Emacs Lisp によるプログラミング

初心者のための入門

Emacs Lisp によるプログラミング

初心者のための入門
Edition 1.04, 2 October 1995
日本語版 0.9, 19 September 1996

Robert J. Chassell 著　松田茂樹 訳

Copyright © 1990, '91, '92, '93, '94, '95 Free Software Foundation, Inc.

Published by the Free Software Foundation, Inc.
59 Temple Place, Suite 330
Boston, MA 02111-1307 USA

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

序文

GNU Emacs というエディタは大部分が Emacs Lisp と呼ばれるプログラミング言語で書かれている。このプログラミング言語の中に書かれているコードは、計算機がコマンドが与えられた時何をすべきかを教えるソフトウェア — 命令の集り — である。 Emacs はあなたが Emacs Lisp で新しいコードを書き、簡単に拡張機能としてエディタに組込むことが出来るよう設計されている。これが Emacs が「extensible editor (拡張可能エディタ)」と呼ばれる所以である。

(実際には、 Emacs は通常エディタが出来ると思われている機能をはるかに上回ることをやってのけるので、むしろ「extensible computing environment (拡張可能計算機環境)」とでもいうべきだろう。まあ、ちょっと長ったらしいフレーズになってしまうし、あなたが Emacs の中でする全てのこと — マヤ暦や月の満ち欠けを調べたり、多項式を整理したり、コードをデバッグしたり、ファイルを扱ったり、手紙を読んだり書いたりといったこと — はもっとも広い意味では編集 (edit) の範疇に属するものではあるが。)

何はともあれ、 Emacs Lisp はテキストエディタに関する言語と思われがちであるが、実際は計算機全体に関するプログラミング言語である。あなたはこの言語を他のプログラミング言語と同じ様に使うことが出来るのだ。

多分、あなたはプログラミングを理解したいと思っていることだろう。あるいは、 Emacs の機能を拡張したいとか、プログラマになりたいと思っているかもしれない。

この Emacs Lisp についての文書は、その出発点を示すことを意図している。つまり、あなたをプログラミングの基本的な事柄に慣れさせ、更により大切なことだが、あなたが自分自身でより先に進むにはどうしたら良いかを示すことを目的としている。

この文書の中には、 Emacs の中で走らせることの出来る小さなサンプルプログラムが登場する。この文書を GNU Emacs の Info の中で読めば、そのプログラムをその場で走らせることが出来る。(これは簡単に実行出来る。その方法はそのプログラムが出てきた時に説明する。) あるいは、この文書を印刷製本し Emacs が走っている計算機のそばに座りながら読むことだって出来る。(私はこの方法が好きだ。私は印刷された本の方が好みなので。) 例えあなたの近くで Emacs が動いていなくても、この本を読むことは出来る。が、その場合はこれを小説か、まだ行ったことのない国の旅行ガイドのように扱うのが良いだろう。面白いにしても実際とは違うものだ。

この文書の多くの部分は、 GNU Emacs の中で使われるコードのガイドツアーというべきことに費やされている。これらのツアーの目的は次の二つである。一つ目は、あなたを実際に役立つコード (あなたが毎日使うコード) に親しませること。二つ目は、あなたを Emacs を活用する方法に親しませることである。エディタがどのように実装されているかを見ることは面白いものである。また、私はあなたがソースコードを眺める習慣を身に付けることを望んでいる。あなたはそこからいろいろなこと学び、そしてアイディアを発掘することが出来る。 GNU Emacs を手に入れるということは、宝が隠されたドラゴンの洞窟を発見したようなものなのである。

これらの例やガイドツアーを通して、エディタとしての Emacs を学んだり、プログラミング言語としての Emacs Lisp を学習するだけでなく、 Emacs が Lisp プログラミング環境であることを理解するきっかけが与えられることだろう。 GNU Emacs はプログラミングをサポートし、あなたが快適に使いこなしたくなるような様々な道具を与えてくれる。例えば `M-.` (`find-tag` コマンドを発生させるキー) なんかがそうだ。また、エディタ環境の一部であるバッファやその他のものを学べる。 Emacs のこれらの特徴を学ぶことは、あなたが故郷に帰る新しいルートを探すことに例えられよう。

最後に私は、あなたがまだ知らないプログラミングの側面を学ぶために Emacs を利用して出来る幾つかのテクニックを伝えることが出来ればと思う。難解な事柄を理解したり、何か新しいことをしたりする際、その解決方法にしばしば Emacs そのものを利用することが出来る。こういった自己完結性があることは、単に気分が良いというだけでなく、実際に非常に便利なものである。

これは誰のために書かれたものか

この文書は、プログラマではない人々への、初歩的な入門書として書かれている。もし、あなたがプログラマであるなら、あなたはこのような簡単なものでは満足出来まい。というのも、あなたは既にリファレンスマニュアルを読むのに熟達しているかもしれないし、その場合この文書のような書き方ではまどろっこしく感じるだろうからだ。

例えば、この文書を読んだある熟練したプログラマは、私に次のように言った。

> 僕は、リファレンスマニュアルから学ぶ方が好きなんだ。各々のパラグラフに「ダイブして」そして各パラグラフの間で「息継ぎをする」感じだ。
>
> 僕はあるパラグラフの最後に辿り着いたら、その主題については終了したものと見做すんだよ。つまり、(次のパラグラフでより詳しいことが説明されるような場合を除いて) 必要なことは全て分ったと考えるわけなんだ。だから無駄な沢山の繰り返しがなくって、必要な情報が載っている個所へのポインタがきちんと整備されているようなのがいいな。

この入門書はこのような人のために書かれたのではない！

まず一つ目に、私は全てのことを少なくとも 3 回は繰り返すようにした。まずはそれを紹介し、次にそれがどんな場合に使われるかを見て、そして、別の使われ方を見るか、それを復習するといった具合だ。

二つ目に、私はその主題についての全ての情報を一箇所にまとめるようなことは殆どしなかった。ましてや、一つのパラグラフに押し込めるようなことは避けた。これは個人的な考えだが、そういうやり方は読者に過剰な負担を強いることになる。その代わりに、私は各々の場合に必要なことだけを説明するように心がけた。(時々は、後で正式に説明する時に戸惑うことのないよう、ちょっと先走った事柄の説明も含めたりしたが。)

この文書を読む時は、あなたは全てのことを初めて学ぶものと考えられている。取り上げられた幾つかの項目については、言わば、会えば会釈する程度の浅い付き合いしかする必要がないものもある。あなたが本当に大切なことが何かに気付き、それに集中することが出来るだけの十分なヒントを提供出来るように、この文書をうまく構成出来ていればと思う。

あなたは、幾つかのパラグラフには「ダイブする」必要がある。それらには、別のもっと楽な読み方があるわけではない。しかし、私はそのようなパラグラフの数はなるべく押さえたつもりだ。この本は、そびえたつ山ではなく、ちゃんと登ることができる丘であるように書かれている。

この『Emacs Lisp によるプログラミング』— 初心者のための入門 — は次の本と兄弟関係にある。

『*The GNU Emacs Lisp Reference Manual*』.[1]

リファレンスマニュアルには、この入門書よりも詳しいことが載っている。またリファレンスマニュアルでは、あるトピックについての全ての情報は一箇所に固まっている。もしあなたが、上に挙げたプログラマのようなタイプなら、こっちを読むべきだろう。また、勿論この入門書を読み終えた後、自分自身でプログラミングをする場合には、リファレンスマニュアルがいろいろと便利であることが納得出来るだろう。

Lisp の歴史

Lisp は最初、1950 年代の終わりに、マサチューセッツ工科大学で人工知能の研究のために開発された。Lisp 言語が持つ素晴しい能力は、例えば、エディタのコマンドを書くなどの、他の目的についても優秀さを発揮した。

GNU Emacs Lisp は Maclisp の影響を多く受けている。Maclisp は 1960 年代に MIT で書かれた。Common Lisp の影響もいく分受けている。こちらは 1980 年代に標準となった。しかしながら、Emacs Lisp は Common Lisp と比べて非常に単純である。(標準的な Emacs の配布には、オプションとして 'cl.el' が含まれている。これは Emacs Lisp に多くの Common Lisp の機能を付け加えるためのものである。

初心者の人へ

もしあなたが GNU Emacs について知らないとしても、この文書には何かしら役に立つことが書かれているだろう。しかしながら、たとえ、コンピュータのスクリーンの中を動き回るということだけでも良いから、Emacs を学ぶことを薦める。あなたはオンラインのチュートリアルを使って自分自身でその使い方を学べる。そのために

[1] 日本語訳もある。『GNU Emacs Lisp リファレンスマニュアル』Bil Lewis, Dan LaLiberte and the GNU Manual Group 著、榎並嗣智 井田昌之監訳、発売 丸善、発行 透土社、定価 6796 円、ISBN 4-924828-39-4

は、 C-h t とタイプするだけでよい。(これは CTRLキーと h キーを同時に押して離し、次に、 t キーを押して離すことを意味する。) (訳註： Mule ならば、 C-h T で日本語や韓国語、中国語等のチュートリアルが行える。)

また、私はしばしば Emacs の標準的なコマンドを、そのコマンドを引き起こすキーを書き、その後に、そのコマンドの名前を括弧でくくって書くことで示したりする。例えば M-C-\ (indent-rigion) といった感じだ。(もし望むなら、そのコマンドを引き起こすためにタイプするキーを交換することも出来る。これは、リバインディング (*rebinding*) と呼ばれる。Section 16.11, “キーマップ”, p. 149, 参照)。この M-C-\ という省略形は、 META キーと CTRL キーと \ キーを同時にタイプすることを示している。このような組み合わせはよくキーコードと呼ばれる。理由は、ピアノでコードを弾くのに似ているからである。もし、キーボードに META キーが無ければ、 ESC キーがその代わりになる。この場合は、 M-C-\ は、まず ESC を押して離し、ついで、 CTRL と \ を同時にタイプすることを意味する。

もし、この文書を GNU Emacs の Info を使って読んでいるなら、この文書全てをただ単にスペースバー、 SPC を押し続けることで読んでしまうことが出来る。(Info について学ぶには、 C-h i とタイプし、ついで Info を選択すれば良い。)

言葉の使い方についての注意だが、私が単独で Lisp という単語を使う場合は、大概、 Lisp の様々な方言にも通用する、一般的な事柄を述べている。しかし、 Emacs Lisp と言うときは特に GNU Emacs Lisp のことを言及している。

謝辞

執筆に際し助けて頂いた全ての人々に感謝します。特に、 Jim Blandy, Noah Friedman, Jim Kingdon, Roland McGrath, Frank Ritter, Randy Simith, Richard M.Stallman, そして、Melissa Weisshaus に感謝します。また、Philip Johnson と David Stampe の忍耐強い励ましにも感謝します。この文書の間違いは、全て私の責任です。

訳者まえがき

この文書は Robert J. Chassell 氏の『Programming in Emacs Lisp: A simple introduction』を訳したものです。個人用の訳なので拙い所も多く、また形式上もあまりしっかりしてはいません。が、一応実用には耐えられるのではないかと判断しています。また、訳者の実力不足のために訳に間違い等が含まれていると思われますが、お気付きの方は連絡していただけると嬉しく思います。

1 List 処理

訓練を受けてない人にとっては、Lisp は奇妙なプログラミング言語である。Lisp のコードの中には到る所に括弧が見受けられる。中には、Lisp と言う名前は、'Lots of Isolated Silly Parentheses' を表わしているのだとほざく人までいる。しかし、これは根拠のない主張である。Lisp は LISt Processing を表わし、リスト (や、リストのリスト) を両側を括弧で挟んで扱うプログラミング言語である。括弧は、リストの境界を示している。時々、リストの頭にアポストロフィ、即ち引用符 '' が付いていたりすることもある。リストは Lisp の基礎である。

1.1 Lisp のリスト

Lisp の中では、リストは '(rose violet daisy buttercup) という格好をしている。このリストの頭には一つのアポストロフィが付いている。これは、よりあなたが親しんでいるであろう次のような形のリストに書くことも出来る。

```
'(rose
  violet
  daisy
  buttercup)
```

このリストの要素は四つの異なる花の名前であり、各々が空白で区切られ、括弧に囲まれている。これは石の壁に囲まれた広場の花を連想させる。

リストはまた、要素として数値を持つことも出来る。例えば、(+ 2 2)なんかがそうだ。このリストはプラスの符号 '+' とその後に続く二つの '2' を持っていて、各々は空白で区切られている。

Lisp の中では、データとプログラムの両方が同じ様に表現される。つまり、これらは共に、空白で区切られ括弧で囲まれた単語や数値や他のリストからなるリストである。(プログラムがデータと似ているために、あるプログラムは容易に他のプログラムのデータとして役に立つ; これが Lisp の極めて強力な特徴の一つである.) (ついでに言っておくと、この直前の括弧で囲まれた補足は Lisp のリストではない。というのも、これらは ';' と '.' という句読点を中に含んでいるからだ。)

(訳註 : 日本語としては変かもしれないが、つじつまを合わせるために ';'と '.' を使った。)

ここで別のリストの例を挙げよう。今度は中にリストを含んでいる。

```
'(this list has (a list inside of it))
```

このリストの構成要素は、'this', 'list', 'has', という単語と、'(a list inside of it)' というリストである。内部にある方のリストは、'a', 'list', 'inside', 'of', 'it' という単語からなっている。

1.1.1 Lisp のアトム

Lisp の中では、今まで単語と読んで来たものは、*atom* (アトム) と呼ばれる。この言葉は歴史的には「これ以上分解出来ない」という意味を表わす単語アトムから来ている。Lisp について話している限りは、我々がリストの中で使ってきた単語は、それ以上小さなプログラムのパートには分解出来ない。それは数字や '+' みたいな一つの文字からなる記号でも同じである。一方リストは、アトムと違って幾つかの部分に分解することが出来る。(Chapter 7, "car cdr & cons: 基本関数", p. 57, 参照。)

リストの中では各々のアトムは互いに空白で区切られている。また括弧のすぐ隣りに位置することが出来る。

技術的な言い方をすると、Lisp のリストとは、空白で区切られた、いくつかのアトムないしは他のリスト、あるいはアトムと他のリストの両方を括弧でくくったものである。中には一つしかアトムが無くても良いし、全く無くても良い。中に何も入っていないリストというのは () であるが、これは空リスト (*null list*) と呼ばれる。空リストは同時にアトムでもリストでもある唯一のものである。

アトムやリストを表示したものは、*symbolic-expression*, もしくはもっと簡単に S 式と呼ばれる。*Expression* という単語自身は、表示された表現か、あるいは計算機の内部に保持されているアトムやリストのどちらか

を表わす。 *Expression* という言葉はしばしば曖昧に用いられる。(また、多くの本では形式 (*form*) という言葉が expression の同意語として使われている。)

(訳註：この訳では以下、 expression を式、ないしはＳ式と訳していることが多い。)

ついでだが、我々の世界を構成しているアトム (原子) がそう名付けられたのは、それらが分割不可能と考えられていた時のことである。しかしその後、物理で言うアトムは分割不可能ではないことが発見された。物質は、一つのアトムかほぼ同じサイズの二つの部分に分裂する。(訳註：意味不明。) 物理で言うアトムはまだその本当の性質が解明されぬままに時期尚早にして名付けられてしまった。 Lisp の中では、ある種のアトム、例えば配列 (array) は、複数の部分に分解出来る。しかし、そのメカニズムはリストを分解する時のメカニズムとは違う。リストの操作に関する限り、リストの中のアトムは分解不可能である。

英語におけるのと同様、 Lisp のアトムを構成する文字要素は、一つの単語としての文字とは違う。例えば、南アメリカナマケモノ (South American sloth) を表わす 'ai' は 'a' と 'i' という二つの単語とは全く異なる。

自然の中には沢山の種類のアトムがあるが、 Lisp の中には数種類のアトムしかない。例えば、 37、 511、 1729 といった数値 (*number*)、そして '+'、'foo'、や 'forward-line' といったシンボル (*symbol*) なんかである。今まで挙げたリストの例の中に出てきた単語は全てシンボルである。普段の Lisp の会話では、アトムという単語はあまり使われないが、それはプログラマは普通今使っているアトムがどの種類のアトムかをより限定的にみようとしているからである。 Lisp のプログラミングはもっぱらリストの中のシンボル (と、ときどき数値) についてのものである。(ついでに言うと、今出てきた三つの単語を括弧でくくったものはちゃんしたリストである。何故なら、それはアトムからなり (今の場合はシンボルである)、空白で区切られ、括弧でくくられており、 Lisp でないような句読点は含んでないからである。)

それに加えて、二つの二重引用符で狭まれたテキストは — それが一つの文であろうと、あるいはパラグラフであろうとも — アトムである。例を挙げておこう。

```
'(this list includes "text between quotation marks.")
```

Lisp では句読点が入っていようが空白があろうが、全ての二重引用符で囲まれた文は一つのアトムである。この種のアトムは 文字列 (*string*) と呼ばれ、計算機が人間が読める形でメッセージを出す用途等に使われる。文字列は数値ともシンボルとも違う種類のアトムであり、異なる使い方をされる。

1.1.2 リストの中の空白

```
'(this list
   looks like this)
```

は、次のリストと全く同じである。

```
'(this list looks like this)
```

どちらの例も、 Lisp にとっては 'this'、'list'、'looks'、'like'、そして 'this' というシンボルがこの順番で並んでいる全く同じリストである。

余分な空白と改行は、人間がよりリストを見やすいようにするために用いられている。 Lisp が表現を読み取る時は、全ての余分な空白を除いてしまう (ただし、最低一個の空白は残している。そうでないとアトムとアトムの区切りが分らなくなってしまう)。

奇妙に思えるかもしれないが、今まで見てきた例で、殆んどの Lisp のリストはカバーされてしまう。 Lisp における他の全てのリストは、多かれ少なかれ、これらの例のどれかに似ている。まあ、もっと長くなったり、複雑になったりはするが。簡単にまとめると、リストは括弧に挟まれ、文字列は二重引用符に挟まれ、シンボルとは単語のようなもので、数値とは数のようなものである。(ある場合には、角括弧や、ドット、それに他の特別な文字が使われたりもする。しかし、それらが無くても我々はかなり先まで進める。)

1.1.3 GNU Emacs によるリストのタイプの支援

もし、あなたが Lisp のＳ式を GNU Emacs の Lisp Interaction mode もしくは Emacs Lisp mode の中でタイプしているなら、幾つかのコマンドを使って、 Lisp のＳ式を整形し読みやすいものにすることが出来る。例

えば TAB キーを押すと、カーソルがある行を自動的に適当な分だけインデントしてくれる。リージョン内のコードをきちんとインデントしてくれるコマンドは、 M-C-\ にバインドされている。インデントは、あなたが、どの要素がどのリストに属しているか見やすいように — サブリストの要素がそれを含んでいるようなリストの要素よりも深くインデントされるように — 設計されている。

更に、あなたが括弧を閉じる時に、 Emacs は一時的にその括弧にマッチする開き括弧の位置にジャンプする。そうすることでどの括弧に対応するかを確かめることが出来る。これは大変便利である。というのも、あなたが Lisp でタイプする全てのリストについて、開き括弧には必ず閉じ括弧がきちんと対応していなければならないからである。 (Section “Major Modes” in *The GNU Emacs Manual*, に Emacs の mode のより詳しい説明が書かれている。)

1.2 プログラムの実行

Lisp におけるリストは、例えどんなリストであってもそのまま走らせることが出来る。実際に走らせてみると (Lisp の専門用語では 評価 (*evaluate*) するという)、計算機は次の 3 つのうちいずれかの動作をする。まずは、そのリストをそのまま返すだけで何もしない、もしくは、あなたにエラーメッセージを返す、あるいは、リストの先頭のシンボルをコマンドとして扱い、なにがしかのことをする。 (普通は勿論、最後の動作があなたの求めるものであろう。)

以前のセクションで挙げた例の中で、リストの頭に付けた単独のアポストロフィ ’ は、引用符ないしは クウォート (*quote*) と呼ばれる。これがリストの頭に付いている場合は、何も付いていない場合と異なり、 Lisp はこのリストについては何もしなくていいんだなと解釈する。しかし、リストの頭に引用符が付いていない場合は、リストの最初の項が特別な意味を持つ。つまり、これは計算機が従うべき命令になるのだ。 (Lisp ではこれらの命令は関数 (*function*) と呼ばれる。) 以前挙げた (+ 2 2) というリストには頭に引用符は付いていないので、 Lisp は + が残りのリストに関する何らかの命令であると解釈する。この場合は、残りの数を加えるというものである。

もし、この文章を GNU Emacs の Info の中で読んでいるのなら、次のようにしてこのようなリストを評価することが出来る。カーソルを次のリストの右側の括弧のすぐ後に持ってきて、 C-x C-e とタイプするのである。

```
(+ 2 2)
```

エコー領域に 4 が見えたはずだ。 (専門用語では、このことを「リストを評価する」と言う。エコー領域というのはスクリーンの最下行のことであり、テキストを表示ないしは「エコー」する。) さて、同じことを頭に引用符が付いたリストでやってみよう。カーソルを下のリストのすぐ後に移動させ、 C-x C-e とタイプする。

```
’(this is a quoted list)
```

この場合はエコー領域に (this is a quoted list) という文が見えたはずだ。

どちらの場合でも、あなたのはコマンドを GNU Emacs の *Lisp* インタプリタ (*Lisp interpreter*) と呼ばれるプログラムに渡している。つまり、インタプリタにそのＳ式を評価しろという命令を与えているのだ。 Lisp インタープリタという名前は、Ｓ式の意味を提供する — つまりそれを解釈する — という人間の仕事を表わす単語から来ている。

リストの一部ではない、単独のアトム — 括弧に囲まれていないもの — を評価することも出来る。この場合も Lisp インタプリタは人間が読むことの出来るＳ式から、計算機の内部の言語への翻訳をする。しかし、これについて議論する前に (Section 1.7, “変数”, p. 6, 参照)、まず我々がエラーを犯した場合に Lisp インタプリタが何をするかを見ていこう。

1.3 エラーメッセージの出力

まず、 Lisp インタプリタがエラーメッセージを出すようなコマンドを与えてみよう。偶然このようなことをしてしまっても何も心配することはない。これは、無害な操作である。実際、我々はしばしば意図的にエラーメッセージを出させることがある。一旦専門用語を理解してしまえば、エラーメッセージから多くの情報を得ることが出来る。そういう意味ではエラーメッセージと言うよりはヘルプメッセージと言うべきだろう。言わば他の国から来た旅行者にとっての道標のようなものである。解読するのは大変だけれども、一度理解してしまえば正しい道を教えてくれるというわけだ。

我々が行うのは、頭に引用符が付いていないにもかかわらず、先頭の要素が意味あるコマンドでないようなリストを評価することである。ここに、我々がさっき使ったリストとほぼ同じだが、頭に引用符が付いていないものがある。これの直後にカーソルを持っていって、 C-x C-e とタイプしてみよう。

```
(this is an unquoted list)
```

今回は次のようなメッセージがエコー領域に表示されたはずだ。

```
Symbol's function definition is void: this
```

(それと、端末がビープ音を鳴らしたかもしれない — 鳴らすものもあるし、鳴らさないものもある。ランプが点滅したりするのもある。これは注意を引くための装置である。) メッセージは他のキーをタイプするとすぐに消えてしまう。例えそれが単にカーソルを移動しただけであってもだ。

我々が既に知っていることを基にして、このエラーメッセージをほぼ解読することが出来る。我々は 'シンボル' という単語の意味を知っている。今の場合、リストの最初のアトムである単語 'this' のことを指している。'関数' という単語は前に一度出て来た。これは極めて重要な単語である。ここでは 関数 (*function*) とは計算機に何かやらせるための命令の集まりである、とでも定義しておけば十分だろう。(技術的には、シンボルは計算機に何処で命令を見つければよいかを教えていることになる。しかし、この辺の事情は複雑だし、しばらくは無視して構わない。)

以上で、'Symbol's function definition is void: this' というエラーメッセージを理解することが出来るようになる。これは、シンボル (つまり、'this' という単語) には計算機が実行出来るような命令セットは定義されていないということである。

この、'function definition is void' というちょっと奇妙なメッセージは Emacs Lisp の実装の仕方を表わすように書かれている。つまり、そのシンボルに対して関数が定義されていない場合、命令が含まれているはずの場所が空 (void) だというわけである。一方で、(+ 2 2) を評価した時にちゃんと 2 に 2 を加えることが出来るということから、シンボル + は計算機が従うべき命令セットを持っており、それらの命令は + に続く数を加えるという物であるに違いないと推測出来る。

1.4 シンボルの名前と関数定義

我々は今まで議論してきたことに基づいて、別の Lisp の特徴 — それも極めて大切な特徴 — を述べることが出来る。即ち + のようなシンボルはそれ自身は計算機が実行出来るような命令ではないということである。その代わりシンボルは、多分、一時的に、命令の定義ないしは命令の集まりの位置を見つける方法として使うことが出来る。我々に見えているのは、それを通して命令を見つけることの出来る名前なのである。人間の名前も同じ働きをする。私は 'Bob' として言及される。しかしながら、私は 'B'、'o'、'b' という文字ではなく、一貫して特定の生命体に付随しているある意識である。名前は私そのものではないが、私を示すのに使われる。

Lisp では、ある命令セットには幾つかの名前が付随している。例えば、数を加えろという計算機の命令には、+ というシンボルと同様、 plusというシンボルも付随させることが出来る。(他の幾つかの Lisp の方言では実際にそうなっている。) 人間の場合でも、私は 'Bob' だけではなく 'Robert' として呼ばれることもあるし、他の呼び方をされることもある。

一方で、一つのシンボルは、一度にただ一つの関数定義しか持つことが出来ない。そうでないと、計算機がどの定義を使うべきか迷ってしまうからだ。仮にこれが人間の場合であれば、'Bob' という名前は世界で一人の人間にしかつけてはいけないということである。ただし、名前に付随する関数定義は簡単に変更することが出来る。(Section 3.2, "関数定義のインストール", p. 20, 参照。)

Emacs Lisp はでかいので、関数が属する Emacs のパートが識別出来るようなシンボルの名前を付け方をするのが普通である。というわけで、 Texinfo を扱うための関数の名前は全て 'texinfo-' で始まっているし、メールを読むための (reading mail) 関数の名前は 'rmail-' で始まっている。

1.5 Lisp インタプリタ

これまでに見てきたことから、リストを評価するよう命令した時に Lisp インタプリタが何をするかを理解することが出来る。まずインタプリタは、 list の頭に、引用符が付いているか見る。もし付いていれば、単にそのリ

ストを我々に渡す。一方、付いていない場合はリストの先頭の要素を見に行き、それが関数の定義を持っているかどうかを調べる。定義されている場合はその定義にある命令を実行し、そうでなければエラーメッセージを表示する。

これが Lisp の動作である。単純だ。これに加えて若干複雑なこともある。それについてはすぐ後で説明するが、基本はこれだけである。勿論、Lisp のプログラムを書く際には、関数の定義を書いたりそれに名前を付けるにはどうすればいいか、あるいはそういうことをあなた自身や計算機が混乱しないように行うにはどうすればいいか等、他にも知らなければいけないことはあるが。

さて、さっき述べた「複雑なこと」の一つ目は、Lisp インタプリタは、リストに加えて引用符が付いておらず、括弧に囲まれてもいない単独のシンボルも評価することが出来るということである。この場合、Lisp インタプリタはそのシンボルの値を変数 (*variable*) として評価しようとする。このことについては変数のセクションで説明することにする。(Section 1.7, “変数”, p. 6, 参照。)

複雑なことの二つ目は、幾つかの関数はその特殊性のために普通のやり方ではうまく働かないことによる。このような関数は 特殊形式 (*special form*) と呼ばれる。これらは、例えば関数を定義したりといった特別な用途を持つ。次の幾つかのセクションでもっと大切な特殊形式を紹介するつもりだ。

三つ目の、そして最後の複雑なことはこうだ。もし現在 Lisp インタプリタが見ている関数が特殊形式ではなく、しかもそれがリストの一部であるなら、Lisp インタプリタはリストが内部にリストを含んでいないかどうかを見る。もし内部にリストを含んでいれば、それがその内部でどういうことをするかを見る。そして、その後にその外側のリストを処理する。また、もしその内部のリストが更に他のリストを中に含んでいれば、まず、それを先に処理してからということになる。即ち、Lisp インタプリタは常に一番内部のリストから処理していく。これは、内部のリストの結果をそれを包む外側のＳ式で使う場合、内部のリストの結果がどうなるかを見てから外側のリストを処理しなければいけないためである。

それ以外の場合は、インタプリタは左から右へＳ式を一つずつ処理していく。

1.5.1 バイトコンパイル

解釈の仕方について、もう一つ興味深いことがある。 Lisp インタプリタは二つの種類の物を解釈出来るということである。一つは人間が読めるコードであり、我々は専らこれに焦点を当てている。もう一つは特別なプロセスコードであり、これはバイトコンパイル (*byte compile*) されたコードで、人間に読める代物ではない。バイトコンパイルされたコードは我々が読めるコードよりも速く走らせることが出来る。

人間が読めるコードをバイトコンパイルされたコードに変換することも出来る。これは `byte-compile-file` 等の、コンパイルコマンドを走らせることによって行う。バイトコンパイルされたコードは普通、‘`.el`’ではなく、‘`.elc`’という拡張子で終わるファイルに保存される。‘`emacs/lisp`’などといったディレクトリィを覗けば、この両方の種類のファイルが見つかる。我々が読むのは ‘`.el`’ という拡張子の付いた方である。

実際問題として、あなたが Emacs をカスタマイズしたり拡張したりする場合には大抵はバイトコンパイルする必要はない。そして私もここではこの話題については述べない。 Section “Byte Compilation” in *The GNU Emacs Lisp Reference Manual*, を見ればバイトコンパイルについての全ての情報が載っている。

1.6 評価

Lisp インタプリタがあるＳ式を処理している時、その動作は 評価 (*evaluation*) と呼ばれる。つまり、「インタプリタはＳ式を評価する」のように言うわけだ。私はこの用語を以前にも何回か用いた。この単語はこの言葉の日常会話での意味、 *Webster's New Collegiate Dictionary* (及び、小学館『プログレッシブ英和中辞典』) によれば「価値や量を確かめる、概算する、見積もる」という意味、から来ている。

Lisp インタプリタはＳ式を評価した後、大抵その関数の定義の中にある計算機に対する命令を実行した時に出力する値を 返し (*return*)、そうでない場合は、おそらくその関数を処理するのを止めてエラーメッセージを出力する。(インタプリタは、急に別の関数を渡されることもあり得るし、あるいは良く言う「無限ループ」に入って現在の処理を何回でも繰り返そうとするかもしれない。が、これらの動作をすることはそれ程多くはないので、我々はこれらの場合は無視することにする。) つまり殆どの場合、インタプリタは値を返す。

インタプリタは値を返すと同時に、カーソルを動かしたり、ファイルをコピーしたりといった他の動作も行う。このような動作は、副作用 (*side effect*) と呼ばれる。例えば結果を表示するとか、我々人間が重要と考える動作もしばしば、Lisp インタプリタにとっては「副作用」でしかない。この専門用語には異和感があるかもしれないが、副作用自体の使い方を学ぶのはかなり簡単だということがその内分るだろう。

以上まとめると、Lisp インタプリタはＳ式を評価した時に、まず殆どの場合ある値を返し、大抵はある副作用を実行する。そして、そうでない場合はエラーを出力する、ということになる。

1.6.1 内部のリストの評価

もしあるリストの内部のリストが評価された場合、その外側のリストが評価される時には最初に内部のリストを評価した時に返った値を情報として使うことが出来る。このことから、何故内部のリストが先に評価されるかが説明出来る。即ち、内部のＳ式が返す値を外部のＳ式が用いるためである。

我々はこのプロセスを次のような例で実際に確かめることが出来る。次の式の直後にカーソルを持っていき、`C-x C-e` とタイプしてみよう。

```
(+ 2 (+ 3 3))
```

数字の 8 がエコー領域に表示されたはずだ。

この例で起こったことはこうだ。まず Lisp インタプリタは内部のＳ式である（+ 3 3）を評価する。それに対して値 6 が返される。それによって外部のＳ式があたかも（+ 2 6）であるように評価され、値 8 を返すことになる。これらを内部に含む評価すべきＳ式はこれ以上存在しないので、インタプリタはこの値をエコー領域に表示する。

さて、ここまでくれば `C-x C-e` で引き起こされるコマンドの名前を理解することは簡単だ。このコマンドの名前は `eval-last-sexp` となっている。この中の `sexp` という文字は 'symbolic expression' (Ｓ式) の略である。ということで、これは直前のＳ式を評価する (evaluate last symbolic expression) ことを表している。

実験的に次に挙げる例のＳ式の直後の行の先頭や、Ｓ式の内部にカーソルを持っていってＳ式を評価してみることも出来る。

これがその例である。

```
(+ 2 (+ 3 3))
```

もしカーソルをＳ式の直後の空行の先頭に置いて `C-x C-e` とタイプしたなら、その場合もやはり 8 がエコー領域に表示されることだろう。では次に、カーソルをＳ式の内部に持っていこう。最後の括弧のすぐ右側の括弧の後にカーソルを持っていって (つまりカーソルは最後の括弧の上に見えている) そこで評価したならば、エコー領域には 6 と表示される。これはコマンドが（+ 3 3）という式を評価したからである。

今度はカーソルを数字の直後に持っていこう。`C-x C-e` とするとその数字そのものが返される。Lisp の中では、数字を評価した場合、その値自身が返される。これが数字がシンボルと違うところである。もし + みたいなシンボルを先頭に持つリストの直後でＳ式を評価したなら、返された値はその名前に付随する関数の定義の中の命令を計算機が実行した結果が返ってくる。が、もしシンボルそのものが評価されたなら別のことが起きる。これについては次のセクションで見ることにしよう。

1.7 変数

Lisp の中では、シンボルは、関数定義を持つことが出来るのと同様に、ある値を持つことが出来る。これら二つは全く別の物である。関数定義は計算機が従うべき命令の集まりであるのに対し、値は数値とか名前のように、何か変化し得るものである。(このようなシンボルが変数と呼ばれるのはこのためである。) どんな Lisp のＳ式もシンボルの値になり得る。例えばシンボルや数値、リスト、文字列なんかが値となることが出来る。値としてのシンボルはしばしば変数 (*variable*) と呼ばれる。

シンボルは、関数定義と値の両方を同時に持つことが出来る。この二つは分離されているのだ。これは、ケンブリッジという名前がマサチューセッツの中のある都市そのものを表わすのと同様に、その都市名に付随するなんらかの情報、例えば、「大きなプログラミングセンター」であるといった情報をも持ち得るのと似ている。

このことを理解するもう一つの考え方を述べよう。それはシンボルを整理棚であると見做す方法である。関数定義はその引き出しの一つに入っており、値はまた別の引き出しに入っている... という具合である。値の入っている引き出しの中身は同じ棚の関数定義の入っている引き出しの中身に影響を与えることなく変えることが可能だし、逆も同様である。

変数 `fill-column` を例にとって、値を持つシンボルというものを説明してみよう。 GNU Emacs の全てのバッファにおいて、このシンボルはある値にセットされている。大抵は 72 か 70 であるが、他の値を持つこともある。この値が何かを見るにはこのシンボル自身を評価してみれば良い。もしこの文章を GNU Emacs の Info で読んでいるなら、カーソルを次のシンボルの直後に持っていって `C-x C-e` とタイプするだけである。

```
fill-column
```

私が今 `C-x C-e` とタイプしてみたところ、 Emacs はエコー領域に 72 と表示した。これが私がこの文章を書いている時に `fill-column` にセットされている値である。あなたの Info のバッファではまた別の値が設定されているかもしれない。変数として評価されて返す値は、ある関数が命令を実行した時に値を返す時と全く同じように表示されることに注意しよう。 Lisp インタプリタの立場からすると、どちらも返り値であることに違いはない。どんな種類のＳ式から来ているかということは一旦その値が分ってしまえば重要ではないのだ。

シンボルはどんな値も持つことが出来る。あるいは専門用語を使っていうなら我々は変数に 72 といった数値や `"such as this"` といった文字列や、（`spruce pine oak`）をバインド (*bind*) (束縛とも言う) することが出来る。変数に関数定義をバインドすることすら可能である。

シンボルに値を設定するには幾つか方法がある。その方法の一つについては、 Section 1.9, “変数の値の設定”, p. 11, を参照のこと。

`fill-column` という単語には周りに括弧がついていないことに注意しよう。これは我々がこのシンボルを関数の名前としては使わなかったためである。 `fill-column` がリストの最初、もしくは唯一の要素であった場合、 Lisp インタプリタはこれに付随する関数定義を見つけようとする。しかし、 `fill-column` は関数定義を持ってはいない。例えば次を評価してみよう。

```
(fill-column)
```

すると次のようなエラーメッセージが出るはずだ。

```
Symbol's function definition is void: fill-column
```

1.7.1 値のないシンボルに対するエラーメッセージ

もし、値を持たないシンボルを評価しようとしたなら、エラーメッセージが返されるはずだ。 2 に 2 を加える例で実験してみよう。次のＳ式のなかで最初の 2 の直前にある `+` の直後にカーソルを持っていって `C-x C-e` とタイプしてみよう。

```
(+ 2 2)
```

次のようなエラーメッセージを受けとるはずだ。

```
Symbol's value as variable is void: +
```

これは我々が見た最初のエラーメッセージとは違っている。さっきは‘`Symbol's function definition is void: this`’. というものだった。今回のケースではシンボルは変数としての値を持っておらず、もう一方のケースではシンボル (‘`this`’ という単語) は関数定義を持たなかった。

この `+` を使った実験の中でやったことは Lisp インタプリタに `+` を評価させ、関数定義では無く変数の値を探させることである。そのためにカーソルを前のようにリストを囲む括弧の後では無く、シンボルのすぐ右隣に持っていったのだった。結果として Lisp インタプリタは直前のＳ式、この場合は `+` そのもの、を評価したのだ。

`+` は関数定義は持っていても値は持ってはいないので、エラーメッセージはシンボルの変数としての値が空だよと報告したのである。

1.8 引数

どうしたら情報が関数に渡されるかを見るために、これまで何回か使ってきた 2 に 2 を足す例をもう一度見てみよう。

```
(+ 2 2)
```

もしこのＳ式を評価したなら数字の 4 がエコー領域に表示される。Lisp インタプリタがやったことは + の後に続く数字を加えることである。

+ によって加えられた数字は関数 + の 引数 (*argument*) と呼ばれる。これらの数字は関数に与えられる、もしくは渡される (*pass* される) 情報である。

「argument」という単語は数学での用法から来ているもので、二人の人が議論するという意味ではない。そうではなく関数、今の場合でいうと + という関数、に与えられる情報のことを指している。Lisp の中では関数に対する引数は関数に続くアトムもしくはリストである。これらのアトムやリストを評価して返された値が関数に渡されることになる。異なる関数は異なる数の引数を取り得る。全く引数を取らない関数もある。[1]

1.8.1 引数のデータ型

関数に渡されるデータの型はその関数がどんな種類の情報を必要としているかによる。+ のような関数の引数は数値でなければならない。というのも + は数値を加える命令だからである。他の関数は引数として他の種類のデータを取る。

例えば、concat という関数は二つ以上のテキストの文字列を連結ないしは合成して一つの文字列を作る。この場合の引数は文字列である。二つの文字文字列 abc, def を連結 (concatenate) すると、abcdef という一つの文字列が出来る。これは次の例で確かめられる。

```
(concat "abc" "def")
```

このＳ式を評価すると "abcdef" という値が返される。

substring のような関数は引数として文字列と数値の両方を取る。これは文字列である最初の引数の部分文字列を返す関数である。引数の数値は３つである。最初の引数は文字からなる文字列である。二番目と三番目の引数は部分文字列の初めと終りの場所を示す数値であり、文字列の最初からの (スペースや句読点も含めた) 文字数を指定する。

例えば次を評価してみよう。

```
(substring "The quick brown fox jumped." 16 19)
```

substring に渡される文字列は例えスペースで区切られた複数の単語からなっていたとしても一つのアトムであることに注意しよう。Lisp は二つの二重引用符の間に狭まれる全てを空白も含めて文字列の一部と数える。substring という関数は一種の「アトム粉砕機 (atom smasher)」だと見做すことが出来る。というのもこいつは他の方法では分離できないアトムを取り込み、その一部を抽出するからである。しかしながら、substring は文字列である引数から部分文字列を抽出することが出来るだけで、他の種類の例えば数値とかシンボルであるアトムについてはそのようなことは出来ない。

---

[1] 「argument」という単語が二つの異なる意味を持つようになった経緯を追ってみるのは、結構興味深いことである。一つは数学での意味であり、もう一つは日常使われている英語での意味である。Oxford English Dictionary によると、この単語は 'to make clear, prove' ('明確にする、証明する') という意味の Latin 語から来ているということだ。その結果、一方では「証明として与えられる証拠」から「与えられた情報」という意味になり、それが Lisp での意味の方に派生していった。しかし、もう一方では「他人が主張するのと対立するような主張をする」という意味に派生し、それが議論することを表わす単語の意味になっていった。(ここでこの英単語が同時に二つの異なる定義を持っていることに注意しよう。これに対して Emacs Lisp では、シンボルは同時に二つの関数定義を持つことは出来ない。)

1.8.2 引数には変数の値やリストも使える

引数は評価された時に値を返すシンボルであることも可能だ。例えば fill-column というシンボルそれ自身が評価された場合には、数値が返される。この数値は加法演算にも使える。次のＳ式の後にカーソルを持っていって C-x C-e とタイプしてみよう。

```
(+ 2 fill-column)
```

返される値は fill-column を単独で評価した場合に得られる数に 2 を加えたものになるはずだ。私には 74 が返される。何故なら fill-column の値が 72 であるからだ。

たった今見た様に、引数は評価された時に値を返すシンボルであり得る。それに加えて、引数は評価された時に値を返すリストでもあっても良い。例えば次のＳ式の中では関数 concat の引数は "The ", "red foxes." そしてリスト（+ 2 fill-column）である。

```
(concat "The " (+ 2 fill-column) " red foxes.")
```

もしこのＳ式を評価したならエコー領域には "The 74 red foxes." が表示されるはずだ。(最後の文字列を表示させるには‘The’という単語の後と‘red’という単語の前に空白を入れなければならないことに注意。) (訳註： 74 の所は勿論場合に応じて変わる。)

1.8.3 可変な数の引数

例えば concat, +, * といった幾つかの関数では引数の数は固定されていない。(* は乗法を表わすシンボルである。) これは次に挙げる各々のＳ式を評価してみると分る。エコー領域に表示されるべき結果はこのテキストの中の‘⇒’の後に書かれている。これは「を評価した結果は」と読み換えられる。

最初は引数がない場合である。

```
(+)        ⇒ 0

(*)        ⇒ 1
```

こちらは一つだけ引数を持っている。

```
(+ 3)      ⇒ 3

(* 3)      ⇒ 3
```

次では、各々三つずつ引数を持っている。

```
(+ 3 4 5) ⇒ 12

(* 3 4 5) ⇒ 60
```

1.8.4 関数に間違った型の引数を与えると

ある関数に間違った型の引数が与えられると Lisp インタプリタはエラーメッセージを返す。例えば、 + という関数は引数の値として数値を要求する。実験的にこれに数値ではなく hello というシンボルを与えてみよう。次のＳ式の後にカーソルを持っていって C-x C-e とタイプしてみよう。

```
(+ 2 'hello)
```

そうするとエラーメッセージが表示される。この場合に起きたことはこうだ。 + は 2 に 'hello を評価して返される値を加えようとした。しかし返された値は hello というシンボルであり数値ではない。だが数値でないものを足すことは出来ない。そのために、 + は加算を実行することが出来なかったのである。

普通はエラーメッセージは助けになるように、つまりあなたが一度読み方を修得した後は意味が分るようなものであろうとする。この場合は

```
Wrong type argument: number-or-marker-p, hello
```

と言ってくる。(訳註：Emacs version 18 では integer-or-marker-p である。Emacs version 19 からは浮動小数点数も扱うようになった。) エラーメッセージの最初の部分は極めて直接的だ。'Wrong type argument' (引数の型が違う) である。次に来るのはミステリアスな専門用語'number-or-marker-p' である。この言葉はあなたに + という関数が要求する引数のタイプを教えようとしている。

number-or-marker-p というシンボルの名前は、Lisp インタプリタが与えられた情報 (引数の値) が数値 (number) もしくはマーカ (marker) (バッファでの位置を表わす特殊なオブジェクト) であるかどうかを判定することを意味している。今の場合、実際に行うのは + にちゃんと加えるべき整数が与えられたかどうかのテストである。そして、マーカと呼ばれる特殊オブジェクトかどうかもテストされる。これは Emacs Lisp 固有のものである。(Emacs の中ではバッファ内での位置がマーカとして記録される。C-@ あるいは C-SPC というコマンドによってマークがセットされるとその位置がマーカとして保持されるのである。このマークは数 — バッファの始まりからその位置までの文字数 — と考えることが出来る。) Emacs Lisp では + はマーカの位置を示す数を数値として加えるのにも使うことが出来るわけである。

number-or-marker-p の中の 'p' は Lisp プログラミングの初期の頃に始まったある試みの具体化である。'p' は「述語 (predicate)」を表わしている。初期の Lisp 研究者の間で用いられていた専門用語では「述語」はある属性が正しいかどうかを判定する関数のことを指していた。従って、'p'は number-or-marker-p が、与えられた引数が数値 (number) もしくは マーカであるかどうかを判定する関数の名前であることを意味している。他の Lisp のシンボルで 'p' で終わるものに zerop がある。これは引数が 0 という値であるかどうかを判定する関数である。同様に listp は引数がリストかどうかを判定する関数である。

さて、いよいよエラーメッセージの最後の部分 hello であるが、これは + に渡された引数の値である。もし加法演算に正しい型のオブジェクトが与えられていたとするなら、渡される値は hello のようなシンボルではなく、例えば 37 のような数値であったはずである。もっとも、その場合は、エラーメッセージは出ないけれども。

1.8.5 関数 message

+ と同様、 message という関数も引数の数は固定されていない。この関数はユーザにメッセージを送るものだが、今説明したように、いろいろと役に立つものである。

メッセージはエコー領域に表示される。例えば次のリストを評価することで、あなたのエコー領域にメッセージを表示させてみよう。

```
(message "This message appears in the echo area!")
```

二重引用符に狭まれた文字列全体が一つの引数になっており、その全体が表示される。(この例ではメッセージそれ自身が二重引用符の中に入っていることに注意しよう。これは、 message という関数が返す値そのものがそうなっているからである。あなたが書くプログラムの中での普通の message の使い方ではエコー領域に表示されるのは副作用によるものである。それには、二重引用符は付いていない。このような例については Section 3.3.1, “インタラクティブな multiply-by-seven”, p. 22, を参照のこと。)

しかしながら、もし '%s' が二重引用符に狭まれた文字文字列に入っていたとしたら、関数 message は '%s' をそのまま表示したりはせずに、文字列に続く引数を見にいく。まずは二番目の引数を評価し、その結果を '%s' のある所に代入して表示する。

次のＳ式の最後にカーソルを持っていって C-x C-e とタイプしてみることで、このことが確かめられる。

```
(message "The name of this buffer is: %s." (buffer-name))
```

Info の中では、 "The name of this buffer is: *info*." という文がエコー領域に表示されたことだろう。 buffer-name という関数は現在のバッファ名を文字列として返す。 message はそれを %s の位置に挿入したのだ。

値を 10 進数として表示するには、'%s' の代わりに '%d' という関数を同じように使う。例えば、現在の fill-column の値をエコー領域に表示させるには次を評価すれば良い。

```
(message "The value of fill-column is %d." fill-column)
```

私の使っているシステムでは、このリストを評価した時点ではエコー領域には "The value of fill-culumn is 72." と表示された。

もし一つ以上の ‘%s’ が二重引用符で狭まれた文字列の中にあったとすると、最初の ‘%s’ の位置にはその文字列に続く最初の引数の値が表示され、二番目の ‘%s’ の位置には二番目の引数の値が表示され... という具合になる。例として次を評価してみよう。

```
(message "There are %d %s in the office!"
         (- fill-column 14) "pink elephants")
```

かなり風変わりなメッセージが表示されたことと思う。私のシステムでは、 "There are 58 pink elephants in the office!"。のように表示された。

ここでは（- fill-culumn 14）というＳ式が評価され、その結果として返された値が ‘%d’ の位置に挿入される。そして二重引用符で狭まれた "pink elephants" という文字列が一つの引数として扱われて、‘%s’ の位置に置かれる。（つまり、二重引用符で挟まれた文字列は、評価された時に、数値と同じ様にそれ自身を値として返すわけである。）

最後に、単なる数の計算ではなく、Ｓ式の中にＳ式を置き、‘%s’ で置き換えることで文章を作る例として、ちょっと複雑なものを挙げておこう。

```
(message "He saw %d %s"
         (- fill-column 34)
         (concat "red "
                 (substring
                  "The quick brown foxes jumped." 16 21)
                 " leaping."))
```

この例では、 message は３つの引数を取る。 "He saw %d %s" という文字列と、（- fill-column 32）というＳ式と、 concat という関数で始まるＳ式である。（- fill-column 32）を評価して得られた結果は ‘%d’ の位置に挿入され、 concat から始まるＳ式を評価して返される値は ‘%s’ の位置に挿入される。

私がこの式を評価したところ、エコー領域には "He saw 38 red foxes leaping." というメッセージが表示された。

(訳註：日本語の例も挙げておく。 Mule の内部では 2 バイトコードは 3 バイトで表現されるため一文字 3 ポイントの勘定になる。)

```
(message "彼は %d 匹の %s のを見た。 "
         (- fill-column 34)
         (concat "赤い"
                 (substring
                  "すばしこい茶色のきつねが跳んだ。 " 24 33)
                 "が飛び跳ねる "))
```

1.9 変数の値の設定

変数に値を格納する方法は幾つかある。一つの方法は、 set もしくは setq という関数を使うことである。また別の方法として、 letを使うというのもある。 (Section 3.6, “let”, p. 24, 参照). (このプロセスのことを専門用語では、変数に値をバインド (束縛) すると言う。)

この後のセクションでは set と setq の働きを説明するだけではなく、引数の渡され方の様子も述べることにする。

1.9.1 set の利用

シンボル flowers の値としてリスト '(rose violet daisy buttercup) をセットするために、次のＳ式の後にカーソルを持っていって C-x C-e とタイプして、この式を評価してみよう。

```
(set 'flowers '(rose violet daisy buttercup))
```

すると（rose violet daisy buttercup）というリストがエコー領域に表示されたはずだ。これは関数 set によって返された (*return* された) 値である。その副作用としてシンボル flowers が、このリストにバインドされる。つまり、シンボル flowers は、変数とみなすことが出来て、その値がこのリストであるようになったということである。(因みにこのプロセスは、値をセットするという Lisp インタプリタにとっては副作用でしかないことが、我々人間にとっては主効果たりうるということの説明になっている。Lisp interpreter はエラーが出ない限り値を返すはずである。しかし、副作用の方は、そのように意図しないと出ないものなのである。)

