))))
```

図 62　汎変数に対するリスト操作.

を使うことは安全ではなく，代わりにこうしなければならないことに触れた.

```
(setq x (nconc x y))
```

この慣用法は concf の中に埋め込まれている．更に特化した conc1f と concnew とは，それぞれ push と pushnew とをリストの逆方向に使うようなものだ．conc1f は 1 個の要素をリスト末尾に追加する．concnew は同様だが要素が既に含まれていないときに限る.

第 2.2 節で，関数名はシンボルの他に λ 式であってもよいと書いた．そのため conc1f の定義のように，λ 式を丸ごと define-modify-macro の第 3 引数に使っても結構だ．ert ページの conc1 を使えば，このマクロは次のようにも書ける.

```
(define-modify-macro conc1f (obj) conc1)
```

第 62図のマクロはある留保の元で使うべきだ．すなわち，末尾に要素を追加していってリストを作ろうと思っているのなら，push を使い，最後にリストを nreverse でインヴァージョンする方が好ましいという点だ．リストの先頭に何か操作を行う方が，末尾に行うよりも楽に済む．末尾に操作を行うには，まずそこまで到達しなければならないからだ．Common Lisp に前者を行うオペレータが多く，後者のためのものが少ないのは，おそらく効率的なプログラミングを励行するためだろう.

11.4　更に複雑なユーティリティ

define-modify-macro だけで setf の上にどんなマクロも構築できるわけではない．例えば，汎変数に破壊的に関数を適用するマクロ f を定義したいとしよう．組込みマクロ incf は，+ に setf を使うことの省略だ．次のようにしなくとも,

```
(setf x (+ x y))
```

ただこうすればよい.

```
(incf x y)
```

新しく作る f はこの考えの一般化だ．incf は + の呼び出しに展開されるが，f は第 1 引数として渡されたオペレータの呼び出しに展開される．例えば cvb ページの scale-objs の定義では，こう書かなければならなかった.

```
(setf (obj-dx o) (* (obj-dx o) factor))
```

f を使えば，これは次のようになる.

```
(_f * (obj-dx o) factor)
```

f の定義で犯す誤りは，次のようなものだろう.

```
(defmacro _f (op place &rest args)          ; 誤り
  `(setf ,place (,op ,place ,@args)))
```

残念だが，define-modify-macro では適切に f を定義することはできない．それは汎変数に適用されるオペレータが引数として与えられるからだ.

このように更に複雑なマクロは手作りで定義しなければならない．そういったマクロを書き易くするため，Common Lisp は関数 get-setf-method を提供している．これは汎変数を取り，その値を取得または設定するために必要な全ての情報を返すものだ．次の式を手で展開することで，この情報の使い方を示す.

```
(incf (aref a (incf i)))
```

get-setf-method を汎変数に対して呼び出すと，マクロ展開内での使うための値が 5 個返される．

```
> (get-setf-method '(aref a (incf i)))
(#:G4 #:G5)
(A (INCF I))
(#:G6)
(SYSTEM:SET-AREF #:G6 #:G4 #:G5)
(AREF #:G4 #:G5)
```

最初の 2 個の値は，一時変数とそれに代入するべき値のリストだ．だから展開形を次のように始めることができる．

```
(let* ((#:g4 a)
       (#:g5 (incf i)))
  ...)
```

一般的な状況では値の式は先に出てきた式を参照しているかも知れないので，これらの束縛は let* 内で作られるべきだ．3 番目と[*30]5 番目の値は，別の一時変数と汎変数の元の値を返す式だ．この値に 1 を加えたいので，後者を 1+ の呼び出し内に入れる．

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  ...)
```

最後に get-setf-method の返す 4 番目の値とは，新しい束縛のスコープ内で行われなければならない代入操作だ．

```
(let* ((#:g4 a)
       (#:g5 (incf i))
       (#:g6 (1+ (aref #:g4 #:g5))))
  (system:set-aref #:g6 #:g4 #:g5))
```

この式は Common Lisp の一部ではない内部関数を参照していることの方が多い．普通 setf はそれらの存在を隠蔽しているが，それらはどこかに存在していなければならない．それに関することはみな実装依存なので，可搬性を持たせたいコードでは system:set-aref のような関数を直接参照せず，get-setf-method の返した値を使うようにするべきだ．

さて f を実装するためには，incf を手で展開するときに行ったこととほとんど同じことを行うマクロを書くことになる．唯一の違いは，let* 内の最後の式を 1+ の呼び出しの中に入れるのではなく，f の引数から作られた式の中に入れることだ．f の定義は第 63図に示したようになる．

このユーティリティはとても便利なものだ．今これが手に入ったので，例えば名前を持った関数ならメモワイズ技法（第 5.3 節）を適用したものと簡単に置き換えることができる[*31]．foo をメモワイズ機能付きにするには，このようにすればよい．

```
(_f memoize (symbol-function 'foo))
```

f を利用すると，setf の上に他のマクロを定義することも簡単になる．例えば conc1f（第 62図）は次のように定義できる．

```
(defmacro conc1f (lst obj)
  `(_f nconc ,lst (list ,obj)))
```

第 63図には setf の上に構築した便利なマクロを他にも示した．2 番目の pull は，組込み関数 pushnew のコンプリメント (complement) として意図されている．これら 2 つは，push と pop が一層明確になったようなものだ．pushnew はオブジェクトが既にリストのメンバでないときだけそれをリストにプッシュし，pull は選択された要素をリストから破壊的に削除する．pull の定義内の&rest パラメータにより，pull は delete と同じキーワード引数を受け付けるようになっている．

[*30] 第 3 の値は現在のところ常に 1 要素から成るリストだ．これがリストとして与えられるのは，汎変数に複数の値を蓄える（これまでのところ実現されていない）可能性を提供するためだ．

[*31] しかし組込み関数にはこの方法でメモワイズ機能を付けるべきではない．Common Lisp は組込み関数の再定義を禁じている．

```
(defmacro _f (op place &rest args)
  (multiple-value-bind (vars forms var set access)
    (get-setf-method place)
    `(let* (,@(mapcar #'list vars forms)
            (,(car var) (,op ,access ,@args)))
       ,set)))

(defmacro pull (obj place &rest args)
  (multiple-value-bind (vars forms var set access)
    (get-setf-method place)
    (let ((g (gensym)))
      `(let* ((,g ,obj)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete ,g ,access ,@args)))
         ,set))))

(defmacro pull-if (test place &rest args)
  (multiple-value-bind (vars forms var set access)
    (get-setf-method place)
    (let ((g (gensym)))
      `(let* ((,g ,test)
              ,@(mapcar #'list vars forms)
              (,(car var) (delete-if ,g ,access ,@args)))
         ,set))))

(defmacro popn (n place)
  (multiple-value-bind (vars forms var set access)
    (get-setf-method place)
    (with-gensyms (gn glst)
                  `(let* ((,gn ,n)
                          ,@(mapcar #'list vars forms)
                          (,glst ,access)
                          (,(car var) (nthcdr ,gn ,glst)))
                     (prog1 (subseq ,glst 0 ,gn)
                       ,set)))))
```

図 63　setf の上に作る更に複雑なマクロ.

```
> (setq x '(1 2 (a b) 3))
(12(AB)3)
> (pull 2 x)
(1 (A B) 3)
> (pull '(a b) x :test #'equal)
(1 3)
>x
(1 3)
```

このマクロはあたかも次のように定義されたかのように考えられる.

```
(defmacro pull (obj seq &rest args)                    ; 誤り
   `(setf ,seq (delete ,obj ,seq ,@args)))
```

しかしこう定義すると評価順と評価回数の両方に関して問題が生じる. 単純に modify-macro で定義した pull も作れる.

```
(define-modify-macro pull (obj &rest args)
   (lambda (seq obj &rest args)
        (apply #'delete obj seq args)))
```

しかし modify-macros は汎変数を第 1 引数として取らなければならないので, 第 1, 第 2 引数を逆順に与えなければならないが, これはやや直観に反する.

更に一般的な pull-if は関数を引数に取り, delete でなく delete-if に展開される.

```
(defmacro sortf (op &rest places)
  (let* ((meths (mapcar #'(lambda (p)
                            (multiple-value-list
                              (get-setf-method p)))
                        places))
         (temps (apply #'append (mapcar #'third meths))))
    `(let* ,(mapcar #'list
                    (mapcan #'(lambda (m)
                                (append (first m)
                                        (third m)))
                            meths)
                    (mapcan #'(lambda (m)
                                (append (second m)
                                        (list (fifth m))))
                            meths))
       ,@(mapcon #'(lambda (rest)
                     (mapcar
                       #'(lambda (arg)
                           `(unless (,op ,(car rest) ,arg)
                              (rotatef ,(car rest) ,arg)))
                       (cdr rest)))
                 temps)
       ,@(mapcar #'fourth meths))))
```

図 64　引数を整列させるマクロ.

```
> (let ((lst '(1 2 3 4 5 6)))
        (pull-if #'oddp lst)
        lst)
(2 4 6)
```

これら 2 つのマクロからは，一般的なポイントが他にも分かる．基盤となる関数がオプショナル引数をとるときは，その上に構築されたマクロもそうなるべきだという点だ．pull と pull-if は両方ともオプショナル引数を内部の delete に渡すようになっている．

第 63図の最後のマクロ popn は，pop の一般化だ．リストに 1 要素だけをポップするのではなく，任意の部分リストをポップして返す.

```
> (setq x '(a b c d e f))
(ABCDEF)
> (popn 3 x)
(ABC)
>x
(DEF)
```

第 64には引数を整列させるマクロを示した．x と y が変数で，x の値の方が小さくはないようにしたいときは，こうすればよい.

```
(if (> y x) (rotatef x y))
```

しかしこれを 3 個以上の変数について行おうとすると，コードはたちまち肥大化する．これを手で書く代わりに，sortf に書かせることができる．このマクロは比較オペレータと任意個の汎変数を取り，それらの値がオペレータによって指定された順に並ぶまで値を入れ換える．一番単純な場合では，引数は普通の変数になる.

```
> (setq x 1 y 2 z 3)
3
> (sortf > x y z)
3
> (list x y z)
(3 2 1)
```

```
(sortf > x (aref ar (incf i)) (car lst))
```

は（ある処理系では）次のように展開される：

```
(let* ((#:g1 x)
       (#:g4 ar)
       (#:g3 (incf i))
       (#:g2 (aref #:g4 #:g3))
       (#:g6 lst)
       (#:g5 (car #:g6)))
  (unless (> #:g1 #:g2)
    (rotatef #:g1 #:g2))
  (unless (> #:g1 #:g5)
    (rotatef #:g1 #:g5))
  (unless (> #:g2 #:g5)
    (rotatef #:g2 #:g5))
  (setq x #:g1)
  (system:set-aref #:g2 #:g4 #:g3)
  (system:set-car #:g6 #:g5))
```

図 65　sortf の呼び出しの展開形.

一般的には，それらは任意のインヴァージョン可能な式であってよい．ここで cake が誰かのケーキの取り分を返すインヴァージョン可能な関数であり，bigger がケーキに関して定義された比較関数だとしよう．moe のケーキは少なくとも larry のケーキ程はあり，それは少なくとも curly のケーキ程はあるという規則を立てたいならば，こうすればよい.

```
(sortf bigger (cake 'moe) (cake 'larry) (cake 'curly))
```

sortf の定義は, 概形としては f の定義に似ている．最初に let* があり, get-setf-method の返した一時変数が汎変数の初期値に束縛される．sortf の中核は真中の式 mapcon であり，これがその一時変数を整列させるためのコードを生成する．マクロのこの部分が生成するコードは引数の数に関して指数的に増大する．整列の後, get-setf-method の返した式を使って汎変数に再び値が代入される．使われているアルゴリズムは $O(n^2)$ オーダのバブル・ソートだが, このマクロは大量の引数について呼ばれることを意図したものではない.

第 65図には，sortf の呼び出しの展開形を示した．最初の let* で，引数とその部分式が注意深く左から右の順に評価される．そしてその次の 3 個の式で一時変数の値を比較し，場合によっては値を入れ換える．第 1 の一時変数が第 2 と比較され，次に第 1 と第 3 が比較され，次に第 2 と第 3 が比較される．最後に汎変数は左から右の順に再代入を受ける．問題になることは滅多にないが，マクロの引数は評価と同様に普通は左から右の順で値を代入されるべきだ.

f や sortf 等のオペレータには，関数の引数を取る関数とのある種の類似性がある．しかしこれらは全く違ったものとして理解されるべきだ．find-if のような関数は関数を取ってそれを呼び出す．f のようなマクロは関数の名前を取り，それを式の car 部に置く．f と sortf は共に関数の引数を取るように書くこともできた．例えば f を次のように書いて,

```
(defmacro _f (op place &rest args)
  (let ((g (gensym)))
    (multiple-value-bind (vars forms var set access)
      (get-setf-method place)
      `(let* ((,g ,op)
              ,@(mapcar #'list vars forms)
              (,(car var) (funcall ,g ,access ,@args)))
         ,set))))
```

(f #'+ x 1) として呼び出すこともできた．しかし元の f はこれができることを全てできるし，扱うものが名前なのでマクロや特殊式の名前も受け付ける．+ の他にも，例えば nif（rty ページ）も使える.

```
> (let ((x 2))
    (_f nif x 'p 'z 'n)
    x)
P
```

```
(defvar *cache* (make-hash-table))

(defun retrieve (key)
  (multiple-value-bind (x y) (gethash key *cache*)
    (if y(values x y)
        (cdr (assoc key *world*)))))

(defsetf retrieve (key) (val)
  `(setf (gethash ,key *cache*) ,val))
```

図 66　非対称なインヴァージョン.

11.5　インヴァージョンを定義する

第 12.1 節ではインヴァージョン可能な参照に展開される任意のマクロ呼び出しは，それ自身がインヴァージョン可能だということを説明した．しかし，オペレータをインヴァージョン可能にするだけのためにそれをマクロとして定義する必要はない．defsetf を使えば，Lisp に任意の関数やマクロ呼び出しをインヴァージョンする方法を教えることができる．

このマクロの使い方は 2 通りある．一番単純な場合では，引数は 2 個のシンボルになる．

```
(defsetf symbol-value set)
```

複雑な形では，defsetf の呼び出しは defmacro の呼び出しと似たようなものになり，更新された値の式のための仮引数が加わる．例えば，次のようにして car に対するインヴァージョンを定義できる．

```
(defsetf car (lst) (new-car)
   `(progn (rplaca ,lst ,new-car)
           ,new-car))
```

defmacro と defsetf には，重要な違いが 1 個ある．後者は引数のために自動的に gensym を生成する点だ．上の定義の下では，(setf (car x) y) は次のように展開されるだろう．

```
(let* ((#:g2 x)
       (#:g1 y))
  (progn (rplaca #:g2 #:g1)
         #:g1))
```

そのため変数捕捉及び評価の回数や順番に煩わされずに defsetf の展開関数を書くことができる．

CLtL2 準拠の Common Lisp では，setf に対するインヴァージョンを defun で直接定義できる．よって上の例は次のようにも書ける．

```
(defun (setf car) (new-car lst)
  (rplaca lst new-car)
  new-car)
```

このような関数では，更新された値が第 1 引数になるべきだ．また，この値を関数の値として返すことも慣習になっている．

これまでの例では汎変数とはデータ構造内の場所を参照するためのものかのように示唆してきた．悪者が人質を地下迷宮の中に連れ去ると，彼女を救うヒーローは外に連れ出すことになる．悪者もヒーローも同じ道を通るが，その方向は逆だ．setf がこのように動作しなければならないという印象を持たれても驚きはしない．定義済みのインヴァージョンは全てこの形式になっているようだからだ．実際は，「場所」はインヴァージョンを受ける仮引数に対する慣用名に過ぎない．

原則的には，setf はもっと一般的なものだ．アクセス用の式とそのインヴァージョンは同じデータ構造に対して働く必要すらない．何かのアプリケーションで，データベースの更新をキャッシュしたいとしよう．これが必要になるのは，例えば動作中に実際に更新を行うのが非効率的だったり，一貫性を保つために更新点をコミット前に検証しなければならないときだ．

world が実際のデータベースだとしよう．単純化のため，これは alist で，その要素は (⟨ キー ⟩ . ⟨ 値 ⟩) という形式だとする．第 66図には retrieve という名前の検索関数を示した．*world* が次の状態のときは，

```
(defun avg (&rest args)
  (/ (apply #'+ args) (length args)))

(defmacro avg (&rest args)
  `(/ (+ ,@args) ,(length args)))
```

図 67　平均値を得るとき，計算処理をコンパイル時にずらす．

```
((a . 2) (b . 16) (c . 50) (d . 20) (f . 12))
```

次のように動作する．

```
> (retrieve 'c)
50
```

car の呼び出しと異なり，retrieve の呼び出しはデータ構造の特定の場所を参照しているのではない．返り値は 2 箇所の場所のうちどちらかから得られることになる．そして retrieve のインヴァージョン（やはり第 66図に示した）は，その片方を参照するだけだ．

```
> (setf (retrieve 'n) 77)
77
> (retrieve 'n)
77
T
```

この検索では第 2 の返り値 t が返るが，これは求める値がキャッシュ内に見つかったことを表す．

マクロそのものと同様に，汎変数も目覚しい力を持った抽象化手段だ．その真価はここで語ったことに留まらないだろう．個々の現場では汎変数の使用がエレガントで強力なプログラムにつながるような方法の方が発見し易いことは確かだ．しかし setf のインヴァージョンを新しい方法で使ったり，同様に便利な変換手法を発見することも可能かも知れない．

12　コンパイル時の計算処理

前の章ではマクロで実装しなければならないオペレータを何種類か論じた．この章で論じる種類の問題は，関数でも解決できるが，マクロの方が効率的になるようなものだ．第 8.2 節ではある状況においてマクロを使うことの長所と短所を列挙した．長所の中には「コンパイル時の計算処理」というものがあった．オペレータをマクロとして定義することで，作業の一部をオペレータが展開されるときに行わせることができる場合がある．この章ではこの可能性を活用するマクロを見ていく．

12.1　新しいユーティリティ

第 8.2 章ではマクロを使って計算処理をコンパイル時に行わせる可能性を提起した．そこでは例として，引数の平均値を返すマクロ avg を定義した．

```
> (avg pi 4 5)
4.047...
```

第 67図では avg を最初は関数として，次にマクロとして定義した．avg がマクロとして定義されると，length はコンパイル時に呼ばれることになる．マクロ版では，実行時に&rest パラメータを操作することの負担も回避している．多くの処理系では avg はマクロとして書いた方が速い．

展開時に引数の数を知っていることから来るような類の処理の節約は，in（sdf ページ）で得られるような処理の節約と組み合わせることができる．そこでは幾つかの引数の評価すら回避できたのだった．第 68図には 2 通りの most-of を示した．これは引数の過半数が真を返すならば真を返すものだ．

```
> (most-of t t t nil)
T
```

マクロ版は, in と同様に, 必要なだけの引数を評価するようなコードに展開される．例えば (most-of (a) (b) (c)) は次のような等価な式に展開される．

```
(defun most-of (&rest args)
  (let ((all 0)
        (hits 0))
    (dolist (a args)
      (incf all)
      (if a (incf hits)))
    (> hits (/ all 2))))

(defmacro most-of (&rest args)
  (let ((need (floor (/ (length args) 2)))
        (hits (gensym)))
    `(let ((,hits 0))
       (or ,@(mapcar #'(lambda (a)
                         `(and ,a (> (incf ,hits) ,need)))
                     args)))))
```

図 68　計算処理をコンパイル時にずらし，さらに回避する．

```
(let ((count 0))
     (or (and (a) (> (incf count) 1))
         (and (b) (> (incf count) 1))
         (and (c) (> (incf count) 1))))
```

一番都合が良い場合には，ちょうど過半数の引数だけが評価される．

マクロは特定の引数の値が知られているときにも，計算処理をコンパイル時にずらせるかも知れない．第 69図にはそのようなマクロの例を示した．関数 nthmost は数 n と数からなるリストを取り，リスト内で n 番目に大きい数を返す．他のシーケンス用関数と同様に，最初は 0 番目とされる．

```
> (nthmost 2 '(2 6 1 5 3 4))
4
```

関数版はとても単純に書かれている．それはリストを整列させ，それを引数に nth を呼ぶ．sort は破壊的なので，nthmost は整列前にリストをコピーしている．このように定義されると，nthmost は 2 点で非効率的になる．コンシングを起こす点と，関心があるのは大きい順に n 個までなのに引数のリスト全体を整列させる点だ．

n がコンパイル時に分かっていれば，この問題には別の方法で取り組める．第 69図の後半ではマクロ版の nthmost を定義した．このマクロが最初に行うことは，第 1 引数を調べることだ．第 1 引数が数定数でないときには，上で見たものと同じコードに展開される．第 1 引数が数定数だと，別の手段に移る．例えば皿の上の 3 番目に大きいクッキーを取りたいときは，それぞれのクッキーを順番に眺め回し，その間ずっとそれまでに見たクッキーの内の大きい順に 3 個を手に取っておけばよい．全てのクッキーを見た後は，手の中の 1 番小さいクッキーが求めていたものだ．n が小さい定数で，クッキーの総数に比例しないならば，この方法により，最初に全てのクッキーを整列させるときよりも少ない労力で求めるクッキーが手に入る．

n が展開時に知られている場合にはこの戦略に従う．上のマクロは展開中に n 個の変数を生成し，nthmost-gen を呼んで，それぞれのクッキーを見ているときに評価されなければならないコードを生成する．第 70図にはマクロ展開の例を示した．マクロ nthmost は元の関数と全く同じ動作をするが，apply の引数として渡せない点が違っている．マクロを使うことを正当化するのは，純粋に効率性という観点だ．マクロ版は実行時にコンシングを起こさず，更に n が小さい定数のときには比較回数が少なくて済む．

効率的なプログラムを得るには，このように大きなマクロを書くような骨折りが必要なのだろうか？この場合には，恐らくそうではないだろう．2 通りの nthmost は一般原則の例として意図されたものだ．幾つかの引数がコンパイル時に知られているならば，マクロを使って効率のよいコードを生成できる．恐らくこの可能性を利用するかどうかは，利益のためにどれ程の負担に耐えられるかということと，効率的なマクロ版を書くためにどれ程の労力が必要かということに依る．マクロ版の nthmost は長く複雑なので，極端な状況でしか書く価値はないだろう．しかしコンパイル時に得られる情報は，それを利用しないことに決めたときさえ含め，常に考慮に値する要素なのだ．

```
(defun nthmost (n lst)
  (nth n (sort (copy-list lst) #'>)))

(defmacro nthmost (n lst)
  (if (and (integerp n) (< n 20))
      (with-gensyms (glst gi)
                    (let ((syms (map0-n #'(lambda (x) (gensym)) n)))
                      `(let ((,glst ,lst))
                         (unless (< (length ,glst) ,(1+ n))
                           ,@(gen-start glst syms)
                           (dolist (,gi ,glst)
                             ,(nthmost-gen gi syms t))
                           ,(car (last syms))))))
      `(nth ,n (sort (copy-list ,lst) #'>))))

(defun gen-start (glst syms)
  (reverse
    (maplist #'(lambda (syms)
                 (let ((var (gensym)))
                   `(let ((,var (pop ,glst)))
                      ,(nthmost-gen var (reverse syms)))))
             (reverse syms))))

(defun nthmost-gen (var vars &optional long?)
  (if (null vars)
      nil
      (let ((else (nthmost-gen var (cdr vars) long?)))
        (if (and (not long?) (null else))
            `(setq ,(car vars) ,var)
            `(if (> ,var ,(car vars))
                 (setq ,@(mapcan #'list
                                 (reverse vars)
                                 (cdr (reverse vars)))
                       ,(car vars) ,var)
                 ,else)))))
```

図 69　コンパイル時に分かっている引数の利用.

12.2　例：Bèzier 曲線

with-系マクロ（第 11.2 節）と同様に，コンパイル時の処理のためのマクロは，汎用ユーティリティとしてよりもある特定のアプリケーションのためのものとして書かれることが多い．汎用ユーティリティは，コンパイル時にどれだけの情報を得られるだろうか？それは与えられた引数の数と，場合によればそれらの値だ．他の制約条件を使いたいならば，それらはきっと個々のプログラムに課されたものでなければならないだろう．

例として，この章では Bèzier（ベジェ）曲線の生成がマクロによりどれだけ高速化できるかを示す．Bèzier 曲線がインタラクティブに操作されるときは，高速に生成しなければならない．曲線が幾つの部分から成るかがあらかじめ分かっているなら，処理の大部分をコンパイル時に行えることが明らかになる．曲線生成ルーチンをマクロとして書くことで，計算済みの値をコード内に織り込むことができる．これより一般的な，値を配列に保持する最適化手法よりも，こちらの方が速いはずだ．

Bèzier 曲線は 4 個の点から定義される——2 個の端点と 2 個の制御点だ．2 次元について考えるとき，これらの点は曲線上の点の x 座標と y 座標を定める方程式の媒介変数表示を定義する．2 個の端点を (x0, y0) と (x3, y3) とし，2 個の制御点を (x1, y1) と (x2, y2) とすると，曲線上の点を定義する方程式は次のようになる．

$$x = (x_3 - 3x_2 + 3x_1 - x_0)u^3 + (3x_2 - 6x_1 + 3x_0)u^2 + (3x_1 - 3x_0)u + x_0$$

$$y = (y_3 - 3y_2 + 3y_1 - y_0)u^3 + (3y_2 - 6y_1 + 3y_0)u^2 + (3y_1 - 3y_0)u + y_0$$

```
(nthmost 2 nums)
```

これは次のように展開される．

```
(let ((#:g7 nums))
  (unless (< (length #:g7) 3)
    (let ((#:g6 (pop #:g7)))
      (setq #:g1 #:g6))
    (let ((#:g5 (pop #:g7)))
      (if (> #:g5 #:g1)
          (setq #:g2 #:g1 #:g1 #:g5)
          (setq #:g2 #:g5)))
    (let ((#:g4 (pop #:g7)))
      (if (> #:g4 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g4)
          (if (> #:g4 #:g2)
              (setq #:g3 #:g2 #:g2 #:g4)
              (setq #:g3 #:g4))))
    (dolist (#:g8 #:g7)
      (if (> #:g8 #:g1)
          (setq #:g3 #:g2 #:g2 #:g1 #:g1 #:g8)
          (if (> #:g8 #:g2)
              (setq #:g3 #:g2 #:g2 #:g8)
              (if (> #:g8 #:g3)
                  (setq #:g3 #:g8)
                  nil))))
    #:g3))
```

図 70　nthmost の展開形.

これらの方程式を 0 から 1 までの u の n 個の値について評価すれば，曲線上の点が n 個得られる．例えば曲線を 20 分割して表示したいときは，上の方程式を $u = .05, .1, ..., .95$ について評価することになる．方程式を $u = 0$ または 1 で評価する必要はない．$u = 0$ では開始端点 (x_0, y_0) が，$u = 1$ では終了端点 $(x3, y3)$ が得られるからだ．

明らかな最適化手段は，n を固定し，u，u^2... をあらかじめ計算し，そしてそれらを $(n-1) \times 3$ 配列に蓄えることだ．曲線生成ルーチンをマクロとして定義することで，効率をずっと上げることができる．n がコンパイル時に知られていれば，プログラムは曲線を描くコマンド n 個に展開できる．あらかじめ計算した u，u^2，...は，配列に蓄えるのではなく，マクロの展開形の中に定数値としてそのまま挿入できる．

第 71図にはこの戦略に基づく曲線生成マクロを示した．これは曲線を即座に描くのではなく，生成した座標を配列に蓄える．曲線がインタラクティブに動かされているときは，個々の曲線は 2 回ずつ描かれなければならない．表示するために 1 回，そして次を描く前に消去するために（訳注：背景色で）1 回だ．その間，どこかに座標を保存しなければならない．

$n = 20$ では，genbez は 21 個の setf に展開される．u，u^2，...はコード内に直接使われるので，それらを実行時に配列から探す負担と，開始時に計算する負担が削減できる．u，u^2，...と同様に，配列のインデックスも展開形内には定数として現れるので，(setf aref) の範囲チェックもコンパイル時に行われる．

12.3　応用

この後の章にはコンパイル時に利用できる情報を活用するマクロが他にも幾つか載っている．よい例は if-match（cvn ページ）だ．パターンマッチャは，2 つのシーケンス（変数を含んでよい）を比較し，変数に値を代入して 2 つのシーケンスを等しくする方法があるかを調べる．if-match の構成は，シーケンスの片方がコンパイル時に分かっていて，そちらだけが変数を含んでいるならば，マッチングを効率的に行えることを示している．実行時に 2 つのシーケンスを比較し，その間に生成された変数束縛を保持するリストを，コンシングで作り上げるのではなく，分かっているシーケンスから決定される単純な比較を行うコードをマクロに生成させ，束縛を Lisp の真の変数に蓄えることができる．

```
(defconstant *segs* 20)
(defconstant *du*       (/ 1-0 *segs*))
(defconstant *pts* (make-array (list (1+ *segs*) 2)))

(defmacro genbez (x0 y0 x1 y1 x2 y2 x3 y3)
  (with-gensyms (gx0 gx1 gy0 gy1 gx3 gy3)
                `(let ((,gx0 ,x0) (,gy0 ,y0)
                                  (,gx1 ,x1) (,gy1 ,y1)
                                  (,gx3 ,x3) (,gy3 ,y3))
                   (let ((cx (* (- ,gx1 ,gx0) 3))
                         (cy (* (- ,gy1 ,gy0) 3))
                         (px (* (- ,x2 ,gx1) 3))
                         (py (* (- ,y2 ,gy1) 3)))
                     (let ((bx (- px cx))
                           (by (- py cy))
                           (ax (- ,gx3 px ,gx0))
                           (ay (- ,gy3 py ,gy0)))
                       (setf (aref *pts* 0 0) ,gx0
                             (aref *pts* 0 1) ,gy0)
                       ,@(map1-n #'(lambda (n)
                                     (let* ((u (* n *du*))
                                            (u^2 (* u u))
                                            (u^3 (expt u 3)))
                                       `(setf (aref *pts* ,n 0)
                                              (+ (* ax ,u^3)
                                                 (* bx ,u^2)
                                                 (* cx ,u)
                                                 ,gx0)
                                              (aref *pts* ,n 1)
                                              (+ (* ay ,u^3)
                                                 (* by ,u^2)
                                                 (* cy ,u)
                                                 ,gy0))))
                                 (1- *segs*))
                       (setf (aref *pts* *segs* 0) ,gx3
                             (aref *pts* *segs* 1) ,gy3))))))
```

図 71　Bèzier 曲線を生成するためのマクロ.

また第 19-24 章で説明する埋め込み言語も，コンパイル時に得られる情報を大部分で活用している．埋め込み言語は一種のコンパイラなので，そのような情報を使うべきであるのは自然なことだ．一般則としては，マクロは精巧であればある程，より多くの制約を引数に課すことになり，これらの制約条件を効率的なコードの生成に利用できる確率が一層高まる.

13　アナフォリックマクロ

第 9 章では，変数捕捉をどれも問題として扱った——うっかりすると起きてしまい，プログラムに悪い影響だけを与えるものとして．この章では変数捕捉は建設的にも使えることを示す．それなしでは書けないような便利なマクロが幾つかあるのだ.

ある式が非 `nil` の値を返すかどうか調べ，そうならば値に何かを行いたいようなことは Lisp プログラミングでは珍しくない．式の評価に大きな負担がかかるならば，普通は次のようにしなければならないだろう.

```
(let ((result (big-long-calculation)))
  (if result
      (foo result)))
```

日常言語と同じように，これを次のように表現できたら楽ではないだろうか？

```
(if (big-long-calculation)
```

```
    (foo it))
```

変数捕捉を利用することで，まさにこのとおりに動作する if の変種を定義できる．

13.1 アナフォリックな変種オペレータ

日常言語では，アナフォラ（前方照応，anaphora）とは会話の中で前に出てきたことを指す表現のことだ．英語で一番馴染み深いアナフォラは恐らく「それ」で，「レンチを取ったら，それを机に置いてくれ」等と使われる（訳注：日本語ではそもそも省かれるので例が不自然になってしまった）．アナフォラは日常会話の中で大変便利なものだ——それを使わないことを想像してみよう——しかしそれはプログラミング言語には出てこない（訳注：Perl や Ruby では $_ で「最後に読み込んだ文字列」が表せる）．ほとんどの局面では，それでいい．アナフォリックな（前方照応的な，anaphoric）表現はしばしばひどく曖昧だが，現代のプログラミング言語は曖昧性を扱うようには設計されていないからだ．

しかし非常に制限された形式のアナフォラを，曖昧さを入れずに Lisp プログラムに導入することは可能だ．アナフォラは捕捉されたシンボルにそっくりだということが分かる．代名詞として働くシンボルを作り，そしてそれらのシンボルを捕捉するマクロを意図的に書くことで，アナフォラをプログラムに導入できる．

新型の if では，シンボル it が捕捉したいものだ．アナフォリックな if は簡潔に aif と呼ぶことにするが，次のように定義され，

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))
```

前の方の例と同じように使われる．

```
(aif (big-long-calculation)
  (foo it))
```

aif を使うときは，テスト節の返す結果にシンボル it が束縛されたままになる．上のマクロ呼び出しでは it はフリーのように見えるが，実際は式 (foo it) は aif の展開に伴い，シンボル it に束縛のあるコード内に挿入される．

```
(let ((it (big-long-calculation)))
  (if it (foo it) nil))
```

よってソースコード内ではフリーに見えるシンボルはマクロ展開によって束縛されたままになる．この章で示すアナフォリックマクロは，全て同じ手法に基づくものだ．

第 72図には幾つかの Common Lisp オペレータのアナフォリックな変種を示した．aif の次の awhen は，when をアナフォリックにしたものだ．

```
(awhen (big-long-calculation)
  (foo it)
  (bar it))
```

aif も awhen も頻繁に役立つが，awhile はアナフォリックでない while（vgp ページで定義された）より多く必要になるという点でアナフォリックマクロの中では恐らくユニークなものだろう．while と awhile のようなマクロは，プログラムが外部データを監視する必要がある状況で使われるのが典型的だ．そしてデータの監視を繰り返す間，それが状態を変化させるのをただ待っているのでなければ，普通は得られたオブジェクトに対し何かを行いたいことだろう．

```
(awhile (poll *fridge*)
  (eat it))
```

aand の定義はそれまでのものより少々複雑になっている．これはアナフォリックな and を与える．それぞれの引数の評価の間，it はその前の引数が返した値に束縛される[*32]．実用では，aand は条件に基づいてクエリを発行するプログラムで使われることが多い．

```
(aand (owner x) (address it) (town it))
```

[*32] and と or は一緒にして考えられがちだが，アナフォリックな or を書いても意味はないだろう．or の引数が評価されるのは，それより前の引数が nil に評価されたときだけだ．だから aor の中でアナフォリックシンボルを参照しても，何も便利なことはないだろう．

```
(defmacro aif (test-form then-form &optional else-form)
  `(let ((it ,test-form))
     (if it ,then-form ,else-form)))

(defmacro awhen (test-form &body body)
  `(aif ,test-form
        (progn ,@body)))

(defmacro awhile (expr &body body)
  `(do ((it ,expr ,expr))
     ((not it))
     ,@body))

(defmacro aand (&rest args)
  (cond ((null args) t)
        ((null (cdr args)) (car args))
        (t `(aif ,(car args) (aand ,@(cdr args))))))

(defmacro acond (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (sym (gensym)))
        `(let ((,sym ,(car cl1)))
           (if ,sym
               (let ((it ,sym)) ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

図 72　Common Lisp の標準オペレータのアナフォリックな変種.

これは x の所有者の住所（所有者がいるならば）のある町（住所があるならば）を返す（町があるならば）．aand なしでは，これは次のように書かなければならないだろう．

```
(let ((own (owner x)))
  (if own
      (let ((adr (address own)))
        (if adr (town adr)))))
```

aand の定義は，展開形がマクロ呼び出しの引数の数によって変わることを示している．引数が 0 個のときは，aand は普通の and と同様にただ t を返すだけだ．そうでなければ展開形は再帰的に生成される．再帰の段階毎に入れ子になった aif の連鎖に新しい層が追加される．

(aif ⟨ 第 1 引数 ⟩
 ⟨ 引数の残り ⟩)

大部分の再帰関数が nil を受け取る段階まで再帰を続けるのとは違い，aand の展開は引数が 1 個の時点で終結しなければならない．選択肢が 1 個もない段階まで再帰が進み過ぎると，展開形は必ず次のような形になる．

```
(aif  c1
        ...(aif  cn t)...)
```

このような式は必ず t か nil のどちらかを返すので，上の例が意図した通りに動作しなくなる．

第 10.4 節で，マクロが常に自分の呼び出しを含む展開形を生成するようでは，展開は永遠に終結しないと警告した．aand も再帰的だが，ベース・ケースでは展開形は aand を参照しないので安全だ．

最後の acond は，cond の節のテスト式以外でテスト式の返した値を使いたい場合のために作られた．(この状況はとてもよく現れるので，Scheme の処理系によっては cond の節の中でテスト式の返した値を使う方法を提供している．)

acond の節の展開形の中では，テスト式の結果は最初に gensym に保持される．これはシンボル it が節の残りの部分でのみ束縛されるようにするためだ．マクロが束縛を生成するときには，必ず可能なかぎり狭いスコープの中で生成

```
(defmacro alambda (parms &body body)
  `(labels ((self ,parms ,@body))
     #'self))

(defmacro ablock (tag &rest args)
  `(block ,tag
     ,(funcall (alambda (args)
                 (case (length args)
                   (0 nil)
                   (1 (car args))
                   (t `(let ((it ,(car args)))
                         ,(self (cdr args))))))
               args)))
```

図 73　アナフォリックなオペレータの続き.

するべきだ．ここで gensym をけちって，代わりに次のように it をテスト式の結果に直接束縛すると，

```
(defmacro acond (&rest clauses)              ; 誤り
  (if (null clauses)
      nil
      (let ((cl1 (car clauses)))
        `(let ((it ,(car cl1)))
           (if it
               (progn ,@(cdr cl1))
               (acond ,@(cdr clauses)))))))
```

it の束縛はその次のテスト式もスコープに含んでしまうだろう．

第 73図には標準オペレータのアナフォリックな変種の，更に複雑なものを示した．マクロ alambda は書かれているままの再帰関数を参照するためのものだ．書かれているままの再帰関数を参照したいときとはいつだろうか？シャープクォート付き λ 式を使えば書かれているままの関数にアクセスできる．

```
#'(lambda (x) (* x 2))
```

しかし第 2 章で説明したように，単純な λ 式だけでは再帰関数は表現できない．代わりに labels でローカル関数を定義しなければならない．次の関数（ert ページから再録）は，

```
(defun count-instances (obj lists)
  (labels ((instances-in (list)
             (if list
                 (+ (if (eq (car list) obj) 1 0)
                    (instances-in (cdr list)))
                 0)))
    (mapcar #'instances-in lists)))
```

オブジェクトとリストを引数に取り，リストの要素毎にそのオブジェクトが幾つ現れたかを返す．

```
> (count-instances 'a '((a b c) (d a r p a) (d a r) (a a)))
(1 2 1 2)
```

アナフォラを使えば書かれているままの再帰関数に相当するものが作れる．マクロ alambda は labels を使ってそのようなものを作るので，例えば階乗関数を表現するのに使える．

```
(alambda (x) (if (= x 0) 1 (* x (self (1- x)))))
```

alambda を使って count-instances と等価なものが次のように定義できる．

```
(defun count-instances (obj lists)
  (mapcar (alambda (list)
            (if list
                (+ (if (eq (car list) obj) 1 0)
                   (self (cdr list)))
                0))
          lists))
```

第 73，72図の他のマクロは全て it を捕捉するが，それらと異なり，alambda は self を捕捉する．alambda は labels に展開されるが，その中では self は定義されている関数自身に束縛されている．alambda は短いだけでなく，見慣れた λ 式と似ており，それを使っているコードを読み易くしている．

ablock の定義では新しいマクロが使われている．組み込み特殊式 block のアナフォリック版だ．block では引数は全て左から右の順に評価される．ablock でもそれは同じだが，それぞれの引数において変数 it は前の引数の値に束縛される．

このマクロは思慮を持って使うべきだ．便利なこともあるが，ablock はうまく関数的になれるはずのプログラムを命令的に変えてしまいがちだ．次のコードは，残念ながら典型的な汚い例だ．

```
> (ablock north-pole
    (princ "ho ")
    (princ it)
    (princ it)
    (return-from north-pole))
ho ho ho
NIL
```

意図的に変数捕捉を行うマクロが別のパッケージにエクスポートされるときにはいつでも，捕捉されるシンボルもエクスポートすることが必要だ．例えば aif がエクスポートされるときにはいつでも it もそうしなければならない．そうしないとマクロ定義内の it はマクロ呼び出し内で使われている it とは別のシンボルになってしまうだろう．

13.2 失敗

Common Lisp ではシンボル nil には少なくとも 3 種類の役割がある．まずなにより空リストであって，次のように働く．

```
> (cdr '(a))
NIL
```

空リストの他に，nil は次のように真理値の偽の表現にも使われる．

```
> (= 1 0)
NIL
```

そして最後に，関数は nil を返すことで失敗を表す．例えば組み込み関数 find-if はリストの要素で何かの条件を満たす最初のものを返す．そのような要素が見付からなければ，find-if は nil を返す．

```
> (find-if #'oddp '(2 4 6))
NIL
```

残念なことに上の場合は，find-if が検索に成功したが，検索したものが nil だった場合と区別がつかない．

```
> (find-if #'null '(2 nil 6))
NIL
```

nil で偽と空リストの両方を表現したところで，実用的には大きな問題は起きない．実際，むしろ便利なこともある．しかし find-if のような関数の返した結果に曖昧性が生まれるので，nil で失敗も表現するのは困ったことだ．

失敗と返り値の nil を区別する問題は，オブジェクトを検索する関数全てで現れる．Common Lisp はこの問題に 3 通りもの解決策を用意している．（多値を返す方法の次に）最もよく使われる方法は，とりあえずリスト構造を返しておくことだ．例えば assoc で検索失敗を見分けるのには何の問題もない．成功したときには assoc はキーと値の対全体を返すからだ．

```
> (setq synonyms '((yes . t) (no . nil)))
((YES . T) (NO))
> (assoc 'no synonyms)
(NO)
```

この方法に倣い，find-if の曖昧性が心配なときは，member-if を使うことにする．これは条件を満たす要素だけを返すのではなく，それから始まる cdr 部全体を返す．

```
> (member-if #'null '(2 nil 6))
(NIL 6)
```

多値が利用可能になって以来，この問題には新しい解決策が加わった．第 1 返り値をデータに，第 2 返り値を成功／失敗の表現に使うのだ．組み込み関数 gethash はそのように動作する．これは常に 2 個の値を返すが，2 番目は何かが見つかったかどうかを示すものだ．

```
> (setf edible                        (make-hash-table)
        (gethash 'olive-oil edible) t
        (gethash 'motor-oil edible) nil)
NIL
> (gethash 'motor-oil edible)
NIL
T
```

だから起こり得る 3 通りの場合を全て判別したいなら，次のような慣用法が使える．

```
(defun edible? (x)
  (multiple-value-bind (val found?) (gethash x edible)
    (if found?
        (if val 'yes 'no)
        'maybe)))
```

これによれば偽を失敗と区別するには次のようにすればよい．

```
> (mapcar #'edible? '(motor-oil olive-oil iguana))
(NO YES MAYBE)
```

Common Lisp では失敗を表すのに 3 番目の方法が利用できる．アクセス関数が，失敗のときに返すための特別なオブジェクト（おそらく gensym を使うことになるだろう）を引数に取るようにすることだ．この手法が使われている get はオプショナルな引数を取り，指定された属性が見つからなかったときにはそれを返す．

```
> (get 'life 'meaning (gensym))
#:G618
```

多値を返せるときには，gethash で使われている手法が一番きれいだ．get でしているように，全てのアクセス関数にオプショナル引数を渡さなければならないようなことは避けたい．そしてリストを使う手法と比べても，多値を使う方が一般性が高い．find-if は 2 個の値を返すように書き換えられるが，gethash はコンシングを起こさずに曖昧さを無くせるようなリスト構造を返すように書き換えることはできない．よって検索用の新関数を書くときや，失敗が起こり得る操作のためには，普通は gethash の手法に倣った方がよい．

edible?で使われている慣用法は正にマクロで隠蔽できる類の事務的処理だ．gethash 等のアクセス関数に対しては，同じ式の真偽を調べて結果を束縛するのでなく，第 1 引数を調べて第 2 引数を束縛するように aif を改変したものが欲しくなる．新しい aif を aif2 と名付けるが，それを第 74図に示した．これを使えば edible?は次のように書ける．

```
(defun edible? (x)
  (aif2 (gethash x edible)
          (if it 'yes 'no)
          'maybe))
```

第 74図には awhen，awhile や acond を同様に改変したものも示した．acond2 の用例については eyoi ページの match の定義を参照．このマクロなしでは遥かに長くなって対称性もなくなってしまう関数も，このマクロを使えば cond の形で表現できるようになる．

組み込み関数 read は get と同じ方法で失敗を表す．これが取るオプショナル引数は eof に出会ったときにエラーを起こすかどうか指定するものと，起こさないときに返す値を指定するものだ．第 75図に示した read の別の定義は，第 2 返り値を使って失敗を表す．read2 はが返す 2 個の値は，入力された式と，eof のときに nil になるフラグだ．これは eof のときに返される gensym 等を引数に read を呼ぶが，read2 が呼ばれる度に gensym を生成する負担を節約するため，コンパイル時に生成される gensym の独自コピーを持ったクロージャとして定義されている．

第 75図には，ファイル内の式について反復を行うための便利なマクロも示した．これは awhile2 と read2 を使って定義されている．do-file を使うことで，例えば一種の load を次のように定義できる．

```
(defun our-load (filename)
  (do-file filename (eval it)))
```

```
(defmacro aif2 (test &optional then else)
  (let ((win (gensym)))
    `(multiple-value-bind (it ,win) ,test
       (if (or it ,win) ,then ,else))))

(defmacro awhen2 (test &body body)
  `(aif2 ,test
         (progn ,@body)))

(defmacro awhile2 (test &body body)
  (let ((flag (gensym)))
    `(let ((,flag t))
       (while ,flag
              (aif2 ,test
                    (progn ,@body)
                    (setq ,flag nil))))))

(defmacro acond2 (&rest clauses)
  (if (null clauses)
      nil
      (let ((cl1 (car clauses))
            (val (gensym))
            (win (gensym)))
        `(multiple-value-bind (,val ,win) ,(car cl1)
           (if (or ,val ,win)
               (let ((it ,val)) ,@(cdr cl1))
               (acond2 ,@(cdr clauses)))))))
```

図 74　多値を返すアナフォリックマクロ.

```
(let ((g (gensym)))
  (defun read2 (&optional (str *standard-input*))
    (let ((val (read str nil g)))
      (unless (equal val g) (values val t)))))

(defmacro do-file (filename &body body)
  (let ((str (gensym)))
    `(with-open-file (,str ,filename)
       (awhile2 (read2 ,str)
                ,@body))))
```

図 75　ファイル用ユーティリティ.