さっきの set を使ったＳ式を評価した後では、 flowers というシンボルを評価することが出来るようになっている。そして、たった今、我々がセットした値を返す。実際に次のシンボルの後にカーソルを持っていって C-x C-e とタイプしてみよう。

```
flowers
```

flowers を評価すると、エコー領域にリスト（rose violet daisy buttercup）が現れたはずだ。

もし間違えて頭に引用符を付けた 'flowers の方を評価したとすると、エコー領域に現れるのはこのシンボル flowers それ自身になる。引用符を付けたやつを次に挙げておくので、評価してみて欲しい。

```
'flowers
```

もう一つ注意すべきことは、 set を利用する時は、 set の引数には、それらを評価しようとする場合を除いて、必ず引用符を付ける必要があるということである。先の例の場合では、引数 flower 及び（rose violet daisy buttercup）のどちらも評価したくはなかったので、両方に引用符を付けている。(set の一番目の引数に引用符を付けなかった場合、まずまっ先にその引数が評価される。これを flowers に対してやったとして、もしこのシンボルがまだ値を持っていなかったとすると、'Symbol's value as variable is void' というエラーメッセージが表示される。一方、もし flowers がある値を返したとすると、 set はそいつにある値を設定しようとする。勿論こうなるのが正しい動作である状況もあるが、そういうことは稀である。)

1.9.2 setq の利用

実際問題として、大抵の場合は第一引数には引用符をつけることになるだろう。 set と引用符付きの第一引数の組み合わせというのは極めてよく使われるために、それに対して名前がついている。それが特殊形式 setq である。この特殊形式はほぼ set と同じなのだが、第一引数に自動的に引用符を付けてくれる所だけが違う。従ってあなた自身で引用符を付ける必要はない。また、そのことに加えて、 setq には、一つのＳ式の中で複数の異なる変数に対して各々に異なる値をセットすることが出来る、という便利さもある。

setq を使って carnivores という変数に '(lion tiger leopard) というリストをセットするには次のＳ式が使われる。

```
(setq carnivores '(lion tiger leopard))
```

これは set を使ったものと、第一引数に引用符が付いていないことを除き、全く同じである。('setq' の q は quote を意味している。) set を使うと上のＳ式は次のようになる。

```
(set 'carnivores '(lion tiger leopard))
```

また、 setq は異なる変数の各々に異なる値をセットすることも出来ると書いた。この場合には一番目の引数には二番目の引数の値がバインドされ、三番目の引数には四番目の引数の値がバインドされ、という風になる。例えば次のＳ式を使うとシンボル tree には樹木のリストを、シンボル herbivores には草食動物のリストをセットすることが出来る。

```
(setq trees '(pine fir oak maple)
      herbivores '(gazelle antelope zebra))
```

(このＳ式は一行で書いても構わないのだが、そうなるとページの大きさには合わなってしまう。人間が読む場合には、こういう形式の方が読みやすいことがお分りいただけると思う。)

(訳註：因みに、日本語なら次のようになる。)

```
(setq 樹木 '(松 モミ 樫 楓)
      草食動物 '(ガゼル レイヨウ シマウマ))
```

これまで何度か「セットする」という言葉を使ってきたけれども、 setとか setq の働きにはもう一つの捉え方がある。それは set や setq はそのシンボルを特定のリストへのポイントにする、というものだ。このような考え方は大変一般的であり、この後の章で、少なくとも一つ、名前の中にそのポインタを持つシンボルに出逢うことになる。そういう名前が付けられたのはそのシンボルに、ある値、より詳しくはリスト、がセットされているためである。もう一つの言い方では、そのシンボルはそのリストへのポインタに設定されているという風に表現出来る。

1.9.3 カウント

次の例は、 setq をカウンタとして利用する方法を説明してくれるものである。これは、プログラムの中で、ある部分が何回現われたかを数えてくれる。まず最初に変数を 0 に設定し、次にプログラムの中で、その部分が現われるごとに一つずつ足していくのである。そのためにはまずカウンタとなる変数一つに加えて二つのＳ式を用意する必要がある。初めにこのカウンタを 0 にセットするための setq の式と、評価されるごとにその変数を一ずつ増やすための setq の式である。

```
(setq counter 0)                ; これをイニシャライザと呼ぼう。

(setq counter (+ counter 1))    ; これはインクリメンタと呼ぶ。

counter                         ; これはカウンタ。
```

(';' に続く文はコメントである。 Section 3.2.1, "関数定義の変更", p. 21, 参照。)

もし最初のイニシャライザであるＳ式（setq counter 0）を評価してから三番目の counter というＳ式を評価したなら、エコー領域には数値 0 が表示される。次に二番目のインクリメンタのＳ式（setq counter (+ counter 1)）を評価すると、カウンタは一つ増える。従って、ここでもう一度 counter を評価するとエコー領域には数字 1 が表示される。その後も二番目のＳ式を評価するごとに値は増えていく。

インクリメンタ（setq counter (+ counter 1)）が評価されると Lisp インタプリタは最初にもっとも内部にあるリストを評価する。このリストを評価するために、まず counter, 次に 1 が評価される。変数 counterを評価すると、その時の値が返される。それが数 1 とともに +に渡されて足し合わされる。で、その和は内部のリストの値として setqに渡され、カウンタに新しい値が設定されることになる。以上のようにして、変数 counter の値が変更される。

1.10 まとめ

Lisp を学ぶことは、最初の道が急な坂であるような丘を登るようなものである。あなたが現在登っているのはもっとも急な部分である。先に進むに従って、より楽に進めるようになるはずだ。

まとめると

- Lisp のプログラムはリストもしくは単一のアトムからなるＳ式で出来ている。
- リストは 0 以上のアトムもしくはリストからなる。これらは空白文字で区切られており、括弧で囲まれている。リストの中身は空であっても良い。
- アトムは幾つかの文字からなるシンボルである。例えば forward-paragraph とか一つの文字からなる +、二重引用符に両側を挟まれた文字列からなる文字列、それに数値などがそうである。
- 数値は評価されるとそれ自身を返す。
- 二重引用符で挟まれた文字列を評価した時もそれ自身を返す。
- シンボルそのものを評価すると、その値が返される。
- リストを評価すると、 Lisp インタプリタはまずそのリストの一番目のシンボルを見にいき、そのシンボルにバインドされている関数定義を見る。そしてその定義の中にある命令を実行する。

- 引用符 ’ があると Lisp インタプリタはその後に続くＳ式を書かれたままの形で返し、二重引用符が無い場合のように評価したりはしないはしない。
- 引数は関数に渡される情報である。ある関数に渡される引数は、その関数を先頭に持つリストの残りの要素を評価したものにある。
- 関数は評価された時は (エラーが起きない限り) 常にある値を返す。また、それに加えて「副作用」と呼ばれる効果を生ずる。多くの場合、その関数を使う第一の目的はその副作用を生じさせることにある。

1.11 練習問題

幾つか単純な練習問題を挙げておく。

- 括弧に挟まれない適当なシンボルを評価することにより、エラーメッセージを発生させなさい。
- 1 ではなく 2 ずつ増えるカウンタを作りなさい。
- 評価された時にエコー領域に何かメッセージを表示するようなＳ式を書きなさい。

2 実際の評価の仕方

Emacs Lisp での関数の定義の書き方を学ぶ前に、ちょっと時間を取って、今まで書いてきた様々なＳ式を評価する方法について述べておこう。これらのＳ式は関数を最初の (そしてしばしば唯一の) 要素として持つリストである。幾つかのバッファに関連する関数は、単純であるし、興味深くもあるので、これらの関数から始めようと思う。また、別のセクションで、同じくバッファに関連する他のコードについて、それらがどのように書かれているか見てみるつもりだ。

Emacs Lisp 対して、カーソルを動かすとかスクリーンをスクロールするなどの編集コマンドを与える場合、常にあなたは (最初の要素が関数であるような) あるＳ式を評価している。これが *Emacs* の動作の基本である。

あなたが何かキーを押すと、そのことで Lisp インタプリタはあるＳ式を評価する。そういう風にしてあなたは求む結果を得るのである。たとえプレーンテキストをタイプしている時でも、 Emacs Lisp の関数を評価させているのである。この場合に呼ばれるのは `self-insert-command` であり、これは単にあなたがタイプした文字を挿入するものである。キーを押すことで評価されるような関数はインタラクティブ (*interactive*) な関数、あるいは、コマンド (*command*) と呼ばれる。関数をインタラクティブにする方法は関数の定義の書き方の章で説明することにする。 Section 3.3, “関数をインタラクティブにする”, p. 21, 参照。

キーボードコマンドをタイプするのに加えて、我々はもう一つＳ式を評価する方法を学んだのだった。それはカーソルをリストの終端に持っていって `C-x C-e` とタイプするやり方である。他のも同様にＳ式を評価する方法がある。それらについては他のセクションでその都度説明するつもりだ。

評価されるのに使われることに加えて、次の幾つかのセクションで出てくる関数は、それらそのものが元々重要である。これらの関数を学習することで、バッファとファイルの違いやバッファを切り替える方法、そしてその中での位置を定める方法についてよく理解出来るようになるだろう。

2.1 バッファの名前

`buffer-name` と `buffer-file-name` という二つの関数を見てみると、ファイルとバッファの違いがよく分るだろう。(`buffer-name`) というＳ式を評価すると、バッファの名前がエコー領域に表示される。一方、(`buffer-file-name`) を評価すると、バッファに関連付けられているファイルの名前がエコー領域に表示される。普通は (`buffer-name`) で返される名前は、それに関連付けられているファイルの名前と一致しており、(`buffer-file-name`) はそのファイルのフルパスを返す。

ファイルとバッファは異なる種類の物である。ファイルというのは計算機に記録されている情報であり、それを消去しない限りはずっと残っているものである。一方でバッファは Emacs の内部の情報であり、その時々のセッション (訳註：つまり、 Emacs を起動して終了するまで) が終わると (もしくはそのバッファを kill すると) 消え去ってしまうものである。普通はバッファはファイルからコピーした情報を保持している。このことを、バッファがそのファイルにビジット (*visit*) しているという。我々はこのコピーに対して編集などの仕事を行う。バッファに対する変更はそのバッファをセーブするまではファイルには反映されない。バッファをセーブすると、バッファの内容はファイルにコピーされて、そのまま記録されることになる。

もしこの文を GNU Emacs の Info で読んでいるのであれば、以下のＳ式の直後にカーソルを持っていって `C-x C-e` とタイプすることで、各々の式を評価することが出来る。

```
(buffer-name)

(buffer-file-name)
```

私がこれらの式を評価してみたところ、(`buffer-name`) を評価すると‘`"introduction.texinfo"`’という値が返され、(`buffer-file-name`) を評価すると‘`"/gnu/work/intro/introduction.texinfo"`’が返ってきた。前者がバッファの名前で、後者はそのファイル名である。(Ｓ式の中の括弧は、 Lisp インタプリタが `buffer-name` と `buffer-file-name` を関数として扱っていることを示している。もし括弧が無ければ、インタプリタはそのシンボルを変数として評価しようとしただろう。 詳しくは Section 1.7, “変数”, p. 6, を参照。)

ファイルとバッファは違うものであるにも関わらず、しばしばバッファのことをファイルと言ったり、また、その逆を言ったりする人は多い。実際、大抵の人は「私はファイルを編集している」といい、「私はファイルにセーブしようとしているバッファを編集している」などとは言ったりはしない。それでも殆どの場合、状況から言いたいことは通じてしまう。しかしながら計算機のプログラムを扱う場合は、その違いを心に留めておくことが重要だ。計算機は人間のように賢くはないのである。

ところで 'buffer' という言葉は衝突の際の衝撃を吸収するクッションを表わす言葉から来ている。初期の計算機では、バッファはファイルと計算機の中央演算処理装置 (CPU) の間のクッションの役目をしていた。ファイルを保存するドラムやテープと CPU は互いに全く異る装置であり、速度も全然違っている。バッファは各々が互いに効果的に作業出来るようにするものだった。その後、バッファは中継ぎとか、一時的な保管場所とかいう意味から変化して作業する場所を示すようになった。この変化は小さな港から巨大な町への変化と似ている。かつては単に船に積まれる貨物が一時的に保管される場所でしかなかった所が、ビジネスや文化の中心としての存在になってしまったのである。

さて、全てのバッファが必ず何らかのファイルに関連付けられているわけではない。例えば `emacs` という単独のコマンドで、ファイルを指定せずに Emacs のセッションを始めた場合、Emacs は '`*scratch*`' というバッファの画面からスタートする。このバッファはどのファイルにもビジットしていない。同様に '`*Help*`' というバッファもどのファイルにも関連付けられてはいない。

もし '`*scratch*`' バッファに切り替えて (`buffer-name`) とタイプし、その直後にカーソルを置いて `C-x C-e` とタイプしてそのＳ式を評価したなら、`"*scratch*"` という名前が返り、エコー領域に表示される。つまり、バッファの名前は `"*scratch*"` である。しかし、同じことを (`buffer-file-name`) でやったなら、エコー領域には `nil` と表示される。nil は「無 (nothing)」を表わすラテン語の単語である。この場合は '`*scratch*`' バッファがどのファイルにも関連付けられていないことを意味している。(Lisp では `nil` は「偽 (false)」という意味や、空のリストである () の同意語としても使われる。)

ついでに言っておくと、もしあなたが、'`*scratch*`' バッファにいて、あるＳ式が返す値をエコー領域ではなく '`*scratch*`' バッファそのものに表示したい場合は、`C-x C-e` の代わりに `C-u C-x C-e` とタイプすれば良い。こうすると、値はＳ式のすぐ後に表示される。即ちバッファには次のように表示される。

```
(buffer-name)"*scratch*"
```

これは Info の中では実行出来ない。というのも Info バッファは読み出しのみ (read-only) であり、勝手にバッファの中身を修正したりすることは出来ないからである。しかし、編集可能なバッファであれば、どのバッファであろうと実行出来る。コードや文書 (例えばこのマニュアルなんか) を書いている時には、この機能は大変便利である。

2.2 バッファの獲得

`buffer-name` という関数はバッファの名前を返す。従って、バッファそのものを取り出すには、別の関数が必要になる。これは `current-buffer` という関数である。コードの中でこの関数を使うと、バッファそのものが得られる。

名前と、その名前がついているオブジェクトないしは実体とは、互いに異なっている。あなたはあなたの名前とは違う。あなたは、他の人からその名前で呼ばれる一人の人間である。もしあなたが George と話したいと頼んだ時に、'`G`', '`e`', '`o`', '`r`', '`g`', '`e`' という文字の書かれたカードを渡されたとしたら、あなたは面白いとは思うかもしれないが、本来の要求は満たされない。あなたは名前と話したいというのではなく、その名前を持つ人間と話したいのだから。バッファもそれと同じだ。スクラッチバッファの名前は '`*scratch*`' である。しかし、その名前はバッファそのものではない。ということで、バッファそのものを取り出すときは、`current-buffer` のような関数が必要なわけである。

しかしながら、事情はもう少し複雑である。例えば `current-buffer` をすぐ後で実験するように、Ｓ式の中で評価したとする。そうすると、バッファの中身ではなく、バッファの名前のみが表示される。Emacs がこのように動作するのには二つの理由がある。そのバッファは何千行もあるかもしれない — これは手軽に表示出来るというものではない。そして他のバッファで、名前は違っても中身は同じものがあるかもしれない。それらを区別するのは、大変大切なことである。

さて、これがこの関数を含むＳ式である。

```
(current-buffer)
```

もし、ごく普通にこのＳ式を評価したとすると、エコー領域には‘#<buffer *info*>’のように表示される。このように特殊な様式で表示することで、バッファの名前ではなく、そのバッファ自身が返されたことを示しているのである。

ついでに言うと、プログラムの中には数値やシンボルをタイプすることは出来るが、表示されているバッファについてはそんなことは出来ない。バッファを取り込む唯一の方法は current-buffer のような関数を使うことなのである。

関連する関数に other-buffer というのがある。これは、今、現在いるバッファ、つまりカレントバッファを除いてもっとも最近選択したバッファを返す。もし直前に‘*scratch*’バッファに行って帰ってきたとしたら、other-buffer は、このバッファを返す。

このことは次のＳ式を評価することで確かめられる。

```
(other-buffer)
```

エコー領域には‘#<buffer *scratch*>’もしくは、あなたが最近行って帰ってきたバッファの名前が表示されたはずだ。

2.3 バッファ間の移動

other-buffer という関数はこれを引数とする関数から呼ばれた場合には実際にそのバッファの中身を提供する。これは other-buffer と switch-to-buffer を使って他のバッファに切り替えることで確かめられる。

が、その前に、 switch-to-buffer という関数を簡単に紹介しておこう。Info バッファから（buffer-name）を評価するために‘*scratch*’バッファに行って戻ってくる際に、多分あなたはまず C-x b とタイプし、そしてミニバッファにどのバッファに移るか尋ねるプロンプトが出た時に‘*scratch*’とタイプしたのではないだろうか。この時のキーストローク C-x b は Lisp インタプリタに対してインタラクティブに Emacs Lisp の関数 switch-to-buffer を評価させているのである。前にも言ったように Emacs はこのようにして動作している。また別のキーストロークはまた別の関数を呼び出したり走らせたりする。例えば C-f は forward-char を呼び出すし、 M-e は forward-sentenceを呼び出す、といった感じである。

Ｓ式の中で switch-to-buffer と書いて、そいつに移るべきバッファを与えると、 C-x b と全く同じようにしてバッファを切り替えることが出来る。

ここにそのＳ式を挙げておく。

```
(switch-to-buffer (other-buffer))
```

switch-to-buffer というシンボルはこのリストの最初の要素であるから、 Lisp インタプリタは、こいつを関数として扱い、それに定義されている命令を実行する。しかし、その前に、内部に括弧に挟まれた other-buffer があることを発見し、こちらを先に解釈する。 other-buffer は、このリストの最初の (そしてこの場合は唯一の) 要素であるから、 Lisp インタプリタはこの関数を走らせる。するとこいつは別のバッファを返す。次にインタプリタは返ってきたバッファを switch-to-buffer に引数として渡して、この関数を走らせる。そうして Emacs は他のバッファに切り替わる。もし、この文書を Info で読んでいるなら早速試してみよう。上のＳ式を評価するのである。 (戻ってくるには、 C-x b RET とタイプする。)

後の方のセクションに出てくるプログラミングの例では switch-to-buffer よりも set-buffer という関数の方をより多く見かけるだろう。これは計算機のプログラムと人間の違いによる。人間は目を持ち今作業している計算機の端末の上にそのバッファを見たいと思うものである。これはいうまでもなく当たり前のことである。しかしながら、プログラムは目を持たない。プログラムがバッファを扱っている時には、バッファはスクリーンに見えていなくたって良いのである。

switch-to-buffer は人間のために設計されており、二つのことを行う。一つは Emacs が注意を向けているバッファを切り替えることであり、もう一つは画面に見えているバッファを切り替えることである。一方、set-buffer の方は一つの仕事しかしない。単に計算機が注意を向けているバッファを別のバッファに切り替えるだけである。画面に見えているウィンドウはそのままだ (勿論、普通はそのコマンドが終了するまでは何も起こらない。)

ここでまた別の専門用語を紹介しよう。呼び出す (*call*) という言葉である。最初のシンボルが関数であるリストを評価すると、その関数が呼び出される。(call される。) この時の呼び出すという単語の使い方は、「呼び出す」ことで実体としての関数が何かをやってくれることから来ている — 本物の配管工を「呼び出す」と水洩れを直してくれるのと一緒である。

2.4 バッファのサイズとポイントの位置

最後に、簡単な関数を幾つか見てみよう。`buffer-size`, `point`, `point-min`, そして `point-max` である。これらはバッファのサイズとその中でのポイントの位置の情報を与えてくれるものである。

`buffer-size` という関数は現在のバッファの大きさを教えてくれる。つまり、そのバッファの中の文字の数を返すのである。

```
(buffer-size)
```

これをいつものように、S式の最後にカーソルを持っていって `C-x C-e`とタイプすることで評価してみよう。

Emacs の中では現在のカーソルの位置はポイントと呼ばれる。(`point`)というS式はバッファの始まりからポイントまでの文字数を返す。それによって現在のカーソルの場所が分る。

次のS式をこのバッファの中でいつも通りに評価すればポイントまでの文字数が分る。

```
(point)
```

これを書いているときには、`point` の値は 121905 であった。この `point` という関数は、このマニュアルの後の方に出てくる幾つかの例の中では頻繁に使われる。

勿論ポイントの値はバッファの中での位置に依存している。次の場所でポイントを評価すれば、その値はさっきよりも大きくなっているはずである。

```
(point)
```

私の場合この場所でのポイントの値は 122376 であった。これは両方のS式の間に (空白も含めて) 471 文字あることを示している。(訳註： Mule では、内部的にマルチバイト文字の先頭にリーディングキャラクタを付けて扱うので、例えば 2 バイト文字一つは 3 ポイントに数えられる。)

`point-min` という関数はちょっと `point` に似ているが、こちらはカレントバッファ中のポイントの値の中で許される最小の値を返す。この値はナローイング (*narrowing*) が有効になっていない限り 1 という数値である。(ナローイングというのは、あなた自身もしくはプログラムの操作をバッファのある部分に制限してしまうある仕組である。詳しくは Chapter 6, “ナローイングとワイドニング”, p. 53, を参照のこと。) 同様に、`point-max` というのはカレントバッファ中のポイントの値の中で許される最大の値を返す。

2.5 練習問題

あなたが作業しているファイルを一つ開いて、その中央辺りまで行きなさい。そして、そのバッファ名、ファイル名、長さ、及びファイルの中での位置を調べなさい。

3 関数定義の書き方

Lisp インタプリタがリストを評価する場合、まずリストの最初のシンボルが関数定義を持っているかどうか、別の言い方をするとシンボルから関数定義へのポインタがあるかを見にいく。もしそうであれば、計算機はその定義に含まれる命令を実行する。関数定義を持つシンボルのことを単に関数と言う。(ただし、正確には定義が関数なのであり、シンボルはそれを指しているだけである。)

全ての関数は C 言語で書かれた少数の プリミティブ (*primitive*) な関数を除いて、別の関数を使って定義されている。関数を書く場合にはそれを Emacs Lisp を使って書き、他の関数を部品として使うことになる。あなたが使う関数はそれ自身 (あなたが書いたものを含めて) Emacs Lisp で書かれたものかもしれないし、C で書かれたプリミティブなものかもしれない。プリミティブな関数も Emacs Lisp で書かれているのものと全く同じように使われ、同じように振舞う。これらが C で書かれているのは、十分な能力を持ち、C を走らせることが出来るどの計算機でも簡単に GNU Emacs が動くようにするためである。

再度強調しておくけれども、Emacs Lisp のコードを書く場合、C で書かれた関数を使う場合と Emacs Lisp で書かれた関数を使う場合とで区別をつける必要は全くない。違いは全く無視して構わない。私がこの区別のことを書いたのは、単に知っていた方が面白いからである。実際、調査でもしない限り、既に書かれている関数が Emacs Lisp で書かれているか C で書かれているかは判断出来ない。

3.1 特殊形式 defun

Lisp では `mark-whole-buffer` のようなシンボルには、その関数が呼ばれた時に計算機が何をすべきかを教えるコードが付随している。このコードのことを 関数定義 (*function definition*) といい、シンボル `defun` で始まるＳ式を評価した時に作られる。(これは *define function* の略である。) `defun` は通常とは別の引数の評価の仕方をするために、特殊形式 (*special form*) と呼ばれる。

この後に続く幾つかのセクションでは、例えば `mark-whole-buffer` のように Emacs のソースコードの中から関数定義を見ていくことにする。が、このセクションではもっと単純な関数定義を説明し、それがどんなものなのかを、まず理解してもらうことにしよう。簡単な例にするために算数についての例にする。算数を使った例が嫌いな人達もいるとは思うが、あなたがそうだとしてもがっくりくる必要はない。この入門書の残りに出てくる殆どのコードには算数とか数学は出てこない。出てくる例は、大体が何がしかの意味でテキストに関わるものである。

関数定義は `defun` という単語に続けて、次のような最大五つの部分を加えたものである。

1. 関数定義が付けられるシンボルの名前。
2. 関数に渡される引数のリスト。もし一つも引数をつけない場合、ここは空リスト `()` にする。
3. 関数を説明する文章。(技術的には省略可能であるが、書くことが強く望まれている。)
4. 省略可能。この関数をインタラクティブにする、つまり `M-x` に続けて関数名をタイプするか、適当なキーないしはキーコードをタイプすることで使うことが出来るようにするためのＳ式を書く。
5. 計算機に何をすべきかを命令するコードを書く。即ち、関数定義の本体である。

この関数定義の五つの部分の各々にスロットを割当てて、次のようなテンプレートにして整理しておくと役に立つだろう。

```
(defun function-name (arguments...)
  "optional-documentation..."
  (interactive argument-passing-info) ; 省略可能
  body...)
```

一つの例として、引数に 7 をかける関数のコードを挙げる。(この例はインタラクティブではない。 Section 3.3, “関数をインタラクティブにする”, p. 21, を見よ。)

```
(defun multiply-by-seven (number)
  "Multiply NUMBER by seven."
  (* 7 number))
```

この定義は括弧と defun というシンボルで始まり、関数の名前が続いている。

関数の名前の次にはこの関数に渡される引数のリストが来る。このリストは引数リスト (*argument list*) と呼ばれる。今の場合、このリストの要素は number というシンボル一つだけである。この関数が使われる時には、このシンボルは関数の引数として使われる値にバインドされる。

引数の名前として number という単語を選ぶ代わりに他の名前を選んでも良い。例えば multiplicand という単語でも良かった。私が 'number' という単語を選んだのは、このスロットにどういう種類の値が要求されるかが分るからである。しかし、このスロットに置かれる値がこの関数の働きの中でどういう役割を果たすかを示すという意味で 'multiplicand' という単語を選ぶのも悪くなかった。 foogle としても構わないが、これは良い選択だとは思えない。これでは何のことか分らない。どういう名前にするかはプログラマ次第だが、関数の意味を明確にするようなものが選ばれるべきである。

実の所、引数リストの中のシンボルにはどんな名前でも選ぶことが出来る。たとえ、他の関数の中で使われているシンボル名であっても良いのである。引数リストの中で使われる名前は、その特定の定義の中だけでしか使われないプライベートなものであるからだ。この定義の中では、他のどんな関数定義の中にある同じ名前のシンボルとも異なる実体を指すのである。例えば、あなたが家族の間では 'Shorty' というニックネームをつけられているとしよう。この場合、あなたの家族の誰かが 'Shorty' という場合には、それはあなたのことを指している。しかしあなたの家族以外の所、例えば映画の中などでは 'Shorty' というのは別の誰かを指している。引数リストの中の名前はその関数の中だけで通用するものなので、この関数の本体の中でそのシンボルの値をどう変えたとしても外での値には何の影響もないというわけだ。似たような効果が let を使ったＳ式の場合でも見られる。(Section 3.6, "let", p. 24, 参照。)

引数リストの次には、関数についての説明である、説明文字列が来る。これはC-h f に続けてその関数の名前をタイプした時に表示される内容である。ついでに言っておくと、あなたが説明文字列を書く場合には、最初の一行は完結した一つの文にすべきである。というのも、 apropos のような幾つかのコマンドは複数行に渡って書かれた説明文の文字列のうちの最初の一行だけしか表示しないからである。また、二行目も書く場合はこれをインデントすべきではない。何故なら、 C-h f (describe-function) を使ったときに変な表示のされかたをするからである。説明文字列は省略可能ではあるが、大変役に立つものなので、関数を書く際にはなるべく書くようにすべきである。

さっきの例での三行目は関数定義の本体をなしている。 (普通の関数定義は勿論これよりもっと長い。) 今の場合、この本体は（* 7 number）というリストであり、 *number* の値に 7 をかけることを定めている。 (Emacs Lisp では * は掛け算を表わす関数である。同様に + は足し算を表わす。)

multiply-by-seven という関数を使う場合、引数 number はあなたが実際に使いたい値に評価される。次の例では multiply-by-seven の使い方を示している。ただし、まだこれを評価しないように！

```
(multiply-by-seven 3)
```

次のセクションで詳しく述べるが、今の例ではシンボル number に 3 という値が与えられた、ないしは「バインドされた」わけである。 number は関数定義の括弧の中にあるが、 multiply-by-seven という関数に渡される引数は括弧の外にあることに注意しておこう。関数定義の中に括弧があるのは、計算機が引数のリストが何処で終わり、残りの部分が何処から始まるかを知るためにあるのである。

この例を評価すると多分エラーメッセージが出るだろう。 (試してみよう！) これは、我々は確かに関数定義を書いたのだが、まだその定義を計算機に伝えていない — つまり、まだ Emacs にこの関数定義をインストール (もしくはロード) していない — ためである。インストールの仕方については次のセクションで説明する。

3.2 関数定義のインストール

もし Emacs の Info の中でこれを読んでいるのなら、先に関数の定義を評価し、次に（multiply-by-seven 3）を評価することで、 multiply-by-seven という関数を試してみることが出来る。この下に関数定義の写しを書いておくので、その直後にカーソルを持っていって C-x C-e とタイプしよう。そうすると、エコー領域に multiply-by-sevenと表示される。 (これは関数定義を評価した時に返す値はその関数の名前であることを示している。) 同時に、こうすることで関数定義がインストールされる。

```
(defun multiply-by-seven (number)
  "Multiply NUMBER by seven."
  (* 7 number))
```

この defun を評価することで、たった今 Emacs に multiply-by-seven がインストールされた。今ではこの関数は forward-word やあなたが編集に使う他の関数と全く同様、Emacs の一部である。(multiply-by-seven はあなたが Emacs を終了するまでインストールされたままになっている。このコードを Emacs を起動するたびに再ロードする方法については、Section 3.5, “コードをずっとインストールしておくには”, p. 24, を見よ。)

multiply-by-seven をインストールしたことでどうなったかは、次の例を評価することで見ることが出来る。カーソルを次のＳ式の直後に持っていって C-x C-e とタイプしてよう。エコー領域には 21 という数字が表示されたはずだ。

```
(multiply-by-seven 3)
```

もし望むなら、C-h f (describe-function) に続けてこの関数の名前 multiply-by-seven をタイプすることで、この関数定義の説明を読むことも出来る。そうすると、‘*Help*’というウィンドウに次のように表示される。

```
multiply-by-seven: Multiply NUMBER by seven.
```

(スクリーンを元の一つのウィンドウに戻すには、C-x 1 とタイプすれば良い。)

3.2.1 関数定義の変更

multiply-by-seven の中のコードを変更したい場合は、単にそれを書き直すだけである。古いものを新しいバージョンに置き換えるには、その関数定義をもう一度評価すれば良い。これが Emacs の中のコードを修正する方法である。極めて単純だ。

例として、multiply-by-seven という関数を 7 をかけるのではなく、その数自身を 7 回加えるものに変更することが出来る。結果としては同じ答えが得られるけれども、途中の道筋は違っている。同時にコードの中にコメントを加えよう。これは Emacs には無視されるけれども、人間が読む場合には便利だなあとか解りやすいとか思うはずだ。今回は、これが二番目のバージョンであるというコメントを書いておくことにする。

```
(defun multiply-by-seven (number) ; 二番目のバージョン
  "Multiply NUMBER by seven."
  (+ number number number number number number number))
```

コメントはセミコロン‘;’の後に続いている。Lisp では、セミコロンの後に続く全てのものはコメントである。行の終わりがコメントの終わりになる。二行以上に渡ってコメントを書きたい場合は各々の行をセミコロンで始める。

Section 16.3, “‘.emacs’ の書き方”, p. 142, 及び Section “Comments” in *The GNU Emacs Lisp Reference Manual*, にコメントについて、より詳しく説明されている。

今のバージョンの multiply-by-seven 関数をインストールするには最初のものをインストールした時と全く同じようにして評価してやれば良い。つまり、カーソルを最後の括弧のすぐ後に持ってきて C-x C-e とタイプすれば良いのである。

まとめると、Emacs でコードを書く方法は次の通りである。まずは関数を書き、インストールし、テストしてみる。で、不具合を修正したり拡張したりして再度インストールするという具合だ。

3.3 関数をインタラクティブにする

関数をインタラクティブなものにするには、特殊形式 interactive で始まるリストを説明文字列のすぐ後に置けばよい。インタラクティブな関数はユーザーが M-x に続けて関数の名前をタイプすることで呼び出すことが出来る。あるいは、その関数にバインドしたキーをタイプしてもよい。例えば、C-n とタイプすると next-line が呼ばれるし、C-x h とタイプすると mark-whole-buffer が呼ばれる。

面白いことに、インタラクティブな関数をインタラクティブに呼び出すと、返される値は自動的にはエコー領域に表示されない。これはインタラクティブな関数はしばしば、単語分や一行分前に進んだりするなどの副作用を目的に呼ぶのであって、返される値のことは気にしないからである。もしキーをタイプするごとにエコー領域に値が表示されたら、随分と鬱陶しいことだろう。

特殊形式 interactive を使いつつエコー領域に値を表示する方法について説明するために、 multiply-by-seven のインタラクティブバージョンを作ってみよう。

これがそのコードである。

```
(defun multiply-by-seven (number) ; インタラクティブバージョン
  "Multiply NUMBER by seven."
  (interactive "p")
  (message "The result is %d" (* 7 number)))
```

このコードはカーソルをコードのすぐ後に持っていって C-x C-e とタイプすることでインストール出来る。ちゃんと、この関数の名前がエコー領域に表示されただろうか。次からこのコードを使うには、まず C-u に続けて数をタイプし、その後に M-x multiply-by-seven とタイプして RETを押せば良い。エコー領域に、'The result is ...' という文に続いて計算結果が表示されるはずだ。

もっと一般的に言うと、このような関数は次の二通りの内どちらかの方法で呼び出すことが出来る。

1. まず渡されるべき数を含む前置引数をタイプする。ついで M-x と関数名をタイプする、例えば C-u 3 M-x forward-sentence、とする。
2. あるいは、その関数にバインドされたキーないしはキーコードをタイプする。例えば C-u 3 M-e。

今挙げた二つの例では、どちらの場合もポイントが 3 つの文だけ前方に移動する。(この関数を例として扱った理由は、 multiply-by-seven はどのキーにもバインドされてはいないので、キーバインディングの例としては使えないからである。)

(コマンドをキーにバインドする方法については Section 16.7, "幾つかのキーバインディング", p. 145, を参照のこと。)

前置引数をインタラクティブな関数に渡すには、 M-3 M-e のように META キーに続けて数をタイプする方法と、 C-u 3 M-e のように C-u に続けて数をタイプする方法とがある。(もし C-u の後に何も数をタイプしなかった場合、デフォルトでは 4 が渡される。)

3.3.1 インタラクティブな multiply-by-seven

インタラクティブバージョンの multiplyo-by-seven の中の特殊形式 interactive の使い方と、関数 message を見てみよう。もう一度この関数の定義を書いておく。

```
(defun multiply-by-seven (number) ; インタラクティブバージョン
  "Multiply NUMBER by seven."
  (interactive "p")
  (message "The result is %d" (* 7 number)))
```

この関数の中ではS式 (interactive "p") は二つの要素からなるリストである。 "p" は Emacs に対し、この関数に前置引数を渡し、その値を関数の引数として使うことを知らせるためのものである。

引数は数になる。これは number というシンボルが

```
(message "The result is %d" (* 7 number))
```

という行の中である数にバインドされるという意味である。例えば、前置引数が 5 だとすると、 Lisp インタプリタはこの行を次のように評価することになる。

```
(message "The result is %d" (* 7 5))
```

(もし、この文章を GNU Emacs の中で読んでいるなら、このＳ式をあなた自身で評価することが出来る。) 最初にインタプリタは内部にあるリストを評価する。今の場合は (* 7 5) である。これは値 35 を返す。次にその外側のリストが評価される。つまり、二番目及び残りの要素の値が message という関数に渡される。

前に見たように、 message は特に一行のメッセージをユーザに送るための Emacs Lisp の関数であった。 (Section 1.8.5, "関数 message", p. 10, 参照。) 手短に言うと、関数 message は最初の引数をエコー領域に表示するのだが、 '%d', '%s', 及び '%c' だけは例外で、これらの制御文字列の内のどれかが現れた時は、残りの二番目以降の引数を順に見に行き、その値をこれらがある場所に挿入して表示する。

multiply-by-seven という関数では、制御文字列として '%d' が使われ、これは数を要求する。今の場合ならその値は (* 7 5) を評価して返された 35 という数である。その結果、 '%d' の場所には 35 が表示され、メッセージは 'The result is 35' となるわけである。

(ここで注意だが、 multiply-by-seven という関数を呼んだ場合、メッセージは引用符無しで表示される。しかし message をそのまま呼んだ場合には、二重引用符で囲まれている。これは message が返す値そのものは message が先頭にあるＳ式を評価した場合にエコー領域に表示されるもので、これには引用符がついているのだが、これが他の関数の中に埋め込まれて使用された場合、表示されるものは message が副作用として出力するもので、これには二重引用符がついていないからである。)

3.4 interactive の他のオプション

上の例で multiply-by-seven は interactive の引数として "p"を使っている。この引数がある場合、 Emacs はあなたが、 C-uに続けて数をタイプするか、 Meta キーに続けて数をタイプするかして、このコマンドに引数として数を渡すものと解釈する。 Emacs はこのような interactive の引数を 20 種類以上用意している。大抵の場合、これらのオプションの内にあなたが望む通りの情報を関数に渡すものが一つか二つはあるはずだ。 (Section "Code Characters for interactive" in *The GNU Emacs Lisp Reference Manual*, 参照。)

例えば 'r' という文字の場合、 Emacs はその関数にその時のリージョンの最初と最後の位置 (その時のポイントとマークの値) を二つの別々の引数として渡す。使い方は例えばこんな風だ。

```
(interactive "r")
```

あるいは、 'B' であれば、 Emacs はバッファの名前を聞いてきて、入力した値をその関数に渡す。もう少し詳しく言うと、 Emacs はミニバッファにプロンプトを出して、ユーザーに名前を要求する。この時のプロンプトには 'B'に続く値が使われるので、例えば、 "BAppend to buffer: " とか書いておく。 Emacs はただ単にプロンプトを出すだけではなくて、判断出来る分だけの入力があれば、 TAB キーを押すことで補完までしてくれる。

関数に二つ以上の引数を渡したい場合、 interactive に続けて複数の文字列を付け加えることで、各々の引数に情報を渡すことが出来る。この場合その情報は、各々の引数に interactive に書いたのと同じ順序で渡される。付け加える文字列は、各々の部分を '\n', 即ち改行コードで区切る。例えば "BAppend to buffer: " に続けて '\n' と 'r' を書いたりする。こうすると、 Emacs はプロンプトでバッファ名を要求すると同時にポイントとマークの値もその関数に渡してくれる — 三つの引数全てを渡してくれるのである。

この場合の関数定義は次のような形式になる。 buffer, start, 及び、 end は interactive がバッファとその時のリージョンの始まりと終わりをバインドするシンボルである。

```
(defun name-of-function (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  body-of-function...)
```

(プロンプトのコロンの後の空白は、プロンプトを出す時の見栄えをよくするためのものである。 append-to-buffer という関数はまさにこのようになっている。 Section 4.4, "append-to-buffer の定義", p. 38, 参照。)

あるいは、もし上に挙げたような特定の文字による引数の与え方があなたの目的に合わない場合、あなた自身の引数をリストとして `interactive` に渡すことも可能である。 Section "Using `Interactive`" in *The GNU Emacs Lisp Reference Manual*, にこの上級テクニックについてより詳しいことが書かれている。

3.5 コードをずっとインストールしておくには

関数定義を評価することで一旦その関数をインストールすると、 Emacs を終了するまでその関数はインストールされたままになっている。一方、次に Emacs の新しいセッションを開始した時は、その関数定義を再度評価するまでその関数はインストールされない。

ある時点で、 Emacs の新しいセッションを始める時に自動的にインストールしたいと思うかもしれない。方法は幾つかある。

- もし、自分自身で書いたコードがあれば、その関数定義のコードをあなたの初期化ファイル '`.emacs`' の中に書き込む。そうすると Emacs を起動した時にこの '`.emacs`' というファイルが自動的に評価されて、その中に書かれた全ての関数がインストールされる。 Chapter 16, "'`.emacs`' ファイル", p. 141, 参照。
- あるいは、インストールしたい関数の定義を一つのファイル、もしくは関数ごとに複数のファイルに書いておき、 `load` という関数を使って Emacs にそれを評価させてそれらの関数をインストールするという方法もある。 Section 16.8, "ファイルのロード", p. 146, 参照。
- また、あなたのいるサイト全体でそのコードを使いたい場合は、普通はその関数を '`site-init.el`' 呼ばれるファイルに書いておく。このファイルは Emacs を作成する時に自動的にロードされる。こうすることで、あなたの計算機を使う全ての人がその関数を利用出来るようになる。 (Emacs distribution の中の '`INSTALL`' ファイルを参照のこと。)

最後に、もしあなたが Emacs を使う全ての人が欲しくなるようなコードを書いたなら、そのコピーをネットワーク上にポストしたり、 Free Software Foundation に送ることが出来る。 (そうする場合には、どうかポストする前にコードに copy left の注意書きを添付して欲しい。) コピーを Free Software Foundation に送った場合、次の Emacs のリリース時にはそれを含めて配布されるかもしれない。大体において、ここ数年はこのような寄付によって Emacs が成長してきたのである。

3.6 `let`

`let` 式は、 Lisp では多くの関数定義の中で必要となる特殊形式である。非常によく使われるものなので、このセクションの中で `let` について説明することにする。

`let` は、シンボルに値をバインドする際に、 Lisp インタプリタがその関数以外の関数の中で使われている同じ名前の変数と混同しないようにするために使用される。何故こんな関数が必要なのかということを納得するために、次の様な状況を考えよう。あなたは自分自身の家を持っていて、それを「家」と呼んでいるとする。例えば、「そろそろ家にもペンキ塗りが必要だなあ」という風に使うわけである。しかし、あなたが友人宅を訪問し、そこのホストの人が「家」と言った場合、大抵それはあなたの家ではなく彼の家のことを指している。即ち、同じ「家」という言葉で言及されてはいても別のものなわけである。もし彼が彼自身の家のことを言っていて、あなたがあなた自身の家を思い浮かべていたなら混乱してしまうことであろう。同じことが Lisp についても言える。ある関数で使われている変数と同じ名前の変数が他の関数でも使われていて、しかもその二つが同じ値を持つことを期待されてはいない時である。

`let` という特殊形式を使うことでこのような混乱を防ぐことが出来る。 `let` は ローカル変数 (*local variable*) と呼ばれるものを発生させる。これは、その `let` 式の外にある同じ名前の変数からは隔離されている変数である。訪問先のホストの人が「家」という時は彼は彼自身の家を指しているのであって、あなたの家を指しているのではないのと似たようなものだ。 (引数リストの中のシンボルも同じような働きをする。 Section 3.1, "特殊形式 `defun`", p. 19, を見よ。)

`let` 式によって発生したローカル変数の値が保持されているのは、その `let` 式の中だけである。 (そして、その中のＳ式はその `let` 式の中だけで呼ばれる。) 従って、ローカル変数はその `let`式の外には全く影響を与えない。

let は一度に複数の変数を発生させることが出来る。また、 letは各々の変数に初期値を設定する。あなたが指定すればその値になるし、そうしなければ nil が設定される。(専門用語では、このことを「その変数に値をバインドする」と言う。) let は変数を発生させ、それに値をバインドした後、本体のコードを実行し、本体の中の最後のＳ式の値を返す。(実行 (Execute) するというのは、リストを評価するという意味の専門用語である。これはこの単語の「実質的な効果を与える (to give practical effect to)」という意味での使い方から来ている (*Oxford English Dictionary*)。あなたはある動作を引き起こすためにＳ式を評価しているのだから、この場合「実行」するというのは評価するのと同義であろう。)

3.6.1 let 式の構成部分

let というＳ式は三つの部分からなるリストである。最初の部分はシンボル let である。二番目の部分は 変数リスト (*varlist*) と呼ばれるリストであり、各々の要素はシンボルそのものであるか、最初の要素がシンボルである二つの要素からなるリストであるかのどちらかであり、三番目の部分は let 式の本体である。本体部分は大抵、一つないしは複数のリストからなる。

let 式のテンプレートは次のように書ける。

(let *varlist body...*)

変数リストの中のシンボルは特殊形式 let 式によって初期値を設定される変数である。単独のシンボルそのものの場合は、初期値 nil が設定される。また、最初の要素がシンボルであるような二つの要素からなるリストの場合、その最初のシンボルに対して Lisp インタプリタが二番目の要素を評価した時に返される値が設定される。

というわけで、変数リストは (thread (needles 3)) という感じの式になる。この場合だと、 let 式の中で Emacs はシンボル threadを初期値 nil に、シンボル needles を初期値 3 にバインドする。

let 式を書く場合にすることは、適切なＳ式を let 式のテンプレートの中に置くことである。

もし、変数リストが二つ要素のリストからなる場合、といっても大抵はそうなのだが、 let 式のテンプレートは次のようになる。

```
(let ((variable value)
      (variable value)
      ...)
      body...)
```

3.6.2 let 式の例

次のＳ式は二つの変数 zebra と tiger を発生させ、それに対して初期値を与えている。 let 式の本体は関数 message を呼び出すリストである。

```
(let ((zebra 'stripes)
      (tiger 'fierce))
  (message "One kind of animal has %s and another is %s."
           zebra tiger))
```

ここで、変数リストは ((zebra 'stripes) (tiger 'fierce)) である。

二つの変数は zebra と tiger である。各々の変数は二つの要素からなるリストの最初の要素であり、その値はそのリストの二番目の要素になっている。 Emacs は変数リストの中で zebra を stripes にバインドし、また、 tiger を fierce にバインドしている。この場合、値は両方とも引用符のついたシンボルである。値は勿論、他のリストや文字列であっても構わない。 let 式の本体は変数を含むリストの後に来る。今の場合は本体は関数 message を使ったリストであり、エコー領域に文字列を表示する。

いつものようにカーソルを最後の括弧の直後に置いて C-x C-e とタイプすることで上の例を評価することが出来る。そうすると、エコー領域に次のように表示されるだろう。

```
"One kind of animal has stripes and another is fierce."
```

以前見たように、 message 関数は最初の引数を ‘%s’ を除いて表示する。今の場合、最初の ‘%s’ の所には変数 zebra の値が表示され、二番目の ‘%s’ の所には変数 tiger の値が表示される。

3.6.3 let 式の変数宣言の中で初期値を設定しなかった場合

もし、 let 式の中で変数に特定の初期値をバインドしなかったとすると、それらの値は自動的に nil にバインドされる。次の例を見てみよう。

```
(let ((birch 3)
      pine
      fir
      (oak 'some))
  (message
   "Here are %d variables with %s, %s, and %s value."
   birch pine fir oak))
```

ここで、変数リストは ((birch 3) pine fir (oak 'some)) である。

このＳ式をいつもの通りに評価したなら、エコー領域には次のように表示されるだろう。

```
"Here are 3 variables with nil, nil, and some value."
```

この場合には Emacs はシンボル birch に 3 という数をバインドし、シンボル pine と fir には nil をバインドし、更にシンボル oak には some をバインドしている。

let 式の最初の部分で、変数 pine と fir は単独のアトムであり、括弧で囲まれてはいないことに注意しよう。これは、これらの変数を nil, 即ち空リストにバインドするためである。しかし、 oak の方は、 some にバインドするので、リスト（oak 'some）の一部となっている。同様に、 birch も 3 にバインドするので、この数と一緒のリストに入っている。(数値は評価された時にそれ自身の値を返すので、引用符は必要ない。また、 message の中にこの数を表示する際は、‘%s’ ではなくて、‘%d’ を使っている。) そして、これら四つの変数がひとつのグループとしてリストの中に入り、 let 式の本体と区別されているわけである。

3.7 特殊形式 if

defun と let に続く三つ目の特殊形式は、条件分岐 (conditional) if である。この特殊形式は計算機になんらかの判断をさせる時に使われる。 if を使わなくても関数定義を書くことも出来るが、頻繁に使うものであるし、重要でもあるので、ここに含めることにする。これは例えば、 beginning-of-buffer という関数の中で使われている。

if の背後にある基本的な考え方は「もし(if)、テストした結果が真ならば(then) Ｓ式を評価する」というものだ。テストした結果が真でなければ、Ｓ式は評価はされない。例えば、「もし、暖かくて太陽が出ていたなら、ビーチへ行こう。」というような判断をするようなものだ。

Lisp で書かれる if 式では、‘then’ という単語は使われない。テストと実行は if を最初の要素とするリストの二番目と三番目の要素である。にも関わらず、 if 式のテスト部分は *if-part* と呼ばれ、二番目の引数は *then-part* と呼ばれることが多い。

また、 if 式が書かれる場合、真か偽かのテストは普通、 if と同じ行に書かれる。しかし、テストが真であった場合に実行される “then-part” は二行目以降に書かれる。こうすることで、 if 式がより読みやすいものになる。

```
(if true-or-false-test
    action-to-carry-out-if-test-is-true)
```

真偽テスト (true-or-false-test) の部分は List インタプリタにより評価されるＳ式である。

次にいつものようにして実行出来る例を挙げておく。テスト部分は 5 が 4 よりも大きいかどうかを判断するものだ。そうであれば、‘5 is greater than 4!’というメッセージが表示される。

```
(if (> 5 4) ; if-part
    (message "5 is greater than 4!"))  ; then-part
```

(関数 > は最初の引数が二番目の引数よりも大きいかどうかを判断し、そうであれば真を返すものである。)

勿論、実際に使う場合には、 if 式のテスト部分は常に (> 5 4) に固定されていたりはしない。その代わりに、少なくとも一つの変数が、前もって分らない値にバインドされる。(もし、前もって値が分っているなら、テストする必要などない！)

例えば、その値は関数定義の引数にバインドされたりする。次の関数定義では、関数に渡される値は動物の性格である。もし、 characteristic にバインドされる値が fierce であれば、'It's a tiger!' というメッセージが表示される。そうでなければ、 nil が返される。

```
(defun type-of-animal (characteristic)
  "Print message in echo area depending on CHARACTERISTIC.  If the
CHARACTERISTIC is the symbol 'fierce', then warn of a tiger."
  (if (equal characteristic 'fierce)
      (message "It's a tiger!")))
```

もし、これを GNU Emacs の中で読んでいるなら、上の関数定義を評価することで、 Emacs にインストールすることが出来る。そうすると、次の二つのＳ式を評価して結果を見ることも出来るようになる。

```
(type-of-animal 'fierce)

(type-of-animal 'zebra)
```

(type-of-animal 'fierce) を評価すると、エコー領域には "It's a tiger" と表示されるはずだ。一方、(type-of-animal 'zebra) を評価すると、 nil が表示される。

3.7.1 関数 type-of-animal の詳細

type-of-animal 関数を詳しく見てみよう。

type-of-animal の関数定義は二つのテンプレートのスロットを埋めることで書かれている。一つは全体の関数定義のテンプレートで、もう一つは if 式のテンプレートである。

インタラクティブでない関数の関数定義のテンプレートは次の通りである。

```
(defun name-of-function (argument-list)
  "documentation..."
  body...)
```

このテンプレートに当てはめると、各部分は次のようになる。

```
(defun type-of-animal (characteristic)
  "Print message in echo area depending on CHARACTERISTIC.  If the
CHARACTERISTIC is the symbol 'fierce', then warn of a tiger."
  body: the if expression)
```