13.3　参照の透明性

アナフォリックマクロは参照透明性を侵すと言われることがある．Gelernter と Jagannathan は参照透明性を次のように定義した.

参照の透明性が保たれているプログラミング言語とは,
a) どの部分式も，値の等しい別の式に置き換えることができ,
b) 式は任意のコンテキスト内のどこで何回使われても同じ値を返す，ようなものだ.

この基準はプログラミング言語に適用されるもので，プログラムには適用されないことに注意しよう．代入操作を持つプログラミング言語はいずれも参照透明性を持たない．次の式で，最初と最後の x は同じ値を持たない.

```
(list x (setq x (not x))
      x)
```

これは setq が介在しているからだ．明らかに，これは汚いコードだ．そのようなコードが可能であるという事実は，Lisp は参照透明性を持たないということだ．

Norvig は，if を次のように再定義したら便利だろうと述べたが，

```
(defmacro if (test then &optional else)
  `(let ((that ,test))
        (if that ,then ,else)))
```

参照透明性を侵すという観点からこのマクロを却下している．

しかし問題は組み込みオペレータを再定義したことに因るもので，アナフォラを使ったことに因るのではない．上の定義の b) 節は，式が「任意のコンテキスト内」で必ず同じ値を返すことを要求している．次の let の中では，シンボル that が新しい変数を表しても問題はない．

```
(let ((that 'which))
  ...)
```

let は当然新しいコンテキストを生成するはずのものだからだ．

上のマクロの問題は if を再定義していることで，そうすると新しいコンテキストが作られることにはならない．アナフォリックマクロに別の名前を与えれば問題は消失する．(それ以前に，CLtL2 にあるように if の再定義は違法だ．) そのようなマクロが aif の定義の一部であることにより，そのマクロが生成したコンテキスト内では it が新しい変数である限り，そのマクロは参照透明性を侵さない．

さて aif は確かに別の慣習に違反するが，それは参照透明性とは関係ない．その慣習とは，新しく生成された変数はソースコード内で何らかの形で分かるようになっているべきというものだ．上の let は that が新しい変数を参照するようになることをはっきり示している．aif 内での it の束縛がはっきり見えないという議論もあるかも知れない．しかしこれは余り説得力のある意見ではない．aif は 1 個の変数を生成するだけだし，しかもその変数を生成することだけが aif を使う理由だからだ．

Common Lisp そのものはこの慣習を不可侵なものとはしていない．CLOS 関数 call-next-method の束縛はコンテキストに依存するが，これは aif の本体内でシンボル it がそうなのとと全く同様だ．(call-next-method をどのように実装するかについてのアイディアについては，xwg ページのマクロ defmeth を参照．) どの場合にしろ，そういった慣習はある目的のための手段と考えられているに過ぎない．読み易いプログラムという目的だ．そしてアナフォラは，英語を読み易くしてくれるのと全く同様に，確かにプログラムを読み易くしてくれる．

14 関数を返すマクロ

第 5 章では関数を返す関数の書き方を示した．マクロはオペレータの組み合わせをずっと容易にしてくれる．この章では第 5 章で定義したものと同じ抽象化構造を，マクロを使ってきれいにかつ効率的に構築する方法を示す．

14.1 関数の構築

f と g が関数ならば，$f \circ g(x) = f(g(x))$ である．第 5.4 節では，演算子 $\circ$ を Lisp の関数 compose として実装する方法を示した．

```
> (funcall (compose #'list #'1+) 2)
(3)
```

この章では，マクロによってよりよい関数生成オペレータを定義する方法について考える．第 76図には，汎用の関数生成オペレータ fn を示した．これは関数の仕様に基づいて合成関数を構築するもので，(⟨オペレータ⟩ . ⟨引数⟩) という形の式を引数に取る．⟨オペレータ⟩ には関数またはマクロの名前か，特別扱いの compose が指定できる．⟨引数⟩ には 1 個の引数を取る関数またはマクロの名前か，fn の引数になり得る式が入る．例えば，

```
(fn (and integerp oddp))
```

が生成する関数は次のものに等価だ．

```
#'(lambda (x) (and (integerp x) (oddp x)))
```

⟨オペレータ⟩ に compose を指定すると，⟨引数⟩ の合成になっている関数が得られる．しかし compose が関数として定義されていたときに必要だった funcall は，陽に使う必要がなくなっている．例えば，

```
(defmacro fn (expr) `#',(rbuild expr))

(defun rbuild (expr)
  (if (or (atom expr) (eq (car expr) 'lambda))
      expr
      (if (eq (car expr) 'compose)
          (build-compose (cdr expr))
          (build-call (car expr) (cdr expr)))))

(defun build-call (op fns)
  (let ((g (gensym)))
    `(lambda (,g)
       (,op ,@(mapcar #'(lambda (f)
                          `(,(rbuild f) ,g))
                      fns)))))

(defun build-compose (fns)
  (let ((g (gensym)))
    `(lambda (,g)
       ,(labels ((rec (fns)
                   (if fns
                       `(,(rbuild (car fns))
                          ,(rec (cdr fns)))
                       g)))
          (rec fns)))))
```

図 76　汎用の関数生成マクロ.

`(fn (compose list 1+ truncate))`
は次のように展開される.
`#'(lambda (#:g1) (list (1+ (truncate #:g1))))`
こうすると `list` や `1+` 等の小さな関数のインライン・コンパイルが可能になる．マクロ `fn` は一般的な意味でのオペレータの名前を取る．すなわち，次の例のようにλ式も使える.
`(fn (compose (lambda (x) (+ x 3)) truncate))`
これは次のように展開される.
`#'(lambda (#:g2) ((lambda (x) (+ x 3)) (truncate #:g2)))`

ここでλ式として表現された関数は確かにインライン・コンパイルされることになる．というのも `compose` の引数として渡されたシャープクォート付きλ式は `funcall` されなければならないからだ.

第 16図には，`fif`，`fint` に `fun` といった 3 個の関数生成オペレータの定義を示した．これらは汎用マクロ `fn` に統合されることになる．`and` を ⟨オペレータ⟩ に指定すると，⟨引数⟩ の関数の交わりが得られる.

```
> (mapcar (fn (and integerp oddp))
          '(c 3 p 0))
(NIL T NIL NIL)
```

それに対し，`or` では合併が得られる.

```
> (mapcar (fn (or integerp symbolp))
          '(c 3 p 0-2))
(T T T NIL)
```

そして `if` では本体が条件によって変わる関数が得られる.

```
> (map1-n (fn (if oddp 1+ identity)) 6)
(2 2 4 4 6 6)
```

しかしこれら 3 個以外の Lisp の関数も利用できる.

```
> (mapcar (fn (list 1- identity 1+))
          '(1 2 3))
((0 1 2) (1 2 3) (2 3 4))
```

そして fn の ⟨引数⟩ 自体も式であってよい.

```
> (remove-if (fn (or (and integerp oddp)
                     (and consp cdr)))
             '(1 (a b) c (d) 2 3-4 (e f g)))
(C (D) 2 3-4)
```

fn に compose を特別扱いさせても，それが強力になる訳ではない．fn の ⟨引数⟩ を入れ子にすれば，関数の合成になるからだ．例えば,

```
(fn (list (1+ truncate)))
```

は次のように展開され,

```
#'(lambda (#:g1)
    (list ((lambda (#:g2) (1+ (truncate #:g2))) #:g1)))
```

次と同様の動作になる.

```
(compose #'list #'1+ #'truncate)
```

マクロ fn が compose を特別扱いするのは，その場合のコードを読み易くするためだけだ.

14.2 Cdr 部での再帰

第 5.5，5.6 節では，再帰関数を生成する関数の書き方を示した．ここからの 2 節では，そこで定義した関数に対してアナフォリックマクロならどのようにきれいなインタフェースを提供できるかを示す.

第 5.5 節では，平坦なリストに対する再帰関数を生成する lrec の定義方法を示した．lrec を使えば，次の関数 our-every を,

```
(defun our-every (fn lst)
  (if (null lst)
      t
      (and (funcall fn (car lst))
           (our-every fn (cdr lst)))))
```

例えば oddp に適用して呼び出すのは，次のように表現できる.

```
(lrec #'(lambda (x f) (and (oddp x) (funcall f)))
      t)
```

ここでマクロが人生の面倒を取り除いてくれる．再帰関数を表現するのに，指定しなければならない事柄はどれ程だろうか？リストの現在の car 部と再帰呼び出しをアナフォラを使って（それぞれ it と rec として）表現できたら，次のようにすればよくなる.

```
(alrec (and (oddp it) rec) t)
```

第 77図には，次のようなことを可能にしてくれるマクロの定義を示した.

```
> (funcall (alrec (and (oddp it) rec) t)
                  '(1 3 5))
T
```

このマクロは，第 2 引数として与えられた式を lrec に渡す関数に変換することで動作する．第 2 引数はアナフォリックに it や rec を参照しているかもしれないので，マクロの展開型内では，関数の本体はそれらのシンボルが生成する束縛のスコープ内に置かれていなければならない.

実際には第 77図には 2 種類の alrec の定義を示してある．上の例で使った方にはシンボル・マクロ（第 7.11 節）が必要になる．シンボル・マクロは最近の Common Lisp でしか使えないので，第 77図には少々利便性の劣る alrec も示した．こちらでは rec はローカル関数として定義されている．関数にしたことの代償は，rec を括弧で囲まなければならないという点だ.

```
(alrec (and (oddp it) (rec)) t)
```

symbol-macrolet を提供する Common Lisp 処理系では，元の方が好ましい.

Common Lisp では関数の名前空間が独立しているので，これらの再帰関数生成マクロで名前のある関数を定義するとぎこちなくなってしまう.

```
(setf (symbol-function 'our-length)
      (alrec (1+ rec) 0))
```

```
(defmacro alrec (rec &optional base)
  "cltl2 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (symbol-macrolet ((rec (funcall ,gfn)))
                                ,rec))
           ,base)))

(defmacro alrec (rec &optional base)
  "cltl1 version"
  (let ((gfn (gensym)))
    `(lrec #'(lambda (it ,gfn)
               (labels ((rec () (funcall ,gfn)))
                 ,rec))
           ,base)))

(defmacro on-cdrs (rec base &rest lsts)
  `(funcall (alrec ,rec #'(lambda () ,base)) ,@lsts))
```

図 77　リストでの再帰のためのマクロ.

```
(defun our-copy-list (lst)
  (on-cdrs (cons it rec) nil lst))

(defun our-remove-duplicates (lst)
  (on-cdrs (adjoin it rec) nil lst))

(defun our-find-if (fn lst)
  (on-cdrs (if (funcall fn it) it rec) nil lst))

(defun our-some (fn lst)
  (on-cdrs (or (funcall fn it) rec) nil lst))
```

図 78　Common Lisp の関数を on-cdr を使って定義した例.

第 78図の最後のマクロは，これを抽象化するためのものだ．on-cdrs を使えば，上の代わりに次のようにすればよい.

```
(defun our-length (lst)
  (on-cdrs (1+ rec) 0 lst))

(defun our-every (fn lst)
  (on-cdrs (and (funcall fn it) rec) t lst))
```

第 78図には，この真マクロを使って定義した既存の Common Lisp の関数を幾つか示した．on-cdrs を使って表現すると，これらの関数は最も基本的な形に還元される．また，そうしなければ明らかにはならなかったこれらの関数の間の類似性にも気付かされる.

第 79図には，on-cdrs を使って容易に定義される新ユーティリティを幾つか示した．始めの 3 個，unions, intersections に differences は，それぞれ合併集合，共通集合に補集合を実装している．Common Lisp にはそれらの操作のための組み込み関数が備わっているが，それらは 1 回に 2 つのリストしか引数に取れない．そのため，3 つのリストの合併を得るためには次のようにする必要がある.

```
> (union '(a b) (union '(b c) '(c d)))
(A B C D)
```

新しく作った unions は union と同様に動作するが，任意個数の引数を取れる．だから次のようにできる.

```
> (unions '(a b) '(b c) '(c d))
(D C A B)
```

union と同様に，unions は要素の元のリスト内での順番を保たない.

```
(defun unions (&rest sets)
  (on-cdrs (union it rec) (car sets) (cdr sets)))

(defun intersections (&rest sets)
  (unless (some #'null sets)
    (on-cdrs (intersection it rec) (car sets) (cdr sets))))

(defun differences (set &rest outs)
  (on-cdrs (set-difference rec it) set outs))

(defun maxmin (args)
  (when args
    (on-cdrs (multiple-value-bind (mx mn) rec
               (values (max mx it) (min mn it)))
             (values (car args) (car args))
             (cdr args))))
```

図 79　on-cdrs を使って定義された新ユーティリティ.

Common Lisp 組み込みの intersection と，より一般的な intersections との関係もこれと同じだ．intersections の定義では，効率性のために最初に空引数を調べる動作が追加された．集合の 1 つが空のときには，これは比較を省略する．

Common Lisp には set-difference という関数も備わっている．これは 2 つのリストを引数に取り，1 番目の要素だが，2 番目の要素ではないものを返す．

```
> (set-difference '(a b c d) '(a c))
(D B)
```

上の新ユーティリティでは，-（マイナス）と同様に複数個の引数を扱える．例えば (differences x y z) は (set-difference x (unions y z)) と等価だ．しかし後者に伴うコンシングが不必要になっている．

```
> (differences '(a b c d e) '(a f) '(d))
(B C E)
```

これらの集合用オペレータは単なる例であって，実際に必要になることはない．組み込み関数 reduce が既に扱っていた，リストに対する再帰の特別な場合を代表しているに過ぎないからだ．例えば次のようにする代わりに，

```
(unions ...)
```

ただこうすればよい．

```
((lambda (&rest args) (reduce #'union args)) ...)
```

しかし，一般的な状況では on-cdrs の方が reduce よりも強力だ．

rec は値でなく呼び出しを参照するので，on-cdrs を使って多値を返す関数を生成できる．第 79図の最後の関数 maxmin はこの可能性を活用し，リストを 1 回探索するうちに最大の要素と最小の要素の両方を発見する．

```
> (maxmin '(3 4 2 8 5 1 6 7))
8 1
```

on-cdrs はこの後の章で出てくるコードに使うこともできるだろう．例えば compile-cmds（ldq ページ）は，

```
(defun compile-cmds (cmds)
      (if (null cmds)
          'regs
          `(,@(car cmds) ,(compile-cmds (cdr cmds)))))
```

次のように簡潔に定義できる．

```
(defun compile-cmds (cmds)
      (on-cdrs `(,@it ,rec) 'regs cmds))
```

14.3 部分ツリーでの再帰

リストに対する再帰へのマクロの応用は，ツリーに対する再帰にも適用できる．この節では、マクロを使って第 5.6 節で定義したツリーに対する再帰関数へのよりきれいなインタフェースを定義する．

第 5.6 節では，ツリーに対する再帰関数を生成する関数を 2 つ定義した．ttrav は常にツリーの全体を探索するもので，また trec の方は複雑だが，再帰の停止を制御できるようになっている．これらの関数を使えば，次の our-copy-tree は，

```
(defun our-copy-tree (tree)
  (if (atom tree)
      tree
      (cons (our-copy-tree (car tree))
            (if (cdr tree) (our-copy-tree (cdr tree))))))
```

こうして表現できる．

```
(ttrav #'cons)
```

また次の rfind-if の，

```
(defun rfind-if (fn tree)
  (if (atom tree)
      (and (funcall fn tree) tree)
      (or (rfind-if fn (car tree))
          (and (cdr tree) (rfind-if fn (cdr tree))))))
```

例えば oddp への適用は，こうして表現できる．

```
(trec #'(lambda (o l r) (or (funcall l) (funcall r)))
      #'(lambda (tree) (and (oddp tree) tree)))
```

前節で lrec に対してインタフェースを作ったのと同様，アナフォリックマクロは trec へのより良いインタフェースを作れる．一般的な場合にも十分対処できるマクロは，以下の 3 つをアナフォリックに参照できるようでなければならない：現在，対象となっているツリー（it と呼ぶことにする），左の部分ツリーに対する再帰（left と呼ぶ），そして右の部分ツリーに対する再帰（right と呼ぶ）だ．これらを慣習として定めれば，上に示した関数は新マクロを使って次のように表現できる．

```
(atrec (cons left right))

(atrec (or left right) (and (oddp it) it))
```

第 80図には，このマクロの定義を示した．

symbol-macrolet をサポートしない Lisp 処理系では，atrec は第 80図の 2 番目の形で定義できる．こちらでは left と right をローカル関数として定義しているので，our-copy-tree は次のように表現される必要がある．

```
(atrec (cons (left) (right)))
```

利便性のために，マクロ on-trees も定義した．これは前節の on-cdrs に似ている．第 81図には，第 5.6 節の関数の on-trees を使った定義を示した．

第 5 章で注意したことだが，そこで定義した再帰関数生成関数の構築した関数は末尾再帰的にはならない．on-cdrs や on-trees を使って関数を定義すると，必ずしも最高の効率を持つ実装はできない．基盤となっている trec や lrec と同様に，これらのマクロは主にプロトタイプや，プログラム中で効率性が至上とされない部分で使うものだ．しかしこの章と第 5 章の基盤となる考えは，関数生成関数を書いて，それにマクロのきれいなインタフェースを被せることができるということだ．これと同じ技法は，特に効率的なコードを生成する関数生成関数を構築するときにも同様に使える．

14.4 遅延評価

遅延評価とは，式を，値が必要になったときにのみ評価することだ．遅延評価の使い道の 1 つは，遅延と呼ばれるオブジェクトを作ることだ．遅延とは何らかの式の値を保持するもので，式の値を，いつか後で必要になったときには確かに与えるという約束を表現する．ところで約束は Lisp のオブジェクトなので，それが表現する値の目的の多くに

```
(defmacro atrec (rec &optional (base 'it))
  "cltl2 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (symbol-macrolet ((left (funcall ,lfn))
                                 (right (funcall ,rfn)))
                                ,rec))
           #'(lambda (it) ,base))))

(defmacro atrec (rec &optional (base 'it))
  "cltl1 version"
  (let ((lfn (gensym)) (rfn (gensym)))
    `(trec #'(lambda (it ,lfn ,rfn)
               (labels ((left () (funcall ,lfn))
                        (right () (funcall ,rfn)))
                 ,rec))
           #'(lambda (it) ,base))))

(defmacro on-trees (rec base &rest trees)
  `(funcall (atrec ,rec ,base) ,@trees))
```

図 80　ツリーに対する再帰のためのマクロ.

```
(defun our-copy-tree (tree)
  (on-trees (cons left right) it tree))

(defun count-leaves (tree)
  (on-trees (+ left (or right 1)) 1 tree))

(defun flatten (tree)
  (on-trees (nconc left right) (mklist it) tree))

(defun rfind-if (fn tree)
  (on-trees (or left right)
            (and (funcall fn it) it)
            tree))
```

図 81　on-trees を使って定義した関数.

```
(defconstant unforced (gensym))

(defstruct delay forced closure)

(defmacro delay (expr)
  (let ((self (gensym)))
    `(let ((,self (make-delay :forced unforced)))
       (setf (delay-closure ,self)
             #'(lambda ()
                 (setf (delay-forced ,self) ,expr)))
       ,self)))

(defun force (x)
  (if (delay-p x)
      (if (eq (delay-forced x) unforced)
          (funcall (delay-closure x))
          (delay-forced x))
      x))
```

図 82　force と delay の実装.

対して役立つ．そして式の値が必要になったときには，遅延が値を返すことができる．Scheme には遅延の組み込みサポートがある．Scheme のオペレータ `force` と `delay` は，Common Lisp では 82図のように実装できる．遅延は 2 個のメンバを持つ構造体 `delay` として表現される．1 番目のメンバはこの遅延がすでに評価されたかどうかを表し，評価されていたときにはその値を含む．2 番目のメンバはクロージャを含むが，これを呼ぶとこの遅延が表現する値が得られる．マクロ `delay` は式を引数に取り，その値を表現する遅延を返す．

```
> (let ((x 2))
    (setq d (delay (1+ x))))
#S(DELAY ...)
```

遅延の中のクロージャを呼ぶことは，遅延に対して強制 (force) を行うことと同じだ．関数 `force` は任意のオブジェクトを引数に取る．普通のオブジェクトに対しては，そのものを返す関数 `identity` として動作するが，遅延に対しては，その遅延が表現する値の要求となる．

```
> (force 'a)
A
> (force d)
3
```

`force` は，遅延かも知れないオブジェクトを扱っているときにはいつでも使える．例えば，遅延を含むかもしれないリストを整列させるときには，次のようにできる．

```
(sort lst #'(lambda (x y) (> (force x) (force y))))
```

遅延をこのような裸の形で使うのはやや不便だ．実際の応用現場では，更に抽象化の層を重ねて隠蔽することになるかも知れない．

15　マクロを定義するマクロ

コード内に生じたパターンは，しばしば新しい抽象化が必要であることを知らせてくれる．このルールはマクロ自身のコードについても全く同じく当てはまる．幾つかのマクロが似た形で定義されているときは，マクロを定義するマクロを書いて，それらを生成させることができるかも知れない．省略名を定義するマクロ，アクセス用マクロを定義するマクロ，そして第 14.1 節で説明した類のアナフォリックマクロを定義するマクロだ．

15.1　省略

マクロの用途の一番単純なものは，省略形として使うものだ．Common Lisp のオペレータにはかなり長い名前のものがある．その中で高い順位を誇るものが（最長とはとても言えないが）`destructuring-bind` で，18 文字になる．Steele の原則（sjl ページ）から導かれる「系」は，よく使われるオペレータの名前は短くあるべきということだ．(「加算操作をコストの低いものと思う理由の一つは，それを 1 文字 “+” で表記できることだ．」) 組み込みマクロ `destructuring-bind` は新たな抽象化の層をもたらすが，簡潔さの面では実際の利益はその長い名前に覆い隠されてしまう．

```
(let ((a (car x)) (b (cdr x))) ...)

(destructuring-bind (a . b) x ...)
```

プログラムは，印刷された文章と同様，1 行がせいぜい 70 文字のときが一番読み易い．1 個の識別子がその 1/4 を占めるようなときには，始めから悪条件の下に置かれたことになる．

幸運なことに，Lisp のようなプログラミング言語では，言語設計者の決定に全てを任せて生きる必要はない．次のように定義しておけば，

```
(defmacro dbind (&rest args)
  `(destructuring-bind ,@args))
```

もう 2 度と長い名前を使う必要はない．更に長く，しかもよく使われる `multiple-value-bind` についても同様だ．

```
(defmacro mvbind (&rest args)
  `(multiple-value-bind ,@args))
```

```
(defmacro abbrev (short long)
  `(defmacro ,short (&rest args)
     `(,',long ,@args)))

(defmacro abbrevs (&rest names)
  `(progn
     ,@(mapcar #'(lambda (pair)
                   `(abbrev ,@pair))
               (group names 2))))
```

図 83　省略形の自動定義.

dbind と mvbind の定義は，どれ程似通っているか注意して欲しい．実際，任意の関数[*33]，マクロや特殊式の省略形を定義するには，この形の &rest 引数とコンマ・アットで十分なのだ．作業を代わりに行ってくれるマクロが手に入るというのに，どうして更に mvbind の方式で定義を作り出さなければならないことがあろうか？

マクロを定義するマクロを定義するには，しばしば入れ子になった逆クォートが必要になる．逆クォートの入れ子は理解し辛いことで悪評が高い．よく使われる形にはいつか慣れるだろうが，逆クォートの付いた任意の式を見て，どのように展開されるかを言えるようになるとは思うべきではない．そうなるのは Lisp の欠陥ではなく，ましてや表記の欠陥でもない．込み入った積分の数式を見て値が何か知ることができないのと同じことだ．困難は問題の中にあり，表記の中にあるのではない.

しかし積分に取り組むときと同様，逆クォートの分析を小さな段階に分け，それぞれは容易に構成を追えるようにすることはできる．ここでマクロ abbrev を書きたいとしよう．これは mvbind をただ次のようにするだけで定義できるようにしてくれるものだ.

```
(abbrev mvbind multiple-value-bind)
```

第 83図には，このマクロの定義を示した．この定義は何から出てきたのだろうか？このようなマクロの定義は，展開例から導ける．展開例は次のようになる.

```
(defmacro mvbind (&rest args)
  `(multiple-value-bind ,@args))
```

逆クォートの中から multiple-value-bind を取り出すと，導出は簡単になる．最終的に作られるマクロではそれが引数になることが分かっているからだ．こうして，次の等価な定義が得られる.

```
(defmacro mvbind (&rest args)
  (let ((name 'multiple-value-bind))
    `(,name ,@args)))
```

これでこの式をテンプレートにすることができる．先頭に逆クォートを付け，変化し得る式を変数に置き換える.

```
`(defmacro ,short (&rest args)
   (let ((name ',long))
     `(,name ,@args)))
```

最後に，内側の逆クォート内の',long を name に置き換えて式を簡略化する.

```
`(defmacro ,short (&rest args)
   `(,',long ,@args))
```

これで第 83図に示したマクロの本体が得られる.

第 83図には abbrevs も示した．これは複数の省略形を同時に定義したいときに使う.

```
(abbrevs dbind  destructuring-bind
         mvbind multiple-value-bind
         mvsetq multiple-value-setq)
```

abbrevs を使うには余計な括弧を挿入する必要はない．abbrevs は group（aqh ページ）を呼び，引数を 2 個ごとにまとめるからだ．一般に，論理的には不要な括弧を打ち込むのをマクロが省いてくれるのはよいことだ．group はそういったマクロの大部分に役立つ.

[*33] 省略形は apply や funcall には渡せないが.

```
(defmacro propmacro (propname)
  `(defmacro ,propname (obj)
     `(get ,obj ',,propname)))

(defmacro propmacros (&rest props)
  `(progn
     ,@(mapcar #'(lambda (p) `(propmacro ,p))
               props)))
```

図 84　アクセス用マクロの自動定義.

15.2　属性

Lisp は，何かの属性をオブジェクトに関連づける方法を数多く提供する．問題とされるオブジェクトがシンボルとして表現されるときは，一番便利な方法は（一番効率が悪いが）シンボルの属性リストを使うものだ．オブジェクト o が属性 p を持ち，その値が v であるという事実を表現するには，o の属性リストを設定する．

```
(setf (get op) v)
```

だから ball1 が色 red を持つことを表現するには，こうすればよい．

```
(setf (get 'ball1 'color) 'red)
```

オブジェクトの属性のうち頻繁に参照するものがあるときは，それを参照するマクロを定義し，

```
(defmacro color (obj)
   `(get ,obj 'color))
```

そして get の場所に color を使う．

```
> (color 'ball1)
RED
```

マクロ呼び出しは setf に対して透明なので（第 12 章を参照）こうも書ける．

```
> (setf (color 'ball1) 'green)
GREEN
```

そのようなマクロには，プログラムがオブジェクトの色を表現するのに使う特定の方法を隠蔽できるという利点がある．属性リストは遅い．プログラムの今後のヴァージョンでは，速度を考慮し，色を構造体のフィールドまたはハッシュ表の項目として表現することになるかも知れない．color のように，データをそれに被せたマクロ経由で扱うと，かなり構築の進んだプログラムにおいてすら，最下層のコードに大きな変更を加えることが容易になる．プログラムで使うものが属性リストから構造体に移行しても，そこに被せたアクセス用マクロより上の段階に変更を加える必要はない．被せたマクロを利用するコードでは，内部で起こった再構築を意識する必要すらない．

重さという属性に対しては，色に対して書いたものと似たマクロを定義すればよい．

```
(defmacro weight (obj)
  `(get ,obj 'weight))
```

前節の省略形と同様に，マクロ color と weight の定義はほぼ同一だ．ここで propmacro（第 84図）は abbrev と同じ役割を担う．

マクロを定義するマクロを構成する過程には他のどのマクロとも違いはない．マクロ呼び出しを，次に展開されて欲しい式を見て，そして前者を後者に変形する方法を理解することだ．次のマクロは，

```
(propmacro color)
```

次のように展開されて欲しい．

```
(defmacro color (obj)
  `(get ,obj 'color))
```

この展開形そのものにも defmacro が使われているが，テンプレートの作り方はやはり次の通りだ．展開形に逆クォートを付け，color が使われている所にコンマを付けた仮引数名を置く．前節と同様，既存の逆クォートの中に color が現れないように展開形を変形することから始める．

```
(defmacro color (obj)
  (let ((p 'color))
    `(get ,obj ',p)))
```

そうしたら更に手順を進めてテンプレートを作り，

```
`(defmacro ,propname (obj)
   (let ((p ',propname))
     `(get ,obj ',p)))
```

これを簡略化して次を得る．

```
`(defmacro ,propname (obj)
   `(get ,obj ',',propname))
```

一群の属性名が全てマクロとして定義されなければならない場合のために，propmacros (84) がある．これは複数個の propmacro を個別に呼び出すコードに展開される．abbrevs と同様，実際にコードの大部分はマクロを定義するマクロを定義するマクロだ．

この節では属性リストを扱ったが，ここで扱った技法は一般的なものだ．これを使って，どのような形で保持されているデータに対してもアクセス用マクロを定義できる．

15.3 アナフォリックマクロ

第 14.1 節では幾つかのアナフォリックマクロの定義を示した．aif や aand のようなマクロを使うときは，何らかの引数の評価中にはシンボル it が他のシンボルの返した値に束縛される．よってこうする代わりに，

```
(let ((res (complicated-query)))
  (if res
      (foo res)))
```

ただこうすればよく，

```
(aif (complicated-query)
     (foo it))
```

またこうする代わりに，

```
(let ((o (owner x)))
  (and o (let ((a (address o)))
           (and a (city a)))))
```

ただこうすればよい．

```
(aand (owner x) (address it) (city it))
```

第 14.1 節では 7 個のアナフォリックマクロを定義した．aif，awhen，awhile，acond，alambda，ablock そして aand だ．この種のアナフォリックマクロのうち，便利なものはこれら 7 個だけなどということはない．実際，Common Lisp のあらゆる関数やマクロについてアナフォリック版を定義できる．これらのマクロの多くは mapcon のようになるだろう．つまり滅多に使われないが，必要なときには欠かせないようなものだ．

例えば aand と同様に，it が常に前の引数の返した値に束縛されるような a+ というものを定義できる．次の関数はマサチューセッツで食事を取るときの費用を計算する．

```
(defun mass-cost (menu-price)
  (a+ menu-price (* it .05) (* it 3)))
```

マサチューセッツ食料税が 5%，また住民はしばしば税の 3 倍をチップとして払う．この法則に従うと，Dolphin Seafood で broiled scrod を食べるときの費用はこうなる．

```
> (mass-cost 7.95)
9.54
```

しかしこれは salad と baked potato も含んでいる．

第 85に示したマクロ a+ は，展開形の生成を再帰関数 a+expand に依存している．a+expand の戦略を大雑把に言うと，マクロ呼び出しの引数から成るリストの cdr 部に対して再帰的に働き，一連の入れ子になった let 式を生成することだ．個々の let は it を異なる引数に束縛したままにするが，個別の gensym をそれぞれの引数に束縛する．展開関数はそれらの gensym をリストにまとめ，引数のリストの終端に達すると，gensym を引数にした + 式を返す．よって次の式は，

```
(defmacro a+ (&rest args)
  (a+expand args nil))

(defun a+expand (args syms)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(a+expand (cdr args)
                      (append syms (list sym)))))
      `(+ ,@syms)))

(defmacro alist (&rest args)
  (alist-expand args nil))

(defun alist-expand (args syms)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(alist-expand (cdr args)
                          (append syms (list sym)))))
      `(list ,@syms)))
```

図 85　a+ と alist の定義.

```
(defmacro defanaph (name &optional calls)
  (let ((calls (or calls (pop-symbol name))))
    `(defmacro ,name (&rest args)
       (anaphex args (list ',calls)))))

(defun anaphex (args expr)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(anaphex (cdr args)
                     (append expr (list sym)))))
      expr))

(defun pop-symbol (sym)
  (intern (subseq (symbol-name sym) 1)))
```

図 86　アナフォリックマクロの自動定義.

(a+ menu-price (* it .05) (* it 3))

次の展開形を生成する.

```
(let* ((#:g2 menu-price) (it #:g2))
  (let* ((#:g3 (* it 0-05)) (it #:g3))
        (let* ((#:g4 (* it 3)) (it #:g4))
           (+ #:g2 #:g3 #:g4))))
```

第 85図には，これと似た alist も示した.

```
> (alist 1 (+ 2 it) (+ 2 it))
(1 3 5)
```

繰り返すが，a+ と alist の定義はほぼ同一だ．そのようなマクロをもっと定義したいようなときは，それらのほとんどがコードの重複になるだろう．プログラムにそれらを生成させればよいではないか？第 86図のマクロ defanaph がそれを行ってくれる．defanaph を使うと，a+ と alist の定義は次のように簡潔になる.

```
(defanaph a+)
(defanaph alist)
```

こうして定義された a+ と alist の展開形は，第 85図のコードによる定義と同一だ．マクロを定義するマクロ defanaph は，引数が関数と同じ通常の評価法則にしたがって評価されるものならば何であってもアナフォリック版を作れる．すなわち，defanaph の対象は，引数が全て評価され，かつ左から右に評価されるものならば何でもよい．だから上に示した defanaph では aif や awhile は定義できないが，どのような関数のアナフォリック版でも定義できる．

a+ が a+expand を呼んで展開形を生成していたのと同様，defanaph は anaphex を呼んで展開形を生成する．汎用の展開関数 anaphex が a+expand と違っているのは，最終的に展開形内に現れる関数名を引数に取る点だけだ．実際，a+ は次のように定義できる．

```
(defmacro a+ (&rest args)
  (anaphex args '(+)))
```

anaphex と a+expand のいずれも独立した関数として定義する必要はない．anaphex は labels や alambda によって defanaph 内部で定義できる．展開形を生成する関数が上で分離されていたのは，コードを明確にするためだけだ．

普通，defanaph は引数の先頭 1 文字（おそらく a）を取り除いたものが展開形内で呼び出されるようにする．（これは pop-symbol によって行われる．）ユーザが代わりを指定したいときは，オプショナル引数で指定できる．defanaph は全ての関数と幾つかのマクロのアナフォリック版を定義できるが，面倒な制限が伴う．

1. 引数が全て評価されるオペレータにしか適用できない．
2. 展開形内では，it は常に 1 個前の引数に束縛される．しかし場合によっては——例えば awhen——it は第 1 引数の値に束縛されたままであって欲しい．
3. setf のように，第 1 引数に汎変数を取るマクロには適用できない．

これらの制限を幾つか取り除く方法を考えよう．第 1 の問題の一部は，第 2 の問題に帰着される．aif のようなマクロを定義する展開形を生成するには，anaphex がマクロ呼び出しの第 1 引数のみを置換するように修正する必要がある．

```
(defun anaphex2 (op args)
  `(let ((it ,(car args)))
     (,op it ,@(cdr args))))
```

上の再帰的でない anaphex は，it がマクロ展開によって一連の引数に必ず束縛されるようにする必要がないので，マクロ呼び出しの引数を必ずしも全て評価しないような展開形を生成できる．評価する必要があるのは第 1 引数だけだ（これは it をその値に束縛するため）．よって aif は次のように定義できる．

```
(defmacro aif (&rest args)
  (anaphex2 'if args))
```

xqg ページの元の定義との違いは，こちらでは aif に誤った数の引数を与えるとエラーになる点だけだ．正しいマクロ呼び出しでは，どちらも同一の展開形を生成する．

defanaph が汎変数に対して動作しないという 3 番目の問題は，展開形内で f（gvx ページ）を使うと解決できる．setf などのオペレータは，anaphex2 を次のように定義し直したものなら扱える．

```
(defun anaphex3 (op args)
  `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

この展開関数はマクロ呼び出しが 1 個以上の引数を持つものと仮定している（第 1 引数を汎変数にする）．これを使うと，asetf は次のように定義できる．

```
(defmacro asetf (&rest args)
       (anaphex3 'setf args))
```

第 87図には，3 種類の展開関数を示したが，それらは単一のマクロである新版 defanaph によって統合されている．プログラマはマクロ展開の望みの方式を，オプショナルなキーワード引数 rule で指定する．すなわち，それによってマクロ呼び出しの引数を評価する際の規則を指定する．それぞれの方式は次の通り．

:all　（デフォルト）マクロ展開は alist 用の方式を取る．マクロ呼び出しの引数は全て評価され，it は常に直前の引数の値に束縛される．

```
(defmacro defanaph (name &optional &key calls (rule :all))
  (let* ((opname (or calls (pop-symbol name)))
         (body (case rule
                 (:all   `(anaphex1 args '(,opname)))
                 (:first `(anaphex2 ',opname args))
                 (:place `(anaphex3 ',opname args)))))
    `(defmacro ,name (&rest args)
       ,body)))

(defun anaphex1 (args call)
  (if args
      (let ((sym (gensym)))
        `(let* ((,sym ,(car args))
                (it ,sym))
           ,(anaphex1 (cdr args)
                      (append call (list sym)))))
      call))

(defun anaphex2 (op args)
  `(let ((it ,(car args))) (,op it ,@(cdr args))))

(defun anaphex3 (op args)
  `(_f (lambda (it) (,op it ,@(cdr args))) ,(car args)))
```

図 87　更に一般的な defanaph.

:first　マクロ展開は aif 用の方式を取る．必ず評価されるのは第 1 引数のみで，it はその値に束縛される．
:place　マクロ展開は asetf 用の方式を取る．第 1 引数は汎変数として扱われ，it はその初期値に束縛される．

新しい defanaph を使うと，ここまでに示した例の幾つかは次のように定義できる．

```
(defanaph alist)
(defanaph aif :rule :first)
(defanaph asetf :rule :place)
```

asetf には，汎変数を使う多種多様なマクロを複数回の評価について心配せずに定義できる長所がある．例えば，incf は次のように定義できる．

```
(defmacro incf (place &optional (val 1))
  `(asetf ,place (+ it ,val)))
```

また，例えば pull（sqk ページ）は次のように定義できる．

```
(defmacro pull (obj place &rest args)
  `(asetf ,place (delete ,obj it ,@args)))
```

16　リードマクロ

Lisp の S 式の生涯で重要な瞬間は 3 つ，すなわち読み込み時，コンパイル時，実行時だ．関数を操れるのは実行時だ．マクロによりプログラムをコンパイル時に変換する機会が得られる．この章ではリードマクロを扱うが，これは読み込み時に機能するものだ．

16.1　マクロ文字

Lisp の基本的思想に従い，リーダは大幅に制御することができる．リーダの動作は完全に実行中の変更が効く属性値と変数で制御できる．リーダは複数の段階でプログラムできる．一番容易に動作を変更する方法は，新しいマクロ文字を定義することだ．

マクロ文字とは Lisp のリーダから特別な扱いを要求する文字だ．例えば小文字の a は普通は小文字の b と全く同じ扱いだが，開き括弧は少々異なる．開き括弧はリストの読み込みが開始したことの印だ．そういった文字にはそれぞれ

```
(set-macro-character #\'
  #'(lambda (stream char)
      (list 'quote (read stream t nil t))))
```

図 88　' はこのようにも定義できる.

関数が関連付けられており，それが Lisp のリーダにその文字を読み込んだ際の指示を与える．既存のマクロ文字に関連付けられた関数を変更することもできるし，独自のマクロ文字を定義することもできる.

組み込み関数 set-macro-character を使うのがリードマクロを定義する方法の 1 つだ．これは引数に 1 個の文字と 1 個の関数を取るが，その後 read がその文字を読み込んだら，read はその関数を呼んだ結果を返すようになる.

Lisp で最古参のリードマクロの 1 つは ' すなわちクォートだ．' がなくとも，常に 'a の代わりに (quote a) とすることもできる．しかしそれは面倒だし，コードも読み辛くなってしまうだろう．読み込みマクロ quote のおかげで 'a を (quote a) の代わりに使える．それは第 88図のように定義することもできる.

read は通常のコンテキスト（例えば "a'b" や |a'b| は除く）で ' を読み込むと，現在のストリームと文字についてこの関数を呼んだ結果を返す.（この関数は第 2 引数を無視する．どうせ必ずクォート文字だからだ.）よって read は 'a を読むと (quote a) を返す.

read の引数の最後の 3 個はそれぞれ以下のことを制御する．すなわち，ファイル終端に達したときにエラーを起こすべきか，起こさないなら何の値を返すか，read を呼び出している間にも read が呼び出されるかどうか．ほとんど全てのリードマクロで第 2 と第 4 引数は t であるべきで，またその帰結として第 3 引数はどうでもよい.

リードマクロと普通のマクロには，共に関数が背後にある．そしてマクロの展開形を生成する関数と同様に，マクロ文字に関連付けられた関数は，読み込み元のストリームに対して以外は副作用を持つべきでない．リードマクロに関連付けられた関数がいつ，またはどれ程頻繁に呼ばれるかについて，Common Lisp は一切の保証を明示していない.（CLtL2 の 543 ページを参照）

マクロとリードマクロは，コードをそれぞれ異なった段階で受け取る．マクロはコードがリーダによって既に Lisp のオブジェクトへとパースされた時点で受け取るが，リードマクロはコードがまだテキストである間にそれに作用する．しかしこのテキストに対し read を呼ぶことで，リードマクロも（望みとあらば）同様にパース済みの Lisp オブジェクトを得ることができる．だからリードマクロは少なくとも普通のマクロと同程度の力を持っている.

実際には，リードマクロの方が少なくとも 2 点でマクロより強力だ．リードマクロは Lisp の読み込むもの全てに影響できるが，マクロはコード内に展開されるだけだ．またリードマクロは一般に read を再帰的に呼ぶので，次のような式は,

```
''a
```

次のように展開される.

```
(quote (quote a))
```

しかし quote の省略形を普通のマクロで定義しようとすると，それは孤立した状態では機能するが,

```
> (eq 'a (q a))
T
```

入れ子になると機能しない．例えば,

```
(q (q a))
```

としても次のようにしかならない.

```
(quote (q a))
```

16.2　マクロ文字のディスパッチング

シャープクォートは，#で始まる他のリードマクロと同様，ディスパッチング・リードマクロと呼ばれる亜種の例だ．それらは 2 文字で使われるが，その 1 文字目はディスパッチング文字と呼ばれる．そのようなリードマクロの目的は，単に ASCII 文字集合を最大限に活用することだけだ．1 文字のリードマクロはせいぜい ASCII 文字集合の数だけしか定義できない.

```
(defmacro q (obj)
  `(quote ,obj))
     (set-dispatch-macro-character #\# #\?
       #'(lambda (stream char1 char2)
           `#'(lambda (&rest ,(gensym))
                ,(read stream t nil t))))
```

図 89　定数関数のためのリードマクロ.

```
(set-macro-character #\] (get-macro-character #\)))

(set-dispatch-macro-character #\# #\[
  #'(lambda (stream char1 char2)
      (let ((accum nil)
            (pair (read-delimited-list #\] stream t)))
        (do ((i (ceiling (car pair)) (1+ i)))
          ((> i (floor (cadr pair)))
           (list 'quote (nreverse accum)))
          (push i accum)))))
```

図 90　リードマクロを定義するデリミタ.

（make-dispatch-macro-character によって）独自のディスパッチング・マクロ文字が定義できるが，# が既にそれとして定義されているので，それを使ってもよい．# で始まる組合せの幾つかは独自利用のために明示的に予約されている．他の組合せは，Common Lisp で予め定義された意味を持たない限り利用できる．網羅的な一覧は CLtL2 の 531 ページにある．

ディスパッチングマクロ文字の新たな組合せは，関数 set-dispatch-macro-character で定義できる．これは set-macro-character と似ているが，文字の引数を 2 個取る点が異なる．プログラマに予約されている組合せの一つは #? だ．第 89図には，この組合せを定数関数のためのリードマクロとして定義する方法を示した．それによると #?2 は任意個数の引数を取って 2 を返す関数として読み込まれる．例:

```
> (mapcar #?2 '(a b c))
(2 2 2)
```

この例では新オペレータはかなり無意味に見えるが，関数を引数に使うことがひどく多いプログラムでは，定数関数もしばしば必要になる．事実，Lisp の方言によっては定数関数を定義するための always という組込み関数が用意されている．

マクロ文字をそのマクロ文字の定義内に使うことは全く問題ないことに注意しよう．他のどの Lisp 式とも同じように，それらは定義が読み込まれたときに消えてしまう．マクロ文字は #? の後に使っても構わない．#? の定義では read を呼んでいるので，' や #' 等のマクロ文字は普通通りに機能する．

```
> (eq (funcall #?'a) 'a)
T
> (eq (funcall #?#'oddp) (symbol-function 'oddp))
T
```

16.3　デリミタ

単純なマクロ文字の次によく使われるマクロ文字は，リストのデリミタだ．ユーザに予約されている組合せには #[もある．第 90図には，それを手の込んだ仕組みを持つ開き括弧の一種として定義する方法を示した．そこでは #[x y] という形の式を x 以上 y 以下の整数全てから成るリストとして読み込ませるようにしている．

```
> #[2 7]
(2 3 4 5 6 7)
```

このリードマクロで新しいのは read-delimited-list を呼んでいる点だけだ．これはちょうどこのような場合のために用意された組込み関数だ．第 1 引数にはリストの終わりとして扱う文字を与える．] が #[を閉じるデリミタとし

```
(defmacro defdelim (left right parms &body body)
  `(ddfn ,left ,right #'(lambda ,parms ,@body)))

(let ((rpar (get-macro-character #\) )))
  (defun ddfn (left right fn)
    (set-macro-character right rpar)
    (set-dispatch-macro-character #\# left
       #'(lambda (stream char1 char2)
           (apply fn
                  (read-delimited-list right stream t))))))
```

図 91　デリミタ・リードマクロを定義するためのマクロ.

```
#.(compose #'list #'1+)
(defdelim #\@{ #\@} (&rest args)
          `(fn (compose ,@args)))
```

図 92　関数合成のためのリードマクロ.

て認識されるためには，先にデリミタの一種に含まれていなければならないので，予め set-macro-character を呼ぶ必要がある.

デリミタ・リードマクロの定義ではほぼ毎回第 90図のコードの多くが繰り返し使われるだろう．マクロならそれを行う仕組みに抽象化されたインタフェイスを備えさせることができる．第 91図には，デリミタ・リードマクロを定義するためのユーティリティの定義例を示した．マクロ defdelim は引数に 2 個の文字，仮引数リスト及び本体になるコードを取る．引数リストと本体コードは暗黙のうちに関数を定義する．defdelim を呼び出すと，1 文字目をディスパッチング・リードマクロとして，2 文字目をその終端として定義する．そして定義された関数をそれらの間のものに適用した結果を返すようにする．ついでに言えば，第 90図の関数本体もユーティリティを欲しがっている．すなわち，既に定義した mapa-b（ckv ページ）だ．defdelim と mapa-b を使うと，第 90図で定義したリードマクロは次のように書ける.

```
(defdelim #\[ #\] (x y)
  (list 'quote (mapa-b #'identity (ceiling x) (floor y))))
```

デリミタ・リードマクロは関数の合成にも使うと便利だ．第 5.4 節では関数合成用のオペレータを定義した.

```
> (let ((f1 (compose #'list #'1+))
        (f2 #'(lambda (x) (list (1+ x)))))
    (equal (funcall f1 7) (funcall f2 7)))
T
```

組込み関数の list と 1+ 等を合成するときには，compose の呼び出しの評価を実行時まで待つ理由はない．第 5.7 節では代替方法を紹介した．リードマクロのシャープドットを compose 式の頭に付けることで，それを読み込み時に評価できる.

ここでは似ているがさらにきれいな解決法を示そう．第 92図のリードマクロは，#@{f 1 f 2 ...fn@}という形の式が f1，f2，...，fn を合成したものとして読み込まれるように定義する．よってこうなる.

```
> (funcall #@{list 1+@} 7)
(8)
```

これは fn（gfd ページ）の呼び出しを生成することで動作する．fn はコンパイル時に関数を生成できるものだ.

16.4　いつ何が起きるのか

最後に，混乱するかもしれない点についてはっきりさせておいた方がよいかもしれない．リードマクロが普通のマクロより先に呼び出されるなら，マクロがリードマクロを含む式に展開されるというのは何なのだろうか？例えば次のマクロは,

```
(defmacro quotable ()
  '(list 'able))
```

クォートを含む式に展開される．さてそれは本当だろうか？実は，このマクロの定義内にあるどちらのクォートも defmacro 式が読み込まれたときに展開され，次のようになる．

```
(defmacro quotable ()
  (quote (list (quote able))))
```

普通はマクロの展開形にリードマクロが含まれると思っていても不都合はないが，それはリードマクロの定義は読み込み時とコンパイル時で変わらないはず（変わるべきでない）からだ．

17　構造化代入

構造化代入とは，代入の一般化だ．オペレータ setq と setf は個々の変数に代入を行う．構造化代入は代入をアクセスと組み合わせる．第 1 引数に変数を単独で置くのではなく，変数のパターンを与える．すると各々に同じ構造の対応する場所に置かれた値が代入される．

17.1　リストに対する構造化代入

CLtL2 では，Common Lisp に新マクロ destructuring-bind が加わった．このマクロは第 7 章で簡単に扱ったが，ここでは詳しく検討する．lst が要素を 3 つ持つリストで，x に第 1 要素を，y に第 2，z に第 3 要素を束縛したいとしよう．CLtL1 のみに対応の Common Lisp では，こうしなければならない．

```
(let ((x (first lst))
      (y (second lst))
      (z (third lst)))
...)
```

新マクロを使うと，代わりにこう書くことができる．

```
(destructuring-bind (x y z) lst
  ...)
```

これは短いだけでなく，形も整っている．ものを読むときには視覚的手がかりは文字上のものより遥かに速く認識される．後の形では x，y と z の関係が示されているが，前の形では推論が必要になる．

このように単純な場合すら構造化代入によってきれいにできるのだから，複雑な場合にどれほど読みやすくなるか想像して欲しい．destructuring-bind の第 1 引数には任意の複雑なツリーを渡せる．次のコードが，let とリストへのアクセス関数を使って書かれたらどうなるか想像して欲しい．

```
(destructuring-bind ((first last) (month day year) . notes)
                    birthday
  ...)
```

これから新しく分かることがある．構造化代入は，プログラムを読み易くするだけでなく書き易くもする．

構造化代入は CLtL1 の Common Lisp にも確かに存在した．上述の例が見慣れたものに思えるなら，それはマクロの仮引数リストと同じ形式を持っているからだ．実際，destructuring-bind はマクロの引数リストを分離するために使われるコードだが，今ではそれが独立したオペレータになったのだ．マクロの仮引数リストに入れられるものなら何でもパターンに入れられる．（ただし些細な例外がある．キーワード&environment だ．）

束縛をいちどきに生成するのは魅力的な発想だ．以下の節ではこの考えの変化形を幾つか説明する．

17.2　他の構造

構造化代入をリストに限る理由はなく，複雑なオブジェクトはどれもその候補になる．この節では，リスト以外のオブジェクト用の destructuring-bind 類似マクロの書き方を説明する．

次の自然な一歩は，シークェンス一般を扱うことだ．第 93図には dbind というマクロを示した．これは destructuring-bind と似ているが，任意の種類のシークェンスに対して機能する．第 2 引数にはリスト，ヴェクタ，またはそれらの任意の組合せが使える．

```
> (dbind (a b c) #(1 2 3)
         (list a b c))
(123)
```

```
(defmacro dbind (pat seq &body body)
  (let ((gseq (gensym)))
    `(let ((,gseq ,seq))
       ,(dbind-ex (destruc pat gseq #'atom) body))))

(defun destruc (pat seq &optional (atom? #'atom) (n 0))
  (if (null pat)
      nil
      (let ((rest (cond ((funcall atom? pat) pat)
                        ((eq (car pat) '&rest) (cadr pat))
                        ((eq (car pat) '&body) (cadr pat))
                        (t nil))))
        (if rest
            `((,rest (subseq ,seq ,n)))
            (let ((p (car pat))
                  (rec (destruc (cdr pat) seq atom? (1+ n))))
              (if (funcall atom? p)
                  (cons `(,p (elt ,seq ,n))
                        rec)
                  (let ((var (gensym)))
                    (cons (cons `(,var (elt ,seq ,n))
                                (destruc p var atom?))
                          rec)))))))))

(defun dbind-ex (binds body)
  (if (null binds)
      `(progn ,@body)
      `(let ,(mapcar #'(lambda (b)
                         (if (consp (car b))
                             (car b)
                             b))
                     binds)
         ,(dbind-ex (mapcan #'(lambda (b)
                                (if (consp (car b))
                                    (cdr b)))
                            binds)
                    body))))
```

図 93　シークェンス一般に構造化代入を行うオペレータ.

```
> (dbind (a (b c) d) '( 1 #(2 3) 4)
        (list a b c d))
(1234)
> (dbind (a (b . c) &rest d) '(1 "fribble" 2 3 4)
        (list a b c d))
(1 #\f "ribble" (2 3 4))
```

リードマクロ `#(`はヴェクタを表現するためのもので, `#\`は文字を表現するためのものだ．`"abc"` = `#(#\a #\b #\c)` なので，`"fribble"`の第 1 要素は文字`#\f` だ．話を単純にするため，`dbind` は`&rest` 及び`&body` キーワードのみに対応する．

これまでに見た大半のマクロと比べて，`dbind` は大きい．このマクロの実装方法は学んでおく価値がある．いかに動作するかが理解できるだけでなく，そこから Lisp プログラミング一般に通じる教訓が得られるからだ．第 3.4 節で触れたように，Lisp プログラムは意図的にテストしやすいように書くことができる．ほとんどのコードでは，そのように書きたい欲求と速度を出す必要性とのバランスを取らなければならない．うまいことに，第 7.8 節で説明したように，マクロ展開コードにとって速度は重要ではない．マクロの展開形を生成するコードを書くときには，自分自身の人生を楽にするように書いてよい．`dbind` の展開形は，2 個の関数 `destruc` と `dbind-ex` によって生成される．恐らく全てを単一パスで行う 1 個の関数にこれらをまとめることもできるだろう．しかしどうしてそんなことに拘るだろうか？ 2 個の別々な関数であれば，テストは簡単だ．どうしてこの利点を，必要もない速度と交換にかけるだろうか？

```
(defmacro with-matrix (pats ar &body body)
  (let ((gar (gensym)))
    `(let ((,gar ,ar))
       (let ,(let ((row -1))
               (mapcan
                 #'(lambda (pat)
                     (incf row)
                     (setq col -1)
                     (mapcar #'(lambda (p)
                                 `(,p (aref ,gar
                                            ,row
                                            ,(incf col))))
                             pat))
                 pats))
         ,@body))))

(defmacro with-array (pat ar &body body)
  (let ((gar (gensym)))
    `(let ((,gar ,ar))
       (let ,(mapcar #'(lambda (p)
                         `(,(car p) (aref ,gar ,@(cdr p))))
                     pat)
         ,@body))))
```

図 94　配列に対する構造化代入.

1 個目の関数 destruc は実行時にパターンを探索し，各々の変数を対応するオブジェクトの位置と関連づける.

```
> (destruc '(a b c) 'seq #'atom)
((A (ELT SEQ 0)) (B (ELT SEQ 1)) (C (ELT SEQ 2)))
```

第 3 引数はオプションの述語で，それはパターン構造とパターンの内容を区別するのに使われる.

アクセスを効率的にするため，新しい変数（gensym）が部分シークェンスそれぞれに束縛される.

```
> (destruc '(a (b . c) &rest d) 'seq)
((A (ELT SEQ 0))
 ((#:G2 (ELT SEQ 1)) (B (ELT #:G2 0)) (C (SUBSEQ #:G2 1)))
 (D (SUBSEQ SEQ 2)))
```

destruc の出力は dbind-ex に送られ，マクロの膨大な展開形が生成される．それは destruc の生成したツリーを入れ子になった幾つかの let に変形する.

```
> (dbind-ex (destruc '(a (b . c) &rest d) 'seq) '(body))
(LET ((A (ELT SEQ 0))
      (#:G4 (ELT SEQ 1))
      (D (SUBSEQ SEQ 2)))
  (LET ((B (ELT #:G4 0))
        (C (SUBSEQ #:G4 1)))
    (PROGN BODY)))
```

dbind は，destructuring-bind と同様に，探しているリスト構造が全て見つかるものと仮定していることに注意しよう．対応する式のなかった変数は，multiple-value-bind のように nil には束縛されない．実行時に与えられたシークェンスに予期された要素が幾つか見つからなかったときは，構造化代入オペレータはエラーを生成する.

```
> (dbind (a b c) (list 1 2))
>>Error: 2 is not a valid index for the sequence (1 2)
```

内部構造を持つオブジェクトは他に何があるだろうか？一般の配列がある．これとヴェクタとの違いは次元が 1 より大きいことだ．構造化代入マクロを配列に定義するとして，パターンをどのようにひょうげんすればよいだろう？ 2 次元配列に対しては，やはりリストを使うのが効率的だ．第 94図にはマクロ with-matrix を示した．これは 2 次元配列に対して構造化代入を行うためのものだ.

```
> (setq ar (make-array '(3 3)))
#<Simple-Array T (3 3) C2D39E>
```

```
(defmacro with-struct ((name . fields) struct &body body)
  (let ((gs (gensym)))
    `(let ((,gs ,struct))
       (let ,(mapcar #'(lambda (f)
                         `(,f (,(symb name f) ,gs)))
                     fields)
         ,@body))))
```

図 95　構造体に対する構造化代入.

```
> (for (r 0 2)
    (for (c 0 2)
      (setf (aref ar r c) (+ (* r 10) c))))
NIL
> (with-matrix ((a b c)
                (d e f)
                (g h i)) ar
               (list a b c d e f g h i))
(0 1 2 10 11 12 20 21 22)
```

大きい配列や 3 次元以上の配列に対しては，異なったアプローチが必要だ．変数を大きい配列の要素それぞれに束縛したいことはまずない．配列の疎表現を行うパターンを作る方が実用的だろう．幾つかの要素だけに対応する変数と，要素を選ぶための座標を含む表現だ．第 94図の 2 個目のマクロはこの原則に基づいて作られている．上記の例の配列の対角成分を求めるにはこのようにする．

```
> (with-array ((a 0 0) (d 1 1) (i 2 2)) ar
    (values a d i))
0 11 22
```

このマクロによって，要素が一定の順番で現れなければならないパターンを卒業しつつある．変数を defstruct で定義された構造体のフィールド（訳注: メンバとも）に束縛するための，類似のマクロが作れる．そのようなマクロを第 95図に示した．パターンの第 1 要素は構造体名に関連付けられた一種の接頭辞として，残りはフィールド名として扱われる．アクセス呼び出しを行うため，このマクロは symb（lgw ページ）を使っている．

```
> (defstruct visitor name title firm)
VISITOR
> (setq theo (make-visitor :name "Theodebert"
                           :title 'king
                           :firm 'franks))
#S(VISITOR NAME "Theodebert" TITLE KING FIRM FRANKS)
> (with-struct (visitor- name firm title) theo
    (list name firm title))
("Theodebert" FRANKS KING)
```

17.3　参照

CLOS はインスタンスに対して構造化代入を行うマクロを提供する．tree が 3 個のスロット（訳注: クラスのインスタンス変数とも）species，age と height を持つクラスで，my-tree が tree のインスタンスだとしよう．次の式の中では，

```
(with-slots (species age height) my-tree
  ...)
```

my-tree のスロットは，あたかも普通の変数であるかのように参照できる．with-slots の本体内では，シンボル height はスロット height を参照する．そこに保持された値に束縛されたというだけでなくスロットそのものを参照するので，次のように書くと，

```
(setq height 72)
```

my-tree のスロット height には 72 という値が与えられる．このマクロは height をスロットの参照に展開されるシンボル・マクロ（第 7.11 節）として定義することで機能している．実際，symbol-macrolet が Common Lisp に追加されたのは with-slots 等のマクロを実現するためだ．

```
(defmacro with-places (pat seq &body body)
  (let ((gseq (gensym)))
    `(let ((,gseq ,seq))
       ,(wplac-ex (destruc pat gseq #'atom) body))))

(defun wplac-ex (binds body)
  (if (null binds)
      `(progn ,@body)
      `(symbol-macrolet ,(mapcar #'(lambda (b)
                                     (if (consp (car b))
                                         (car b)
                                         b))
                                 binds)
                        ,(wplac-ex (mapcan #'(lambda (b)
                                               (if (consp (car b))
                                                   (cdr b)))
                                           binds)
                                   body))))
```

図 96　シークェンスに対する構造化代入の参照渡し版.

with-slots が本当の構造化代入マクロであろうとなかろうと，実用的には destructuring-bind と同じ役割を持つ．古典的な構造化代入が値渡しであるのに対し，この種のものは参照渡しだ．それを何と呼ぶにしても，便利なのは間違いなさそうだ．同じ原則に基づいて，どんなマクロが他に定義できるだろうか？

任意の構造化代入マクロの参照渡し版が，let でなく symbol-macrolet に展開するようにすることで定義できる．第 96図には，dbind を with-slots のように振る舞うように修正したものを示した．with-places は，dbind を使うのと同じように使える．

```
> (with-places (a b c) #(1 2 3)
    (list a b c))
(1 2 3)
```

しかしこの新マクロを使うと，with-slots で行うのと同様に，シークェンス内のある位置に setf を適用することもできる．

```
> (let ((lst '(1 (2 3) 4)))
    (with-places (a (b . c) d) lst
      (setf a 'uno)
      (setf c '(tre)))
    lst)
(UNO (2 TRE) 4)
```

ここでは with-slots の場合と同様，変数は構造体の対応する場所を参照している．しかし他にも重要な違いがある．それらの疑似変数に値を設定するには setq でなく setf を使わなければならない．マクロ with-slots は本体内で使われた setq を setf に置き換えるために code-walker（rcx ページ）を呼び出さなければならない．ここで code-walker を書いても，コード量の割に改善点は小さくなってしまう．

with-places が dbind より一般的だと言うなら，どうしてそちらを常に使わないのだろうか？ dbind は変数を値に束縛するが，with-places は変数を，値を見つけるための一連の指示に関連付ける．参照の度に検索が必要になる．dbind が変数 c を (elt x 2) の値に束縛する所を，with-places は c を (elt x 2) に展開されるシンボル・マクロにする．だから c が本体内で n 回評価されるときは，elt の呼び出しを n 回伴う．構造化代入で作られた変数に本当に setf を使いたいときでなければ，dbind の方が速いだろう．

with-places の定義は dbind の定義（第 93とわずかに異なるだけだ．wplac-ex（dbind-ex の改良版）の中では let は symbol-macrolet に変わっている．似たような変更によって，任意の普通の構造化代入マクロの参照渡し版を作れる．

```
(defun match (x y &optional binds)
  (acond2
    ((or (eql x y) (eql x '_) (eql y '_)) (values binds t))
    ((binding x binds) (match it y binds))
    ((binding y binds) (match x it binds))
    ((varsym? x) (values (cons (cons x y) binds) t))
    ((varsym? y) (values (cons (cons y x) binds) t))
    ((and (consp x) (consp y) (match (car x) (car y) binds))
     (match (cdr x) (cdr y) it))
    (t (values nil nil))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))

(defun binding (x binds)
  (labels ((recbind (x binds)
                    (aif (assoc x binds)
                         (or (recbind (cdr it) binds)
                             it))))
    (let ((b (recbind x binds)))
      (values (cdr b) b))))
```

図 97　マッチング関数.

17.4　マッチング

構造化代入が代入の一般化であるのと同様，パターンマッチングは構造化代入の一般化だ．「パターンマッチング」という言葉には幾つも意味がある．ここでの文脈では，それは（ときには変数を含む）2 つの構造を比較し，それらを同一にするような変数への値の代入方法がないかどうか調べることだ．例えば?x と?y が変数ならば，次の 2 つのリストは ?x = a かつ ?y = b のときにマッチする．

```
(p ?x ?y c ?x)
(p a b c a)
```

また次のリストは ?x = ?y = c のときにマッチする．

```
(p ?x b ?y a)
(p ?y b c a)
```

外部とメッセージをやりとりすることで動作するプログラムを考えよう．メッセージに反応するには，そのメッセージがどの種類のものか判別しなければならず，またそのメッセージの内容を抽出しなければならない．マッチング・オペレータを使うと，2 つの段階を組み合わせられる．

そのようなオペレータを書けるようにするには，変数を他から区別する方法が必要になる．シンボルはみな変数だなどと言うことはできない．シンボルには引数としてパターン内に現れてほしいからだ．ここではパターン変数はクエスチョンマークで始まる変数としよう．それで不便なことがあったら，述語 var?を再定義するだけでその決まりを変更できる．

第 97図には，幾つかの Lisp 入門書に出て来るのと似たパターンマッチング関数を示した．match には 2 つのリストを与える．それらがマッチするようにできるならば，そのマッチを実現するリストが返される．

```
> (match '(p a b c a) '(p ?x ?y c ?x))
((?Y . B) (?X . A))
T
> (match '(p ?x b ?y a) '(p ?y b c a))
((?Y . C) (?X . ?Y))
T> (match '(a b c) '(a a a))
NIL
NIL
```

match は引数を要素毎に比較しつつ，束縛と呼ばれる，変数への値の関連付けを生成し，変数 binds の中に保持する．match はマッチに成功すると生成された束縛を返し，失敗すると nil を返す．マッチが成功しても必ずしも束縛

```
(defmacro if-match (pat seq then &optional else)
  `(aif2 (match ',pat ,seq)
         (let ,(mapcar #'(lambda (v)
                           `(,v (binding ',v it)))
                       (vars-in then #'atom))
           ,then)
         ,else))

(defun vars-in (expr &optional (atom? #'atom))
  (if (funcall atom? expr)
      (if (var? expr) (list expr))
      (union (vars-in (car expr) atom?)
             (vars-in (cdr expr) atom?))))

(defun var? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

図 98　遅いマッチングオペレータ

が生成される訳ではないので，match は gethash と同様に第 2 返り値でマッチに成功したかどうかを表す．

```
> (match '(p ?x) '(p ?x))
NIL
T
```

match が上のように nil と t を返したときは，マッチに成功したが束縛は作られなかったことを示す．

Prolog と同様，match は _（下線）をワイルドカードとして扱う．これは全てにマッチし，束縛には影響しない．

```
> (match '(a ?x b) '(_ 1 _))
((?X . 1))
T
```

match があれば，パターンマッチングを行う dbind を書くのは容易だ．第 98図には if-match というマクロを示した．dbind と同様，第 1，第 2 引数はパターンとシークェンスで，パターンをシークェンスと比較することで束縛が生成される．しかし本体の代わりに引数がさらに 2 つある．match がマッチングに成功したとき，そこで作られた束縛の下で評価される then 節と，マッチングに失敗したときに評価される else 節だ．次に if-match を使った簡単な関数を示す．

```
(defun abab (seq)
  (if-match (?x ?y ?x ?y) seq
            (values ?x ?y)
            nil))
```

マッチに成功すると，?x と?y に対応する値が定められ，返り値になる．

```
> (abab '(hi ho hi ho))
HI
HO
```

関数 vars-in はパターン内の全てのパターン変数を返す．これは var?を呼んで変数かどうかの判断を行っている．この時点では var?は varsym?（第 97図）と同一の関数だ．これは束縛リスト内の変数を判別するのに使われる．2 つの関数を個別に用意したのは， 2 種類の変数に異なった表現を使いたくなったときのためだ．

第 98図を見ると，if-match は短いが，余り効率がよくない．処理を実行時に行い過ぎる．第 1 のシークェンスがコンパイル時に分かっているときですら，両方のシークェンスを実行時に探索するようになっている．さらにいけないのが，マッチングの過程で変数束縛を保持するためにリストを一々コンシングしている点だ．コンパイル時に知られている情報を活用して if-match の新版を書けば，不必要な比較を行わず，全くコンシングしないようにできる．

シークェンスの片方がコンパイル時に知られており，そちらだけが変数を含むときは，別のやり方で行ける．match を呼び出すときは，どちらの引数が変数を含んでもよいことになっている．変数を if-match の第 1 引数のみに制限することで，コンパイル時にどの変数がマッチングに関わるかを判定できる．すると変数束縛のリストを作らずに，変数の値を変数そのものの中に保持することができる．

```
(defmacro if-match (pat seq then &optional else)
  `(let ,(mapcar #'(lambda (v) `(,v ',(gensym)))
                 (vars-in pat #'simple?))
     (pat-match ,pat ,seq ,then ,else)))

(defmacro pat-match (pat seq then else)
  (if (simple? pat)
      (match1 `((,pat ,seq)) then else)
      (with-gensyms (gseq gelse)
                    `(labels ((,gelse () ,else))
                       ,(gen-match (cons (list gseq seq)
                                         (destruc pat gseq #'simple?))
                                   then
                                   `(,gelse))))))

(defun simple? (x) (or (atom x) (eq (car x) 'quote)))

(defun gen-match (refs then else)
  (if (null refs)
      then
      (let ((then (gen-match (cdr refs) then else)))
        (if (simple? (caar refs))
            (match1 refs then else)
            (gen-match (car refs) then else)))))

(defun match1 (refs then else)
  (dbind ((pat expr) . rest) refs
         (cond ((gensym? pat)
                `(let ((,pat ,expr))
                   (if (and (typep ,pat 'sequence)
                            ,(length-test pat rest))
                       ,then
                       ,else)))
               ((eq pat '_) then)
               ((var? pat)
                (let ((ge (gensym)))
                  `(let ((,ge ,expr))
                     (if (or (gensym? ,pat) (equal ,pat ,ge))
                         (let ((,pat ,ge)) ,then)
                         ,else))))
               (t `(if (equal ,pat ,expr) ,then ,else)))))

(defun gensym? (s)
  (and (symbolp s) (not (symbol-package s))))

(defun length-test (pat rest)
  (let ((fin (caadar (last rest))))
    (if (or (consp fin) (eq fin 'elt))
        `(= (length ,pat) ,(length rest))
        `(> (length ,pat) ,(- (length rest) 2)))))
```

図 99　高速マッチング・オペレータ.

```
(if-match (?x 'a) seq
  (print ?x))
```

これは次のように展開される：

```
(let ((?x '#:g1))
  (labels ((#:g3 nil nil))
    (let ((#:g2 seq))
      (if (and (typep #:g2 'sequence)
               (= (length #:g2) 2))
          (let ((#:g5 (elt #:g2 0)))
            (if (or (gensym?  x) (equal ?x #:g5))
                (let ((?x #:g5))
                  (if (equal 'a (elt #:g2 1))
                      (print ?x)
                      (#:g3)))
                (#:g3)))
          (#:g3)))))
```

図 100　if-match の展開形.

if-match の新版は第 99図に示した．どのコードが実行時に評価されるか予測がつくときは，コンパイル時にコードを生成できる．ここでは match の呼び出しへ展開させる代わりに，必要な比較だけを行うコードを生成することにする．

変数?x を使って?x の束縛を保持するつもりなら，まだマッチングによって束縛が決定されていない変数をどのように表現すればよいだろうか？ここではパターン変数が gensym に束縛されているときに未束縛を表すことにする．よって if-match は，パターン内の変数を全て gensym に束縛するコードの生成から始まる．この場合 with-gensyms へと展開せずに，コンパイル時に一度に gensym を生成して，展開形内に直接挿入しても大丈夫だ．

展開形の残りは pat-match が生成する．このマクロは if-match と同じ引数を取る．唯一の違いは，こちらはパターン変数には新しい束縛を設けない点だ．状況によってはこの方が都合がよい．また第 19 章では pat-match ならではの用法が出て来る．

新マッチングオペレータでは，パターンの中身とパターンの構造との区別は関数 simple?によって定義される．パターン内にクォートされたリテラルを使えるようにしたいなら，構造化代入を行うコード（及び vars-in）は，第 1 要素がクォートであるリストの中には進まないように指示されなくてはならない．新マッチング・オペレータでは，単にリストをクォートするだけでパターンの要素として使える．

dbind と同様，pat-match は destruc を呼んで，実行時に引数を切り分けてくれる関数呼び出しのリストを得ている．このリストは，ネストしたパターン用のマッチングコードを再帰的に生成する gen-match に渡され，次に，パターンのツリーの葉それぞれに対するマッチングコードを生成する match1 に渡される．

if-match の展開形内のコードの大部分は，第 99図の match1 に因るものだ．この関数は 4 通りの場合を考慮している．パターンの要素が gensym のときは，それは部分リストの保持のために destruc に作られた不可視な変数の一つであって，実行時に行う必要があるのは，それが適切な長さを持っているか調べることだけだ．要素がワイルドカード（_）のときは，コードを生成する必要はない．要素が変数のときは，match1 が，その変数を実行時に与えられるシークェンスの対応する部分に対してマッチするコードまたはその変数に対応する部分を代入するコードを生成する．これら以外のときは，要素はリテラル値として扱われ，match1 はそれをシークェンスの対応する場所と比較するコードを生成する．

展開形の一部が生成される様子の例を見よう．次のマクロ呼び出しから始める．

```
(if-match (?x 'a) seq
  (print ?x)
  nil)
```

パターンは，シークェンスを表現する幾つかの gensym（読み易くするために g と呼ぼう）と共に destruc に渡される．

```
(destruc '(?x 'a) 'g #'simple?)
```

この結果は次のようになる．

((?x (elt g 0)) ((quote a) (elt g 1)))
このリストの先頭に (g seq) をコンスし，
((g seq) (?x (elt g 0)) ((quote a) (elt g 1)))
この全体を gen-match に送る．length のナイーヴな実装（grs ページ）と同様に，gen-match は最初にリスト末尾までえんえんと再帰的に作用し，戻る過程で返り値を構築する．要素が尽きると gen-match は then 部を返すが，それが ?x になる．再帰から戻る過程で，この返り値は match1 に then 部として渡される．こうして次のような呼び出しが得られ，
(match1 '(((quote a) (elt g 1))) '(print ?x) ' else function)
この結果は次のようになる．

```
(if (equal (quote a) (elt g 1))
    (print ?x)
    else function)
```

今度はこれが match1 の別の呼び出しの then 部になり，その値が最後の match1 の then 部になる．この if-match の展開形の全貌は第 100図に示した．

この展開形では，gensym は全く関連のない 2 通りで使われている．まず実行時にツリーの一部を保持する変数は，捕捉を避けるために gensym の名前を持っている．また変数?x の初期値には，マッチングによって値を代入されていないことを表すため gensym が使われる．

新しい if-match では，パターンの要素は，暗黙の内にクォートされずに評価される．これは Lisp の変数が，クォートされた式と同様にパターン内に使えるということだ．

```
> (let ((n 3))
    (if-match (?x n 'n '(a b)) '(1 3 n (a b))
      ?x))
1
```

新版は destruc（第 93図）を呼んでいるので，改善点がさらに 2 つ出て来る．パターンにキーワード&rest や&body が使えるようになった（match はこれらがあって問題にはならない）．また destruc はシークェンス一般用のオペレータ elt と subseq を使っているので，if-match の新版はどのようなシークェンスについても使える．abab が新版 if-match で定義されると，ヴェクタや文字列にも使えるようになる．

```
> (abab "abab")
#\a
#\b
> (abab #(1 2 1 2))
1 2
```

実際，パターンは dbind に渡せるものと同じ位複雑であってもよい．

```
> (if-match (?x (1 . ?y) . ?x) '((a b) #(1 2 3) a b)
    (values ?x ?y))
(A B)
#(2 3)
```

第 2 返り値でヴェクタの要素が表示されていることに注意．ヴェクタをこのように表示するには，*print-array*を t に設定すること．

この章では，プログラミングの新たな領域への一歩を踏み出した．構造化代入のための単純なマクロから話は始まった．if-match の最終版では，むしろ独自の言語に見えるようなものが得られた．以降の章では，同じ考えに基づいて働く種類のプログラム全体について説明する．

18　クエリ・コンパイラ

前章で定義したマクロの幾つかは大規模なものだった．if-match の展開形を得るためには，第 99図のコード全てに加え，第 93図の destruc が必要になる．これ位の大きさのマクロからは，最後の論点へと自然に話がつながる．埋め込み言語だ．小さなマクロが Lisp の拡張に当たるなら，大規模なマクロは Lisp 内部に部分言語を定義している——そこには独自の構文や制御構造が備わっていることもある．それへの第一歩を if-match の中に見た．これは変数を独自の方法で表現していた．

```
(defun make-db (&optional (size 100))
  (make-hash-table :size size))

(defvar *default-db* (make-db))

(defun clear-db (&optional (db *default-db*))
  (clrhash db))

(defmacro db-query (key &optional (db '*default-db*))
  `(gethash ,key ,db))

(defun db-push (key val &optional (db *default-db*))
  (push val (db-query key db)))

(defmacro fact (pred &rest args)
  `(progn (db-push ',pred ',args)
          ',args))
```

図 101　データベースの基本となる関数.

Lisp の中に実装された言語は埋め込み言語と呼ばれる．「ユーティリティ」と同様，この言葉に正確な定義はない．`if-match` は恐らくユーティリティに数えられようが，境界線にずいぶん近付いている．

埋め込み言語は，古典的なコンパイラやインタプリタで実装された言語とは異なる．それは既存のプログラミング言語の内部に，普通はコード変形を用いて実装される．基盤のプログラミング言語と拡張部分の間に境界は必要ない．二つは自由に交じり合わせることができるべきだ．実装者にとっては，これは労力の大きな削減になり得る．必要な部分だけを埋め込んで，残りは基盤となったプログラミング言語を利用すればよい．

Lisp においては，コード変形はマクロを示唆する．プリプロセッサを用いても，ある程度は埋め込み言語を実装できる．しかし普通プリプロセッサはテキストにのみ作用するが，マクロは Lisp の独特な性質を活用できる．リーダとコンパイラの間では， Lisp プログラムは Lisp のオブジェクトのリストとして表現される．この段階でコード変形を非常に巧く行うことができる．

埋め込み言語の例で一番有名なのは，CLOS すなわち Common Lisp Object System だ．既存の言語のオブジェクト指向版が欲しくなったら，新しいコンパイラを書かなければならないだろう．Lisp ではそうではない．コンパイラのチューニングは CLOS の性能を向上させるだろうが，原則的にはコンパイラを変更する必要は一切ない．全てを Lisp で書くことができる．

以降の章では埋め込み言語の例を示す．この章では Lisp の中にデータベースへのクエリに答えるプログラムを埋め込む方法を説明する．（そのプログラムはある意味で `if-match` と似ていることに気づくだろう．）第 1 節ではクエリを解釈するシステムの書き方を説明する．さらにそのプログラムをクエリ・コンパイラ——本質的には 1 つの大きなマクロ——として再実装する．すると効率は向上し，Lisp との統合性も高まる．

18.1　データベース

当面の目標のためには，データベースのスキーマは大した問題にならない．ここでは簡便さを取り，情報をリストの形で保存することにする．例えば Joshua Reynolds が英国人画家で，1723 年から 1792 年まで生きたという事実は，次のように表現する．

```
(painter reynolds joshua english)
(dates reynolds 1723 1792)
```

情報をリストにまとめる決まった方法はない．大きな 1 つのリストを使ってもよいだろう．

```
(painter reynolds joshua 1723 1792 english)
```

データベースのエントリをどのように構成するかはユーザ次第だ．唯一の制限は，エントリ（事実）は第 1 要素（述語）をキーに区別されるということだ．その制限に従う限りどんな形式でも一貫性があればよい．ただ，形式によってクエリへの反応に速い遅いの違いが出ることがある．

```
query    : (symbol  argument *)
         : (not  query )
         : (and  query *)
         : (or  query *)
argument : ? symbol
         :  symbol
         :  number
```

図 102　クエリの文法

どのデータベースにも最低 2 つの機能が必須だ．すなわちデータベースの内容の修正と，内容の問い合わせだ．第101図のコードはそれらの機能を基本的な形で提供する．データベースは，事実のリストから成り，それらの述語をキーとするハッシュ表で表現される．

第 101図で定義されたデータベース関数は複数のデータベースに対応しているが，デフォルトでは*default-db*に対して作用する．Common Lisp のパッケージシステムと同様，複数のデータベースが必要でないプログラムはどのデータベースを使うかについて一切触れる必要がない．この節では，全ての例で*default-db*のみを使う．

システムの初期化には clear-db を呼ぶ．これは現在使っているデータベースを空にする．与えられた述語から事実を検索するには db-query を，新しい事実をデータベースのエントリに挿入するには db-push を使う．第 12.1 節で説明したように，インヴァージョン可能な参照に展開されるマクロはそれ自身インヴァージョン可能だ．db-query はそのように定義されているので，述語で db-query を呼んだ結果に push で新しい事実をただプッシュすればよい．Common Lisp ではハッシュ表のエントリは（指定のない限り）nil に初期化されるので，どのキーにも最初には空リストが関連付けられている．最後に，マクロ fact がデータベースに新しい事実を追加する．

```
> (fact painter reynolds joshua english)
(REYNOLDS JOSHUA ENGLISH)
> (fact painter canale antonio venetian)
(CANALE ANTONIO VENETIAN)
> (db-query 'painter)
((CANALE ANTONIO VENETIAN)
     (REYNOLDS JOSHUA ENGLISH))
T
```

db-query の第 2 返り値として t が返されているのは，db-query が gethash に展開されるせいだ．エントリが見つからない場合と，値が nil のエントリを発見した場合を区別するため，gethash は第 2 返り値にフラグを返す．

18.2　Pattern-Matching クエリ

db-query を呼ぶのは，データベースの利用方法としては柔軟さに欠ける．普通，ユーザは事実の第 1 要素だけに依存する質問だけをするのではない．クエリ言語とは，複雑な質問を表現するための言語だ．典型的なクエリ言語では，ユーザは条件の適当な組合せを満たす全ての値を検索できる．例えば「1697 年生まれの全ての画家の名字」だ．

これから宣言的クエリ言語を提供するプログラムを作る．宣言的クエリ言語では，答えが満たす必要のある条件をユーザが指定し，システムに答えの生成を任せる．このようなクエリの表現は，私達が日常会話で使う形式に近い．我々のプログラムでは，(painter x ...) という形の事実と (dates x 1697 ...) という形の事実が存在するような全ての x を探すよう命じることでクエリを表現できるようにしよう．1697 年生まれの全ての画家を指定するには，次のようにする．

```
(and (painter ?x ?y ?z)
       (dates ?x 1697 ?w))
```

我々のプログラムは，述語と幾つかの引数から成る単純なクエリの他，and，or 等の論理演算子で組み合わされた任意の複雑なクエリに解答できるようにする．クエリ言語の文法は第 102図に示した．

事実は述語で区別されるので，変数が述語の位置に来ることはできない．述語について順番付けのできる利点を諦めてよいなら，常に同じ述語を使い，第 1 引数を事実上の述語として扱うことでこの制限を回避できる．

類似の多くのシステムと同様に，このプログラムは事実に対して懐疑的だ．分かっている幾つかの事実の他は偽であ

```
(defmacro with-answer (query &body body)
  (let ((binds (gensym)))
       `(dolist (,binds (interpret-query ',query))
            (let ,(mapcar #'(lambda (v)
                              `(,v (binding ',v ,binds)))
                          (vars-in query #'atom))
              ,@body))))

(defun interpret-query (expr &optional binds)
  (case (car expr)
        (and (interpret-and (reverse (cdr expr)) binds))
        (or      (interpret-or (cdr expr) binds))
        (not (interpret-not (cadr expr) binds))
        (t       (lookup (car expr) (cdr expr) binds))))

(defun interpret-and (clauses binds)
  (if (null clauses)
       (list binds)
       (mapcan #'(lambda (b)
                   (interpret-query (car clauses) b))
                (interpret-and (cdr clauses) binds))))

(defun interpret-or (clauses binds)
  (mapcan #'(lambda (c)
              (interpret-query c binds))
            clauses))

(defun interpret-not (clause binds)
  (if (interpret-query clause binds)
       nil
       (list binds)))

(defun lookup (pred args &optional binds)
  (mapcan #'(lambda (x)
              (aif2 (match x args binds) (list it)))
            (db-query pred)))
```

図 103　クエリ・インタプリタ.

るとする．演算子 not は，問われた事実がデータベース内にないときにマッチに成功する．「偽」ということを明示的に指定するには，Wayne's World の方法がある程度使える．

```
(edible motor-oil not)
```

しかし演算子 not はこれらの事実を他の事実と全く同様に扱う．

プログラミング言語にとって，インタプリタとコンパイラの違いは根元的だ．この章では，クエリについて同じ疑問を再び考える．クエリ・インタプリタは受け取ったクエリを元にデータベースから答えを引き出す．クエリ・コンパイラはクエリを受け取って，実行時に同じ結果を与えるプログラムを生成する．以降の節ではまずクエリ・インタプリタを，次にクエリ・コンパイラを説明する．

18.3　クエリ・インタプリタ

宣言的クエリ言語を実装するために，第 18.4 節で定義したパターンマッチング・ユーティリティを使う．第 103図に示した関数は，第 102図に示した形のクエリを解釈する．中核となる関数は interpret-query で，複雑なクエリの構造に対し再帰的に動作し，束縛を順々に生成する．複雑なクエリの評価は，Common Lisp の関数の引数の評価と同様，左から右の順で行われる．

再帰が事実を表すパターンまで辿り着くと，interpret-query は lookup を呼び出す．ここでパターンマッチング

```
(clear-db)
(fact painter hogarth william english)
(fact painter canale antonio venetian)
(fact painter reynolds joshua english)
(fact dates hogarth 1697 1772)
(fact dates canale 1697 1768)
(fact dates reynolds 1723 1792)
```

図 104　事実の例.

が行われる．関数 lookup は，述語と引数リストから成るパターンを引数に取り，パターンがデータベース内のいずれかの事実にマッチするような束縛全てから成るリストを返す．lookup は述語に対応するデータベースのエントリそれぞれについて，match（xck ページ）を呼んでパターンと比べる．マッチが成功する度に束縛のリストが返されるが，lookup はそれらのリスト全てから成るリストを返す.

```
> (lookup 'painter '(?x ?y english))
(((?Y . JOSHUA) (?X . REYNOLDS)))
```

これらの事実は，外側の論理演算子によって処理されたり組み合わされる．最終結果は束縛の集合から成るリストとして返される．第 104図に示した事実があるとして，この章の始めの方で示した例を実行してみた.

```
> (interpret-query '(and (painter ?x ?y ?z)
                         (dates ?x 1697 ?w)))
(((?W . 1768) (?Z . VENETIAN) (?Y . ANTONIO) (?X . CANALE))
 ((?W . 1772) (?Z . ENGLISH) (?Y . WILLIAM) (?X . HOGARTH)))
```

一般則として，クエリを組み合わせたり入れ子にする際に制限はない．場合によってはクエリの文法の些細な制限が顕在化するが，その点についてはこのコードの使われ方の例を幾つか見た後に考えた方がよい.

マクロ with-answer は，このクエリ・インタプリタを Lisp プログラム内でうまく使う方法を提供する．第 1 引数には任意の可能なクエリを取り，残りの引数は本体コードとなる．with-answer は，クエリの生成した束縛全てを集め，クエリ内の変数をそれぞれの束縛が指定する通りに束縛して本体コードを繰り返すコードに展開される．with-answer に渡したクエリ内に出て来る変数は（大抵は）本体コード内でも使える．クエリが，成功はするものの変数を含まないときは，with-answer は本体コードを 1 度だけ評価する.

第 104図で定義したデータベースの下でクエリを行った例を，第 105図に，自然言語への翻訳付きで示した．パターンマッチングが match によって行われているため，パターン内でアンダースコア（下線，_）をワイルドカードとして使える.

例を長くしないために，クエリの本体内では結果の表示以外のことを行っていない．しかし，一般には with-answer の本体には任意の Lisp の式が使える.

18.4　束縛に関する制限

クエリが変数を束縛する際に制限が幾つかある．例えば次のクエリは,

```
(not (painter ?x ?y ?z))
```

どうして?x と?y に束縛を一切与えないのだろう？ 画家の内の誰かの名前でない?x と?y には無限通りの組合せがある．よって，次の制限を加えることにする．演算子 not は，次のように，既に生成された束縛に対して処理を行うが,

```
(and (painter ?x ?y ?z) (not (dates ?x 1772 ?d)))
```

それ自身だけで束縛を作ってくれるものではない．画家を探して束縛の集合を生成しなければならないわけだが，それをするのは 1772 年に生まれていない人物を選び出せるようになる前だ．節を逆順に指定していたら,

```
(and (not (dates ?x 1772 ?d)) (painter ?x ?y ?z))                    ; wrong
```

1772 年に生まれた画家が一人でもいたら結果として nil を得るだろう．第 1 の例ですら，?d の値が with-answer 式の内部で使えると期待してはならないのだ.

また (or q1 ... qn) という形の式では，束縛が与えられることが保証されているのは全ての qi の中に現れる変数に対してのみだ．with-answer に次の形のクエリが含まれていたとき,

Hogarth という名字の全ての画家のファーストネームと国籍.

```
> (with-answer (painter hogarth ?x ?y)
(princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

1697 年に生まれた全ての画家のラストネーム.（最初に提示した例）

```
> (with-answer (and (painter ?x _ _)
(dates ?x 1697 _))
(princ (list ?x)))
(CANALE)(HOGARTH)
NIL
```

1772 年または 1792 年に亡くなった人のラストネームと誕生年.