今の場合、関数の名前は type-of-animal である。これは引数を一つ取る。引数リストの後に、複数行に渡る説明文字列が続く。例の中にもちゃんと説明文字列を入れるのは、全ての関数定義に説明文字列を入れるのが良い習慣であるからである。関数定義の本体部分は if 式からなっている。

if 式のテンプレートは次の通りである。

```
(if true-or-false-test
    action-to-carry-out-if-the-test-returns-true)
```

type-of-animal 関数の場合、 if 式の実際のコードは次のようになる。

```
(if true-or-false-test
    action-to-carry-out-if-the-test-returns-true)
```

type-of-animal 関数では、 if に関する実際のコードは次のようになっている。

```
(if (equal characteristic 'fierce)
    (message "It's a tiger!")))
```

ここで、真偽テストの部分は次のＳ式である。

```
(equal characteristic 'fierce)
```

Lisp では、 equal は最初の引数と二番目の引数が等しいかどうかを判定する関数である。二番目の引数は引用符つきのシンボル 'fierce であり、一番目の引数は characteristic というシンボルの値 — つまり、この関数に渡される引数の値である。

先に行った type-of-animal の最初の実行では、 fierce が type-of-animal に渡した。 fierce は fierce と等しいので、Ｓ式（equal characteristic 'fierce）は真の値を返す。この場合は、 if は二番目の引数、つまり if 式の then-part である（message "It's tiger!"）を評価する。

一方、 type-of-animal の二番目の実行では zebra を引数として渡した。 zebra は fierce ではないので、 then-part は評価されず、 if 式は nil を返す。

3.8 If–then–else 式

if 式は、オプションとして *else-part* と呼ばれる三番目の引き数を持つことが出来る。これは真偽テストが偽を返した場合のためのものである。この場合は if 式の二番目の引数である then-part は評価されず、代わりに三番目の引数である else-part が評価される。このことは、曇りの日なんかに、「もし、晴れて暖かかったらビーチに行こう。そうでなかったら、本でも読むか！」というふうに代わりの選択肢を考えるのに当てはめてみれば納得出来るだろう。

"else" という単語は Lisp のコードの中には出てこない。単に、 else-part は if 式の中の then-part の次にくる要素というだけである。実際に Lisp のコードを書く場合、 else-part は新しく改行してから then-part よりは少ないインデントで書き始めるのが普通である。

```
(if true-or-false-test
    action-to-carry-out-if-the-test-returns-true)
  action-to-carry-out-if-the-test-returns-false)
```

例えば、次の if 式はいつも通りに評価すると、'4 is not greater than 5!' というメッセージを表示するものである。

```
(if (> 4 5) ; if-part
    (message "5 is greater than 4!")  ; then-part
  (message "4 is not greater than 5!")) ; else-part
```

異なるレベルのインデントをすることで、 then-part と else-part の区別がしやすくなることに注意しよう。(GNU Emacs は if 式を自動的に正しくインデントするコマンドを幾つか備えている。 Section 1.1.3, "GNU Emacs によるリストのタイプの支援", p. 2, を参照のこと。)

type-of-animal の if 式に新しく else-part の部分を追加するだけで、この関数を拡張することが出来る。

次のバージョンの type-of-animal の関数定義を評価してインストールすれば、その後の、異なる引数を渡すような二つのＳ式を評価して、結果を見ることも出来るようになる。

```
(defun type-of-animal (characteristic) ; 二番目のバージョン
  "Print message in echo area depending on CHARACTERISTIC.  If the
CHARACTERISTIC is the symbol ‘fierce’, then warn of a tiger; else say
it’s not fierce."
  (if (equal characteristic ’fierce)
      (message "It’s a tiger!")
    (message "It’s not fierce!")))

(type-of-animal ’fierce)

(type-of-animal ’zebra)
```

(type-of-animal ’fierce) を評価すると、エコー領域には "It’s a tiger" と表示されるはずだ。一方、(type-of-animal ’zebra) を評価すると、 "It’s not fierce!" と表示される。

(勿論、 *characteristic* が ferocious であったなら、 "It’s not fiece!" と表示されるだろう。そして、これは間違いである。コードを書く場合は、 if によってこのような引数がテストされる可能性を考慮に入れて、それに応じてプログラムを書かなければならない。

3.9 Lisp における真と偽

if 式の中の真偽テストについては、触れておかなければならない重要な側面がある。今までは、真とか偽とかいう言葉を、述語の値としてあたかも新しい Lisp のオブジェクトとして話してきた。しかし実際は「偽」というのは単に我々が親しんできた良き友 nil なのである。そして、その他は全て — どんなものであっても —「真」である。

真であるか否かテストされるＳ式は、評価した結果が nil 以外の値であれば *true* と判断される。別の言い方をすると、テストの結果は返された値が 47 のような数値や、 "hello" というような文字列、あるいは flowers というような (nil 以外の) シンボルや、リストであったり、はたまたバッファであったりする場合でも真であると解釈されるわけである。

このことを説明する前に、 nil について説明せねばならないだろう。

Lisp ではシンボル nil は二つの意味を持つ。一つ目は空リストである。そして二つ目は偽であり、真偽テストの結果が間違いであった場合に返る値である。 nil は空リストとして () のように書いても良いし、 nil と書いてもよい。 Lisp にとっては、 () と nil は全く同じものである。しかし、人間にとっては nil は偽を表わすのに使い、 () は空リストを表わすのに使うのが普通だろう。

Lisp では nil以外 — 即ち空リスト以外 — の全ての値は真と解釈される。これは、評価した時に空リスト以外の何かを返すものは、 if 式のテストで真と判断されることを意味する。例えば、もしテストの部分に数値を置いたとすると、 Lisp では数値は評価された時には自分自身を返すように出来ているのだから、結果としてこの場合の if 式でのテストは真となる。テストの結果が偽となるのはＳ式を評価して nil、即ち、空リストが返る時だけである。

次に挙げる二つのＳ式を評価してみることで、このことを確かめることが出来る。

最初の例では、 if 式のテストの結果、数字の 4 が評価され、その数自身が返される。結果として then-part が評価され、エコー領域に ‘true’ が表示される。二番目の例としては、 nil は偽を表わすので 結果として else-part が評価され、エコー領域に ‘false’ と表示される。

```
(if 4
    ’true
  ’false)

(if nil
    ’true
  ’false)
```

ついでながら、もしテストで真を返すような適当な値が見つからなければ、 Lisp インタプリタは真を表わすためにシンボル `t` を返す。例えば、（`> 5 4`）は評価された場合 `t` を返す。これもいつものようにして確認出来る。

```
(> 5 4)
```

一方、この関数は偽である場合は `nil` を返す。

```
(> 4 5)
```

3.10 save-excursion

関数 `save-excursion` は、この章で説明する 4 番目の、そして最後の特殊形式である。

編集用の Emacs Lisp プログラムでは、関数 `save-excursion` は大変よく使用されるものである。これは、ポイントとマークの位置をセーブし、関数の本体を実行し、そしてその際にポイントとマークの位置が変わったなら、それを元の位置に戻す。ポイントとマークが意外な場所に動いて、驚いたり、困惑したりするのを防ぐことがこの関数の主な目的である。

しかしながら、 `save-excursion` について議論する前に GNU Emacs において、ポイントとマークがどういうものであるかを復習しておくのが良いだろう。 *Point* は現在のカーソルの位置である。カーソルが何処にあろうと、それがポイントである。より正確に言うと、カーソルが文字の上にあるような端末では、ポイントは、その文字の左端の位置にある。 Emacs Lisp ではポイントは整数である。バッファの最初の文字は 1 であり、二番目の文字は 2 であり... といった感じだ。関数 `point` は現在の位置を数として返す。各々のバッファがそれ自身のポイントの位置を持っている。

(訳註：より具体的に言うと、カーソルがバッファの先頭の文字の上にある場合には、ポイントの値は 1 になることになる。また、二文字目以降は、日本語等の場合は事情が変わる。例えば Mule では 2 バイトコードは 3 バイトで表現されるので、ポイントの数は一文字あたり 3 ずつずれる勘定になる。)

mark は、ポイントとはまた別の、バッファにおけるある位置を指し示すものである。その値は `C-SPC` (`set-mark-command`) で設定される。マークが設定されている場合、 `C-x C-x` (`exchange-point-and-mark`) を使ってカーソルをマークにジャンプさせ、以前ポイントがあった場所にマークを設定することが出来る。更に、もし他のマークが既にあった場合、以前のマークはマークリングに保存される。多くのマークをこうして保存することが出来る。 `C-u C-SPC` とタイプすることで、以前保存されたマークに順にさかのぼって行くことが出来る。

バッファの中で、ポイントとマークに挟まれた部分を リージョン (*region*) と呼ぶ。リージョンに関するコマンドは沢山ある。例えば、 `center-region`、 `count-lines-region`、 `kill-region`、そして `print-region` なんかがそうである。

特殊形式 `save-excursion` はポイントとマークの位置を保存し、この特殊形式内の本体部分のコードが Emacs Lisp によって評価された後に、その位置を復元する。 こうして、例えばポイントがあるテキストの先頭にあり、あるコードがその位置をバッファの最後に移動してしまったとしても、本体部分のＳ式が評価され終わった後に `save-excursion` がポイントを元の位置に戻してくれる。

Emacs ではユーザーが予期しない所でも、しばしば内部的にポイントの位置を動かしていることがある。例えば `count-lines-region` なんかもポイントの位置を移動している。予期しない、また (ユーザーの立場からみて) 不必要な移動によってユーザーが煩わされるの防ぐために、 `save-excursion` は頻繁に使われる。 `save-excursion` を使うのは家の中を住み心地の良いものにしておくことと一緒である。

家の中がきちんと整理されているかどうかを確認するために、 `save-excursion` は、たとえ、その内部のコードがうまく動かない場合にも (専門用語を使ってより正確に言うと、「異常終了した場合にも」)、ちゃんとポイントとマークの位置を復元してくれる。この機能は大変役に立つ。

`save-excursion` は単にポイントとマークの値を記録するだけでなく、カントバッファも保持していて、それを復元してくれる。これは、バッファを変更するようなコードを書いても、 `save-excursion` が元のバッファに戻してくれることを意味している。実際、 `append-to-buffer` の中では、 `save-excursion` はその目的で使われている。 (Section 4.4, “`append-to-buffer` の定義”, p. 38, 参照。)

3.10.1 save-excursion 式のテンプレート

save-excursion のテンプレートは極めて単純である。

```
(save-excursion
  body...)
```

関数の本体は一つないしは複数のＳ式であり、それらが順に Lisp インタプリタによって評価されていく。もし本体に二つ以上のＳ式があったとすると、 save-excursion という関数自体の値としては、それらの内最後のＳ式の値が返される。他のＳ式は単に副作用として評価されるのである。 save-excursion 自体は (ポイントとマークを復元するという) 副作用だけのために用いられる。

より詳しく書くと、 save-excursion 式のテンプレートは次のように書ける。

```
(save-excursion
  first-expression-in-body
  second-expression-in-body
  third-expression-in-body
   ...
  last-expression-in-body)
```

ここでＳ式は、シンボルそのものか、もしくはリストである。

Emacs Lisp のコードでは、 save-excursion 式は、 let 式の中でよく使われる。次のような感じだ。

```
(let varlist
  (save-excursion
    body...))
```

3.11 復習

ここまでの幾つかの章で、かなりの数の関数と特殊形式を紹介してきた。ここでそれらを簡単にまとめておくと同時に、そこでは触れなかった類似の関数なんかも紹介しておくことにする。

eval-last-sexp
: 現在のポイントの位置より前方の最後のＳ式を評価する。通常はＳ式が返す値はエコー領域に表示されるが、引数つきで呼び出された時だけはカレントバッファに表示する。普通は、 C-x C-e にバインドされている。

defun
: 関数を定義する。この特殊形式は最大 5 つの部分からなる。関数の名前、関数に渡される引数のリスト、説明文字列、オプションのインタラクティブ宣言、そして関数定義の本体である。
例)

```
(defun back-to-indentation ()
  "Point to first visible character on line."
  (interactive)
  (beginning-of-line 1)
  (skip-chars-forward " \t"))
```

interactive
: インタプリタに対してその関数がインタラクティブに使用出来ることを宣言する。この特殊形式の後に関数に渡す引数に関する情報として複数の部分からなる文字列が続くことがある。これらの部分の中にはインタプリタに対してその情報を求めるプロンプトを出すよう要求するものもある。この文字列の中では各々の部分が改行コード '\n' によって区切られている。

  よく使われるコード文字には、次のようなものがある。

  b　　現在存在するバッファ。

  f　　現在存在するファイル。

p 数値である 前置引数。(‘p’ が小文字であることに注意。)

r 二つの数値引数としてのポイントとマークを、小さい方を先に並べたもの。このコード文字のみが二つの引数の列を特定する。(他は一つだけ。)

コード文字の完全なリストについては次を参照。 Section “Code Characters for ‘interactive’” in *The GNU Emacs Lisp Reference Manual.*

let 変数リストが let の本体部分のコードのみに対して使われることを宣言し、かつ、それらの変数に nil ないしは指定した初期値を与える。そして、 let 式の本体部分にあるＳ式を評価し、その中の最後のＳ式を評価した値を返す。 let 式の本体の内部では、 Lisp インタプリタはその let 式の外部にある同じ名前の変数は見ない。

例)

```
(let ((foo (buffer-name))
      (bar (buffer-size)))
  (message
   "This buffer is %s and has %d characters."
   foo bar))
```

save-excursion
この特殊形式の本体部分を評価する直前のポイントとマークの位置、及びカレントバッファを保存する。そして本体実行後、ポイントとマークの位置及びバッファを復元する。

例)

```
(message "We are %d characters into this buffer."
         (- (point)
            (save-excursion
              (goto-char (point-min)) (point))))
```

if この関数の最初の引数を評価する。そして、もしその値が真であれば、二番目の引数を評価し、そうでない場合、三番目の引数があればそれを評価する。

特殊形式 if は条件分岐 (*conditional*) と呼ばれる。 Emacs Lisp には他にも条件分岐があるが、 if はその中でも多分もっともよく使われるものであろう。

例)

```
(if (string= (int-to-string 19)
             (substring (emacs-version) 10 12))
    (message "This is version 19 Emacs")
  (message "This is not version 19 Emacs"))
```

equal
eq 二つのオブジェクトが同じであるかどうかをテストする。 equal の方は、もし両者が同じ構造と中身を持つ場合に真を返す。もう一方の eq の方は、引数が両方とも実際に同じオブジェクトである時のみ真を返す。

<
>
<=
>= 関数 < は、最初の引数が二番目の引数よりも小さいかどうかをテストする。これと対をなす関数 > は、最初の引数が二番目よりも大きいかどうかをテストする。同様に、 <= は最初の引数が二番目の引数以下であるかどうかを、>= は最初の引数が二番目より以上であるかどうかをテストする。どの場合でも引数は両方とも数値でなければいけない。

message エコー領域にメッセージを表示する。このメッセージは一行のみでなければならない。最初の引数は ‘%s’, ‘%d’, あるいは ‘%c’ を中に含む文字列であり、これらはその後に続く引数におきかえられる。‘%s’ の部分に入る引数は文字列かシンボルでなければならない。また、‘%d’ の部分に入る引数は、数でなければならない。‘%c’ の所に入る引数も数でなければならないが、こちらはそれに対応する ascii コードの文字が表示される。

`setq`
`set` 関数 `setq` は最初の引数に二番目の引数の値をセットする。最初の引数は `setq` により、自動的に引用符が付けられる。続けて引数のペアを書いた場合、それらについても同じことをする。もう一方の関数 `set` の方は、二つの引数しか取ることが出来ない。また、その引数は両方ともまず最初に評価され、その後、二番目の引数を評価して返された値を最初の引数を評価して返された値にセットする。

`buffer-name`
引数は持たず、バッファ名前を文字列として返す。

`buffer-file-name`
引数は持たず、バッファがビジットしているファイルの名前を返す。

`current-buffer`
Emacs がアクティブであるようなバッファの中身を返す。これはスクリーンに見えているバッファとは限らない。

`other-buffer`
(`other-buffer` に引数として渡したバッファ及び、カレントバッファを除いて) もっとも最近選択したバッファの中身を返す。

`switch-to-buffer`
Emacs にとってアクティブなバッファを選択し、カレントウィンドウに表示してユーザーが見えるようにする。普通は `C-x b` にバインドされている。

`set-buffer`
これからプログラムを走らせるバッファに Emacs を切り替える。ただし、表示しているウィンドウはそのままである。

`buffer-size`
カレントバッファの文字数を返す。

`point` 現在のカーソル位置の値をそのバッファの最初の位置からの文字数として返す。

`point-min`
カレントバッファの中で許される最小のポイントの値を返す。ナローイングが有効でない場合は 1 である。

`point-max`
カレントバッファの中で許される最大のポイントの値を返す。ナローイングが有効でない場合はバッファの最後の位置に一致する。

3.12 練習問題

- 引数の値を二倍するインタラクティブでない関数を書きなさい。また、それをインタラクティブにしなさい。
- 現在の `fill-column` の値が関数に渡される引数よりも大きいかどうか判定し、もしそうなら適当なメッセージを表示するような関数を書きなさい。

4 バッファに関する幾つかの関数

この章では、GNU Emacs で使われている関数を、幾つか詳しく見てみることにする。こういうことは、“walk-through” と呼ばれる。これらの関数は Lisp コードの例として扱うのであるが、最初の単純化された例を除いて、決して「机上」のものではない。実際に GNU Emacs で使われている現実のコードである。これらの関数定義から多くのことを学ぶことが出来る。この章で説明する関数は、皆バッファに関連したものである。他のものについては後で学ぶことにしよう。

4.1 情報の探し方

この walk-through の中では、新しく出て来た関数はその都度、時には細かく、時には簡単に、説明することにする。出て来た関数に興味を持った場合は、`C-h f` に続けてその関数名をタイプして `RET` を押せば、どんな Emacs Lisp 関数についても何時でも、完全な説明文を見ることが出来る。同様に変数についても `C-h v` に続けてその変数名 (と `RET`) を押せば、その変数の完全な説明文を見ることが出来る。

また、もしその関数の元のソースを見たくなったら、関数 `find-tags` を使ってその関数に飛ぶことが出来る。まず `M-.` とタイプし (即ち、`META` キーとピリオドキーを同時に押すか、または `ESC` キーを押してからピリオドキーを押し), 出てきたプロンプトに対してソースコードを見たい関数の名前、例えば `mark-whole-buffer` 等、をタイプする。すると、Emacs はバッファを切り替えて、その関数のソースコードをスクリーンに表示してくれる。元のバッファに戻るには、`C-x b RET` とタイプすれば良い。

あなたの使っている Emacs のデフォルトの初期値いかんでは ‘tags table’ を指定する必要があるかもしれない。これは ‘`TAGS`’ と呼ばれるファイルである。‘`emacs/src`’ ディレクトリにあるファイルを指定することになる場合が殆どであろう。その場合 `M-x visit-tags-table` コマンドを使って、

‘`/usr/local/lib/emacs/19.23/src/TAGS`’

などのようにパス名を指定する。 Section “Tag Tables” in *The GNU Emacs Manual*, 参照。また、自分自身のものを作成する方法については Section 12.5, “自分自身の ‘TAGS’ ファイルの作成”, p. 105, を見よ。

Emacs Lisp に慣れてしまってからは、ソースコードを眺めるために頻繁に `find-tags` を使うことになるだろう。そして自分自身の ‘`TAGS`’ テーブルを作ることになるはずだ。

ついでに付け加えておくと、Lisp コードを含むファイルは便宜上ライブラリ (`library`) と呼ばれる。このメタファは、一般的なライブラリではなく、例えば法律とか工学とかのような特定分野のライブラリから来ている。各々のライブラリないしはファイルには特定のトピックや動作に関する関数が含まれている。例えば、‘`abbrev.el`’ には省略等のショートカットキーを扱う関数が、また ‘`help.el`’ にはオンラインヘルプを扱う関数が収められている、といった感じである。(時には複数のライブラリが一つの動作に関するコードを提供していることもある。電子メールを読むためのコードを提供する ‘`rmail...`’ ファイルなんかがそうだ。) The GNU Emacs Manual を見れば、「`C-h p` コマンドによって標準的な Emacs Lisp ライブラリをトピックをキーワードにして検索することが出来る」と言った記述を見つけることが出来る。

4.2 簡略版 beginning-of-buffer の定義

まず `beginning-of-buffer` という関数から始めるのが良いであろう。この関数には既に慣れているだろうし、理解するのも易しいからだ。インタラクティブなコマンドとして使われるものであろうと、`beginning-of-buffer` はカーソルをバッファの先頭に移動させ、以前の位置にマークを置く。この関数は大抵は `M-<` にバインドされている。

この章では、この関数の短縮されたバージョンを説明することで、これがどういうふうに使われることが多いのかを示そう。この短縮版も上に書いたような動作をするが複雑なオプションは付いていない。完全版については別のセクションで説明することにする。 (Section 5.3, “`beginning-of-buffer` の完全な定義”, p. 47, 参照。)

コードを見る前に、どんな関数定義が含まれているかを考えてみよう。まず、関数をインタラクティブにするための、即ち `M-x beginning-of-buffer` とタイプしたり、あるいは `M-<` といったキーコードをタイプすることで呼び出せるようにするためのＳ式を含んでいなければならない。また、バッファの元の位置にマークするためのコードも必要だ。更にカーソルをバッファの先頭に移動するコードも入っていなくてはならない。

これがこの関数の短縮版の完全なテキストである。

```
(defun simplified-beginning-of-buffer ()
  "Move point to the beginning of the buffer;
leave mark at previous position."
  (interactive)
  (push-mark)
  (goto-char (point-min)))
```

他の関数定義と同様に、この定義も特殊形式 defun に続く五つの部分からなっている。

1. 名前: この場合は simplified-beginning-of-buffer。
2. 引数リスト: この場合は空リスト ()。
3. 説明文字列。
4. インタラクティブにするためのＳ式。
5. 本体。

この関数定義では、引数リストは空である。これは、この関数が引数を必要としないことを意味する。(もっとも完全版の方の定義を見れば、そちらにはオプションの引数を渡すことが出来ることが分る。)

インタラクティブにするためのＳ式は、Emacs にこの関数がインタラクティブに使われることを伝えるための物である。今の場合 interactive は引数を持たない。というのも simplified-beginning-of-buffer は引数を必要としないからである。

関数の本体部分は次の二行からなる。

```
(push-mark)
(goto-char (point-min))
```

一行目は（push-mark）というＳ式である。このＳ式が Lisp インタプリタによって評価されると、カーソルが何処にあろうと、その位置にマークが設定される。マークの位置はマークリングに保存される。

二行目は（goto-char（point-min））である。このＳ式はカーソルをバッファ内での最小ポイント、即ちバッファの先頭に移動する。(あるいは、もしナローイングがかかっていれば、アクセス出来る範囲内での最初の位置に移動する。Chapter 6, “ナローイングとワイドニング”, p. 53.)

push-mark コマンドによって、（goto-char（point-min））でカーソルがバッファの先頭に飛ぶ前に、そのカーソルの位置にマークがセットされる。結果として、もし元の位置に戻ろうと思えば、 C-x C-x によって元の位置に戻ることが出来る。

これがこの関数定義の全てである！

もし、このようなコードを読んでいて goto-char のようによく知らない関数に出くわしたとする。その場合には describe-function コマンドを使ってこれがどんな関数かを見ることが出来る。このコマンドを使うには、まず C-h f とタイプし、次に知りたい関数の名前をタイプしてから RETを押せば良い。 describe-function コマンドは ‘*Help*’ ウィンドウにその関数の説明文字列を表示してくれる。例えば goto-char の説明文は次の通りである。

```
One arg, a number.  Set point to that number.
Beginning of buffer is position (point-min),
end is (point-max).
```

(describe-function はプロンプトを出す時にデフォルトの値としてその時のカーソルの直前にあるシンボルの値を設定する。従って、カーソルをその関数の直後に持っていって C-h f RET とタイプすることも出来る。)

end-of-buffer 関数についても beginning-of-buffer とほぼ同じである。ただし、関数の本体部分で、(goto-char（point-min）) の部分が（goto-char（point-max））となっている。

4.3 mark-whole-buffer の定義

関数 mark-whole-buffer も、関数 simplified-beginning-of-buffer と同じくらい簡単に理解出来る。ただし、今回は単純化したものではなく完全な関数を見ることにする。

mark-whole-buffer 関数は beginning-of-buffer 関数ほどには頻繁には使われない。しかし、同じくらい有用である。この関数はポイントをバッファの先頭に、マークをバッファの最後に置くことで、バッファ全体をリージョンとして指定する。また、大抵は C-x h にバインドされている。

この関数の完全なコードは次の通りである。

```
(defun mark-whole-buffer ()
  "Put point at beginning and mark at end of buffer."
  (interactive)
  (push-mark (point))
  (push-mark (point-max))
  (goto-char (point-min)))
```

他の全ての関数と同様、 mark-whole-buffer も関数定義のテンプレートにきちんと当てはまっている。このテンプレートは次のようなものだった。

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

この関数の動作であるが、関数名は mark-whole-buffer である。引数リストが空リスト '()' になっているので、この関数は引数を必要としないことが分る。次に説明文字列が来ている。

次の行は (interactive) である。これをつけると Emacs はこの関数をインタラクティブに使えるものとみなす。ここら辺のことは前節の simplified-beginning-of-buffer 関数の所でも説明した通りである。

4.3.1 mark-whole-buffer の本体

関数 mark-whole-buffer の本体は次の三行からなる。

```
(push-mark (point))
(push-mark (point-max))
(goto-char (point-min))
```

この三行のうち最初の行はＳ式 (push-mark (point)) である。

この行は関数 simplified-beginning-of-buffer の本体の最初の行の (push-mark) と全く同じ働きをする。どちらの場合にも Lisp インタプリタはカーソルの現在の位置にマークを設定する。

私には何故 mark-whole-buffer 関数では (push-mark (point))と書かれ、 beginning-of-buffer 関数では (push-mark) と書かれているか分らない。多分、このコードを書いた人は (push-mark) の引数は省略可能で、引数を受け取らない場合、デフォルトでは自動的に現在のポイントの位置にマークが設定されるということを知らなかったんだろうと思う。或いはこのＳ式が次の行と同じような構造になるように書いたのかもしれない。いずれにしてもこの行によって Emacs はポイントの位置を決定し、その位置にマークを設定する。

mark-whole-buffer の次の行は (push-mark (point-max)) である。このＳ式は、バッファの中でポイントの数が最大の位置にマークを設定する。これは大抵はバッファの最後である。(もし、バッファがナローイングされていれば、バッファの中のアクセス可能な範囲での最後尾になる。ナローイングについての詳細は Chapter 6, “ナローイングとワイドニング”, p. 53, 参照。)

マークを設定されると、現在のポイントの位置に設定されていた以前のマークは解除される。しかし、 Emacs は最近設定されたマークの位置と同様、その位置を記憶している。もし戻りたければ、 C-u C-SPC を二度タイプすることで、その位置に戻ることが出来る。

さて、いよいよ最後の行の (goto-char (point-min)) である。これは beginning-of-buffer と全く同じように書かれている。このＳ式はカーソルをバッファの中でポイントが最小の位置に移動させる。つまり、バッファの先頭 (もしくはアクセス可能な範囲での先頭) に移動させるのである。その結果、ポイントはバッファの先頭に移動し、バッファの最後尾にマークが設定される。このようにして、バッファ全体がリージョンになる。

4.4 append-to-buffer の定義

append-to-buffer コマンドも mark-whole-buffer コマンドと同じくらい単純な構造をしている。このコマンドがすることはカレントバッファのリージョン (つまり、バッファ中のポイントとマークある部分) を指定したバッファに複写することである。

append-to-buffer コマンドは、リージョンを複写するのに関数 insert-buffer-substring を使う。この関数は名前から察っせられる通り、バッファのある部分から文字からなる文字列、即ち「部分文字列」を取り出して、それを他のバッファに挿入する。関数 append-to-buffer の大部分は insert-buffer-substringがうまく動作するような状態に設定することに関するものである。このコードはテキストが写されるバッファとそのコピー元となるリージョンの両方の指定が必要となる。この関数の完全なコードは次の通りである。

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region.
It is inserted into that buffer before its point.

When calling from a program, give three arguments:
a buffer or the name of one, and two character numbers
specifying the portion of the current buffer to be copied."
  (interactive "BAppend to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (insert-buffer-substring oldbuf start end))))
```

この関数は一連のテンプレートが埋められたものと見做すと理解しやすい。

最も外側のテンプレートは、関数定義のものである。今の場合、次のようになっている (幾つか既に埋めてあるスロットもある)。

```
(defun append-to-buffer (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  body...)
```

最初の一行は関数の名前とその三つの引数を含んでいる。引数はテキストが複写される行き先のバッファ及び、複写元のリージョンを指定する start と end である。

関数の次の部分には簡潔で完全な説明が記されている。

4.4.1 インタラクティブ式 append-to-buffer

append-to-buffer という関数はインタラクティブに使われる関数なので、 interactive 式が必要である。(interactive について復習するには、 Section 3.3, “関数をインタラクティブにする”, p. 21, を見よ。)

```
(interactive "BAppend to buffer: \nr")
```

このＳ式は二重引用符に囲まれた引数を持ち、その引数は ‘\n’ によって分けられた二つの部分からなっている。

一つ目の部分は ‘BAppend to buffer:’ である。ここで、 ‘B’ は Emacs に対し、この関数に渡すバッファ名を尋ねるように要求している。これによって Emacs はミニバッファに ‘B’ に続く文字列 — 今の場合は ‘Append

to buffer: ’— からなるプロンプトを出して、ユ - ザーに名前を入力するよう促す。 Emacs はこの関数の引数リストにある変数 bufferに、そこで指定されたバッファをバインドするのである。

改行コード ‘\n’ は引数の最初の部分と二つ目の部分を分けるために用いられる。‘\n’ の次には ‘r’ が続いているが、これは関数の引数リストの中の buffer に続く二つのシンボル (つまり、 start と end) にポイントとマークの値をバインドすることを示している。

4.4.2 append-to-buffer の本体

関数 append-to-buffer の本体部分は let から始まっている。

以前見たように (Section 3.6, “let”, p. 24, を参照) let 式の目的は、 let 式内部だけで使われる変数を一つないしは複数作り、それに初期値を与えることである。これは、そのような変数は let 式外部の同じ名前の変数と混同されることがない、ということを意味している。

let 式の部分をアウトラインにして append-to-buffer のテンプレートを書いてみる。これによって let 式がどういうふうに関数に組み込まれているかが分る。

```
(defun append-to-buffer (buffer start end)
  "documentation..."
  (interactive "BAppend to buffer: \nr")
  (let ((variable value))
       body...)
```

let 式は三つの要素を持っている。

1. シンボル let。
2. 変数リストの中身。今の場合は二つの要素からなるリスト（*variable value*）が一つあるだけ。
3. let 式の本体。

関数 append-to-buffer の中では変数リストは次のようになっている。

```
(oldbuf (current-buffer))
```

let 式のこの部分では、変数 oldbuf が (current-buffer) を評価して返された値にバインドされる。変数 oldbuf は現在作業しているバッファの記録のために使われるものである。

変数リストの要素は一組の括弧に囲まれている。これによって Lisp インタプリタは変数リストと let 式の本体を区別出来るのである。その結果として変数リストの中の、二つの要素からなるリストであるような要素は更に一組の括弧によって囲まれることになる。ということで、この行は次のようになる。

```
(let ((oldbuf (current-buffer)))
  ... )
```

ひょっとすると oldbuf の前に二つの括弧があることに驚くかもしれないが、 oldbuf の前の最初の括弧は変数リストの区切りを示すものであり、次の括弧は二つの要素からなるリスト (oldbuf (current-buffer)) の最初の括弧であることさえ分れば問題ないだろう。

4.4.3 append-to-buffer の中の save-excursion

append-to-buffer の中の let 式の本体は save-excursion 式からなっている。

関数 save-excursion はポイントとマークの位置を保存し、 save-excursion の本体部分のＳ式の実行が完了した時点でそれらを元の位置に戻す。更に save-excursion は元々のバッファが何であったかも憶えていて、そのバッファに戻してくれる。 save-excursion は append-to-buffer の中でこのように使われている。

ついでだが、ここで Lisp の関数の書式としては、通常複数の行に広がって括弧で閉じられているようなものは全て、最初のシンボルよりも深くインデントされていることに注意しておいた方が良いだろう。今回の関数定義の中では次のように let 式は defun よりも深くインデントされている。

```
(defun ...
  ...
  ...
  (let...
    (save-excursion
      ...
```

このように書式を工夫することで、 save-excursion の本体部分の二行が save-excursion に付随する括弧で囲まれていることが簡単に見てとれる。同様に、 save-excursion それ自身が let に付随する括弧で囲まれていることもすぐに分る。

```
(let ((oldbuf (current-buffer)))
  (save-excursion
    (set-buffer (get-buffer-create buffer))
    (insert-buffer-substring oldbuf start end))))
```

関数 save-excursion の使い方は、次のテンプレートのスロットが埋められているものだと思うとよく理解出来るだろう。

```
(save-excursion
  first-expression-in-body
  second-expression-in-body
   ...
  last-expression-in-body)
```

この関数では save-excursion には二つのＳ式しか含まれていない。その本体は次の通りである。

```
(set-buffer (get-buffer-create buffer))
(insert-buffer-substring oldbuf start end)
```

関数 append-to-buffer が評価されると、 save-excursion の本体部分の二つのＳ式が順に評価される。最後のＳ式の値が save-excursion関数の値として返される。もう一方のＳ式は単に副作用だけのために評価される。

save-excursion の本体の最初の行は、現在のバッファを append-to-buffer の引数の最初の引数で指定されたものに切り替えるために関数 set-buffer を使っている。(バッファの切替は副作用である。前にも言ったように Lisp にとっての副作用こそが我々の主目的であることが多い。) 二行目がこの関数の本来の作業を行う。

関数 set-buffer は Emacs の注意をテキストをコピーしようとしているバッファの方に向けさせる。そのバッファから save-excursion によって元のバッファに帰ってくるのである。

```
(set-buffer (get-buffer-create buffer))
```

このリストのもっとも内部にあるＳ式は (get-buffer-create buffer)というものである。このＳ式は get-buffer-create 関数を使っている。これはその名前のバッファの内容を取り込むか、もし存在しない場合は新しくその名前を持つバッファを作成する。これは append-to-buffer 関数を使うことで、それ以前には存在してなかったバッファにもテキストを出力することが出来ることを示している。

get-buffer-create はまた set-buffer が不必要なエラーに遭遇するのを防いでもいる。つまり set-buffer を使う時は行き先のバッファが存在している必要があるのだ。もしありもしないバッファを指定したりすると Emacs はそこで仕事をやめてしまう。存在しないバッファである場合には get-buffer-create がそのバッファを作ってくれるので、 set-buffer は常にバッファを得ることが出来るというわけである。

append-to-buffer の最後の行はテキストを追加する働きをする。

```
(insert-buffer-substring oldbuf start end)
```

関数 insert-buffer-substring は最初の引数で指定されたバッファから文字列をコピーし、それを現在のバッファに挿入する。今の場合は insert-buffer-substring の引数は let で生成されバインドされた変数 oldbuf の値であり、これは append-to-buffer コマンドを実行した時点でのカレントバッファである。

insert-buffer-substring が作業を終えると、 save-excursionが元のバッファに戻してくれる。そうして append-to-buffer の仕事は完了する。

骨組みだけ書くと、本体部分がやっていることは次の通りである。

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion                          ; バッファの状態の保存
    change-buffer
    insert-substring-from-oldbuf-into-buffer)

  change-back-to-original-buffer-when-finished
let-the-local-meaning-of-oldbuf-disappear-when-finished
```

以上まとめると、 append-to-buffer は次のような働きをする。まずはカレントバッファの値を oldbuf という変数に保存する。次に別のバッファをユーザーから聞き出し、もし必要なら新規に作成して、そのバッファに移る。 oldbuf の値を利用して元のバッファのリージョン内のテキストを取り出し新しいバッファに挿入する。そして最後に save-excursion を用いて元居たバッファに戻る。

append-to-buffer を見ていく中で、かなり複雑な関数を探検したことだろう。 let や save-excursion の使い方も見られたし、複数のバッファ間の行き来の仕方も見た。他の多くの関数定義の中でもこのようにして let, save-excursion, そして set-buffer を利用している。

4.5 復習

ここで、この章で議論した様々な関数について簡単にまとめておく。

describe-function
describe-variable
: 関数ないしは変数の説明文字列を表示する。通常は、 C-h f と C-h v にバインドされている。

find-tag
: 関数のソースや変数を含むファイルを探して、そのバッファに移り、その関数や変数が定義されている位置にポイントを移動する。通常は M-. にバインドされている。(これは、 META キーを押して終止符を押すという意味である。)

save-excursion
: ポイントとマークの位置を保存し、 save-excursion の引数が評価された後で、それらの値を元に戻す。また、カレントバッファも憶えていて、そのバッファに戻してくれる。

push-mark
: マークを現在の位置に設定し、直前のマークの値を、マークリングに記録する。このマークは、バッファ内の位置を示し、たとえそのバッファにテキストが追加されたり削除されたりしても、その相対的な位置を保持してくれる。

goto-char
: ポイントを引数の値で指定された位置に移動する。引数は数値あるいはマーカ、もしくは（point-min）のように位置を表わす数値を返すようなＳ式でなければならない。

insert-buffer-substring
: この関数に引数として渡されたバッファのリージョンを現在のバッファに挿入する。

mark-whole-buffer
: バッファ全体をリージョンに設定する。普通は C-x h にバインドされている。

`set-buffer`
: Emacs が注目しているバッファを他のバッファに切り替える。が、ディスプレイしているウィンドウは変更しない。人間が他のバッファで作業したい時よりかはプログラムの中でよく使われるものである。

`get-buffer-create`
`get-buffer`
: ある名前を持つバッファを見つける。その名前のバッファが無い場合は新しく作る。 `get-buffer` はもしその名前のバッファが無い場合には `nil`を返す。

4.6 練習問題

- あなた自身の `simplified-end-of-buffer` を書きなさい。そして、それが実際に動くことかどうかテストしてみなさい。
- `find-tag` を利用して、 `copy-to-buffer` 関数のソースを見つけなさい。

5 もう少し複雑な関数

この章では、前の章で学んだことを基礎により複雑な関数を見ていくことにする。 copy-to-buffer 関数は、一つの定義の中で save-excursion を二回使う方法を示してくれる。一方、 insert-buffer 関数の方は interactive 式の中での * の使い方と、 or の使い方、そして、オブジェクトの名前と、その名前のついたオブジェクトとの違いについて教えてくれる。

5.1 copy-to-buffer の定義

append-to-buffer がどのように動作するかを理解した後であれば、 copy-to-buffer の方も簡単に理解出来る。この関数はテキストを他のバッファにコピーする。ただし、そのバッファに付け加えるのではなく、以前あったテキストを置き換えてしまう。 copy-to-buffer 関数のコードは append-to-buffer のコードとほぼ同じであるが、 erase-bufferを使い、また append-to-buffer を二回使っている点が違う。(append-to-buffer の説明については Section 4.4, “append-to-buffer の定義”, p. 38, を見よ。)

copy-to-buffer の本体は次の通りである。

```
...
(interactive "BCopy to buffer: \nr")
  (let ((oldbuf (current-buffer)))
    (save-excursion
      (set-buffer (get-buffer-create buffer))
      (erase-buffer)
      (save-excursion
        (insert-buffer-substring oldbuf start end)))))
```

このコードは append-to-buffer とそっくりである。 append-to-buffer の定義との違いが現れるのは、テキストがコピーされるバッファに移ってからである。つまり、この後 copy-to-buffer 関数では以前の内容を消去してしまう。(これが、「置き換え」と言った意味である。 Emacs ではテキストを置き換える場合、以前の内容をまず消去し、次に新しいテキストを挿入する。) 以前の内容を消去した後、もう一度 save-excursion が使われる。そして新しいテキストが挿入される。

何故 save-excursion が二回も使われるか？ それを理解するために、もう一度この関数が何をやっているか見てみよう。

大ざっぱには、 copy-to-buffer の本体は次のようになっている。

```
(let (bind-oldbuf-to-value-of-current-buffer)
  (save-excursion          ; 最初の save-excursion。
    change-buffer
      (erase-buffer)
      (save-excursion      ; 二番目の save-excursion。
        insert-substring-from-oldbuf-into-buffer)))
```

最初の save-excursion は Emacs にテキストのコピー元のバッファを返している。これは当たり前のことで、 append-to-buffer でも同じような使われ方をしていたのだった。では何故二番目が必要なのか？ 理由は、 insert-buffer-substring が常にポイントを挿入されたリージョンの最後に移動してしまうからである。二番目の save-excursionがあるおかげで、 Emacs は挿入されたテキストの最初の位置にポイントを移動する。大抵の状況では、ユーザーは挿入されたテキストの最初の位置にポイントがあるほうを好むものである。(勿論、 copy-to-buffer 関数は終了時に元居たバッファに戻る — しかし、もしユーザが次にコピー先のバッファに移ったとすると、ポイントはそのテキストの先頭に移動してくれるのである。このように、二番目の save-excursion はちょっぴり気の効いた働きをしてくれるものなのだ。)

5.2 insert-buffer の定義

insert-buffer もまた、バッファに関する関数である。このコマンドは他のバッファをカレントバッファの中に挿入する。これは append-to-buffer や copy-to-buffer と逆である。これらの方は、カレントバッファからリージョンを取り出して他のバッファにコピーするのであった。

更に、このコードは *read-only* なバッファに関する interactiveの使い方や、オブジェクトの名前と、その名前のついたオブジェクトそのものとの違いの重要性を理解させてくれる。これがそのコードである。

```
(defun insert-buffer (buffer)
  "Insert after point the contents of BUFFER.
Puts mark after the inserted text.
BUFFER may be a buffer or a buffer name."
  (interactive "*bInsert buffer: ")
  (or (bufferp buffer)
      (setq buffer (get-buffer buffer)))
  (let (start end newmark)
    (save-excursion
      (save-excursion
        (set-buffer buffer)
        (setq start (point-min) end (point-max)))
      (insert-buffer-substring buffer start end)
      (setq newmark (point)))
    (push-mark newmark)))
```

他の関数定義と同様に、テンプレートを使ってこの関数のアウトラインを見ることが出来る。

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  body...)
```

5.2.1 insert-buffer の中のインタラクティブ式

insert-buffer ではインタラクティブ宣言の引数には二つの部分がある。一つはアスタリスク ‘*’ であり、もう一つは ‘bInsert buffer: ’である。

書き込み不可のバッファ

アスタリスクはバッファが read-only— つまり変更出来ないバッファである場合のためのものである。もし、insert-buffer が read-only なバッファで呼ばれた場合、エコー領域にその旨を知らせるメッセージが表示され、ビープ音が鳴るか、画面が明滅する。「あなたはこのバッファに如何なるものも挿入してはいけませんよ」というわけである。アスタリスクの後には、次の引数と分離するために改行コードを付ける必要はない。

インタラクティブ式の中の ‘b’

インタラクティブ式の次の引数は小文字の ‘b’ から始まっている。(これは append-to-buffer のコードの中のとは違うことに注意。こっちでは大文字の ‘B’ を使っていたのだった。 Section 4.4, “append-to-buffer の定義”, p. 38, 参照。) 小文字の ‘b’ は Lisp インタプリタに insert-buffer の引数が存在するバッファないしは、その名前でなければならないことを示している。(大文字の ‘B’ オプションの方は存在しないバッファであっても良い。) Emacs はプロンプトを出してバッファの名前を要求し、同時にデフォルトのバッファも提示してくれる。名前の補完もやってくれる。もし、ここで存在しないバッファを指定すると、 “No match” というメッセージを受け取る。端末によってはビープ音がするかもしれない。

5.2.2 関数 insert-buffer の本体部分

関数 insert-buffer の本体は、主に二つの部分からなっている。一つは or 式であり、もう一つは let 式である。or 式を使う目的は、引数 buffer がバッファの名前ではなくちゃんとバッファそのものにバインドされていることを確認することである。let 式の本体は、他のバッファをカレントバッファに挿入するコードを含んでいる。

大まかに言って、二つのＳ式は次のように insert-buffer 関数の中に組込まれている。

```
(defun insert-buffer (buffer)
  "documentation..."
  (interactive "*bInsert buffer: ")
  (or ...
      ...
  (let (varlist)
       body-of-let... )
```

引数 buffer がバッファの名前ではなくバッファそのものにバインドされているかを or 式がどのようにして確かめているかを理解するには、最初に関数 or を理解する必要がある。

が、その前にまずこの部分を if を使って書き直してみたい。そうすればより身近な方法で何がなされているかを見ることが出来るだろう。

5.2.3 or の代わりに if を使った insert-buffer

やるべき仕事は buffer の値がバッファそのものであり、バッファ名ではないことを確認することである。もし値が名前であったら、その名前のついたバッファを持ってこないといけない。

まずあなたが何かのコンファレンスに参加しており、案内係があなた方の名前が書かれたリストをもってあなたを探している状況を思い浮かべて欲しい。この状態では案内係はあなた自身ではなく、あなたの名前に「バインド」されている。しかし、案内係があなたを見つけてあなたの腕を取った時点で、案内係はあなた自身に「バインド」されたことになる。

Lisp ではこの状況を次のように説明出来る。

```
(if (not (お客をつかまえている))
    (お客を見つけて腕を取る))
```

我々はこれと同じことをバッファについてやりたいわけである — バッファそのものでない場合も、バッファを見つけて取って来たいのだ。

bufferp という (バッファの名前ではなく) バッファかどうかを判定してくれる述語を使うと、このコードを次のように書くことが出来る。

```
(if (not (bufferp buffer))              ; if-part
    (setq buffer (get-buffer buffer)))  ; then-part
```

コメントにもあるが、上の式で、 if 式の真偽テストの部分は(not (bufferp buffer)) であり、一方 then-part の部分は(setq buffer (get-buffer buffer)) である。

このテストの中では bufferp 関数は引数がバッファであれば真の値を返す — しかしバッファの名前であれば偽の値を返す。(bufferp という関数の名前の最後は 'p' で終わっている。以前見たようにこの 'p' は、この関数が述語、つまり、ある性質を持つか否かを判定する関数であることを示すための工夫として付けられている。Section 1.8.4, "関数に間違った型の引数を与えると", p. 9, を参照のこと。)

(bufferp buffer) 式の前には関数 not がある。即ち真偽テストは次の通りである。

```
(not (bufferp buffer))
```

not は引数が偽であれば真を返し、偽であれば真を返す関数である。従って、もし（bufferp buffer）が真を返せば、 not 式が偽を返すし、偽を返せば真を返す。「真でない」というのは偽ということだし、「偽でない」というのは真だというわけである。

if は、このテストを使って次のように動作する。変数 bufferの値がバッファの名前ではなく実際にバッファだった場合、真偽テストの結果は偽である。すると、 if 式は then-part を評価しない。これはまともな動作である。というのは、もし変数 buffer が本当にバッファであれば、これに対して特に何もする必要はないからである。

一方で、 buffer の値がもしバッファそのものではなくバッファの名前であったとすると、真偽テストでは真が返り、結果として then-part が評価される。今の場合、 then-part は（setq buffer（get-buffer buffer））である。このＳ式は、その名前のついた実際のバッファを得るために get-buffer関数を使っている。そして、 setq で、変数 buffer の値をバッファの名前から実際のバッファそのものに置き換えている。

5.2.4 本体部分の or

insert-buffer 関数で or 式を使う目的は引数 bufferがバッファの名前ではなくバッファそのものにバインドされているかどうかを確認することにある。前節で、どのようにして if を使ってこの作業をするかを見た。しかし、 insert-buffer 関数では実際は or を使っている。この理由を理解するには、 or 関数の働きを理解しなければならない。

関数 or はいくらでも引数を持つことが出来る。この関数は各引数を順に評価していき、 nil でなかった最初の引数の値を返す。また、これが or の重要な特徴なのだが、この関数は引数で nil 以外の値を返すものがあった場合、その後の引数はもはや評価しない。

今の場合の or 式は次の通りである。

```
(or (bufferp buffer)
    (setq buffer (get-buffer buffer)))
```

or の最初の引数は Ｓ式（bufferp buffer）である。この式は、もし buffer がバッファそのものでありバッファの名前ではない場合は、真を返す。 or 式では、この場合、 or 式の値として真を返し、その後の引数は評価されない — 我々にはこの方が都合が良い。というのもこの場合にはつまり buffer が真のバッファである場合には、この値について何もする必要がないからである。

一方で、もし（bufferp buffer）が nil であれば、それは buffer の値がバッファの名前であるということである。この場合、 Lisp インタプリタは or 式の次の引数も評価する。これは（setq buffer（get-buffer buffer））という式である。この式は値 nil を返す。これが変数 buffer にセットされた値である — そしてこの値はバッファそのものであり、バッファの名前ではない。

以上のことから、シンボル buffer は常に、あるバッファの名前ではなくバッファそのものにバインドされることになる。また、以上の手続きは全てが必要なことである。というのも、次の行の set-buffer 関数はバッファに対してしか働かないからである。

ついでながら、 or を使うと案内係というのは次のように書ける。

```
(or (holding-on-to-guest) (find-and-take-arm-of-guest))
```

5.2.5 insert-buffer の中の let 式

insert-buffer 関数では、変数 buffer がバッファの名前ではなくバッファそのものを表わすことを確認した後には、 let 式が続く。ここでは三つの変数 start, end, newmark を指定している。これらの変数は let 式の内部でのみ使われ、 let 式の評価が終わるまでの間は、一次的に Emacs 内の同じ名前の他の変数は隠されている。

let 式の本体は二つの save-excursion を使っている。まずは、内部にある方の save-excursion 式を細かく見ていこう。この式は次の通りである。

```
(save-excursion
  (set-buffer buffer)
  (setq start (point-min) end (point-max)))
```

S式 (set-buffer buffer) は、Emacs が注目するバッファをテキストがコピー元のバッファに切り替える。このバッファの中で、point-min コマンド及び、point-max コマンドを使って変数 start と 変数 end に各々バッファの始まりと終わりの位置を設定する。setqを使って、一つのS式で二つの変数を同時に設定する方法についてはもう既に説明したのだった。setq の最初の引数は二番目の値に、三番目の引数は四番目の引数の値にセットされる。

外側の save-excursion 式は次の通りである。

```
(save-excursion
  (inner-save-excursion-expression
     (go-to-new-buffer-and-set-start-and-end)
  (insert-buffer-substring buffer start end)
  (setq newmark (point)))
```

関数 insert-buffer-substring はテキストを buffer の中の start と end で指定されたリージョンからカレントバッファの中にテキストをコピーする。コピー元のバッファ全体が start と end の間にあるために、そのバッファ全体が現在編集中のバッファにコピーされることになる。次に、この時点で挿入されたテキストの最後にあるポイントの値が、変数 newmark に記録される。

外側の save-excursion の本体が評価された後、ポイントとマークの位置は元の位置に戻される。

しかしながら、ポイントとマークの位置は新しく挿入されたテキストの最初と最後にある方が、何かと都合が良い。newmark 変数は挿入されたテキストの最後の位置を記録する。let 式の最後の行で (push-mark newmark) というS式でマークをこの位置に設定している。(以前のマークの位置移動することも可能である。この値はマークリングに保存されていて、C-u C-SPC で遡ることが出来る。) 一方、ポイントの方は初めから挿入されたテキストの最初の位置にある。これは insert 関数を呼び出す前のポイントの位置がここだからである。

完全な let 式は次の通りである。