```
> (with-answer (or (dates ?x ?y 1772)
(dates ?x ?y 1792))
(princ (list ?x ?y)))
(HOGARTH 1697)(REYNOLDS 1723)
NIL
```

ヴェネツィアの画家の誰とも同じ年に生まれていない全ての英国人画家のラストネーム.

```
> (with-answer (and (painter ?x _ english)
(dates ?x ?b _)
(not (and (painter ?x2 _ venetian)
(dates ?x2 ?b _))))
(princ ?x))
REYNOLDS
NIL
```

図 105　クエリ・インタプリタの用例.

`(or (painter ?x ?y ?z) (dates ?x ?b ?d))`
`?x` には必ず束縛が与えられる．2 つの部分クエリのどちらが成功しても，`?x` には束縛が与えられるからだ．しかし`?y` と`?b` のいずれについても，クエリが束縛を与える保証はない．（どちらか片方は必ず束縛が与えられるのだが.）クエリが束縛を与えなかったパターン変数は，繰り返しの内のその回の間は `nil` になる．

18.5　クエリ・コンパイラ

第 103図のコードは所望の動作を実現するが，効率がよくない．これはクエリの構造を実行時に解析しているが，構造はコンパイル時にもう分かっている．また，変数束縛を保持するためにリストをコンシングしているが，変数自身に値を持たせてもよいはずだ．どちらの問題点も，`with-answer` の別の実装で解決できる．

第 106図では新しい `with-answer` を定義している．新型は `avg`（xlx ページ）から始まり，`if-match`（rez ページ）へつながった傾向に沿っている．すなわち，旧型が実行時に行っていた処理の大半をコンパイル時に行っている．第 106図のコードは一見第 103図のコードと似ているが，その中のコードで実行時に呼ばれるものは一切ない．束縛を生成するのではなく，`with-answer` の展開形の一部をなすコードを生成する．実行時には，そのコードはデータベースの現状に基づいてクエリを満たす全ての束縛を生成する．

実際のところ，このプログラムは一つの巨大なマクロなのだ．第 107図には `with-answer` の展開形を示した．処理の大半は `pat-match`（wgc ページ）が行うが，それもまたマクロだ．すると実行時に必要になる新たな関数は，第 101図に示した基本的データベース関数だけだ．

`with-answer` がトップレベルから呼ばれると，クエリのコンパイルは利点をほとんど生かせない．クエリを表現するコードは生成され，評価された後，廃棄される．しかし `with-answer` が Lisp プログラムの中で使われたときは，クエリに対応するコードはマクロ展開の一部となる．よっってそれを含むプログラムがコンパイルされたとき，全てのクエリに対するコードがプロセスの中にインラインでコンパイルされる．

新手法の最大の利点は速度だが，`with-answer` を呼び出すコードとの統合性を高めることにもつながっている．そ

```
(defmacro with-answer (query &body body)
  `(with-gensyms ,(vars-in query #'simple?)
       ,(compile-query query `(progn ,@body))))

(defun compile-query (q body)
  (case (car q)
        (and (compile-and (cdr q) body))
        (or     (compile-or (cdr q) body))
        (not (compile-not (cadr q) body))
        (lisp `(if ,(cadr q) ,body))
        (t      (compile-simple q body))))

(defun compile-simple (q body)
  (let ((fact (gensym)))
       `(dolist (,fact (db-query ',(car q)))
             (pat-match ,(cdr q) ,fact ,body nil))))

(defun compile-and (clauses body)
  (if (null clauses)
       body
       (compile-query (car clauses)
                           (compile-and (cdr clauses) body))))

(defun compile-or (clauses body)
  (if (null clauses)
       nil
       (let ((gbod (gensym))
                (vars (vars-in body #'simple?)))
            `(labels ((,gbod ,vars ,body))
              ,@(mapcar #'(lambda (cl)
                                 (compile-query cl `(,gbod ,@vars)))
                        clauses)))))

(defun compile-not (q body)
  (let ((tag (gensym)))
       `(if (block ,tag
                ,(compile-query q `(return-from ,tag nil))
                t)
            ,body)))
```

図 106　クエリ・コンパイラ.

のことは 2 か所の改善として現われる．まず，クエリ内部の引数が評価されるようになるので，次のように書ける．

```
> (setq my-favorite-year 1723)
1723
> (with-answer (dates ?x my-favorite-year ?d)
        (format t "~A was born in my favorite year.~%" ?x))
REYNOLDS was born in my favorite year.
NIL
```

クエリ・インタプリタでできない訳ではないが，eval を明示的に呼ぶという代償を払わないといけない．そのときでさえ，クエリ引数内のレキシカル変数を参照することは無理だ.

クエリ内部の引数が評価されるようになり，評価結果が自分自身にならない任意のリテラル引数（自然言語文など）にはクォートをつける必要がある．（第 108図を参照）

新手法の利点その 2 は，一般の Lisp の式をクエリ内に混ぜることがずっと容易になったことだ．クエリ・コンパイラではオペレータ lisp を追加されたが，それには任意の Lisp 式を続けることができる．演算子 not と同様，lisp 自身は束縛を作れないが，式が nil を返すような束縛を選び出すことはできる．オペレータ lisp は > などの組込み述語を利用するのに便利だ.

```
(with-answer (painter ?x ?y ?z)
  (format t "~A ~A is a painter. %" ?y ?x))
```

これはクエリ・インタプリタにより次のように展開される：

```
(dolist (#:g1 (interpret-query '(painter ?x ?y ?z)))
  (let ((?x (binding '?x #:g1))
        (?y (binding '?y #:g1))
        (?z (binding '?z #:g1)))
    (format t "~A ~A is a painter.~%" ?y ?x)))
```

そしてクエリ・コンパイラにより次のように展開される：

```
(with-gensyms (?x ?y ?z)
  (dolist (#:g1 (db-query 'painter))
    (pat-match (?x ?y ?z) #:g1
      (progn
        (format t "~A ~A is a painter.~%" ?y ?x))
        nil)))
```

図 107　同じクエリの 2 通りの展開形.

Hogarth という名字の全ての画家のファーストネームと国籍.

```
> (with-answer (painter 'hogarth ?x ?y)
    (princ (list ?x ?y)))
(WILLIAM ENGLISH)
NIL
```

ヴェネツィアのどの画家とも同じ年に生まれていない英国人画家のラストネーム.

```
> (with-answer (and (painter ?x _ 'english)
                    (dates ?x ?b _)
                    (not (and (painter ?x2 _ 'venetian)
                              (dates ?x2 ?b _))))
    (princ ?x))
REYNOLDS
NIL
```

1770 年から 1800 年の間に亡くなった全ての画家のラストネームと没年.

```
> (with-answer (and (painter ?x _ _)
                    (dates ?x _ ?d)
                    (lisp (< 1770 ?d 1800)))
    (princ (list ?x ?d)))
(REYNOLDS 1792)(HOGARTH 1772)
NIL
```

図 108　クエリ・コンパイラの用例.

```
> (with-answer (and (dates ?x ?b ?d)
                    (lisp (> (- ?d ?b) 70)))
               (format t "~A lived over 70 years.~%" ?x))
CANALE lived over 70 years.
HOGARTH lived over 70 years.
NIL
```

高度に作りこまれた埋め込み言語は，基盤のプログラミング言語と一体化したインタフェイスを持つことができる．

これらの追加機能——引数の評価とオペレータ lisp——を除けば，クエリ・コンパイラの提供するクエリ言語はインタプリタの提供するものと同じだ．第 108図には，第 104図で定義されたデータベースに基づいたクエリ・コンパイラの動作結果の例を示した．

第 17.2 節では，式をコンパイルすることが eval にリストとして渡すことより優れている理由を 2 つ挙げた．コンパイルした方が速いし，式をその外側のレキシカル環境内で評価することができる．クエリのコンパイルの利点も全く

Common Lisp でシンボルの「シンボル値」と「シンボル関数」と呼ぶものを Scheme では区別しない．Scheme の変数は単一の値を持つが，それは関数でも何らかのオブジェクトでもよい．そのため，シャープ・クォートや `funcall` は Scheme では必要ない．Common Lisp のこのコードは，

```
(let ((f #'(lambda (x) (1+ x))))
  (funcall f 2))
```

Scheme では次のようになる．

```
(let ((f (lambda (x) (1+ x))))   (f 2))
```

Scheme の名前空間は 1 つだけなので，代入するためのオペレータがそれぞれ個別に（`defun` と `setq` のように）存在しなくてもよい．代わりに `defvar` に似た `define` と，`setq` の代わりになる `set!` が存在する．グローバル変数は `define` で定義してからでないと `set!` で値を設定できない．
Scheme では名前の付いた関数は普通は `define` で定義される．`defvar` だけでなく `defun` の役割もあるのだ．Common Lisp のこのコードは，

```
(defun foo (x) (1+ x))
```

Scheme では 2 通りに書ける．

```
(define foo (lambda (x) (1+ x)))
(define (foo x) (1+ x))
```

Common Lisp では，関数の引数は左から右の順に評価される．Scheme では評価順は意図的に未定義とされた．（それを忘れた人間は慌てふためいて実装者の笑いものになる．）
`t` と `nil` の代わりに，Scheme には `#t` と `#f` がある．空リスト `()` を真に評価する処理系もあれば偽に評価する処理系もある．
オペレータ `cond` では `default` 節，オペレータ `case` ではキー `else` が使えるが，これらは Common Lisp の `t` の役割を持つ．
組込みオペレータの名前が違う．`consp` が `pair?`で `null` が `null?`, `mapcar` は（ほぼ）`map` に対応する，等．普通，文脈から違いは明らかだ．

図 109　Scheme と Common Lisp の相異点.

同じことだ．今まで実行時に行われていた処理が，コンパイル時に行われるようになった．クエリがその外側の Lisp コードの断片と共にコンパイルされるせいで，レキシカル・コンテキストが活用できる．

19　継続

継続とは，動作中に凍結したプログラムだ．すなわち計算処理の状態を含んだ一つの関数的オブジェクトだ．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．プログラムの状態を保存し，後に再開できる能力は，ある種の問題解決に素晴しい威力を発揮する．例えば並列処理では，中断されたプロセスを継続で表すのが便利だ．非決定的探索問題では，継続は探索ツリーのノードを表現できる．

継続の理解は難しいかも知れない．この章ではその話題に 2 段階で取り組む．この章の前半では継続の組込みサポートのある Scheme での用例を見る．継続の振舞を説明し終わったら，後半では Common Lisp プログラムで継続を生成するマクロの使い方を示す．第 22–24 章のいずれでも，ここで定義したマクロを利用する．

19.1　Scheme の継続

Scheme と Common Lisp の主要な相異点には，継続を明示的にサポートする点がある．この節では Scheme で継続がどのように動作するかを示す．（第 109図には Scheme と Common Lisp の他の相異点を列挙した．）

継続は計算処理の未来を表現する関数だ．式が評価されるときには，必ず何かがその返り値を待っている．例えば次のコードで `(- x 1)` が評価された時点では，

```
(/ (- x 1) 2)
```

外側の `/` 式がその値を待っており，さらに別の何かが `/` 式の値を待っている...等が，`print` の待っているトップレベルまで延々と続く．

任意の時点の継続は，1 引数関数と考えられる．上の例がトップレベルに打ち込まれたとき，部分式 (- x 1) が評価された時点での継続は次のように考えられる．

```
(lambda (val) (/ val 2))
```

すなわち計算処理の残りは (- x 1) の返り値を上の関数に与えることで模倣できる．また，式 (- x 1) が次の文脈の中に現われており，f1 がトップレベルから呼ばれた場合は，

```
(define (f1 w)
  (let ((y (f2 w)))
    (if (integer? y) (list 'a y) 'b)))

(define (f2 x)
  (/ (- x 1) 2))
```

(- x 1) が評価された時点での継続は次のようになる．

```
(lambda (val)
  (let ((y (/ val 2)))
    (if (integer? y) (list 'a y) 'b)))
```

Scheme では, 継続は関数と同格のファーストクラス・オブジェクトだ．Scheme では現在の継続 (current continuation) を求めると計算処理の未来を表現する 1 引数関数が得られる．このオブジェクトは好きなように保存できるし，保存した継続を呼び出せば，それが生成された時点で始まろうとしていた計算処理を再開する．

継続はクロージャの一般化として理解できる．クロージャとは，関数とそれが生成された時点で見えていたレキシカル変数へのポインタをまとめたものだった．継続とは，関数とそれが生成された時点で溜っているスタック全体へのポインタをまとめたものだ．継続が評価されると，現在のスタックを無視し，それが保持しているスタックのコピーに基づいて値を返す．継続が T1 で生成され T2 で評価された場合，T1 において溜っているスタックに基づいた評価が行われる．

Scheme プログラムは組込みオペレータ call-with-current-continuation（略して call/cc）を通じて現在の継続にアクセスできる．プログラムが call/cc を 1 引数関数に対して呼び出すと，

```
(call-with-current-continuation
  (lambda (cc)
    ...))
```

その 1 引数関数には現在の継続を表現する別の関数が渡される．上で cc の値をどこかに保存することで，call/cc の時点での計算処理の状態が保存できる．

上の例では，最後の要素が call/cc 式の返り値であるようなリストに append を適用している．

```
> (define frozen)
FROZEN
> (append '(the call/cc returned)
          (list (call-with-current-continuation
                  (lambda (cc)
                    (set! frozen cc)
                    'a))))
(THE CALL/CC RETURNED A)
```

call/cc は a を返すが，最初に継続をグローバル変数 frozen に保存する．

frozen を呼び出すと，call/cc の時点での古い計算が再開される．frozen に渡した値は，いずれも call/cc の返り値になる．

```
> (frozen 'again)
(THE CALL/CC RETURNED AGAIN)
```

継続は，評価されても消費はされない．他の普通の関数と同様，繰りかえし呼ぶこともできる．

```
> (frozen 'thrice)
(THE CALL/CC RETURNED THRICE)
```

継続を他の処理の内部で呼ぶとき，古いスタックに立ち戻ることの意味がはっきり見えるようになる．

```
> (+ 1 (frozen 'safely))
(THE CALL/CC RETURNED SAFELY)
```

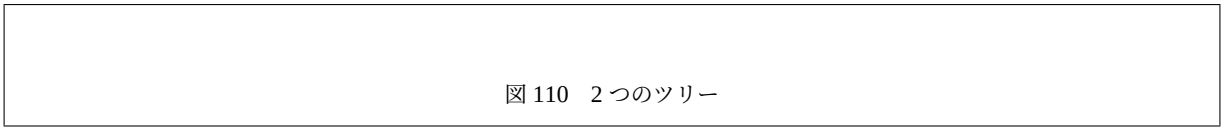

図 110　2 つのツリー

ここでは，frozen が呼ばれると結果を待っている + は無視される．frozen は，最初に生成された時点で溜っていたスタックに従い，まず list，次に append を通じ，トップレベルまで戻る．frozen が普通の関数呼び出しと同様に値を返していたら，上の式は + が 1 をリストに足し算しようとしたことでエラーになっていただろう．

継続はスタックのコピーを個別に保持するのではない．他の継続や，進行中の計算処理と変数を共有できる．次の例では，2 つの継続が同じスタックを共有している．

```
> (define froz1)
FROZ1
> (define froz2)
FROZ2
> (let ((x 0))
    (call-with-current-continuation
      (lambda (cc)
        (set! froz1 cc)
        (set! froz2 cc)))
    (set! x (1+ x))
    x)
1
```

よってどちらを呼んでも 1 ずつ増える数列が得られる．

```
> (froz2 ())
2
> (froz1 ())
3
```

call/cc 式の値は破棄されるので，froz1 や froz2 に与える引数は何でもよい．

さて，計算処理の状態を保存できるようになった訳だが，それで何をすればよいのだろう？ 第 21–24 章は継続を使うアプリケーションに充てられている．ここでは「保存された状態」を使うプログラミングの雰囲気がよく現われる単純な例を考えよう．ツリーの集合が与えられたとき，各ツリーから要素を 1 つずつ取って作ったリストを，何らかの条件を満たす組合せになるまで生成しよう．

ツリーは入れ子になったリストとして表現できる．grq ページではある種のツリーをリストとして表現する方法を説明した．ここでは別の方法を使い，内部ノードが（アトムの）値を持て，任意個の子を持てるようにした．この表現では，内部ノードはリストになる．Car 部はノードとしての値を保持し，cdr 部はそのノードの子の表現を保持する．例えば第 110図の 2 つのツリーは次のように表現できる．

```
(define t1 '(a (b (d h)) (c e (f i) g)))
(define t2 '(1 (2 (3 6 7) 4 5)))
```

第 111図にはそのようなツリーに対して深さ優先探索を行う関数を示した．実際のプログラムでは，ノードに行き着くごとにそれに対して何かの処理をしたくなる．ここでは単に表示するだけだ．比較に用いる関数 dft は，通常の深さ優先探索を行う．

```
> (dft t1)
ABDHCEFIG()
```

関数 dft-node の方は，ツリーの辿り方は同じだがノードを個別に扱う．dft-node がノードに達すると，ノードの car 部に進み，cdr 部を探索するという継続を *saved* にプッシュする．

```
> (dft-node t1)
A
```

restart を呼ぶと最後に保存された継続をポップして呼び出すことで探索が再開される．

```
> (restart)
B
```

最後に保存された状態が使い果たされると，restart は done を返すことでそれを知らせる．

```
(define (dft tree)
  (cond ((null? tree) ())
        ((not (pair? tree)) (write tree))
        (else (dft (car tree))
              (dft (cdr tree)))))

(define *saved* ())

(define (dft-node tree)
  (cond ((null? tree) (restart))
        ((not (pair? tree)) tree)
        (else (call-with-current-continuation
                (lambda (cc)
                  (set! *saved*
                    (cons (lambda ()
                            (cc (dft-node (cdr tree))))
                          *saved*))
                  (dft-node (car tree)))))))

(define (restart)
  (if (null? *saved*)
      'done
      (let ((cont (car *saved*)))
        (set! *saved* (cdr *saved*))
        (cont))))

(define (dft2 tree)
  (set! *saved* ())
  (let ((node (dft-node tree)))
    (cond ((eq? node 'done) ())
          (else (write node)
                (restart)))))
```

図 111　継続を使ったツリーの探索.

```
...> (restart)
G> (restart)
DONE
```

最後に，関数 dft2 は上で手動で行っていたことをきれいに隠蔽する.

```
> (dft2 t1)
ABDHCEFIG()
```

dft2 の定義には明示的な再帰も反復もないことに注意．ノードが次々と表示されるのは，restart の呼び出した継続により，必ず dft-node の同じ cond 節にまで戻るからだ.

この種のプログラムは鉱脈のように働く．dft-node を呼び出すことで最初の穴を掘る．返り値が done でない限り，dft-node の呼び出しに続くコードは restart を呼び，それが再びスタックに制御を送る．（訳注: このあたりは曖昧な表現が多くて意味不明）この経過はまで返り値が鉱脈は空であると知らせるまで継続する．その値を表示する代わりに，dft2 は#f を返す．継続による探索は，プログラムに関する新たな考え方を表現している．適切なコードをスタックに入れ，繰りかえしそこに戻ることで結果を手にするのだ.

dft2 のように一度に一つのツリーしか扱わないなら，この手法にこだわる理由はない．dft-node の利点は同時に複数の処理を進行させられることだ．2 つのツリーに対し，深さ優先探索で要素の直積を得たいとしよう.

```
> (set! *saved* ())
()
> (let ((node1 (dft-node t1)))
       (if (eq? node1 'done)
           'done
           (list node1 (dft-node t2))))
```

```
(A 1)
> (restart)
(A 2)
...
> (restart)
(B 1)
...
```

普通の手法を使うと，2 つのツリーにおける位置を保存する段階を明示的に踏む必要がある．継続を使うと，2 つの実行中の探索の状態は自動的に扱われる．このような単純な例では，ツリー中の位置を保存することは大して難しくない．ツリーは不変なデータ構造だから，ツリー内の「自分の場所」を把握する方法は少なくとも何かある．継続がすばらしいのは，対応する不変なデータ構造がない場合ですら，任意の計算処理の途中で容易に進み具合を保存できる点だ．計算処理の状態は，再開したいものが有限個である限り，有限個である必要すらない．

第 24 章で見るように，それらの考慮の両方が Prolog の実装で重要だと分かる．Prolog プログラムではプログラムが結果を生成する上での「探索ツリー」は実際のデータ構造ではなく，暗黙のものである．それらツリーはしばしば無限の大きさを持つが，その場合，あるツリー全体の探索を，別のツリーの探索の前に行うことは期待できない．どうにかして「場所」を保存する他に選択肢はないのだ．

19.2 継続渡しマクロ

Common Lisp は `call/cc` を提供しないが，少々の労力で Scheme と同じことができるようになる．この節では，マクロを使って Common Lisp プログラムで継続を実現する方法を示す．Scheme の継続は 2 種類のものを提供してくれた．

1. 継続が生成された時点での全ての変数の束縛.
2. 計算処理の状態——その後に何がおきるはずだったか.

レキシカル・スコープを持つ Lisp では，クロージャで第 1 の機能が得られる．実はクロージャを使って計算処理の状態も変数束縛に格納すれば第 2 の機能も得られることが分かる．

第 112図のマクロを使うと，継続を保存しつつ関数を呼び出せる．これらのマクロは Common Lisp で関数を定義したり呼び出したり，値を返すための組込みオペレータの代わりになる．

継続を利用したい関数（もしくは継続を利用する関数を呼びたい関数）は `defun` でなく`=defun` を使って定義する．`=defun` の構文は `defun` と同じだが，機能は微妙に異なる．単に関数を定義するだけでなく，`=defun` は関数を定義し，その関数の呼び出しに展開されるマクロを定義する．（そのマクロは，関数が自分自身を呼び出す場合に備えて最初に定義しなければならない.）関数の本体は`=defun` に渡されたものになるが，引数リストには `*cont*` が追加される．マクロの展開形では，定義された関数は `*cont*` を他の引数と一緒に受け取る．よって次の関数定義は,

```
(=defun add1 (x) (=values (1+ x)))
```

次のように展開される.

```
(progn (defmacro add1 (x)
         `(=add1 *cont* ,x))
       (defun =add1 (*cont* x)
         (=values (1+ x))))
```

`add1` を呼ぶとき実際に呼ばれるのは関数でなくマクロだ．マクロは関数呼び出しに展開される[*34]が, 引数 `*cont*` を 1 つ余計に受け取っている．

よって `=defun` で定義されたオペレータの呼び出しの際には，その時点での `*cont*` の値が必ず渡される．

`*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．`=values` の定義を読むとその継続の使い方が分かる．`=defun` で定義された任意の関数は `=values` を使って制御を戻すか，, それを行う別の関数を呼ばなければいけない．`=values` の構文は Common Lisp の組込みオペレータ `values` と同じだ．多値を返すには，`=bind` に同じ数の仮引数を与えておく．しかしトップレベルに多値を返すことはできない.

[*34] `=defun` にはインターンされた名前が意図的に使われるが，それはトレースできるようにするためだ．トレースが必要でないと分かっていれば，名前に gensym を使うのが安全だろう.

```
(setq *cont* #'identity)

(defmacro =lambda (parms &body body)
   `#'(lambda (*cont* ,@parms) ,@body))

(defmacro =defun (name parms &body body)
   (let ((f (intern (concatenate 'string
                                          "=" (symbol-name name)))))
       `(progn
            (defmacro ,name ,parms
               `(,',f *cont* ,,@parms))
            (defun ,f (*cont* ,@parms) ,@body))))

(defmacro =bind (parms expr &body body)
   `(let ((*cont* #'(lambda ,parms ,@body))) ,expr))

(defmacro =values (&rest retvals)
   `(funcall *cont* ,@retvals))

(defmacro =funcall (fn &rest args)
   `(funcall ,fn *cont* ,@args))

(defmacro =apply (fn &rest args)
   `(apply ,fn *cont* ,@args))
```

図 112　継続渡しマクロ.

仮引数 *cont* は，=defun で定義された関数に，返り値に対して何を行うかを伝える．=values がマクロ展開されると *cont* を捕捉するようになり，それによって関数から戻ることをシミュレートする．次の式は,

```
> (=values (1+ n))
```

次のように展開される.

```
(funcall *cont* (1+ n))
```

トップレベルでは *cont* の値は identity だ．これは渡された値をそのまま返す関数だ．トップレベルから (add1 2) を呼ぶと，それは等価な次のコードに展開される.

```
(funcall #'(lambda (*cont* n) (=values (1+ n))) *cont* 2)
```

cont の参照はこの場合グローバルな束縛を持つ．=values 式は等価な次の式に展開される.

```
(funcall #'identity (1+ n))
```

これは単に 1 から n までを足し合わせて返すものだ.

add1 などの関数では，we go through all this trouble just to simulate what Lisp function call and return do anyway:

```
> (=defun bar (x)
        (=values (list 'a (add1 x))))
BAR
> (bar 5)
(A 6)
```

大事なのは，今や私達は独自に制御できる関数呼び出しと戻り return を手にしており，他のこともその気になればできるということだ.

継続の効果は，*cont* を通じて利用する．*cont* はグローバルな値を持ってはいるが，グローバルな値そのものが使われることはまずない．*cont* はほとんど常に=values や =defun で定義されたマクロに捕捉される仮引数だ．例えば add1 の内部では *cont* は仮引数でグローバル変数ではない．この区別が重要なのは，これらのマクロは *cont* がローカル変数だからこそ機能するからだ．だからこそ *cont* に初期値を与えるために defvar でなく setq を使っている．defvar ではスペシャル変数を定義することになってしまう.

第 112図の 3 番目のマクロ =bind は，multiple-value-bind と同様の用途を想定している．引数には仮引数のリスト，式，実行部本体となるコードを取る．仮引数は渡された式の返り値に束縛され，その下で本体コードが評価され

る．=defun で定義された関数の呼び出し後に付加的な式が評価されなければいけないときにはこのマクロを使わなければいけない．

```
> (=defun message ()
          (=values 'hello 'there))
MESSAGE
> (=defun baz ()
          (=bind (m n) (message)
                 (=values (list m n))))
BAZ
> (baz)
(HELLO THERE)
```

=bind の展開形は新変数 *cont* を作っていることに注意．baz の本体は次のようにマクロ展開される．

```
(let ((*cont* #'(lambda (m n)
                  (=values (list m n)))))
  (message))
```

そしてさらに次のようになる．

```
(let ((*cont* #'(lambda (m n)
                  (funcall *cont* (list m n)))))
  (=message *cont*))
```

cont の新しい値は =bind 式の本体になるので，*cont* に funcall を適用することで message から「戻る」ときには，結果は「コード本体を評価すること」になる．しかし（ここが大事なのだが） =bind の本体内では次のようになっており，

```
#'(lambda (m n)
    (funcall *cont* (list m n)))
```

=baz に引数として渡された *cont* が見えるままなので，コード本体が =values を評価する番になると，元の呼出側関数に戻ることができる．クロージャは互いに縫い合わされている: *cont* の個々の束縛は *cont* の以前の束縛を含むクロージャで，グローバルな値に戻ってゆくまでの連鎖を形成する．

同じ現象を，もっと小さなスケールで観察できる．

```
> (let ((f #'identity))
    (let ((g #'(lambda (x) (funcall f (list 'a x)))))
      #'(lambda (x) (funcall g (list 'b x)))))
#<Interpreted-Function BF6326>
> (funcall * 2)
(A (B 2))
```

この例では g への参照を含むクロージャを作っているが，g そのものも f への参照を含むクロージャである．これと似たクロージャの連鎖が，xez ページのネットワーク・コンパイラによって生成される．

残りのマクロ =apply と =funcall は =lambda で定義された関数と共に使う．=defun で定義された「関数」は実際はマクロなので，apply や funcall の引数に使えないことに注意．この問題の解決策は ldt ページで触れたトリックに似ている．つまり呼び出しを別の=lambda の内部にくるんでしまう．

```
> (=defun add1 (x)
        (=values (1+ x)))
ADD1
> (let ((fn (=lambda (n) (add1 n))))
        (=bind (y) (=funcall fn 9)
           (format nil "9 + 1 = ~A" y)))
"9+1=10"
```

1. =defun で定義された関数の引数リストは仮引数名以外を含んではならない．
2. 継続を利用する関数，もしくはそのような関数を呼び出す関数は，=lambda または =defun で定義しなければならない．
3. そのような関数が終了するときは， =values で値を返すか同様な関数を呼び出すかのいずれかでなければならない．

```
(=defun foo (x)
        (=bind (y) (bar x)
               (format t "Ho ")
               (=bind (z) (baz x)
                      (format t "Hum.")
                      (=values x y z))))
```

図 113　継続渡しマクロに付随する制限.

```
(defun dft (tree)
  (cond ((null tree) nil)
        ((atom tree) (princ tree))
        (t (dft (car tree))
           (dft (cdr tree)))))

(setq *saved* nil)

(=defun dft-node (tree)
        (cond ((null tree) (restart))
              ((atom tree) (=values tree))
              (t (push #'(lambda () (dft-node (cdr tree)))
                       *saved*)
                 (dft-node (car tree)))))

(=defun restart ()
        (if *saved*
            (funcall (pop *saved*))
            (=values 'done)))

(=defun dft2 (tree)
        (setq *saved* nil)
        (=bind (node) (dft-node tree)
               (cond ((eq node 'done) (=values nil))
                     (t (princ node)
                        (restart)))))
```

図 114　継続渡しマクロを使ったツリーの探索.

4. =bind，=values，=apply または=funcall がコードの一部に使われている場合，それは末尾呼び出しでなければならない．=bind の後に評価されるべきコードは，必ずその本体内に入れなければならない．よって複数の =binds を順に使いたい場合，それらは入れ子にならなければならない.

第 113図には継続渡しマクロによって生じる制限をまとめた．継続を保存しないし，継続を保存する関数を呼び出しもしない関数は，これらのマクロを使う必要はない．例えば list などの組込み関数は除かれる.

第 114図は，第 112図の Scheme コードを Common Lisp に直したものを含み，継続渡しマクロを Scheme の継続の代わりに使っている．同じ例のツリーに対して dft2 は全く同様に動作する.

```
> (setq t1 '(a (b (d h)) (c e (f i) g))
          t2 '(1 (2 (3 6 7) 4 5)))
(1 (2 (3 6 7) 4 5))
> (dft2 t1)
ABDHCEFIG
NIL
```

複数の探索の状態を保存するのも Scheme と同様に機能するが，例は少し長くなる.

```
> (=bind (node1) (dft-node t1)
        (if (eq node1 'done)
            'done
            (=bind (node2) (dft-node t2)
```

```
               (list node1 node2))))
(A 1)
> (restart)
(A 2)
...> (restart)
(B 1)
...
```

レキシカル・クロージャの連鎖をつなぎ合わせることで，Common Lisp プログラムでも独自の継続を作ることができる．うまいことに，第 112図の込み入ったマクロによってクロージャはつなぎ合わされており，ユーザは実現過程を想像する必要はなく，きれいな側面だけを見ていればよい．

第 21–24 章はいずれもある意味で継続に依存している．これらの章では，継続がすさまじい力の抽象化であることを示す．継続は，特にあるプログラミング言語上のマクロとして実装された場合には，最高に速いとは言えない．しかしそれらを利用して行える抽象化により，ある種のプログラムははるかに速く書けるようになり，そのような速度のプログラムも重宝する場合というものはある．

19.3 Code-Walkers と CPS 変換

前節で説明したマクロは妥協の産物だ．継続の力が得られるが，それはプログラムを特定の形で書いたときに限られる．第 113図のルール 4 によれば，必ず次のようにしなければならず，

```
(=bind (x) (fn y)
       (list 'a x))
```

次のようにはできない．

```
(list 'a                                                    ; wrong
      (=bind (x) (fn y) x))
```

本物の call/cc はプログラマにそのような制限を課すことはない．call/cc は任意の時点で任意の形のプログラムの継続を取り出せる．call/cc の力の全てを実現するオペレータを実装することもできるが，それにはさらに手間がかかるだろう．この節ではその方法の概要を示す．

Lisp プログラムは「継続渡しスタイル」という形に変換できる．完全に CPS 変換を施したプログラムは読めたものではないが，部分的に変換されたコードを読むことでその過程のエッセンスを掴むことはできる．次の関数はリストを逆転させるものだが，

```
(defun rev (x)
  (if (null x)
      nil
      (append (rev (cdr x)) (list (car x)))))
```

等価な継続渡し版コードは次のようになる．

```
(defun rev2 (x)
  (revc x #'identity))

(defun revc (x k)
  (if (null x)
      (funcall k nil)
      (revc (cdr x)
            #'(lambda (w)
                (funcall k (append w (list (car x))))))))
```

継続渡しスタイルでは，関数には引数（ここでは k）が余分に増える．その値は継続になる．継続は，その関数の現在の値を使って何をすべきかを表現するクロージャだ．再帰の最初の段階では継続は恒等関数だ．何をすべきかと言えば，関数は現在の値をそのまま返せばよい．再帰の第 2 段階では，継続は次と等価だ．

```
#'(lambda (w)
        (identity (append w (list (car x)))))
```

つまり，何をするべきかと言えば，リストの car 部を現在の値に付加して返すということだ．

CPS 変換を覚えれば call/cc を書くのは簡単だ．CPS 変換を経たプログラムでは，全体に対する現在の継続が常に存在し，call/cc はその引数として何らかの関数を呼び出す単純なマクロとして実装できる．

CPS 変換には，code-walker, すなわちプログラムのソースコードを表現するツリーを探索するプログラムが必要になる．Common Lisp に対する code-walker を書くのはひどい骨折りだ．使えるものになるには，code-walker は単に式を探索するだけでは済まず,．その式の意味もかなり理解できなければならない．例えば，code-walker はシンボルだけを考えていれば済むものではない．シンボルが表現するのは，いくつか挙げると，シンボル自身，関数，変数，ブロック名，go のためのタグなどだ．Code-walker は文脈からシンボルの種類を判別し，それに従って動作しなければならない．

Code-walker を書くのはこの本の範囲を超えるので，この章で説明したマクロは実用的な代理品に過ぎない．この章のマクロは継続を構築する仕事をユーザと分担して行う．ユーザが CPS に十分近いプログラムを書けば，残りはマクロがやってくれる．第 4 の規則は本当はそういうことだ．`=bind` の後に続く処理がその本体内にあれば，`*cont*` の値と `=bind` の本体内のコードの間で，プログラムは現在の継続を作るための十分な情報を得ることができる．

マクロ `=bind` はこのスタイルのプログラミングを自然に思わせる意図を持って書かれた．実際には継続渡しマクロの課す制限は我慢の範囲内だろう．

20　複数プロセス

前章では，継続を利用すれば実行中のプログラムが自分の状態を把握し，それを保存して後で再開することができることを示した．この章で扱う計算処理のモデルは，計算機が単一のプログラムを実行するのでなく，独立したプロセスの集合を実行するというものだ．プロセスの概念はプログラムの状態という概念と深い対応関係にある．前章で示したマクロの上にさらにもう一層のマクロを書くことで，Common Lisp プログラムの中に複数プロセスを埋め込むことができる．

20.1　プロセスの抽象化

複数プロセスは，複数の処理を同時に行わなければならないプログラムを表現するのに便利だ．伝統的な CPU はインストラクションを 1 つずつ実行する．「複数プロセスは同時に複数の処理を行う」というのは，このハードウェア上の制限をどうにか乗り越えるという意味ではない．それを使って，抽象化の新たな段階でものを考えられるようになるということだ．つまりコンピュータがある時点で何を行うか，逐一指定しなくともよい．仮想メモリによって，コンピュータが実際に備えているより多くのメモリがあるかのように思えるのと同様，プロセスの概念によって，コンピュータが同時に複数のプログラムを実行できるかのように思っていられる．

プロセスの研究は伝統的には OS の分野で行われていた．しかし抽象化の観点としてのプロセスの利便性は OS の世界に限られてはいない．プロセスは他のリアルタイムアプリケーションやシミュレーションでも同様に便利なものだ．

複数プロセスに基づいてなされた仕事の大部分は，ある種の問題の回避に費されてきた．デッドロックは複数プロセスに伴う古典的問題の一つだ．すなわち，相手より先に敷居を越えるのを互いに拒む 2 人の人のように，2 つのプロセスが何もせずに，共にもう片方が何かを行うのを待ち続けることだ．他にも，一貫していない状態にあるシステムに対するクエリの問題もある．例えば，システムが預金をある口座から別の口座に移している最中に残高が問い合わせられる場合だ．この章で扱うのはプロセスによる抽象化のみだ．ここで示されたコードは，デッドロックや一貫していない状態を防ぐアルゴリズムのテストに使えるかもしれないが，コードそのものはそれらの問題に対する防御措置を一切取っていない．

この章における実装は，この本の全てのプログラムが暗黙に従っている規則に従っている．すなわち「なるだけ Lisp とぶつかるな」だ．気持ちとしては，プログラムはできる限りプログラミング言語の修正に近付くべきで，そのプログラミング言語で書かれた個別のアプリケーションになるべきではない．プログラムを Lisp と調和するように作れば，ぴったり合った部品で作られた機械のように頑健になる．また，労力の節約にもなる．驚く程の量の処理を Lisp に肩代りさせられることもある．

この章の目的は，複数プロセスに対応したプログラミング言語を作ることだ．ここでの戦略は，新オペレータをいくつか追加して，Lisp をそのようなものに変えてゆくことだ．新プログラミング言語の基本要素は次のようなものだ．

関数は，前章のマクロ `=defun` や `=lambda` で定義される．

プロセスは関数呼び出しによって生成される．（一つの関数から生成されていても）アクティブなプロセスの数に制

```
(defstruct proc pri state wait)

(proclaim '(special *procs* *proc*))

(defvar *halt* (gensym))

(defvar *default-proc*
  (make-proc :state #'(lambda (x)
                        (format t "~%>> ")
                        (princ (eval (read)))
                        (pick-process))))

(defmacro fork (expr pri)
  `(prog1 ',expr
     (push (make-proc
             :state #'(lambda (,(gensym))
                        ,expr
                        (pick-process))
             :pri      ,pri)
           *procs*)))

(defmacro program (name args &body body)
  `(=defun ,name ,args
           (setq *procs* nil)
           ,@body
           (catch *halt* (loop (pick-process)))))
```

図 115　プロセス構造体とその生成.

限はない．各プロセスには優先度が定まっており，その初期値はプロセスを生成するときの引数で与えられる．

wait 式を関数内で使うことができる．wait 式は，変数とテスト式と，実行本体となるコードを引数に取る．プロセスが wait 式に出会うと，テスト式が真を返すまでその時点で処理が中断される．プロセスが再始動すると，wait 式に与えた変数がテスト式の返り値に束縛された状態でコード本体が評価される．テスト式は普通は副作用を持つべきではない．いつ，どの位の頻度で評価が行われるかの保証がないからだ．

スケジューリングは優先度に基づいて行われる．再始動し得る全てのプロセスのうち，最高の優先度を持つプロセスをシステムが選んで始動させる．

他に始動するプロセスがないときにはデフォルト・プロセスが始動する．それは read-eval-print ループの一種だ．

大抵のオブジェクトは実行中に生成や破棄ができる．実行中のプロセスで新しい関数を定義することもできるし，別のプロセスを始動したり終了させたりもできる．

継続により，Lisp プログラムの状態を保持することができる．複数の状態を同時に保持していられることは，複数のプロセスがあることと大して違わない．複数プロセスを実装するには，前章で定義したマクロから始めて 60 行弱のコードしか必要ない．

20.2　実装

第 115図と第 116図には，複数プロセスに対応するために必要な全てのコードを示した．第 115図のコードは，基本的データ構造，デフォルト・プロセス，プロセスの初期化や生成のためのものだ．プロセスすなわち procs は以下の構造を持つ．

pri　はプロセスの優先度で，正の整数.

state　は中断されたプロセスの状態を表す継続．プロセスの再始動は，この状態を funcall に渡すことで行われる．

wait　は関数で，これが真を返さないとプロセスは再始動できない．しかし新たに生成されたばかりのプロセスでは nil になっている．wait が nil のプロセスは常に再始動できる．

```
(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 arbitrator (test cont)
      (setf (proc-state *proc*) cont
            (proc-wait *proc*) test)
      (push *proc* *procs*)
      (pick-process))

(defmacro wait (parm test &body body)
      `(arbitrator #'(lambda () ,test)
                   #'(lambda (,parm) ,@body)))

(defmacro yield (&body body)
      `(arbitrator 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)))
```

図 116　プロセスのスケジューリング.

プログラムは 3 つのグローバル変数を使う．*procs*は中断中のプロセスのリスト，*proc*は実行中のプロセス，*default-proc*はデフォルト・プロセスを格納する．

デフォルト・プロセスが実行されるのは，他に実行できるプロセスがないときだ．ちょうど Lisp のトップレベル環境に似せている．ユーザはこのループ内で，プログラムを停止させたり，中断中のプロセスを再始動させられる式を入力したりできる．デフォルト・プロセスは eval を陽に呼んでいることに注意．ここはそうすることが理に適う数少ない状況の一つだ．一般に，実行時に eval を呼ぶのはよい考えではない．その理由は以下の 2 つだ．

効率が悪い　eval は生のリストを渡されるが，それをその場でコンパイルなりインタプリタを使って評価なりせざるを得ない．どちらにせよ，コードを予めコンパイルしておいて呼び出すだけのときより遅くなる．

表現力が低い　式がレキシカル・コンテキストなしで評価されるからだ．とりわけ，評価される式の外側で見えている普通の変数にアクセスできない．

普通，eval を陽に呼ぶことは空港の土産物屋で買いものをするようなものだ．延々と待たされたあげく，品揃えの悪い，それも二流品ばかりの店に高い金を支払うことになる．

```
(defvar *open-doors* nil)

(=defun pedestrian ()
  (wait d (car *open-doors*)
    (format t "Entering ~A~%" d)))

(program ped ()
  (fork (pedestrian) 1))
```

図 117　wait 式 1 つを持つプロセス.

今の状況は上の議論がどちらも当てはまらない稀な例だ．ここでは式を予めコンパイルすることはあり得ない．その時点で読み込んでいるのだから，予めなどというものはない．同様に，式はそれを包むレキシカル環境を元々参照できない．トップレベルに入力された式はヌル・レキシカル環境の中にあるからだ．実際，この関数の定義は「ユーザが入力したものを読み込んで評価する」をそのまま反映している．

マクロ fork は関数呼び出しによってプロセスを生成する．関数は =defun によって通常通り定義できる．

```
(=defun foo (x)
  (format t "Foo was called with ~A.~%" x)
  (=values (1+ x)))
```

さて関数呼び出しと優先度を引数に fork を呼び出すと，

```
(fork (foo 2) 25)
```

新しいプロセスが *procs* にプッシュされる．その新プロセスは優先度が 25 で，（proc の）wait フィールドが nil（まだ始動させられていないため），state フィールドが 2 を引数とした foo の呼び出しになっている．

マクロ program によって，一群のプロセスを生成し，一緒に実行できる．次の定義は，2 つの fork 式が，中断されたプロセスを消去するコードと実行するプロセスを繰り返し選ぶコードに挟まれたものにマクロ展開される．

```
(program two-foos (a b)
  (fork (foo a) 99)
  (fork (foo b) 99))
```

このループの外側ではマクロがタグを設定しており，そこに制御を移すことでプログラムを終了できる．このタグは gensym なので，ユーザのコードが設定したタグと衝突しない．program で定義されたプロセスたちは特に値を返さず，トップレベルから呼ばれることのみを意図している．

プロセスが生成された後，第 116図に示したプロセスのスケジューリング用のコードが入れ替わりに起動する．関数 pick-process は，再始動可能なプロセスのうち優先度が一番高いものを選んで始動させる．そのプロセスを選ぶのは most-urgent-process の仕事だ．停止中のプロセスが始動できるのは，wait 式が中にないか，または wait 式が真を返したときだ．始動可能なプロセスのうちで最も優先度が高いものが選ばれる．選ばれたプロセスと，その中に wait 式があればその返り値が pick-process に渡される．デフォルト・プロセスが常に始動しようとしているので，必ずいずれかのプロセスが選ばれることになる．

第 116図の残りのコードはプロセス間で制御を切替えるためのオペレータを定義している．待機するための標準的な式は wait で，第 116図内の関数 pedestrian でも使われている．この例では，プロセスはリスト *open-doors* に要素が含まれない間は待機し，要素があるときはメッセージを表示する．

```
> (ped)
>> (push 'door2 *open-doors*)
Entering DOOR2
>> (halt)
NIL
```

wait は =bind（xgo ページ）に似ており，それと同様に，最後に評価されなければならないという制限を持つ．wait の後に行いたい処理は，全てその中に入れなければならない．よって wait を複数回使いたければ，wait 式はネストさせなければならない．互いを考慮した事実を仮定することで，プロセスは第 118図のようにゴールに到達するための共同作業が行える．

```
(defvar *bboard* nil)

(defun claim   (&rest f) (push f *bboard*))

(defun unclaim (&rest f) (pull f *bboard* :test #'equal))

(defun check   (&rest f) (find f *bboard* :test #'equal))

(=defun visitor (door)
  (format t "Approach ~A. " door)
  (claim 'knock door)
  (wait d (check 'open door)
    (format t "Enter ~A. " door)
    (unclaim 'knock door)
    (claim 'inside door)))

(=defun host (door)
  (wait k (check 'knock door)
    (format t "Open ~A. " door)
    (claim 'open door)
    (wait g (check 'inside door)
      (format t "Close ~A.~%" door)
      (unclaim 'open door))))

(program ballet ()
  (fork (visitor 'door1) 1)
  (fork (host 'door1) 1)
  (fork (visitor 'door2) 1)
  (fork (host 'door2) 1))
```

図 118　黒板を使った同期.

visitor と host により生成されたプロセスは，同じドアを与えられたとき，黒板上のメッセージを通じて制御をやりとりする.