```
(let (start end newmark)
  (save-excursion
    (save-excursion
      (set-buffer buffer)
      (setq start (point-min) end (point-max)))
    (insert-buffer-substring buffer start end)
    (setq newmark (point)))
  (push-mark newmark))
```

append-to-buffer 関数の時と同じように、insert-buffer 関数も let, save-excursion, そして set-buffer を使っている。それに加えて、この関数は or のある使い方も一つ教えてくれている。これらの関数は皆大切な部品であり、これから何度でも出てくることだろう。

5.3 beginning-of-buffer の完全な定義

beginning-of-buffer 関数の基本的な構造については、既に議論したことがあるのだった。(Section 4.2, "簡略版 beginning-of-buffer の定義", p. 35.) このセクションではこの定義の複雑な部分を説明することにする。

以前説明した通り、引数無しで beginning-of-buffer を呼び出すと、その時点でのポイントの位置にマークを設定して、カーソルがバッファの先頭に移動する。しかしながら、もしこのコマンドが 1 から 10 までの数と一緒に呼び出された場合、この関数はバッファ全体を 10 としてこの数の割合の分を計算し、バッファの先頭からその割合の分だけ進んだ位置にカーソルを移動させる。というわけで、M-< でこのコマンドを呼び出せば、カーソルはバッファの先頭に移動するし、もし例えば C-u 7 M-< として呼び出せば、カーソルはバッファ全体の 70% 分の位置にポイントを移動する。もし引数が 10 より大きかった場合はバッファの最後に位置に移動する。

beginning-of-buffer 関数は、引数無しで呼び出すことが出来る。つまり、引数は省略可能である。

5.3.1 省略可能な引数

特にそうでないと言われない限り、Lisp は、関数定義で引数が指定された関数はその引数のための値とともに呼ばれるものと思っている。もし、そうでなかった場合、エラーが出され、'Wrong number of arguments' というメッセージを受け取ることになる。

しかしながら、Lisp の特徴として省略可能引数というものがある。あるキーワード (*keyword*) を使うと Lisp インタプリタにその引数が省略可能だと伝えることが出来るのだ。そのキーワードとは &optional である。('optional' の前の '&' もこのキーワードの一部分である。) 関数定義の中でこのキーワード &optional の後に続く引数は、関数が呼ばれる際に必ずしも値を渡される必要はない。

というわけで、 beginning-of-buffer の関数定義の最初の行は次のようになる。

```
(defun beginning-of-buffer (&optional arg)
```

また、関数定義全体のアウトラインは次のようになっている。

```
(defun beginning-of-buffer (&optional arg)
  "documentation..."
  (interactive "P")
  (push-mark)
  (goto-char
    (if-there-is-an-argument
        figure-out-where-to-go
      else-go-to
      (point-min))))
```

この関数は simplified-beginning-of-buffer と似ている。違うのは、 interactive 式に "P" が引数として使われていることと、引数が指定されている場合にカーソルを何処に移動するかを判断する goto-char 関数に続いて if-then-else 式が来ていることである。

interactive 式の中の "P" は、Emacs に、関数に前置引数を渡すように伝えるものである。前置引数というのは META キーに続いて数をタイプするか、 C-u に押してから数をタイプすることで作ることが出来る。(もし C-u に続いて数を入力しない場合には、デフォルトとして 4 が渡される。)

if 式の真偽テストの部分は単純である。単に arg だけだ。もし、 arg が nil 以外の値を持てば、即ち beginning-of-buffer が引数付きで呼ばれた場合は、真偽テストでは真が返り、 if 式の then-part が評価される。一方、もし beginning-of-buffer が引数とともに呼ばれなかったなら、 argの値は nil になり、 if 式の else-part の方が評価される。 else-part は単に point-min であり、この場合には goto-char 式の全体は (goto-char (point-min)) になる。これは、以前の短縮版 beginning-of-buffer 関数で見た通りである。

5.3.2 引数付きで呼び出された場合の beginning-of-buffer

beginning-of-buffer が引数付きで呼び出された場合は、 goto-char に渡す値を計算する、あるＳ式が評価される。このＳ式は、ちょっと見にはかなり複雑に見える。内部に if 式を含んでいるし、算数を沢山使っている。実際のコードは次の通りである。

```
(if (> (buffer-size) 10000)
    ;; 大きなバッファサイズの場合のオーバーフローを避ける！
    (* (prefix-numeric-value arg) (/ (buffer-size) 10))
  (/
   (+ 10
      (*
       (buffer-size) (prefix-numeric-value arg))) 10))
```

他の複雑そうなＳ式と同様、これもテンプレートに当てはめてみれば理解しやすくなる。この場合は、 if-then-else 式のテンプレートに当てはめる。骨組みは次のようになる。

```
(if (buffer-is-large
     divide-buffer-size-by-10-and-multiply-by-arg
  else-use-alternate-calculation
```

この中の `if` 式の真偽テストはバッファのサイズをチェックしている。その理由は、 Emacs Lisp の version 18 は約 80,000 以下の数値しか扱わないのだが (そんなに大きな数は必要ない), この計算に従うと、大きなバッファの場合は Emacs は大き過ぎる数を扱うことになってしまうからだ。コードのコメントの中に出てくる 'overflow' とは数が大き過ぎることを意味している。

ということで、バッファが大きい場合と、そうでない場合の二つに分けている。

大きなバッファの場合

`beginning-of-buffer` の中で内部にある方の `if` 式はバッファのサイズが 10,000 文字を越えているかどうかをテストしている。そのために、`>` と `buffer-size` という関数を使っている。その行は次の通りである。

```
(if (> (buffer-size) 10000)
```

もしバッファが大きい場合は、 `if` 式の then-part が評価される。この部分を読みやすいよう書き直すと次のようになる。

```
(*
  (prefix-numeric-value arg)
  (/ (buffer-size) 10))
```

このＳ式は乗法であり、二つの引数を関数 `*` に渡している。

最初の引数は（`prefix-numeric-value arg`）である。 `interactive` 宣言で `"P"` が指定されている場合、引数は関数に「未処理の前置引数」(raw prefix argument) として渡され、数値としては渡されない。 (リストの中では数値である。) 算数で使えるようにするには変換する必要があり、その仕事を `prefix-numeric-value` がやっているのである。

二番目の引数は（`/`（`buffer-size`）`10`）である。この式は数値としてのバッファの値を 10 で割る。これはバッファサイズの一割にどれだけの文字数が含まれているかを計算するものだ。 (Lisp では `/` は除法に使われる。 `*` が乗法に使われるのと同様である。)

乗法の式全体としては、この値に前置引数の値を掛けることになる — こんな感じだ。

```
(* numeric-value-of-prefix-arg
   number-of-characters-in-one-tenth-of-the-buffer)
```

例えば、仮に前置引数が '7' だったとするなら、この 10 分の 1 の数に 7 を掛けることで、バッファの中の 70% の位置が得られることになる。

結局、バッファが大きい場合の `goto-char` 式全体は次のようになる。

```
(goto-char (* (prefix-numeric-value arg)
              (/ (buffer-size) 10)))
```

これでカーソルは望む場所に移動してくれる。

小さなバッファの場合

バッファが 10,000 文字よりも小さかった場合は、ちょっと違った計算方法が取られる。さっきの方法でも出来るのだから、こんなことをする必要はないと思うかもしれない。しかし、バッファが小さい場合、初めの方法ではカーソルを望む場所にきちんと移動してくれない。次に説明する二番目の方法の方がちゃんとした仕事をしてくれる。

コードは次の通りである。

```
(/ (+ 10 (* (buffer-size) (prefix-numeric-value arg))) 10))
```

このコードの中で何をしているかは、括弧の中にどのように関数が埋め込まれているかを見れば分る。各々の式をそれを含む括弧より深くインデントしてやれば、より見やすくなる。

```
(/
 (+ 10
    (*
     (buffer-size)
     (prefix-numeric-value arg)))
 10))
```

これらの括弧を見ると、もっとも内部にある関数は (prefix-numeric-value arg) であることが分る。これは未処理の前置引数を数値に変換するものである。この数値は次の式によってバッファサイズと掛け合わされる。

```
(* (buffer-size) (prefix-numeric-value arg)
```

この掛け算では、バッファのサイズの値よりも大きな数が生ずる — 例えば引数が 7 なら 7 倍大きな数になる。これに 10 が加えられ、最後にこの大きな数が 10 で割られる。こうして得られる値は、バッファのサイズ中の目的とする割合よりも一文字分だけ大きな数である。

これら一連の操作の結果で得られた数が goto-char に渡され、カーソルがその位置に移動する。

5.3.3 完全版 beginning-of-buffer

ここで、 beginning-of-buffer の完全なコードを挙げておく。

```
(defun beginning-of-buffer (&optional arg)
  "Move point to the beginning of the buffer;
leave mark at previous position.
With arg N, put point N/10 of the way
from the true beginning.
Don't use this in Lisp programs!
\(goto-char (point-min)) is faster
and does not set the mark."
  (interactive "P")
  (push-mark)
  (goto-char
   (if arg
       (if (> (buffer-size) 10000)
           ;; 大きなバッファサイズの場合のオーバーフローを避ける！
           (* (prefix-numeric-value arg)
              (/ (buffer-size) 10))
         (/ (+ 10 (* (buffer-size)
                     (prefix-numeric-value arg)))
            10))
     (point-min)))
  (if arg (forward-line 1)))
```

二つの細かな点を除けば、これまでの議論でこの関数がどう動くかが分るだろう。細かな点の一つ目は、説明文字列についてであり、二つ目は、この関数の最後の行に関するものである。

で、一点目だが、説明文字列の中に、あるＳ式についてのリファレンスがある。

```
\(goto-char (point-min))
```

この式の最初の括弧の前に '\' が使われている。この '\' は Lisp インタプリタに対し、このＳ式は説明として表示されるものであって、Ｓ式として評価されるものではないことを示すものである。

次に二点目だが、 beginning-of-buffer コマンドの最後の行は、この関数が引数とともに呼ばれた場合に、ポイントを次の行の最初に持ってくためのものである。

```
(if arg (forward-line 1)))
```

これによって、カーソルはバッファの中の適当な割合の位置の次の行の先頭に移動する。これが最良の位置であろう。カーソルは少くとも要求された割合よりかは常に大きい位置に移動するからだ。この動作はちょっと几帳面過ぎるかもしれないが、かと言ってそうしなければ、必ず文句を言う人がいるだろう。

5.4 復習

ここで、この章の幾つかのトピックを手短にまとめておく。

or 引数を順に評価していき、 nil でない最初の値を返す。ただし、もしどの引数も nil を返した場合は nil を返す。一言で言うと、引数の中で真である最初の値を返す。つまり、ある引数が真であるかまたは (or) その他の引数のどれかが真であれば真の値を返す。

and 引数を順に評価していき、どれか一つでも nil があれば、 nilを返し、どれも nil で無ければ最後の引数の値を返す。一言で言うと、全ての引数が真である場合のみ真である値を返す。つまり、ある引数が真であり、かつ (and) その他の引数も全て真である場合に真の引数を返す。

&optional
関数定義での引数が省略可能であることを示すために使うキーワード。これがあると、その引数がなくともその関数を評価することが出来る。

forward-line
ポイントを次の行の先頭に移動する。 1 より大きい引数が与えられた場合はその数の行だけ進む。進めない場合は行けるところまで進み、行くはずではあったが進めなかった分の行数を値として返す。

erase-buffer
カレントバッファの内容を全て削除する。

bufferp 引数がバッファの場合、 t を返す。そうでない場合は nil を返す。

5.5 &optional 引数の練習問題

省略可能な引数を持ち、その引数の数が fill-column の値よりも大きいか小さいかをテストして結果を教えてくれるようなインタラクティブな関数を書きなさい。ただし、引数なしの場合は 56 をデフォルトとして使うようにしなさい。

6 ナローイングとワイドニング

ナローイングはバッファの特定部分にのみ注目し、その他の部分を間違って変更したりすることなく作業するための Emacs の機能である。ナローイングは普通は使えないようになっている。これは初心者を混乱させないためである。

ナローイングを使うと、バッファのその他の部分はあたかも存在しないように見えなくなる。これは、例えばバッファのある部分でのみ単語を置き換えたい場合なんかに有効である。つまり特定の部分にナローイングしてそこで置換を実行すると、その部分のみ置換が実行されて残りの部分は無視される。検索についても同様で、ナローイングされた部分でのみ検索を行い、外の部分は探さない。というわけで、文書のある部分のみ修正したい場合など、そのリージョンにナローイングをかけることで、間違って修正不要な箇所まで見てしまうのを防ぐことが出来るのである。

しかしながら、ナローイングするとバッファの残りの部分が見えなくなるので、意図せずナローイングしてしまった場合、ファイルのその部分を削除してしまったのではないかと不安に思う人も出てくるかもしれない。更に、 `undo` コマンド (これは大抵 `C-x u` にバインドされている) でもナローイングは解除されない (それに、そうすべきではない)。従って、もし `widen` コマンドでまた見えるように出来るということを知らなかった場合、パニックに陥いるかもしれない。(Emacs version 18 では、 `widen` のキーバインディングは `C-x w` だった。version 19 では `C-x n w` になっている。)

ナローイングは、人間だけではなく Lisp インタプリタにとっても便利なようになっている。 Emacs Lisp の関数はしばしば、バッファのある限定された部分に対してのみ働くように設計されている。逆に言うと、 Emacs Lisp 関数はナローイングされたバッファ全体に対して働くようになっているとも言える。例えば `what-line` 関数は、もしバッファにナローイングがかかっていればそれを解除し、仕事が終わるとまたナローイングを元に戻す。一方、 `what-line` 関数の中から呼び出される `count-lines` 関数は、まずナローイングを用いてバッファの目的とする部分に限定して仕事を行い、その後ナローイングを解除して元の状態に戻す。

6.1 特殊形式 save-restriction

Emacs Lisp では、 `save-restriction` という特殊形式を使うことで、ナローイングのかかった状態を保持することが出来るようになっている。 Lisp インタプリタが `save-restriction` 式に出逢った場合、まず `save-restriction` 式の本体部分を評価し、次にそのコードを実行することでナローイングの状況が変わった場合にそれを元に戻す。例えば、バッファがナローイングされていて、 `save-restriction` の本体で、ナローイングが解除された場合、実行後に `save-restriction` はバッファにナローイングされたリージョンを返してくれる。 `what-line` コマンドでは、バッファにどんなナローイングが設定されていても `save-restriction` に続く `widen` コマンドで解除してしまう。元のナローイングはこの関数が終了する直前に元に戻されるわけである。

`save-restriction` 式のテンプレートは単純である。

```
(save-restriction
  body... )
```

`save-restriction` の本体は一つないしは複数のＳ式であり、それらは Lisp インタプリタによって順に評価される。

最後に注意しておくことがある。それは `save-excursion` と `save-restriction` の両方を、片方のすぐ後にもう片方を続けて使う場合、 `save-excursion` の方を外側にすることだ。これを逆にすると、 `save-excursion` を呼び出してからナローイングが設定されたバッファに移動した場合にそのナローイングの状況を記録しそこねることがあるのだ。ということで、もし `save-excusion` と `save-restriction` を両方一緒に使うなら、次のように使うことになる。

```
(save-excursion
  (save-restriction
    body...))
```

6.2 what-line

what-line コマンドは現在カーソルがある場所までの行数を教えてくれるものである。この関数を見ることで、 save-restriction と save-excursion の使い方を知ることが出来る。以下がこの関数の完全なコードである。

```
(defun what-line ()
  "Print the current line number (in the buffer) of point."
  (interactive)
  (save-restriction
    (widen)
    (save-excursion
      (beginning-of-line)
      (message "Line %d"
               (1+ (count-lines 1 (point)))))))
```

この関数は一行の説明文字列を持ち、あなたが予想した通り、インタラクティブな関数である。次の二つの行では save-restriction と widen を使っている。

特殊形式 save-restriction は、カレントバッファにどのようなナローイングがかかっていようと、それを記録して本体のコードが評価し終わった後にその状態に戻してくれる。

特殊形式 save-restriction の後には widen が続いている。この関数は、 what-line が呼ばれた時にカレントバッファのナローイングを解除する。(ここでのナローイングは、 save-restriction が憶えているナローイングである。) これによって行数を数えるコマンドは、バッファの最初からの行数を数えることが出来るのである。そうしない場合は、現在アクセス可能なリージョンの範囲内でしか行数をカウントしない。元のナローイングは特殊形式 save-restriction によってこの関数が終了する直前に元の状態に戻される。

widen の呼び出しに続いて save-excursion が使われている。これはカーソルの位置、(即ち、 point の位置) とマークの位置を保存し、本体部分のコードの中の beginning-of-line 関数によって動かされたポイントの位置を元に戻す、という働きをしている。

((widen) 式が save-restriction と save-excursion の間にあることに注意しよう。これら二つの save- ... 式を続けて使う場合は save-excursion を外側に書かないといけないのであった。)

what-line 関数の最後の二つの行は、バッファ内の行数を数え、それをエコー領域に表示するものである。

```
(message "Line %d"
         (1+ (count-lines 1 (point)))))
```

message 関数は Emacs の画面の最下行に一行メッセージを表示する関数である。最初の引数は二重引用符に挟まれていて、これは文字文字列を表示するものである。ただし、‘%d’, ‘%s’, ‘%c’ も入れることが出来て、これらはこの文字列の後に続く引数を表示する。‘%d’ は引数を 10 進数として表示するので、メッセージは例えば ‘Line 243’ と言った感じになる。

‘%d’ の場所に表示される数字は、この関数の最後の行で計算される。

```
(1+ (count-lines 1 (point)))
```

これがやっていることは 1 で示された位置、つまりバッファの最初の位置から (point) までの行数を数え、それに 1 を加えることである。(1+という関数は引数に 1 を加える関数である。)1 を加えるのは、例えば二行目はその前に一行しかないからである。 count-lines はカレント行の前の行までしか数えないので、こうしないと一行ずれてしまうのだ。

count-lines が仕事を終えてメッセージがエコー領域に表示された後は save-excrusion によってポイントとマークの位置が元の状態に戻される。そして、 save-restriction がナローイングの状態を元に戻す。

6.3 ナローイングの練習問題

カレントバッファの最初の 60 文字を表示するような関数を書きなさい。その際、例えばナローイングにより後半部分しかアクセス不能であり、最初の行が見えない状態であってもきちんと表示するようなものにしなさい。また、ポイント、マーク、及びナローイングを復元するようにすること。この練習問題のためには、 `save-restriction`、 `widen`、 `goto-char`、 `point-min`、 `buffer-substring`、 `message` 及び他の関数をうまく組み合わせて使う必要がある。

7 car, cdr, cons: 基本関数

Lisp では car、 cdr、そして cons が基本的な関数である。 cons 関数はリストを作るのに使われ、 car 関数と cdr 関数はそれらを分解するのに使われる。

copy-region-as-kill 関数を “walk-through” する時に、 consと共に cdr の二つの変種である setcdr と nthcdr についても見るつもりだ。 (Section 8.5, “copy-region-as-kill”, p. 71).

cons 関数の名前は特に非合理的なものではない。この名前は ‘construct’ という単語を略したものである。一方、 car と cdrの名前の由来は難解である。 car は ‘Contents of the Address part of the Register’ というフレーズの頭文字から来ており、 cdr (‘could-er’ と発音する) は ‘Contents of the Decrement part of the Register’ というフレーズから来ている。これらのフレーズは、 Lisp が開発された頃の極めて初期のハードウェアの特定の部分に基づくものであるが、単に時代遅れであるというだけでなく、実に 25 年以上もの間、 Lisp に関わる人々にとって全く見当はずれのものであった。だがしかし、これらの関数をもっと合理的な名前で呼ぼうとした学者も何人かいたにも関わらず、現在でもこの古い用語が使われている。特に Emacs Lisp のソースコードでもこの用語が使われているので、この入門書でもこれらを使うことにしよう。

7.1 car と cdr

リストの car とは、単にそのリストの最初の要素のことである。従って、リスト (rose violet daisy buttercup) の car は roseである。

もし、この文章を GNU Emacs の Info で読んでいるなら、次を評価してみることでこのことを確認出来る。

```
(car '(rose violet daisy buttercup))
```

このＳ式を評価すると、エコー領域に rose と表示されたはずだ。

明らかに car には first という名前の方がふさわしいし、よくそう提案されてもいる。

car は最初の要素を除いたりはしない。ただ単にそれが何かを伝えるだけである。あるリストに car を施した後も、そのリストは元のままである。専門用語では、 car は「非破壊的 (non-destructive)」であると言ったりする。この特徴は、後で重要であることが分るだろう。

List の cdr はリストの残りの部分である。つまり、 cdr 関数はリストの最初の要素を除いた部分を返す。従って、'(rose vilolet daisy buttercup) というリストの car が rose であるように、このリストの残りの部分、つまり cdr を施して返される値は (violet daisy buttercup) である。

このことも、次をいつも通り評価してみることで確かめられる。

```
(cdr '(rose violet daisy buttercup))
```

これを評価すると、エコー領域に (violet daisy buttercup) と表示されたはずだ。

car と同様、 cdr もリストから要素を取り除いたりしない — ただ単に二番目以降の要素のリストを返すだけである。

ついでに言っておくと、上の例では花のリストに引用符が付いている。もしこれが付いていなければ Lisp インタプリタは rose を関数として呼び出して評価しようとするだろう。この例では、そういうことをしたいのではなかった。

明らかに cdr には rest という名前の方が、ふさわしい。

(ここでちょっと教訓。あなたが新しい関数に名前をつける場合、それについて極めて注意深く考えないといけない。というのも、あなたは多分、あなたが思うよりずっと長い間、その名前につきまとわれることになるからだ。この文章でこれらの名前を使っているのは、 Emacs Lisp のソースコードでこれらを使っているからであり、そうしないとコードを読むのがつらいだろうからである。しかし、どうかあなた自身はこれらの単語を使わないで欲しい。そうすれば後からやってきた人々から感謝されるだろう。)

car や cdr を (pine fir oak maple) といったシンボルからなるリストに施した時、 car によって返される要素はシンボル pine であり、括弧は付いてはいない。 pine はこのリストの最初の要素である。しかしなが

ら、リストの cdr は（fir oak maple）というリストである。それは、次の式をいつものように評価してみればすぐに分る。

```
(car '(pine fir oak maple))

(cdr '(pine fir oak maple))
```

一方、リストのリストでは、そのリストの最初の要素がそれ自身リストである。この場合、 car は最初のリストとしての要素を返す。例えば次のリストは三つのサブリストを含んでいる。各々、肉食動物、草食動物、海洋哺乳類のリストである。

```
(car '((lion tiger cheetah)
       (gazelle antelope zebra)
       (whale dolphin seal)))
```

この場合、最初の要素、つまり全体のリストの car は肉食動物のリスト（lion tiger cheetah）である。そしてリストの残りは（(gazelle antelope zebra) (whale dolphin seal)）である。

```
(cdr '((lion tiger cheetah)
       (gazelle antelope zebra)
       (whale dolphin seal)))
```

ここでもう一度、 car と cdr が非破壊的であると言っておいたほうがいいだろう。これらの関数はリストを変化させないのだ。このことはこれらの関数を使うにあたって極めて重要になる。

また最初の章でアトムについて議論した時に、 Lisp では「ある種のアトム、例えば配列 (array)、は複数の部分に分解出来る。しかし、そのメカニズムはリストを分解する時のメカニズムとは違う。リストの操作に関する限り、リストのアトムは分解不可能である。」と書いた。(Section 1.1.1, “Lisp Atoms”, p. 1). car 関数と cdr 関数はリストを分解するのに使われ、 Lisp の基本と考えられている。しかし、これらの関数は配列を分解したり、部分を取り出したりするのには使えない。従って、配列はリストではなくアトムであると考えられるわけである。また、もう一つの基本的な関数である cons も、リストを構築することは出来るが、配列を作ることは出来ない。(配列は、配列を扱う特殊な関数を使って操作する。詳しくは Section “Arrays” in *The GNU Emacs Lisp Reference Manual*, を参照のこと。)

7.2 cons

cons は、リストを作る関数である。 car や cdr とは逆の働きをするものだと言える。例えば、 cons は三つの要素を持つリスト（fir oak maple）から四つの要素を持つリストを作ることが出来る。つまり、

```
(cons 'pine '(fir oak maple))
```

を評価すると、

```
(pine fir oak maple)
```

というリストがエコー領域に表示される。このように、 cons はリストの先頭に新しい一つの要素を付け加える。リストに要素を追加するわけである。

cons には、まず要素を付け加えるべきリストが必要になる。[1] 何も無いところからスタートすることは出来ないのである。もし、リストを作りたいならば、少くともまず初めに空リストを用意する必要がある。次に挙げるのは、 cons を連続して使って花のリストを作るものである。もし、この文章を Emacs の Info で読んでいるのなら、各々のＳ式をいつものように評価することが出来る。表示される値が ‘⇒’の後に書かれている。これを「次のように評価される」と置き換えて読んで欲しい。

[1] 実際は、アトムに要素を cons してドット対を作ることも出来る。が、ドット対のことはここでは説明しない。これについては次を参照せよ。 Section “Dotted Pair Notation” in *The GNU Emacs Lisp Reference Manual*.

```
(cons 'buttercup ())
     ⇒ (buttercup)

(cons 'daisy '(buttercup))
     ⇒ (daisy buttercup)

(cons 'violet '(daisy buttercup))
     ⇒ (violet daisy buttercup)

(cons 'rose '(violet daisy buttercup))
     ⇒ (rose violet daisy buttercup)
```

最初の例では、空リストが () という形で出ている。そして、 buttercup とこの空リストから新しいリストが作られる。見ればわかると思うが、空リストは作成されたリストの要素としては出て来ない。(buttercup) というリストが見えるだけである。空リストは要素としては数えられない。というのも、空リストには要素が一つもないからである。一般的にいうと、空リストは見えないのである。

二番目の例では (cons 'daisy '(buttercup)) を評価することで daisy を buttercup の前に付け加えて、二つの要素を持つリストを作っている。三番目の例では daisy と buttercup の前に violet を加えて三つの要素を持つリストを作っている。

7.2.1 リストの長さを調べる: length

あるリストがどれだけ多くの要素を持つかを見るには、 length という Lisp の関数を使えばよい。次のように感じである。

```
(length '(buttercup))
     ⇒ 1

(length '(daisy buttercup))
     ⇒ 2

(length (cons 'violet '(daisy buttercup)))
     ⇒ 3
```

三番目の例では、三つの要素を持つリストを作るために cons 関数が使われ、その結果出来たリストが length 関数に引数として渡されている。

length 関数を使って、空リストの中の要素の数を数えることも出来る。

```
(length ())
     ⇒ 0
```

予想した通りだと思うが、空リストの中の要素の数は零個と数えられる。

リストが無い場合にその長さを求めさせてみるのも面白い実験である。つまり、引数として空リストすら渡さずに length を呼び出すのである。

```
(length )
```

やってみれば分るが、これを評価すると、エラーメッセージが表示される。

```
Wrong number of arguments: #<subr length>, 0
```

これは引数の数が間違っているという意味のメッセージである。今の場合なら、本当は一つの引数が渡されなければならない。例えば引数がリストなら、そのリストの長さをこの関数が測ることになるはずだったのである。(リストが沢山の要素を持っていたとしても、一つのリストは一つの引数であることに注意。)

(訳註 : 因みに length の引数は必ずしもリストでなくともよい。例えば length に文字列を渡すと、その文字列の長さが返る。)

エラーメッセージの中の '#<subr length>' という部分は、関数の名前である。これは、特殊な表記法で書かれている。'#<subr' は lengthが Emacs Lisp ではなく C で書かれたプリミティブな関数であることを表し

ている。(‘subr’ は ‘subroutine’ の略である。) Section “What Is a Function?” in *The GNU Emacs Lisp Reference Manual*, に subroutine についてのより詳しい情報がある。

7.3 nthcdr

nthcdr 関数は、 cdr 関数に関連した関数である。これは、リストの cdr を繰り返し取るものである。

(pine fir oak maple) の cdr を取ると、(fir oak maple) が返される。もしこの返された値にもう一度同じ操作をすると、(oak maple) が返される。(勿論、リストに何回 cdr を施しても、元のリストは変化しない。 cdr は対象リストを変化させはしないからである。従って、 cdr の cdr を評価する必要がある。) これを続ければ、結果として空リストが返る。この場合、 () ではなく nil が表示される。

復習のために、以下に cdr の繰り返しの例を書いておく。‘⇒’ の後に表示される結果が書いてある。

```
(cdr '(pine fir oak maple))
     ⇒ (fir oak maple)

(cdr '(fir oak maple))
     ⇒ (oak maple)

(cdr '(oak maple))
     ⇒ (maple)

(cdr '(maple))
     ⇒ nil

(cdr 'nil)
     ⇒ nil

(cdr ())
     ⇒ nil
```

複数の cdr を、間に表示される値を挟まずに続けて書くことも出来る。

```
(cdr (cdr '(pine fir oak maple)))
     ⇒ (oak maple)
```

この場合は、 Lisp インタプリタは内側のリストを先に評価する。最も内側にあるリストには引用符が付いているので、それはそのままリストとして返され、内側の cdr に渡される。この内側の cdr はこのリストの二番目以降の要素からなるリストを返し、外側の cdr に渡す。これは、元のリストの三番目以降の要素からなるリストを返すというわけである。この例のように、 cdr 関数を二回続けて使うと、元のリストの最初の二つの要素を除いたリストが返されることになる。

nthcdr 関数は、 cdr 関数を繰り返して使うのと同じことをしてくれる。次の例では引数 2 がリストと一緒に nthcdr 関数に渡され、その結果最初の二つの要素を除いたリストが返されている。これはまさに cdr を二回繰り返してリストに施したのと同じ結果である。

```
(nthcdr 2 '(pine fir oak maple))
     ⇒ (oak maple)
```

元の 4 つの要素を持ったリストを使って、 0、 1、 5 などを含むいろいろな引数を nthcdr に与えると何が起こるかを見てみよう。

```
;; リストはそのまま。
(nthcdr 0 '(pine fir oak maple))
     ⇒ (pine fir oak maple)

;; 最初の要素を除いた残りのコピーを返す。
(nthcdr 1 '(pine fir oak maple))
     ⇒ (fir oak maple)
```

```
;; 三つの要素を除いた残りのコピーを返す。
(nthcdr 3 '(pine fir oak maple))
     ⇒ (maple)
;; 四つ全部を除いた残りを返す。
(nthcdr 4 '(pine fir oak maple))
     ⇒ nil
;; これも全部を除いた残りを返す。
(nthcdr 5 '(pine fir oak maple))
     ⇒ nil
```

nthcdr も cdr と同様に元のリストを変化させないことに一言触れておくほうがいいだろう。この関数もまた非破壊的なのである。これは setcar 関数や setcdr 関数とはっきり違っているところである。

7.4 setcar

この二つの名前から連想されるように、 setcar と setcdr という関数は、リストの car や cdr に新しい値を持たせる。これらは car や cdr が値を変化させないのとは異なり、実際に元のリストの値を変化させてしまう。これがどういうことなのかを見る一つの方法は、実験してみることだろう。まずは setcar 関数から始めてみよう。

最初にリストを作り、 setq を使ってある変数の値にそのリストをセットする。動物のリストでやってみよう。

```
(setq animals '(giraffe antelope tiger lion))
```

GNU Emacs の Info でこの文章を読んでいるなら、このＳ式をいつものように評価してみよう。カーソルをＳ式の後に持っていき、 C-x C-e とタイプするのである。(私はこう書く時には実際にこの操作を試している。これは計算機環境にインタプリタが組込まれていることの利点の一つである。)

変数 animal を評価してみれば、この変数が（giraffe antelope tiger lion）というリストにバインドされていることが確かめられる。

```
animals
     ⇒ (giraffe antelope tiger lion)
```

別の言い方をすれば、変数 animal はリスト（giraffe antelope tiger lion）を指しているとも言える。

次に、関数 setcar に変数 animal と引用符付きのシンボル hippopotamus の二つ引数を渡して評価してみよう。これは三つの要素を持つ（setcar animals 'hippopotamus）というリストを書いて、これをいつものように評価することで行うことが出来る。

```
(setcar animals 'hippopotamus)
```

このＳ式を評価した後で、変数 animals をもう一度評価してみよう。すると、動物のリストが変化していることが分る。

```
animals
     ⇒ (hippopotamus antelope tiger lion)
```

リストの最初の要素 giraffe が hippopotamus に変わっている。

以上のことから分るように、 setcar は cons のように新しい要素を付け加えるのではなく、単に giraffe を hippopotamusに置き換えているだけである。即ち、この関数はリストの最初の要素を置き換えるのだ。

7.5 setcdr

setcdr 関数も、 setcar 関数と似ているのだが、こちらはリストの最初の要素ではなく、二番目以降の要素を置き換える。

実際どのように働くかを見るために、まず次のＳ式を評価して、変数の値を家畜のリストにセットしておく。

```
(setq domesticated-animals '(horse cow sheep goat))
```

この時点でこの変数を評価したなら、リスト（horce cow sheep goat)が返されるはずだ。

```
domesticated-animals
     ⇒ (horse cow sheep goat)
```

次に、 setcdr に、このリストを値に持つ変数名とそのリストの cdr の部分を置き換えるリストの二つの引数を与えて、評価してみる。

```
(setcdr domesticated-animals '(cat dog))
```

このＳ式を評価すると、リスト（cat dog）がエコー領域に表示される。これがこの関数によって返される値である。我々にとって興味があるのは副作用の方であるが、こちらは変数 domesticated-anmimal を評価してみればどんなものか理解出来る。

```
domesticated-animals
     ⇒ (horse cat dog)
```

実際、このリストは（horse cow sheep goat）から（horce cat dog）に変化している。このリストの cdr が（cow sheep goat)から（cat dog）に置き換わっているのである。

7.6 練習問題

四種類の鳥のリストを幾つかの cons を使ったＳ式を評価することで作りなさい。また、あるリストにそれ自身を cons した時に、何が起きるかを見なさい。四種類の鳥のリストの最初の要素を魚で置き換えなさい。更に、そのリストの残りの要素も他の魚で置き換えなさい。

8 テキストの切取りと保存

GNU Emacs で 'kill' コマンドを使ってバッファからテキストをカットないしは切り取った (clip した) 場合、常にその情報はリストとして保持されており、それをヤンク (yank) コマンドで取り出すことが出来る。

(Emacs で特に何らかの実体を破壊しないプロセスに対して 'kill' という言葉を使うのは、不幸な歴史的出来事である。これには 'clip' という単語の方がずっとふさわしい。これこそ kill コマンドがやっていることに他ならないからである。このコマンドはバッファからテキストを 'clip' し、それをまた取り出すことが出来るような置場所に移しているのだ。私はしばしば Emacs のソースの中に出てくるあらゆる 'kill' を 'clip' に、あらゆる 'killed' を 'clipped' に置き換えたい衝動にかられる。)

テキストがバッファから切り取られた場合、そのテキストはあるリストに保存される。次にまたテキストを切り取りると、そのテキストもまたこのリストの中に続けて保存される。従ってリストは次のようになる。

```
("a piece of text" "last piece")
```

関数 cons を使えば、以下のように、このリストにテキストを加えることが出来る。

```
(cons "another piece"
      '("a piece of text" "last piece"))
```

このＳ式を評価すると、次の三つの要素を持つリストがエコー領域に表示されるはずだ。

```
("another piece" "a piece of text" "last piece")
```

関数 car や nthcdr を使うと、この中からどの部分でも取り出すことが出来る。例えば次に挙げるコードでは nthcdr 1 ... が最初の要素を除いたリストを返し、 car がその残りのリストの中の最初の要素 — 元々のリストでは二番目の要素 — を返す。

```
(car (nthcdr 1 '("another piece"
                 "a piece of text"
                 "last piece")))
     ⇒ "a piece of text"
```

勿論、実際に Emacs で使われている関数はこれよりももっと複雑である。テキストをカットしたり取り出したりするためのコードは Emacs があなたが取り出したいテキストがどれか分るように書かれていなければならない。それが一番目にあろうと二番目にあろうと三番目にあろうと、あるいはもっと他の場所にあってもである。また、もしリストの最後まで来た場合、そこで終わりとはせずに再度最初の要素に戻らなくてはなるまい。

テキストの部分を保存しているリストは *kill* リング (*kill ring*) と呼ばれる。この章では少しずつ kill リングについて話していくことにする。そのために、まず zap-to-char 関数がどのように働くかを見ることで、 kill リングの使い方を見ていきたいと思う。山の頂上を目指す前にまず手頃な丘に登ってみるわけである。

実際にテキストがどのようにバッファから切り取られ、或いは取り出されるのかについては、次の章で説明する。 Chapter 10, "Yanking Text Back", p. 81, 参照。

8.1 zap-to-char

zap-to-char 関数は、 GNU Emacs の version 18 と version 19 とでは異なる書き方がされている。version 19 での実装の方が単純で、ちょっとだけ違った動作をする。まずは version 19 用に書かれた方を見て、その後に version 18 の方を見ることにしよう。

インタラクティブ関数 zap-to-char の Emacs version 19 用の実装では、カーソルの位置 (つまりポイントの位置) から次に指定した文字が現れるまでの、その文字も含めた領域のテキストを取り去る。 zap-to-char によって取り除かれたテキストは、 kill リングに置かれる。これは C-y (yank) とタイプすることで取り出すことが出来る。もし、このコマンドが引数と共に呼び出されたなら、その回数だけ指定された文字が出てくるまでの間のテキストを取り去る。従って、もしカーソルが次の文の先頭にあり、特定の文字が 's' だったとすると、 'Thus' が取り去られることになる。

Thus, if the cursor were at the beginning of this sentence and the character were 's', 'Thus' would be removed.

(訳註：日本語の文章では zap-to-char は使えないので、止むなく英文のままにした。)

またもし引数が 2 であれば、'Thus , if the curs' が取り去られる。ここでも最後の 's' が含まれていることに注目しよう。

Emacs version 18 での実装では、指定した文字までのその文字を含まない領域を取り去るようになっている。従って、先のパラグラフでの例では、's' は取り除かれない。

更に Emacs の version 18 での実装では、もし指定した文字が現れなかった場合にはバッファの最後までカーソルが移動した。しかし、version 19 の方では単にエラーを返すだけである。(従って、テキストを取り除いたりもしない。)

どれだけのテキストを取り去るかを決定するために、どちらのバージョンの場合でも検索関数を使っている。検索はもっぱらテキストを扱うコードの中で使われる。そして、削除コマンドだけでなく検索関数にも注目するのは意味のあることである。

以下が version 19 でのこの関数の完全なコードである。

```
(defun zap-to-char (arg char)  ; version 19 implementation
  "Kill up to and including ARG'th occurrence of CHAR.
Goes backward if ARG is negative; error if CHAR not found."
  (interactive "*p\ncZap to char: ")
  (kill-region (point)
               (progn
                 (search-forward
                  (char-to-string char) nil nil arg)
                 (point))))
```

8.1.1 interactive 式

zap-to-char コマンド内のインタラクティブ式は、次の通りである。

```
(interactive "*p\ncZap to char: ")
```

引用符の中の部分は、"*p\ncZap to char: " であり、三つのことを指定している。一番目の部分は単なるアスタリスク '*' で、バッファが read-only だったら警告のためにエラーを出すためのものである。つまり、read-only なバッファで zap-to-char を使おうとしてもテキストを取り除くことは出来ず、代わりに "buffer is read-only" というメッセージを受け取ることになることを意味する。端末によってはビープ音もなる。

*p\ncZap to char: " の二番目の部分は 'p' である。この部分は改行 '\n' で終っている。'p' はこの関数の最初の引数として、処理された前置引数の値が渡されることを意味する。前置引数は C-u に続いてある数を、もしくは M- に続いてある数をタイプすることで渡される。もし、この関数が前置引数なしで呼び出された場合、1 が引数として渡される。

*p\ncZap to char: " の三番目の部分は、'cZap to char: 'である。この部分の最初の小文字の 'c' は interactive がプロンプトを出して、文字の入力を要求することを指示するものである。プロンプトは 'c' の後に続くもので、この場合は文字列 'Zap to char: 'である。(コロンの後の空白は、単に見栄えをよくするためのものである。)

結局これがすることは、正しい型の zap-to-char の引数を用意し、ユーザーに対してプロンプトを出すということである。

8.1.2 zap-to-char の本体部分

zap-to-char 関数の本体部分は、現在のカーソルの位置から指定した文字までの間の領域のテキストを削除するコードを含んでいる。その最初の部分は次の通りである。

```
(kill-region (point) ...
```

(point) は現在のカーソルの位置である。

この後には、 progn を使った式が続いている。 progn の本体部分では、 search-forward と point を呼び出している。

まず search-forward を学んでからの方が progn の働きを理解しやすいので、先にこちらを見て、次に progn について述べることにしよう。

8.1.3 関数 search-forward

search-forward 関数は、 zap-to-char の中で、削除する範囲の端を定める文字 (以下 zapped-to-character と書く) の位置を定めるのに使われている。もし検索が成功すると、 search-forward は目的の文字列の最後の文字のすぐ後にポイントを移動する。(今の場合、目的の文字列はちょうど一文字である。) もし検索が後方に向かうものであれば、 search-forward は目的の文字列の最初の一文字のすぐ前にポイントを移動する。また、 search-forward 自身は値として t を返す。(従って、ポイントの移動は副作用である。)

zap-to-char の中で search-forward は次のように使われている。

```
(search-forward (char-to-string char) nil nil arg)
```

関数 search-forward は四つの引数を取る。

1. 最初の引数は、検索しようとするターゲットである。これは '"z"' のように文字列でなければならない。

   今回の場合は、たまたま zap-to-char に渡される引数は単独の文字である。計算機の設計上、 Lisp インタプリタは単独の文字を一文字からなる文字列とは区別して扱う。計算機の中では単独の文字は一文字からなる文字列とは別の電気的なフォーマットを持っているのである。(単独の文字はたいてい計算機の中で 1 バイトで記録されている。(訳註：勿論日本語とかでは事情は別である。) しかし、文字列の方は長いものもあれば短いものもあるので、計算機の方でそれに備えなければならないのである。) search-forward は文字列を検索するものなので、 zap-to-char 関数に引数として渡された文字を計算機の内部で別のフォーマットに変換しないといけない。でないと、 search-forward は検索に失敗してしまう。この変換をするために、 char-to-string という関数が使われている。
2. 二番目は、検索の範囲を指定するためのものである。これはバッファ内の位置として指定される。今の場合、検索はバッファの最後まで行うので、範囲を制限したりはしない。従って、二番目の引数は nil である。
3. 三番目は、この関数に検索が失敗し場合にどうするか — 警告してエラーを表示するか、単に nil を返すか — を伝えるものである。 三番目の引数として nil が指定されていると、検索に失敗した場合にはエラーを出して警告する。
4. search-forward の四番目の引数は、繰り返しの回数 — つまり検索対象の文字列が何回現れるのを調べるか — を指定するものである。この引数は省略可能であり、もしこの引数なしで関数が呼び出された場合、この引数として 1 が渡される。引数が負であれば、検索は後方に向かってなされる。

テンプレートで書くと、 search-forward 式は次のようになる。

```
(search-forward "target-string"
                limit-of-search
                what-to-do-if-search-fails
                repeat-count)
```

次に progn を見ることにしよう。

8.1.4 関数 progn

progn は引数を各々順番に評価していき、最後の式の値を返すような関数である。それ以前のＳ式はその副作用のためだけに評価され、これらが返す値は無視される。

progn 式のテンプレートは極めて単純である。

```
(progn
  body...)
```

zap-to-char の中では、 progn 式は二つのことを行う。まず、ポイントをちょうど正しい位置に置くこと。次に、ポイントの位置を返して、 kill-region がどこまでテキストを削除すればよいかを教えることである。

progn の最初の引数は search-forward である。 search-forward は検索する文字列を見つけた場合、その文字列の最後の文字のすぐ後にポイントを移す。(今の場合、検索文字列は一文字である。) ただし、もし検索が後方方向なら最初の一文字の直前の位置にポイントを移す。ポイントの移動は副作用である。

二番目の、そして最後の progn の引数は (point) である。このＳ式はポイントの位置を返す。今の場合なら、これは search-forwardによって移動されたポイントの位置である。この値が progn 式の値として返され、 kill-region に kill-region の二番目の引数として渡される。

8.1.5 zap-to-char についての総括

さて、以上で search-forward と progn の働きを見てきた。ここで zap-to-char 全体がどのように動作するかを見ることにしよう。

zap-to-char コマンドが与えられた時の kill-region の最初の引数はカーソルの位置 — その時点でのポイントの値 — である。また progn の中で検索コマンドによってポイントが zapped-to-character のすぐ後に移され、 point によってその位置の値が返される。 kill-region 関数はこれら二つの値の最初の値をリージョンの始まりに、二番目の値をリージョンの終わりに指定して、そのリージョンを削除する。

progn 関数が必要なのは、 kill-region 関数の引数が二つであるからである。仮に search-forward 式と point 式が続けて二つの引数として書かれていたとしたら、うまく動作してはくれない。 prognは kill-region にとって一つの引数であり、 kill-region が第二引数として必要な一つの値を返すのである。

8.1.6 Version 18 での実装

zap-to-char の version 18 での実装は version 19 での実装とはちょっとばかり違っている。こちらも zapped-to-character までのテキストを削除するのだが、それには zapped-to-character 自身は含めない。また、指定された文字列が見つからない場合にはバッファの最後まで削除してしまう。

違いは kill-region の第二引数にある。 Version 19 での実装では次のようになっていた。

```
(progn
  (search-forward (char-to-string char) nil nil arg)
  (point))
```

しかし version 18 では次のようになっている。

```
(if (search-forward (char-to-string char) nil t arg)
    (progn (goto-char
            (if (> arg 0) (1- (point)) (1+ (point))))
           (point))
  (if (> arg 0)
      (point-max)
    (point-min)))
```

これはかなり複雑に見える。しかし、各々の部分は容易に理解することが出来る。

最初の部分はこうだ。

```
(if (search-forward (char-to-string char) nil t arg)
```

これは次の仕事をするような if 式のテンプレートにきちん当てはまっている。

```
(if  able-to-locate-zapped-to-character-and-move-point-to-it
     then-move-point-to-the-exact-spot-and-return-this-location
   else-move-to-end-of-buffer-and-return-that-location)
```

if 式を評価することで、 kill-region の二番目の引数が指定される。最初の引数としてはポイントの位置が与えられるので、これらから、 kill-region はポイントから zap する位置までのテキストを削除出来る。

search-forward が副作用としてどのようにポイントを移動するかについては、既に説明したのだった。search-forward はもし検索に成功したなら t を返す。失敗した場合に nil を返すか、エラーメッセージを出すかは search-forward の三番目の引数による。今の場合三番目の引数は t なので、検索に失敗すると nil が返る。後で見るように、 nil が返される場合のコードの扱いは簡単である。

zap-to-char の version 18 での実装では、検索は if 式の真偽テストの部分で検索のためのＳ式が評価されることによって起動される。もし検索に成功すれば Emacs は if 式の then-part を評価する。一方、もし検索に失敗すれば、 Emacs は else-part の方を評価する。

if 式の中で検索が成功すれば、 progn 式が実行される — つまりプログラムとして走らされる。

前にも言ったように、 progn は引数を順に評価していき、その最後のＳ式の値を返す関数である。それ以前のＳ式は単に副作用だけのために評価され、それらが返す値は捨てられる。

上で説明したように、このバージョンの zap-to-char では progn 式は search-forward 関数が検索する文字を見つけた場合に実行される。 progn 式は二つのことを実行しなければならない。一つはポイントを正しい位置に移動することであり、もう一つはその位置を返すことにより kill-region にどこまで削除すればよいかを伝えることである。

progn が使われる理由は search-forward が検索文字列を見つけた場合に、目的の文字列の最後の文字の直後までポイントを移動してしまうことにある。 (今の場合は目標となる文字列は一つの文字である。) また、検索が後方方向の場合は、 search-forward はポイントを目的の文字列の最初の文字の直前に置いてしまう。

しかし、このバージョンの zap-to-char は検索した文字自身は削除しないようになっている。例えば zap-to-char が ‘z’ という文字までのテキストを削除する場合、このバージョンのものでは ‘z’ という文字は削除してはいけない。そこで、ポイントをその文字を削除しないような所まで移動しなければならないというわけである。

progn 式の本体

progn 式の本体は二つのＳ式からなっている。異なる部分間の区切を明確にするために間を空けて、更にコメントを加えると、 progn 式は次のようになる。

```
(progn

  (goto-char                ; progn の最初の式。
        (if (> arg 0)       ; もし arg が正なら、
            (1- (point))    ;   一文字分戻り、
          (1+ (point))))    ;   そうでなければ一文字分進む。

  (point))                  ; progn の二番目の式。
                            ;   ポイントの位置を返す。
```

この progn 式がやることは次の通りである。もし検索が前方ならば、 (つまり arg が正の数なら) Emacs は検索した文字の直後にポイントを移動している。そこで一文字分前に戻ることで、その文字を対象範囲から除く。この場合、 progn 式の中のＳ式は次のように読める。 (goto-char (1- (point)))。これはポイントの位置を一文字分だけ戻す。 (1- という関数は引数から 1 を引く関数である。同様に 1+ は引数に 1 を足す。) 一方、もし zap-to-char の引数が負ならば、検索は後方に向かって行われる。 if がこれを検知し、この場合は (goto-char (1+ (point)))というＳ式が実行されることになる。 (繰り返しになるが、 1+ は引数に 1 を加える関数である。)

progn の二番目の、そして最後の引数は、(point) というＳ式である。このＳ式は progn の最初の引数によって移動したポイントの位置を返す。この値はこのＳ式を含む if 式によって返され、 kill-region の第二引数として渡される。

手短にまとめると、この関数の働きは以下の通りである。 kill-regionの最初の引数は zap-to-char コマンドが与えられた時のカーソルの位置 — つまりその時点でのポイントの位置 — である。検索に成功すれば検索を行う関数がそこまでポイントを移動する。そして progn 式によって、指定した文字が削除する領域に入らないようにポイントの位置をずらしてから、その位置を返す。そして、 kill-region 関数がそのリージョンを削除するというわけである。

最後に、 if 式の else-part で、指定した文字が見つからなかった場合の対処をしている。 zap-to-char 関数の引数が正である場合 (もしくは引数が与えられていなかった場合)、指定した文字が見つからなければ、その時のカーソルの位置から先のアクセス可能な範囲の全てのテキスト (もしナローイングがかかっていなければバッファの最後まで) が削除される。 arg が負の場合は、指定した文字が見つからなければ上の操作がその時にアクセス可能な範囲の始めの方に向って行われる。この部分のコードは単純な if 式である。

```
(if (> arg 0) (point-max) (point-min))
```

これは、もし arg が正の数なら、 point-max そうでなければ point-min の値を返すことを表している。

復習のために、コメント付きで kill-region を含むコードを挙げておこう。