```
> (ballet)
Approach DOOR2. Open DOOR2. Enter DOOR2. Close DOOR2.
Approach DOOR1. Open DOOR1. Enter DOOR1. Close DOOR1.
>>
```

他に wait 式の単純なものがある．yield の目的は，他の優先度の高いプロセスに始動を機会を譲ることだけだ．プロセスは式 setpri（これはカレント・プロセスの優先度を再設定する）の実行後に yield を実行するとよいかもしれない．wait と同様に，yield の後に実行したいコードは全てその内部に入れなければならない.

第 119図のプログラムでは，これらのオペレータがどのように協調して動作するかを示した.（訳注： capture は「捕らえる」，plunder は「略奪する」，take は「取る」，liberate は「(支配から解き放って）自由にする」，fortify は「要塞化する」，loot は「略奪する」，ransom は「身代金を要求する」，refinance は「財務を再構築する」という程の意味.）最初，蛮族 (barbarians) には 2 つの目的がある．ローマの制圧とそこでの略奪だ．ローマの制圧の方が（わずかに）優先度が高いので，そちらが先に実行される．しかしローマが制圧された後は，プロセス capture の優先度は 1 に下がる．そこでプロセス選択が行われ，その結果，優先度が最大のプロセス plunder が始動する.

```
> (barbarians)
Liberating ROME.
Nationalizing ROME.
Refinancing ROME.
Rebuilding ROME.
>>
```

蛮族がローマの宮殿で略奪を行い，貴族の身代金を奪った後にのみプロセス capture は停止し，蛮族は拠点の要塞化に移る.

```
                (wait d (car *bboard*) (=values d))

                (=defun capture (city)
                  (take city)
                  (setpri 1)
                  (yield
                    (fortify city)))

                (=defun plunder (city)
                  (loot city)
                  (ransom city))

                (defun take    (c) (format t "Liberating ~A.~%" c))
                (defun fortify (c) (format t "Rebuilding ~A.~%" c))
                (defun loot    (c) (format t "Nationalizing ~A.~%" c))
                (defun ransom  (c) (format t "Refinancing ~A.~%" c))

                (program barbarians ()
                  (fork (capture 'rome) 100)
                  (fork (plunder 'rome) 98))
```

図 119　優先度の変更の効果.

基盤となっている式 wait は，arbitrator を一般化したものだ．この関数はカレント・プロセスを保存し，pick-process を呼んで，何らかのプロセス（大抵は同じもの）を始動させる．これは 2 つの引数，テスト式と継続を取る．テスト式は停止中のプロセスの proc-wait として保持され，それを再始動するかどうか決定するために後で呼び出される．継続は proc-state に保持されるが，これを呼び出すと停止しているプロセスが再始動することになる．

マクロ wait と yield はその継続関数を生成するが，やっていることは単にその本体を λ 式で包むことだけだ．例えば，

```
(wait d (car *bboard*) (=values d))
```

は

```
(arbitrator #'(lambda () (car *bboard*))
  #'(lambda (d) (=values d)))
```

に展開される．

コードが第 113図の制限を満たしていれば，wait 本体のクロージャを生成するとき，現在の継続全体が保存される．第 2 引数の=values が展開されると，次のようになる．

```
#'(lambda (d) (funcall *cont* d))
```

クロージャが*cont*への参照を含むため，関数 wait で停止させられたプロセスは，停止した時点ではどこに置かれていたか，という目印を持つことになる．

オペレータ halt は，制御をプログラムの展開形が設定したタグに移すことでプログラム全体を停止させる．オプショナル引数にはプログラム全体の値として返すための値を渡せる．デフォルト・プロセスが常に始動しようとしているので，プログラムの終了には明示的に halt を呼ぶしかない．halt の後にどのようなコードが続いても，評価されることがないので意味がない．

各プロセスは式 kill を呼ぶことで終了させられる．引数無しでこれを呼ぶと現在実行中のプロセスが終了させられる．その場合，kill はカレント・プロセスの状態を保持しない wait の一種のようなものだ．kill に渡した引数は，プロセスのリストから削除するものを指定する．現状のプログラムでは参照すべきプロセスの属性が多くないので，kill について説明することは大してない．しかしもっと手の込んだシステムでは，他にタイムスタンプ，オーナ等の情報をプロセスに関連づけることになるだろう．デフォルト・プロセスはリスト *procs* に含まれていないので，kill を適用できない．

20.3 「早い」だけではないプロトタイプ

継続を使って模倣したプロセスには，本物の OS のプロセス程の効率は望めない．ならば，この章で示したプログラムの用途は一体何だろう？

それらの便利さは，ちょうどスケッチと同じだ．実験的なプログラミングやラピッド・プロトタイピングでは，プログラムはそれ自身で完成品なのではなく，むしろ考えを展開させるための乗り物だ．他の多くの領域では，この目的に使われるものはスケッチと呼ばれる．建築家は，原則的には，建物全体を頭の中で創り上げることができる．しかし，ほとんどの建築家は鉛筆を手にしていた方が考えが進むようだ．普通、建物のデザインは手始めにスケッチを何枚か描く中で考える．

ラピッド・プロトタイピングはソフトウェアのスケッチだ．建築家の下描きのように，ソフトウェアのプロトタイプは軽々としたタッチで描かれることが多い．アイディアを形にする最初期の一歩では，費用と効率の考慮は無視される．この段階での結果は，建築不可能な建物や，救いようもなく非効率的なソフトウェアになりがちだ．それであってもスケッチの価値は変わらない．なぜなら

1. 情報を簡潔に運ぶ入れ物であり，
2. 実験する機会を与える

からだ．

この章で説明したプログラムは，その前の章のプログラムと同様，スケッチであり，複数プロセスの概要を大雑把な筆遣いで提示している．商用ソフトウェアで使える程効率的ではないだろうが，複数プロセスの他の側面を使って，スケジューリングのアルゴリズムなどを実験するにはとても便利だ．

第 22–24 章では，継続の他の応用を示す．いずれも商用ソフトウェアで使える程効率のよいものではない．Lisp とラピッドプロトタイピングは共に発展してきたので，Lisp には特にプロトタイプを意図した，属性リスト，キーワード引数，そしてリストといった非効率的だが便利な機能が沢山備わっている．継続も恐らくこの仲間に入る．継続はプログラムが必要にするより多くの状態を保存する．だから，例えば後で出てくる継続ベースで実装された Prolog は，そのプログラミング言語を理解する方法としてはよいが，実装方法としては非効率的だ．

この本が重視するのは Lisp で実現できる抽象化であって，効率性に関わる話題ではない．しかし，Lisp はプロトタイプを書くための言語であるばかりでなく，商用ソフトウェアを書くための言語でもあるということを認識しておくことは重要だ．「Lisp は遅い」との評判があるとすれば，それは多分に多くのプログラマがプロトタイプで止まってしまったせいだ．Lisp では速いプログラムを簡単に書けるが，残念なことに，遅いプログラムも簡単に書ける．Lisp プログラムの初期版はダイアモンドのようなものだ．小さく，透き通っていて，そして非常に高価だ．それをそのままにしておきたいという欲求は大きいかもしれない．

他のプログラミング言語では，書いたプログラムを動作させるという骨の折れる課題に成功しさえすれば，その効率性はすでに許容可能なものだろう．床にタイルを敷くとき，タイルが爪程の大きさだったら，無駄になるタイルも少ない．この考えに基づいてプログラムを開発していた人には，「プログラムが動作したらそこで開発は完了する」という考えを乗り越えるのは難しいかもしれない．「Lisp を使うとプログラムが手間要らずで書ける．ああ，しかしそうすると遅いんだ.」と彼は思うかもしれない．実際は，どちらも当てはまらない．速いプログラムは実現できるし，しかしそれには手間をかけなければならない．この点では，Lisp を使うのは，貧しい国でなく豊かな国に暮らすようなものだ．痩せた体型を維持するために働かなければならないのは困ったことかもしれないが，命をつなぐために働いており，太ることなど問題外，という状況よりはずっとよい．

抽象度の低いプログラミング言語では，機能を求めて開発することになる．Lisp では，速度を求めて開発することになる．幸運なことに，速度を求めて開発する方が容易だ．大抵のプログラムでは，スピードに深刻に影響する部分は数か所しかない．

21 非決定性

プログラミング言語を使うことで，膨大な量の詳細に飲み込まれないで済んでいる．Lisp がよいプログラミング言語なのは，それ自身が多くの詳細を扱ってくれて，プログラマが耐えられる複雑さの限界を有効に使わせてくれている．この章ではマクロで Lisp にさらに別の種類の詳細を扱わせる方法を示す．非決定的なアルゴリズムを決定的なものに変換することの詳細だ．

この章は 5 つの部分に分けられる．まず，非決定性とは何かを説明する．次に，非決定的な *choose* と *fail* を継続を使って Scheme で実装する．3 番目では，第 20 章の継続渡しマクロを基礎に Common Lisp で実装した *choose* と *fail* を示す．4 番目では，オペレータ *cut* を Prolog とは独立に理解する方法を示す．最後に，非決定的オペレータの改良について考察する．

この章で定義される非決定的な選択オペレータは，第 23 章の ATN コンパイラと第 24 章の埋め込み Prolog を実装するのに使われる．

21.1 概念

非決定的アルゴリズムはある意味では超自然的な予見に基づいて動作するものだ．超能力を持ったコンピュータに触れることのない私達に，どうしてそんなものが必要なのだろうか？ それは非決定的アルゴリズムを決定的アルゴリズムでシミュレートできるからだ．純粋に関数的なプログラム——すなわち副作用の一切ないもの——では，非決定性は特に直截的になる．純粋に関数的なプログラムでは非決定性はバックトラックを用いた探索で実現できる．

この章では，関数的なプログラムにおいて非決定性をシミュレートする方法を示す．非決定性のシミュレータがあれば，非決定的なコンピュータが出すはずの答えを得ることが期待できる．多くの場合，超自然的な予見に頼ったプログラムを書く方がそうでないプログラムを書くより簡単だ．よって非決定性のシミュレータがあるとうれしい．

この節では非決定性によって得られる力のクラスを定義する．次の節ではサンプルにおいてそれらの利用を試してみる．これらの章の例は Scheme で書かれている．（Scheme と Common Lisp のいくらかの相異点は lxj ページにまとめてある）

非決定的アルゴリズムが決定的アルゴリズムと違うのは，2 つの特殊なオペレータ *choose* と *fail* が使える点だ．*choose* は有限集合を引数に取ってその要素を 1 つ返す関数だ．*choose* がどのように「選択する」かを説明するには，まず「未来の計算」という概念を導入する必要がある．

ここでは *choose* を，リストを引数に取ってその要素を 1 つ返す関数 `choose` として表現する．各要素について，それが選ばれた場合に計算が進み得る未来の集合が存在する．次の式では，

```
(let ((x (choose '(1 2 3))))
  (if (odd? x)
      (+ x 1)
      x))
```

計算が `choose` に行き着いたとき，3 つの未来があり得る．

1. `choose` が 1 を返す場合，計算は `if` の `then` 節へ進み，返り値は 2 になる．
2. `choose` が 2 を返す場合，計算は `if` の `else` 節へ進み，返り値は 2 になる．
3. `choose` が 3 を返す場合，計算は `if` の `then` 節へ進み，返り値は 4 になる．

この場合，`choose` の返り値が分かれば即座に，未来の計算がどうなるか正確に分かる．一般の場合では，各選択肢は可能な未来の集合と関連づけられている．未来の中にはさらに `choose` を使うものがあり得るからだ．例えば次の式では，

```
(let ((x (choose '(2 3))))
  (if (odd? x)
      (choose '(a b))
      x))
```

1 つ目の `choose` の時点で，2 つの未来の集合がある．

1. `choose` が 2 を返す場合，計算は `if` の `else` 節へ進み，返り値は 2 になる．

2. choose が 3 を返す場合，計算は if の then 節へ進む．この時点で，計算の道筋は 2 つの可能性に分岐する．a が返されるものと，b が返されるものだ．

第 1 の集合には 1 つの未来が，第 2 の集合には 2 つの未来が含まれるので，計算には合わせて 3 つの未来があることになる．

重要なのは，choose に複数の選択肢が与えられた場合，各選択肢に可能な未来の集合が関連づけられているという点だ．そのうちどれを返すべきだろうか？ choose の動作は，次のようになると思ってよい．

1. 選ぶ選択肢は，fail の呼び出しを含まない未来に進み得るもののみ．
2. 選択肢のない choose は fail と等価．

例えば次の式では，

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (fail)
      x))
```

各選択肢には未来が 1 つずつある．1 を選んだ場合の未来には fail の呼び出しが含まれるので，2 のみが選ばれる．よって上の式全体は決定的だ．つまり，必ず 2 を返す．

しかし，次の式は決定的ではない．

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (let ((y (choose '(a b))))
        (if (eq? y 'a)
            (fail)
            y))
      x))
```

1 つ目の choose において，1 を選んだ場合には 2 つの可能な未来があり，2 を選ぶと 1 つの可能な未来がある．しかし 1 を選ぶと，未来は実は決定的だ．a を選ぶと fail の呼び出しにつながってしまうからだ．よって式全体の返り値は b または 2 になり得る．

最後に，次の式の返り値は 1 つしかあり得ない．

```
(let ((x (choose '(1 2))))
  (if (odd? x)
      (choose '())
      x))
```

なぜなら，1 が選ばれた場合，未来は選択肢のない choose に行き着くからだ．つまりこの例は 2 つ前のものと等価だ．

ここまでの例からはまだ明らかにならないかもしれないが，たまげる程強力な抽象化手法が手に入ったのだ．非決定的アルゴリズムでは「後でどうなっても *fail* の呼び出しに行き着かないような要素を選べ」と言うことができる．例えば，次のコードは完全に正しい非決定的アルゴリズムで，Igor という名前の人が先祖にいるかどうかを調べる．

```
Function Ig(n)
  if name(n) = 'Igor'
    then return n
  else if parents(n)
    then return Ig(choose(parents(n)))
  else fail
```

オペレータ *fail* は *choose* の返り値に影響を与えるために使われている．*fail* に出会うことがあれば，*choose* の選んだ選択肢は誤っていたということだ．定義から *choose* は適切な選択肢を選ぶので，計算処理がある経路に進まないようにしたいときは，その経路のどこかに *fail* を置けば，その経路に計算が進むことはない．よって関数 Ig は祖先の世代を再帰的に辿る過程において，分岐点それぞれにおいて父方を選ぶか母方を選ぶか決定し，Igor 氏につながる経路を選ぶことができる．

それはあたかもプログラムが，*choose* に選択肢の中から何らかの要素を選ばせ，その返り値を必要とする限り何かに使い，そしてその後，*choose* がどれを選んでいたらよかったかを，*fail* を拒否の基準として，遡って決定できるような

```
(define (descent n1 n2)
  (if (eq? n1 n2)
      (list n2)
      (let ((p (try-paths (kids n1) n2)))
        (if p (cons n1 p) #f))))

(define (try-paths ns n2)
  (if (null? ns)
      #f
      (or (descent (car ns) n2)
          (try-paths (cdr ns) n2))))
```

図 120　決定的なツリーによる検索.

```
(define (descent n1 n2)
  (cond ((eq? n1 n2) (list n2))
        ((null? (kids n1)) (fail))
        (else (cons n1 (descent (choose (kids n1)) n2)))))
```

図 121　非決定的なツリーによる検索.

```
(define (two-numbers)
  (list (choose '(0 1 2 3 4 5))
        (choose '(0 1 2 3 4 5))))

(define (parlor-trick sum)
  (let ((nums (two-numbers)))
    (if (= (apply + nums) sum)
        `(the sum of ,@nums)
        (fail))))
```

図 122　サブルーチン内の choice.

ものだ．すると，そう，それがまさに *choose* の返した値だと分かる．よってモデルとしては *choose* に予知能力があることになる．

実際には *choose* に超自然的な力はない．*choose* のどの実装も，推定が誤っていたときにはバックトラックを行うことで，適切な推定をシミュレートしなければならない．ちょうどネズミが迷路の出口を探すようなものだ．しかしバックトラックは完全に裏で行うことができる．何らかの *choose* と *fail* が手に入れば，どちらの親をたどるべきかの推定が本当に可能かのように，上のようなアルゴリズムが書ける．*choose* を使うと，問題空間を探索するアルゴリズムを書くだけで，問題空間内で答えを探すアルゴリズムが書ける．

21.2　探索

古典的問題の多くは探索問題として定式化できるが，そのような問題ではしばしば非決定性が便利な抽象化手法だと分かる．nodes はツリーのノードのリストに束縛されており，(kids n) はノード n の子孫を返すか，または子孫がないなら#f を返す関数だとしよう．欲しい関数 (descent n1 n2) は，ノード n2 がノード n1 の子孫のとき，n1 から n2 までの経路上のノードのリストを返すものだ．第 120図には非決定性を使わない実装を示した．

非決定性を使うと経路を発見する際の詳細を忘れていられる．「目的とする節点につながる経路があるようなノード n を見つけろ」と choose に単に命ずるだけでよい．非決定性を使うと第 121図のように descent をもっと簡潔に描ける．

第 121図の方では適切な経路上のノードを明示的に探してはいない．choose が望ましい性質を持つ節点 n を選んだという仮定に基づいて書かれている．決定的なプログラムに見慣れていると，choose は，どの n ならば引き続く計算処理が失敗しないか推定できるかのように機能しなければならないことに気付かないかもしれない．

きっと choose の威力が一層際立つ例は呼出側の関数においてすらどのようなことが起きるか推測できる能力だろう．第 122図には，和が呼出側の関数の与えた数になるような 2 つの数を推定する関数の組を示した．最初の関数 two-numbers は非決定的に 2 つの数を選び，それらをリストに入れて返す．parlor-trick を呼び出したとき，中では two-numbers が呼ばれて 2 つの整数のリストが得られる．選択をするとき，two-numbers はユーザの入力した整数にアクセスできないことに注意しよう．

choose の推定した 2 つの整数の和がユーザの入力と等しくない場合，処理は失敗する．choose は失敗に終わる処理経路を避けることを利用してよい．よって呼出側が適切な範囲の数を与えたときは，choose は正しい選択をするものと仮定してよい（実際もそうなる）[*35]．

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

単純な探索では Common Lisp の組込み関数 find-if でも十分かもしれない．非決定的な探索の利点はどこにあるのだろう？ 選択肢のリストを，望ましい性質の要素を求めて頭から順に調べるのではいけないのだろうか？ choose と古典的な反復との決定的な違いは，失敗について無限のエクステントを持っていることだ．非決定的な choose は，任意の長さの未来を見通せる．choose の選んだ選択肢が将来のいずれかの時点で失敗に終わるときは，choose はその選択肢を避けると思っていてよい．parlor-trick で見たように，オペレータ fail は，制御が choose を呼んだ関数から戻った後ですら機能する．

この類の探索失敗は，例えば Prolog の行う探索でも見られる．非決定性が Prolog で便利なのは，その中心機能の一つがクエリに一つずつ答える能力だからだ．Prolog は，条件を満たす答えを全て一度に返すのではなくこのような道筋を辿ることで，下手をすると無限に大きい解集合を吐きかねない再帰的な規則を扱うことができる．

descent に対する第一印象はマージソートに対するものと似ているかもしれない．処理はどのように行われるのだろう？ マージソートと同様，処理は暗黙のうちに行われるが，しかし確かに行われる．第 22 章 3 節では，これまで示したコード例がいずれも実際のプロセスとして走るような choose の実装を解説する．

これらの例は抽象化技法としての非決定性の価値を示した．プログラミング言語の最高の抽象化は，キータイプ量だけでなく思考を節約させてくれる．オートマトン理論では非決定性を使わないと想像すら難しい証明もある．非決定性の使えるプログラミング言語はプログラマに同様の利点を与えてくれる．

21.3 Scheme での実装

この節では継続を使って非決定性をシミュレートする方法を示す．第 123図には choose と fail の Scheme による実装を示した．choose と fail は，裏側ではバックトラックによって非決定性をシミュレートする．バックトラックを使う探索プログラムは，どうにかして，選択肢が失敗に終わった場合に別の選択肢を辿るための十分な情報を保存しなければならない．その情報は継続の形でグローバルなリスト *paths* に保存される．

関数 choose には取り得る選択肢のリストが渡される．選択肢が空の場合は choose は fail を呼ぶが，これは計算を一つ前の choose まで戻す．選択肢が (first . rest) の形をしていれば，choose はまず *paths* に choose が rest に対して呼ばれるような継続をプッシュし，次に first を返す．

関数 fail の方が単純で，*paths* から継続をポップして呼び出すだけだ．*paths* が空リストのときは fail はシンボル @ を返す．しかし fail は，普通の関数が値を返すように @ を返すことはしない．それでは @ は一番近くの choose の返り値になってしまう．本当に望ましいのは @ をトップレベルに返すことだ．このために cc を fail が定義された時点での継続（恐らくはトップレベル）に束縛する．cc を呼び出すことで fail はトップレベルまで一気に戻ることができる．

第 123図の実装は *paths* をスタックとして使っており，失敗のときには必ず一番近い choice まで戻る．この戦略は時系列バックトラックと呼ばれ，結果的に問題空間の深さ優先探索を行うことになる．「非決定性」という言葉はしばしば深さ優先の実装の同義語かのように使われる．非決定的アルゴリズムについての Floyd の古典的な論文では「非決定性」をこの意味で使っており，またこれは非決定的パーザや Prolog で使われているものでもある．しかし第 123図の実装は唯一可能なものという訳ではなく，正しくすらないということに注意して欲しい．原則的には choose は任意

[*35] Common Lisp では引数の評価順が左から右となっているのに対し，Scheme では定められていないため，この呼び出しからは (THE SUM OF 5 2) が返されるかもしれない．

```
(define *paths* ())
(define failsym '@)

(define (choose choices)
  (if (null? choices)
      (fail)
      (call-with-current-continuation
        (lambda (cc)
          (set! *paths*
            (cons (lambda ()
                    (cc (choose (cdr choices))))
                  *paths*))
          (car choices)))))

(define fail)

(call-with-current-continuation
  (lambda (cc)
    (set! fail
      (lambda ()
        (if (null? *paths*)
            (cc failsym)
            (let ((p1 (car *paths*)))
              (set! *paths* (cdr *paths*))
              (p1)))))))
```

図 123　choose と fail の Scheme による実装.

の計算可能な条件を満たすようなオブジェクトを選べるべきだ．しかし上の choose と fail を使ってグラフを探索するプログラムは，グラフが閉路を含んでいると永久に終了しないかもしれない．

現実的には，非決定性とはすなわち第 123図に示した深さ優先探索を行うことであり，探索空間のループを避ける方法はプログラマに委ねられている．しかし興味のある読者のために，この章の最終節では本物の choose と fail の実装方法を示した．

21.4　Common Lisp での実装

この節では Common Lisp において choose と fail を書く方法を示す．前節で見たように，call/cc は Scheme で非決定性を実現するときに役立った．継続は，計算の未来という理論的概念を具現化したものを提供する．Common Lisp では代わりに第 20 章の継続渡しマクロを使う．これらのマクロを使うと choose は前節の Scheme 版に比べて少々汚くなるが，それでも実質的には等価になる．

第 124図には Common Lisp による fail の実装と，2 種類の choose の実装を示した．Common Lisp の choose の構文は Scheme 版とは少々異なる．Scheme の choose は，値を選ぶための選択肢リストを 1 つ引数に取った．Common Lisp 版は progn と同じ構文になっている．引数には任意個の式を渡し，それらから評価される式が選ばれる．

```
> (defun do2 (x)
    (choose (+ x 2) (* x 2) (expt x 2)))
DO2
> (do2 3)
5
> (fail)
6
```

トップレベルでは，非決定的な探索の背後にあるバックトラッキングが明確に見えてくる．変数 *paths* はまだ辿っていない経路を保存するために使われる．計算が複数の選択肢を持つ choose に辿り着くと，最初の選択肢が評価され，残りは *paths* に保存される．後にプログラムが fail に出会うと，最後に保存された選択肢が *paths* からポップされ，再始動する．再始動するための選択肢が残っていないときは，fail は特別な値を返す．

```
(defparameter *paths* nil)
(defconstant failsym '@)

(defmacro choose (\&rest choices)
  (if choices
      `(progn
         ,@(mapcar \#'(lambda (c)
                        `(push \#'(lambda () ,c) *paths*))
                   (reverse (cdr choices)))
         ,(car choices))
      '(fail)))

(defmacro choose-bind (var choices \&body body)
  `(cb \#'(lambda (,var) ,@body) ,choices))

(defun cb (fn choices)
  (if choices
      (progn
        (if (cdr choices)
            (push \#'(lambda () (cb fn (cdr choices)))
                  *paths*))
        (funcall fn (car choices)))
      (fail)))

(defun fail ()
  (if *paths*
      (funcall (pop *paths*))
      failsym))
```

図 124　Common Lisp による非決定的オペレータ.

```
> (fail)
9
> (fail)
@
```

第 124図では定数 failsym は失敗を表すが，シンボル @ として定義されている．@ も通常の返り値に含めたいときは代わりに gensym を failsym にすればよい.

非決定的な選択オペレータには他に choose-bind があるが，こちらは構文が多少異なる．引数にはシンボルと選択肢のリストと実行本体となるコードを取る．choose-bind は選択肢のリストに choose を適用し，選ばれた値を渡されたシンボルに束縛し，その下でコードを評価する.

```
> (choose-bind x '(marrakesh strasbourg vegas)
    (format nil "Let's go to ~A." x))
"Let's go to MARRAKESH."
> (fail)
"Let's go to STRASBOURG."
```

Common Lisp 版の実装でオペレータ choice を提供しているのは単に簡便のためだ．choose の機能は，choose-bind があれば十分得られる．choose を使う次のコードは,

```
(choose (foo) (bar))
```

次のように変換すればよい.

```
(choose-bind x '(1 2)
  (case x
    (1 (foo))
    (2 (bar))))
```

しかしこの場合は別々のオペレータがあった方が読み易くなる[*36].

[*36] 望みとあらば，このコードが外部に見せるインタフェイスはオペレータ一つだけでもよい．(fail) は (choose) と等価だからだ.

```
(=defun two-numbers ()
  (choose-bind n1 '(012345)
    (choose-bind n2 '(0 12345)
      (=values n1 n2))))

(=defun parlor-trick (sum)
  (=bind (n1 n2) (two-numbers)
    (if (= (+ n1 n2) sum)
        `(the sum of ,n1 ,n2)
        (fail))))
```

図 125　サブルーチン内で Common Lisp 版の choice を使う.

Common Lisp 版のオペレータ choice は，クロージャと変数捕獲により関連する変数の束縛を保存する．choose と choose-bind はマクロなので，それを包む式のレキシカル環境の中で展開される．それらが *paths* にプッシュするものは保存される選択肢を含むクロージャで，その中で参照されているレキシカル変数の全ての束縛を閉じ込めていることに注意．例えば次の式では,

```
(let ((x 2))
  (choose
    (+ x 1)
    (+ x 100)))
```

x の値は保存された選択肢が再始動するときに必要になる．だからこそ choose は引数を λ 式で包むように書かれているのだ．上の式は次のようにマクロ展開される．

```
(let ((x 2))
  (progn
    (push #'(lambda () (+ x 100))
          *paths*)
    (+ x 1)))
```

paths に保管されるオブジェクトは x へのポインタを含むクロージャだ．Scheme と Common Lisp とでオペレータ choice の構文が違うのは，クロージャ内の変数を保存する必要があるためだ.

choose と fail を第 20 章の継続渡しマクロと共に使うと，継続を保持する変数 *cont* へのポインタも同様に保管される．関数を =defun で定義し，=bind で呼び出し，その中では =values で値を返すようにすることで，任意の Common Lisp プログラムで非決定性が利用できるようになる.

これらのマクロを使うと，サブルーチン内で非決定的な選択を行う例をうまく動作させることができる．第 125図には Common Lisp 版の parlor-trick を示した．この動作は Scheme 版と同様だ.

```
> (parlor-trick 7)
(THE SUM OF 2 5)
```

これが動作するのは,

```
(=values n1 n2)
```

という式が choose-binds の中で次のようにマクロ展開されるからだ.

```
(funcall *cont* n1 n2)
```

choose-bind はそれぞれ順に，本体内で参照されている全ての変数（*cont* など）へのポインタを保持しているクロージャへとマクロ展開される.

choose，choose-bind と fail の使い方に関する制限は第 113図で示した継続渡しマクロを使うコードの制限と同じだ．choice を使うときは，それが最後に評価されなければならない．よって Common Lisp 版の choice を連続して使いたいときは入れ子にしなければならない.

```
> (choose-bind first-name '(henry william)
    (choose-bind last-name '(james higgins)
      (=values (list first-name last-name))))
(HENRY JAMES)
> (fail)
(HENRY HIGGINS)
```

```
> (=defun descent (n1 n2)
    (cond ((eq n1 n2) (=values (list n2)))
          ((kids n1) (choose-bind n (kids n1)
                       (=bind (p) (descent n n2)
                         (=values (cons n1 p)))))
          (t (fail))))
DESCENT
> (defun kids (n)
    (case n
      (a '(b c))
      (b '(d e))
      (c '(d f))
      (f '(g))))
KIDS
> (descent 'a 'g)
(ACFG)
> (fail)
@
> (descent 'a 'd)
(ABD)
> (fail)
(ACD)
> (fail)
@
> (descent 'a 'h)
@
```

図 126　Common Lisp における非決定的探索.

```
> (fail)
(WILLIAM JAMES)
```

これは今までと同様に深さ優先探索になる.

第 20 章で定義したオペレータには最後に評価されなければならないという際だった制限があった．この制限は新たなマクロの層では別の形として現れる．`=values` は `choose` の中で使わなければならず，その逆はできない．すなわち次のコードは動作するが,

```
(choose (=values 1) (=values 2))
```

次のコードは動作しない.

```
(=values (choose 1 2))                          ; 誤り
```

（後者では `=values` の展開形内に出てくる `*cont*` を `choose` の展開形が捕捉できない.）

ここと第 113図で示された制限に従う限り，Common Lisp でも非決定的選択が Scheme と同様に機能する．第 126図には第 121図で示した非決定的なツリーの探索プログラムの Common Lisp 版を示した．Common Lisp 版 `descent` は，少々長くて読み辛いが，直訳で書ける.

こうしてバックトラックを陽に使わなくとも非決定的探索が可能な Common Lisp ユーティリティが手に入った．それらを書く手間をかけた後は，それら無しでは長くごちゃごちゃとなりそうなプログラムをわずかな行数で書くことで利点を享受できる．ここで提示したユーティリティの上にさらにマクロの層を重ねることで，ATN コンパイラを 1 ページ分のコード（第 23 章）で，Prolog のプロトタイプを 2 ページ分のコード（第 24 章）で実現できる.

`choose` を使う Common Lisp プログラムは末尾呼び出しの最適化を行ってコンパイルしなければならない——速くするためだけでなく，スタック空間を使い果たすのを避けるためだ．継続を呼び出すことで値を「返す」プログラムからは，実際には最後に失敗するまで決して制御が戻らない．末尾呼び出しを最適化しないと，スタックはひたすら伸び続けてしまう.

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (newline)
    (let* ((store (choose '(1 2)))
           (box (choose '(1 2))))
      (let ((triple (list city store box)))
        (display triple)
        (if (coin? triple)
            (display 'c))
        (fail)))))

(define (coin? x)
  (member x '((la 1 2) (ny 1 1) (bos 2 2))))
```

図 127　当たりキャンディー検索の網羅的な例.

21.5　カット

この節では非決定的選択を行う Scheme プログラムでカットを使う方法を示す．カットという言葉は Prolog から来ているが，その概念そのものは一般の非決定性で通用する．非決定的選択を行うどのようなプログラムでもカットは使えるとうれしい.

カットは Prolog とは独立して考えると分かり易い．現実的な例を考えてみる．Chocoblob キャンディーの製造元が販促活動を始めるとしよう．キャンディーのうち何箱かには当たりの印のメダルが入っている．公平にするため，同じ街に 2 つ以上の当たり箱が送られないようにする.

販促活動の開始後，当たりの印のメダルが小さいので子供が飲み込む恐れがあることが分かった．法的な観点に駆り立てられ，Chocoblob 社の法務部は全ての当たり箱を必死に探すことになった．Chocoblob キャンディーを扱う店は街ごとにいくつもあり，キャンディーの箱は店ごとにいくつもある．しかし検査のためには全部の箱を開ける必要はない．ある街で当たり箱を 1 つ見つければ，その街では他の箱を開ける必要は一切ない．ある街にある当たり箱は高々 1 つだからだ．そしてこれを実現するのがカットなのだ.

カットとは何かというと，検索ツリーの一部分だ．キャンディー会社の例では検索ツリーは物理的に存在した．ルート・ノードは会社の統括オフィスだ．ルート・ノードの子ノードは当たり箱の送られた街で，それらの子ノードはそれぞれの街の店，そしてそれらの子ノードは店の中のキャンディーの箱を表す．このツリーに対し検索を行う法務部が当たり箱の一つを見つけたときは，現在いる街から伸びる全ての未探索の枝を枝刈りできる.

実際はカットには 2 つの動作が絡む．カットを行うのは検索ツリーのある部分が不要と分かったときだが，最初にツリーのカットできる部分にマークを付けなければならない．キャンディー会社の例では，ツリーへのマークは街を訪れるときにすればよいとすぐに分かる．Prolog のカットが何を行うのかを抽象的に説明するのは難しい．マークが暗黙のうちに行われるからだ．マークを陽につけるオペレータがある方がカットの動作は理解し易い.

第 127図にはキャンディー会社の例のツリーの小さいものに対し非決定的に検索を行うプログラムを示した．箱を開けてみるたびにプログラムは (**街 店 箱**) のリストを表示する．箱にメダルが入っていたらその後に c が表示される.

```
> (find-boxes)
(LA 1 1)
(LA 1 2)
C
(LA 2 1)
(LA 2 2)
(NY 1 1)
C
(NY 1 2)
(NY 2 1)
(NY 2 2)
(BOS 1 1)
```

```
(define (mark) (set! *paths* (cons fail *paths*)))

(define (cut)
  (cond ((null? *paths*))
        ((equal? (car *paths*) fail)
         (set! *paths* (cdr *paths*)))
        (else
          (set! *paths* (cdr *paths*))
          (cut))))
```

図 128　検索ツリーのマークと枝刈り.

```
(define (find-boxes)
  (set! *paths* ())
  (let ((city (choose '(la ny bos))))
    (mark)                                          ; 変更
    (newline)
    (let* ((store (choose '(1 2)))
           (box (choose '(1 2))))
      (let ((triple (list city store box)))
        (display triple)
        (if (coin? triple)
            (begin (cut) (display 'c)))             ; 変更
        (fail)))))
```

図 129　当たりキャンディー検索の枝刈りを用いた例.

```
(BOS 1 2)
(BOS 2 1)
(BOS 2 2)
C
@
```

上でキャンディー会社の発見した効率の良い検索方法を実装するには，2 つの新しいオペレータが必要になる．mark と cut だ．第 128図にはそれらの実装例を示した．非決定性そのものは個々の実装とは独立して考えられるが，検索ツリーの枝刈りは最適化手法であり，choose の実装法に強く依存する．第 128図に示した mark と cut は深さ優先の実装による choose に適している．

mark の基本的な仕組みは，マークを未探索の分岐点のリスト *paths* に保存することだ．cut を呼ぶと一番近いマークへの全ての経路を *paths* からポップする．さてマークには何を使おう？ 例えばシンボル m を使う方法もあるが，そうすると fail は m に出会ったときに無視するよう書き直さなければならない．うまいことに，関数もデータオブジェクトだから，fail がそのまま使えるマークが少なくとも一つある．それは関数 fail そのものだ．すると fail がマークとして現れたとき，単に自分自身を呼び出すことになる．

第 129図にはそれらのオペレータを使ってキャンディー会社の例の検索ツリーで枝刈りをする方法を示した．（変更のあった行はセミコロンで示した.）新たな街を訪れるごとに mark を呼ぶ．その時点では *paths* には「残りの街で検索を行う」という一つの継続が含まれる．

当たり箱を見つけたときには cut を呼ぶ．すると *paths* は最後に mark を呼んだ時点の値に戻る．cut の効果は次に fail を呼ぶときまでは明らかにならない．しかしそのときになると，当たり発見の表示の後，次に呼び出された fail は，途中に尽くされていない選択肢があっても，処理を一番根元の choose まで戻す．要するに，当たり箱を見つけたら即座に次の街の検索に移ることになる．

```
> (find-boxes)
(LA 1 1)
(LA 1 2)
C
(NY 1 1)
C
```

図 130　ループのある有向グラフ.

```
(define (path node1 node2)
  (bf-path node2 (list (list node1))))

(define (bf-path dest queue)
  (if (null? queue)
      '@
      (let* ((path (car queue))
             (node (car path)))
        (if (eq? node dest)
            (cdr (reverse path))
            (bf-path dest
                     (append (cdr queue)
                             (map (lambda (n)
                                    (cons n path))
                                  (neighbors node))))))))
```

図 131　決定的な検索.

```
(define (path node1 node2)
  (cond ((null? (neighbors node1)) (fail))
        ((memq node2 (neighbors node1)) (list node2))
        (else (let ((n (true-choose (neighbors node1))))
                (cons n (path n node2))))))
```

図 132　非決定的な検索.

```
(BOS 1 1)
(BOS 1 2)
(BOS 2 1)
(BOS 2 2)
C
@
```

こうすると 12 個も箱を開けなくとも，7 個で済む.

21.6　真の非決定性

グラフに対する決定的な検索プログラムは，ループに捕まるのを避けるための手立てを陽に講じなければならない．第 130図にはループのある有向グラフを示した．ノード a からノード e までの経路を探すプログラムには a, b, c から成るループに捕まる危険がある．決定的な手法では，乱数や広さ優先探索を使うか，ループを陽に検出しない限り，検索はいつまでも終了しないかもしれない．第 131図に示した path の実装は広さ優先探索によってループを避けている．

原則としては，非決定性を使うとループに対する考慮すら省くことができるはずだ．第 22–3 節で与えた，深さ優先の choose と fail の実装はループに捕まる危険性を孕んでいるが，几帳面にやれば，非決定的な choose が計算可能な仕様を満たす任意のオブジェクトを選べるようにできる．この例も例外ではない．choose を適切に書いておけば，第 132図に示したように path を短く簡潔に書ける．

この節ではループに対しても安全な choose と fail の実装方法を示す．第 133図には真の非決定的な choose と fail の Scheme 版を示した．ここで実装した choose と fail を使うプログラムは，等価な非決定的アルゴリズムが解を見つけられるときにはいつでも解を見つけ，ハード的な能力のみに制限される．

第 133図に示した真の choose の実装は保存された経路のリストをキューとして扱うことで機能する．真の choose を使うプログラムは状態空間を広さ優先で探索する．プログラムが分岐点に到達したとき，それぞれの選択肢を選ぶ継

```
(define *paths* ())
(define failsym '@)

(define (true-choose choices)
  (call-with-current-continuation
    (lambda (cc)
      (set!  *paths* (append *paths*
                             (map (lambda (choice)
                                    (lambda () (cc choice)))
                                  choices)))
      (fail))))

(define fail)

(call-with-current-continuation
  (lambda (cc)
    (set!  fail
      (lambda ()
        (if (null?  *paths*)
            (cc failsym)
            (let ((p1 (car *paths*)))
              (set!  *paths* (cdr *paths*))
              (p1)))))))
```

（Scheme の map は Common Lisp の mapcar と同じ.）

図 133　Scheme による真の choose.

続が保存された経路のリストの末尾に付加される．その後 fail が呼ばれるが，ここは変わっていない．

この実装の choose を使うと，第 132図の実装の path は第 130図に示したグラフの a から e への経路（実は最短経路）を発見できる．

記述の完全性のためにこの節では choose と fail の本当に正しい実装を示したが，元の実装でも大抵は十分だ．プログラミング言語の抽象化技法の価値は，その実装が形式的に正しくないからといって減りはしない．プログラミング言語によっては，利用できる最大の整数が 32767 なのに，全ての整数が利用できるかのようにコードを書くことがある．幻想にどこまで従えるかを知っている限りほとんど危険はない——少なくとも，実現できる抽象化と秤にかけてよい程度には少ない．次の 2 つの章で示すプログラムが簡潔なのは，多くは非決定的な choose と fail を使ったことに依る．

22　ATN を使ったパージング

この章では埋め込み言語の非決定的なパーザの実装法を示す．まず ATN パーザとは何か，またどのように文法規則を表現しているのかを説明する．次に前章で定義された非決定的オペレータを使う ATN コンパイラを示す．最後に小さな ATN 文法を取り上げ，サンプル入力をパーズするさまを示す．

22.1　背景

ATN とは Augmented Transition Networks の略称で，1970 年に Bill Woods により提唱されたパーザの一形態だ．それ以来，ATN は自然言語のパージングの形式的手法として広く用いられている．ちょっとした英文をパーズする ATN 文法は 1 時間もあれば書けるだろう．このため，初めてこれに出会った人はしばしば魔法に包まれたような気分になる．

1970 年代には，いつか ATN は真に知的なプログラムの一部になるかもしれないと考えた者もいた．この立場を取る者は現在では少ないが，ATN にはニッチ（適した居場所）が見つかった．人間ほど上手くは英文をパーズできないが，それでも表現力を持ったかなりの文をパーズできる．

```
(defnode s
  (cat noun s2
       (setr subj *)))

(defnode s2
  (cat verb s3
       (setr v *)))

(defnode s3
  (up `(sentence
         (subject ,(getr subj))
         (verb ,(getr v)))))
```

図 134　非常に小規模な ATN.

図 135　小規模な ATN のグラフ.

以下の制限に注意すれば ATN は役に立つ.

1. 特定のデータベースのフロントエンド等，意味論的に制限のある領域で使うこと.
2. 非常に込み入った入力を与えないこと．とりわけ，文法からひどく外れた文を人間並に理解できると思わないこと.
3. 英語等，語順が文法構造を決定する言語に対してのみ使うこと．ATN はラテン語等の屈折語のパージングにはあまり役立たない.
4. 常に機能するとは思わないこと．9 割の場合で機能すればありがたいと言えるようなときに使い，100% 機能しないと致命的ならば使わないこと.

これらの制限を満たす有用な応用法は数多い．オーソドックスな例はデータベースのフロントエンドだ．データベースに ATN を利用したインタフェイスを取り付ければ，ユーザは形式的なクエリを投げなくとも制限付きの英文で質問できる.

22.2　形式的な説明

ATN を理解するためにはフルネーム Augmented Transition Networks（拡張遷移ネットワーク）を思い出そう．遷移ネットワークとは矢印で結ばれたノードの集合で，本質的にはフローチャートだ．ノードのうち一つが始発ノードとして，別の幾つかのノードが終着ノードとして定められる．各矢印に条件が付加され，矢印を辿るにはそれを満たさなければならない．ATN には文が入力されるが，文には現在の単語を指すポインタが付属する．矢印を辿るとポインタが前進する．遷移ネットワークにおいて文のパージングとは始発ノードから終着ノードへの経路を探すことだ．ここで経路に沿った条件が全て満たされていなければならない.

ATN はこのモデルにさらに 2 つの条件を加えたものだ.

1. ATN にはレジスタすなわちパージングが進むと共に情報を格納する名前付きのスロットがある．条件を調べる他に，矢印はレジスタの内容を変更できる.
2. ATN は再帰的だ．矢印を辿る際には，パージングが何らかの副ネットワークの通過に成功することを要求してよい.

終着ノードはレジスタに累積された情報を使ってリスト構造を生成し，関数が値を返すのと全く同様にそれを返す．実際，非決定性を用いる点を除けば，ATN は関数的プログラミング言語とそっくりの動作をする.

第 134図で定義された ATN はほぼ一番単純なものだ．これは “Spot runs.” という形の名詞–動詞から成る文をパーズする．この ATN のネットワーク表現は第 135図に示した.