```
(kill-region
 (point)                    ; リージョンの始まり
 (if (search-forward
      (char-to-string char) ; 検索対象
      nil                   ; 検索の限界: 無し
      t                     ; 失敗したら nil を返す。
      arg)                  ; 繰り返しの回数
     (progn                 ; then-part
       (goto-char
        (if (> arg 0)
            (1- (point))
          (1+ (point))))
       (point))

   (if (> arg 0)            ; else-part
       (point-max)
     (point-min))))
```

これを見ればわかるように、 version 19 での実装は、 version 18 での実装よりもやることがちょっと少ないが、その分コードはずっと単純になっていたのである。

8.2 kill-region

zap-to-char は kill-region 関数を使っている。これはとても単純な関数で、説明文字列の一部を省略すると、次のように書かれている。

```
(defun kill-region (beg end)
  "Kill between point and mark.
The text is deleted but saved in the kill ring."
  (interactive "*r")
  (copy-region-as-kill beg end)
  (delete-region beg end))
```

主に注目すべき点は delete-region と copy-region-as-kill を使っているところである。これらの関数については以降のセクションで説明する。

8.3 delete-region: ちょっと脱線して C の話を

zap-to-char コマンドは kill-region 関数を使っている。そして、これはこれでまた別の二つの関数 copy-region-as-kill 及び delete-region を利用している。 copyo-region-as-kill についてはこの後のセクションでも説明するが、リージョンを kill-ring にコピーしてヤンク出来るようにするものである。 (Section 8.5, “copy-region-as-kill”, p. 71, 参照。)

delete-region 関数は、リージョンの中身を消去してしまうもので、この場合、それを取り戻すことは出来ない。

これまでに議論してきたコードとは異なり、 delete-region は Emacs Lisp では書かれていない。これは C で書かれており、 GNU Emacs system のプリミティブの一つである。これは大変単純なものなので、ちょっとの間 Lisp から脱線して、これについて説明することにしよう。

他の多くの Emacs のプリミティブと同様、 delete-region も C のマクロのインスタンス、つまりコードのテンプレートとしてのマクロとして書かれている。このマクロの最初の部分は次のようである。

```
DEFUN ("delete-region", Fdelete_region, Sdelete_region, 2, 2, "r",
  "Delete the text between point and mark.\n\
When called from a program, expects two arguments,\n\
character numbers specifying the stretch to be deleted.")
```

マクロの記述の細かい点に立ち入ることはしないが、このマクロが DEFUN という単語から始まっていることを指摘しておく。ここで DEFUN という単語を選んだのは、このコードが Lisp における defun と同じ目的を持っているためである。単語 DEFUN の後に続く括弧の中に 7 つの部分がある。

- 最初の部分は、この関数に Lisp の中で与えられる関数の名前である。この場合は delete-region である。
- 二番目は、この関数の C での名前 Fdelete_region である。便宜上、‘F’ から始まっている。 C では名前にハイフンを使わないので、代わりに下線 (underscore) が使われている。
- 三番目は、内部でこの関数の情報を記録しておくための C の constant な構造体の名前である。これは、 C での関数名であるが、 ‘F’ ではなく ‘S’ で始まっている。
- 四番目と五番目は、この関数が引数として取れる最小の数と最大の数を指定している。今の場合はちょうど 2 つの引数を取る。
- 六番目は Lisp の関数定義のインタラクティブ宣言に続く引数と同じようなものである。文字と、大抵はそれに続くプロンプトが書かれる。今の場合、その文字は "r" である。これは、この関数の引数がそのバッファのリージョンの最初と最後の位置であることを示している。この場合はプロンプトはない。
- 七番目は、説明文字列である。これも Emacs Lisp の関数定義の場合とほぼ同じであるが、全ての改行が明示的に ‘\n’ に続いてバックスラッシュと復帰コードを書くことで示されている。

次に形式的なそのオブジェクトの種類の宣言と一緒にパラメータが来て、そして、このマクロの本体部分と呼ばれる部分が来る。 delete-region の場合は、本体は次の三行からなる。

```
validate_region (&b, &e);
del_range (XINT (b), XINT (e));
return Qnil;
```

最初の関数 validate_region は、リージョンの始まりと終わりとして渡された値が正しい型で、適切な範囲かどうかをチェックしている。二番目の関数、 del_range は、実際にテキストを削除する関数である。もしこの関数がエラーを起こさず仕事を終えたら、三行目がそのことを示すために Qnilを返す。

del_range は複雑な関数なので、立ち入らないことにする。これはバッファを更新するなどの働きをする。しかしながら、 del_range に渡される二つの引数には注目しておいた方がいいだろう。XINT(b) とXINT(e) の二つである。 C 言語に関する限り、 b と e は削除するリージョンの始まりと終わりを表わす二つの 32 ビットの整数である。

8.4 defvar を用いた変数の初期化

delete-region 関数と違って、 copy-region-as-kill 関数は Emacs Lisp で書かれている。これはバッファのリージョンを複写して kill-ring と呼ばれる変数に保存するものである。このセクションではこの変数がどのようにして生成され初期化されるのかを説明する。

(ここでも、 kill-ring という言葉は誤用と言えるだろう。バッファから切り取られたテキストは取り戻すことが出来るのである。これは、死体の輪なんかではなく、復活出来るテキストの輪なのだ。)

Emacs Lisp では、 kill-ring のような変数は defvar という特殊形式によって生成され、初期値を与えられる。この名前は "define variable" から来ている。

defvar という特殊形式は、変数に値を設定する点で setqと似ている。 setq と違う点は二つある。一つ目は、その変数がまだ値を持っていない場合にのみ値を設定するということである。既に値がある場合には、defvar はその値を上書きしたりはしない。二つ目は defvar には説明文字列があるということである。

現在の変数の値は、どんな値であれ、 describe-variable 関数を使って見ることが出来る。これは普通は C-h v をタイプすることで呼び出せる。 C-h v とタイプして、プロンプトが出た所で kill-ring (と改行) を入力すると、現在の kill リングの中身がどうなっているか見ることが出来る。これは、滅茶苦茶多いこともある！逆に、もし今回 Emacs を起動してからこの文書を読むだけで他には何もしていなければ、中には何もないはずだ。

また、'*Help*' バッファの最後に kill-ring の説明がなされていることも見てとれる。

```
Documentation:
List of killed text sequences.
```

この kill リングは defvar を使って次のように定義されている。

```
(defvar kill-ring nil
  "List of killed text sequences.")
```

この変数定義の中では、この変数は初期値 nil を与えられている。これはもっともなことである。何もセーブしないうちから、ヤンク (yank) コマンドで何かを取り出したいとは思わないだろう。説明文字列は defun の説明文字列と同じように書かれている。 defun のものと同様、最初の行はそれだけで完全な文になっているべきだ。何故なら apropos のような幾つかのコマンドは、最初の一行しか表示しないからである。また、その後に続く行はインデントすべきではない。そうしないと C-h v (describe-variable) を使って見た場合に見栄えがよくない。

大抵の変数は Emacs の内部のものである。が、幾つか edit-options コマンドを使って簡単に設定出来るようになっているものもある。(ただし、ここでの設定はその時のセッション限りのものである。その設定を永続的に使用したい場合は、 '.emacs' ファイルを使う。詳しくは Chapter 16, "Your '.emacs' File", p. 141, を参照のこと。)

簡単に設定出来る変数は、 Emacs の中で、説明文字列の一桁目にアスタリスク '*' を付けることで他の変数と区別されている。

例えば、

```
(defvar line-number-mode nil
  "*Non-nil means display line number in mode line.")
```

これは edit-options コマンドを使って line-number-mode の値を変化させることが出来ることを示している。

勿論、 setq 式を使って line-number-mode の値を変化させることも出来る。例えば次のようにすればよい。

```
(setq line-number-mode t)
```

Section 1.9.2, "Using setq", p. 12, 参照。

8.5 copy-region-as-kill

copy-region-as-kill 関数はバッファからリージョンのテキストを複写して、 kill-ring と呼ばれる変数の中に保存する。

もし、 kill-region コマンドを前回の kill-region コマンドの直後に呼び出したとすると、 Emacs は新しく複写したテキストを以前複写したテキストに付加 (append) する。これは、もしそのテキストをヤンクした場合、以前複写した分と今回複写した分を合わせた全体を戻すことになるということを意味する。一方、もし copy-region-as-kill に先立って他のコマンドを使用した場合には、複写したテキストは kill リングの新しい別の部分に、それまでに複写したものと区別されて保存される。

以下が version 18 での copy-region-as-kill の完全な定義である。ただし、見やすいように幾つかのコメントを加えて整形してある。

```
(defun copy-region-as-kill (beg end)
  "Save the region as if killed, but don't kill it."
  (interactive "r")
  (if (eq last-command 'kill-region)

      ;; then-part: 新しくコピーされたテキストを
      ;;   前回コピーされたものと結合
      (kill-append (buffer-substring beg end) (< end beg))

    ;; else-part: 新しくコピーされたテキストを新たな要素として
    ;;   kill リングに追加し、必要なら kill リングを短くする。
    (setq kill-ring
          (cons (buffer-substring beg end) kill-ring))
    (if (> (length kill-ring) kill-ring-max)
        (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
  (setq this-command 'kill-region)
  (setq kill-ring-yank-pointer kill-ring))
```

これまで同様、この関数もまとまりのある幾つかの部分に分けることが出来る。

```
(defun copy-region-as-kill (argument-list)
  "documentation..."
  (interactive "r")
  body...)
```

引数は beg と end であり、 "r" によってインタラクティブ宣言がなされている。つまり、二つの引数は各々リージョンの始まりと終わりを示すものである。もしこの文書をきちんと最初から読んできたなら、この部分を理解するのは殆どルーティーンワークにすぎないだろう。

この関数の本体は、 if 式で始まっている。この式でやっているのは二つの状況を区別することである。つまり、このコマンドが kill-region コマンドが呼び出された直後に実行されたかどうかの判断である。もし直後に実行されたなら、新しく複写されるリージョンは以前複写された部分に付加される。そうでない場合は、 kill リングの先頭に、以前複写されたものとは区別されて保存される。

この関数定義の最後の二行は、二つの setq 式である。一つは変数 this-command を kill-region に、もう一つは変数 kill-ring-yank-poiter を kill リングに設定する。

この copy-region-as-kill の本体部分は詳しく見る価値がある。

8.5.1 copy-region-as-kill の本体部分

copy-region-as-kill 関数は、二行以上の kill を一つの項目につなげるように書かれている。もし、それらを kill リングからヤンクして取り出すと、一箇所にまとめて取り出される。更に、現在のカーソルの位置から前

方 (つまりテキストの終わりの方) に向かって kill すると、その部分は以前 kill したテキストの後に付け加えられ、後方 (つまりテキストの始まりの方) に向かって kill すると、以前 kill したテキストの前に付け加えられる。こうしてテキストの部分は正しい順序に保たれることになる。

この関数は、現在、及びその一つ前に Emacs が実行したコマンドを憶えておくために二つの変数 `this-command` と `last-command` を使っている。

通常、Emacs はどんな関数を実行している場合でも `this-command` の値を現在実行中のコマンドの値 (今の場合なら `copy-region-as-kill`) にセットしている。しかしながら、`copy-region-as-kill` の場合は違う。この関数は `this-command` の値を `kill-region` にセットする。これは `copyo-region-as-kill` を呼び出している関数の名前である。

`copy-region-as-kill` の本体部分の最初のパートでは、`if` 式で `last-command` が `kill-region` かどうかを判定している。もしそうであれば、`kill-append` 関数を使って今回複写したテキストを kill リングの最初の要素 (つまり CAR) のテキストと結合する。一方、もしも `last-command` の値が `kill-region` で無ければ `copy-region-as-kill` 関数は kill リングに新しい要素を付け加えることになる。

`if` 式は次の通りである。これは `eq` を使っているが、この関数はここで初めて出てきたものである。

```
(if (eq last-command 'kill-region)
    ;; then-part
    (kill-append (buffer-substring beg end) (< end beg))
```

`eq` 関数は、最初の引数が二番目の引数と同じ Lisp オブジェクトかどうかを判定するものである。`eq` 関数は `equal` 関数と、等しいかどうかの判定をするという点では似ているが、異なる名前を持つものが、計算機の内部で実際に同じオブジェクトかどうかを判定するのかそうでないか、という点で異なっている。`eq` の方は同じオブジェクトの場合でないと真を返さないが、`equal` 関数の方は二つの式が同じ構造でありかつ同じ内容でありさえすれば真を返す。

(訳註：ここはちょっと分り難いかもしれない。詳しくは Section "Equalty Pradicates" in *The GNU Emacs Lisp Reference Manual*, を参照。なお、Chapter 9, "リストはどのように実装されているか", p. 77, も参考になる。)

関数 `kill-append`

`kill-append` 関数は次のようなものである。

```
(defun kill-append (string before-p)
  (setcar kill-ring
          (if before-p
              (concat string (car kill-ring))
              (concat (car kill-ring) string))))
```

この関数をパートごとに見ていこう。`setcar` 関数は `concat` を使って新しいテキストを kill リングの CAR と結合している。新しい方のテキストを前に持ってくるか、後に持ってくるかは `if` 式の結果による。

```
(if before-p                            ; if-part
    (concat string (car kill-ring))     ; then-part
  (concat (car kill-ring) string))      ; else-part
```

もし、今回切り取ったリージョンが直前のコマンドで切り取ったテキストよりも前にあれば、今回切り取ったものは以前保存したものの前につなげるべきであるし、逆に、今回切り取ったテキストが以前切り取ったテキストの後に続くものなら、以前保存したものの後につなげるべきである。`if` 式は今回保存するテキストを以前保存しておいたテキストの前につけるか後につけるかを `before-p` という述語で判断している。

シンボル `before-p` は `kill-append` の引数の名前の一つである。`kill-append` 関数が評価されると、これは実際の引数を評価して返された値にバインドされる。今の場合であれば、(< end beg) の値がバインドされる。このＳ式は、今回のコマンドで切り取られたテキストが直前のコマンドで切り取られたテキストの前に属しているか、それとも後に属しているかを、直接判定しているわけではない。この式がやっているのは変数 `end` の値

が変数 beg の値よりも小さいかどうかの判定だけである。もし小さければ、それはユーザがおそらくはバッファの先頭に向かっているということを意味する。というわけで、この場合は以前のテキストの前に今回のテキストを追加するようになっている。一方、もし end の値の方が beg の値よりも大きかった場合、今回のテキストは以前のテキストの後尾に追加される。

新しく保存したテキストが以前保存したテキストの前に追加される場合は、その新しいテキストを前にして古いテキストと結合される。

```
(concat string (car kill-ring))
```

逆に、テキストが後に追加される場合は、新しいテキストを古いテキストの後にして結合する。

```
(concat (car kill-ring) string))
```

この関数の動作を理解するためには、 concat 関数の復習から始めなければならないだろう。 concat 関数は二つのテキスト文字列を一つにつなげるものである。結果も文字列になる。例えば次のような感じである。

```
(concat "abc" "def")
     ⇒ "abcdef"

(concat "new "
        (car '("first element" "second element")))
     ⇒ "new first element"

(concat (car
        '("first element" "second element")) " modified")
     ⇒ "first element modified"
```

以上で kill-append 関数が理解出来るようになる。これは、 kill リングの中身を修正するものである。 kill リングはリストであり、各々の要素は保存されたテキストである。 setcar 関数が実際にこのリストの最初の要素を置き換える役目を果たしている。そのために、まず concat で kill リングの最初の要素 (kill リングの CAR) と新しいテキストを結合し、元の要素をその新しく出来た要素で置き換えている。新しく保存するテキストを元々のテキストの前に結合するか後に結合するかは、それが古いテキストを切り取った場所の前にあるか後にあるかによる。結合された結果が kill リングの新しい最初の要素となる。

ついでながら、私の現在の kill リングの最初の方を挙げておこう。

```
("concatenating together" "saved text" "element" ...
```

copy-region-as-kill の else-part

さて、 copy-region-as-kill の説明に戻ろう。

もし直前のコマンドが kill-region でなかったとすると、この場合は kill-append ではなく、次の S 式の else-part が呼ばれることになる。

```
(if true-or-false-test
    what-is-done-if-test-returns-true
  ;; else-part
  (setq kill-ring
        (cons (buffer-substring beg end) kill-ring))
  (if (> (length kill-ring) kill-ring-max)
      (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
```

else-part の setq の行では、 kill リングの新しい値として、古い kill リングの値に新しく削除した文字列を加えた結果をセットしている。

これがどういうことをやっているかを簡単な例で見てみることにしよう。

```
(setq example-list '("here is a clause" "another clause"))
```

このＳ式を C-x C-e で評価すると、 example-list が評価出来るようになる。その結果は次の通り。

```
example-list
     ⇒ ("here is a clause" "another clause")
```

さて、次のＳ式を評価することで、このリストに新しい要素を付け加えてみよう。

```
(setq example-list (cons "a third clause" example-list))
```

これを評価すると、次の値が返されるのが分る。

```
example-list
     ⇒ ("a third clause" "here is a clause" "another clause")
```

かくして、三番目の要素が cons によってリストに加えられた。

この関数での setq や cons も上と同じことをやっているのである。違うのは、リージョンのテキストを取り出して cons に渡すために、 buffer-substring を使っていることである。その行をもう一度抜き出しておく。

```
(setq kill-ring (cons (buffer-substring beg end) kill-ring))
```

copy-region-as-kill の else-part の次の部分は、もう一つの if 式である。この if 式は kill リングが長くなりすぎるのを防ぐものである。この部分は以下のようになっている。

```
(if (> (length kill-ring) kill-ring-max)
    (setcdr (nthcdr (1- kill-ring-max) kill-ring) nil)))
```

このコードはまず kill リングの長さが最大許容範囲にを越えていないかどうかチェックしている。最大許容範囲とは kill-ring-max の値 (デフォルトでは 30) のことである。もしこの長さがそれより長ければ、このコードは kill リングの最後の要素を nil にセットする。そのために二つの関数 nthcdr と setcdr を利用している。

先に setcdr の方を見ることにする。 (Section 7.5, “setcdr”, p. 61, 参照。) これはリストの CDR を設定するものである。 setcar がリストの CAR を設定するのと同様だ。しかしながら、この場合 setcdr は kill リング全体の cdr を設定するのではない。

nthcdr 関数は繰り返しリストの CDR を取ることで動作する。つまり、 CDR の CDR を取り、そのまた CDR を取り ... という操作を N 回繰り返して、その値を返すのである。

もし四つの要素のリストがあって、それを (最初の) 三つの要素のリストにしたい場合、二回 CDR を取って出来たリストの CDR を nil に設定すればよい。このようにして、リストを短く出来るのである。

以上のことは、下の三つのＳ式を順に評価していくことで理解出来る。まずは trees の値を （maple oak pine birch） にセットし、次にその二番目の CDR の CDR を nil にセットする。そして trees の値を見てみるのである。

```
(setq trees '(maple oak pine birch))
     ⇒ (maple oak pine birch)

(setcdr (nthcdr 2 trees) nil)
     ⇒ nil


trees
     ⇒ (maple oak pine)
```

(setcdr 式そのものの値は nil になる。これは、 CDR を nil にセットしたからである。)

繰り返しのために nthcdr 関数を使って、 copy-region-as-killに対し kill リングの最大許容範囲よりも 1 少ない回数だけ CDR を取り、最後に返された CDR の値 (これは kill リングの最後の要素である) を nil にセットする。こうして kill リングが長くなり過ぎることを防いでいるのである。

copy-region-as-kill 関数の次の行は

```
(setq this-command 'kill-region)
```

である。この行は内側のＳ式にも外側の if 式にも属していない。従って、 copy-region-as-kill が呼ばれるたびに毎回評価される。ここの部分で this-command が kill-region に設定されていることが分るだろう。前に見たように、次のコマンドが与えられると、この値は last-command の値になる。

最後に copy-region-as-kill の最後の行を見てみよう。

```
(setq kill-ring-yank-pointer kill-ring)
```

kill-ring-yank-pointer は kill-ring に設定されるグローバル変数である。

kill-ring-yank-pointer はポインタ (‘pointer’) とは呼ばれているけれども、実際は kill-ring と同じような変数である。ただし、このような名前がついているのは、この変数の使われ方がよく分るようにするためである。この変数は yank や yank-pop などの関数で使用される。 (Chapter 10, “テキストのヤンク”, p. 81, 参照。)

この辺りの話をするにはやはり、バッファから切り取ったテキストを取り戻すためのコマンド — ヤンクコマンドについて話さなければなるまい。が、その話をする前に、リストが計算機の内部でどのように扱われるかを説明しておいた方が良いだろう。そうすることで、「ポインタ」といったミステリアスな言葉を明確に理解出来るようになるだろう。

8.6 復習

この辺りで導入した関数についての簡単なまとめを載せておく。

car
cdr　　car はリストの最初の要素を返す。 cdr はリストの二番目以降の要素からなるリストを返す。

例)

```
(car '(1 2 3 4 5 6 7))
     ⇒ 1
(cdr '(1 2 3 4 5 6 7))
     ⇒ (2 3 4 5 6 7)
```

cons　　cons は最初の引数を二番目の引数の前に加えることで新たなリストを作成する。

例)

```
(cons 1 '(2 3 4))
     ⇒ (1 2 3 4)
```

nthcdr　　リストに対して ‘n’ 回 cdr を取った結果を返す。言わば ‘残り’ の ‘残り’ である。

例)

```
(nthcdr 3 '(1 2 3 4 5 6 7))
     ⇒ (4 5 6 7)
```

setcar
setcdr　　setcar はリストの最初の要素を置き換える。 setcdr はリストの二番目以降の要素を置き換える。

例)

```
(setq triple '(1 2 3))

(setcar triple '37)

triple
     ⇒ (37 2 3)

(setcdr triple '("foo" "bar"))

triple
     ⇒ (37 "foo" "bar")
```

progn 引数を順に評価していき、最後の値を返す。

例)

```
(progn 1 2 3 4)
     ⇒ 4
```

save-restriction
: カレントバッファのナローイングの情報をどんな状態であれ保存し、引数を評価し終った後、その状態を復元する。

search-forward
: 文字列を検索し、もし文字列が見つかれば、そこにポイントを移動する。

  引数は次の四つ

  1. 検索文字列
  2. 省略可能。検索範囲。
  3. 省略可能。検索に失敗した時に nil を返すかエラーメッセージを出すかを指定。
  4. 省略可能。検索を何回実行するかを指定。負数の場合は後方検索になる。

kill-region

delete-region

copy-region-as-kill
: kill-region はバッファのポイントとマークの間のテキストを切り取ってそれを kill リングに保存する。このテキストはヤンクによって取り出すことが出来る。

  delete-region はポイントとマークの間のテキストをバッファから削除して捨ててしまう。復活させることは出来ない。

  copy-region-as-kill はポイントとマークの間のテキストを kill リングに複写する。これはヤンクによって取り出すことが出来る。この関数はバッファからテキストを切り取ったり削除したりはしない。

8.7 検索についての練習問題

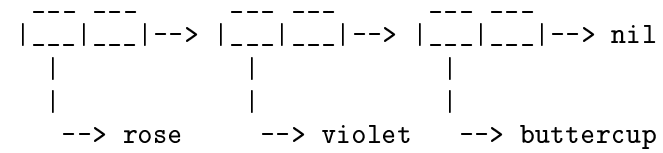
- 文字列を検索するインタラクティブな関数を書きなさい。もし文字列の検索に成功すればその直後にポイントを移動し、「見つけた！」というメッセージを表示するようにしなさい。(この関数の名前に search-forward という名前を使ってはいけない。そうしてしまうと、元の Emacs で定義された現在の search-forward を上書きしてしまう。例えば test-searchという名前を使いなさい。
- 現在の kill リングの三番目の内容をエコー領域に表示するような関数を書きなさい。その際、もし kill リングに三番目の要素が無い場合にも適切なメッセージを表示するようにしなさい。
- version 19.29 では copy-region-as-kill はもはや this-command に値をセットしない。この変更で、どのような影響が出るか？ また、何故このような変更が行われたと思うか。

9 リストはどのように実装されているか

Lisp では、アトムは直接的な方法で記録される。たとえ現実の実装方法が直接的ではなかったとしても、理論上は直接的である。例えば ‘rose’ というアトムは四つの隣接する文字 ‘r’, ‘o’, ‘s’, ‘e’ として記録される。一方、リストはもっと異なる方法で記録される。メカニズムとしては同じくらい単純なのだが、そのアイディアに慣れるのにはちょっと時間がかかるかもしれない。リストは、ポインタのペアの列として記録される。この列の中で、各々のペアの一番目のポインタは、あるアトムか、もしくは他のリストを指している。そして、二番目のポインタは次のペアか、シンボル nil を指している。 nil の場合はリストの終わりということである。

ポインタそれ自身は、指されている対象の極めて単純な電気的な住所だと言える。従って、リストも電気的な住所の列として記録されていると言える。例えば、(rose violet buttercup) というリストは ‘rose’, ‘violet’, ‘buttercup’ という三つの要素を持っている。計算機の中では、‘rose’ の電気的な住所は、ある計算機のメモリのセグメントに ‘violet’ というアトムの電気的な住所を示すアドレスに並んで記されている。そしてこっちの住所 (‘violet’ の場所を示すもの) は ‘buttercup’ というアトムの電気的な住所を示すアドレスに並んで記されている。

このように書くと何やら複雑に思えるかもしれないが、図に書いてみれば簡単に理解出来るだろう。

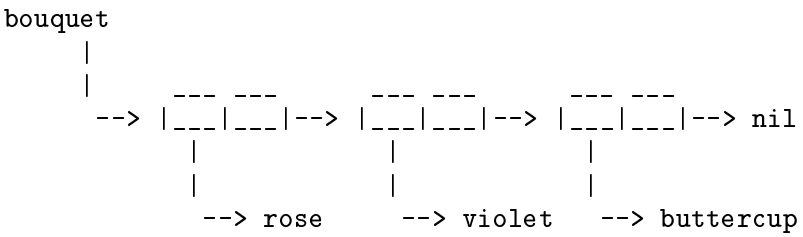


上の図の中で、各々の箱は Lisp オブジェクトを保持する計算機のメモリのワードを表現している。これは普通はメモリアドレスの形をしている。これらの箱、即ち住所は、ペアになっている。各々の矢印は、そのアドレスを持つ住所を指している。その場所には、アトムか他のアドレスのペアがある。最初の箱は ‘rose’ の電気的な住所であり、矢印は ‘rose’ を指している。二番目の箱は次の箱のペアを指しており、こっちのペアの一番目の箱は ‘violet’ の住所、そして二番目の箱はまた次のペアのアドレス、という具合になっている。そして最後の箱は、シンボル nil を指している。これは、リストの最後を示している。

setq などの関数によってある変数がこのリストにセットされた場合、このリストの先頭の箱のアドレスがこの変数にセットされる。従って、

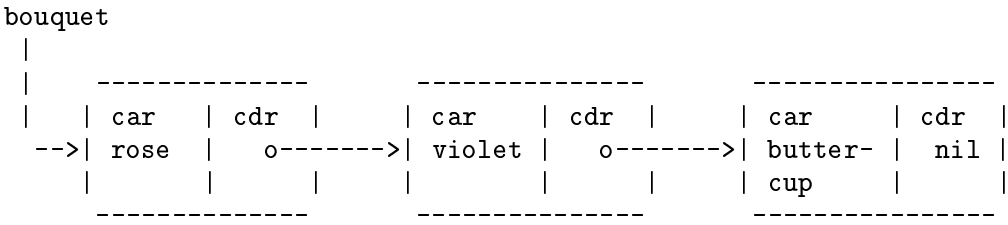
```
(setq bouquet '(rose violet buttercup))
```

というＳ式を評価すると、以下のような状況になる。



この場合、シンボル bouquet が最初のペアのアドレスを持つことになる。実際、シンボル bouquet は幾つかのアドレスボックスから出来ている。一つは表示される単語の ‘bouquet’ の住所であり、二つ目はこのシンボルに定義されている関数のアドレスであり、三つ目はリスト (rose violet buttercup) というリストの最初のアドレスボックスのペアのアドレスであり、というふうである。

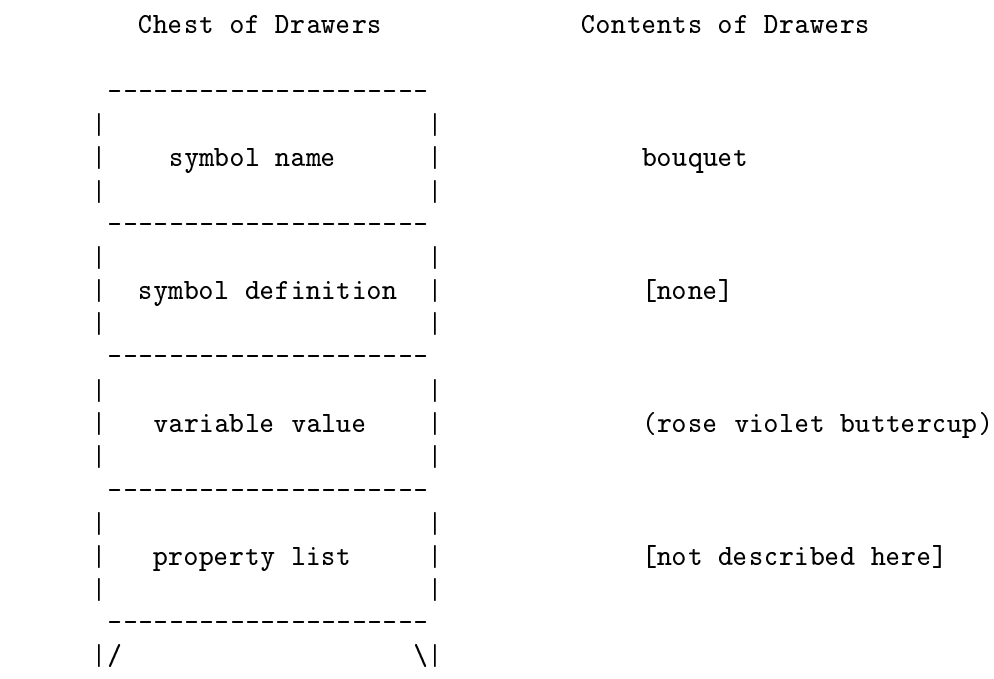
同じリストをちょっと箱の書き方を変えた別のイラストで描いてみると、次のようになる。



最初の方のセクションでは、シンボルを引き出しが幾つかある整理棚だと見做すと良いと書いた。関数定義がある引き出しに入っており、値は他の引き出しに入っている、という説明だ。値がある引き出しの中身は他の引き出しの中身に影響を与えずに変えることが出来るし、その逆も可能だとも書いた。実際には、各々の引き出しに入っているのは値や関数定義のアドレスである。これは、屋根裏部屋で古い棚を見つけて、その引き出しの一つを空けてみると、宝が埋まっている場所が描かれた地図が見つかったということに例えられよう。

(名前だけではなく、シンボルの定義や変数の値、また、その他の情報を取り出すための 属性リスト (*property list*) の「引き出し」というものもある。属性リストについてはここでは説明はしない。 Section “Property Lists” in *The GNU Emacs Lisp Reference Manual*, を参照のこと。)

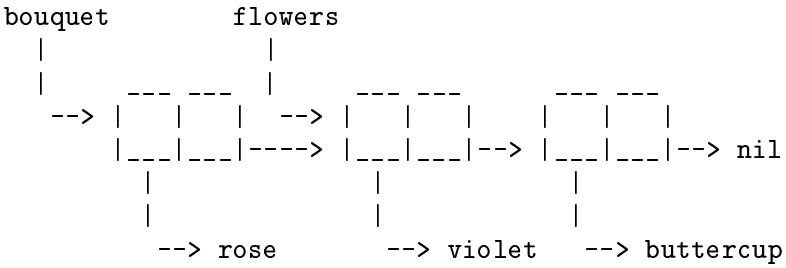
次は空想上の整理棚の絵である。



もし、シンボルがあるリストの CDR にセットされたなら、そのリスト自身は変化しない。そのシンボルは単にそのリストのアドレスを持つようになるだけである。(専門用語では、 CAR や CDR は「非破壊的 (non-destructive)」 であると言う。) 従って、次のＳ式を評価すると、

```
(setq flowers (cdr bouquet))
```

以下のような状況が生ずる。



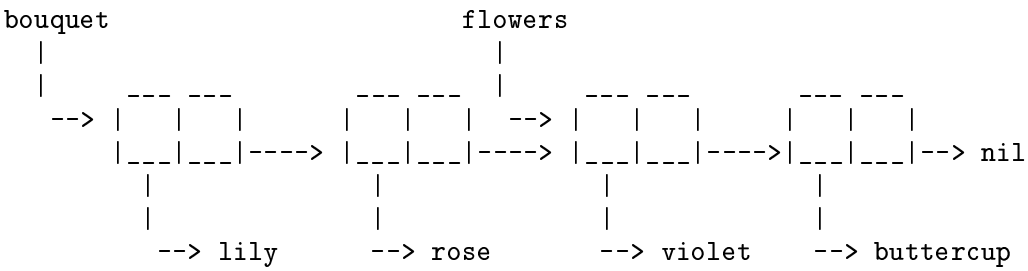
flowers の値は（violet buttercup）であるが、これはシンボル flower がアドレスボックスのペアのアドレスを持ったということである。このペアの最初の箱には violet のアドレスがあり、二番目の箱には butercup のアドレスが入っている。

アドレスボックスのペアはコンスセル (*cons cell*) もしくはドット対 (*dotted pair*) と呼ばれる。Section “List Type ” in *The GNU Emacs Lisp Reference Manual*, 及び Section “Dotted Pair Notation” in *The GNU Emacs Lisp Reference Manual*, にコンスセルとドット対についてのより詳しい情報がある。

関数 cons は、上に述べたようなアドレスの列の先頭に新しいアドレスペアを付け加える。例えば、

```
(setq bouquet (cons 'lily bouquet))
```

を評価すると



という状況になる。この操作ではシンボル flowers の値を変更しているわけではない。これは次を評価することで確かめられる。

```
(eq (cdr (cdr bouquet)) flowers)
```

これは、t を返すはずだ。再設定されない限り、flowers はずっと（violet buttercup）という値を持ったままである。つまり、最初のアドレスが violet であるコンスセルのアドレスを持っているということだ。また上の操作は、既に存在しているコンスセルを変えたりはしない。それらは全てそのままである。

従って、Lisp でリストの CDR を得るには、単にその列の隣りのコンスセルのアドレスを得ればよく、また CAR を得るには、単にそのリストの最初の要素のアドレスを得ればよい。更に、リストに新しい要素を cons するには、リストの先頭に新しいコンスセルを加えればよい。以上がこの話題に関する全てである。Lisp の裏側は驚くほど明解で単純なのだ。

では、コンスセルの列の最後のアドレスはどこに関連づけられているのか？ これは nil, 即ち、空リストである。

まとめると、Lisp の変数にある値にセットされる時は、その変数には、その変数が見にいくリストのアドレスが与えられるというわけである。

9.1 練習問題

`flowers` に `violet` と `buttercup` をセットしなさい。また、このリストに更に二つの花を cons し、新しく出来たリストを `more-flowers` にセットしなさい。 `flowers` の CAR に魚の名前をセットしなさい。この時、`more-flowers` のリストの中身はどうなるか。

10 テキストのヤンク

GNU Emacs で 'kill' コマンドでバッファからテキストを切り取った場合、そのテキストは常に、ヤンク (yank) コマンドで取り戻すことが出来る。バッファから切り取られたテキストは kill リングに置かれており、ヤンクコマンドは、この kill リングの中から該当する内容を取り出してバッファに挿入する (これは、元のバッファとは限らない)。

単なる C-y (yank) コマンドは、kill リングの最初の要素をカレントバッファに挿入する。もし、C-y コマンドのすぐ後に続けて M-y とタイプすると、その最初の要素が二番目の要素と取り換えられる。続けて何回も M-y をタイプすると、その回数だけ最初の要素を入れ替えていき、一回りするとまた最初に戻ってそれを繰り返す。(そういうわけで kill リングは「リスト」ではなく「リング」と呼ばれるのである。しかし、実際にテキストを保存しているデータの型はリストに他ならない。リストをリングとして扱う方法についてのより詳しい話は Appendix B, "kill リングの扱い", p. 165, を参照。)

10.1 Kill リングについての概観

kill リングはテキスト形式の文字列のリストである。例えば次のような形をしている。

```
("some text" "a different piece of text" "yet more text")
```

もし私の kill リングが上のものだったとして、ここで私が C-y と押すと、'some text' という文字列が私が現在いるバッファのカーソルの位置に挿入される。

yank コマンドはまた、テキストを複写するのにも使われる。コピーされるテキストはバッファから切り取られるのではなく、一旦 kill リングに置かれ、その後ヤンクされることで挿入されることになる。

kill リングのテキストを取り出すには三つの関数が使われる。通常 C-yにバインドされている yank, 同じく通常 M-y にバインドされている yank-pop、そして、この二つの関数によって使われる rotate-yank-pointer である。

```
(insert (car kill-ring-yank-pointer))
```

yank や yank-pop の働きを理解するには、まず変数 kill-ring-yank-pointer と、関数 rotate-yank-pointerを理解する必要がある。

10.2 変数 kill-ring-yank-pointer

kill-ring-yank-pointer は、kill-ring と同じく変数である。一般に変数は、あるものの値にバインドされることで、それを指し示す働きをする。

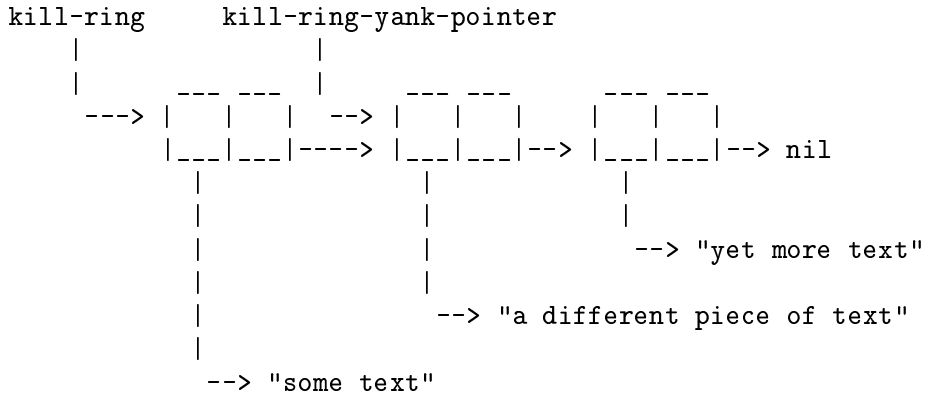
従って、もし kill リングの値が

```
("some text" "a different piece of text" "yet more text")
```

であり、また kill-ring-yank-poiner が二番目の位置を指していたとすると、kill-ring-yank-pointer の値は

```
("a different piece of text" "yet more text")
```

である。前章で説明したように (Chapter 9, "リストはどのように実装されているか", p. 77)、計算機は kill-ring と kill-ring-yank-pointer とに指されたテキストを各々別々に保持しているわけではない。"a different piece of text" と "yet more text" の二つの言葉は、二重に複製されているのではないのである。その代わり、二つの Lisp 変数は同じテキストの集りを指している。図に表わすと次の通りである。



変数 kill-ring と 変数 kill-ring-yank-pointer は両方ともポインタである。しかし、kill リングそのものは、実際に要素を組み合わせたものとして説明されることが多い。つまり、kill-ring はリストの計算機内での位置を指しているというよりは、リストそのものの意味で使われる。それに対し、kill-ring-yank-pointer の方は、リストへのポインタという意味で使われることが多い。

全く同じことについて、こういうふうに二つの言い方を使うと最初は混乱するように思えるかもしれない。しかしよく考えてみれば合理的である。kill リングは一般に、Emacs のバッファから最近切り取られたテキストのデータからなっていると捉えられている。一方 kill-ring-yank-pointer の方は、kill リングの中でその最初の要素 (つまり CAR) がこれから挿入するデータであるような部分、を指し示すために使われる。

rotate-yank-pointer 関数は、kill-ring-yank-pointer が指し示す要素を交換するものである。より詳しく言うと、ポインタが kill リングの最後の要素を越えて更に次の要素に移ろうとする際に、自動的にポインタを kill リングの最初の要素に移動してくれるのだ。このようにしてリストがあたかもリングであるように扱えるのである。rotate-yank-pointer 関数それ自身は単純であるが、その中身には多くの細かい説明が必要である。この関数及び、これよりずっと単純な yank, yank-pop 関数については次の所で説明する。Appendix B, “kill リングの扱い”, p. 165.

10.3 yank と nthcdr についての練習問題

- C-h v (describe-variable) を使って、あなたの kill リングの値を見なさい。幾つか kill リングの要素を増やしてから、もう一度 kill リングの中身を見なさい。M-y (yank-pop) を使って kill リングの中を動き回りなさい。あなたの kill リングの中にはどのくらい多くの要素がありますか？ kill-ring-max の値を見なさい。kill リングはもう一杯ですか？それともまだテキストのブロックを追加出来ますか？
- nthcdr と car を使って、各々リストの一番目、二番目、三番目、及び四番目の要素を返すような四つの S 式を作りなさい。

11 ループと再帰

Emacs Lisp は、一つないしは複数のＳ式を繰り返し評価させるのに、主として二つの方法を持っている。一つは while ループを利用するもの、もう一つは再帰 (*recursion*) を使うものである。

繰り返しは非常に役に立つものである。例えば四つの文の分だけ先に移動したい場合には、一つの文だけ移動するプログラムを書いて、そのプロセスを四回繰り返せば良い。計算機には飽きるとか疲れるとかいった感情はないので、このような繰り返しをいくらやらせたところで、人間のように過度の繰り返しをさせてしまってミスを犯しやすくなるとかいう弊害は無い。

11.1 while

特殊形式 while は、最初の引数を評価して返された値が真か偽かをテストする。ここまでは if の時と同じである。しかし、次に Lisp インタプリタがやることはちょっと違っている。

while 式においても、返された値が偽の場合は、Lisp インタプリタは残りのＳ式 (このＳ式の本体部分) をスキップし、それを評価したりはしない。しかし、返された値が真の場合は、Lisp インタプリタは本体部分を評価した後、もう一度 while の最初の引数が真か偽かをテストする。もし、返された値がまたもや真であったなら、Lisp インタプリタは再度本体部分を実行する。

while 式のテンプレートは次の通りである。

```
(while true-or-false-test
  body...)
```

while 式の真偽テストの部分が真を返している限り、本体部分は繰り返し評価される。このプロセスはループと呼ばれる。これは、Lisp インタプリタが何回も同じことを繰り返す様子が、飛行機の宙返りと似ているためである。真偽テストの結果が偽の場合は、Lisp インタプリタはそこで while 式の残りの部分を評価するのを止めて「ループを抜ける」。

当たり前のことだが、もし while 式の最初の引数が常に真を返すとすると、その後の本体部分は何回でも無限に評価され続ける。逆にもし真になり得ないとすると、本体部分の式は決して評価されることはない。while 式のループを書く工程とは、本体部分のＳ式の評価を繰り返したい回数だけ真偽テストが真の値を返し、そしてその後、偽を返すような仕掛を作ることである。

while 式を評価した場合に返される値は真偽テストが返す値である。従って面白いことに、エラーなく評価された while ループは、たとえ 100 回ループを繰り返した後でも一度もループしなかった場合でも、必ず nilを返す。正しく評価された while 式は決して真の値を返すことはないわけだ！ このことは、while 式は常に副作用を目的として評価されることを意味する。即ち、while ループの中の本体部分のＳ式を評価するために評価されるのである。これは理にかなっている。単にループを繰り返すことが目的なのではなく、ループの中のＳ式を繰り返し評価した時に生じる結果こそが望むものなのだから。

11.1.1 while ループとリスト

while ループをコントロールする一般的な方法は、リストが要素を持っているかどうかをテストすることである。もし要素が一つもなければ、それは空リストであり、結果として空リスト () が返される。これは、nil もしくは偽 (false) の同義語である。一方、要素を持つリストの場合は、それらの要素そのものが返される。Lisp は nil 以外のものは全て真と見做すので、この場合は while ループでは真が返されたことになる。

例えば、次のＳ式を評価すると、変数 empty-list の値を nilにセットすることが出来る。

```
(setq empty-list ())
```

この setq 式を評価した後、変数 empty-list をいつものようにカーソルをこのシンボルのすぐ後に持っていって C-x C-e とタイプすことで評価してみよう。エコー領域に nil が表示されるはずだ。

```
empty-list
```

一方、変数を要素を持つリストにセットした場合は、その変数を評価するとそのリストが表示される。これは次の二つのＳ式を評価することで確かめられる。

```
(setq animals '(giraffe gazelle lion tiger))

animals
```

ということで、リスト animals に要素があるかをテストに使用するような while loop を作るには、ループの最初の部分を次のように書けばよい。

```
(while animals
       ...
```

while が最初の引数をテストすると、変数 animals が評価される。これはリストを返す。このリストが要素を持っている限り、 whileはテストの結果が真だと判断する。しかし、リストが空になると、結果は偽だと判断される。

while ループが無限に続くことを避けるためには、最終的にリストが空になるような何らかの仕掛が必要である。よく使われるテクニックは、 while 式の本体部分の形式の一つでそのリストの値をそのリストの cdr で置き換えるようにしておくことである。 cdr 関数が評価されるごとにそのリストは短くなっていき、ついには空リストが残されるというわけである。そうなると while ループのテストは偽を返し、 whileの引数はもはや評価されず、ループが終了する。

例えば、動物のリストが変数 animals にバインドされていた時は、次のＳ式を使ってその変数を元のリストの CDR にセットすることが出来る。

```
(setq animals (cdr animals))
```

もし一つ前のＳ式を評価してあったなら、このＳ式を評価するとエコー領域に (gazelle lion tiger) が表示されるはずだ。もう一度このＳ式を評価すると、今度は (lion tiger) が表示される。更に評価すると、(tiger) が表示され、そこでまた評価すると、やっと空リストになって、 nil が表示される。

cdr 関数を繰り返し使うことで最終的に真偽テストで偽を返す while ループのテンプレートは次のようになる。

```
(while test-whether-list-is-empty
  body...
  set-list-to-cdr-of-list)
```

このテストと cdr の利用を組合せて、リストの各要素を各々一行ごとに表示するような関数を作ることが出来る。これを次に説明することにしよう。

11.1.2 リストを使ったループの例: print-elements-of-list

リストを使った while ループのことを理解するには print-elements-of-list 関数を見ると良い。

この関数は、出力を表示するために幾つかの行を必要とする。エコー領域は一行しかないので、今まで説明して来たように Info の中で評価するようなやり方ではその動作をうまく描写出来ない。代わりに必要なＳ式を '*scratch*' バッファにコピーして、そこでそれらを評価する必要がある。コピーするには、まず対象となるリージョンの最初を C-SPC (set-mark-command) でマークしてからカーソルをリージョンの最後に移動し、M-w (copy-region-as-kill) でそのリージョンをコピーする。そして '*scratch*' バッファで C-y (yank) することで、そのＳ式を取り出せばよい。

'*scratch*' バッファにそのＳ式をコピーしたら、今度はそのＳ式を評価するのだが、最後の (print-elements-of-list animal) は C-u C-x C-e とタイプして評価する必要がある。つまり eval-last-sexp に前置引数を与えるのである。すると、評価した結果はエコー領域ではなく *scratch* バッファに表示される。(単に C-x C-e とするだけだと、返された値はエコー領域に ^Jgiraffe^J^Jlion^J^Jtiger^Jnil のように表示されてしまう。ここで出て来る '^J' は改行を表している。従って *scratch* バッファでは各々の単語は一行ごとに表示されることになるわけである。このことは、今 Info バッファで次のＳ式を評価してみても確かめられる。)

```
(setq animals '(giraffe gazelle lion tiger))

(defun print-elements-of-list (list)
  "Print each element of LIST on a line of its own."
  (while list
    (print (car list))
    (setq list (cdr list))))

(print-elements-of-list animals)
```

これらの三つのＳ式を順に‘*scratch*’バッファで評価していくと、そのバッファの中で次のように表示されるはずだ。

```
giraffe

gazelle

lion

tiger
nil
```

リストの各要素が各々一行ごとに表示され (これは print の仕事である), そして最後にこの関数自体が返した値が表示される。この関数の中の最後のＳ式は while ループであり、 while ループは常に nil を返すので、nil がリストの最後の要素の後に表示される。

11.1.3 増加するカウンタを使ったループ

ループは止まるべき時に止まってくれないことには役に立たない。リストを使ったループの制御以外の一般的な方法としては、最初の引数として、正しい回数だけ繰り返すと偽を返すテストを書くという方法がある。これは、ループがカウンタ — つまりループを繰り返した回数を数えるＳ式 — を持つということである。

真偽テストとしては、(< count desired-number) といったＳ式が使える。これは count の値が期待する繰り返しの数 desired-number よりも小さければ、 t という真の値を返し、 desired-number 以上であれば、偽の値 nil を返す。カウンタを増加させるＳ式は (setq count (1+ count)) といったごく簡単な setq 式でよい。ここで 1+ は Emacs Lisp の組み込み関数で、引数に 1 を加えるものである。((1+ count) は (+ count 1) としても同じだが、この方が人間にとって読みやすいであろう。)

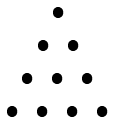
ということで、増加するカウンタを使った while ループのテンプレートは次のようになる。

```
set-count-to-initial-value
(while (< count desired-number)          ; 真偽テスト
  body...
  (setq count (1+ count)))                       ; インクリメンタ
```

この場合 count の初期値を定めなければならないことに注意しよう。通常は 1 にセットする。

増加カウンタの例

あなたは浜辺で遊んでおり、小石で三角形を作ろうと思ったとしよう。まず、一行目に 1 つの小石を置き、二行目に 2 つ置き、三行目に 3 つ置く..., という具合に続けていくのだ。図で描くと次のような感じになる。



(約 2500 年前、ピタゴラス一派はこのような問題を考えて数論の初歩を発展させていった。)

ここで、七行の三角形を作るにはいくつの小石が必要かを知りたいと思ったとする。

当然、ここであなたがしなければならないのは 1 から 7 までの数を加えることである。これを行うには二通りの方法がある。小さい数から始めて 1、 2、 3、 4、... という数列を加えて行くか、大きい方から始めて 7、 6、 5、 4、... という数列の和を取るかである。どちらの場合にも while ループを書く時の共通の方法を説明してくれるので、この下から登るのと、上から降りるのと両方の例を作ってみようと思う。最初の例として 1 に 2、 3、 4 を加えていくものから始めよう。

和を取る数のリストが短い場合は、一度に全部足してしまうのがもっとも簡単な方法である。しかしながら、もし前もって数のリストがどれくらいの長さになるか分らなかったり、或いは極めて長いリストの場合にも対処したい場合には、複雑なプロセスを一度にやるよりかは単純なプロセスを沢山繰り返して和を求めるような方法を考える必要がある。

例えば、小石を一度に全部加える代わりに、まず最初の行の小石の数 1 に二行目の小石の数 2 を加え、次にその合計に三行目の小石の数 3 を加え、今度はその和に四行目の数 4 を加え、という操作を続けるのである。

このプロセスの大切な特徴は、繰り返し行う操作が単純だということである。今の場合、各々のステップでやっていることは、二つの数を加えるということだけである。その時点での行の小石の数と、それまでの合計は既に分っている。この二数の和を取るプロセスは、最後の行の数をそれまでの合計に加えるまで何回でも繰り返される。もっと複雑なループでは、繰り返し行う操作はそれ程単純ではない。しかしそれでも全てを一度にやるよりはずっと簡単なのである。

関数定義の各部分

前節での分析は、我々の関数定義の骨組みを与えてくれる。まず、小石の全体の数を表わす変数が要る。これは total としていいだろう。これがこの関数の返す値である。

次に、この関数には三角形の全行数を表わす引数が要る。これは number-of-rows として良いだろう。

最後に、カウンタとして使う変数が必要である。これは counter としても構わないのだが、それよりも row-number とする方が良い。何故なら、このカウンタがやることは行を数えることであり、プログラムは出来るだけ分りやすく書くべきものだからである。

Lisp インタプリタがこの関数内のＳ式の評価を開始すると、まず totalの値が零にセットされる。これはまだ何も加えていないからである。次に、最初の行の小石の数が足される。そして二行目の数を加え、三行目の数を加え、というふうに続き、最後の行の小石を加えたところで終了する。

total も row-number も共にこの関数内部でしか使われない。従って、 let を使って局所変数として宣言し、初期値を与えればよい。明らかに total の初期値は 0 であり、また row-number の初期値は 1 である。ということで、 let 式は次のようになる。

```
(let ((total 0)
      (row-number 1))
  body...)
```

内部変数が宣言されて初期値にバインドされたなら、 while ループを開始する。テストの部分のＳ式は row-number が number-of-rows以下である場合その時のみ t を返すようなものでなければならない。(row-number と number-of-rows が一致する場合も含めないと最後の行の小石の数が加えられないことに注意。)

Lisp には <= という、最初の引数が二番目の引数以下の場合に真を返し、そうでない場合は偽を返す関数がある。これを使うと、 while 式のテスト部分のＳ式は次のように書ける。

```
(<= row-number number-of-rows)
```

全体の小石の数は、各々の行の小石の数をそれまでの合計に繰り返し加えていくことで計算される。ある行の小石の数はその行の番号と一致するので、全体の数は行の番号を加えていくことで求まることになる。(言うまでもないが、状況が複雑な場合には、ある行の小石の数とその行の番号とはもっと複雑な関係で結ばれている。そういう場合は行番号は他の適当なＳ式で置き換えることになる。)

```
(setq total (+ total row-number))
```

このＳ式では total の新しい値としてそれまでの合計に現在の行の小石の数を加えたものをセットしている。

total の値をセットしたら、次のループに移る場合のために条件を整えておかなければならない。それには、カウンタ用の変数 row-number の値に 1 を加えれば良い。 row-number の値が 1 増やされると、次のループの while 式の先頭の真偽テストで、この新しい値が number-of-rowの値以下であるかが判定される。もしそうであれば、変数 row-numberの新しい値が前回のループでの total の値に加えられる。

Emacs Lisp の組み込み関数 1+ は数に 1 を加えてくれる。従って、 row-nubmer 変数は次のＳ式で 1 増加させることが出来る。

```
(setq row-number (1+ row-number))
```

各部分の総合

これまでで関数定義の各々の部分を書いたので、ここでそれらを一つにまとめることにしよう。

まず while 式の中身は次の通り。

```
(while (<= row-number number-of-rows)    ; 真偽テスト
  (setq total (+ total row-number))
  (setq row-number (1+ row-number)))     ; インクリメンタ
```

これに let 式の変数リストを加えればほぼ関数定義の本体部分は完成するが、最後にほんの少しだけ加えるべき要素がある。

それは変数 total それ自身を while 式の後に書くことである。そうしないと関数全体が返す値は let 式の本体部分の最後のＳ式が返す値、即ち while 式が返す値になるので、常に nil が返されてしまうからである。

このことはぱっと見ただけでは気がつかないかもしれない。値を 1 増加させる式が最後だから良いように思えるかもしれないが、これはあくまで while式の一部なのである。シンボル while から始まるリストの最後の要素なのだ。更に while ループ全体は let 式の本体部分の中のリストなのである。

大ざっぱに書くと関数は次のような形をしている。

```
(defun name-of-function (argument-list)
  "documentation..."
  (let (varlist)
    (while (true-or-false-test)
      body-of-while... )
      ... )                          ; ここに最後の式が来る。
```

このように定義された関数が返す値は let 式が返す値になる。というのも let は defun 以外の他のどのリストにも含まれてはいないからである。だから、もし while が let 式の本体の最後のＳ式であれば、この関数は常に nil を返してしまうことになる。これは我々が望む結果ではない！ これを回避するには、単にシンボル total を let で始まるリストの最後に置けば良い。この式はこのリストの他のＳ式が評価された後に評価される。これによって、全体として正しい値が返されることになる。

let で始まるリスト全体を一行で表示してみると、理解しやすいだろう。こう書くと変数リスト *varlist* と while 式が let 式の二番目と三番目の要素であることがよく分る。そして最後の要素が total になるわけである。

```
(let (varlist) (while (true-or-false-test) body-of-while... ) total)
```

以上を一つにまとめると、 triangle 関数の定義は次のようになる。

```
(defun triangle (number-of-rows)    ; インクリメンタを使った
                                    ;  バージョン。
  "Add up the number of pebbles in a triangle.
The first row has one pebble, the second row two pebbles,
the third row three pebbles, and so on.
The argument is NUMBER-OF-ROWS."
  (let ((total 0)
        (row-number 1))
    (while (<= row-number number-of-rows)
      (setq total (+ total row-number))
      (setq row-number (1+ row-number)))
    total))
```

これを評価して triangle をインストールした後、実際に試してみよう。二つの例を挙げる。

```
(triangle 4)

(triangle 7)
```

最初の 4 つの数の和は 10 になり、最初の 7 つの数の和は 28 になる。

11.1.4 減少するカウンタを使ったループ

while ループのテスト部分のもう一つの一般的な書き方としては、カウンタが零を越えるかどうかを判定するというものがある。カウンタが零より大きい間はループを繰り返すが、零以下になると止まるというわけである。このように動作させるためにはカウンタは零以上の数から始めて繰り返しの部分が評価されるごとに小さくなるようにしなければならない。

テスト部分は（> counter 0）のようになる。これは counter の値が零よりも大きければ真として t を返し、零以下なら偽として nil を返すというものである。数を次第に小さくしていくには、（setq counter (1- counter)）という単純な setq 式を使う。ここで、 1- は引数を 1 減らす Emacs Lisp の組み込み関数である。

ということで、 while ループのテンプレートは次のようになる。

```
(while (> counter 0)                    ; 真偽テスト
  body...
  (setq counter (1- counter)))          ; デクリメンタ
```

減少するカウンタを使った例

減少するカウンタを使ったループを説明するために、 triangle 関数を、このようなカウンタを使って書き直してみることにする。

数え方は、この関数の前回のバージョンの逆である。今回は、例えば三行からなる三角形の小石の数を求めるために、まず三行目の 3 つの小石を加え、次にその前の二行目の 2 個の小石の数を加え、更にその合計に 1 行目の小石の数 1 を加えるという操作をすることになる。

同様にして、七行からなる三角形の小石の数を求めるには、七行目の小石の数 7 にその前の行の小石の数 6 を加え、次にその合計に その前の行に小石の数 5 を加え、とやっていくことになる。前回の例と同じく、各々の足し算ではそれまでの合計と現在の行の小石の数の二つの数を加えているだけである。このプロセスは、足す小石の数がなくなるまで繰り返される。

最初に加える小石の数も分る。最後の行の小石の数は全体の行の数に等しいからである。三角形が七行からなっていれば、最終行の小石の数は 7 である。同様に、次々と足していく小石の数も分る。それは、前回足した数から 1 を引いたものである。

関数定義の各部分

まず三つの変数が必要である。三角形の行の総数、各々の行の小石の数、そして、小石の総数だ。この最後の数が今回求めようとしている数である。これら三つの変数を各々 number-of-rows, number-of-pebbles-in-row, total と名付ることにする。

total も number-of-pebbles-in-row もこの関数の内部でしか使われないので、 let を用いて宣言される。 total の初期値は勿論零である。一方、 number-of-pebbles-on-row の初期値は三角形の行の数に等しくなるべきである。これは、もっとも長い行から小石の数を数えていくためである。

従って let 式の始めの部分は次のようになる。

```
(let ((total 0)
      (number-of-pebbles-in-row number-of-rows))
  body...)
```

小石全体の数は、各々の行の小石の数を順番に数えていくことで求められる。つまり、繰り返し次のＳ式を評価していけばよい。

```
(setq total (+ total number-of-pebbles-in-row))
```

number-of-pebbles-on-row は total に加えられた後、次のループに備えて一つだけ値を減らさなければならない。というのも次に加えられる行は一段上の行であり、現在の長さよりも一つ分短いからである。これは次のＳ式を評価することでなされる。

```
(setq number-of-pebbles-in-row
      (1- number-of-pebbles-in-row))
```

最後に、小石が無くなった時点で while loop を抜けなければならないが、そのための条件部は次のような簡単なものでよい。

```
(while (> number-of-pebbles-in-row 0)
```

各部分の総合

以上のＳ式をひとまとめにすると、関数定義は次のようになる。

```
;;; デクリメンタを使った最初のバージョン。
(defun triangle (number-of-rows)
  "Add up the number of pebbles in a triangle."
  (let ((total 0)
        (number-of-pebbles-in-row number-of-rows))
    (while (> number-of-pebbles-in-row 0)
      (setq total (+ total number-of-pebbles-in-row))
      (setq number-of-pebbles-in-row
            (1- number-of-pebbles-in-row)))
    total))
```

これはこれで、うまく動作する。

しかしながら、実は変数 number-of-pebbles-in-row は必要でない！

triangle 関数が評価されると、シンボル number-of-rows は初期値を与えられて、ある数にバインドされる。この数は関数の本体の中であたかも局所変数であるかのごとく変化させることが可能で、この関数の外でのこの変数の値には何の影響も与える心配はない。これは Lisp の大変便利な特徴であるが、このことから関数内の number-of-pebbles-in-row を全て変数 number-of-rows で置き換えても良いことが分る。

ということで、以下に、この関数の少し整理したバージョンを挙げる。

```
(defun triangle (number)                ; 二番目のバージョン。
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1- number)))
    total))
```

まとめてみよう。きちんと書かれた while ループは以下の三つの部分からなる。

1. ループを正しい回数繰り返した後、偽を返すテスト
2. 繰り返し評価された後、最後に望む値を返すようなＳ式
3. 正しい回数だけループを繰り返した後にテストが偽を返すよう、真偽テストに渡される値を変化させるＳ式

11.2 再帰

再帰関数とは、自分自身を評価するようなコードを含んでいるものである。関数が自分自身を評価した場合、それはまた自分自身を評価するコードに出くわす。結果としてその関数はまた自分自身を評価し ... となってこれがずっと続く。再帰関数は自分自身止める条件を与えられない限り、永久に自分自身を繰り返し呼び出し続けることになる。

再帰関数は、典型的には次のような三つの部分からなる条件分岐部を含んでいる。

1. この関数がもう一度呼び出されるかどうかを決定する真偽テスト。ここでは *do-again-test* と呼ぶ。
2. この関数の名前
3. 条件分岐部が正しい回数だけ繰り返しを行った後に偽を返すようにするためのＳ式。ここでは *next-step-expression* と呼ぶ。

再帰関数は、他の種類の関数に比べて最も簡単な形に書ける。実際、再帰関数を使い始めると、しばしば奇妙な程単純な形になってしまい、不可解な感じがすることが多いようである。再帰関数の定義を読むためには、ある種のコツが必要で、最初は難しく思えても、慣れると単純であることが分ってくる。初めて自転車に乗るときと同じである。

再帰関数のテンプレートは次のようになる。

```
(defun name-of-recursive-function (argument-list)
  "documentation..."
  body...
  (if do-again-test
    (name-of-recursive-function
          next-step-expression)))
```

再帰関数が評価されるごとに、ある引数が next-step-expression の値にバインドされ、そしてその値が do-again-test で使われる。next-step-expression は関数をもう繰り返す必要がなくなった場合に do-again-test が偽を返すように設計されている。

do-again-test は *stop condition* とも呼ばれることもある。これはテストが偽の場合に繰り返しが止まるからである。

11.2.1 List を使った再帰

先に挙げた、 while ループを使ったリストの要素を表示する関数の例は、再帰的に書くことも可能である。次に、変数 animals の値をあるリストにセットするＳ式も含めて、そのコードを書いてみることにしよう。

この例は '*scratch*' バッファにコピーして、各々のＳ式をそのバッファで評価してやらなければならない。そして (print-elements-recursively animals) を評価する際は C-u C-x C-e を使う必要がある。でないと、Lisp インタプリタは結果をエコー領域に一行分だけしか表示してくれない。

また、カーソルを print-elements-recursively 関数の最後の閉じ括弧の直後の、コメントの手前の位置に持っていって評価しないといけない。そうしないと、 Lisp インタプリタはコメントまで評価しようとしてしまう。

```
(setq animals '(giraffe gazelle lion tiger))

(defun print-elements-recursively (list)
  "Print each element of LIST on a line of its own.
Uses recursion."
  (print (car list))                  ; 本体
  (if list                            ; do-again-test
      (print-elements-recursively     ; 再帰呼び出し
       (cdr list))))                  ; next-step-expression

(print-elements-recursively animals)
```

print-elements-recursively 関数は最初にリストの一番目の引数、即ち、リストの CAR を表示する。そしてもしリストが空でなければ、この関数内で自分自身を呼び出す。ただし引数としては、全体のリストではなく二番目以降の要素からなるリスト、即ち、そのリストの CDR を渡す。

この時の評価では、この関数は引数として受け取ったリストの最初の要素 (これは元々のリストでは二番目の要素であたる) を表示する。そして if 式が評価され、それが真であれば、この関数は自分自身を今回受け取った引数の CDR (これは、元々のリストの CDR の CDR である) を引数として再度自分自身を呼び出す。

関数が自分自身を呼び出す度に、引数として渡されるリストは元々のリストに比べて短くなっていき、結果として最後には空リストとともに呼び出すことになる。 print 関数は空リストを nil として表示する。そして次に条件分岐の部分が list の値をテストする。 list は nil なので、 if 式は偽を返し、もはや then-part は実行しない。関数全体としては nil が返されるので、この関数を評価すると最後に二回 nil が現れる。

(print-elements-recursively animals) を '*scratch*' バッファで評価すると、次のような結果が表示されるはずだ。

```
giraffe

gazelle

lion

tiger

nil
nil
```

(最初の nil は空リストの値が表示されたものであり、二番目の nil は関数全体の値である。)

11.2.2 カウンタの代わりに再帰を使う

前節で説明した triangle 関数もまた再帰的に書ける。これは次のようになる。

```
(defun triangle-recursively (number)
  "Return the sum of the numbers 1 through NUMBER inclusive.
Uses recursion."
  (if (= number 1)                    ; do-again-test
      1                               ; then-part
    (+ number                         ; else-part
       (triangle-recursively          ; 再帰呼び出し
        (1- number)))))               ; next-step-expression

(triangle-recursively 7)
```

これを評価することでこの関数をインストールし、試しに（`triangle-recursively 7`）を評価してみよう。(カーソルを関数定義の直後の、コメントの手前の位置に持っていって評価することを忘れずに。)

この関数がどのように動作するかを確かめるため、この関数に 1、 2、 3、 4 等の様々な引数を与えた場合に何が起きるかを考えてみよう。

まず引数の値が 1 だとどうなるか？

この関数では説明文字列の後に `if` 式がくる。これは `number` の値が 1 かどうかテストするものである。もし 1 であれば、 Emacs は `if` 式の then-part を評価する。この場合は関数の値として 1 が返される。(一行からなる三角形の中には小石は 1 つしかない。)

では引数の値が 2 であればどうだろう。この場合は Emacs は `if` 式の else-part を評価する。

今の場合、 else-part は足し算と `triangle-recursively` の再帰呼び出し、そしてデクリメントからなっている。具体的には次の通り。

```
(+ number (triangle-recursively (1- number)))
```

Emacs がこのＳ式を評価する時は、まず最も内部のＳ式から評価していき、順に他の部分を評価していく。詳しく書くと次のようなステップを踏むことになる。

Step 1 最も内部のＳ式の評価。
今の場合、最も内部のＳ式は（`1- number`）なので、 Emacs は `number` を 2 から 1 に減らす。

Step 2 `triangle-recursively` 関数の評価。
この関数が自分自身の内部に含まれていることとは関係なく、 Emacs は Step 1 の結果をこの関数の引数として渡す。
今の場合、 Emacs は `triangle-recursively` を引数 1 とともに評価する。さっき見たように、この場合この関数は 1 を返す。

Step 3 `number` の値の評価。
ここでいう変数 `number` は `+` で始まるリストの二番目の要素。その値は 2 である。

Step 4 `+` 式の評価。
`+` 式は二つの引数を受け取る。一つ目は `number` を評価して返された値 (Step 3) であり、二つ目は `triangle-recursively` を評価して返された値 (Step 2) である。
足し算の結果は 2 と 1 の和であり、 3 が返される。これは正しい結果である。二行からなる三角形の中には小石は 3 個含まれる。

引数 3 の場合

`triangle-recursively` が引数 3 とともに呼び出されたとする。

Step 1 *do-again-test* の評価。
まずは `if` 式が評価される。これは do-again-test であり、偽が返される。従って、 `if` 式の else-part が評価される。(この例では、テストの結果が真の時ではなく偽の時に自分自身を再帰呼び出しすることに注意しよう。)

Step 2 *else-part* のもっとも内部のＳ式の評価。
else-part の最も内部のＳ式が評価され、3 が 2 にデクリメントされる。これが next-step-expression である。

Step 3 `triangle-recursively` 関数の評価。
数値 2 が `triangle-recursively` 関数に渡される。

前節で説明した通り、`triangle-recursively` は引数 2 とともに評価されると 3 を返すのであった。

Step 4 足し算の評価。
足し算の式では 3 がこの時の `number` の値 3 に加えられる。

全体として、この関数が返す値は 6 になる。

以上で `triangle-recursively` に引数 3 を与えるとどうなるかが分った。もはや引数が 4 の場合に何が起きるかは明らかであろう。次のような感じだ。

再帰呼び出しで

```
(triangle-recursively (1- 4))
```

が評価され、結果として

```
(triangle-recursively 3)
```

の値を返す。これは 6 であり、この値に三行目の足し算で 4 が加えられる。

全体としてこの関数が返す値は 10 になる。

`triangle-recursively` が評価されるごとに、より小さい引数とともに自分自身を評価し、その状況が、引数がもはや再帰呼び出しを起こさない程小さくなるまで続けられるというわけである。

11.2.3 `cond` を使った再帰の例

以前説明したバージョンの `triangle-recursively` は特殊形式 `if` を用いて書かれていた。これは `cond` と呼ばれる特殊形式を用いても書くことが出来る。特殊形式 `cond` の名前は `conditional` という単語の短縮形から来ている。

特殊形式 `cond` は Emacs Lisp では `if` ほど頻繁に使われているとは言えないが、ここで説明する価値がある程度には使われている。

`cond` 式のテンプレートは次の通りである。

```
(cond
 body...)
```

ここで *body* はリストの列である。

本体の中身をもっと詳しく書くと次のような感じになる。

```
(cond
 ((first-true-or-false-test  first-consequent)
  (second-true-or-false-test  second-consequent)
  (third-true-or-false-test  third-consequent)
  ...)
```

Lisp インタプリタが `cond` 式を評価する時は、まず `cond` の本体のＳ式の列の最初のＳ式の最初の要素 (CAR つまり真偽テストの部分) から評価する。

もし、真偽テストが `nil` を返したなら、その式の残りの部分 (これを結果部 (consequent) と呼ぼう) はスキップされて、次のＳ式の真偽テストが評価される。こうして、もしあるＳ式で真偽テストが `nil` 以外の値を返したなら、そのＳ式の結果部が評価される。結果部は一つでも複数でも構わない。複数の場合は各々の式が順に評

価されていき、最後の値が返される。もしそのＳ式が結果部を持たなければ、真偽テストの結果が返される。(訳註：そして、真偽テストが真であったＳ式以降は無視される。)

どのＳ式の真偽テストも偽を返した場合は `cond` 式は `nil` を返す。

`cond` を使って書くと、 `triangle` 関数は次のようになる。

```
(defun triangle-using-cond (number)
  (cond ((<= number 0) 0)
        ((= number 1) 1)
        ((> number 1)
         (+ number (triangle-using-cond (1- number))))))
```

この例では、 `cond` は number が 0 以下の場合は 0 を返し、 1 の場合は 1 を返し、 1 より大きい場合は `(+ number (triangle-using-cond (1- number)))` が評価される。

11.3 ループについての練習問題

- 各々の行の値が行番号の自乗であるような場合に、 `triangle` 関数と同様な関数を書きなさい。ただし、 `while` ループを使うこと。
- `triangle` 関数と同様だが、各々の行の値を足すのでは無く、掛けていくような関数を書きなさい。
- 上の二つの関数を再帰的な関数に書き直しなさい。また、 `cond` を使って書き直しなさい。
- Texinfo モードのために、パラグラフの中に含まれる全ての '`@dfn`' に対する索引の項目をそのパラグラフの最初に作成するような関数を書きなさい。(Texinfo ファイルでは、 '`@dfn`' が定義の印になっている。詳しくは "Indicating Definitions, Commands, etc." in *Texinfo, The GNU Documentation Format*, 参照。)

12 正規表現の検索

GNU Emacs の中では、正規表現の検索が徹底的に活用されている。例えば、forward-sentence とか forward-paragraph といった関数を調べてみれば、こういった検索についてよく理解出来るだろう。

正規表現の検索は Section "Regular Expression Search" in *The GNU Emacs Manual*, の中や Section "Regular Expressions" in *The GNU Emacs Lisp Reference Manual*, の中で説明されている。この章を書く際にも、私は読者が少なくともこれらをある程度は知っていることを想定している。大事な点は、正規表現を使うことで具体的な文字列そのものだけではなく、パターンをも検索出来るということである。例えば forward-sentence のコードは文 (sentence) の終わりを示すパターンを検索し、その場所にポイントを移動する。

実際に foward-sentence 関数のコードを見る前に、文の終わりを示すパターンがどんなものであるべきかを考えておいた方が良いだろう。このパターンについては次のセクションで議論することにする。その次に、正規表現の検索を行う関数である re-foward-search の説明をする。forward-sentence 関数の説明はその後である。この章の最後の節では、foward-paragraph 関数の説明をする。foward-paragraph は複雑な関数なので、幾つか新しい特徴を紹介することになる。

12.1 sentence-end の正規表現

シンボル sentence-end は文末 (訳註：ここでは勿論英語の文章を想定している。) を示すあるパターンにバインドされている。この正規表現はどうあるべきだろうか？

明らかに、文末には終止符か疑問符、もしくは感嘆符が来る。実際、この三つの文字の中の一つで終了する文節だけが文末と見做されるべきである。これはパターンの中に次の文字集合が含まれるということだ。

```
[.?!]
```

しかしながら、forward-sentence が単に終止符や疑問符、感嘆符に移動するというだけではまずい。というのも、これらの文字は文中にも使われることがあるからである。例えば終止符は略語の後にも使われる。従って、他の情報も必要になる。

慣習的に、普通文末には二つの空白を打つが、文中の終止符、疑問符、感嘆符のの後には一つの空白しか打たない。従って、終止符、疑問符、感嘆符に続いて二つの空白というのが文末の良い目印になるだろう。ただし、ファイルの中では二つの空白はタブや行末であっても良い。つまり、正規表現の中には、これら三つのどれかというものが含まれる。この部分は次のように書ける。

```
\\($\\| \\|  \\)
       ^   ^^
      TAB  SPC
```

ここで '$' は行末を表わす。それと、タブと二つの空白がこの表現の中に入っていることを指摘しておく。両方とも、この表現の中に実際の文字が挿入されている。

括弧や縦棒の前には二つのバックスラッシュ '\\' が必要になる。最初のバックスラッシュは、Emacs の中でその後に続くバックスラッシュを quote するためのものであり、二番目のバックスラッシュは、その後に続く括弧や縦棒が特殊文字であることを示すものである。

また、文の後には次のように一つ以上の復帰コードが続くこともある。

```
[
]*
```

タブや空白と同様、復帰コードも実際のコードを埋め込むことで正規表現の中に挿入される。末尾のアスタリスクは、RET が零回以上繰り返すことを示す。

ただ、文末が必ずしも終止符や疑問符、ないしは感嘆符に続いて空白で終っているとは限らない。閉引用符や何らかの閉括弧が空白の前に来るかもしれない。実際の所、このような記号が空白の前に二つ以上続くこともある。これらのために、次のような表現が必要になる。

```
[]\"')}]*
```

この表現の中で、最初の ‘]’ が最初に来ていることに注意しよう。(訳註：‘[’ と ‘]’ で狭んで定める文字集合の中に ‘]’ を含めるには、このように文字集合の最初に ‘]’ を記述する。) また、二番目の文字は ‘"’ である。前の ‘\’ は Emacs にこれが特殊文字ではなく文字列の一部だと伝えるためのものである。残りの三文字は、‘’’、‘)’、‘}’ そのものを表わす。これらの文字が零回以上現れるということになる。

これら全てを合せたものが、あるべき文末の正規表現を形成している。そして、実際に sentence-end を評価してみると、次のような値が返される。(訳註：Mule や Nemacs では、これに加えて日本語のための拡張がなされているので、更に複雑になっている。)

```
sentence-end
     ⇒ "[.?!][]\"')}]*\\($\\|     \\|  \\)[
]*"
```

12.2 関数 re-search-forward

The re-search-forward 関数は、search-forward 関数と非常によく似ている。(後者については、Section 8.1.3, “関数 search-forward”, p. 65, 参照。)

re-search-forward は正規表現を検索するためのものである。もし検索が成功すれば、ただちに目的とする文字の後にポイントを移動する。後方検索の場合は目的の文字の直後に移動する。検索成功時には re-search-forward は t を返す。(訳註：Emacs version 19 ではポイントの位置を返す。) (従って、ポイントの移動は「副作用」である。)

search-forward と同じく re-search-forward 関数も四つの引数を持つ。

1. 最初の引数は、検索する正規表現である。正規表現は引用符に囲まれた文字列でなければならない。
2. 二番目の引数は省略可能であり、関数が検索する範囲を制限するために用いる。これはバッファの中の位置として指定される。
3. 三番目の引数も省略可能で、検索に失敗した場合の挙動を決めるためのものである。もし引数が nil なら失敗時にはエラーが返され、メッセージが表示される。他の値の場合は失敗時には nil が返り、成功時にはtが返される。(訳註：Emacs version 19 ではポイントの位置が返される。)
4. 四番目の引数は繰り返しの回数である。負の引数を与えると、後方検索になる。

re-search-forward のテンプレートは次の通りである。

```
(re-search-forward "regular-expression"
                   limit-of-search
                   what-to-do-if-search-fails
                   repeat-count)
```

二番目から四番目までの引数は省略可能である。しかし、最後の二つの片方ないしは両方に値を渡したい場合は、それ以前の全ての引数を与えなければならない。そうしないと Lisp インタプリタはどの引数を何処へ渡すかを間違えてしまう。

forward-sentence 関数では、正規表現は変数 sentence-end の値である。つまり、次の通りである。

```
"[.?!][]\"')}]*\\($\\|  \\|  \\)[
]*"
```

検索の限界はパラグラフの終わりまでである (文がパラグラフを越えて続くことはないので)。検索に失敗した場合は nil が返される。また、繰り返しの回数は forward-sentence の引数として与えられる。

12.3 forward-sentence

カーソルを文の前方に移動するコマンドは、Emacs Lisp での正規表現検索の使い方をストレートに説明してくれる。この関数は、実際以上に長くて複雑そうに見えるが、それは、前方に検索するだけではなく後方にも検索

出来るようになっていたり、オプションとして複数の文を移動することが出来るようになっているためである。この関数は通常は M-e というキーにバインドされている。

以下が forward-sentence のコードである。

```
(defun forward-sentence (&optional arg)
  "Move forward to next sentence-end.  With argument, repeat.
With negative argument, move backward repeatedly to sentence-beginning.
Sentence ends are identified by the value of sentence-end
treated as a regular expression.  Also, every paragraph boundary
terminates sentences as well."
  (interactive "p")
  (or arg (setq arg 1))
  (while (< arg 0)
    (let ((par-beg
           (save-excursion (start-of-paragraph-text) (point))))
      (if (re-search-backward
           (concat sentence-end "[^ \t\n]") par-beg t)
          (goto-char (1- (match-end 0)))
        (goto-char par-beg)))
    (setq arg (1+ arg)))
  (while (> arg 0)
    (let ((par-end
           (save-excursion (end-of-paragraph-text) (point))))
      (if (re-search-forward sentence-end par-end t)
          (skip-chars-backward " \t\n")
        (goto-char par-end)))
    (setq arg (1- arg))))
```

ぱっと見ただけだと、この関数は長く感じてしまう。まずは骨組みを見て、それから肉の部分を見ていくのが賢明だろう。骨組みを見るには、桁が左にあるものから見て行けば良い。

```
(defun forward-sentence (&optional arg)
  "documentation..."
  (interactive "p")
  (or arg (setq arg 1))
  (while (< arg 0)
    body-of-while-loop
  (while (> arg 0)
    body-of-while-loop
```

こう書き直すとぐっと解りやすくなる。この関数定義は説明文字列と interactive 式、そして or 式と while ループからなっているのである。

では順に各々の部分を見ていくことにしよう。

まず、説明文字列が過不足なくかつ理解しやすく書かれていることに注意しよう。

この関数は interactive "p" 宣言を持っている。これは、(もし与えられたなら) 処理された前置引数が引数としてこの関数に渡されることを意味する。(これは数値である。) もしこの関数が引数を渡されなければ (引数は省略可能である) その場合、引数 arg は 1 にバインドされる。また、 forward-sentence が non-interactive に引数無しで呼ばれた場合には、 arg は nil にバインドされる。

前置引数を扱うのは or 式である。この式では、 arg の値がある値にバインドされている場合はそのままにしておき、もし nil にバインドされていた場合は 1 にセットしている。

while ループ

or 式の後には、二つの while ループが続く。最初の while ループには、前置引数が負の値ならば真を返すような真偽テストが含まれている。これは後方検索のためのものである。このループの本体は二番目の while ループの本体とそっくりであるが全く同じではない。取り敢えずこちらの方はとばして、二番目のループの方に集中することにしよう。

二番目の while ループはポイントを前方に移動するものである。骨組みは次の通りである。

```
(while (> arg 0)            ; 真偽テスト
  (let varlist
    (if (true-or-false-test)
        then-part
      else-part
  (setq arg (1- arg))))     ; while ループのデクリメンタ
```

この while ループはデクリメントタイプの物である (Section 11.1.4, “減少カウンタを使ったループ”, p. 88, 参照)。この中にはカウンタ (今の場合は変数 arg) が零よりも大きい間は真を返すような真偽テストが含まれている。そして、ループを繰り返すごとにカウンタの値を 1 減らすようなデクリメンタが含まれている。

もし forward-sentence に前置引数が与えられなかったなら、といってもこれが普通の使い方だが、その場合 while ループは一度だけ繰り返す。これは arg の値が 1 だからである。

while ループの本体は let 式からなる。これは局所変数を作成する。また、その本体として if 式を持っている。

while ループの本体は次のようである。

```
(let ((par-end
       (save-excursion (end-of-paragraph-text) (point))))
  (if (re-search-forward sentence-end par-end t)
      (skip-chars-backward " \t\n")
    (goto-char par-end)))
```

ここで let 式は局所変数 par-end を生成、バインドしている。この後見るように、この局所変数は正規表現検索に限界ないしは制限を与えるために用いられている。もしこの検索でパラグラフ内に適切な文末が見つからなければパラグラフの終端で検索をやめる。

が、その前にまずどうやって par-end がパラグラフの終端の値にバインドされるかを見てみよう。ここでは let 式を使って、次の S 式を Lisp インタプリタが評価した際に返される値に par-end の値をセットしている。

```
(save-excursion (end-of-paragraph-text) (point))
```

この式では (end-of-paragraph-text) によってポイントがパラグラフに移動し、(point) によってそのポイントの値が返される。そして、 save-excursion によって元の位置にポイントが戻されるというわけである。このようにして let は par-end に save-excursion式が返す値、つまりパラグラフの終端の位置をバインドする。(end-of-paragraph-text 関数は forward-paragraph を使っている。これについては後で簡単に触れる。)

Emacs は次に let 式本体の本体を評価する。これは次のような if 式である。

```
(if (re-search-forward sentence-end par-end t) ; if-part
    (skip-chars-backward " \t\n")              ; then-part
  (goto-char par-end)))                        ; else-part
```

if は最初の引数が真かどうかテストし、もし真なら then-part を評価し、そうでなければ else-part を評価する。今の場合 if の真偽テストは正規表現検索である。

forward-sentence のような関数の実際の動作は奇妙に感じられるかもしれない。しかし Lisp ではこのような操作が行われるのは、ごく一般的なことである。

正規表現の検索

re-search-forward 関数は文末、つまり正規表現 sentence-endで定義されたパターンを検索する。もしパターンが見つかったなら — 即ち文末が見つかったなら — その時は re-search-forward 関数は二つのことを行う。

1. re-search-forward 関数は副作用を実行する。即ち、ポイントを見つけた文末まで移動する。
2. re-search-forward 関数は真の値を返す。これは if によって返される値であり、検索が成功したことを意味する。

副作用としてのポイントの移動は if 関数が検索成功の結果として値を返すよりも前の時点に行われる。

if 関数が検索に成功した re-search-forward から呼び出されて真の値を返す際には if は then-part、即ち（skipo-chars-backward "\t\n"）の評価も行う。このＳ式はタブや改行などを含む全ての種類の空白文字を越えて、目に見える文字 (printed character) の所まで前に戻り、その文字の直後にポイントを置く。ポイントは既に文末のパターンの所まで移動しているので、この動作で文が目に見える文字で終わっている部分の直後に来ることになる。通常はピリオドだろう。

一方、もし re-search-forward 関数が文末パターンを見つけられなかった場合には、関数は偽を返す。この場合は if は三番目の引数を評価する。これは（goto-char par-end）である。これは paragraph の終わりにまでポイントを移動する関数である。

正規表現検索は極めて便利なものであり、 forward-sentence— その中では検索は if 式のテストになっている — で説明されたパターンは手軽に使えるものである。あなたもこのパターンを取り入れたコードを見たり書いたりするであろう。

12.4 forward-paragraph: 関数の金脈

forward-paragraph 関数はポイントをパラグラフの終わりまで移動する。これは通常 M-} にバインドされており、例えば let*、 match-beginning、 looking-at のようなそれ自身も重要であるような他の幾つかの関数を利用している。

forward-paragraph 関数の定義は forward-sentence 関数の定義に比べてかなり長い。これは各々の行が fill-prefix (行詰め接頭辞) で始まるようなパラグラフも相手にしなければならないためである。

Fill prefix は 各々の行の先頭に繰り返し現れる文字列からなる。例えば Lisp コードでは便宜上パラグラフの各々の行が ‘;;; ’ から始まる。またテキストモードでは四つの空白文字インデントされたパラグラフの fill prefix としてよく使われる。 (Fill prefix についてのより詳しい情報は Section “Fill Prefix” in *The GNU Emacs Manual*, を参照せよ。)

Fill prefix があるということは、 forward-paragraph 関数は、中の行が左端から始まっているようなパラグラフの終わりを見つけるだけではなく、そのバッファの全て、ないしは多くの行がある fill prefix で始まっているよう場合にもパラグラフの終わりを見つけなければならないということを意味する。

更に、時には fill prefix があっても無視した方が良い場合だってある。特に空行でパラグラフが区切られているような場合なんかだ。これにより更に複雑さが増える。

ここでは forward-paragraph 関数を全て書き出すのはやめて、その中の一部だけを見ることにする。準備なしに読もうとすると、ちょっと臆してしまうかもしれない。

この関数のアウトラインは次のようである。

```
(defun forward-paragraph (&optional arg)
  "documentation..."
  (interactive "p")
  (or arg (setq arg 1))
  (let*
      varlist
    (while (< arg 0)        ; 後方に進む場合のコード
      ...
      (setq arg (1+ arg)))
    (while (> arg 0)        ; 前方に進む場合のコード
      ...
      (setq arg (1- arg)))))
```

関数の最初の部分はいつもの通りである。引数のリストには一つ省略可能な引数があるだけである。次に説明文字列が続く。

インタラクティブ宣言の中の小文字の 'p' はもし前置引数があれば、それを処理してから関数に渡すことを意味する。これは数値であり、いくつ分のパラグラフを移動するかを表わす。次の行の or 式は関数に一つも引数が与えられなかった場合を扱うための物である。これはこの関数がインタラクティブではなく他のコードから呼び出された場合に起きる。これについては以前説明した。(Section 12.3, "forward-sentence", p. 96, 参照。) ここまでは、今まで慣れ親しんできた部分である。

let* 式

forward-paragraph の次の行は let* 式で始まる。これは以前に出てた式とは異なる。このシンボルは let* であって let ではない。

特殊形式 let* は基本的に let と同じなのだが、 Emacs が変数を順にセットしていくため、変数リストの中で後に出てくる変数がそれ以前に出てきた変数の値を参照することが出来る、という点のみが異なっている。

この関数の let* 式の中では Emacs は二つの変数 fill-prefixと paragraph-separate をバインドしているのだが、 pargraph-separate がバインドされる値は fill-prefix-regexpの値に依存しているのである。

各々を順に見ていこう。シンボル fill-prefix-regexp は次のリストを評価した値にセットされる。

```
(and fill-prefix
     (not (equal fill-prefix ""))
     (not paragraph-ignore-fill-prefix)
     (regexp-quote fill-prefix))
```

これは最初の要素が関数 and であるようなＳ式である。

関数 and は各々の引数をそのどれかが nil を返すまで評価していく。どれかが nil を返した場合は and は nil を返す。しかし、もしどの引数も nil を返さなければ、最後の引数を評価して返された値を返す。(この場合、その値は nil ではないので、 Lisp では真と見なされる。) 別の言い方をすれば、 and は引数全てが真である場合にのみ真を返すわけである。

今の場合なら、 fill-prefix-regexp は後に続く四つのＳ式を評価して全て真 (即ち、非 nil) が返された場合にのみ、非 nil の値を返す。そうでない場合は fill-prefix-regexp は nil にバインドされる。

fill-prefix
: この変数を評価すると、もしあれば fill prefix の値が返される。 fill prefix が無い場合は nil が返る。

(not (equal fill-prefix ""))
: このＳ式は fill prefix があった場合にそれが空文字列、つまり文字を一つも含まない文字列かどうかを判定する。空文字列は fill prefix としての役には立たない。

(not paragraph-ignore-fill-prefix)
: このＳ式は、もし変数 paragraph-ignore-fill-prefix が t 等の真の値にセットされている場合に nil を返す。

(regexp-quote fill-prefix)
: これは and 関数の最後の引数になる。もし and の全ての引数が真であれば、and の値としてはこのＳ式を評価して返された値が返されることになる。そしてそれが fill-prefix-regexp の値になる。

この and 式を評価して真が返された場合、fill-prefix-regexpは regexp-quote によって修正された fill-prefix の値にバインドされる。regexp-quote は、文字列を読み取り、それのみにマッチし、その他の文字列にはマッチしないような正規表現を返す。結局、fill prefix が存在する場合、fill-prefix-regexp はその fill prefix の値にちょうどマッチする値にセットされ、そうでなければ nil にセットされる。

let* 式の二番目の局所変数は paragraph-separate である。これは次のＳ式を評価して返された値にバインドされる。

```
(if fill-prefix-regexp
    (concat paragraph-separate
            "\\|^" fill-prefix-regexp "[ \t]*$")
  paragraph-separate)))
```

このＳ式を見れば何故 let ではなく let* 式を使ったかが分る。if 式の真偽テストは、変数 fill-prefix の値が nil かその他の値かということに依存しているのである。

もし fill-prefix-regexp が値を持たなかったなら、(訳註: つまり nil であれば) Emacs は if 式の else-part を評価し、paragraph-separate を現在の局所的な値にバインドする。(paragraph-separate はパラグラフの区切りにマッチする正規表現である。)

しかし、もし fill-prefix-regexp が値を持てば、(訳註: 真の値を持てば) Emacs は if 式の then-part を評価し、paragraph-separate の値を fill-prefix-regexp をパターンの一部として含むような正規表現にバインドする。

より詳しく言うと paragraph-separate は、元々の paragraph-separate の値を fill-prefix-regexp に空行を加えた表現を連結した値にセットされることになる。‘^’ は fill-prefix-regexp が行頭に来なければならないことを意味し、その後に空白が来ても良いことが、"[\t]*$" で定義されている。‘\\|’ は、この正規表現か元の paragraph-separate かどちらかがマッチしなければならないことを示すものである。

では let* 式の本体部分に入ろう。本体の最初の部分では、この関数に負の引数が与えらた場合、即ち後方に移動する場合を扱っている。この部分は省略することにする。

前方に移動する場合の while ループ

let* 式の本体の二番目の部分は前方に移動する場合を扱っている。これは arg が零よりも大きい間は繰り返すような while ループである。この関数を使う場合は大抵この引数は 1 である。従って while ループの本体はちょうど一度だけ実行され、それによりカーソルはパラグラフを一つだけ移動する。

この while ループは次のような形をしている。

```
(while (> arg 0)
  (beginning-of-line)

  ;; パラグラフとパラグラフの間に居る場合
  (while (prog1 (and (not (eobp))
                     (looking-at paragraph-separate))
           (forward-line 1)))
```

```
;; パラグラフ内で、 fill prefix がある場合
(if fill-prefix-regexp
    ;; fill prefix があるので、 paragraph-start は関係無し。
    (while (and (not (eobp))
                (not (looking-at paragraph-separate))
                (looking-at fill-prefix-regexp))
      (forward-line 1))
  ;; パラグラフ内で、 fill prefix がない場合
  (if (re-search-forward paragraph-start nil t)
      (goto-char (match-beginning 0))
    (goto-char (point-max))))

(setq arg (1- arg)))
```

まず直ちに分ることは、これは減少カウンタの while ループであり、デクリメンタとして (setq (1- arg)) というＳ式を使っているということである。このループの本体は次の三つのＳ式からなる。

```
;; パラグラフの間に居る場合
(beginning-of-line)
(while
    body-of-while)
;; パラグラフ内で、 fill prefix がある場合
(if  true-or-false-test
    then-part
;; パラグラフ内で、 fill prefix がない場合
  else-part
```

Emacs Lisp のインタプリタが while ループの本体を評価する場合、まず最初に (beginning-of-line) という式を評価して行頭に移動する。次に内部の while ループに入る。この while ループは、カーソルがパラグラフとパラグラフの間の空白にあった場合に外に移動させるためのものである。最後に、実際にパラグラフの終りにポイントを移動する if 式が来る。

パラグラフとパラグラフの間での動作

まずは内部の while ループを見ることにしよう。このループはポイントがパラグラフとパラグラフの間にあった場合を扱うためのものである。ここでは三つの新しい関数が登場する。 prog1、 eobp、そして looking-at である。

- prog1 関数は progn とよく似ているのだが、引数を順に評価していった後、最初の引数が返した値をこの関数全体の値として返す、という点が異なっている。 (progn 関数は最後の引数が返した値を返すのだった。) prog1 の二番目以降の引数は副作用だけのために評価されるわけである。
- eobp は 'End Of Buffer P' の略であり、ポイントがバッファの最後にある場合に真を返す関数である。
- looking-at は、ポイントの後に続くテキストが lookingo-at の引数として与えられた正規表現にマッチする場合に真を返す関数である。

while ループは次の通りである。

```
(while (prog1 (and (not (eobp))
                   (looking-at paragraph-separate))
              (forward-line 1)))
```

この while ループは本体部分を持たない！ 真偽テストは次のＳ式である。

```
(prog1 (and (not (eobp))
            (looking-at paragraph-separate))
       (forward-line 1)))
```

prog1 の最初の引数は and 式である。この中には、ポイントがバッファの最後にあるかどうか、そしてポイントの直後にパラグラフの区切りを示す正規表現にマッチするパターンがあるかどうかのテストが含まれている。

もしカーソルがバッファの最後ではなく、またカーソルの後にパラグラフの区切りを示す文字があれば、 and 式は真になる。 Lisp インタプリタは and 式を評価した後、 prog1 の二番目の引数である forward-line を評価する。これはポイントを一行だけ前に移動する。ただし、 prog1 が返す値は最初の引数の値である。従って、 whileループはポイントがバッファの最後にはなく、またパラグラフとパラグラフの間にある間だけ繰り返される。最終的にポイントがあるパラグラフに移動したなら、 and 式は偽を返す。いずれにしろ forward-line コマンドは実行されることに注意しよう。これはポイントがパラグラフとパラグラフの間から移動した場合、ポイントはパラグラフの二行目の先頭に置かれることになることを意味している。

パラグラフの内部での動作

外側の while ループでの次のＳ式は if 式である。 Lisp インタプリタは fill-prefix-regexp 変数が nil 以外の値を持っている場合には if 式の then-part を評価し、 nil の場合、つまり fill prefix が無い場合は else-part を評価する。

Fill prefix が無い場合

まず最初に fill prefix が無い場合のコードを見るのが単純で良いだろう。このコードは次のようにまた別の if 式からなっている。

```
(if (re-search-forward paragraph-start nil t)
    (goto-char (match-beginning 0))
  (goto-char (point-max)))
```

このＳ式は、大抵の人が forward-paragraph コマンドの主目的と考えるであろう仕事を実際に行う。つまり、まず次のパラグラフの始まりを前方に向かって検索するような正規表現の検索を行い、もし見つかれば、ポイントをその位置に移動する。別のパラグラフの始まりが見つからない場合は、現在のバッファでアクセス可能な最後の位置まで移動する。

この中で、これまでで慣れ親しんでいない部分は match-beginning の使い方だけである。この関数自体も初めて出てくるものだ。 match-beginning 関数は、最後に行った正規表現検索でマッチしたテキストの始まりの位置を表わす数値を返す関数である。

ここで match-beginning 関数が使われているのは前方検索の性質のためである。つまり、前方検索が成功した場合、通常の検索か正規表現の検索かどうかにかかわらず、ポイントを見つけたテキストの位置にまで移動してしまう。今の場合なら、検索に成功した場合はポイントは paragraph-start のパターンの終わりにまで移動するのだが、これは現在のパラグラフの終わりではなく、次のパラグラフの始まりである。

しかしながら、我々の目的はポイントを次のパラグラフの先頭ではなく現在のパラグラフの終わりにまで移動することである。この二つの位置は、パラグラフとパラグラフの間に幾つかの空行がある場合などでは、当然異なる。

引数 0 で呼ばれた場合、 match-beginning は最も最近、正規表現の検索に成功した位置を返す。今の場合、最も最近の正規表現の検索は paragraph-start を探すものなので、 match-beginning はそのパターンの開始位置を返す。(終了位置ではない。) この開始位置はパラグラフの最後である。

(ついでにいうと、引数として正の数が渡された場合、 match-beginning 関数は最後の正規表現の中の括弧でくくられた部分の表現の開始位置を返す。これは便利な機能である。)

Fill prefix がある場合

さっき議論した、内部の方の `if` 式は、それを含む fill prefix があるかどうかをテストする `if` 式の else-part になっていたのであった。fill prefix がある場合は、then-part の方が評価される。この部分は次の通りである。

```
(while (and (not (eobp))
            (not (looking-at paragraph-separate))
            (looking-at fill-prefix-regexp))
  (forward-line 1))
```

このＳ式がやることは、次の三つの条件が真である間だけ、ポイントを一行ずつ前方に移動することである。

1. ポイントはバッファの最後ではない。
2. ポイントに続くテキストはパラグラフの区切りではない。
3. ポイントに続くパターンは fill prefix を表わす正規表現である。

最後の条件はちょっと戸惑うかもしれないが、ポイントが `forward-paragraph` 関数によって既に行頭に移動していることを思い出せば納得出来るだろう。つまり、テキストに fill prefix があれば、`looking-at` 関数がそれを見ることになるのである。

まとめ

以上をまとめてみよう。前方に移動する場合の `forward-paragraph` 関数は次のことを行う。

- ポイントを行頭に移動する。
- パラグラフとパラグラフの間の行を飛ばす。
- fill prefix があるかどうかチェックする。もしあれば、
  — その行がパラグラフの区切りの行でない間だけ、一行ずつ前に進む。
- もし fill prefix がなければ、
  — 次のパラグラフの開始パターンを検索する。
  — そのパラグラフの開始パターンの始まりの位置に移動する。これは、その前のパラグラフの終了位置である。
  — そのパターンが見つからない場合は、そのバッファのアクセス可能な最大位置まで移動する。

復習のために、これまで議論してきたコードを見やすいよう整形して挙げておく。

```
(interactive "p")
(or arg (setq arg 1))
(let* (
       (fill-prefix-regexp
        (and fill-prefix (not (equal fill-prefix ""))
             (not paragraph-ignore-fill-prefix)
             (regexp-quote fill-prefix)))

       (paragraph-separate
        (if fill-prefix-regexp
            (concat paragraph-separate
                    "\\|^"
                    fill-prefix-regexp
                    "[ \t]*$")
          paragraph-separate)))

  backward-moving-code (omitted) ...
```

```
(while (> arg 0)                ; 前方に進む場合のコード
  (beginning-of-line)

  (while (prog1 (and (not (eobp))
                     (looking-at paragraph-separate))
           (forward-line 1)))

  (if fill-prefix-regexp
      (while (and (not (eobp))  ; then-part
                  (not (looking-at paragraph-separate))
                  (looking-at fill-prefix-regexp))
        (forward-line 1))
                                ; else-part: 内部の if
    (if (re-search-forward paragraph-start nil t)
        (goto-char (match-beginning 0))
      (goto-char (point-max))))

  (setq arg (1- arg)))))        ; デクリメンタ
```

forward-paragraph 関数の完全な定義は、上に挙げた前方に進むコードだけでなく、後方に戻るコードも含んでいる。

もし、この文書を GNU Emacs の中で読んでいるなら、この関数全体のコードを見たい場合には M-. (find-tag) とタイプし、プロンプトが出たら関数名をタイプすれば良い。 find-tag 関数がまず 'TAGS' テーブルの名前を聞いてきた場合は、あなたの 'emacs/src' ディレクトリの中の 'TAGS' ファイルの名前を答える。これは '/usr/local/lib/emacs/19.23/src/TAGS'. のようなパス名である。 (正確な 'emacs/src' ディレクトリのパス名はあなたの Emacs がどのようにインストールされているかに依る。もしパス名が分らなければ、 C-h i として Info に入り、 C-x C-f とタイプして emacs/info ディレクトリのパス名を知ることが出来る。 'TAGS' ファイルのパスは対応する 'emacs/src' パスであることが多い。 (訳註：無いことも多い。) もっとも Info ファイルが他の場所にあることも多いが。

或いは、 'TAGS' ファイルがないディレクトリに、自分自身の 'TAGS'ファイルを作成することも出来る。

12.5 自分自身の 'TAGS' ファイルの作成

簡単にソースの場所までジャンプ出来るように、あなた自身の 'TAGS' ファイルを作成することが出来る。例えば、もしあなたの '~/emacs' ディレクトリに沢山のファイルがあったとすると — 何を隠そう、私もこの場所に 137 個の '.el' ファイルがあり、その内 17 個をロードしているのだが — そのディレクトリに 'TAGS' ファイルを作ることで、 grep やその他の道具で関数名を検索するよりはずっと簡単に、特定の関数の位置にジャンプすることが出来るようになる。

TAGS ファイルは、 Emacs の配布に含まれる etags プログラムを使って作成出来る。普通、 etags は Emacs が構築された時に一緒にコンパイルされインストールされる。 (ただし、 etags は Emacs Lisp 関数や Emacs の一部分ではない。これは C のプログラムである。)

'TAGS' ファイルを作成するには、まずこのファイルを作成したいディレクトリに移動する。 Emacs の中だと、 M-x cd コマンドを使うか、そのディレクトリにあるファイルをビジットするか、或いは C-x d (dired) を使うことで移動することが出来る。そして