この ATN は（spot runs）という入力に対してどう反応するだろう？ 最初のノードから出る矢印は 1 つだけだ．cat すなわちカテゴリ (category) 矢印で，ノード s2 につながる．これは実質的には「現在見ている単語が名詞 (noun) ならば矢印を辿ってよい．そしてそのときは現在見ている単語（* で表される）をレジスタ subj に格納せよ」という意味だ．よってこのノードを離れる際にはレジスタ subj に spot が格納されている．

現在見ている単語を指すポインタが常に存在する．それは最初は文の先頭の単語を指している．カテゴリ矢印を辿るとき，そのポインタは 1 単語だけ前進する．よってノード s2 に到達したとき，見ている単語は 2 番目の runs になる．2 つ目の矢印は最初のものと同様だが，動詞 (verb) を求めている．runs が見つかるとそれをレジスタ v に格納し，処理は s3 に進む．

最後のノード s3 にはポップ矢印すなわち終端矢印しかない．（ポップ矢印を持つノードは点線で描いた．）入力を消費するにつれてポップ矢印に到達したので，パーズは成功だ．ポップ矢印はその中のバッククォート付きの式を返す．

```
(sentence (subject spot)
          (verb runs))
```

ATN はそれがパーズする言語の文法に対応している．英文をパーズするための立派な規模の ATN は，文をパーズするための主ネットワークと，名詞句，前置詞句，修飾句等をパーズするための副ネットワークから成る．

> “the key on the table in the hall of the house on the hill”

という風に，名詞句の中に前置詞句が含まれ，その中に名詞句が含まれ...と無限に続いてもよいことを考えれば，再帰構造が必要なのは明らかだ．

22.3 非決定性

先程の小さな例では現れなかったが，ATN では非決定性を使ってよい．一つのノードから複数の矢印が出ていてよく，与えられた入力に対してそれらのうち複数を辿ってよい．例えばちゃんとした ATN は命令的な文と宣言的な文の両方をパーズできるようでなくてはならない．よって始発ノードからは外向きのカテゴリ矢印が 2 つ，名詞（文のため）と動詞（命令のため）に対応して伸びているだろう．

文の最初の単語が “time” のように名詞でも動詞でもあり得るものならどうするのか？ パーザはどちらの矢印を辿ればよいかどのように判断するのだろう？「ATN は非決定性を持つ」と言うとき，それはどの矢印を辿るかパーザが適切に推定してくれると思ってよいということだ．パーズ失敗にしか行き着かない矢印をパーザが辿ることはない．

現実にはパーザは未来を予見することはできない．矢印や入力が尽きたときにはバックトラックを行うことで，適切な推定をシミュレートする．ATN コンパイラの生成したコードにはバックトラックの仕組みが自動的に挿入される．ATN はどの矢印を辿るかパーザが本当に推定できるかのように書ける．

非決定性を使う多くの（恐らくは大半の）プログラムと同様，ATN は深さ優先の実装を使う．さて英文をパーズしようとするとすぐ分かることだが，任意の英文をパーズする方法は数多くあるが，ほとんどは意味をなさない．伝統的なシングルプロセッサのマシンでは，意味のあるパーズ結果を素早く得たいものだ．全てのパーズ結果を一度に得るのではなく「一番ありそうなもの」のみを得ることにする．それについてまともな解釈ができれば，他のパーズ結果を求める手間が省けた訳だ．解釈ができなければ，fail を呼んでさらに別のパーズ結果を得る．

パーズ結果が生成される順番を制御するためには，プログラマはどうにかして choose が選択肢を選ぶ順番を制御する必要がある．深さ優先の実装が探索の順番を制御する唯一の方法ではない．ランダム以外のどのような実装でも何らかの順番がもたらされる．しかし ATN は，Prolog と同様に，概念的に深さ優先の実装と一体化している．ATN では，あるノードから伸びている矢印は定義された順に試される．この規約によってプログラマは矢印を優先度順に並べることができる．

22.4 ATN コンパイラ

普通，ATN を基盤としたパーザには 3 つの部分が要る．ATN そのもの，それを探索するためのインタプリタ，そして，例えば “runs” が動詞であると判断するための辞書だ．辞書はまた別の話題だ．ここでは初歩的な手製の辞書を使う．またネットワークインタプリタを扱う必要もない．ATN を直接 Lisp コードに変換するからだ．ここで解説する

```
(defmacro defnode (name &rest arcs)
  `(=defun ,name (pos regs) (choose ,@arcs)))

(defmacro down (sub next &rest cmds)
  `(=bind (* pos regs) (,sub pos (cons nil regs))
          (,next pos ,(compile-cmds cmds))))

(defmacro cat (cat next &rest cmds)
  `(if (= (length *sent*) pos)
       (fail)
       (let ((* (nth pos *sent*)))
         (if (member ',cat (types *))
             (,next (1+ pos) ,(compile-cmds cmds))
             (fail)))))

(defmacro jump (next &rest cmds)
  `(,next pos ,(compile-cmds cmds)))

(defun compile-cmds (cmds)
  (if (null cmds)
      'regs
      `(,@(car cmds) ,(compile-cmds (cdr cmds)))))

(defmacro up (expr)
  `(let ((* (nth pos *sent*)))
     (=values ,expr pos (cdr regs))))

(defmacro getr (key &optional (regs 'regs))
  `(let ((result (cdr (assoc ',key (car ,regs)))))
     (if (cdr result) result (car result))))

(defmacro set-register (key val regs)
  `(cons (cons (cons ,key ,val) (car ,regs))
         (cdr ,regs)))

(defmacro setr (key val regs)
  `(set-register ',key (list ,val) ,regs))

(defmacro pushr (key val regs)
  `(set-register ',key
                 (cons ,val (cdr (assoc ',key (car ,regs))))
                 ,regs))
```

図 136　ノードと矢印のコンパイル.

プログラムは，ATN 全体をコードに変換するので，ATN コンパイラと呼ばれる．ノードは関数に，矢印はそのなかのコードブロックに変換される．

第 6 章では関数を表現の一形態として導入した．普通，そうするとプログラムは速くなる．ここで「速くなる」というのは実行時にネットワークを解釈するオーヴァヘッドがなくなるということだ．欠点は何かがおかしくなったときに調べられる情報が少ないことだ．特に使っている Common Lisp の実装が関数 `function-lambda-expression` を提供しないときに研著だ.

第 136図には ATN を Lisp コードに変換するために必要なコードの全てを示した．マクロ `defnode` はノードの定義に使う．これはそれぞれの矢印に対して生成された式に単に `choose` を適用するだけの短いコードを生成する．ノードに対応する関数の 2 つの仮引数のうち，`pos` は現在の入力ポインタ（整数）で，`regs` は現在のレジスタの集合（a リストのリスト）だ.

マクロ `defnode` は対応するノードと同名のマクロを定義する．ノード `s` はマクロ `s` として定義されるのだ．この規

```
(defnode s
  (down np s/subj
        (setr mood 'decl)
        (setr subj *))
  (cat v v
       (setr mood 'imp)
       (setr subj '(np (pron you)))
       (setr aux nil)
       (setr v *)))
```

これは次のようにマクロ展開される．

```
(=defun s (pos regs)
  (choose
    (=bind (* pos regs) (np pos (cons nil regs))
           (s/subj pos
                   (setr mood 'decl
                         (setr subj * regs))))
    (if (= (length *sent*) pos)
        (fail)
        (let ((* (nth pos *sent*)))
          (if (member 'v (types *))
              (v (1+ pos)
                 (setr mood 'imp
                       (setr subj '(np (pron you))
                             (setr aux nil
                                   (setr v * regs)))))
              (fail))))))
```

図 137　ノードに対応する関数の展開形．

約のおかげで，矢印の指すノードを参照するときは単にその名前のマクロを呼べばよい．これはノード名を既存の関数やマクロの名前と被らせてはいけないということでもある．それらが再定義されてしまうからだ．

ATN のデバッグにはある種のトレース機能が必要だ．ノードは関数になるので独自のトレース機能を書く必要はなく，Common Lisp 組込みの関数 `trace` を使えばよい．rjz ページで触れたように，`=defun` を使ってノードを定義すると，`(trace =mods)` とすることでパージング処理がノード `mods` を通過するのをトレースできる．

ノードの本体内の矢印は単なるマクロ呼び出しで，`defnode` の生成する，ノードに対応する関数に埋め込まれるコードを返す．パーザは，各ノードにおいて，伸びる矢印を表現するコードそれぞれに対して `choose` を適用することで非決定性を使う．次のような複数の矢印が外に伸びるノードは，

```
(defnode foo
  ⟨矢印 1⟩
  ⟨矢印 2⟩)
```

次の形の関数定義に変換される．

```
(=defun foo (pos regs)
  (choose
    ⟨矢印 1 の変換結果⟩
    ⟨矢印 2 の変換結果⟩))
```

第 137図には第 144図の ATN の例の最初のノードのマクロ展開結果を示した．`s` のようなノードに対応する関数は，実行時に呼び出されたとき，非決定的に辿る矢印を選ぶ．仮引数 `pos` は入力文での現在の位置になり，`regs` は現在のレジスタの内容になる．

カテゴリ矢印は，元の例で見たように，入力の中の現在の単語がある文法的カテゴリに属することを表す．カテゴリ矢印の本体コード内ではシンボル `*` が入力中の現在の単語に束縛される．

`down` で定義されるプッシュ矢印では副ネットワークの呼び出しが成功しなければならない．それが引数に取るのは 2 つの目的地ノードで，副ネットワークの終着ノード `sub` と，現在のネットワークの中で次に進むノード `next` だ．カ

テゴリ矢印から生成されたコードは単にネットワーク内の次の矢印を呼び出しているだけだが，プッシュ矢印から生成されたコードは =bind を使っている点に注意．プッシュ矢印ではそれが指すノードに進む前に副ネットワークの呼び出しが成功しなければならない．空のレジスタの集合 (nil) は，regs の先頭にコンスされた後，副ネットワークに渡される．他の種類の矢印の本体内では，シンボル * が入力の現在見ている単語に束縛されるが，プッシュ矢印では副ネットワークの返した式に束縛される．

ジャンプ矢印は回路の短絡のようなものだ．パーザはそれが指すノードに即座に進む．条件判断は必要なく，入力ポインタも動かない．

最後にポップ矢印という種類があり，up で定義される．ポップ矢印は特別で，どのノードをも指さない．Lisp の return ではサブルーチンでなく呼出側の関数に処理が進むように，ポップ矢印からは新しいノードでなく「呼出側の」プッシュ矢印に進む．ポップ矢印内の =values は一番近接したプッシュ矢印の =bind に値を「返す」．しかし第 20-2 節で説明したように，Lisp の関数から普通に戻っているわけではない．=bind の本体は継続にまとめ上げられ，引数を通じて矢印に順繰りに渡されてゆき，最後にポップ矢印の =values が「返り値」を引数にそれを呼び出す．

第 22 章では非決定的な choose の実装を 2 通り説明した．探索空間にループがあると終了する保証のない高速版 choose（crq ページ）と，遅目だがそのようなループでも安全に機能する本物の choose（giz ページ）だ．もちろん ATN にはループがあり得るが，各ループ内で最低 1 単語は入力ポインタが前進する限り，パーザはいつしか文を読み尽くす．問題なのは入力ポインタを前進させないループだ．そのとき選択肢は 2 つある．

1. 遅目だが本物の非決定的オペレータ choice（giz ページで示した深さ優先版）を使う．
2. 高速版 choose を使うが，ジャンプ矢印のみから成るループを含むネットワークの定義はエラーと定める．

第 136図で定義したコードでは 2 番目の方法を取った．

第 136図の残り 4 つのマクロ定義は，矢印本体内でレジスタを読んだり書き換えるためのものだ．このプログラムではレジスタの集合は a リストとして表現されている．ATN はレジスタの集合を扱うのではなくレジスタの集合の集合を扱う．パーザが副ネットワークの処理に移るとき，空のレジスタの集合が既存のレジスタの集合の集合の上にプッシュされて与えられる．よってレジスタの集合全体は任意の時点において a リストのリストだ．

レジスタに対する定義済みオペレータは「現在の」すなわち一番上のレジスタの集合に作用する．getr はレジスタを読み，setr はレジスタを書き換え，pushr は値をレジスタにプッシュする．getr と pushr は共にプリミティブなレジスタ操作マクロ set-register を使う．

レジスタは宣言する必要がないことに注意．set-register に何らかの名前が与えられると，その名前のレジスタが作られる．

レジスタに対するオペレータはいずれも一切破壊的でない．cons，cons，cons，と set-register は繰り返す．このため処理は遅く，ごみも大量に生まれるが，ugz ページで説明したように，プログラムのうち継続が作られる部分で使われるオブジェクトには破壊的な変更を加えてはならない．一つの制御スレッド内のオブジェクトは今のところ中断中の別のスレッドと共有されているかもしれない．この場合，あるパーズ結果内のレジスタは他の多くのパーズ結果内のレジスタと構造を共有しているだろう．速度が問題になる場合は，レジスタを a リストでなくヴェクタに格納し，使用済のヴェクタを共通のプールを使って再利用する手がある．

プッシュ，カテゴリ，ジャンプの各矢印はいずれも式を本体として持つことができる．普通は単なる setr だろう．本体内で compile-cmds を呼ぶことで，それらの種類の矢印の展開形の関数は一連の setr を単一の式に撚り合わせることができる．

```
> (compile-cmds '((setr a b) (setr c d)))
(SETR A B (SETR C D REGS))
```

それぞれの式には次の式が最終引数として挿入される．ただし最後の式には regs が挿入される．よって矢印本体内の一連の式は新しいレジスタを返す単一の式に変換される．

この手法によって，任意の Lisp コードを progn で包むことで矢印の本体に挿入することができる．例：

```
> (compile-cmds '((setr a b)
                  (progn (princ "ek!"))
                  (setr c d)))
(SETR A B (PROGN (PRINC "ek!") (SETR C D REGS)))
```

```
(defmacro with-parses (node sent &body body)
  (with-gensyms (pos regs)
    `(progn
       (setq *sent* ,sent)
       (setq *paths* nil)
       (=bind (parse ,pos ,regs) (,node 0 '(nil))
         (if (= ,pos (length *sent*))
             (progn ,@body (fail))
             (fail))))))
```

図 138　最上階のマクロ.

図 139　大き目の ATN のグラフ.

幾つかの変数が矢印本体内のコードで参照できるようになっている．文全体はグローバルな *sent* に束縛される．また 2 つのレキシカル変数が参照できる．現在の入力ポインタを含む pos と現在のレジスタの集合を含む regs だ．これも意図的な変数捕獲の一例だ．これらの変数を参照できないようにするのが望ましければ gensym で置き換えればよい.

第 138図で定義したマクロ with-parses によって ATN を呼び出せるようになる．これを呼び出すには始発ノードの名前，パーズする式，パーズ結果を使って何をするかを記述したコード本体を与える．with-parses 内のコード本体は成功したパージングの結果それぞれにつき 1 回評価される．本体内ではシンボル parse が現在のパーズ結果に束縛される．一見，with-parses は dolist 等のオペレータに似ているが，内部では単なる反復ではなくバックトラッキング検索を使っている．with-parses はいずれは @ を返す．それが選択肢を使い切ったときの fail の返り値だからだ.

表現力のある ATN を見る前に，上で定義したちっぽけな ATN の生成したパーズ結果を見てみよう．第 136図の ATN コンパイラは types を呼び出して単語の文法的役割を決定するコードを生成する．よって最初にその定義が必要だ.

```
(defun types (w)
  (cdr (assoc w '((spot noun) (runs verb)))))
```

それでは始発ノードの名前を第 1 引数にして with-parses を呼んでみよう.

```
> (with-parses s '(spot runs)
    (format t "Parsing: ~A~%" parse))
Parsing: (SENTENCE (SUBJECT SPOT) (VERB RUNS))
@
```

22.5　ATN の例

ATN コンパイラ全体の説明が終わったので，サンプルのネットワークを使ってパージングを試してみよう．ATN パーザに豊富な種類の文を扱わせるには ATN 自身を複雑にすることになる．しかし ATN コンパイラはそのままでよい．ここで示されたコンパイラがおもちゃだというのは主に遅いせいであって，能力が限られている訳ではない.

パーザの機能（速度とはまた別）は文法の作り込みに依るところが多く，この限られた紙幅ではおもちゃ程度のものを作ることしかできない．第 141図から第 144図では第 139図で表現された ATN（もしくは ATN の集合）を定義している．このネットワークは，パーザに食わせる入力としては古典的な “Time flies like an arrow.” に対して複数のパーズ結果を生成できる程度には規模の大きいものだ.

複雑な入力のパージングにはそれなりに大きい辞書が必要になる．関数 types（第 140図）は最低限の辞書を提供する．22 語から成る語彙が定義され，各単語が 1 個もしくはそれ以上の単純な文法機能に関連付けられている.

ATN の要素はそれ自体が ATN だ．ここで定義する ATN の集合の中で一番単純なものは第 142図のもので，修飾語の列をパーズする．ただしここでは単なる名詞の列とした．第 1 のノード mods は名詞を受け付ける．第 2 のノード mods/n は，さらに名詞を探すか，パーズ結果を返すかのどちらかだ.

```
(defun types (word)
  (case word
    ((do does did) '(aux v))
    ((time times) '(n v))
    ((fly flies) '(n v))
    ((like) '(v prep))
    ((liked likes) '(v))
    ((a an the) '(det))
    ((arrow arrows) '(n))
    ((i you he she him her it) '(pron))))
```

図 140　僅かばかりの辞書.

```
(defnode mods
  (cat n mods/n
       (setr mods *)))

(defnode mods/n
  (cat n mods/n
       (pushr mods *))
  (up `(n-group ,(getr mods))))
```

図 141　修飾語の列に対する副ネットワーク.

第 3-4 節では関数的スタイルでプログラムを書くとどれ程テストし易くなるかを説明した.

1. 関数的プログラムでは部品を個別にテストできる.
2. Lisp では関数をトップレベル・ループでインタラクティブにテストできる.

これら 2 つの原則が相俟ってインタラクティブな開発が実現する．Lisp で関数的なプログラムを書くとき，各部分を書くにつれてテストできる.

ATN は関数的なプログラムにとても似ており——この実装では ATN は関数的プログラムにマクロ展開される——インタラクティブな開発がやはり可能だ．ATN のテストはどのノードから始めることもできる．単にノード名を with-parses の第 1 引数にすればよい.

```
> (with-parses mods '(time arrow)
    (format t "Parsing: ~A~%" parse))
Parsing: (N-GROUP (ARROW TIME))
@
```

次の 2 つのネットワークは相互に再帰的なので，一緒に説明しなければならない．第 142図で定義したネットワークはノード np から始まり，名詞句のパージングに使われる．第 143図で定義したネットワークは前置詞句をパーズする．名詞句は前置詞句を，前置詞句は名詞句を含んでいてよい．そのため 2 つのネットワークはそれぞれもう片方を呼ぶプッシュ矢印を含んでいる.

名詞句のネットワークには 6 個のノードがある．1 番目のノード np には 3 つの選択肢がある．代名詞を読み込んだ場合，処理はノード pron に進んでネットワークからのポップが行われる.

```
> (with-parses np '(it)
    (format t "Parsing: ~A~%" parse))
Parsing: (NP (PRONOUN IT))
@
```

その他の矢印はどちらもノード np/det につながる．片方の矢印は限定詞（“the” 等）を読み込み，もう片方は入力を読まず単にジャンプするだけだ．ノード np/det では両方の矢印が np/mods につながっている．np/det では副ネットワーク mods にプッシュして修飾語句の列を読み込むか，ジャンプするかの選択肢がある．ノード np-mods は名詞を読み込んで np/n に進む．np/n では結果をポップするか前置詞句ネットワークにプッシュして前置詞句を読み込むかのいずれかが可能だ．最後のノード np/pp は結果をポップする.

```
(defnode np
  (cat det np/det
       (setr det *))
  (jump np/det
        (setr det nil))
  (cat pron pron
              (setr n *)))

(defnode pron
  (up `(np (pronoun ,(getr n)))))

(defnode np/det
  (down mods np/mods
        (setr mods *))
  (jump np/mods
        (setr mods nil)))

(defnode np/mods
  (cat n np/n
       (setr n *)))

(defnode np/n
  (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))))
  (down pp np/pp
        (setr pp *)))

(defnode np/pp
  (up `(np (det ,(getr det))
           (modifiers ,(getr mods))
           (noun ,(getr n))
           ,(getr pp))))
```

図 142　名詞句に対する副ネットワーク.

```
(defnode pp
  (cat prep pp/prep
       (setr prep *)))

(defnode pp/prep
  (down np pp/np
        (setr op *)))

(defnode pp/np
  (up `(pp (prep ,(getr prep))
           (obj ,(getr op)))))
```

図 143　前置詞句に対する副ネットワーク.

```
(defnode s
  (down np s/subj
        (setr mood 'decl)
        (setr subj *))
  (cat v v
       (setr mood 'imp)
       (setr subj '(np (pron you)))
       (setr aux nil)
       (setr v *)))

(defnode s/subj
  (cat v v
       (setr aux nil)
       (setr v *)))

(defnode v
  (up `(s (mood ,(getr mood))
          (subj ,(getr subj))
          (vcl (aux ,(getr aux))
               (v ,(getr v)))))
  (down np s/obj
        (setr obj *)))

(defnode s/obj
  (up `(s (mood ,(getr mood))
          (subj ,(getr subj))
          (vcl (aux ,(getr aux))
               (v ,(getr v)))
          (obj ,(getr obj)))))
```

図 144　文に対するネットワーク.

名詞句の種類が異なればパージング過程も異なる．名詞句ネットワークについての 2 つのパージング結果を示そう．

```
> (with-parses np '(arrows)
    (pprint parse))
(NP (DET NIL)
    (MODIFIERS NIL)
    (NOUN ARROWS))
@
> (with-parses np '(a time fly like him)
    (pprint parse))
(NP (DET A)
    (MODIFIERS (N-GROUP TIME))
    (NOUN FLY)
    (PP (PREP LIKE)
        (OBJ (NP (PRONOUN HIM)))))
@
```

1 番目のパージング結果は np/det にジャンプし，np/mods に再びジャンプし，名詞を読み込み，np/n でポップしたところで成功裡に終了した．2 番目は一切ジャンプせず，まず修飾語句の列に対してプッシュし，前置詞句に対して再びプッシュした．パーザによくあることだが，構文的に整った式が意味論的には無意味で人間には構文構造の判別すら難しい程だということがある．ここで名詞句 “a time fly like him” は “a Lisp hacker like him” と同じ構造を持っている．

さて，これから必要なのは文構造の認識のためのネットワークだ．第 144図に示したネットワークは命令文と叙述文の両方をパーズする．始発ノードは慣習にしたがって s と名付けた．そこからつながったノードの 1 番目は文の主語となる名詞句に対してプッシュする．s から伸びる 2 番目の矢印は動詞を読み取る．文が構文的に曖昧なときはどちらの矢印でも辿り得ることがあり，第 144図に示したように，究極的には 2 つ以上のパージング結果につながる．1 番目のパージング結果は “Island nations like a navy” に似ており（訳注：“Time flies” が主語，“like” が動詞），2 番目は “Find someone like a policeman” に似ている（訳注：“Time” は「時間を計る」という動詞，“like” は前置詞）．さらに

```
> (with-parses s '(time flies like an arrow)
    (pprint parse))

(S (MOOD DECL)
   (SUBJ (NP (DET NIL)
             (MODIFIERS (N-GROUP TIME))
             (NOUN FLIES)))
   (VCL (AUX NIL)
        (V LIKE))
   (OBJ (NP (DET AN)
            (MODIFIERS NIL)
            (NOUN ARROW))))

(S (MOOD IMP)
   (SUBJ (NP (PRON YOU)))
   (VCL (AUX NIL)
        (V TIME))
   (OBJ (NP (DET NIL)
            (MODIFIERS NIL)
            (NOUN FLIES)
            (PP (PREP LIKE)
                (OBJ (NP (DET AN)
                         (MODIFIERS NIL)
                         (NOUN ARROW)))))))
@
```

図 145　1 つの文に対する 2 つのパージング結果.

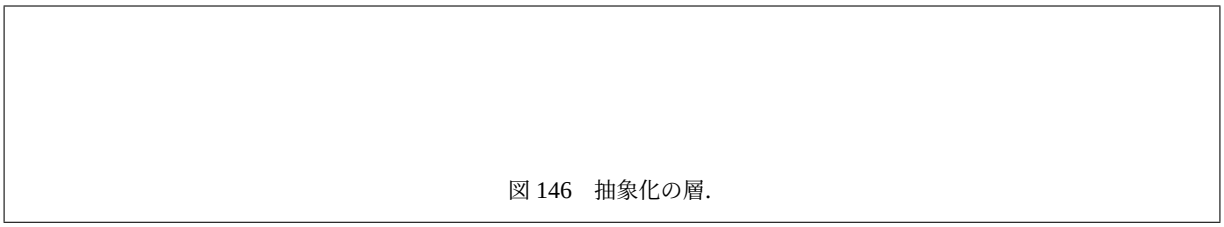

図 146　抽象化の層.

複雑な ATN では “Time flies like an arrow” に対し 6 つ以上のパージング結果が得られる.

この章の ATN コンパイラは商品ソフトウェアというより ATN というアイディアの真髄として提示したものだ．容易に思い付く変更をいくつか加えるだけであのコードはずっと効率が上がるだろう．速度が重要なときはそもそもクロージャで非決定性をシミュレートするというアイディアそのものが遅過ぎるかもしれない．しかし速度が本質的でないとき，ここで説明したプログラミング技法は非常に簡潔なプログラムにつながる.

23　Prolog

この章では埋め込み言語として Prolog を実装する方法を説明する．第 19 章ではデータベースに対する複雑なクエリに答えるプログラムを書く方法を示した．ここでは新たに要素を一つ加える．規則，すなわち既に知られている事実から別の事実を推論できるようにする仕組みだ．規則の集合は含意のツリーを定義する．無限個の事実を含意する規則を扱うには，その含意ツリーで非決定的な検索を行わなければならない.

Prolog は埋め込み言語の素晴しい例になる．それはパターンマッチング，非決定性，規則の 3 要素を組み合わせたものだ．第 18，22 章では最初の 2 つを独立に示した．今すでに手にしているパターンマッチングと非決定的選択のためのオペレータの上に Prolog を構築することで，多層構造を持つ本物のボトムアップシステムの例が得られる．第 146図には関係する抽象化の層を示した.

この章の第 2 の狙いは Prolog そのものの学習だ．経験豊かなプログラマにとって Prolog の最も役立つ解説はその実装の概論だろう．Lisp で Prolog を書くのは特に興味深い．2 つの言語の間の類似点が明らかになるからだ.

23.1 概念

第 19 章では変数を含む複雑なクエリを受け付け，クエリを真とするデータベース内の全ての束縛を生成するデータベースシステムを書く方法を示した．以下の例では（clear-db の呼び出し後）2 つの事実を仮定し，その下でデータベースにクエリを発行する．

```
> (fact painter reynolds)
(REYNOLDS)
> (fact painter gainsborough)
(GAINSBOROUGH)
> (with-answer (painter ?x)
    (print ?x))
GAINSBOROUGH
REYNOLDS
NIL
```

概念的には Prolog は規則を追加したデータベースシステムだ．規則により，データベース内を探すだけでなく，他の既知の事実から推論することでクエリを成立させられる．例えば次の規則を持っているとき（訳注：「腹を減らしていてテレピン油が臭う人は画家である」），

If (hungry ?x) and (smells-of ?x turpentine)
Then (painter ?x)

データベースに (hungry raoul) と (smells-of raoul turpentine) が登録されていれば (painter raoul) が登録されていなくともクエリ (painter ?x) は ?x = raoul という束縛で成立する．

Prolog では規則の if 節は本体と呼ばれ，then 節は頭部と呼ばれる．（論理学では前提と帰結という言葉を使うが，Prolog の推論は論理学的な含意とは違うことを強調するためにここで別の言葉を使うのはもっともなことだ．）クエリに対して束縛[*37]を作ろうとするとき，プログラムは先ず規則の頭部に注目する．プログラムの答えようとしているクエリに頭部がマッチしたとき，プログラムは規則の本体に対する束縛を生成しようとする．定義により，本体を成立させる束縛は頭部も成立させる．

規則の本体内で使われる事実は別の規則から推論されたものであってよい．

If (gaunt ?x) or (eats-ravenously ?x)
Then (hungry ?x)

そして規則は次のように再帰的であってもよい．

If (surname ?f ?n) and (father ?f ?c)
Then (surname ?c ?n)

Prolog は，規則の間に既知の事実に至る何らかの経路を見つけられるならばクエリに対して束縛を生成できる．よって Prolog は本質的には検索機構だ．Prolog は経路を求めて規則の形成する論理学的含意のツリーを探索する．

規則と事実は異なる種類のオブジェクトのように思えるが，概念的には交換して構わないものだ．規則は実質的に事実とみなせる．大型で獰猛な獣は珍しいという発見をデータベースに反映させたいときは，(species x), (big x), (fierce x) の成立する全ての x を探し，(rare x) を追加することもできるだろう．しかし，例えば次のような規則を定義することで同じ効果が得られ，

If (species ?x) and (big ?x) and (fierce ?x)
Then (rare ?x)

実際に (rare x) を適当な全ての x について追加する必要もない．無限個の事実を含意する規則の定義すらできる．そのため，規則が使えれば，問いに答える時点での余分な処理と引き換えにデータベースは小さくなる．

一方，事実は規則の縮退した形式だ．任意の事実 F は本体が常に真の規則と同じものだ．

If true
Then F

実装を簡素にするため，この原則を利用して事実を本体のない規則として表現することにする．

[*37] この意味での束縛を含む，この章で使われる概念の多くは，第 18-4 節で説明してある．

```
(defmacro with-inference (query &body body)
  `(progn
     (setq *paths* nil)
     (=bind (binds) (prove-query ',(rep_ query) nil)
            (let ,(mapcar #'(lambda (v)
                              `(,v (fullbind ',v binds)))
                          (vars-in query #'atom))
              ,@body
              (fail)))))

(defun rep_ (x)
  (if (atom x)
      (if (eq x '_) (gensym "?") x)
      (cons (rep_ (car x)) (rep_ (cdr x)))))

(defun fullbind (x b)
  (cond ((varsym? x) (aif2 (binding x b)
                           (fullbind it b)
                           (gensym)))
        ((atom x) x)
        (t (cons (fullbind (car x) b)
                 (fullbind (cdr x) b)))))

(defun varsym? (x)
  (and (symbolp x) (eq (char (symbol-name x) 0) #\?)))
```

図 147　トップレベルで使うマクロ.

23.2　インタプリタ

第 18-4 節では `if-match` の 2 通りの定義方法を示した．最初のものは単純だが非効率的だ．次に書いたものは処理の大部分をコンパイル時に行っていたので高速だった．ここでも似た戦略を取る．関連するポイントの導入のため，単純なインタプリタから始める．その後で同じことを行うずっと効率的なプログラムの書き方を示す.

第 147図から第 149図には単純な Prolog インタプリタのコードを示した．第 19-3 節のクエリ・インタプリタと同じクエリを受け付けるが，束縛を生成するのにデータベースでなく規則を使う．クエリ・インタプリタは `with-answer` というマクロから起動されるものだった．Prolog インタプリタへのインタフェイスも `with-inference` という名前の似たマクロを使うことになる．`with-answer` 同様に，`with-inference` にはクエリと一連の Lisp の式を与える．クエリ内の変数はクエスチョンマークで始まるシンボルだ.

```
(with-inference (painter ?x)
  (print ?x))
```

`with-inference` の呼び出しはクエリの生成した束縛の集合それぞれについて Lisp の式を評価するコードに展開される．例えば上の呼び出しは `(painter x)` を推論し得る全ての `x` を表示する.

第 147図には `with-inference` とそれが束縛を取得するために呼び出す関数の定義を示した．`with-answer` と `with-inference` との注意すべき違いは，前者が全てのあり得る束縛を単に集めていただけという点だ．`with-inference` は非決定的な検索を行う．それは `with-inference` の定義に見られる．ループに展開されるのではなく，束縛の集合から一つを返す部分に次に検索を再始動するための `fail` を続けたコードに展開される．こうして次のように反復が暗黙のうちに行われる.

```
> (choose-bind x '(0 1 2 3 4 5 6 7 8 9)
    (princ x)
    (if (= x 6) x (fail)))
0123456
6
```

関数 `fullbind` からは `with-answer` と `with-inference` との違いがまた一つ分かる．一連の規則を処理するうち

```
(=defun prove-query (expr binds)
  (case (car expr)
    (and (prove-and (cdr expr) binds))
    (or  (prove-or (cdr expr) binds))
    (not (prove-not (cadr expr) binds))
    (t   (prove-simple expr binds))))

(=defun prove-and (clauses binds)
  (if (null clauses)
      (=values binds)
      (=bind (binds) (prove-query (car clauses) binds)
             (prove-and (cdr clauses) binds))))

(=defun prove-or (clauses binds)
  (choose-bind c clauses
    (prove-query c binds)))

(=defun prove-not (expr binds)
  (let ((save-paths *paths*))
    (setq *paths* nil)
    (choose (=bind (b) (prove-query expr binds)
                   (setq *paths* save-paths)
                   (fail))
            (progn
              (setq *paths* save-paths)
              (=values binds)))))

(=defun prove-simple (query binds)
  (choose-bind r *rlist*
    (implies r query binds)))
```

図 148　クエリの解釈.

に，変数が他の変数のリストに束縛されているような束縛リストができ上がることもあり得る．クエリの結果を利用するには束縛を取得するための再帰関数が必要だ．それが fullbind の目的だ.

```
> (setq b '((?x . (?y . ?z)) (?y . foo) (?z . nil)))
((?X ?Y . ?Z) (?Y . FOO) (?Z))
> (values (binding '?x b))
(?Y . ?Z)
> (fullbind '?x b)
(FOO)
```

クエリに対する束縛は with-inference の展開形の中で prove-query を呼ぶことで生成される．第 148図には prove-query とそれが呼び出す関数の定義を示した．このコードは第 19-3 節で説明したクエリ・インタプリタと構造的に同型だ．どちらのプログラムもマッチングに同じ関数を使うが，クエリ・インタプリタでは写像や反復を使っていたところで，Prolog インタプリタは等価な choose を使っている.

反復でなく非決定的な検索を使ったことで否定を含むクエリの解釈はさらに複雑になった．次のようなクエリが与えられたとき,

```
(not (painter ?x))
```

クエリ・インタプリタでは (painter ?x) に対して束縛の生成を試みた後，一つでも生成されたら nil を返せばよかった．非決定的な検索ではさらに注意を払わなければならない．(painter ?x) の解釈で not のスコープ外にまでバックトラックされては困るし，後で探索を再始動するための経路を保存してもらっても困る．よって (painter ?x) を解釈するときは状態を保存するリストには一時的な空リストを使い，解釈終了後に古いリストを復元することにする.

Prolog インタプリタとクエリ・インタプリタでは，単純なパターン——(painter ?x) 等，述語と引数のみから成る式——の解釈にも違いがある．クエリ・インタプリタが単純なパターンに対して束縛を生成するときは，lookup（kga ページ）を呼んでいた．ここでは lookup を呼ぶのではなく，規則の含意する任意の束縛を得る必要がある.

```
(defvar *rlist* nil)

(defmacro <- (con &rest ant)
  (let ((ant (if (= (length ant) 1)
                 (car ant)
                 `(and ,@ant))))
    `(length (conc1f *rlist* (rep_ (cons ',ant ',con))))))

(=defun implies (r query binds)
  (let ((r2 (change-vars r)))
    (aif2 (match query (cdr r2) binds)
          (prove-query (car r2) it)
          (fail))))

(defun change-vars (r)
  (sublis (mapcar #'(lambda (v)
                      (cons v (symb '? (gensym))))
                  (vars-in r #'atom))
          r))
```

図 149　規則の関わるコード.

```
⟨規則⟩     : (<- ⟨文⟩ ⟨クエリ⟩ )
⟨クエリ⟩   : (not ⟨クエリ⟩ )
           : (and ⟨クエリ⟩* )
           : (or ⟨クエリ⟩* )
           : ⟨文⟩
⟨文⟩       : ( ⟨シンボル⟩ ⟨引数⟩* )
⟨引数⟩     : ⟨変数⟩
           : ⟨シンボル⟩
           : ⟨数⟩
⟨変数⟩     : ? ⟨シンボル⟩
```

図 150　規則の構文.

規則を定義し，使うためのコードは第 149図に示した．規則はグローバルなリスト `*rlist*` に保持される．規則は本体と頭部のドット対として表現される．規則が定義された時点で全ての下線（アンダースコア，_）は一意な変数に置き換えられる．

`<-` の定義はこの種のプログラムでしばしば見られる 3 つの慣習に従っている．

1. 新しい規則はリストの先頭ではなく末尾に追加され，定義された順に適用されるようになっている．
2. 規則は頭部を先として表現される．プログラムが規則を扱う順がそうなっているからだ．
3. 本体内に複数の式を入れると暗黙の `and` で括られる．

`<-` の展開形内の一番外側にある `length` は `<-` がトップレベルから呼ばれたときに巨大なリストの表示を避けるためだけのものだ．

規則の構文は第 150図に示した．規則の頭部は事実に対するパターン，すなわち述語の後に 0 個以上の引数を続けたリストでなければならない．本体には第 19 章のクエリ・インタプリタの扱える任意のクエリが使える．次のものはこの章の始めの方で示した規則だ．

```
(<- (painter ?x) (and (hungry ?x)
                      (smells-of ?x turpentine)))
```

もしくは単に次のように書ける．

```
(<- (painter ?x) (hungry ?x)
                 (smells-of ?x turpentine))
```

クエリ・インタプリタと同様，`turpentine` 等の引数は評価されないので引用符を付ける必要がない．

prove-simple がクエリに対して束縛を生成するとき，規則に対し非決定的な choose を適用し，規則とクエリの両方を implies に送る．すると関数 implies はクエリと規則の頭部とのマッチを試みる．マッチが成功すると implies は prove-query を呼んで本体に対する束縛を生成する．こうして含意のツリーを再帰的に探索できる．

関数 change-vars は規則内の変数を全て新しいものに置き換える．ある規則内で使われている変数 ?x は別の規則内の ?x とは別のものである筈だ．既存の束縛との衝突を避けるため，規則が使われる度に change-vars が呼ばれる．

簡便のため，規則内で _（下線）をワイルドカード変数として使える．規則が定義されたとき，関数 rep が呼ばれて各々の下線を本物の変数に置き換える．下線は with-inference に与えるクエリでも使える．

23.3 規則

この節では埋め込み Prolog のための規則の書き方を示す．第 24-1 節の 2 つの規則から始めよう．

```
(<- (painter ?x) (hungry ?x)
                 (smells-of ?x turpentine))

(<- (hungry ?x) (or (gaunt ?x) (eats-ravenously ?x)))
```

次の事実を仮定すれば,

```
(<- (gaunt raoul))
(<- (smells-of raoul turpentine))
(<- (painter rubens))
```

規則の生成する束縛が，定義された順に得られる．

```
> (with-inference (painter ?x)
    (print ?x))
RAOUL
RUBENS
@
```

マクロ with-inference は変数束縛について with-answer と全く同じ制限を持つ．（第 19-4 節を参照）

任意の形の事実が全ての可能な束縛について真となることを含意する規則が書ける．これは例えば何らかの変数が規則の頭部内で使われているが本体内では使われていないときだ．次の規則は，?x が暴食家ならば ?x は何でも食べることを示す.

```
(<- (eats ?x ?f) (glutton ?x))
```

?f は本体内では使われていないので，(eats ?x y) という形の任意の事実が，?x について束縛を生成するだけで示せる．eats への第 2 引数にリテラル値を与えてクエリを発行すると,

```
> (<- (glutton hubert))
7
> (with-inference (eats ?x spinach)
    (print ?x))
HUBERT
@
```

任意のリテラル値が使える．第 2 引数に変数を与えると,

```
> (with-inference (eats ?x ?y)
    (print (list ?x ?y)))
(HUBERT #:G229)
@
```

gensym が帰って来る．クエリ内の変数の束縛として gensym を返すのは，任意の値をそこに持ってきても真となることを示している．この規約を明示的に活用してプログラムを書ける．

```
> (progn
    (<- (eats monster bad-children))
    (<- (eats warhol candy)))
9
> (with-inference (eats ?x ?y)
    (format t "~A eats ~A.~%"
            ?x
            (if (gensym? ?y) 'everything ?y)))
```

Prolog インタプリタの文法は本物の Prolog 文法と次のような点で異なる．

1. 変数が大文字でなくクエスチョンマークで始まるシンボルで表現される．Common Lisp では普通は大文字小文字の区別をしないので，大文字の単語を変数とみなすようにすると無駄に面倒になる．
2. `[]` の代わりに `nil` を使う．
3. `[x | y]` という形の式の代わりに `(x . y)` を使う．
4. `[x, y, ...]` という形の式の代わりに `(x y ...)` を使う．
5. 述語が括弧の中に移動し，引数を区切るコンマがなくなった．すなわち `pred(x, y, ...)` の代わりに `(pred x y ...)` を使う．

よって Prolog による append の定義は，

```
append([ ], Xs, Xs).
append([X | Xs], Ys, [X | Zs]) <- append(Xs, Ys, Zs).
```

次のようになる．

```
(<- (append nil ?xs ?xs))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
    (append ?xs ?ys ?zs))
```

図 151　Prolog 文法との対比．

```
HUBERT eats EVERYTHING.
MONSTER eats BAD-CHILDREN.
WARHOL eats CANDY.
@
```

最後にある形の事実が任意の引数について真だと指定したいときは，本体を引数なしの接続詞とすればよい．式 `(and)` は常に真となる事実の役割を果たす．マクロ `<-`（第 149図）では本体が与えられなかったときに `(and)` が使われる．よって常に真となる規則では本体が省ける．

```
> (<- (identical ?x ?x))
10
> (with-inference (identical a ?x)
    (print ?x))
A
@
```

Prolog の知識をお持ちの方のために，第 151図には Prolog 文法から Lisp による Prolog インタプリタへの翻訳方法を示した．伝統的な Prolog プログラムとしては先ず append が挙げられるが，第 151図の最後に示したようになる．append の使って 2 つの短いリストをつなげて 1 つの長いリストを作れる．これらのリストのうちどの 2 つを定めても残りの 1 つのリストがどうなるべきかが定まる．Lisp の関数 append は 2 つの短いリストを引数に取ってつなげた長いリストを返すが，Prolog の append はより一般性が高い．第 151図の 2 つの規則は，関連するどの 2 つのリストが与えられても 3 つ目を得られるプログラムを定義する．

```
> (with-inference (append ?x (c d) (a b c d))
    (format t "Left: ~A~%" ?x))
Left: (A B)
@
> (with-inference (append (a b) ?x (a b c d))
    (format t "Right: ~A~%" ?x))
Right: (C D)
@
> (with-inference (append (a b) (c d) ?x)
    (format t "Whole: ~A~%" ?x))
Whole: (A B C D)
@
```

それだけでなく，最後のリストを与えられただけでも，1 番目と 2 番目のリストの可能性を全て見つけることができる．

```
> (with-inference (append ?x ?y (a b c))
    (format t "Left: ~A Right: ~A~%" ?x ?y))
Left: NIL Right: (A B C)
```

```
Left: (A) Right: (B C)
Left: (A B) Right: (C)
Left: (A B C) Right: NIL
@
```

append の振舞いは Prolog と他のプログラミング言語との大きな違いを示している．Prolog の規則の集合は必ずしも確定した値を持たなくともよく，プログラムの他の部分の定める制約と組み合わせたときになって特定の値を定めるような制約を持つことができる．例えば member を次のように定義すると，

```
(<- (member ?x (?x . ?rest)))
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
```

Lisp の関数 member と同じように，リストへの要素の所属を調べるのに使える．

```
> (with-inference (member a (a b)) (print t))
T
@
```

しかしそれは所属という制約を定めるためにも使える．その制約は他の制約と組み合わさったときに特定のリストを定める．さらに car 部が a であるような 2 要素のリスト全てに真を返す述語 cara を定めると，

```
(<- (cara (a _)))
```

これと member によって Prolog は十分に確定した答えを導き出せる．

```
> (with-inference (and (cara ?lst) (member b ?lst))
    (print ?lst))
(A B)
@
```

これはかなり自明な例だが，複雑な問題も同じ原則の下に立って構築できる．部分解を組み合わせてプログラミングを行いたいときには Prolog は役立つかもしれない．実際，驚く程多様な問題がその方式で表現できる．例えば第 159図には解についての制約の集積として表現されたソートアルゴリズムを示した．

23.4 非決定性の必要性

第 22 章では決定的な検索と非決定的な検索との関係について説明した．決定的な検索プログラムはクエリを受け取ってそれを満たす解を全て生成することになる．非決定的検索プログラムは choose を使って解を 1 つずつ生成し，他の解が必要とあらば fail を呼んで検索を再開する．

手にしている規則の生成する束縛がいずれも有限個で，それら全てを一度に知りたいとき，非決定的検索を選ぶ理由はどこにもない．2 つの戦略の違いは無限個の束縛を生成するクエリを前にすると明らかになる．その束縛の中から有限個の部分集合を得たいのだ．例えば次の規則は，

```
(<- (all-elements ?x nil))
(<- (all-elements ?x (?x . ?rest))
        (all-elements ?x ?rest))
```

(all-elements x y) という形を持つあらゆる事実を含意する．ここで y のメンバはいずれも x と equal だ．バックトラックなしではクエリを次のように扱うことになるだろう．

```
(all-elements a (a a a))
(all-elements a (a a b))
(all-elements ?x (a a a))
```

しかしクエリ (all-elements a ?x) を満たす ?x には無限個の候補がある．nil，(a)，(a a) 等だ．このクエリに対し反復を使って解を生成しようとするとその反復はいつまでも終了しない．そのうち一つが欲しいだけであっても，Lisp の式の処理を開始する前にクエリに対し全ての束縛を生成する必要のある実装では結果はいつまでも得られない．

この理由から with-inference では束縛の生成を本体の評価と交互に行っている．クエリが無限個の解を持つ場合には，解を一つずつ生成し，中断した検索を再始動させて次の解を得る手立てしかない．埋め込み Prolog では choose と fail を使っているため，次のような場合も扱える．

```
> (block nil
    (with-inference (all-elements a ?x)
      (if (= (length ?x) 3)
          (return ?x)
```

```
(defmacro with-inference (query &rest body)
  (let ((vars (vars-in query #'simple?)) (gb (gensym)))
    `(with-gensyms ,vars
       (setq *paths* nil)
       (=bind (,gb) ,(gen-query (rep_ query))
         (let ,(mapcar #'(lambda (v)
                           `(,v (fullbind ,v ,gb)))
                       vars)
           ,@body)
         (fail)))))

(defun varsym? (x)
  (and (symbolp x) (not (symbol-package x))))
```

図 152　新しいトップレベル用マクロ.

```
        (princ ?x))))
NIL
(A)
(A A)
(A A A)
```

他の Prolog の実装と同様，埋め込み Prolog では非決定性をシミュレートするのにバックトラックを伴う深さ優先探索を行っている．理論的には「論理プログラミング」は本物の非決定性の下で動作する．実際には Prolog の実装は必ず深さ優先探索を使う．その選択を行ったことで困るのとは正反対で，典型的な Prolog プログラムはそれに依存している．真に非決定的な世界では次のクエリは解を持つが,

```
(and (all-elements a ?x) (length ?x 3))
```

その解が何かを知るには任意の長さの時間がかかる.

Prolog は深さ優先の非決定性の実装を使っているだけでなく，zxi ページで定義した版と同じものを使っている．そこで説明したように，その実装では必ず終了することが保証されていない．よって Prolog プログラマは探索空間でのループを慎重に避けなければならない．例えば member を逆の順で定義していたら,

```
(<- (member ?x (_ . ?rest)) (member ?x ?rest))
(<- (member ?x (?x . ?rest)))
```

論理的には同じ意味を持つはずだが，Prolog プログラムとしては異なるものになる．member の元の定義ではクエリ (member 'a ?x) に対して解の無限のストリームを生成することになるが，逆にした定義では無限の再帰が起き，解は一つも得られない.

23.5　新しい実装

この節では親しみのあるパターンがまた出てくる．第 18-4 節では，プロトタイプを書いた後に if-match をずっと速くする方法に気付いた．コンパイル時に分かっている情報を活用することで，実行時に行う処理の少ない新しいヴァージョンに書き改めることができた．第 19 章では同じ現象が大規模になって出てきた．クエリ・インタプリタが等価だが速度の速いものに置き換えられた．ここでは Prolog インタプリタでも同じことをやろうというのだ.

第 152，153，155図では Prolog を別のやり方で定義している．マクロ with-inference は Prolog インタプリタへの単なるインタフェイスだったが，ここではプログラムの大部分を占める．新しい Prolog インタプリタは概形では古いものと変わらないが，第 153図で定義された関数のうち，実行時に呼ばれるのは prove のみだ．他の関数は with-inference がその展開形を作るために呼ぶためのものだ.

第 152図には with-inference の新しい定義を示した．if-match や with-answer と同様，パターン変数は最初は gensym に束縛されており，マッチングによって本当の値が与えられる前ということを示している．よって match と fullbind が変数を判別するために呼ぶ関数 varsym? は，gensym を探すよう変更しなければならない.

クエリに対する束縛を定めるコードを生成するため, with-inference は gen-query（第 153図）を呼ぶ．gen-query が先ず行う処理は，第 1 引数を見て，それが and や or 等のオペレータで始まる複雑なクエリかどうか調べることだ.

```
(defun gen-query (expr &optional binds)
  (case (car expr)
    (and (gen-and (cdr expr) binds))
    (or  (gen-or (cdr expr) binds))
    (not (gen-not (cadr expr) binds))
    (t   `(prove (list ',(car expr)
                       ,@(mapcar #'form (cdr expr)))
                 ,binds))))

(defun gen-and (clauses binds)
  (if (null clauses)
      `(=values ,binds)
      (let ((gb (gensym)))
        `(=bind (,gb) ,(gen-query (car clauses) binds)
           ,(gen-and (cdr clauses) gb)))))

(defun gen-or (clauses binds)
  `(choose
     ,@(mapcar #'(lambda (c) (gen-query c binds))
               clauses)))

(defun gen-not (expr binds)
  (let ((gpaths (gensym)))
    `(let ((,gpaths *paths*))
       (setq *paths* nil)
       (choose (=bind (b) ,(gen-query expr binds)
                      (setq *paths* ,gpaths)
                      (fail))
               (progn
                 (setq *paths* ,gpaths)
                 (=values ,binds))))))

(=defun prove (query binds)
        (choose-bind r *rules* (=funcall r query binds)))

(defun form (pat)
  (if (simple? pat)
      pat
      `(cons ,(form (car pat)) ,(form (cdr pat)))))
```

図 153　クエリのコンパイル.

この処理は単純なクエリに到達するまで再帰的に続き，単純なクエリは prove の呼び出しに展開される．最初の実装ではそういった論理的構造は実行時に分析されていた．規則の本体内に使われた複雑な式は，その規則が使われる度に改めて分析する必要があった．規則とクエリの論理的構造は予め分かっているのだから，これは無駄が多い．新しい実装では複雑な式をコンパイル時に分解する．

最初の実装と同様に，with-inference の呼び出しは，パターン変数を規則の定めた値にそれぞれ束縛して，クエリに伴う Lisp コードを反復実行するコードに展開される．with-inference の展開形は fail で終わっており，任意の保存された状態が再始動される．

第 153図の残りの関数は複雑なクエリ——and，or や not で結合されたもの——の展開形を生成する．次のようなクエリがあったとき，

```
(and (big ?x) (red ?x))
```

2 つの条件が共に満たされるような ?x についてのみ Lisp コードを評価したい．よって and を使った式の展開形を生成するには，第 2 の条件の展開形を第 1 の中に入れ子にする．(big ?x) が真のときに (red ?x) を調べ，それも真だったら Lisp の式を評価する．よって展開形の全体は第 154図のようになる．

and では入れ子を使い，or では choose を使う．次のようなクエリがあったときは，

```
(with-inference (and (big ?x) (red ?x))
  (print ?x))
```

これは次のようにマクロ展開される.

```
(with-gensyms (?x)
  (setq *paths* nil)
  (=bind (#:g1) (=bind (#:g2) (prove (list 'big ?x) nil)
                      (=bind (#:g3) (prove (list 'red ?x) #:g2)
                             (=values #:g3)))
        (let ((?x (fullbind ?x #:g1)))
          (print ?x))
        (fail)))
```

図 154　オペレータ and の展開形.

```
(defvar *rules* nil)

(defmacro <- (con &rest ant)
  (let ((ant (if (= (length ant) 1)
                 (car ant)
                 `(and ,@ant))))
    `(length (conc1f *rules*
                     ,(rule-fn (rep_ ant) (rep_ con))))))

(defun rule-fn (ant con)
  (with-gensyms (val win fact binds)
    `(=lambda (,fact ,binds)
       (with-gensyms ,(vars-in (list ant con) \#'simple?)
         (multiple-value-bind
           (,val ,win)
           (match ,fact
                  (list ',(car con)
                        ,@(mapcar \#'form (cdr con)))
                  ,binds)
           (if ,win
               ,(gen-query ant val)
               (fail)))))))
```

図 155　規則の定義のためのコード.

`(or (big ?x) (red ?x))`
どちらかの部分クエリが定めた `?x` の値に対して Lisp の式を評価したい．関数 `gen-or` はそれぞれ `gen-query` を適用した引数に対しての `choose` に展開される．`not` については，`gen-not` は `prove-not`（第 148図）とほぼ同じだ.

第 155図には規則を定義するためのコードを示した．規則は `rule-fn` の生成する Lisp コードに直接変換される．`<-` が規則を Lisp コードへと展開するようになったので，規則の定義を並べたファイルのコンパイルは規則をコンパイル済み関数に変える働きを持つ.

`rule-function` にパターンを与えると，そのパターンをそれが表現するルールの頭部に対してマッチさせようとする．マッチが成功すると `rule-function` は本体に対する束縛を生成しようとする．この処理は本質的に `with-inference` と同じで，実際，`rule-fn` は最後に `gen-query` を呼んでいる．結局 `rule-function` は規則の頭部内で使われている変数に対する束縛を返す.

23.6　Prolog の機能の追加

ここまでに提示したコードで大抵の「純粋な」Prolog プログラムが実行できる．最後にカット，算術演算，I/O 等を追加する.

Prolog の規則にカットを加えると検索木の枝刈りがなされる．普通，プログラムが fail に出会うと，最後に通った choice の地点までバックトラックする．第 22-4 章の choose の実装は非決定的選択を行った場所をグローバル変数 *paths* に保持していた．fail を呼ぶと検索は一番近くの選択場所で再始動するが，その場所は *paths* の car 部に保持されている．カットのせいでさらに複雑になる．プログラムがカットに出会うと，*paths* に保持されていた選択場所のうち近いものを幾つか——詳しく言えば最後に prove が呼ばれた以降のものを——捨てる．

カットの効果は規則を互いに排他的にすることだ．カットを使って Prolog プログラムで case 文の機能が実現できる．例えば minimum を次のように実装すると，

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)))
(<- (minimum ?x ?y ?y) (lisp (> ?x ?y)))
```

間違った動作はしないが効率が悪い．次のようなクエリに対し，

```
(minimum 1 2 ?x)
```

Prolog は 1 番目の規則から即座に ?x = 1 を定める．人間はそこで終了とするが，プログラムは第 2 の規則からさらに解を得ようと時間を浪費する．それは 2 つの規則が互いに排他的だと知らされていないからだ．平均的に上の実装の minimum は 50% だけ余分な処理を行う．この問題は 1 番目の規則の後にカットを付けることで解決する．

```
(<- (minimum ?x ?y ?x) (lisp (<= ?x ?y)) (cut))
(<- (minimum ?x ?y ?y))
```

すると Prolog が 1 番目の規則の処理を終えたとき，次の規則を調べる前に処理がここから抜けてクエリまで移る．

埋め込み Prolog を修正してカットを扱えるようにするのはとても容易だ．prove を呼ぶ毎に，その時点での *paths* を引数として渡すようにする．プログラムがカットに出会ったときは，*paths* を引数で渡された古い値に設定すればよい．第 156，157図のコードはカットへの対応のために修正したものだ．（変更のあった行は目印のセミコロンを付けた．全ての変更がカット対応のせいではない．）

単にプログラムの効率を上げるだけのカットは green cut と呼ばれる．最も基本的なカットは green cut だった．プログラムの動作を変えるカットは red cut と呼ばれる．例えば，述語 artist を次のように定義したとしよう．

```
(<- (artist ?x) (sculptor ?x) (cut))
(<- (artist ?x) (painter ?x))
```

こうすると，彫刻家がいれば，クエリはそこで終了するようになる．彫刻家がいないときは，画家が芸術家とみなされる．

```
> (progn (<- (painter 'klee))
         (<- (painter 'soutine)))
4> (with-inference (artist ?x)
     (print ?x))
KLEE
SOUTINE
@
```

しかし彫刻家がいるときは，カットは 1 番目の規則のみで推論を止める．

```
> (<- (sculptor 'hepworth))
5
> (with-inference (artist ?x)
    (print ?x))
HEPWORTH
@
```

カットは Prolog のオペレータ fail と組み合わせて使われることがある．埋め込み Prolog では fail が同じ機能を担う．カットを規則に埋め込むと，規則が一方通行になる．そこに入り込んだ時点で，その規則のみを使うと決めたことになるのだ．規則に cut-fail の組を使うのは治安の悪い街の一方通行の路地のようなものだ．そこに入り込んだ時点で，何も得ずに出て行くと決めたことになる．典型的な例が not-equal の実装の中に見られる．

```
(<- (not-equal ?x ?x) (cut) (fail))
(<- (not-equal ?x ?y))
```

ここで最初の規則は嫌な入力に対する罠だ．(not-equal 1 1) という形の事実を示そうとしたとき，それは最初の規則の頭部にマッチし，はいそれまで，となる．それに対し，クエリ (not-equal 1 2) は最初の規則の頭部にマッチせず，2 番目の規則に進み，そこでマッチが成功する．

```
(defun rule-fn (ant con)
  (with-gensyms (val win fact binds paths)                    ;
    `(=lambda (,fact ,binds ,paths)                           ;
       (with-gensyms ,(vars-in (list ant con) #'simple?)
         (multiple-value-bind
           (,val ,win)
           (match ,fact
                  (list ',(car con)
                        ,@(mapcar #'form (cdr con)))
                  ,binds)
           (if ,win
               ,(gen-query ant val paths)                     ;
               (fail)))))))

(defmacro with-inference (query &rest body)
  (let ((vars (vars-in query #'simple?)) (gb (gensym)))
    `(with-gensyms ,vars
       (setq *paths* nil)
       (=bind (,gb) ,(gen-query (rep_ query) nil '*paths*)    ;
              (let ,(mapcar #'(lambda (v)
                                `(,v (fullbind ,v ,gb)))
                            vars)
                ,@body)
              (fail)))))

(defun gen-query (expr binds paths)                           ;
  (case (car expr)
    (and  (gen-and (cdr expr) binds paths))                   ;
    (or   (gen-or (cdr expr) binds paths))                    ;
    (not  (gen-not (cadr expr) binds paths))                  ;
    (lisp (gen-lisp (cadr expr) binds))                       ;
    (is   (gen-is (cadr expr) (third expr) binds))            ;
    (cut  `(progn (setq *paths* ,paths)                       ;
                  (=values ,binds)))                          ;
    (t    `(prove (list ',(car expr)
                        ,@(mapcar #'form (cdr expr)))
                  ,binds *paths*))))                          ;

(=defun prove (query binds paths)                             ;
  (choose-bind r *rules*
               (=funcall r query binds paths)))               ;
```

図 156　新しいオペレータへの対応.

```
> (with-inference (not-equal 'a 'a)
    (print t))
@
> (with-inference (not-equal '(a a) '(a b))
    (print t))
T
@
```

第 156，157図のコードでは算術演算，I/O，Prolog のオペレータ is の実装も行っている．第 158図には規則とクエリの完全な構文を示した.

Lisp への抜け道を作ることで算術演算（を含むあれこれ）が追加された．and や or 等のオペレータに加え，オペレータ lisp を作った．この中では任意の Lisp の式を使うことができる．Lisp の式は中の変数がクエリの生成した束縛に束縛された状態で評価される．Lisp の式の評価結果が nil のときは，オペレータ lisp の呼び出し全体が (fail) と等価になる．そうでなければ (and) と等価になる.

オペレータ lisp の利用例として，要素が昇順に並んでいるリストに対して成立する ordered の Prolog による定義

```
(defun gen-and (clauses binds paths)                              ;
  (if (null clauses)
      `(=values ,binds)
      (let ((gb (gensym)))
        `(=bind (,gb) ,(gen-query (car clauses) binds paths)   ;
           ,(gen-and (cdr clauses) gb paths)))))                  ;

(defun gen-or (clauses binds paths)                               ;
  `(choose
     ,@(mapcar #'(lambda (c) (gen-query c binds paths))        ;
               clauses)))

(defun gen-not (expr binds paths)                                 ;
  (let ((gpaths (gensym)))
    `(let ((,gpaths *paths*))
       (setq *paths* nil)
       (choose (=bind (b) ,(gen-query expr binds paths)        ;
                 (setq *paths* ,gpaths)
                 (fail))
               (progn
                 (setq *paths* ,gpaths)
                 (=values ,binds))))))

(defmacro with-binds (binds expr)
  `(let ,(mapcar #'(lambda (v) `(,v (fullbind ,v ,binds)))
                 (vars-in expr))
     ,expr))

(defun gen-lisp (expr binds)
  `(if (with-binds ,binds ,expr)
       (=values ,binds)
       (fail)))

(defun gen-is (expr1 expr2 binds)
  `(aif2 (match ,expr1 (with-binds ,binds ,expr2) ,binds)
         (=values it)
         (fail)))
```

図 157　新しいオペレータへの対応.

```
⟨規則⟩    : (<- ⟨文⟩ ⟨クエリ⟩ )
⟨クエリ⟩  : (not ⟨クエリ⟩ )
          : (and ⟨クエリ⟩* )
          : (lisp ⟨Lisp の式⟩ )
          : (is ⟨変数⟩ ⟨Lisp の式⟩ )
          : (cut)
          : (fail)
          : ⟨文⟩
⟨文⟩      : ( ⟨シンボル⟩ ⟨引数⟩* )
⟨引数⟩    : ⟨変数⟩
          : ⟨Lisp の式⟩
⟨変数⟩    : ? ⟨シンボル⟩
```

図 158　規則の新たな構文.

を考えよう．

```
(<- (ordered (?x)))
(<- (ordered (?x ?y . ?ys))
    (lisp (<= ?x ?y))
    (ordered (?y . ?ys)))
```

日本語で言うと，1 要素のリストは整列されており，2 要素以上のリストが整列されているというのは，先頭要素が第 2 要素以下で，先頭要素を除いたリストが整列されているときだ．

```
> (with-inference (ordered '(1 2 3))
    (print t))
T
@
> (with-inference (ordered '(1 3 2))
    (print t))
@
```

Lisp のオペレータを使うことで典型的な Prolog の実装が提供する他の機能も提供できる．Prolog の I/O 用述語は Lisp の I/O オペレータを Lisp の式の中で呼び出すことで代替できる．Prolog の `assert` は副作用として新たな規則を定義するが，マクロ`<-`を Lisp の式の中で呼び出すことで代替できる．

オペレータ `is` は一種の代入を提供する．パターンと Lisp の式の 2 つを引数に取り，パターンを式の返り値に対しマッチさせようとする．マッチに失敗すると，プログラムは `fail` を呼び出す．成功すれば新たな束縛の元で処理が進む．よって式 `(is ?x 1)` は `?x` を `1` に設定する効果がある．正確に言えば，`?x` は `1` だとの事実を主張している．例えば階乗の計算で `is` が必要になる．

```
(<- (factorial 0 1))
(<- (factorial ?n ?f)
    (lisp (> ?n 0))
    (is ?n1 (- ?n 1))
    (factorial ?n1 ?f1)
    (is ?f (* ?n ?f1)))
```

この定義を使うには第 1 引数が数 `n` で第 2 引数が変数のクエリを与える．

```
> (with-inference (factorial 8 ?x)
    (print ?x))
40320
@
```

オペレータ `lisp` や `is` の第 2 引数の中で使われる変数は値を返す式への束縛をすでに生成している必要がある．この制限は実際のどの Prolog 実装でも存在する．例えば次のクエリは，

```
(with-inference (factorial ?x 120)                    ; wrong
  (print ?x))
```

上のような `factorial` の定義では機能しない．オペレータ `lisp` の呼び出しが評価されるときには `?n` が未知だからだ．だから全ての Prolog プログラムが `append` のような性格を持つ訳ではない．`factorial` のように，引数の幾つかが実際の値であることを要求するものもある．

23.7 例

この節では埋め込み Prolog での Prolog プログラムの書き方を示す．第 159図の規則はクイックソートを定義している．それらの規則は `(quicksort x y)` という形の事実を含意する．ここで `x` はリスト，`y` は同じ要素が昇順にソートされたリストだ．第 2 引数に変数を使える．

```
> (with-inference (quicksort '(3 2 1) ?x)
    (print ?x))
(1 2 3)
@
```

I/O ループは埋め込み Prolog のテストになっている．オペレータ `cut`，`lisp`，`is` を利用しているからだ．ソースは第 160図に示した．それらの規則はクエリ `(echo)` を引数なしで示そうとすることで呼び出される．そのクエリが 1 番目の規則にマッチし，`?x` が `read` の返り値に束縛され，次にクエリ `(echo ?x)` を示そうとする．この新しいクエリは

```
(setq *rules* nil)

(<- (append nil ?ys ?ys))
(<- (append (?x . ?xs) ?ys (?x . ?zs))
    (append ?xs ?ys ?zs))

(<- (quicksort (?x . ?xs) ?ys)
    (partition ?xs ?x ?littles ?bigs)
    (quicksort ?littles ?ls)
    (quicksort ?bigs ?bs)
    (append ?ls (?x . ?bs) ?ys))
(<- (quicksort nil nil))

(<- (partition (?x . ?xs) ?y (?x . ?ls) ?bs)
    (lisp (<= ?x ?y))
    (partition ?xs ?y ?ls ?bs))
(<- (partition (?x . ?xs) ?y ?ls (?x . ?bs))
    (lisp (> ?x ?y))
    (partition ?xs ?y ?ls ?bs))
(<- (partition nil ?y nil nil))
```

図 159　クイックソート.

```
(<- (echo)
    (is ?x (read))
    (echo ?x))
(<- (echo 'done)
    (cut))
(<- (echo ?x)
    (lisp (prog1 t (format t "~A~%" ?x)))
    (is ?y (read))
    (cut)
    (echo ?y))
```

図 160　Prolog における I/O ループ.

2，3 番目のどちらのクエリにもマッチし得る．?x = done のときは，クエリは 2 番目の規則で終了する．そうでなければクエリは 3 番目の規則にしかマッチしないが，そちらは読み込まれた値を印字し，同じ過程を再び繰り返す．

これらが合わさると，done と入力するまで入力を印字し続けるプログラムが定義される．

```
> (with-inference (echo))
hi
HI
ho
HO
done
@
```

このようなプログラムは Prolog の抽象モデルに対し逆行しているので読み辛い．echo の理解は，Lisp による内部表現を見た方が分かり易いかもしれない．

```
(defun echo (&rest args)
  (cond ((null args) (echo (read)))
        ((eq (car args) 'done) nil)
        (t (format t "~A~%" (car args))
           (echo (read)))))
```

これは Common Lisp では普通は次のように書くものだ．

```
(defun echo (&optional (arg (read)))
  (unless (eq arg 'done)
    (format t "~A~%" arg)
```

```
    (echo)))
```

23.8 コンパイルという言葉の意味

「コンパイル」という言葉には幾つかの意味がある．最も広い意味では，コンパイルとはプログラムの何らかの抽象的な記述を低レベルのコードに変換することだ．この章で記述されたプログラムはこの意味で確かにコンパイラだ．規則を Lisp の関数に変換するからだ．

狭い意味では，コンパイルとはプログラムをマシン語に変換することだ．優れた Common Lisp の実装は関数をネイティブなマシン語にコンパイルする．gxz ページで触れたように，クロージャを生成するコードがコンパイルされると，コンパイル済みクロージャが得られる．よってここまでに説明したプログラムは狭い意味でもコンパイラだ．優れた Lisp の実装では埋め込み Prolog はマシン語に変換されるのだ．

しかしここまでに説明したプログラムはまだ Prolog コンパイラとは呼べない．プログラミング言語については「コンパイル」という言葉にさらに狭い意味がある．単にマシン語を生成するだけではその意味での定義を満たすには不足だ．あるプログラミング言語のコンパイラは変換と同様に最適化を行わなければならない．例えば Lisp コンパイラが次のような式をコンパイルするとき，

```
(+ x (+ 2 5))
```

(+ 2 5) を評価するのに実行時まで待つ理由などないが，コンパイラにはそれを実現できる賢さが要求される．プログラムはそれを 7 に置換する最適化を施され，次のようにコンパイルされることになる．

```
(+ x 7)
```

この章の埋め込み Prolog では全てのコンパイルは Lisp コンパイラによって行われた．そして Lisp コンパイラは Prolog ではなく Lisp のための最適化を狙っている．Lisp コンパイラの行う最適化に誤りはないが，低レベル過ぎる．Lisp コンパイラにはコンパイルしているコードが規則を表現するものだとは分からない．本物の Prolog コンパイラはループに変換できる規則を探すが，この章の Prolog は定数を与える式やスタックに割り当てられるクロージャを探している．

埋め込み言語によって抽象化を最高に活用できるが，それは魔法ではない．非常に抽象的な表現から高速なマシン語に至る道のりを歩き通したいならば，やはり誰かがコンピュータに方法を教えなければならない．この章では道のりのかなりの部分を驚く程少ないコードで進んで来たが，それは本物の Prolog コンパイラを書くこととは違うのだ．

24 オブジェクト指向 Lisp

この章では Lisp によるオブジェクト指向プログラミングについて論じる．Common Lisp にはオブジェクト指向のプログラムを書くためのオペレータが揃っている．それらをまとめて，Common Lisp Object System，または CLOS と呼ぶ．ここでは CLOS を単にオブジェクト指向のプログラムを書く一手段としてではなく，Lisp プログラムそのものとして捉える．CLOS をこの観点から眺めることが Lisp とオブジェクト指向プログラミングとの関係を理解する鍵となる．

24.1 Plus ça Change

オブジェクト指向プログラミングとはプログラムの構成の変化を指す．その変化はプロセッサの処理能力の分布について起こったものと似ている．1970 年，「マルチユーザのコンピュータシステム」と言えば一つか二つの大型メインフレームが多数のダム端末に接続されたものを指していた．今では多数のワークステーションがネットワークでそれぞれ接続されたものを指すことの方が多い．システムの処理能力は一つの巨大なコンピュータに集中してはおらず，個々のユーザに分散している．

オブジェクト指向プログラミングは古典的なプログラムを全く同様に分割する．不活性な大量のデータに操作を行う単一のプログラムを作るのではなく，データ自身に行うべき動作を伝えておき，プログラムはこれらの新しいデータ「オブジェクト」の相互作用によって暗黙のうちに動作する．

例えば 2 次元図形の面積を求めるプログラムを書きたいとしよう．やり方としては，一つの関数の中で，引数の種類を調べてそれに従って動作させるものがある．

```
(defun area (x)
  (cond ((rectangle-p x) (* (height x) (width x)))
        ((circle-p x) (* pi (expt (radius x) 2)))))
```

オブジェクト指向のアプローチは図形オブジェクトそれぞれに自分の面積を計算させるものだ．関数 area はばらばらに分割され，各分岐節が適当なオブジェクトのクラスに分散する．長方形のクラスのメソッド method は次のようなものになるだろうし，

```
#'(lambda (x) (* (height x) (width x)))
```

縁のクラスでは次のようになるだろう．

```
#'(lambda (x) (* pi (expt (radius x) 2)))
```

このモデルでは，オブジェクトにその面積がどれだけか尋ねると，そのオブジェクトが自分のクラスで定義されたメソッドに従って反応を返す．

CLOS の到来は Lisp がオブジェクト指向のパラダイムを取り込み始めた兆に見えるかも知れない．実際には，Lisp はオブジェクト指向のパラダイムを今も変わらず含んでいると言う方が正確だ．しかし Lisp の底を流れる原則には名前が無く，オブジェクト指向プログラミングには名前があるので，現在，Lisp がオブジェクト指向言語だと説く傾向がある．「Lisp は拡張可能な言語で，その内部でオブジェクト指向プログラミングが容易に行える」と言う方が真実に近いだろう．

CLOS は規格として定義済みなので，Lisp がオブジェクト指向言語だと宣伝するのは嘘ではない．しかしそれでは Lisp を単なるオブジェクト指向言語だと見るという制限を設けてしまっている．確かに Lisp はオブジェクト指向言語だが，それはオブジェクト指向モデルを採用しているからではない．むしろ Lisp の底を流れる抽象化技法の適用例にオブジェクト指向モデルが加わったに過ぎないと分かる．その証拠に，Lisp で書かれたプログラム CLOS は Lisp をオブジェクト指向言語に変える．

この章の狙いは，CLOS を埋め込み言語の例として考察することで Lisp とオブジェクト指向プログラミングとの結び付きを提示することだ．これは CLOS そのものを理解するにもよい方法だ．結局のところプログラミング言語の機能を一番よく説明するのはその実装の概略をおいて他にない．第 7-6 節ではマクロについてそのようにして説明した．次節では Lisp の上にオブジェクト指向の抽象化層を構築することに関して似た概略を与える．そのプログラムは第 25-3 節から第 25-6 節で CLOS の説明をするためのとっかかりとなる．

24.2 素の Lisp によるオブジェクト

Lisp は様々な種類のプログラミング言語に形を変えることができる．オブジェクト指向プログラミングの諸概念と Lisp の基本的抽象化技法との間には特に直接の対応が付けられる．CLOS は大規模なのでこの事実が霞みがちだ．そこで CLOS で何ができるかを見る前に素の Lisp で何ができるかを見ることにしよう．オブジェクト指向プログラミングに求めたいものはほとんど既に Lisp の中に用意されている．足りない分は驚く程短いコードで追加できる．この節では 2 ページ分のコードで実際のアプリケーションの多くに対して十分なオブジェクトシステムを定義する．オブジェクト指向プログラミングには，最低限で以下のことが必要だ．

1. オブジェクトには属性があり，
2. メッセージに反応し，
3. 親から属性とメソッドを継承する．

既に Lisp には属性をまとめて保持するための方法が幾つかある．一つはオブジェクトをハッシュ表として表現し，属性をその項目として表現する方法がある．そうすると個々の属性は gethash を通じて参照できる．

```
(gethash 'color obj)
```

関数もデータオブジェクトなので属性として保持できる．これはメソッドも実現できるということだ．オブジェクトのあるメソッドを呼び出すには，メソッドと同じ名前の属性を funcall する．

```
(funcall (gethash 'move obj) obj 10)
```

この考えに基づき，Smalltalk 風のメッセージ渡し構文が定義できる．

```
(defun tell (obj message &rest args)
  (apply (gethash message obj) obj args))
```

```
(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (get-ancestors obj)))

(defun get-ancestors (obj)
  (labels ((getall (x)
                   (append (list x)
                           (mapcan #'getall
                                   (gethash 'parents x)))))
    (stable-sort (delete-duplicates (getall obj))
                 #'(lambda (x y)
                     (member y (gethash 'parents x))))))

(defun some2 (fn lst)
  (if (atom lst)
      nil
      (multiple-value-bind (val win) (funcall fn (car lst))
        (if (or val win)
            (values val win)
            (some2 fn (cdr lst))))))
```

図 161　多重継承.

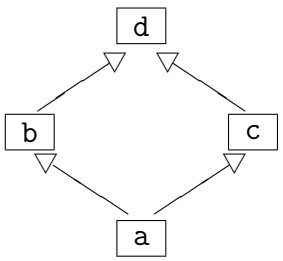

図 162　スーパークラスへの複数の経路.

オブジェクト obj に 10 だけ動くよう命じるには，こうすればよい.

```
(tell obj 'move 10)
```

実際，素の Lisp に欠けている概念は継承だけだが，初歩的な継承は 6 行のコードで実現できる．gethash の再帰版を定義すればよい.

```
(defun rget (obj prop)
  (multiple-value-bind (val win) (gethash prop obj)
        (if win
              (values val win)
              (let ((par (gethash 'parent obj)))
                (and par (rget par prop))))))
```

gethash の所に rget を使うだけで属性とメソッドの継承が実現できる．このときオブジェクトの親は次のように指定する.

```
(setf (gethash 'parent obj) obj2)
```

ここまでで実現できたのは単一継承のみで，オブジェクトはの持てる親は一つだけだ．しかし属性 parent をリストにし，rget を第 161図とすることで多重継承が実現できる.

オブジェクトの属性を取得するとき，単一継承では先祖に再帰的に遡って検索するだけでよかった．求める属性の情報がそのオブジェクト自身にないときはその親を調べ，なければさらにその親へと繰り返していく．多重継承でも同じような検索をしたいが，オブジェクトの先祖が単なるリストでなくグラフを形成し得るという事実が困難の元になる．そのグラフを単に深さ優先探索するだけでは駄目だ．複数の親が許されるとき，第 162図のような階層構造ができることがある．a の親は b と c で，それらは共に d を親に持つ．深さ優先（と言うより高さ優先）探索は a から始めて b,

```
(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (gethash 'parents obj) parents)
    (ancestors obj)
    obj))

(defun ancestors (obj)
  (or (gethash 'ancestors obj)
      (setf (gethash 'ancestors obj) (get-ancestors obj))))

(defun rget (obj prop)
  (some2 #'(lambda (a) (gethash prop a))
         (ancestors obj)))
```

図 163　オブジェクトを生成する関数.

d, c, d の順にオブジェクトを訪れる．求める属性が d と c との両方にあったら，c ではなく d に保持されている値が得られる．これではサブクラスは親の提供するデフォルト値をオーヴァライドするという原則が破られてしまっている．

普通の意味での継承を実装したければ，あるオブジェクトを決してその子孫より先に訪れてはならない．この場合，適切な検索順は a, b, c, d の順だ．どうすれば必ず子孫を先にして検索すると保証できるだろうか？ 一番単純なのは，元のオブジェクトの先祖の一覧をリストにまとめ，それを適当にソートしてどのオブジェクトもその子孫より先に現れないようにし，そうしてから各要素を順に訪れる方法だ．

この戦略は get-ancestors が使っているが，これは適切に並べ替えたオブジェクトとその先祖のリストを返す関数だ．リストをソートするとき，get-ancestors は sort でなく stable-sort を呼んでいる．これは等しい高さにある先祖たちを並べ替えてしまう可能性を排除するためだ．ソートが済みのリストがあれば，rget は求める属性を持った最初のオブジェクトを探すだけだ．（ユーティリティ some2 は，some を，gethash 等の検索の成功/失敗を第 2 返り値で示す関数と共に使うようにしたものだ.）

オブジェクトの先祖のリストは縁の近いものから遠いものへと並んでいる．orange が citrus（柑橘類）の子で，それが fruit の子のとき，リストは (orange citrus fruit) となる．

オブジェクトに複数の親があるとき，優先度は左から右の順になる．すなわち次のように書くと，

```
(setf (gethash 'parents x) (list y z))
```

継承される属性を探すとき, y は z より先に調べることになる．例えば「愛国心に溢れた (patriotic) チンピラ (scoundrel)」はあくまでもチンピラで，「チンピラっぽい愛国者 (patriot)」ではないのだとしよう．

```
> (setq scoundrel (make-hash-table)
        patriot (make-hash-table)
        patriotic-scoundrel (make-hash-table))
#<Hash-Table C4219E>
> (setf (gethash 'serves scoundrel) 'self        ; チンピラは自分自身のために生きる
        (gethash 'serves patriot) 'country       ; 愛国者は国に仕える
        (gethash 'parents patriotic-scoundrel)
          (list scoundrel patriot))
(#<Hash-Table C41C7E> #<Hash-Table C41F0E>)
> (rget patriotic-scoundrel 'serves)             ; 「愛国的チンピラ」は何に仕えるか？
SELF
T
```

この骨組み段階のシステムに改善を加える．まずオブジェクトを生成する関数から始めよう．この関数はオブジェクトが生成される時点でオブジェクトの先祖のリストを作り上げる．今はクエリが与えられた時点でリストを作るコードになっているが，それより早く作ってはいけない理由はない．第 163図では，新しいオブジェクトを作り，その内部に先祖のリストを保持させる関数 obj を定義している．先祖を内部に保持したことを活用するため，rget も再定義した．

他に改善の余地があるのはメッセージ呼び出しの構文だ．tell そのものは必然性のない邪魔者で，そのせいで動詞が 2 番目に来るようになり，これでは普通の Lisp のような前置構文として読めない．

```
(tell (tell obj 'find-owner) 'find-owner)
```

```
(defmacro defprop (name &optional meth?)
  `(progn
     (defun ,name (obj &rest args)
       ,(if meth?
            `(run-methods obj ',name args)
            `(rget obj ',name)))
     (defsetf ,name (obj) (val)
              `(setf (gethash ',',name ,obj) ,val))))

(defun run-methods (obj name args)
  (let ((meth (rget obj name)))
    (if meth
        (apply meth obj args)
        (error "No ~A method for ~A." name obj))))
```

図 164　関数風の構文.

tell を使う構文は，第 164図のように属性を同じ名前の関数として定義すれば取り除ける．オプショナル引数 meth? が真のときはその属性はメソッドとして扱う．それ以外では属性はスロットとして扱われ，rget の取ってきた値がそのまま返される．どちらの種類の属性でも，defprop に名前を与えると，

```
(defprop find-owner t)
```

その属性に関数呼び出しで参照できるようになり，再びコードが Lisp 風に読める．

```
(find-owner (find-owner obj))
```

こうして先程の例はいくらか読み易くなった．

```
> (progn
    (setq scoundrel (obj))
    (setq patriot (obj))
    (setq patriotic-scoundrel (obj scoundrel patriot))
    (defprop serves)
    (setf (serves scoundrel) 'self)
    (setf (serves patriot) 'country)
    (serves patriotic-scoundrel))
SELF
T
```

今の実装ではオブジェクトが持ち得るある名前のメソッドは高々一つだ．オブジェクトはそれ固有の，もしくは継承されたメソッドを持ち得る．ここで，固有のメソッドと継承されたメソッドを組み合わせられるような柔軟性を加えると便利だろう．例えば何らかのオブジェクトのメソッド move について，その親のメソッド move と同じものであってほしいが，その実行前や後に別のコードを実行したいことがある．

そのような可能性を許すため，プログラムを修正して before，after，around の各メソッドを含めるようにする．before メソッドというのは「でも先にこれをやっといて」というものだ．最も特定的なものから順に，他のメソッドの前座として実行される．after メソッドというのは「付け足し：これもやってよ」というものだ．最も特定的なものが最後になる順で，メソッド呼び出しのしんがりとして実行される．それらの間に，これまではメソッド本体であったものを実行する．それはこれから基本メソッドと呼ぶことにする．after メソッドが後で呼び出されるときも，基本メソッドの返り値がメソッド全体の返り値になる．

before 及び after メソッドを使うことで基本メソッドの呼び出しを新たな動作で包める．around メソッドは同じことをするが，徹底したやり方になる．around メソッドが定義されていると基本メソッドの代わりにそちらが呼び出される．そこで判断次第では around メソッド自身が基本メソッドを呼ぶこともあり得る（第 166図の call-next を経由する）．

補助メソッドの実現のため，run-methods と rget を第 165図のように修正した．これまで，オブジェクトのメソッドを実行したときはただ一つの関数，最も特定的な基本メソッドを実行しただけだった．つまり先祖のリストを辿って最初に見つけたメソッドを実行していた．補助メソッドを考慮すると呼び出し手順は次のようになる．

1. 最も特定的な around メソッド（どこかで定義されていれば）.

2. なければ，次の順に従う．
   （a）全ての before メソッドを，最も特定的なものから順に．
   （b）最も特定的な基本メソッド（今まではこれだけを呼び出していた）．
   （c）全ての after メソッドを，最も特定的でないものから順に．

メソッドが，単一の関数でなく 4 つの部分に分かれた構造を持つことにも注意しよう．（基本）メソッドを定義するには，

```
(setf (gethash 'move obj) #'(lambda ...))
```

とするのではなく，次のようにする．

```
(setf (meth-primary (gethash 'move obj)) #'(lambda ...))
```

こういった理由から，次の目標はメソッドを定義するマクロの定義だ．

第 165図にはそういうマクロの定義を示した．コードの大半はメソッドが他のメソッドの参照のために使う 2 つの関数の実装に充てられている．around メソッドと基本メソッドでは call-next により「次の」メソッドを呼び出せる．すなわちそのとき実行中のメソッドが存在しなかったときに実行されていたはずのコードだ．例えば実行中のメソッドが唯一の around メソッドだったら，「次の」メソッドは before メソッド，最も特定的な基本メソッド，after メソッドのサンドイッチだ．最も特定的な基本メソッドの中では「次の」メソッドは 2 番目に特定的な基本メソッドだろう．call-next の動作は呼ばれた場所によって異なるため，それを defun でグローバルに定義することは決してなく，defmeth で定義された各メソッドの内部でローカルに定義される．

around メソッドや基本メソッドは next-p により「次の」メソッドが存在するかどうかを調べられる．例えば親のないオブジェクトの基本メソッドを実行しているなら「次の」メソッドは存在しない．次のメソッドがないときは call-next はエラーになるので，普通は先に next-p を呼んで調べておく必要がある．call-next と同様，next-p は各メソッド内部でローカルに定義される．

新しいマクロ defmeth は次のように使う．オブジェクト rectangle（矩形）にメソッド area（面積）を定義したいときは次のようにする．

```
(setq rectangle (obj))
(defprop height)
(defprop width)
(defmeth (area) rectangle (r)
  (* (height r) (width r)))
```

するとインスタンスの面積は属するクラスのメソッドに従って算出される．

```
> (let ((myrec (obj rectangle)))
       (setf (height myrec) 2
             (width myrec) 3)
       (area myrec))
6
```

さらに複雑な例として，オブジェクト filesystem にメソッド backup を定義したとしよう．

```
(setq filesystem (obj))
(defmeth (backup :before) filesystem (fs)
  (format t "Remember to mount the tape.~%"))
(defmeth (backup) filesystem (fs)
  (format t "Oops, deleted all your files.~%")
  'done)
(defmeth (backup :after) filesystem (fs)
  (format t "Well, that was easy.~%"))
```

普通，呼び出しは次のような順に従う．

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
DONE
```

次にバックアップにかかる時間を計りたくなり，次のような around メソッドを定義した．

```
(defstruct meth around before primary after)

(defmacro meth- (field obj)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (and (meth-p ,gobj)
            (,(symb 'meth- field) ,gobj)))))

(defun run-methods (obj name args)
  (let ((pri (rget obj name :primary)))
    (if pri
        (let ((ar (rget obj name :around)))
          (if ar
              (apply ar obj args)
              (run-core-methods obj name args pri)))
        (error "No primary ~A method for ~A." name obj))))

(defun run-core-methods (obj name args &optional pri)
  (multiple-value-prog1
    (progn (run-befores obj name args)
           (apply (or pri (rget obj name :primary))
                  obj args))
    (run-afters obj name args)))

(defun rget (obj prop &optional meth (skip 0))
  (some2 #'(lambda (a)
             (multiple-value-bind (val win) (gethash prop a)
               (if win
                   (case meth (:around (meth- around val))
                     (:primary (meth- primary val))
                     (t (values val win))))))
         (nthcdr skip (ancestors obj))))

(defun run-befores (obj prop args)
  (dolist (a (ancestors obj))
    (let ((bm (meth- before (gethash prop a))))
      (if bm (apply bm obj args)))))

(defun run-afters (obj prop args)
  (labels ((rec (lst)
             (when lst
               (rec (cdr lst))
               (let ((am (meth- after
                                (gethash prop (car lst)))))
                 (if am (apply am (car lst) args))))))
    (rec (ancestors obj))))
```

図 165　補助メソッド.

```
(defmeth (backup :around) filesystem (fs)
  (time (call-next)))
```

すると filesystem の子に対して backup が呼ばれる度に，（さらに特定的な around メソッドが間に入らない限り）この around メソッドが呼ばれる．これが実行するコードは backup が普通に呼び出されたときに実行するコードと同じだが，それを time の呼び出しの内部で実行する．time の返り値が backup の返り値になる．

```
> (backup (obj filesystem))
Remember to mount the tape.
Oops, deleted all your files.
Well, that was easy.
Elapsed Time = .01 seconds
```

```
(defmacro defmeth ((name &optional (type :primary))
                   obj parms &body body)
  (let ((gobj (gensym)))
    `(let ((,gobj ,obj))
       (defprop ,name t)
       (unless (meth-p (gethash ',name ,gobj))
         (setf (gethash ',name ,gobj) (make-meth)))
       (setf (,(symb 'meth- type) (gethash ',name ,gobj))
             ,(build-meth name type gobj parms body)))))

(defun build-meth (name type gobj parms body)
  (let ((gargs (gensym)))
    `#'(lambda (&rest ,gargs)
         (labels
           ((call-next ()
                       ,(if (or (eq type :primary)
                                (eq type :around))
                            `(cnm ,gobj ',name (cdr ,gargs) ,type)
                            '(error "Illegal call-next.")))
            (next-p ()
                    ,(case type
                       (:around
                         `(or (rget ,gobj ',name :around 1)
                              (rget ,gobj ',name :primary)))
                       (:primary
                         `(rget ,gobj ',name :primary 1))
                       (t nil))))
           (apply #'(lambda ,parms ,@body) ,gargs)))))

(defun cnm (obj name args type)
  (case type
    (:around (let ((ar (rget obj name :around 1)))
               (if ar
                   (apply ar obj args)
                   (run-core-methods obj name args))))
    (:primary (let ((pri (rget obj name :primary 1)))
                (if pri
                    (apply pri obj args)
                    (error "No next method."))))))
```

図 166　メソッドの定義.

```
(defmacro undefmeth ((name &optional (type :primary)) obj)
  `(setf (,(symb 'meth- type) (gethash ',name ,obj))
         nil))
```

図 167　メソッドの削除.

DONE

ひとたび時間計測が済んでしまえば around メソッドは取り除きたい．それには undefmeth（第 167図）を呼ぶ．これには defmeth の第 1，第 2 引数と同じものを渡す.

```
(undefmeth (backup :around) filesystem)
```

他には，オブジェクトの先祖リストを変更したいときがある．しかしその変更後にはそのオブジェクトの全ての子について先祖リストを更新しなければならない．これまではオブジェクトからその子を得る方法を用意してなかったので，属性 children を追加しなければならない.

第 168図にはオブジェクトの親と子を操作するためのコードを示した．親と子を求めるには gethash でなくオペレータ parents と children を使う．後者はマクロなので setf に対して透過的だ．前者は関数で，defsetf でその

```
(defmacro children (obj)
  `(gethash 'children ,obj))

(defun parents (obj)
  (gethash 'parents obj))

(defun set-parents (obj pars)
  (dolist (p (parents obj))
    (setf (children p)
          (delete obj (children p))))
  (setf (gethash 'parents obj) pars)
  (dolist (p pars)
    (pushnew obj (children p)))
  (maphier #'(lambda (obj)
               (setf (gethash 'ancestors obj)
                     (get-ancestors obj)))
           obj)
  pars)

(defsetf parents set-parents)

(defun maphier (fn obj)
  (funcall fn obj)
  (dolist (c (children obj))
    (maphier fn c)))

(defun obj (&rest parents)
  (let ((obj (make-hash-table)))
    (setf (parents obj) parents)
    obj))
```

図 168　親と子へのリンクの管理.