```
M-! etags *.el
```

とタイプしてやれば良い。 etags プログラムは普通のシェルで使える全てのワイルドカードを理解出来る。例えば、もし二つのディレクトリに対して一つの 'TAGS' ファイルを作りたい場合は、二番目のディレクトリを '../elisp/' だとして、次のようにタイプすればよい。

```
M-! etags  *.el ../elisp/*.el
```

また、 etags が受け付けるオプションのリストを見たい場合には

```
M-! etags --help
```

とタイプする。

etags プログラムは Emacs Lisp, Common Lisp, Scheme, C, Fortran, Pascal, LaTeX, そして大抵のアセンブラを扱うことが出来る。このプログラムには言語を特定するためのスィッチはない。その代わりにファイル名や中身からそのファイルの言語を認識するのである。

また 'etags' は、自分自身でコードを書いたり、既に書いた関数を後から参照したりするのにも大変便利である。新しい関数を書いたら、ときおり etags プログラムを走らせよう。そうすることでそれらの関数が 'TAGS' ファイルに付け加わる。

12.6 復習

ここでは最近導入した関数の簡単なまとめを載せておく。

while 本体部分のＳ式を、本体の最初の要素が真を返す間だけ繰り返し評価する。最後は nil を返す。(つまり、この式は副作用のためだけに評価される。)

例)

```
(let ((foo 2))
  (while (> foo 0)
    (insert (format "foo is %d.\n" foo))
    (setq foo (1- foo))))

 ⇒      foo is 2.
        foo is 1.
        nil
```

(insert 関数は引数をポイントに挿入する。また format 関数は引数を message 関数が整形するのと同様に整形して出来たストリングを返す。 \n は改行である。

re-search-forward
パターンを検索し、発見した場合はその直後にポイントを移動する。

search-forward と同様、四つの引数を取る。

1. 検索パターンを表わす正規表現。
2. 省略可能。検索の限界。
3. 省略可能。検索が失敗した場合にどうするか。 nil を返すかエラーメッセージを出すか。
4. 省略可能。検索を何回繰り返すか。負の数の婆は後方に検索する。

let* 幾つかの変数を特定の値に局所的にバインドし、残りの引数を評価する。値としては最後の引数の値を返す。局所変数をバインドする際、より先にバインドした変数があれば、その値を利用出来る。

例)

```
(let* ((foo 7)
       (bar (* 3 foo)))
  (message "'bar' is %d." bar))
    ⇒ 'bar' is 21.
```

match-beginning
最後の正規表現で見つかったテキストの始まりの位置を返す。

looking-at
ポイントに続くテキストが引数の正規表現にマッチした場合に真として t を返す。

`eobp` ポイントがそのバッファのアクセス可能な最大位置にある場合に真として `t` を返す。アクセス可能な範囲というのは、もしナローイングがかかっていない場合はバッファの最後の位置であり、ナローイングがかかっている場合は、その部分の最後である。

`prog1` 引数を順に評価していき、最初の引数の値を返す。

例)

```
(prog1 1 2 3 4)
     ⇒ 1
```

12.7 re-search-forward についての練習問題

- 二行以上続く空行にマッチする正規表現を検索する関数を書きなさい。
- 'the the' のように二度続く単語を検索する関数を書きなさい。二つの同じ部分からなる文字列にマッチする regexp (regular expression = 正規表現) の書き方については Section "Syntax of Regular Expression" in *The GNU Emacs Manual*, を参考にしなさい。 regexp の書き方はいく通りもある。その内の幾つかは残りのものよりも良い。私が利用している関数は、幾つかの正規表現と一緒に次の所に記されている。 Appendix A, "関数 the-the", p. 163.

13 カウント: 繰り返しと正規表現

繰り返しと正規表現の検索は、Emacs Lisp でコードを書く際によく使われる大変強力な道具である。この章では、正規表現の検索の利用の仕方を while ループと再帰を使った単語を数えるコマンドの作成を通して説明していくことにする。

標準的な Emacs の配布にはリージョン内の行数を数える関数が含まれている。しかし同じことを単語について行う関数はない。

文章の種類によっては単語の数を数えなければならないようなことがある。エッセイを書く場合は 800 単語以内に制限した方がよいであろうし、小説を書く場合など、一日 1000 単語は書くぞと決心することもあるだろう。個人的には Emacs に単語を数えるコマンドがないのは変だと思う。多分 Emacs を使う人は大抵コードとかドキュメントを書いていて、単語の数を数える必要などないのだろう。或いは Operating system 附属の単語数を数えるコマンドである wc しか使わなかったりするのかもしれない。はたまた編集者の都合に合わせて、文書の中の単語の数を文字数を 5 で割ったものとして数えている人もいるかもしれない。何はともあれ、以下で単語数を数えるコマンドを紹介することにする。

13.1 関数 count-words-region

単語数を数えるコマンドとしては、行、パラグラフ、或いはリージョンやバッファなどの中に含まれる単語の数を数えるものが考えられる。コマンドとしてはどんなことまで出来ればよいだろうか。コマンドをバッファ全体の単語を数えるように設計することも出来るが、Emacs の伝統からいって、もっと柔軟性を持たせた方がよいだろう。例えばバッファ全体ではなく、あるセクションの中の単語の数を教えるようなコマンドが欲しい場合も出てくるかもしれない。従って、リージョンの中の単語数を数えるように設計した方が合理的だろう。一度、このような count-words-region というコマンドを作ってしまえば、C-x h (mark-whole-buffer) を使ってバッファ全体をマークすることで、バッファ全体の単語を数えることだって出来る。

単語を数えることは、明らかに繰り返しを伴う動作だ。リージョンの始まりからスタートして最初の単語を数え、次に二番目に行き、三番目に行き、というふうにしてリージョンの最後に来るまで続けるわけである。これは単語を数えるという行為が、もともと再帰や while ループに適しているということを意味する。

まずは、while ループを使って、次に再帰を使って、単語を数えるコマンドを実際に作ってみることにする。これらのコマンドは勿論インタラクティブであるべきである。

これまでにも何度も見たように、インタラクティブな関数の定義のテンプレートは次のようになっている。

```
(defun name-of-function (argument-list)
  "documentation..."
  (interactive-expression...)
  body...)
```

このスロットを埋めればよい。

関数の名前はそれ自体で意味が通じ、かつ既にある関数 count-lines-region の名前と似たものであるべきである。これは名前を覚えやすくするためだ。count-words-region なんかが良いだろう。

この関数はリージョン内の単語の数を数える。従って引数リストには、リージョンの始まりと終わりの二つの位置各々にバインドされるシンボルが含まれていなければならない。これらの二つの位置は ‘beginning’ と ‘end’ と呼べばいいだろう。説明文字列の最初の行は一つの文であるべきである。というのは apropos のようなコマンドで表示される部分は一行目が全てであるからである。インタラクティブ式は ‘(interactive "r")’ という形になる。こうすることで、リージョンの始まりと終わりの位置をこの関数の引数リストに渡すことが出来る。ここまではルーティーンワークである。

関数の本体は、三つの仕事をするように書かれなければならない。一番目は while ループが単語を数える時の条件を設定すること、二番目は while ループを走らせること、そして三番目はユーザにメッセージを送ることである。

ユーザが `count-words-region` を呼び出した時点では、ポイントはリージョンの始まり、もしくは終わりにある。しかし、カウントは常にリージョンの始まりの位置から開始しなければならない。従って、そうでない場合はまずその位置までポイントを移動しておいて欲しい。（`goto-char beginning`）を実行することで、このことが保証される。勿論、関数が仕事を終えたらポイントの位置は元の位置に戻っていて欲しい。このために、本体部分は `save-excursion` 式で囲む必要がある。

本体の中心部分は、あるＳ式で次の単語に一つずつジャンプし、もう一つのＳ式でジャンプの回数を数える `while` ループからなる。 `while` ループの真偽テストはポイントがもう先へは進めなくなるまでは真を返し、リージョンの最後まできたら偽を返すようなものでなければならない。

ポイントを単語ごとに移動するＳ式としては（`forward-word 1`）を使ってもよい。しかし、正規表現の検索を使うなら、 Emacs が何を単語と見做すかを見る方が簡単である。

単語のパターンを検索する正規表現の検索では、ポイントはマッチしたパターンの最後の文字の位置に移動する。従って、続けて検索が成功している間はポイントは単語を一つずつ移動していくことになる。

現実問題として、正規表現の検索では、単語だけではなく単語間の空白や句読点をジャンプするようになっていて欲しい。実際、単語間の空白を越えて進まないような正規表現では二単語以上先には進めなくなってしまう。つまり、正規表現の中には単語だけでなく、その後に続く空白や句読点等も含まれていなければならないわけである。（単語はバッファの最後に来るかもしれないので、後に必ずしも空白がくるとは限らない。従って、空白等の部分の正規表現はオプションになる。）

というわけで、我々が求めている正規表現は、まず幾つかの単語構成文字が続き、オプションとして単語の構成要素以外の文字が一つ以上の続くようなパターンである。このような正規表現は次のように書ける。

```
\w+\W*
```

どの文字が単語構成文字であり、どの文字がそうでないかは、そのバッファのシンタックステーブルによる。（詳しくは Section 14.2, “単語やシンボルは何から構成されているか”, p. 120, 或いは Section “The Syntax Table” in *The GNU Emacs Manual*, や Section “Syntax Tables” in *The GNU Emacs Lisp Reference Manual*, を参照。）

検索のためのＳ式は次の通りである。

```
(re-search-forward "\\w+\\W*")
```

(‘`w`’ や ‘`W`’ の前にはバックスラッシュが二つずつ組になっていることに注意しよう。単独のバックスラッシュは Emacs インタプリタにとって特殊な意味を持っている。つまり、後に続く文字が普通とは異なる意味に解釈されるのである。例えば、‘`\n`’ という二つの文字は、バックスラッシュに続く‘`n`’ ではなく、改行コードを表わす。バックスラッシュが二つ続いて初めて通常の「特殊でない」バックスラッシュを表わすのである。)

(訳註：つまり、バックスラッシュが二つ要るのは、それが一度 Emacs インタプリタによって一つのバックスラッシュに解釈された後の結果が正規表現として解釈されるためである。ここは混乱しやすいので注意しよう。)

いくつ単語があったかを数えるためのカウンタも必要である。この変数は最初は 0 にセットされ、 while ループを一つ繰り返す度に一つずつ増えていく。この一つ増やすためのＳ式は単に

```
(setq count (1+ count))
```

だけでよい。さて、最後にユーザに対してリージョンにいくつ単語があったかを知らせたい。このようにユーザに情報を表示する目的には、 `message` 関数が用意されている。今の場合、メッセージは単語の数がいくつあった場合でもきちんとした文章になっていなければならない。例えば、 “there are 1 words in the region” というふうになっては困る。単語が一つなのに複数形が使われるのは文法的に正しくないからである。この問題は、条件分岐式を用いて単語数によってメッセージを変えるようにすれば回避出来る。可能性は三つある。一つはリージョン内には全く単語が無い場合、もう一つは一個だけある場合、そして、複数の単語がある場合である。このような場合には `cond` を使うとよいのであった。

以上を総合すると、次のような関数定義になる。

```
;;; 最初のバージョン: バグあり！
(defun count-words-region (beginning end)
  "Print number of words in the region.
Words are defined as at least one word-constituent
character followed by at least one character that
is not a word-constituent.  The buffer's syntax
table determines which characters these are."
  (interactive "r")
  (message "Counting words in region ... ")
;;; 1. 適切な状況の設定。
  (save-excursion
    (goto-char beginning)
    (let ((count 0))
;;; 2.  while loop を走らせる。
      (while (< (point) end)
        (re-search-forward "\\w+\\W*")
        (setq count (1+ count)))
;;; 3. ユーザにメッセージを送る。
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

が、以下に書かれている通り、この関数は全ての状況できちんと動作するわけではない。

13.1.1 count-words-region の空白文字に関するバグ

前節で説明した count-words-region コマンドには、二つのバグ、というか二つの現れ方をする一つのバグがある。まず、例えリージョンがある文章の途中にある空白文字しか含まない場合でも、 count-words-region は単語が一つあると言ってくる。また、バッファないしはアクセス可能な部分の最後にある空白文字しか含まない場合は、

```
Search failed: "\\w+\\W*"
```

というエラーメッセージが返ってしまう。この文を GNU Emacs の Info で読んでいるなら、このバグを実際に自分で確かめることが出来る。

まずはこの関数をいつも通り評価してインストールしよう。

お望みなら、次の式を評価することでこのコマンドのためのキーバインディングもインストール出来る。

```
(global-set-key "\C-c=" 'count-words-region)
```

最初のテストをするために、次の行の最初と最後にマークとポイントを設定して C-c = (或いは C-c = というキーバインディングをインストールしていなければ M-x counts-words-region) とタイプしよう。

```
        one   two  three
```

Emacs はリージョンには三つの単語があると正しく答えるはずだ。

次に、マークを行頭にセットして、ポイントをちょうど 'one'という単語の前に置いて同じテストをしてみよう。 C-c = (或いは M-x count-words-region) とタイプした場合、 Emacs はリージョンには一つも単語がないと答えるべきである。リージョンには行頭の空白文字だけしか含まれないからだ。しかし実際は Emacs はリージョンには一つの単語があると答えてくる！

三番目に、サンプルの行を '*scratch*' バッファの最後にコピーして行の終わりに幾つか空白文字をタイプしてから、マークを 'three' の直後に、ポイントを行末に設定する。 (即ち、行末をバッファの最後にする。) ここでさっきと同じく C-c = (或いは M-x count-words-region) とタイプしよう。空白文字しかないのだから、今回も Emacs は単語は一つもないと答えるべきである。しかし、 Emacs は 'Search failed' というエラーメッセージを表示する。

この二つのバグは共通する問題から生じたものだ。

最初のバグを考えてみよう。コマンドを実行すると、空白だけしかないはずの行頭部分に一つの単語が含まれていると言ってきたのであった。これは何故だろうか？ M-x count-words-region コマンドは、まずポイントをリージョンの最初に移動させる。 while テストは point の値が endの値よりも小さいかどうかテストする。これは正しいので、結果として最初の単語を見つける正規表現の検索が実行され、最初の単語が見つかる。ポイントは単語の後ろに移され、 count は 1 にセットされる。 while ループはもう一度実行されるが、今度はポイントの値が end の値よりも大きくなるので、ループはそこで終了する。そしてこの関数は、リージョン内の単語は一つだと言うメッセージを表示するというわけである。手短にいうと、正規表現の検索では、単語がリージョンの外にあるにも関わらず、単語を発見してしまうのである。

二番目のバグの場合、リージョン内にはバッファの最後の空白文字だけしかなかった。 Emacs は 'Search failed' と言う。これは何故か？ まず最初の while ループでは真偽テストは真になるので、検索が実行される。しかし、バッファの最後には単語は一つもないので、検索は失敗するというわけである。

どちらの現象も、検索がリージョンを越えて実行される所に問題があった。

解決法は、検索をリージョンに制限することである — これはかなり単純な動作に思えるが、実際に考えてみると、思った程簡単にはいかないことが分る。

今まで見てきたように、 re-search-forward 関数は最初の引数を検索パターンとする。しかし、この最初の必須の引数の他に、三つの省略可能な引き数を取ることが出来る。省略可能である二番目の引数は検索の限界である。オプションである三番目の引数は、もし t なら検索に失敗した場合にエラーを出さずに nil を返す。オプションである四番目の引数には繰り返しの回数を指定する。 (Emacs では、関数定義の説明文字列を見るには C-h f に続けて関数名をタイプして RET キーを押せばよい。)

count-words-region の定義内では、変数 end にリージョンの最後の値がこの関数の引数として渡され保持される。そこで、正規表現の検索の引数として、この end の値を与えてみよう。

```
(re-search-forward "\\w+\\W*" end)
```

しかし、もし count-words-region の値にこの変更だけしか加えずに、新しいバージョンの関数を、空白だけの場所でテストすると、'Search failed' というメッセージが返される。

これはどうしてかというと、リージョン内には単語を構成する文字が無いために、検索をそこに制限すると、検索が失敗してしまうのである。そのためにエラーメッセージが返される。しかし、この場合にエラーメッセージは受け取りたくない。期待されるのは "The region does NOT have any words." というメッセージである。

この問題を回避するには re-search-forward に三番目の引数として t を与えれば良い。すると関数はエラーメッセージの代わりに nil を返す。

しかし、この変更を加えてからもう一度テストしてみると、 "Counting words in region ... " というメッセージが表示された後いくら待っても何も表示されない。結局 C-g (keyboard-quit) とタイプするまでずっとこのメッセージを見続けることになる。

どうしてこうなるかだが、検索はリージョンに制限されているのでさっきと同じように失敗する。これはリージョンには単語を構成する文字が含まれないのだから期待通りである。結果として re-search-forward 式は nil を返す。そして他のことは何もしない。特に、ポイントを動したりもしない。これは検索に成功した場合の副

作用だからだ。 re-search-forward が nil を返した後、 while ループの次のＳ式が評価される。これはカウンタを一つ増やす。そして、次のループに移る。 re-search-forward式でポイントが移動しないために、ポイントの値は end の値よりも小さいままであり、真偽テストは真になる。そしてループが繰り返される。これがずっと続く ...

検索が失敗した場合は while ループの真偽テストが偽を返すようにするために、またもや count-words-region の定義に変更を加える必要がある。即ち、カウンタを一つ増やす前に行う真偽テストでは二つの条件が満たされなければならないのだ。一つはポイントがリージョン内にあること、もう一つは検索によって単語が見つかることである。

最初の条件も二番目の条件も同時に真にならなければいけないので、次のようにリージョンテストを行うＳ式と検索を行うＳ式の二つのＳ式を and 関数で連結して while ループの真偽テストに埋め込むことにしよう。

```
(and (< (point) end) (re-search-forward "\\w+\\W*" end t))
```

(and 関数についての詳細は、 Section 12.4, “forward-paragraph: 関数の金脈”, p. 99, を参照。)

re-search-forward 式は、もし検索が成功した場合は真を返し、副作用としてポイントを移動する。結果として、単語が見つかった場合にはポイントはリージョンを越えてしまうことがある。検索が失敗した場合やポイントがリージョンの終わりに来てしまった場合は、真偽テストは偽を返し、 while ループが終了して、メッセージが表示される。

これらの最終的な変更を加えると、 count-words-region は (少なくとも私が見た範囲では！) バグ無しに動作するようになる。コードは次の通りである。

```
;;; 最終バージョン: while
(defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")
  (message "Counting words in region ... ")

;;; 1. 適切な状況の設定
  (save-excursion
    (let ((count 0))
      (goto-char beginning)

;;; 2.  while loop を走らせる。
      (while (and (< (point) end)
                  (re-search-forward "\\w+\\W*" end t))
        (setq count (1+ count)))

;;; 3. ユーザにメッセージを送る
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

13.2 再帰を使った単語数のカウント

単語を数える関数は while ループではなく再帰を使っても書くことが出来る。実際にどのようになるかを見てみよう。

まず、 count-words-region 関数は三つの仕事をするということをはっきり認識しておこう。初めに、カウントを行う際の適当な条件を整え、次にリージョン内の単語を数え、最後に単語の数を知らせるメッセージをユーザに対し表示するというわけである。

これらの仕事を全て一つの再帰関数でやらせようとすると、全ての再帰呼び出しでメッセージを受け取ってしまうことになる。例えばリージョンに 13 の単語があった場合、 13 個のメッセージが並んでしまうのだ。これでは困る。そこで、これらの仕事を二つの関数に分けてやらせることにする。一つを再帰関数にして、それをもう一方で呼び出すのである。片方は条件を整え、メッセージを表示するものであり、もう片方は、単語を数えるものである。

まずはメッセージを表示する方の関数から始めよう。今回も関数の名前は count-words-region を使うことにする。

こちらはユーザが呼び出す方の関数であり、インタラクティブなものになる。実の所、これはこの関数の前回のバージョンにそっくりである。ただし、リージョン内の単語を数えるために、途中で recursive-count-words を呼び出している所だけが違う。

前回の関数を参考にすれば、簡単にこの関数のテンプレートを作ることが出来る。

```
;; 再帰バージョン; 正規表現検索を利用
(defun count-words-region (beginning end)
  "documentation..."
  (interactive-expression...)

;;; 1. 適切な状況の設定
  (explanatory message)
  (set-up functions...

;;; 2. 単語数のカウント
     recursive call

;;; 3. ユーザにメッセージを送る。
     message providing word count))
```

この定義は非常に素直に書かれているが、再帰呼び出しで返されたカウントを単語の数を表示するメッセージに渡す所が、幾分ややこしいかもしれない。これは、ちょっと考えれば、 let 関数を使えばよいと気付く。つまり、 let 式の変数リストの変数に、再帰関数から返されたリージョン内の単語数をバインドすればよいのである。その後、 cond 式を利用してその値をユーザに表示する。

let 式の中での変数のバインドというと、何か関数の二義的な仕事に思われることも多い。しかし、今の場合には、関数の主な仕事だと思われる単語を数えるということが、 let 式の中で行われることになる。

let を使うと、関数定義は次のようになる。

```
(defun count-words-region (beginning end)
  "Print number of words in the region."
  (interactive "r")
;;; 1. 適切な状況の設定
  (message "Counting words in region ... ")
  (save-excursion
    (goto-char beginning)
;;; 2. 単語数のカウント
    (let ((count (recursive-count-words end)))
```

```
;;; 3. ユーザに対しメッセージを送る
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
            ((= 1 count)
             (message
              "The region has 1 word."))
            (t
             (message
              "The region has %d words." count))))))
```

次に、再帰関数の方に移ろう。

再帰関数は少なくとも三つの部分を持たねばならない。‘do-again-test’、‘next-step-expression’、そして、再帰呼び出しである。

do-again-test は関数が自分自身をもう一度呼ぶかどうかを決定するものである。ここではリージョン内の単語を数え、その中の単語の分だけ前に移動するのであるから、 do-again-test ではポイントがまだリージョンの内部にあるかどうかをチェックすれば良い。つまり、ポイントの値を調べて、それがリージョンの終わりの値よりも小さいか大きいかを比べるわけである。ポイントの値を調べるには `point` 関数を使えばよい。リージョンの終わりの値は、勿論、この再帰関数の引数として渡す必要がある。

更に do-again-test では検索で単語が見つかったかどうかもテストしなければならない。無かった場合は再帰呼び出しをすべきではない。

next-step-expression は再帰関数が自分自身の呼び出しを止めるべき時に止めるように値を変化させるものである。より正確に言うと、 next-step-expression は適切な時に do-again-test が再帰呼び出しを繰り返すのを止めるように、値を変化させていく。今の場合なら、 next-step-expression はポイントを単語ごとに移動させていくＳ式になる。

再帰関数の三つ目の部分は、再帰呼び出しである。

また、どこかにこの関数の実際の「仕事」をする部分、つまり単語数を数える部分を書く必要がある。これが一番重要な部分だ！

が、ともかく、再帰関数のアウトラインは出来た。

```
(defun recursive-count-words (region-end)
  "documentation..."
    do-again-test
    next-step-expression
    recursive call)
```

このスロットを埋めていけばよい。まずは、一番単純な場合から始めよう。もしポイントがリージョンの終わりもしくはそれ以降の位置にあった場合は、リージョン内には単語はない。従って、関数は 0 を返すべきである。同様に検索が失敗した場合も単語はないので、 0 を返すべきである。

一方、もしポイントがリージョンの中にあって検索が成功した場合には、関数は自分自身を再度呼び出すべきである。

というわけで、 do-again-test は次のようになる。

```
(and (< (point) region-end)
     (re-search-forward "\\w+\\W*" region-end t))
```

検索をするＳ式も do-again-test の一部であることに注意しよう。この関数は検索が成功すれば `t` を返し、失敗すれば `nil` を返す。 (Section 13.1.1, “`count-words-region` の空白文字に関するバグ”, p. 111, を見れば、 `re-search-forward` がどのように動作するかが分る。)

do-again-test は `if` 式の真偽テストである。明らかに、もし真偽テストが真だった場合には、`if` 式はこの関数を再度呼び出さなければならない。が、もし偽だった場合には、else-part によって零が返されるべきである。何故なら、単語が見つからなかったということは、ポイントがリージョンの外だったり検索に失敗したということだからだ。

しかし、再帰呼び出しについて考える前に、next-step-expression について考える必要があるのだった。これは今の場合何か？ 興味深いことに、これは do-again-test の検索部分なのである。

do-again-test で `t` や `nil` を返すことに加えて、`re-search-forward` は検索成功時には副作用としてポイントを前方に移動する。これこそが、ポイントがリージョン内を移動しきった場合に、再帰関数が自分自身を呼び出すのを止めるようにポイントの値を変化させていく動作なのである。というわけで、`re-search-forward` 式が next-step-expression になっているのである。

従って `recursive-count-words` のアウトラインは次のようになる。

```
(if do-again-test-and-next-step-combined
    ;; then
    recursive-call-returning-count
  ;; else
  return-zero)
```

これにカウントを数えさせるような機構を組込むには、どうしたらよいだろうか。next-step-expression では一単語ごとに前方に進み、従って再帰呼び出しは一つ単語を進むごとに行われるわけだから、カウントの仕組は `recursive-count-words` が呼ばれるごとに数を一つ増やすようなものでなければならない。

幾つかの場合を考えてみよう。

- リージョン内に二つの単語があった場合、この関数は、最初の単語を数える時に返した値にリージョンの残りの単語数、今の場合なら 1、を足した数に 1 を加えた値を返さなければならない。
- リージョン内に一つだけ単語があった場合、この関数は、最初の単語を数える時に返した値にリージョンの残りの単語数、今の場合なら 0、を足した数に 1 を加えた値を返さなければならない。
- リージョン内に一つも単語がなければ、この関数は 0 を返すべきである。

以上のことから、`if` 式の else-part では、単語が一つもなかった場合のために 0 を返せばよいことが分る。これは、`if` 式の then-part では残りの単語の合計の値に 1 加えた値を返すべきだということを意味する。

実際のＳ式は次のようになる。ここで、`1+` は引数に 1 加える関数である。

```
(1+ (recursive-count-words region-end))
```

以上から `recursive-count-words` 関数全体は次のようになる。

```
(defun recursive-count-words (region-end)
  "documentation..."

;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))

;;; 2. then-part: 再帰呼び出し
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

実際にどのような動作をするかを見てみよう。

もし、リージョンに単語が一つもなければ、 if 式の else-part が評価され、結果としてこの関数は 0 を返す。

もし、リージョンに単語が一つあれば、ポイントの値は region-end の値よりも小さく、検索は成功する。この場合、 if 式の真偽テストは真を返し、 then-part が評価され、カウント式が評価される。この式が返す値が関数全体の返す値になるのだが、これは、再帰呼び出しで返された値に 1 加えた値である。

ところで、この時同時に next-step-expression によってポイントがリージョン内の最初の単語（といっても今はこの一個だけだが）を越えて移動する。つまり、（recursive-count-words region-end）が二回目に評価された時には、ポイントの位置はリージョンの終わり以降の位置にあることになる。従って、今度は recursive-count-words は 0 を返す。これにさっきの 1 が加えられ、結果として元の recursive-count-words は 0 足す 1、つまり、 1 を返すことになる。これは正しい数である。

ここまでくれば後はもうお分りだと思うが、リージョンの中に単語が二つある場合には、最初に呼び出された recursive-count-words は次に残りの単語を含むリージョンで呼び出された recursive-count-words の値に 1 加えた値、つまり、 1 + 1 = 2 という値を返す。これも正しい数である。

同様にして、もしリージョンの中に単語が三つある場合も、最初に呼び出された recursive-count-words は残りのリージョンで呼び出された recursive-count-words の値に 1 加えた値を返す。このような感じで、単語が沢山ある場合も正しい数を返してくれる。

説明文字列も完全につけると、二つの関数は次のようになる。

再帰関数:

```
(defun recursive-count-words (region-end)
  "Number of words between point and REGION-END."
;;; 1. do-again-test
  (if (and (< (point) region-end)
           (re-search-forward "\\w+\\W*" region-end t))
;;; 2. then-part: 再帰呼び出し
      (1+ (recursive-count-words region-end))

;;; 3. else-part
    0))
```

外側の関数 (wrapper):

```
;;; 再帰バージョン
(defun count-words-region (beginning end)
  "Print number of words in the region.

Words are defined as at least one word-constituent
character followed by at least one character that is
not a word-constituent.  The buffer's syntax table
determines which characters these are."
  (interactive "r")
  (message "Counting words in region ... ")
  (save-excursion
    (goto-char beginning)
    (let ((count (recursive-count-words end)))
      (cond ((zerop count)
             (message
              "The region does NOT have any words."))
```

```
((= 1 count)
 (message "The region has 1 word."))
(t
 (message
  "The region has %d words." count))))))
```

13.3 練習問題: 句読点のカウント

`while` ループを使って、リージョン内の句読点 — 終止符、カンマ、セミコロン、コロン、感嘆符、疑問符 — の数を数える関数を書きなさい。同じ関数を再帰関数を使って書きなさい。

14 defun 内の単語のカウント

次の計画の目標は、関数定義の中の単語の数を数えることである。当たり前のことだが、これは count-word-region の使い方をちょっと工夫すれば出来てしまう。 Chapter 13, “カウント: 繰り返しと正規表現”, p. 109, を参照のこと。例えば、ある一つの定義の中の単語数を数えたければ、 C-M-h (mark-defun) コマンドを使って定義部分をマークしてから count-word-region を呼び出せばよい。

しかしながら、ここではもっと大きなことをやってみたいと思う。 Emacs のソースの中の全ての定義の中の単語とシンボルの数を数えて、そこにどれだけの関数があり、各々がどのくらいの長さかをグラフにして出力するとか、どれだけの関数が 40 から 49 の単語とシンボルを持つか、 50 から 59 ではどうか、といったことを調べるのである。私はしばしば典型的な関数というのがどのくらいの長さかを知りたくなる。これは、そういったことを教えてくれるのである。

はっきり言って、このヒストグラムを書く計画は人をひるませる類のものである。しかし、これをいくつもの細かいステップに分けて、各々を一つずつ見ていくことにすれば、それほど恐れるほどのものではない。そこで、どんなステップに分けるべきかを書いてみることにする。

- まず最初に、一つの定義の中に単語がどれだけあるかを数える関数を書く。ここでは、単語と同じくシンボルをどう扱うかが問題になる。
- 二番目に各々の関数の中の単語数をリストにする関数を書く。この関数では count-words-in-defun を使うことが出来る。
- 三番目に、各ファイルの中の各関数の中の単語数をリストにする関数を書く。これをやろうとすると、必然的に、自動的に様々なファイルを見つけてそれらに移り、それに含まれる関数定義の中の単語数を数えるということをしなければならなくなる。
- 四番目に、三番目のステップで作成した数のリストをグラフとして出力するのに適した形に変換する関数を書く。
- 五番目に結果をグラフとして表示する関数を書く。

これはかなりの大計画である。しかし、各々のステップをゆっくりと進んでいけば、それ程困難なものではない。

14.1 何を数えればよいか？

関数定義の中の単語数を数えるにはどうしたらよいか、を最初に考え始めた時に、まず疑問に思うこと (或いは、考えるべきこと) は、我々は何を数えればよいかということである。 Lisp の関数定義に関して単語のことを話す場合、実際には大抵シンボルのことを言っている。例えば、次の multiply-by-seven 関数は、 defun、 multiply-by-seven、 number、 *、そして 7という 5 個のシンボルを含んでいる。これに加えて説明文字列の中に ‘Multiply’、 ‘NUMBER’、 ‘by’、そして ‘seven’ という単語が含まれている。 ‘number’ は繰り返して使われているので、関数定義の中には合計 10 個の単語とシンボルが含まれていることになる。

```
(defun multiply-by-seven (number)
  "Multiply NUMBER by seven."
  (* 7 number))
```

ところが、もし multiply-by-seven の定義を C-M-h (mark-defun) でマークして、そこで count-words-region を呼び出してみると、 10 個ではなく 11 個の単語があるという答えが帰ってくる。何かがおかしい！

実は、問題は二重になっている。 count-words-region は ‘*’ を単語とは数えないが、逆に、一つのシンボル multiply-by-seven を三つの単語だと数えてしまうのである。ハイフンは一つの単語内でのつながりを示すものではなく、単語間の間の空白と同じように扱われるのである。従って、‘multiply-by-seven’ は ‘multiply by seven’ と書かれているように扱われることになる。このような混乱の原因は、 count-words-regionの定義内で一つの単語ずつ移動する際に使っている正規表現にある。標準的な count-words-region のバージョンで使われている正規表現は

```
"\\w+\\W*"
```

である。この正規表現は一つ以上単語構成文字が続いた後に 0 個以上の非単語構成文字が続くというパターンである。「単語構成文字」によって何が意味されるかという問題は、構文 (syntax) の問題になる。これには一つのセクションを割当てて論じる価値がある。

14.2 単語やシンボルは何から構成されているか

Emacs では、各々の文字はある構文カテゴリ (*syntax categories*) に属するものとして扱われる。例えば正規表現 '\\w+' は一つ以上の「単語構成文字」(*word constituent*) が続くパターンを表している。単語構成文字というのは、ある一つの構文カテゴリーの要素のことである。他の構文カテゴリーの要素は、例えば終止符やカンマ等の句読点文字のクラス、スペースやタブ等の空白文字のクラスを含んでいる。(より詳しいことについては Section "The Syntax Table" in *The GNU Emacs Manual*, 及び Section "Syntax Tables" in *The GNU Emacs Lisp Reference Manual*, を参照のこと。)

構文テーブルとはどの文字がどのカテゴリーに属するかを定めるものである。普通、ハイフンは「単語構成文字」には分類されない。そうではなく「シンボルの名前ではあるが、単語ではないものの一部をなす文字のクラス」に分類される。これは `count-words-region` 関数がハイフンを単語間の空白文字と同じ扱いをすることを意味する。これが `count-words-region` が'`multiply-by-seven`' を三つの単語だと数える理由である。

Emacs に `multiply-by-seven` を一つの単語だと数えさせるには二つの方法がある。一つは構文テーブルを書換える方法、もう一つは正規表現を書き直す方法である。

Emacs が各々のモードに対して持つ構文テーブルを書換えることで、ハイフンを単語構成文字だと再定義することが出来る。この動作は我々の目的に殆んど合うのだが、ハイフンだけが単語の中には現れずシンボルの中には出てくる文字というわけではない。似たような文字は他にもある。

代わりに、 `count-words` の定義中の正規表現の方を書き直してシンボルを含むようにすることも出来る。こちらの方法の方がより簡明である。ただし、実際にやることは少々トリッキーだ。

最初の部分は十分に単純である。パターンとしては「少なくとも一つ以上続く、単語ないしはシンボルの構成要素」にマッチするもの、つまり、

```
\\(\\w\\|\\s_\\)+
```

になる。'\\(' は '\\w' と '\\s_' のいずれかを表わす正規表現のグループの開始を示す部分である。対象となる二つの部分は '\\|' で区切られている。'\\w' は任意の単語構成文字にマッチし、'\\s_'はシンボル名の一部になり得るが、単語構成文字ではないような任意の文字にマッチする。後に続く '+' は、このグループに属する文字、即ち単語かシンボルの構成文字が少くとも一回はマッチしなければならないことを意味する。

しかしながら、正規表現の二番目の部分はもっと難しい。欲しいものは、一番目の正規表現に続けて「単語の一部にもシンボルの一部にもならない文字が一つ以上続いてもかまわない」ことを表わす表現である。まず思い浮んだのは次のようなものである。

```
\\(\\W\\|\\S_\\)*"
```

上の大文字の 'W' と 'S' は単語、ないしはシンボルの構成文字ではないような文字にマッチする。しかし、この表現では、単語構成文字ではないか、もしくはシンボル構成文字ではない文字に一致してしまう。これではどんな文字にもマッチしてしまう。

次に私は、テストしているリージョン内の全ての単語やシンボルの後には空白文字 (スペース、タブ、もしくは改行) があることに気がついた。そこで、単語かシンボルの構成文字が一つ以上続くというパターンの後に、一つ以上の空白文字が続くというパターンを置いてみた。しかし、これも失敗した。通常は単語やシンボルは空白で終わるのだが、実際のコードでは、シンボルの後に括弧が来たり、単語の後に句読点が来たりすることだってある。というわけで、結局、単語かシンボルの構成文字の後に 0 個以上の空白文字以外の文字が続き、その後に 0 個以上の空白文字が来る、というパターンにすることにした。

次がその正規表現である。

```
"\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
```

14.3 関数 count-words-in-defun

以前見たように、 count-word-region 関数を書く方法は幾つかあった。が、今回 count-words-in-defun を書く際には、この内の一つの方法だけを採用することにする。

方法としては while ループを使う方法が理解しやすいだろうから、こちらを採用することにする。 count-words-in-defun は、より複雑なプログラムの一部になるので、インタラクティブである必要も、メッセージを出す必要もなく、ただカウントを返しさえすればよい。これらのことを考慮すると、定義は少し単純になる。

一方、 count-words-in-defun は関数定義を含むバッファの中で使われる。従って、現在ポイントが関数定義内にあるか判定し、もしそうであればその定義内の単語を数えるというふうにするのが合理的であろう。こうすると、ちょっとコードが複雑にはなるが、関数に引数を与える手間は省ける。

以上のことを考慮すると、テンプレートは次のようになる。

```
(defun count-words-in-defun ()
  "documentation..."
  (set up...
     (while loop...)
   return count)
```

いつも通り、やるべきことはこの中の空きスロットを埋めていくことである。

まずは set up から。

この関数は、関数定義が含まれるバッファの中で呼び出されることを想定されている。現在のポイントは関数定義の中にあるか、外にあるかどちらかである。 count-words-in-defun が動作してくれるためには、ポイントが関数定義の先頭に移動し、カウンタがゼロから始まり、ポイントが関数定義の最後に来たらループが終了するようになっていてくれなければならない。

beginning-of-defun 関数は後方に向かって行頭の '(' などの開き括弧を検索し、そこにポイントを移動するか、検索の限界まで移動する。実際には、 beginning-of-defun はポイントを現在ポイントが含まれている関数定義もしくはポイント以前のポイントに最も近い関数定義の開始位置、或いはバッファの先頭まで移動することになる。従って、 beginning-of-defun を使うことで望みの位置までポイントを移動することが出来る。

while ループでは、数えた単語やシンボルの数を保持しておくカウンタが必要である。 let 式によって、この目的のための変数を作り、その値をゼロに初期化することが出来る。

end-of-defun 関数は beginning-of-defun と同じような働きをするのだが、ポイントを関数定義の終了位置に移動する点だけが異なっている。 end-of-defun は関数定義の終了位置を決定するＳ式の一部として使うことが出来る。

ということで、 count-words-in-defun の set up 部分はあっさり書けてしまう。まずは、関数定義の最初にポイントを移動し、次にカウンタのための局所変数を用意し、最後に while ループが止まるべき所で止まれるように関数定義の終了位置を記録しておくのである。

コードは次の通りである。

```
(beginning-of-defun)
(let ((count 0)
      (end (save-excursion (end-of-defun) (point))))
```

このコードは単純である。ちょっとややこしいのは end に関するところだろう。これには、関数定義の終了位置がバインドされる。その際、 save-excursion 式の中で一時的に end-of-defun で関数定義の終了位置に移動した後にポイントの位置を返すという方法を用いている。

さて、 count-words-in-defun の set up に続く二番目の部分は while ループである。

このループでは、ポイントを単語やシンボルごとに移動するＳ式、及びジャンプの回数を記録するが必要である。また、 while ループの真偽テストでは、ポイントがまだジャンプすべきなら真を返し、定義の終了位置まで到達したなら偽を返すようなものであるべきである。目的のための正規表現は既に再定義してしまっているので、(Section 14.2, “単語やシンボルは何から構成されているか”, p. 120, 参照) ループは簡単に書ける。

```
(while (and (< (point) end)
            (re-search-forward
             "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*" end t)
  (setq count (1+ count)))
```

関数定義の三番目の部分は単語やシンボルの数を返す部分である。この部分は let 式の本体部分の最後の部分だが、極めて単純に局所変数 count を書いておくだけでよい。これを評価すると数が返るわけである。

以上をまとめると、 count-words-in-defun は次のようになる。

```
(defun count-words-in-defun ()
  "Return the number of words and symbols in a defun."
  (beginning-of-defun)
  (let ((count 0)
        (end (save-excursion (end-of-defun) (point))))
    (while
        (and (< (point) end)
             (re-search-forward
              "\\(\\w\\|\\s_\\)+[^ \t\n]*[ \t\n]*"
              end t))
      (setq count (1+ count)))
    count))
```

これをテストするにはどうしたらよいだろうか。この関数はインタラクティブではないが、ちょっとS式をかぶせることで簡単に interactive にすることが出来る。これには count-words-region の再帰関数版とほぼ同じコードが使える。

```
;;; インタラクティブバージョン
(defun count-words-defun ()
  "Number of words and symbols in a function definition."
  (interactive)
  (message
   "Counting words and symbols in function definition ... ")
  (let ((count (count-words-in-defun)))
    (cond
     ((zerop count)
      (message
       "The definition does NOT have any words or symbols."))
     ((= 1 count)
      (message
       "The definition has 1 word or symbol."))
     (t
      (message
       "The definition has %d words or symbols." count)))))
```

便宜上 C-c = というキーバインディングをもう一度使うことにしよう。

```
(global-set-key "\C-c=" 'count-words-defun)
```

以上で、 count-words-defun をテストするための準備が整った。まずは、 cout-words-in-defun 及び count-words-defun を両方ともインストールして、キーバインディングの設定もしてしまおう。そして、カーソルを次の定義の中に移動して実験してみる。

(defun multiply-by-seven (number)
  "Multiply NUMBER by seven."
  (* 7 number))
    ⇒ The definition has 10 words or symbols.

成功だ！ この定義の中には確かに 10 個の単語とシンボルがある。

次の問題は、一つのファイルの中の幾つかの定義にある単語とシンボルの数を数えることである。

14.4 一つのファイルにある複数の defun を数える

‘simple.el’ のようなファイルの中には 80 以上の関数定義が含まれていたりする。我々の最終的な目標は沢山のファイルについての統計を取ることであるが、その最初のステップとして、まずは一つのファイルについての統計を取ることを目標にしよう。

情報は数の列の形で与えられ、各々の数は関数定義の長さになる。これらの数はリストの中に保持しておくことが出来る。

一つのファイルについての情報は最終的には多くのファイルについての情報の形に統合されることになる。従って、ここで作成する一つのファイル内の関数定義の長さを数える関数は単に「長さ」のリストを返すだけでよく、特にメッセージとかを表示する必要はない。

単語を数えるコマンドには、単語ごとにポイントを前方に進めるＳ式とジャンプの回数を数えるＳ式が含まれていた。定義の長さを測る関数も、同じように設計することが出来る。この場合は定義ごとにポイントを進めるＳ式と長さのリストを作成するようなＳ式が含まれることになる。

問題をこのように言い替えてしまえば、関数定義を書くのは簡単なことである。明らかに、カウントはファイルの先頭から始めなければならない。従って、最初のコマンドは (goto-char (point-min)) である。次に while ループに入る。ここでは、ループの真偽テストは次の関数定義を探す正規表現に取れる — 検索が成功している間はポイントを進め、本体を評価するわけである。本体内には長さのリストを作成するＳ式が必要である。これには cons というリストを構成するコマンドが使える。やるべきことの殆どは、以上で終わっている。

部分的にコードを書くと次のようになる。

```
(goto-char (point-min))
(while (re-search-forward "^(defun" nil t)
  (setq lengths-list
        (cons (count-words-in-defun) lengths-list)))
```

この他にやることは、関数定義を含むファイルを見つけることである。

今までの例では、この Info file を使うか、‘*scratch*’ バッファなどの他のバッファに一旦戻って、また帰ってくるということしかしていなかった。

ファイルを見つける (find) ことは、この文書では初めて出てくるプロセスである。

14.5 ファイルを見つける

まずは、 find-file のソースを見てみよう。(関数のソースは find-tag を使うと見つけることが出来る。)

```
(defun find-file (filename)
  "Edit file FILENAME.
Switch to a buffer visiting file FILENAME,
creating one if none already exists."
  (interactive "FFind file: ")
  (switch-to-buffer (find-file-noselect filename)))
```

定義には、簡潔な説明文字列が付いており、インタラクティブ式では対話的に用いた時のプロンプトが指定されている。で、定義の本体を見ると、 find-file-noselect 及び switch-to-buffer という二つの関数が使われている。

C-h f (describe-function コマンド) で表示される説明によると、 find-file-noselect は指定されたファイルをバッファに読み込み、そのバッファを返す。しかしながらバッファは選択されない、つまり Emacs は注意をそのバッファには向けない。 (或いは find-file-noselect を名前のついたバッファに対し使った場合は、あなたの注意も引かない。) この仕事は switch-to-buffer がやってくれる。この関数は、 Emacs が注目するバッファを指定するものである。更にこの関数は、ウィンドウに表示されているバッファを新しいバッファに切り替える。バッファの切り替えについては、また別の場所で議論することにしよう。 (Section 2.3, “バッファ間の移動”, p. 17, 参照。)

今やろうとしているヒストグラム計画では、定義の長さを調べる際にいちいち一つ一つのファイルをスクリーンに表示する必要はない。というわけで switch-to-buffer ではなく set-buffer を使うことにしよう。これも Emacs が注目するバッファを切り替えるのだが、スクリーンに表示するバッファはそのままである。従って、我々の目的のためには find-fileは使えず、そのためのコードを書くことになる。

といっても、やることは簡単だ。単に find-file-noselect と set-buffer を使えばよいのである。

14.6 lengths-list-file についての詳細

length-list-file 関数の核心部分は、 defun から defun へ移動していく関数を含む while ループと、各々の defun の中に含まれる単語やシンボルの数を数える関数である。そしてその周辺に、例えば、ファイルを見つけたり、ポイントが必ずファイルの先頭部分からスタートするようにしたりする、といった他の様々な仕事をする関数が来る。結局、定義は次のようになる。

```
(defun lengths-list-file (filename)
  "Return list of definitions' lengths within FILE.
The returned list is a list of numbers.
Each number is the number of words or
symbols in one function definition."
  (message "Working on `%s' ... " filename)
  (save-excursion
    (let ((buffer (find-file-noselect filename))
          (lengths-list))
      (set-buffer buffer)
      (setq buffer-read-only t)
      (widen)
      (goto-char (point-min))
      (while (re-search-forward "^(defun" nil t)
        (setq lengths-list
              (cons (count-words-in-defun) lengths-list)))
      (kill-buffer buffer)
      lengths-list)))
```

この関数は一つの引数を取る。これは作業対象となるファイルの名前である。説明文字列は四行あるが、インタラクティブ宣言はされていない。本体の一行目では、使った人が計算機が壊れたのではないかと心配しないように、最初に何をやっているかを表示するようにしている。

次の行で save-excursion が使われているので、 Emacs は仕事が終わった後、ちゃんと元のバッファに注意を戻してくれる。こうすると、この関数を他の関数の中に埋め込んでいる場合などにも、ポイントを元の位置に戻してくれるので便利である。

let 式の変数リストの所で、 Emacs はファイルを見つけて局所変数 buffer をそのファイルを中身とするバッファにバインドする。同時に Emacs は lengths-list を局所変数として生成する。

次に Emacs は注意をそのバッファに向ける。

次の行では Emacs はバッファを読み出し専用にしている。理論上は、この行は不要である。関数定義内の単語やシンボルの数を数える関数の中で、バッファを書換えたりするようなものはないし、例えそのバッファが変更されたとしても保存されたりはしない。こういう警戒をするのは、これらの関数が Emacs のソース上で作業するために、万が一にでもファイルを修正してしまったりすると非常に不都合であるという理由のためである。言うまでもないが、私自身は実験が失敗して私の Emacs のソースファイルが修正されるなんていう事態に会わない限り、この行が必要だと思うことはないだろう。

次に、バッファがナローイングされている場合には、それを広げるということをやっている。これは普通は必要ない —Emacs はそのファイルに対応するバッファが無い場合は新規にバッファを開くからだ。しかし、既にある場合には Emacs はそのバッファを返す。この場合、もしそのバッファがナローイングされていたなら、それを解除する必要がある。真にユーザーフレンドリーな関数にしたい場合には、ナローイングやポイントの位置なんかを保存しておくべきだろうが、ここでは何もしないことにする。

`(goto-char (point-min))` 式でポイントをバッファの先頭に移動する。

そして `while` ループが来る。ここで、この関数の仕事が実行されることになる。このループでは、 Emacs は各々の定義の長さを調べ、その長さのリストを作っていく。

あるバッファでの作業が終わると Emacs はそのバッファを kill する。これは、 Emacs 内部でのスペースの節約のためである。私が使っている Emacs 19 のバージョンには 300 以上ものソースファイルが含まれており、これらに `lengths-list-file` が適用される。もし Emacs がこれら全てを読み込んで一つも kill しなかったら、私の計算機は仮想記憶を使い切ってしまうだろう。

終りまで来ると、 `let` 式の中の最後のＳ式である `lengths-list`という変数が評価される。この値が関数全体の値となる。

この関数をいつも通りインストールして試してみることが出来る。インストールが終わったら、カーソルを次のＳ式の直後に持っていってから `C-x C-e` (`eval-last-sexp`) とタイプして、評価してみよう。

```
(lengths-list-file "../lisp/debug.el")
```

(多分、ファイルのパス名を変更する必要があるだろう。上に挙げたものは、この Info ファイルのあるディレクトリと Emacs のソースがあるディレクトリが `/usr/local/emacs/info` と `/usr/local/emacs/lisp` のように隣同士にある場合だけである。変更する場合は、このＳ式を一旦 '`*scratch*`' バッファにコピーしてから、それを修正して評価する。)

私が使っているバージョンの Emacs では、'`debug.el`' の list の長さを調べるのに 7 秒かかり、結果は次のようになった。

```
(75 41 80 62 20 45 44 68 45 12 34 235)
```

ファイルの中の最後の定義の長さは、リストの最初に現れることに注意しよう。

14.7 異なるファイルの中の定義を数える

前節では、各ファイルの中に含まれる各関数の長さのリストを返すような関数を作成したのだった。今度は、ファイルのリストが与えられた時に、それらのファイルの中の関数の長さのマスターリストを返すような関数を定義してみたい。