インヴァージョンは set-parents だと定義する．それが 2 重リンクの張り巡らされた世界で一貫性を保つために必要な手続きを全て行ってくれる．

部分ツリー内の全てのオブジェクトの祖先を更新するため，set-parents は maphier を呼んでいる．これは継承の階層構造に対する mapc のようなものだ．mapc がリスト内の全ての要素について関数を呼ぶように，maphier はオブジェクトとその子孫全てについて関数を呼ぶ．それらが形成するのがツリーでない（グラフになっている）限り，関数が同じオブジェクトについて複数回呼ばれ得る．しかし get-ancestors は複数回呼ばれても全く同じ動作をするので，ここではそれは問題にならない．

こうしてオブジェクトの属性 parents に setf を使うだけで継承階層を変更できるようになった．

```
> (progn (pop (parents patriotic-scoundrel))
         (serves patriotic-scoundrel))
COUNTRY
T
```

階層構造に変更を加えると，影響のある子と先祖のリストが自動的に更新される．（子を直接操作するつもりはないが，set-parents に似た set-children を定義するとそうなるかも知れない）第 168図の最後には新しいコードを使うよう再定義した obj も示した．

最後の改善としては，メソッド結合の新たな方法を指定できるようにする．現在は最も特定的な基本メソッドだけが呼び出される（その中で call-next を使って他を呼び出せるけれど）．そうではなく，オブジェクトの先祖各々の基本メソッドを組み合わせられるようにしたい．例えば my-orange が orange の子で，それは citrus（柑橘類）の子だとしよう．メソッド props が，citrus では (round acidic) を，orange では (orange sweet) を，my-orange では (dented) を返すとき，(props my-orange) でそれらの値全ての合併 (dented orange sweet round acidic) を得られると便利だ．

```
(defmacro defcomb (name op)
  `(progn
     (defprop ,name t)
     (setf (get ',name 'mcombine)
           ,(case op
              (:standard nil)
              (:progn '#'(lambda (&rest args)
                           (car (last args))))
              (t op)))))

(defun run-core-methods (obj name args &optional pri)
  (let ((comb (get name 'mcombine)))
    (if comb
        (if (symbolp comb)
            (funcall (case comb (:and #'comb-and)
                       (:or #'comb-or))
                     obj name args (ancestors obj))
            (comb-normal comb obj name args))
        (multiple-value-prog1
          (progn (run-befores obj name args)
                 (apply (or pri (rget obj name :primary))
                        obj args))
          (run-afters obj name args)))))

(defun comb-normal (comb obj name args)
  (apply comb
         (mapcan #'(lambda (a)
                     (let* ((pm (meth- primary
                                       (gethash name a)))
                            (val (if pm
                                     (apply pm obj args))))
                       (if val (list val))))
                 (ancestors obj))))

(defun comb-and (obj name args ancs &optional (last t))
  (if (null ancs)
      last
      (let ((pm (meth- primary (gethash name (car ancs)))))
        (if pm
            (let ((new (apply pm obj args)))
              (and new
                   (comb-and obj name args (cdr ancs) new)))
            (comb-and obj name args (cdr ancs) last)))))

(defun comb-or (obj name args ancs)
  (and ancs
       (let ((pm (meth- primary (gethash name (car ancs)))))
         (or (and pm (apply pm obj args))
             (comb-or obj name args (cdr ancs))))))
```

図 169　メソッドの組み合わせ.

そうするには，メソッドが，最も特定的な基本メソッドの返り値をただ返すのではなく，基本メソッド全ての返り値について何らかの関数を適用できるようにすればよい．第 169図には，メソッド結合の方法を定義できるようにするマクロと，メソッド結合を行う新しい run-core-methods を示した．

あるメソッドについてのメソッド結合は defcomb で定義するが，これはメソッド名と結合方法を指定するための第 2 引数を取る．普通はこの第 2 引数には関数を使うが，:progn, :and, :or, :standard のいずれかを与えてもよい．最初の 3 つでは基本メソッドは対応するオペレータと同様に結合されるが，:standard では基本的なメソッド結合のみが行われる．

第 169図の中核は新しい run-core-methods だ．呼ばれたメソッドに属性 mcombine がなければ，メソッド呼び出しはこれまで通りに行われる．メソッドに mcombine があるとき，その値は関数（+ 等）またはキーワード（:or）だ．前者の場合，その関数は全ての基本メソッドの返り値から成るリストにそのまま適用される[*38]．後者の場合，キーワードに関連付けられた関数で基本メソッドの返り値について反復を行う．

オペレータ and と or は第 169図のように特別扱いする必要がある．それは単にそれらが特殊式だからというのではなく，短絡評価を行うせいだ．

```
> (or 1 (princ "wahoo"))
1
```

ここでは何も印字されない．or は非 nil の引数を見つけた時点で制御を返すからだ．同様に，or 結合の対象となる基本メソッドは，それより特定的なメソッドが真値を返すときには決して呼ばれてはならない．and と or でそのような短絡評価を実現するため，関数 comb-and と comb-or を別個に用意した．

上記の例を実現するには，次のようにする．

```
(setq citrus (obj))
(setq orange (obj citrus))

(setq my-orange (obj orange))

(defmeth (props) citrus (c) '(round acidic))
(defmeth (props) orange (o) '(orange sweet))
(defmeth (props) my-orange (m) '(dented))

(defcomb props #'(lambda (&rest args) (reduce #'union args)))
```

こうすると props は基本メソッド全ての返り値の合併を返す[*39]．

```
> (props my-orange)
(DENTED ORANGE SWEET ROUND ACIDIC)
```

ついでに言うと，この例は Lisp でオブジェクト指向プログラミングを行うときのみに生まれる選択肢を示唆している．情報をスロットとメソッドのどちらに格納するか，ということだ．

その後でメソッド props に普通の動作に戻って欲しくなったら，ただメソッド結合を標準に戻せばよい．

```
> (defcomb props :standard)
NIL
> (props my-orange)
(DENTED)
```

before 及び after メソッドは標準メソッド結合でのみ実行されることに注意しよう．しかし around メソッドはこれまで同様に機能する．

この節で示したプログラムが意図しているのはモデルで，オブジェクト指向プログラミングの本物の基盤ではない．効率ではなく簡潔さを求めて書かれている．そうは言っても実際に動作するモデルであり，実験やプロトタイプに使えるものだ．このプログラムをそういう目的に使うつもりが本当にあるなら，ちょっとした変更でずっと効率が上がる．親が一つだけのオブジェクトで先祖リストを保持または算出しないことだ．

[*38] このコードを改良するならコンシングを避けるためにここで reduce を使ってもよい．

[*39] props の使うメソッド結合用関数は union を呼ぶので，リストの要素は必ずしもこの順にはならない．

24.3 クラスとインスタンス

前節のプログラムは，あの程度の小さなプログラムでできる限り CLOS に似せて書かれた．それを理解することで CLOS の理解はすでにかなり進んだことになる．以降の数節では CLOS 自体を取り上げる．

我々の「スケッチ」ではクラスとインスタンス，またはスロットとメソッドについて構文上の区別を一切していなかった．CLOS ではクラスの定義にはマクロ defclass を使い，スロットも同時に宣言する．

```
(defclass circle ()
  (radius center))
```

この式は，クラス circle はスーパークラスを持たず，2 個のスロット radius と center を持つことを示している．クラス circle のインスタンスを作るには次のようにする．

```
(make-instance 'circle)
```

circle のインスタンスのスロットを参照する方法を定義してないので，寂しいことに，これから作られるインスタンスはどれもかなり不活性なものだ．スロットを参照するにはアクセサ関数を定義する．

```
(defclass circle ()
  ((radius :accessor circle-radius)
   (center :accessor circle-center)))
```

すると circle のインスタンスを作ったとき，そのインスタンスのスロット radius と center は対応するアクセサ関数に setf を使うことで値を設定できる．

```
> (setf (circle-radius (make-instance 'circle)) 2)
2
```

このような初期化は make-instance を呼ぶと同時に行うこともできる．それができるようにスロットを定義すればよい．

```
(defclass circle ()
  ((radius :accessor circle-radius :initarg :radius)
   (center :accessor circle-center :initarg :center)))
```

スロット定義内のキーワード:initarg はその次の引数が make-instance のキーワード引数になることを指定する．キーワード引数の値がスロットの初期値になる．

```
> (circle-radius (make-instance 'circle
                                :radius 2
                                :center '(0 . 0)))
2
```

:initform を定義することでスロットが自分自身を初期化するように定義できる．クラス shape のスロット visible は，

```
(defclass shape ()
  ((color   :accessor shape-color   :initarg :color)
   (visible :accessor shape-visible :initarg :visible
            :initform t)))
```

既定値 t を持つ．

```
> (shape-visible (make-instance 'shape))
T
```

スロットに initarg と initform が両方あるとき，initarg を指定するとそちらが優先する．

```
> (shape-visible (make-instance 'shape :visible nil))
NIL
```

スロットはインスタンスとサブクラスに継承される．あるクラスが複数のスーパークラスを持つとき，それらのスロットの合併が継承される．クラス screen-circle を circle と shape 両方のサブクラスとして定義すると，

```
(defclass screen-circle (circle shape)
  nil)
```

screen-circle のインスタンスは，スロットを 2 つずつ継承して 4 つ持つことになる．クラスは必ずしも新しく独自のスロットを作る必要はないことに注意しよう．クラス screen-circle は circle と shape の両方から性質を継承したインスタンスを提供するためだけに存在している．

アクセサと initargs が screen-circle のインスタンスに及ぼす影響は，circle や shape のインスタンスに及ぼす影響と全く同じだ．

```
> (shape-color (make-instance 'screen-circle
                              :color 'red :radius 3))
RED
```

defclass のスロット color に initform を指定することで screen-circle の各インスタンスに色の初期値を持たせることができる．

```
(defclass screen-circle (circle shape)
  ((color :initform 'purple)))
```

これで screen-circle のインスタンスは初めから紫色になる．

```
> (shape-color (make-instance 'screen-circle))
PURPLE
```

ただし initarg の:color で陽に指定することでスロットを初期化することも依然として可能だ．

我々の作ったオブジェクト指向プログラミングの「スケッチ」ではインスタンスは値を親クラスのスロットから直接継承していた．CLOS ではインスタンスの持つスロットはクラスのそれとは意味が異なる（訳注：ここではクラスも一種のオブジェクトだというのが暗黙の前提）．親クラスで initform を定義することで，継承された初期値をインスタンスに定義する．ある意味ではこちらの方が柔軟だ．initform には，定数の他に，評価される毎に違う値を返す式を使えるからだ．

```
(defclass random-dot ()
  ((x :accessor dot-x :initform (random 100))
   (y :accessor dot-y :initform (random 100))))
```

random-dot のインスタンスを作る度に x 及び y 座標は 0 から 99 のランダムな整数で初期化される．

```
> (mapcar #'(lambda (name)
              (let ((rd (make-instance 'random-dot)))
                (list name (dot-x rd) (dot-y rd))))
          '(first second third))
((FIRST 25 8) (SECOND 26 15) (THIRD 75 59))
```

我々の「スケッチ」では，値がインスタンス毎に異なるスロットとクラスの全てのインスタンスで等しい値を持つスロットの区別もしていなかった（訳注：C++ 等で言う static メンバのこと）．CLOS ではこれこれのスロットが共有されると指定することができる．つまりその値は全てのインスタンスで等しくなる．そうするにはスロットの宣言に :allocation :class を含める．（対になるのはスロットに:allocation :instance を指定することだが，そちらが既定値なので明示的に書く理由はない．）例えば全てのフクロウ (owl) が夜行性 (nocturnal) ならば，クラス owl のスロット nocturnal を共有スロットとして初期値 t を与える．

```
(defclass owl ()
  ((nocturnal :accessor owl-nocturnal
              :initform t
              :allocation :class)))
```

こうすればクラス owl の全てのインスタンスがこのスロットを継承する．

```
> (owl-nocturnal (make-instance 'owl))
T
```

あるインスタンスのこのスロットの「ローカルな」値を変更するとき，実際にはクラスの保持する値を変更している．

```
> (setf (owl-nocturnal (make-instance 'owl)) 'maybe)
MAYBE
> (owl-nocturnal (make-instance 'owl))
MAYBE
```

これは混乱の元になり得るので，そのようなスロットは読み取り専用にしたいことがある．スロットに対するアクセサ関数を定義するとき，スロットの値を読み取る方法と変更する方法の両方を作っていることになる．値を読み取れるが変更不可にしたいとき，スロットには両機能を持つアクセサ関数でなくリーダ関数のみを定義すればよい．

```
(defclass owl ()
  ((nocturnal :reader owl-nocturnal
              :initform t
              :allocation :class)))
```

するとインスタンスのスロット nocturnal を変更しようとするとエラーになる．

```
> (setf (owl-nocturnal (make-instance 'owl)) nil)
>>Error: The function (SETF OWL-NOCTURNAL) is undefined.
```

24.4 メソッド

我々の「スケッチ」はレキシカル・クロージャを提供するプログラミング言語でのスロットとメソッドとの類似性を強調している．我々のプログラムでは基本メソッドはスロットの値と同様に保持され，継承されていた．スロットとメソッドの唯一の違いは，次のようにしてスロットをある名前で定義すると，

```
(defprop area)
```

単に値を求めて返す関数 area が作られることだ．しかし次のようにしてそれをメソッドとして定義すると，

```
(defprop area t)
```

値を求めた後にそれを引数に対して funcall する関数 area が作られる．

CLOS でも関数の役割をするものはやはりメソッドと呼ばれており，それら各々があるクラスの属性のように見えるように定義できる．ここではクラス circle にメソッド area を定義する．

```
(defmethod area ((c circle))
  (* pi (expt (circle-radius c) 2)))
```

このメソッドの仮引数リストによれば，これはクラス circle のインスタンスに適用される 1 引数関数だと分かる．

このメソッドは関数のように呼び出せる．我々の「スケッチ」と全く同様だ．

```
> (area (make-instance 'circle :radius 1))
3.14...
```

さらに引数を取るメソッドも定義できる．

```
(defmethod move ((c circle) dx dy)
  (incf (car (circle-center c)) dx)
  (incf (cdr (circle-center c)) dy)
  (circle-center c))
```

このメソッドを circle のインスタンスに対して呼び出すと，その中心が dx, dy だけ移動する．

```
> (move (make-instance 'circle :center '(1 . 1)) 2 3)
(3 . 4)
```

このメソッドの返り値は circle オブジェクトの新たな位置を反映している．

我々の「スケッチ」と同様に，あるインスタンスのクラスに対してメソッドが定義されており，そのクラスのスーパークラスにも定義されていたら，最も特定的なものが実行される．よってもし unit-circle が circle のサブクラスで，次のように定義されたメソッド area を持っていたら，

```
(defmethod area ((c unit-circle)) pi)
```

unit-circle のインスタンスに対して area を呼ぶと，より一般的なメソッドではなくこのメソッドが実行される．

クラスに複数のスーパークラスが存在するとき，それらの優先度は左から右の順だ．クラス patriotic-scoundrel（愛国心溢れるチンピラ）を次のように定義すると，

```
(defclass scoundrel nil nil)
(defclass patriot nil nil)
(defclass patriotic-scoundrel (scoundrel patriot) nil)
```

それは第一にチンピラなのであって，チンピラっぽい愛国者ではないと定めたことになる．複数のスーパークラスに適用可能なメソッドが存在するとき，

```
(defmethod self-or-country? ((s scoundrel))
  'self)

(defmethod self-or-country? ((p patriot))
  'country)
```

クラス scoundrel（チンピラ）のメソッドが実行される．

```
> (self-or-country? (make-instance 'patriotic-scoundrel))
SELF
```

ここまでの例は CLOS のメソッドがあるオブジェクトの持つメソッドだと言う幻想を壊すことはなかった．しかし実はそれらにはさらに一般性がある．メソッド move の引数リストで，(c circle) という部分は特定化された仮引数と呼ばれる．このメソッドは第 1 引数がクラス circle のインスタンスのときに適用されることを示す．CLOS のメソッドでは複数の仮引数で特定化ができる．次のメソッドは特定化された仮引数を 2 つ，オプショナルな特定化されていない仮引数を 1 つ持つ．

```
(defmethod combine ((ic ice-cream) (top topping)
                    &optional (where :here))
  (append (list (name ic) 'ice-cream)
          (list 'with (name top) 'topping)
          (list 'in 'a
                (case where
                  (:here 'glass)
                  (:to-go 'styrofoam))
                'dish)))
```

これは最初の 2 つの引数がそれぞれ ice-cream と topping のインスタンスのときに呼び出される．インスタンスを作るために必要な最小限のクラスを定義しておけば，

```
(defclass stuff () ((name :accessor name :initarg :name)))
(defclass ice-cream (stuff) nil)
(defclass topping (stuff) nil)
```

このメソッドを定義し，実行してみることができる．

```
> (combine (make-instance 'ice-cream :name 'fig)
           (make-instance 'topping :name 'olive)
           :here)
(FIG ICE-CREAM WITH OLIVE TOPPING IN A GLASS DISH)
```

メソッドが複数の引数を特定化しているとき，それをあるクラスの属性とみなし続けることは無理がある．上のメソッド combine は，クラス ice-cream に属しているのだろうか，それともクラス topping だろうか？ CLOS ではメッセージに反応するオブジェクトというモデルは呆気なく消え去ってしまう．そのモデルは，メソッド呼び出しを次のようにする限りは自然に思える．

```
(tell obj 'move 2 3)
```

ここでは明らかに obj のメソッド move を呼び出している．しかしこの構文を捨てて関数風にしてしまうと，

```
(move obj 2 3)
```

第 1 引数についてディスパッチを行うように move を定義しなければならない．すなわち第 1 引数の種類を見て適切なメソッドを呼び出すようにする．

この段階に至ると次の疑問が頭をもたげる．なぜディスパッチが行えるのは第 1 引数のみなのだろう？ CLOS の答え：さあ，そんな理由あるんでしょうか．CLOS ではメソッドは任意個の仮引数を特定化できる．そしてそれはユーザ定義のクラスのみではなく，Common Lisp 組込みの型でも[*40]，それどころか個々のオブジェクトに対しても可能だ．次には文字列に適用されるメソッド combine を示す．

```
(defmethod combine ((s1 string) (s2 string) &optional int?)
  (let ((str (concatenate 'string s1 s2)))
    (if int? (intern str) str)))
```

これは，もうメソッドはクラスの属性ではないというだけでなく，クラスを一切定義せずともメソッドを使えるということでもある．

```
> (combine "I am not a " "cook.")
"I am not a cook."
```

次では第 2 仮引数がシンボル palindrome に対して特定化されている．

[*40] 正確に言えば，「Common Lisp の型階層と対応するように CLOS で定義済みのクラスに対しても」ということだ．

```
(defmethod combine ((s1 sequence) (x (eql 'palindrome))
                                  &optional (length :odd))
  (concatenate (type-of s1)
               s1
               (subseq (reverse s1)
                       (case length (:odd 1) (:even 0)))))
```

このメソッドは任意のシークェンスの要素を回文 (palindrome) にする.[*41]

```
> (combine '(able was i ere) 'palindrome)
(ABLE WAS I ERE I WAS ABLE)
```

この時点で，オブジェクト指向プログラミングではなく，より一般的なものを我々は手にしている．メソッドの下にはディスパッチという概念があり，それは複数の引数に対して行うことができ，それが基づくのは引数のクラスだけに留まらないという理解の下で CLOS は作られた．メソッドがこの一般的な概念の上に構築されたとき，個々のクラスから独立したものとなる．メソッドは概念的にクラスに密着するのでなく，同じ名前の他のメソッドに密着するようになる．CLOS ではそのようなメソッドの集まりをジェネリック関数と呼ぶ．ここまで定義してきた複数のメソッド combine は暗黙のうちにジェネリック関数 combine を定義していたのだ.

ジェネリック関数はマクロ defgeneric で明示的に定義することもできる．ジェネリック関数を定義するためには defgeneric を呼び出すことは必須ではない．しかしドキュメントやエラー対策の安全ネットを入れるのに都合のよい場所だ．ここでは両方を行ってみた.

```
(defgeneric combine (x y &optional z)
  (:method (x y &optional z)
    "I can't combine these arguments.")
  (:documentation "Combines things."))
```

combine 内で定義されたメソッドはどの引数も特定化していないので，他に適用可能なメソッドがない場合に呼ばれる.

```
> (combine #'expt "chocolate")
"I can't combine these arguments."
```

こうする前にはこのような呼び出しはエラーになっていたところだ.

ジェネリック関数は，メソッドがオブジェクトの属性だったときにはなかった制限を一つ課す．名前が同じ全てのメソッドが一つのジェネリック関数にまとめられるとき，引数リストが一致しなければならない．メソッド combine のいずれにもオプショナル引数が伴っていたのはそのせいだ．引数を 3 個まで取る最初のメソッド combine を定義した後は，引数を 2 個しか取らない combine を定義しようとするとエラーになる.

CLOS は名前が同じ全てのメソッドの引数リストが合同でなければならないと要求する．引数リストが合同なのは，同数の必須引数とオプショナル引数を持ち，さらに &rest と &key の用法に互換性がある場合だ．異なるメソッドが受け付ける実際のキーワード引数は必ずしも同じでなくともよいが，defgeneric によって全てのメソッドが最低限受け付けるキーワード引数の集合を定めることもできる．以下の引数リストの対はいずれも合同だ.

```
(x)              (a)
(x &optional y)  (a &optional b)
(x y &rest z)    (a b &rest c)
(x y &rest z)    (a b &key c d)
```

また，以下の対はいずれも合同でない.

```
(x)              (a b)
(x &optional y)  (a &optional b c)
(x &optional y)  (a &rest b)
(x &key x y)     (a)
```

メソッドの再定義は関数の再定義と全く同様だ．必須引数のみが特定化できるので，各メソッドはそれが属するジェネリック関数と必須引数の型により一意に同定される．同じ特定化方法を持つ別のメソッドを定義すると元のメソッドが上書きされる．よって次のようにすると，

```
(defmethod combine ((x string) (y string)
                    &optional ignore)
  (concatenate 'string x "+"y))
```

[*41] ある（このことさえなければ）優れた Common Lisp の実装では concatenate がコンスを第 1 引数に受け付けず，上の例は機能しない.

```
(defmacro undefmethod (name &rest args)
  (if (consp (car args))
      (udm name nil (car args))
      (udm name (list (car args)) (cadr args))))

(defun udm (name qual specs)
  (let ((classes (mapcar #'(lambda (s)
                             `(find-class ',s))
                         specs)))
    `(remove-method (symbol-function ',name)
                    (find-method (symbol-function ',name)
                                 ',qual
                                 (list ,@classes)))))
```

図 170　メソッドを削除するマクロ.

第 1，第 2 引数が文字列の場合の combine の動作を再定義したことになる.

メソッドを再定義するのでなく削除したいとき，残念なことに defmethod の逆の働きのオペレータは組込みで用意されていない．幸運なことにこれは Lisp なので，自分で書くことができる．手動でメソッドを削除する方法の詳細は第 170図の undefmethod の実装に要約されている．このマクロを使うには defmethod に渡すものと似た引数を渡す．ただし第 2 または第 3 引数に引数リスト全体を渡すのでなく，必須引数のクラス名のみを渡す．よって 2 つの文字列に対するメソッド combine を削除するには次のようにする.

```
(undefmethod combine (string string))
```

特定化されていない引数は暗黙のうちにクラス t となるので，必須だが特定化されていない仮引数を持つメソッドを定義していたら,

```
(defmethod combine ((fn function) x &optional y)
  (funcall fn x y))
```

次のようにすることでそれを削除できる.

```
(undefmethod combine (function t))
```

ジェネリック関数全体を削除したいときは，どのような関数の定義を削除するときとも同じように fmakunbound を呼ぶ.

```
(fmakunbound 'combine)
```

24.5　補助メソッドとメソッド結合

我々の「スケッチ」では補助メソッドは大体 CLOS と同様に機能していた．これまで基本メソッドだけを見てきたが，before，after，around の各メソッドも使える．そのような補助メソッドは，defmethod の呼び出しでメソッド名の後に限定キーワードを付けて定義する．基本メソッド speak をクラス speaker に対して次のように定義したとき,

```
(defclass speaker nil nil)

(defmethod speak ((s speaker) string)
  (format t "~A" string))
```

speak を speaker のインスタンスに対して呼び出すと，第 2 引数が単に印字されるだけだ.

```
> (speak (make-instance 'speaker)
         "life is not what it used to be")
life is not what it used to be
NIL
```

before 及び after メソッドで基本メソッド speak を包むようなサブクラス intellectual（知的階層の人）を定義すると,

```
(defclass intellectual (speaker) nil)

(defmethod speak :before ((i intellectual) string)
  (princ "Perhaps "))

(defmethod speak :after ((i intellectual) string)
  (princ " in some sense"))
```

必ず最初と最後に言葉を付け加えるような speaker のサブクラスが実現できる．

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps life is not what it used to be in some sense
NIL
```

標準メソッド結合ではメソッドは我々の「スケッチ」で説明したのと同じように呼ばれる．before メソッドが最も特定的なものから順に呼ばれ，最も特定的な基本メソッドが呼ばれ，after メソッドが最も特定的でないものから順に呼ばれる．よってスーパークラスの speaker に before または after メソッドを定義すると，

```
(defmethod speak :before ((s speaker) string)
  (princ "I think "))
```

それらはサンドイッチの間で呼ばれる．

```
> (speak (make-instance 'intellectual)
         "life is not what it used to be")
Perhaps I think life is not what it used to be in some sense
NIL
```

どの before または after メソッドが呼ばれるかに関わらず，ジェネリック関数の返り値は最も特定的な基本メソッドの返り値だ．この場合では format の返した nil になる．

around メソッドがある場合には話が変わる．オブジェクトの先祖ツリーのうちのあるクラスに around メソッドがあるとき——正確に言えばジェネリック関数の引数に対し特定化された around メソッドがあるとき，around メソッドが最初に呼ばれ，メソッドの残りの部分は around メソッドで実行すると判断したときのみ実行される．我々の「スケッチ」と同様，around または基本メソッドは関数によって次のメソッドを呼び出せる．「スケッチ」での call-next に当たるものは CLOS では call-next-method と呼ばれる．さらに next-p と同様の next-method-p もある．around メソッドを使うとさらにもって回った言い方をする speaker のサブクラスが作れる．

```
(defclass courtier (speaker) nil)

(defmethod speak :around ((c courtier) string)
  (format t "Does the King believe that ~A? " string)
  (if (eq (read) 'yes)
      (if (next-method-p) (call-next-method))
      (format t "Indeed, it is a preposterous idea.~%"))
  'bow)
```

speak の第 1 引数がクラス courtier（廷臣）のインスタンスのとき，その発言は around メソッドが担当する．

```
> (speak (make-instance 'courtier) "kings will last")
Does the King believe that kings will last? yes          ; 王は王制が続くとお信じだろうか
I think kings will last                                  ; 私は王制は続くと思います
BOW
> (speak (make-instance 'courtier) "the world is round")
Does the King believe that the world is round? no        ; 王は地球が丸いとお信じだろうか
Indeed, it is a preposterous idea.                       ; 実際，馬鹿げた考えですな
BOW
```

before または after メソッドと異なり，around メソッドの返り値がジェネリック関数の返り値になる．

一般にメソッドは次のように実行される（第 25-2 節でも示した）．

1. 最も特定的な around メソッド（どこかで定義されていれば）．
2. なければ，次の順に従う．
   （a）全ての before メソッドを，最も特定的なものから順に．

（b）最も特定的な基本メソッド．
（c）全ての after メソッドを，最も特定的でないものから順に．

このようにメソッドを組み合わせる方法を標準メソッド結合と呼ぶ．我々の「スケッチ」と同様，それとは別の方法で結合されるメソッドを定義することもできる．例えばジェネリック関数が適用可能な基本メソッドの返り値全ての和を返すようにできる．

我々のプログラムではメソッドの組み合わせ方は defcomb を呼ぶことで指定した．始めはメソッドは上記の概要に従って組み合わされるが，例えば次のようにすることで，

```
(defcomb price #'+)
```

関数 price が適用可能な全てのメソッドの和を返すようにできる．

CLOS ではこれをオペレータメソッド結合と呼ぶ．我々のプログラムと同様，そのようなメソッド結合は Lisp の式を評価するのだと理解できる．その式の第 1 要素は何らかのオペレータで，引数は（特定的な順に並べた）適用可能な基本メソッドの呼び出しになっているのだ．値を + で結合するようにジェネリック関数 price を定義してあり，適用可能な around メソッドがなければ，その動作はちょうど次のように定義されたと思えばよい．

```
(defun price (&rest args)
  (+ (apply ⟨最も特定的な基本メソッド⟩ args)
     ...
     (apply ⟨最も特定的でない基本メソッド⟩ args)))
```

適用可能な around メソッドがあるときは，標準メソッド結合と同様にそちらが優先になる．オペレータメソッド結合の下では，around メソッドはやはり次のメソッドを call-next-method で呼び出せる．しかし基本メソッドは call-next-method を使えない．（ここには我々の「スケッチ」と違いがある．そのようなメソッドでも call-next は使えるままになっていた．）

CLOS ではジェネリック関数の使うメソッド結合の種類を defgeneric のオプショナル引数:method-combination で指定できる．

```
(defgeneric price (x)
  (:method-combination +))
```

するとメソッド price はメソッド結合に + を使う．値段を持つクラスを何か定義するとき，

```
(defclass jacket nil nil)
(defclass trousers nil nil)
(defclass suit (jacket trousers) nil)

(defmethod price + ((jk jacket)) 350)
(defmethod price + ((tr trousers)) 200)
```

suit のインスタンスの価値を求めると，適用可能なメソッド price の和が得られる．

```
> (price (make-instance 'suit))
550
```

以下のシンボルが defmethod の第 2 引数や defgeneric のオプション:method-combination の値に使える．

```
+   and   append   list   max   min   nconc   or   progn
```

define-method-combination を呼ぶことでさらに別のメソッド結合が定義できる．詳細は CLtL2 の 830 ページを参照すること．

ジェネリック関数の従うメソッド結合を一旦指定すると，そのジェネリック関数の全てのメソッドで同じようにしなければならない．price を定義する defmethod の第 2 引数に別のオペレータ（または:before や:after）を使おうとするとエラーになる．price に使われるメソッド結合を本当に変えたいなら，fmakunbound を呼んでジェネリック関数全体を削除しなければならない．

24.6 CLOS と Lisp

CLOS は埋め込み言語のよい例になっている．普通，こういったプログラムには利点が 2 つある．

1. 埋め込み言語は，その中でも変わらない概念で思考を続けることができるように，その環境と概念的に深く統合することができる．
2. 埋め込み言語は，基盤の言語がすでに扱えることを全て活用できるので，強力になり得る．

CLOS はどちらでも成功している．Lisp とは非常によく統合されており，さらに Lisp の元々持っていた抽象化技法をうまく活用している．実際，Lisp はしばしば CLOS 越しに見ることができる．オブジェクトの貌を，それを覆うシートを通して見る感じだ．

普通，マクロの層を通じて CLOS を扱うのは偶然ではない．マクロは変換を行うし，CLOS は，本質的には，オブジェクト指向の抽象化技法に従うプログラムを取って Lisp の抽象化技法に従うプログラムに翻訳するプログラムだ．

25-1, 2 節で示唆したように，オブジェクト指向プログラミングの抽象化技法は見事に Lisp のそれに対応させることができ，前者を後者の特別な場合と呼ぶこともできそうな位だ．オブジェクト指向プログラミングのオブジェクトは Lisp のオブジェクトとして，またそのメソッドはクロージャとして容易に実装できる．そのような同型写像の活用により，原始的なオブジェクト指向プログラミングをコード数行で，また CLOS のスケッチを数ページで実現できた．

CLOS は我々の「スケッチ」より遥かに巨大で強力だが，埋め込み言語としての本性を隠す程には至らない．`defmethod` を例に取ろう．CLtL2 では明言こそしていないが，CLOS のメソッドにはレキシカルクロージャの持つ力が全て備わっている．ある変数のスコープ内で複数のメソッドを定義すると，

```
(let ((transactions 0))
  (defmethod withdraw ((a account) amt)
    (incf transactions)
    (decf (balance a) amt))
  (defmethod deposit ((a account) amt)
    (incf transactions)
    (incf (balance a) amt))
  (defun transactions ()
    transactions))
```

実行時には変数へのアクセスがクロージャ同様に共有される．メソッドでこれができるのは，構文の裏側ではそれらはクロージャだからだ．`defmethod` の展開形では，その本体はシャープ・クォート付き λ 式の本体内にそのまま出てくる．

第 7-6 節では，マクロの動作を理解する方が意味を認識するより容易だと指摘した．同様に，CLOS を理解する鍵はそれがどのように Lisp の基本的抽象化構造に対応するかを理解することだ．

24.7 いつオブジェクトを使うのか

オブジェクト指向スタイルは幾つかの異なる利点をもたらす．その利点をどの程度必要とするかはプログラム毎に異なる．連続スペクトルの一端には，オブジェクト指向による抽象化で最も自然に表現されるプログラム——例えばシミュレーション——がある．もう一端には，主に拡張可能にするためにオブジェクト指向スタイルで書かれたプログラムがある．

実際，拡張性はオブジェクト指向スタイルの最大の利点の一つだ．プログラムは単一のモノリシックなコードの塊になるのではなく，小さい部品毎に書かれ，各々に目的に従ってラベルが付く．よって後日，誰か別の人がプログラムを修正したいと思ったとき，変更の必要な部分は容易に見つかる．タイプ `ob` のオブジェクトの画面への表示方法を変更したいと思ったら，クラス `ob` のメソッド `display` を変更すればよい．`ob` と似ているが幾つかの点で異なる，オブジェクトの新しいクラスを作りたいと思ったら，`ob` のサブクラスを作ればよい．サブクラス内では望みの属性に変更ができ，残りは全てデフォルトでクラス `ob` から継承される．また `ob` の一つのオブジェクトだけを残りと違う挙動にしたいなら，`ob` の子を作って子の属性を直接修正すればよい．プログラムが始めから注意して書かれていたら，コードの他の部分を見もせずにどのような修正でも行える．この観点からすると，オブジェクト指向プログラムとは表のように構成されたプログラムだ．適切な項目を探すことで直ちに安全に変更できる．

拡張性のためにオブジェクト指向スタイルが必要になる，ということはまずない．実際，拡張可能なプログラムはオブジェクト指向である必要は全くない．前の章で示したのは，Lisp プログラムがモノリシックなコードの塊である必然性はないということに過ぎない．Lisp は拡張性のあらゆる選択肢を提供する．例えば，正に文字通り表のように構

成されたプログラムが書ける．配列に保持したクロージャの集合から成るプログラムだ．

必要なのが拡張性なら，「オブジェクト指向」プログラムと「伝統的」プログラムの間で選択を行う必要はない．Lisp プログラムには，オブジェクト指向の技法に頼らずに，必要としているだけの拡張性を与えることができることが多い．クラス内のスロットはグローバル変数だ．仮引数を使えるところでグローバル変数を使うのがエレガントでないのと全く同様，素の Lisp で労力をかけずに同じことができるときには，クラスとインスタンスの世界を構築するのはエレガントではない．CLOS が追加されたことで Common Lisp は広く利用されているプログラミング言語のうち最も強力なオブジェクト指向言語になった．皮肉なことに，オブジェクト指向言語の必要性が一番低い言語でもある．

付録 A　パッケージ

Common Lisp ではパッケージによってコードをモジュールにまとめる．初期の Lisp 方言には oblist と呼ばれるシンボル表があり，システムが読み込んだシンボルの一覧を保持していた．シンボルが oblist 内に持つエントリを通じてシステムはその値や属性リスト等を参照していた．oblist に含まれるシンボルはインターン (intern) されていると言われた．

最近の Lisp 方言は oblist の概念を複数のパッケージに分割した．現在，シンボルは単にインターンされているのではなく，ある特定のパッケージにインターンされている．あるパッケージにインターンされたシンボルが他のパッケージで参照できるようにするには明示的な宣言が必要なので（誤魔化す方法はあるが），パッケージはモジュール性を促進する．

パッケージは Lisp オブジェクトの一種だ．現在のパッケージは常にグローバル変数 *package* に保持されている．Common Lisp 処理系が起動したとき，現在のパッケージはユーザ用パッケージ，すなわち user（CLtL1 準拠の実装）または common-lisp-user だ（CLtL2 準拠の実装）．

普通，パッケージを区別するものはその名前で，それは文字列だ．現在のパッケージの名前を探すには次のようにする．

```
> (package-name *package*)
"COMMON-LISP-USER"
```

普通，シンボルは読み込まれた時点のパッケージにインターンされる．シンボルがインターンされたときのパッケージを探すには symbol-package を使えばよい．

```
> (symbol-package 'foo)
#<Package "COMMON-LISP-USER" 4CD15E>
```

返り値は実際のパッケージオブジェクトだ．後で使うために foo に値を定めよう．

```
> (setq foo 99)
99
```

in-package を呼ぶことで新しいパッケージに入ることができる．存在していないなら必要に応じて新たに作られる[*42]．

```
> (in-package 'mine :use 'common-lisp)
#<Package "MINE" 63390E>
```

この時点で奇妙な調べを耳にすることになるだろう．違う世界に移ったからだ．foo はここでは別物になる．

```
MINE> foo
>>Error: FOO has no global value.
```

どうしてこうなったのだろう？ 先ほど値を 99 に設定した foo はここ mine 内の foo とは別だからだ[*43]．ユーザ用パッケージの外から元の foo を参照するには，パッケージ名とコロン 2 つを前に付ける必要がある．

```
MINE> common-lisp-user::foo
99
```

よって同一の印字名を持つ異なるシンボルはパッケージを別々にすれば共存できる．パッケージ common-lisp-user 内に一つ目の foo が，パッケージ mine 内に別の foo があってもよく，それらは別個のシンボルだ．実際，そのことは

[*42] 古い Common Lisp 処理系ではキーワード引数 :use を使わないこと．

[*43] Common Lisp 処理系の中には，ユーザ用パッケージ内以外ではトップレベルのプロンプトの前にパッケージ名を表示するものがある．必須の仕様ではないが，便利な機能だ．

パッケージの中核をなす．プログラムを分離したパッケージ内で書けば，関数や変数の名前を選ぶ際に，誰か別人が別のものに同じ名前を使わないかと心配することはない．例え同じ名前を使われても同じシンボルにはならないのだ．

パッケージは情報隠蔽の手段にもなる．プログラムは名前を使って関数と変数を参照する．それらに与えられた名前をパッケージ外では利用できないようにすれば，別パッケージ内のコードがその名前の参照する実体を使ったり変更することはできなくなる．

普通，プログラム内で 2 重コロンでパッケージ名を前置するのはよいスタイルではない．そうするとパッケージが提供するはずのモジュール性を侵していることになる．シンボルを 2 重コロン付きで参照しなければならないのは，誰かが参照してほしくないと思っているせいだ．

普通はエクスポートされたシンボルだけを参照すべきだ．シンボルを，それが存在するパッケージからエクスポートすることで，別のパッケージからも見えるようにできる．シンボルのエクスポートには（ご想像通り）export を呼ぶ．

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
T> (setq bar 5)
5
```

すると mine に戻ったときも bar はコロン 1 つだけで参照できる．パブリックに利用できる名前になったからだ．

```
> (in-package 'mine)
#<Package "MINE" 63390E>
MINE> common-lisp-user:bar
5
```

bar を mine にインポートすることで，さらに一歩深く進み，mine が実際にシンボル bar をユーザ用パッケージと共有するようにできる．

```
MINE> (import 'common-lisp-user:bar)
TMINE> bar
5
```

bar をインポートした後はパッケージ限定子を付けずに参照できるようになる．このとき 2 つのパッケージは同じシンボルを共有している．個別に mine:bar を作ることもできない．

すでに同じ名前のシンボルがあったらどうなるだろう？ その場合，import の呼び出しはエラーになる．例えば foo をインポートしようとすると次のようになる．

```
MINE> (import 'common-lisp-user::foo)
>>Error: FOO is already present in MINE.
```

先ほど mine 内で foo を評価しようとして失敗したとき，結果的にシンボル foo をそこでインターンした．foo にはグローバルな値はなく，そのためエラーが出た．しかしインターンは，名前を打ち込んだ結果として行われていた．よって今 foo を mine にインポートしようとしたとき，すでに同じ名前のシンボルがあったことになる．

あるパッケージを別のパッケージで使うと定義することでシンボルを一気にインポートできる．

```
MINE> (use-package 'common-lisp-user)
T
```

するとユーザ用パッケージでエクスポートされた全てのシンボルが自動的に mine でインポートされる．（foo がユーザ用パッケージでエクスポートされていたらこれもエラーになる．）

CLtL2 では，組み込みオペレータと変数の名前を含むパッケージは lisp でなく common-lisp と呼ばれ，新パッケージは最初はそれを使わないようになっている．mine を作った in-package の呼び出しでこのパッケージを使ったので，Common Lisp 組み込みの名前は全てそこで見える．

```
MINE> #'cons
#<Compiled-Function CONS 462A3E>
```

現実的には，新パッケージでは必ず common-lisp （もしくは Lisp の組み込みオペレータを含むいずれかのパッケージ）を使わざるを得ない．そうしないと新パッケージから抜けることもできなくなる．

コンパイルのときは，普通はパッケージに関する操作はトップレベルのようには行われない．パッケージに関するオペレータの呼び出しはソースファイル内に含まれる．普通はファイル先頭に in-package と defpackage を置けば十分だ．（マクロ defpackage は CLtL2 で新登場したものだが，古い処理系でも提供しているものがある．）異なるパッケージのコードを含むファイル先頭では次のようにすることになるだろう．

```
(in-package 'my-application :use 'common-lisp)

(defpackage my-application
  (:use common-lisp my-utilities)
  (:nicknames app)
  (:export win lose draw))
```

こうするとファイル内のコード——正確には，ファイル内の名前——がパッケージ my-application に入る．このパッケージは common-lisp の他に my-utilities を使っており，それがエクスポートしたシンボルはみなパッケージ名を前置せずに使える．

パッケージ my-application 自身は 3 個のシンボルをエクスポートしているだけだ．すなわち win，lose と draw だ．in-package の呼び出しによって my-application にニックネーム app がついたので，他のパッケージのコードはそれらに app:win などとして参照できる．

パッケージの提供するモジュール性は実際は少々変わっている．それはオブジェクトではなく名前のモジュールだ．common-lisp を使う全てのモジュールは cons という名前をインポートする．なぜなら common-lisp にはその名前の関数が含まれるからだ．しかし結果としては，cons という名前の変数が common-lisp を使う全てのパッケージで見えるようになる．Common Lisp の他の名前空間についても同様だ．パッケージが分かり辛いとしたらこのせいだろう．オブジェクトではなく名前を基本にしているからだ．

パッケージに関わる処理は実行時ではなく読み込み時に行われることが多いが，これはいくぶん混乱の元になり得る．2 番目に入力してみせた式が値を返したのは，

```
(symbol-package 'foo)
```

入力が答えを創り出したからだ．この式を評価するには Lisp はそれを read しなければならないが，それは foo のインターンを伴う．

別の例として，先ほど提示したものについて考えよう．

```
MINE> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (export 'bar)
```

普通，トップレベルに入力された 2 つの式はそれらを 1 つの progn で括ったものと等価だ．しかしここではそうではない．次のようにするとエラーになってしまう．

```
MINE> (progn (in-package 'common-lisp-user)
             (export 'bar))
>>Error: MINE::BAR is not accessible in COMMON-LISP-USER.
```

それは評価の前に progn 式全体が read で処理されるからだ．read が呼ばれた時点でのパッケージは mine なので，bar は mine:bar と解釈される．common-lisp-user:bar でなくそちらをユーザ用パッケージからエクスポートしようとしていたのだ．

パッケージの定義方法のせいでシンボルをデータとして使うプログラムは書き辛くなっている．例えば noise を次のように定義してみよう．

```
(in-package 'other :use 'common-lisp)
(defpackage other
  (:use common-lisp)
  (:export noise))

(defun noise (animal)
  (case animal
    (dog 'woof)
    (cat 'meow)
    (pig 'oink)))
```

そして別のパッケージで noise をパッケージ限定のないシンボルを引数にして呼ぶと，普通は case の分岐節の末尾を超えて nil が返る．

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise 'pig)
NIL
```

これは引数として渡したものは common-lisp-user:pig なのに（←からかってるんじゃないよ）case 式のキーは other:pig だからだ．noise を期待どおりに機能させるには，中で使われている 6 個のシンボルを全てエクスポートし，noise を使うあらゆるパッケージでそれらをインポートしなければならない．

この場合，普通のシンボルの代わりにキーワードを使うことで問題を回避できる．noise が次のように定義されていたら，

```
(defun noise (animal)
  (case animal
        (:dog :woof)
        (:cat :meow)
        (:pig :oink)))
```

どのパッケージからも安全に呼び出せる．

```
OTHER> (in-package 'common-lisp-user)
#<Package "COMMON-LISP-USER" 4CD15E>
> (other:noise :pig)
:OINK
```

キーワードは黄金のようなもので，どこでも通用し，それ自身が価値を持つ．つまりどこでも見えるしクォートする必要は一切ない．defanaph（xzq ページ）のようなシンボルを中心にした関数はほぼ必ずキーワードを使って書くことになる．

パッケージは混乱の元になりがちだ．この案内ではほとんど表面を撫でてすらいない．詳細については CLtL2 の第 11 章を参照のこと．

訳者あとがき

まだ修正しないといけない点がたっぷりあるはずなので，気長にお待ち下さい．翻訳に手を付けたのは 2003 年頭だったのですがね．

最終更新日: 2004 年 11 月 27 日