リストの中の各ファイルに対する作業は繰り返しの動作なので、 `while`ループや再帰を使って行うことが出来る。

`while` ループを使った方法はルーティーンワークである。関数に渡す引き数はファイルのリストになる。以前見たように (Section 11.1.1, "ループの例", p. 83 参照)、このリストが要素を含んでいる時のみループの本体を実行し、要素が無くなったら抜けるようにすることが出来るのであった。これがうまく動作するためには、本体部分で、本体が一回評価されるごとにこのリストを短くしていき、結果として最後にはリストが空になるように、Ｓ式を書いておく必要がある。このためには、本体が評価されるごとに、リストにそのリストの CDR の値をセットするという技法を用いるのが普通である。

テンプレートは次のようになる。

```
(while test-whether-list-is-empty
  body...
  set-list-to-cdr-of-list)
```

さて、while ループは常に (真偽テストの結果として) nil を返し、本体内のＳ式の値を返したりすることはないのだった。(従って、ループの本体のＳ式は副作用として評価される。) しかしながら、長さのリストをセットするＳ式は本体の一部である — にもかかわらず、その関数全体の値として返して欲しいのもこの値である。そこで、while ループを let 式で包んで、let 式の最後の要素が長さのリストの値を含むようにする。(“増加するカウンタを使ったループの例”, p. 85, を参照。)

以上のことを考えれば、目的の関数が殆ど書けてしまう。

```
;;; while ループ を使う。
(defun lengths-list-many-files (list-of-files)
  "Return list of lengths of defuns in LIST-OF-FILES."
  (let (lengths-list)

;;; 真偽テスト
    (while list-of-files
      (setq lengths-list
            (append
             lengths-list

;;; 長さのリストの生成
              (lengths-list-file
               (expand-file-name (car list-of-files)))))

;;; ファイルのリストを短くする。
      (setq list-of-files (cdr list-of-files)))

;;; リストの長さの最終的な値を返す。
    lengths-list))
```

expand-file-name は built-in 関数であり、ファイル名を絶対パスも含めた省略無しの形に戻すものである。従って、例えば

```
debug.el
```

は

```
/usr/local/emacs/lisp/debug.el
```

と展開される。

その他の、上の関数内に出てくる新しい要素は append だけである。これには一つのセクションを割当てた方が良いだろう。

14.7.1 関数 append

append 関数は、あるリストを、もう一つのリストに追加するものである。例えば、

```
(append '(1 2 3 4) '(5 6 7 8))
```

の結果は次のようになる。

```
(1 2 3 4 5 6 7 8)
```

lengths-list-file によって作成された二つの長さのリストを一つにまとめる際は、このような形になって欲しいのだった。 cons を使った場合と比較してみよう。

```
(cons '(1 2 3 4) '(5 6 7 8))
```

こっちだと、 cons の最初の引数が出来たリストの最初の要素になってしまう。

```
((1 2 3 4) 5 6 7 8)
```

14.8 異なるファイルの定義を再帰を使って数える

while ループではなく再帰を使っても各々のファイルのリストに対して作業することが出来る。再帰を使った lengths-list-many-files は短くて単純な形をしている。

再帰関数は、普通は 'do-again-test'、'next-step-expression'、そして再帰呼び出しの部分からなっている。'do-again-test' では、この関数が自分自身をもう一度呼び出すかどうかを決定する。今の場合は list-of-files がまだ残りの要素を持っているかどうかを調べることになる。'next-step-expression' では、 list-of-files をそれ自身の CDR で置き換える。結果として最後にはリストは空になる。実際の完全なコードは、この説明よりも短い！

```
(defun recursive-lengths-list-many-files (list-of-files)
  "Return list of lengths of each defun in LIST-OF-FILES."
  (if list-of-files                     ; do-again-test
      (append
       (lengths-list-file
        (expand-file-name (car list-of-files)))
       (recursive-lengths-list-many-files
        (cdr list-of-files)))))
```

一言で言うと、この関数は list-of-files の最初のファイルについての長さのリストを、 list-of-files の残りを引数に自分自身を呼び出した結果に追加している。

実際に、各ファイルに対して lengths-list-file を走らせながら recursive-lengths-list-many-files をテストしてみるには、次のようにする。

まず、まだやっていなければ recursive-lengths-list-many-files と lengths-list-files をインストールし（訳註：count-words-in-defun もインストールする必要がある)、その後、次に挙げるＳ式を評価する。ただし、ファイルのパス名は変更する必要があるかもしれない。以下の式では、 Info ファイルと Emacs のソースファイルが通常の位置に置いてある場合に有効である。これを変更したい場合は、これらの式を '*scratch*' バッファにコピーして、それらを編集した後、評価すればよい。

結果は '⇒' の後に示されている。(これらの結果は Emacs version 18.57 についてのものである。他のバージョンの Emacs については、また別の結果が出ることだろう。)

```
(lengths-list-file
 "../lisp/macros.el")
     ⇒ (176 154 86)

(lengths-list-file
 "../lisp/mailalias.el")
     ⇒ (116 122 265)

(lengths-list-file
 "../lisp/makesum.el")
     ⇒ (85 179)
```

```
(recursive-lengths-list-many-files
 '("../lisp/macros.el"
   "../lisp/mailalias.el"
   "../lisp/makesum.el"))
       ⇒ (176 154 86 116 122 265 85 179)
```

このように recursive-lengths-list-many-files は期待した結果を返してくれるはずだ。

次のステップは、結果をグラフに表示するためのデータのリストを準備することである。

14.9 データをグラフに表示するための準備

recursive-lengths-list-many-files 関数は、数のリストを返す。各々の数は関数定義の長さの記録である。我々がこれからやらねばならないのは、このデータをグラフの表示に適した形の数値のデータに変換することである。新しく出来るリストからは、10 より少ない単語やシンボルしか含まない関数定義がどれだけあるかとか、10 から 19 や、20 から 29 までならまででではどうか等ということが分るようになる。

手短に言うと、recursive-lengths-list-many-files 関数が生成したリストを見ていって、各々の範囲に入る関数がどれだけあるかを数えて、それらの数のリストを作ろうというのである。

これまでの経験から、長さのリストを‘CDR’しつつ各々の値を見ていき、それがどの範囲に入るのかを調べてその範囲についてのカウンタを増やす関数を書くことは、特に難しくはないものと察しがつくだろう。

しかしながら、実際に関数を書き下す前に、長さのリストをまずソートして、少ない方から大きい方に並べることによって得られるメリットについて考えるべきである。まず、ソートすることで、各々の範囲に属する関数の数を数えるのが楽になる。これは隣同士の数は同じ範囲に属するか隣同士の範囲に属するかどちらかになるからである。また、リストをソートしてしまえば最大の数と最小の数を簡単に見つけることが出来る。またそこから、後で必要となる最大と最小の差も決定出来ることになる。

14.9.1 リストのソート

Emacs は sort と呼ばれるリストをソートするための関数を持っている。sort 関数は二つの引数を持つ。ソートされるべきリストと、二つの要素の大小を比較する際の述語 (predicate) である。

以前説明したように (Section 1.8.4, “関数に間違ったタイプの引数を与えると”, p. 9, 参照)、述語とは、ある性質が真か偽かを判断する関数のことである。sort 関数は、リストの要素を述語が使用する性質に従って並べ換える。これは、数値以外のリストも、数値以外の基準で — 例えばアルファベットの順番で — sort を利用して並べ換えることが出来ることを示している。

数値で比較する際には < 関数が使われる。例えば、

```
(sort '(4 8 21 17 33 7 21 7) '<)
```

の結果は次のようになる。

```
(4 7 7 8 17 21 21 33)
```

(この例では、引数が sort に渡される際に評価されないように、どちらのシンボルにも引用符が付いていることに注意しよう。)

recursive-lengths-list-many-files 関数によって返されたリストをソートするのは簡単である。

```
(sort
 (recursive-lengths-list-many-files
  '("../lisp/macros.el"
    "../lisp/mailalias.el"
    "../lisp/makesum.el"))
 '<)
```

とするだけだ。結果は次のようになる。

```
(85 86 116 122 154 176 179 265)
```

(この例では sort の最初の引数には引用符がついていない。これは、 sort に渡される前にこのＳ式を評価して、リストを生成する必要があるからである。)

14.9.2 ファイルのリストの作成

recursive-lengths-list-many-files 関数は引数としてファイルのリストを必要とする。これまで実験した例では、これらのリストは手で作っていた。しかし、 Emacs Lisp のソースディレクトリは大変大きいので、これらを一々手で書いているわけにはいかない。そこで、代わりに directory-files 関数を作って、このようなリストを作成する必要がある。

directory-files 関数は、三つの引数を取る。最初の引数はディレクトリの名前であり、文字列である。二番目の引数が non-nil なら、この関数はファイルの絶対パス名を返す。三番目の引数には nil か正規表現を指定する。正規表現を指定した場合、それにマッチするパス名を持つものだけが返されることになる。

(訳註： Version 19 からは引数の数は四つになった。四番目の引数が non-nil なら結果は sort されない。)

従って、例えば私のシステムで

```
(length
 (directory-files "../lisp" t "\\.el$"))
```

とやると、私の version 19.25 の Lisp のソースディレクトリには 307 の‘.el’ファイルがあることが分る。recursive-lengths-list-many-files が返すリストをソートするためのＳ式は次のようになる。

```
(sort
 (recursive-lengths-list-many-files
  (directory-files "../lisp" t "\\.el$"))
 '<)
```

我々の取り敢えずの目標は、 10 未満の単語やシンボルしか含まない関数定義の数はどれだけか、 10 以上、 20 未満ではどうか、 20 以上、 30 未満ではどうか、といったことを調べることである。ソートされた数のリストを使うと、これは簡単である。まずは、 10 未満の要素がどれだけあるかを数え、ついで、その次の要素から 20 未満の要素がどれだけか数え、また次の数から 30 未満の要素がどれだけか数える、というふうに続けていく。 10、 20、 30、 40 等の数は、その範囲の数の最大よりも大きい数になる。これらの数からなるリストは、 top-of-rangesリストと呼べばよいだろう。

しようと思えば、このようなリストを自動的に生成することも可能である。が、今回は手で書いた方が早いだろう。次のような感じである。

```
(defvar top-of-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
 "List specifying ranges for `defuns-per-range'.")
```

範囲を変更するには、このリストを編集すればよい。

次に、この各々の範囲に属する定義の数のリストを作る関数を書く必要がある。明らかに、この関数は引数として sorted-lengths と top-of-ranges リストを取ることになる。

defuns-per-range 関数は、二つ作業を何回も繰り返すことになる。一つは現在の top-of-range の値によって特定される範囲の定義の数を数えること、もう一つはその範囲の数を数え終わったら次に大きな top-of-rangesの値に移ることである。これらの動作は繰り返しなので、 while ループを使うことが出来る。片方の

ループで現在の top-of-range の値で決まる範囲の定義の数をカウントし、もう片方のループでは順に top-of-range の値を選択していく。

各々の範囲について、 sorted-lengths リストの中の幾つかのエントリがカウントされる。従って、 sorted-lengths リストについてのループは top-of-ranges リストのループの中に置かれることになる。大きなギヤのなかの小さなギヤみたいな感じだ。

内部のループでは、該当する範囲の定義の数がカウントされる。これは、今までに何回も見たような単純なループである。(Section 11.1.3, "増加するカウンタを使ったループ", p. 85, を参照。) ループの真偽テストは sorted-lengths リストの中の数が現在の top-of-range の値よりも小さいかどうかを見ることになる。もしそうなら、カウンタを一つ増やして、次の sorted-lengths のエントリに移動する。

結局、内部のループは次のようになる。

```
(while length-element-smaller-than-top-of-range
  (setq number-within-range (1+ number-within-range))
  (setq sorted-lengths (cdr sorted-lengths)))
```

外部のループは top-of-ranges リストの最小値から始まって、順に大きな値に移っていくことになる。そのためには、次のようにすればよい。

```
(while top-of-ranges
  body-of-loop...
  (setq top-of-ranges (cdr top-of-ranges)))
```

これらを合わせると、二つのループは次のようになる。

```
(while top-of-ranges

  ;; 現在の範囲にある要素の数のカウント
  (while length-element-smaller-than-top-of-range
    (setq number-within-range (1+ number-within-range))
    (setq sorted-lengths (cdr sorted-lengths)))

  ;; 次の範囲に移動
  (setq top-of-ranges (cdr top-of-ranges)))
```

更に、一回の外部ループごとに、 Emacs にその範囲に属する定義の数を記録させる必要がある。(リストの中の number-within-range の値である。この目的には、 cons が使える。(Section 7.2, "cons", p. 58, を参照。)

cons 関数はほぼうまく動作するのだが、一つ難点がある。出来るリストでは最初に大きい方の範囲に入る定義の数がきて、最後に小さい方の範囲の数がきてしまうのだ。これは、 cons が新しい要素をリストの先頭に加えていくことと、上の二つのループは小さい方から大きい方へ長さのリストを作成していくために、 defuns-per-range-list が最大の数で終わることからの当然の帰結である。しかし、グラフを表示する際には小さい値の方を先に表示したい。この問題を解決するには、 defuns-per-range-list の順序を逆にしてしまえばよい。これは、 nreverse というリストの順序を逆にする関数を使うとあっさり解決する。

例えば、

```
(nreverse '(1 2 3 4))
```

とすると、

```
(4 3 2 1)
```

が返る。

注意して欲しいのは、 nreverse は「破壊的」であるということである。これは、作用させたリストを変更してしまうことを意味している。(訳註：これは逆の順のリストに設定されるということではなくて、文字通り破壊されてしまうということである。) 今の場合、元の defuns-per-range-list は必要ないので、これが破壊さ

れても何の問題もない。(一方、 reverse 関数は元のリストを逆に並べ換えた新しいリストを返す。この場合は元のリストは変化しない。)

以上を全て組合わせると、 defuns-per-range は次のようになる。

```
(defun defuns-per-range (sorted-lengths top-of-ranges)
  "SORTED-LENGTHS defuns in each TOP-OF-RANGES range."
  (let ((top-of-range (car top-of-ranges))
        (number-within-range 0)
        defuns-per-range-list)

    ;; 外部のループ
    (while top-of-ranges

      ;; 内部のループ
      (while (and
              ;; 数値引数として数が必要
              (car sorted-lengths)
              (< (car sorted-lengths) top-of-range))

        ;; 現在の範囲に入る関数の数を数える
        (setq number-within-range (1+ number-within-range))
        (setq sorted-lengths (cdr sorted-lengths)))

      ;; 内部のループは抜けるが、外部のループには入ったまま

      (setq defuns-per-range-list
            (cons number-within-range defuns-per-range-list))
      (setq number-within-range 0)      ; カウンタを 0 にリセット

      ;; 次の範囲に移動
      (setq top-of-ranges (cdr top-of-ranges))
      ;; 次の範囲のトップを特定
      (setq top-of-range (car top-of-ranges)))

    ;; 外部のループを抜けて最も大きいの範囲に属する関数定義の数
    ;; を数える
    (setq defuns-per-range-list
          (cons
           (length sorted-lengths)
           defuns-per-range-list))

    ;; 昇順で並ぶ関数定義の長さのリストを返す。
    (nreverse defuns-per-range-list)))
```

この関数はちょっとした特徴を除いては、非常に単純である。内部のループの真偽テストは次の通り。

```
(and (car sorted-lengths)
     (< (car sorted-lengths) top-of-range))
```

であって

```
(< (car sorted-lengths) top-of-range)
```

ではない。このテストの目的は sorted-lengths リストの最初の要素がその時点での top-of-range の値よりも小さいかどうかを決定することである。

後に挙げた単純な方のテストでも sorted-lengths リストが nilになるまではうまく動作する。しかし、nil になると、(car sorted-lengths) 式は nil を返す。< 関数は数値と nil、即ち空リストとを比較できないため、 Emacs はここでエラーを出し、関数はそこで実行を止めてしまう。

sorted-lengths リストはカウンタがリストの最後まで辿りつけば常に nil になる。従って、この defuns-per-range 関数の真偽テストの単純なバージョンの方は常に失敗することになる。

この問題は、(car sorted-lengths) 式と and 式を組み合わせることで解決することが出来る。(car sorted-lengths) 式はリストが最低一つでも要素を持てば、non-nil を返す。そしてリストが空になった時だけ nil を返す。and 式は最初に (car sorted-lengths) 式を評価し、もしそれが nil なら < 式を評価する前に偽を返す。しかし、もし (car sorted-lengths) 式が non-nil な値を返せば、< 式も評価し、その値を and 式全体の値を返す。

こうしてエラーが回避出来ることになる。and についての詳細は、Section 12.4, "forward-paragraph: 関数の金脈", p. 99, を参照のこと。

次に defuns-per-range についての簡単なテストを載せておく。最初に (短縮した) リストを top-of-ranges にバインドするＳ式を評価し、次に sorted-lengths リストをバインドするＳ式を評価し、最後に defuns-per-range 関数を評価してみよう。

```
;; (後で使うものよりかは短いリスト)
(setq top-of-ranges
 '(110 120 130 140 150
   160 170 180 190 200))

(setq sorted-lengths
      '(85 86 110 116 122 129 154 176 179 200 265 300 300))

(defuns-per-range sorted-lengths top-of-ranges)
```

次のようなリストが返されるはずである。

```
(2 2 2 0 0 1 0 2 0 0 4)
```

実際、sorted-lengths リストには、110 未満の二つの要素が二つ、110 から 119 までの要素も二つ、120 から 129 までも二つ、といった感じになっている。また、200 以上の値の要素は四つある。

15 グラフを描く準備

我々の目標は、 Emacs のソースの中の様々な長さの関数定義の数をグラフにして表示することであった。

実際には、グラフを生成するには gnuplot のようなプログラムを使っていることだと思う。(gnuplot は GNU Emacs と相性がよい。) しかしながら、今回は一からプログラムを書いていくことにする。そしてその過程で、今まで学んできた事柄を再度確認しつつ、新しいことも学んでいくことにしよう。

この章では、まずグラフを表示する単純な関数を書いてみる。最初の定義は雛型 (*prototype*) であり、グラフを作成するという未知の領域を偵察するために取り敢えず書いてみるといった類のものである。我々は竜を発見するかもしれないし、或いはそれが単なる神話であることが分るかもしれない。ともかく地理感覚が掴めてしまえば、自信もつくし、関数を拡張して軸に自動的にラベルをつけることも出来るようになるだろう。

Emacs はどんな種類のターミナルでも動作するよう柔軟に設計されている。その中にはキャラクターだけしか表示出来ないターミナルも含まれているので、グラフは「タイプライタ」の文字から出来ている必要がある。取り敢えずはアスタリスクを使うのがよいだろう。後から関数を拡張して、この文字をユーザーが選択出来るようにすることも可能だ。

この関数を、 graph-body-print と呼ぶことにしよう。これは引数として numbers-list を取る。現段階ではグラフのラベルは出力せず、本体部分だけを表示することにする。

graph-body-print 関数は、 numbers-list の各々の値に対して、アスタリスクを垂直に並べて表示する。それぞれの高さは、 numbers-list の各要素の値によって決まる。

アスタリスクを垂直に並べる動作は繰り返しである。従って、 while ループか再帰を使って書くことが出来る。

最初の困難は、如何にしてアスタリスクを縦に並べたものを表示するかである。普通、 Emacs 上でタイプすると、文字はスクリーン上に水平に、行ごとに表示されていく。解決への道は二通りある。一つは自分で垂直に挿入するような関数を書くこと、もう一つは元々 Emacs にそのような関数がないか探すことである。

Emacs に特定の機能を持ったものがないか探す場合には、 M-x apropos コマンドを使うことが出来る。このコマンドは、 C-h a (command-apropos) コマンドとほぼ同様なのだが、 command-apropos の方はコマンドだけしか検索しない。 M-x apropos の方は、インタラクティブでないものも含めて正規表現にマッチするものは全てリストしてくれる。我々が探しているものは縦の文字列 (column) を表示 (print) したり挿入 (insert) したりするコマンドである。可能性としては、そのような関数は 'print' とか 'insert' とか 'column' といった文字列を含んでいそうである。そこで、単純に M-x apropos RET print\|insert\|column RET として、結果を見てみる。私のシステムでは、このコマンドの実行には暫く時間がかかり、結局 79 個の関数及び変数のリストが表示された。このリストを探してみると、我々の仕事に役立ちそうなのは、 insert-rectangle だけである。実際、これが我々の求めていた関数である。説明を読んでみよう。

```
insert-rectangle:
Insert text of RECTANGLE with upper left corner at point.
RECTANGLE's first line is inserted at point,
its second line is inserted at a point vertically under point, etc.
RECTANGLE should be a list of strings.
```

簡単なテストを行って、これが本当に求めるものかを確かめてみよう。

以下が、 insert-rectangle 式の直後にカーソルを持っていって、 C-u C-x C-e (eval-last-sexp) とタイプしてみた結果である。この関数は、'"first"', '"second"' そして 'third"' をポイントの下に表示する。関数全体としては、 nil が返る。

```
(insert-rectangle '("first" "second" "third"))first
                                              second
                                              thirdnil
```

(訳註：広く配布されている Mule 2.3 (Emacs version 19.28) では、この関数にバグがある。これは、‘lisp/rect.el’ の最後の方にある move-to-column-strictly の関数定義の中で、四行目の clm を (progn (if force (indent-to column)) column) で置換えることで修正出来る。)

勿論、我々は insert-rectangle 式そのものをグラフを描くバッファに挿入したいのではない。そうではなく、我々のプログラムからこの関数を呼び出したいのである。ただその際、ポイントがバッファの中で、ちゃんと insert-rectangle 関数で縦の文字列を挿入すべき位置にあるかどうかを確かめなければならない。

もしこの文章を Info の中で読んでいるなら、まず‘*scratch*’などの別バッファに移り、ポイントをバッファのどこかに置きつつ M-ESCとタイプして、続けてミニバッファで insert-rectangle 式をタイプしてやることで、このような動作が可能であることが分る。ここでは、S式を評価するのはミニバッファの中だったのだが、その際のポイントの値としては、‘*scratch*’バッファの中のポイントの値が使われたのであった。(M-ESC は eval-expression のキーバインディングである。)

実際にやってみると、ポイントが直前に入力した行の最後にある状態で終わること — つまり、この関数は副作用として、ポイントを移動することが分る。このコマンドを続けて実行すると、次の挿入は前回の位置から下方及び右方向に移動した所から行われる。これでは困る。棒グラフを作成するには、縦の列が隣同士に並ばなければならない。

というわけで、棒グラフを挿入する while ループの各々のサイクルで、ポイントの位置を適切な位置に再配置する必要があることが分った。また、その位置は縦の列の一番上であって底ではない。更に、このグラフを表示する際に、個々の縦の列の長さが全て揃うことはまずない。つまり、各々の縦棒のてっぺんの高さは、前のものと異るのが普通である。従って、単に前と同じ行にポイントを移動していくだけでは駄目である。そうではなくて ...

我々はグラフをアスタリスクで表示しようとしていたのであった。アスタリスクの数は numbers-list の現在の要素で指定される。このアスタリスクを要素とするリストを、 insert-rectangle を呼び出すたびに作成する必要がある。もし、このリストが単に必要な数のアスタリスクだけからなっていたとすると、ポイントの位置をグラフの基準線から正しい行数だけ上の位置におかなければならない。しかし、これは難しい。

その代わり、もし insert-rectangle に対して常に一定の長さのリストを渡すことが出来れば、ポイントも常に同じ行の一つ右の桁に移動すればよくなる。しかし、この場合には insert-rectangle に渡されるリストのいくつかの要素はアスタリスクではなく空白になる。例えば、もしグラフの最大の高さが 5 であり、現在の高さが 3 であったとすると、 insert-rectangle は、次のような引数を必要とする。

```
(" " " " "*" "*" "*")
```

この最後の方法は、縦の列の高さを決定出来さえすれば、それほど難しくはない。縦の列の高さを特定するには二つの方法がある。一つは我々が綺麗に表示されるような高さを勝手に指定してしまうというもの、もう一つは、数のリストを検索して、最大のものをグラフの最大の高さとして使うというものである。後者の方法が難しい場合は、前者を採用するのが楽である。が、 Emacs には、初めから引数の中で最大のものを決定する関数が組込まれている。この関数を使おう。関数の名前は max で、全ての引数の中で最大のものを返す。引数は数でなければならない。従って、例えば、

```
(max  3 4 6 5 7 3)
```

は 7 を返す。(これに対して min という関数は、引数の中で最小のものを返す。)

しかしながら、単に numbers-list の上で max を呼び出すことは出来ない。 max は数のリストではなく、数を引数として要求するからだ。従って、

```
(max  '(3 4 6 5 7 3))
```

とやると、次のようなエラーメッセージが返される。

```
Wrong type of argument:  number-or-marker-p, (3 4 6 5 7 3)
```

従って、リストの中身を引数として関数に渡すような関数が必要になる。 apply と呼ばれる関数がこの役目を果してくれる。この関数はその最初の引数である関数に、残りの引数を適用 (apply) する。この時、最後の引数はリストでなければならない。

例えば

```
(apply 'max 3 4 7 3 '(4 8 5))
```

は 8 を返す。

(ついでにいうと、私はこの関数をこの文書のようなものを使わずに見つけだす方法を知らない。他の search-forward や insert-rectangle といった関数ならば、名前の一部分を推測して apropos を使えばよい。けれども apply については、名前の元となるメタファ — 最初の引数に残りを 'apply' する — は明らかであるにしても、初心者が apropos や他の助けを借りてこのような特定な単語に辿り着けるとは思わない。勿論、私が間違っているのかもしれない。結局は、関数はそれを作った人によって名付けられるものなのである。)

apply の二番目以降の引数は省略可能である。従って、 apply を使って、関数にリストの要素を引数として渡して呼び出すことも出来る。例えば、次は 8 を返す。

```
(apply 'max '(4 8 5))
```

この最後の例が我々の目的にあっている。 recursive-lengths-list-many-files 関数は、数のリストを返すが、これを max に適用することが出来るわけである。(max にソートされたリストを渡すことも出来るが、この場合はソートされていてもされていなくても関係ない。)

というわけで、グラフの最大の高さを求める操作は次の通りである。

```
(setq max-graph-height (apply 'max numbers-list))
```

では、グラフの縦の列を描くための文字列のリストをどうやって作ればよいかという問題に戻ろう。この関数はグラフの最大の高さと個々の縦の列の中のアスタリスクの数から、 insert-rectangle コマンドが挿入する文字列のリストを返す必要がある。

各々の縦の列は空白とアスタリスクからなる。高さとアスタリスクの数が分れば、空白の数はその差として求められる。そして空白とアスタリスクの数が分れば、二つの while ループを使ってリストを作ることが出来る。

```
;;; 最初のバージョン
(defun column-of-graph (max-graph-height actual-height)
  "Return list of strings that is one column of a graph."
  (let ((insert-list nil)
        (number-of-top-blanks
         (- max-graph-height actual-height)))

    ;; アスタリスクを詰める
    (while (> actual-height 0)
      (setq insert-list (cons "*" insert-list))
      (setq actual-height (1- actual-height)))

    ;; 空白を詰める
    (while (> number-of-top-blanks 0)
      (setq insert-list (cons " " insert-list))
      (setq number-of-top-blanks
            (1- number-of-top-blanks)))

    ;; リスト全体を返す
    insert-list))
```

この関数をインストールしてから次のＳ式を評価すると、求めるリストが得られるはずだ。

```
(column-of-graph 5 3)
```

を評価すると、

```
(" " " " "*" "*" "*")
```

が返されるというわけである。

前にも書いたが、 column-of-graph には大きな欠陥がある。空白と印のために用いられる記号はスペースとアスタリスクに「ハードコード」されている。これは雛型としてはよい。が、あなたや他のユーザは他の記号を使いたいと思うことも多いだろう。例えば、グラフ関数をテストしてみる際には、 insert-rectangle 関数が呼ばれた時にポイントの位置がきちんと移動されているかどうかを見るために、スペースの代わりに終止符を使いたいと思うだろう。また、アスタリスクの代わりに '+' 等の記号を使ったりしたいと思うこともあるに違いない。更に、グラフの桁数をディスプレイの一桁の幅よりも広く取りたいと思うこともあるだろう。そういうわけでプログラムはもっと柔軟であるべきである。そのための方法としては、スペースとアスタリスクを graph-blank と graph-symbol という二つの変数で置き換えて、これらの変数を別に定義することが考えられる。

また、説明文字列も解りやすいとは言えない。以上のことを考慮すると、次のような関数に辿り着く。

```
(defvar graph-symbol "*"
  "String used as symbol in graph, usually an asterisk.")

(defvar graph-blank " "
  "String used as blank in graph, usually a blank space.
graph-blank must be the same number of columns wide
as graph-symbol.")
```

(defvar の説明については、 Section 8.4, "defvar を用いた変数の初期化", p. 70, を参照。)

```
;;; 二番目のバージョン
(defun column-of-graph (max-graph-height actual-height)
  "Return list of MAX-GRAPH-HEIGHT strings;
ACTUAL-HEIGHT are graph-symbols.
The graph-symbols are contiguous entries at the end
of the list.
The list will be inserted as one column of a graph.
The strings are either graph-blank or graph-symbol."

  (let ((insert-list nil)
        (number-of-top-blanks
         (- max-graph-height actual-height)))

    ;; graph-symbols を詰める
    (while (> actual-height 0)
      (setq insert-list (cons graph-symbol insert-list))
      (setq actual-height (1- actual-height)))

    ;; graph-blanks を詰める
    (while (> number-of-top-blanks 0)
      (setq insert-list (cons graph-blank insert-list))
      (setq number-of-top-blanks
            (1- number-of-top-blanks)))


    ;; リスト全体を返す
    insert-list))
```

もし望むなら、 column-of-graph も書換えて、オプションとして棒グラフだけでなく、線グラフも書けるようにも出来る。これはそんなに難しいことではない。線グラフをどう描くかだが、例えば棒グラフにおいて、一番上の点から下は全て空白にしてしまえば、それはもう棒グラフとは呼べないだろう。線グラフのための縦の文字列を作るには、まずは値よりも一つだけ少ない数の空白のリストを作り、ついで cons を用いてリストにグラフ記号を追加し、最後に、上の余白部分の数の空白を付け足せばよい。

こういう関数を実際に書くのも簡単だが、今の所は必要ないので、書かないでおくことにする。が、ともかく書くことは出来るし、またこの関数を一度書いてしまえば、 column-of-graph で使うことも出来る。ここで大切

なことは、他の部分の書き換えは殆どしなくても良いということである。即ち、拡張しようと思えば簡単に出来るわけだ。

さて、やっと、実際の最初のグラフを書いてみる所まで来た。ここではグラフの本体だけを表示し、縦軸や横軸のラベルは表示しない。そこで、この関数を graph-body-print と呼ぶことにする。

15.1 関数 graph-body-print

前節までの準備の後では、 graph-body-print 関数はあっというまに出来てしまう。この関数は、数値のリストから各々の縦の列の中のアスタリスクの数を決定し、一桁おきにアスタリスクを用いたグラフを表示する。これは繰り返しの動作なので、現象 while ループか、再帰を使って書くことが出来る。このセクションでは、while ループを用いて定義を書いてみよう。

column-of-graph 関数は、引数としてグラフの高さを必要とする。この値を定めたなら、局所変数として記録しておくべきである。

これらのことから、このバージョンにおいては次のような while ループのテンプレートが出来る。

```
(defun graph-body-print (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))

    (while numbers-list
      insert-columns-and-reposition-point
      (setq numbers-list (cdr numbers-list)))))
```

この中の空きスロットを埋めていくことになる。

当然、グラフの高さの決定には（apply 'max numbers-list）式を使うことが出来る。

while ループは numbers-list の各要素に対して一回ずつ回る。このリストは（setq numbers-list (cdr numbers-list)）式によって短くなっていき、各時点でのリストの CAR が column-of-graph に引数として渡される。

この while ループの各サイクルで、 column-of-graph によって返されたリストが insert-rectangle 関数に渡される。 insert-rectangle 関数はポイントを挿入された矩形の右下のポイントに移動するので、それを矩形が挿入される前の位置に戻してから、次の位置に水平方向に移動してやる必要がある。そこで次の insert-rectangle が呼ばれるわけである。

単独の空白とアスタリスクを使った場合などのように、もし挿入される棒グラフが一桁の幅ならば、移動のためのコマンドは単に（forward-char 1）となる。しかし、この幅はもっと大きくなるかもしれない。従って、（forward-char symbol-width）と書く方が良い。 symbol-widthは graph-blank の長さであり、（length graph-blank）という式で求めることが出来る。 symbol-width という変数をグラフの幅にバインドする一番良い位置は let 式の変数リストの中である。

以上のことを総合すると次のような関数定義になる。

```
(defun graph-body-print (numbers-list)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values."

  (let ((height (apply 'max numbers-list))
        (symbol-width (length graph-blank))
        from-position)
```

```
        (while numbers-list
          (setq from-position (point))
          (insert-rectangle
           (column-of-graph height (car numbers-list)))
          (goto-char from-position)
          (forward-char symbol-width)
          ;; 各桁ごとのグラフの描写
          (sit-for 0)
          (setq numbers-list (cdr numbers-list)))
        ;; X 軸のラベルのためにポイントを移動
        (forward-line height)
        (insert "\n")
    ))
```

この関数では、ひとつ予期していなかったＳ式が出てくる。それは `while` ループの中の（`sit-for 0`）式である。この式を使うことで、グラフを表示する過程を見るのが楽しくなる。この式は Emacs に 0 時間待ってから画面を再描画させるものである。これを上の位置に置くと棒グラフが一つずつ描かれていくことになる。逆に置かなければ、関数が仕事を全て終了するまでグラフは描かれない。

次のようにすれば、この `graph-body-print` を短いリストに対してテストしてみることが出来る。

1. まず `graph-symbol`、`graph-blank`、`column-of-graph` そして `graph-body-print` をインストールする。
2. 次のＳ式をコピーする。

   ```
   (graph-body-print '(1 2 3 4 6 4 3 5 7 6 5 2 3))
   ```

3. '`*scratch*`' バッファに移り、グラフを表示させたい位置にカーソルを置く。
4. `M-ESC` (`eval-expression`) とタイプする。
5. `C-y` (`yank`) を使って `graph-body-print` 式をミニバッファに yank する。
6. `RET` を押して、`graph-body-print` 式を評価する。

Emacs は次のようなグラフを表示するはずである。

```
                    *
                  *   **
                  *  ****
                 *** ****
                ********* *
               ***********
              ************
```

15.2 関数 `recursive-graph-body-print`

`graph-body-print` 関数は、再帰を使って書くことも出来る。この場合は、二つの部分に分けて書くことになる。外側の関数 (wrapper) で `let` 式を使って、グラフの最大の高さのように一度だけ決めればよいような幾つかの変数の値を設定し、内側の関数で、再帰呼び出しを使ってグラフを表示するわけである。

外側の関数 (wrapper) は特に複雑ではない。

```
(defun recursive-graph-body-print (numbers-list)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values."
  (let ((height (apply 'max numbers-list))
        (symbol-width (length graph-blank))
        from-position)
    (recursive-graph-body-print-internal
     numbers-list
     height
     symbol-width)))
```

再帰関数の方は、ちょっとばかり難しい。これは 'do-again-test'、グラフ表示コード、再帰呼び出し、そして 'next-step-expression' の四つの部分からなる。'do-again-test' は `number-list` にまだ要素が残っているかを判定する `if` 式である。もし残っていれば、一本の棒グラフを書いてからまた自分自身を呼び出す。関数が自分自身をもう一度呼び出すかどうかは、結局は 'text-step-expression' が生成する値による。これは、短縮版の `number-list` に作用するような呼び出しを行う。

```
(defun recursive-graph-body-print-internal
  (numbers-list height symbol-width)
  "Print a bar graph.
Used within recursive-graph-body-print function."
  (if numbers-list
      (progn
        (setq from-position (point))
        (insert-rectangle
         (column-of-graph height (car numbers-list)))
        (goto-char from-position)
        (forward-char symbol-width)
        (sit-for 0)     ; 各桁ごとのグラフの描写
        (recursive-graph-body-print-internal
         (cdr numbers-list) height symbol-width))))
```

これをインストールして実際にテストしてみることが出来る。次にサンプルを挙げる。

```
(recursive-graph-body-print '(3 2 5 6 7 5 3 4 6 4 3 2 1))
```

これを実際に評価してみると、次のようなグラフが描かれる。

```
                  *
                 **   *
                ****  *
                **** ***
              * *********
              ************
              ************
```

`graph-body-print` も `recursive-graph-body-print` もグラフの本体部分のみを描く関数である。

15.3 軸を表示する

グラフには、それが何を表わすかを示すために軸を表示する必要がある。一度だけしかグラフを描かないのならば、Emacs の Picture mode を利用して手で軸を描くのも結構である。しかし、この関数は何度も利用するかもしれない。

というわけで、基本的な `print-graph-body` 関数を拡張して、自動的に横軸と縦軸のラベルを表示するようにしてみた。この関数には特に新しい事柄は含まれていないので、これについての説明は次の所ですることにする。 Appendix C, “ラベルと軸が付いたグラフ”, p. 173.

15.4 練習問題

上と同様なグラフを表示する関数で、横棒のグラフを描くバージョンを作りなさい。

16 '.emacs' ファイル

「Emacs を好きになるために Emacs を好きになる必要はない。」– この、一見パラドクスのように聞こえる格言こそが、 GNU Emacs の秘密である。ただ箱を開けて取り出したままの Emacs は一般的な道具でしかない。大抵の人は、それを自分自身が使いやすいようにカスタマイズする。

GNU Emacs は多くの部分が Emacs Lisp で書かれている。これは、 Emacs Lisp の式を書くことで、変更したり拡張したり出来るということを意味する。

Emacs のデフォルトの設定を素直にありがたく使う人達もいる。それでも Emacs は C のファイルを編集する際は C mode になるし、 Fortran のファイルを編集する際は Fortran mode になるし、特に設定されていないファイルについては Fundamental mode になる。誰が Emacs を使うか分らない状況では、これは合理的な設定である。人々がどのファイルに対して何をしたいかを把握出来る人など何処にもいないだろう。 Fundamental mode は把握出来ないようなファイルに対しての正しいデフォルトである。ちょうど C mode が C のコードを編集する際の正しいデフォルトであるのと同じである。しかし、もし誰が Emacs を使おうとしているかが分る場合 — 例えば自分自身が使う場合 — なら Emacs をカスタマイズするのが合理的である。

例えば私は、適当なモードが見つからないようなファイルに対しては Fundamental mode にしたいことは殆どない。そういう場合には Text mode を使うことが多い。そこで Emacs をカスタマイズする。結果として、私にとって使いやすいものになる。

Emacs をカスタマイズしたり拡張したりするには '~/.emacs' ファイルを書くことになる。これは個人用の初期化ファイルである。中身は Emacs Lisp で書かれ、 Emacs に何をするかを伝える。

この章では、簡単な '.emacs' の説明をする。より多くの情報については、以下を参照して欲しい。 Section "The Init File" in *The GNU Emacs Manual*, Section "The Init File" in *The GNU Emacs Lisp Reference Manual*.

16.1 サイトごとの初期化ファイル

Emacs は個人的な初期化ファイルに加えて、サイト用の初期化ファイルが存在すれば、それも自動的にロードする。これらは '.emacs' と同様な形式で書かれるが、全員に対してロードされる。

このサイト用の初期化ファイルの中の 'site-load.el' と 'site-init.el' の二つは、 Emacs が dump される際に、ロードされ、かつ標準的なものとして 'dump' される。(Dump することで、 Emacs はより速くロード出来る。しかしながら、一旦ファイルがロードされ dump されると、もう一度自分でロードしたり再 dump したりしない限り、変更は Emacs に反映されない。これについては Section "Building Emacs" in *The GNU Emacs Lisp Reference Manual*, 及び 'INSTALL' ファイルを参照のこと。)

これら以外にも三つのサイト用の初期化ファイルがあり、それらは存在すれば Emacs の起動時に自動的にロードされる。この三つのファイルというのは、 '.emacs' の前にロードされる 'site-start.el'、及び '.emacs' の後にロードされる 'default.el' と端末タイプファイルである。

'.emacs' での設定や定義は、 'site-start.el' での設定や定義を上書きする。そして、 'default.el' や端末タイプファイルが更にその設定や定義を上書きすることになる。(端末タイプファイルによる上書きを防ぐには、 `term-file-prefix` を `nil` にする。 Section 16.10, "ちょっとした拡張", p. 148, を参照。)

配布の中に含まれる 'INSTALL' ファイルには、 'site-init.el' と 'site-load.el' についての説明がある。

'loadup.el'、 'startup.el'、そして 'loaddefs.el' ファイルがロードをコントロールしている。これらのファイルは Emacs の配布の中の 'lisp' ディレクトリの中にあるので、読んでみると良い。

'loaddefs.el' ファイルには、何を個人的な '.emacs' ファイルの中に書き、何をサイト用の初期化ファイルに書くべきなのかということについて多くの有益な示唆が書かれている。

16.2 あるセッションでの変数の設定

私の使用している Emacs version 19.23 には、 edit-options コマンドを利用して設定出来る 392 のオプションがある。これらの「オプション」というのは以前説明したように、 defvar を使って定義された変数に過ぎない。

Emacs では、これらの変数が「容易に設定出来るように意図されたものか」ということを、説明文字列の最初の一文字で判別している。つまり、もし最初の文字が ‘*’ だったら、その変数はユーザが設定出来るオプションである。 (Section 8.4, “defvar を用いた変数の初期化”, p. 70, を参照。)

edit-options コマンドは Emacs Lisp のライブラリを書いた人々が簡単に設定出来るようにするべきだと考えた Emacs の変数を全て列挙してくれる。そして、変数を簡単に再設定出来るようなインターフェースを与えてくれる。

一方で、 edit-options で設定されたオプションは現在の編集セッションの間にしか効力を持たない。新しく設定した値は次のセッションには保存されない。 Emacs は起動される度に新しくソースコードから defvar されたオリジナルの値を読み取るわけである。次のセッションでもその変数の設定を使いたい場合には、‘.emacs’ 等の起動の度にロードされるファイルで setq 式を使って値を上書きする必要がある。

私にとっての edit-options コマンドの主な使い道というのは、自分の ‘.emacs’ ファイルで設定したくなるような変数を示唆してもらうというものである。このコマンドが出すリストを見てみることを強く薦めておく。

Section “Editing Variable Values” in *The GNU Emacs Manual*, を見れば、より詳しいことが書かれている。

16.3 ‘.emacs’ の書き方

Emacs を起動すると、コマンドラインに ‘-q’ を指定して読み込まないようにした場合を除き、‘.emacs’ ファイルがロードされる。 (emacs -q コマンドでは、素のままの Emacs が起動される。)

‘.emacs’ ファイルは Lisp 式を含んでいる。大抵は値を設定するだけだが、時には関数定義が書かれていることもある。

初期化ファイルの簡単な説明については、次を参照。 Section “The Init File ‘~/.emacs’” in *The GNU Emacs Manual*.

この章でも同じようなことを説明するのだが、その際、私が長く使ってきた私自身の ‘.emacs’ を素材にすることにする。

ファイルの最初の部分はコメントからなる。これは個人的なメモとして利用している。勿論、今はその内容を把握しているが、最初はそうではなかった。

```
;;;; Bob's .emacs file
; Robert J. Chassell
; 26 September 1985
```

日付を見てみよう！ このファイルを使い初めたのは随分と昔だ。それ以来、内容を追加してきたわけである。

```
; Each section in this file is introduced by a
; line beginning with four semicolons; and each
; entry is introduced by a line beginning with
; three semicolons.
```

(日本語訳)

```
; このファイルの各セクションは行頭がセミコロンであるような
; 行から始まる。また、各項目は行頭が三つのセミコロンである
; ような行から始まる。
```

ここでは Emacs Lisp のコメントの通常の便宜上の慣習を説明している。セミコロンで始まる行は全てコメントである。二つ、三つ、ないしは四つのセミコロンがセクションやサブセクションのために使われる。(コメントについての説明は次を参照。 Section “Comments” in *The GNU Emacs Lisp Reference Manual.*)

```
;;;; The Help Key
; Control-h is the help key;
; after typing control-h, type a letter to
; indicate the subject about which you want help.
; For an explanation of the help facility,
; type control-h three times in a row.
```

(日本語訳)

```
;;;; ヘルプキーについて
; Control-h がヘルプキーである。
; control-h をタイプした後、見たいヘルプの内容を示す
; 文字をタイプする。ヘルプ自体の簡単な説明を見るには
; control-h を三回タイプすればよい。
```

ここでは特に C-h を三回タイプすることでヘルプが表示されることを注意しておく。(訳註 : これは訳者の Mule-2.3 (Emacs version 19.28) では二回押すだけで表示された。仕様が変わったのだと思う。)

```
; To find out about any mode, type control-h m
; while in that mode.  For example, to find out
; about mail mode, enter mail mode and then type
; control-h m.
```

(日本語訳)

```
; あるモードがどんなものか知りたい場合は、そのモードで
; control-h m とタイプする。例えば、 mail mode について
; 知りたい場合は mail mode に入ってから control-h m と
; タイプする。
```

私はこれを「モードヘルプ (mode help)」と読んでいるのだが、大変便利である。普通はこれを読むだけで必要なことが殆ど分ってしまう。

勿論、このようなコメントは ‘.emacs’ ファイルに含める必要はない。私がこれらを含めたのは、よくモードヘルプのことやコメントの書き方を忘れたためだ。で、これを見れば思い起こすことが出来たわけである。

16.4 Text モードと Auto Fill モード

さて、 Text mode や Auto Fill mode をオンにする箇所まで来た。

```
;;; Text mode and Auto Fill mode
; The next two lines put Emacs into Text mode
; and Auto Fill mode, and are for writers who
; want to start writing prose rather than code.
```

(日本語訳)

```
;;; Text mode と Auto Fill mode
; 次の二つの行は、 Emacs を Text mode や Auto Fill mode
; にするためのものである。コードを書くよりは普通の文章
; を書くことが多い人向けの設定である。

(setq default-major-mode 'text-mode)
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

これが、この ‘.emacs’ の中で、どこかの忘れっぽい人間のための備忘録以外のちゃんとした役目を果たす最初の部分である。

この括弧で挟まれた二行の内の最初の一行は、ファイルを読み込む際に、そのファイルに C mode のような特定のモードが設定されていない場合には Text mode に入るようにするものである。

Emacs は、ファイルを読む際にその拡張子を見る。(拡張子というのは ‘.’の後に続く部分である。もしファイルが ‘.c’ や ‘.h’ という拡張子で終わっていれば、 Emacs は C mode になる。また、 Emacs はファイルの最初の空行でない行も見る。例えば最初の行に ‘-*- C -*-’ と書かれていれば、 Emacs は C mode になる。Emacs は自動判別のための拡張子とモード指定のリストを持っている。更に Emacs は、そのバッファの最後のページ付近に “local variable list” があれば、それも見る。

これについては、 *The GNU Emacs Manual* の “How Major Modes are Chosen” や “Local Variables in Files” 等のセクションを参照のこと。

さて、‘.emacs’ ファイルに戻ろう。

もう一度、さっきの行を書いておく。これは何をするのものだろう？

```
(setq default-major-mode 'text-mode)
```

この行は、短いながらも完全な Emacs Lisp の式である。

我々は既に `setq` には慣れ親しんでいる。これは、後に続く変数 `default-major-mode` をその後の `text-mode` という値にセットするものだ。 `text-mode` の先頭の引用符は、 Emacs に `text-mode` がシンボルとして何を表しているかを見るのではなく、文字通りそのまま扱うよう指示するものである。 `setq` のより詳しい説明については次を参照。 Section 1.9, “変数の値の設定”, p. 11. 大切なポイントは、‘.emacs’ で値をセットする手続きは、 Emacs の他の場所での手続きと全く違いはないということである。

次は二行目である。

```
(add-hook 'text-mode-hook 'turn-on-auto-fill)
```

この行では、 `add-hook` コマンドで `turn-on-auto-fill` を `text-mode-hook` という変数に加えている。`turn-on-auto-fill`はプログラムの名前であり、名前から推察される通りのことを行う。 Auto Fill mode をオンにするのである。

Emacs は Text mode に入るたびに、 Text mode にフック (hook) されたコマンドを実行する。従って、Emacs が Text mode になるごとに Auto Fill mode もオンになるわけである。

まとめると、最初の行は、特に拡張子や最初の空でない行や局所変数からどのモードにしてよいか分らない場合には Text mode になるよう設定するものである。

Text mode では、物書きには便利な構文テーブルが設定されたりする。例えば、アポストロフィ ‘’’ は普通の文字ように単語の一部だと解釈するが、終止符や空白は単語の一部だとは考えない。従って `M-f` では ‘it’s’ 全体を飛び越えて進む。一方、 C mode では、‘it’s’ のちょうど ‘t’の直後の位置で止まる。

また二行目は、 Text mode に入る際に同時に Auto Fill mode もオンにするためのものである。 Auto Fill mode では、 Emacs は自動的に長い行を次の行に折り返してくれる。その際は、単語を寸断したりせずちゃんと単語と単語の間を折り返すようになっている。

Auto Fill mode がオフになっている場合は、行はあなたがタイプしていくとそのまま右に伸びていく。行が右端まで到達した場合、 `truncate-lines` の値をどう設定しているかによって、右の部分が見えなくなったり、或いは汚い形で表示されたりする。

16.5 メールのエイリアス

以下が mail aliases 等を使えるようにする `setq` 式、及びそれについての備忘録である。

```
;;; Mail mode
; To enter mail mode, type ‘C-x m’
; To enter RMAIL (for reading mail),
; type ‘M-x rmail’

(日本語訳)
;;; Mail mode
; Mail mode に入るには、‘C-x m’ とタイプする。
; RMAIL (mail を読むためのもの) にはいるには、
; ‘M-x rmail’ とタイプする。

(setq mail-aliases t)
```

この setq コマンドは、変数 mail-aliases の値を t にセットする。 t は真を意味するので、この行は「どうぞ、 mail aliases を使って下さい」という意味になる。

Mail aliases というのは、長い電子メールアドレスや電子メールアドレスのリストに対する便宜上の短い名前のことである。あなたの ‘aliases’ を保存しておくファイルは ‘~/.mailrc’ である。 alias は次のような形式で書けばよい。

```
alias geo george@foobar.wiz.edu
```

George にメッセージを書く場合は、アドレスに ‘geo’ と書くだけでよい。するとメイラーは自動的に ‘geo’ をフルアドレスに展開してくれる。

16.6 Indent Tabs モード

Emacs はデフォルトでは、リージョンを整形する際に沢山の空白の連続にはタブを挿入する。(例えばあなたは、沢山の行を一度にインデントするために indent-region を使ったりすることだろう。) タブは端末や普通の印刷では綺麗に見える。しかし TEX ではタブは無視されるために、 TEX や Texinfo を使う際には出力が汚くなってしまう。

次のようにすれば、 Indent Tabs mode をオフに出来る。

```
;;; Prevent Extraneous Tabs
(setq-default indent-tabs-mode nil)
```

この行では、今まで使ってきた setq ではなく setq-default を使っていることに注意しよう。 setq-default コマンドは、その変数に対するバッファローカルな値がない場合に限り、その値をセットする。

The GNU Emacs Manual の中の “Tabs vs. Spaces” や “Local Variables in Files” を参照。

16.7 幾つかのキーバインディング

次は、幾つかの個人的なキーバインディングの設定である。

```
;;; Compare windows
(global-set-key "\C-cw" 'compare-windows)
```

compare-windows は、現在のウィンドウと次のウィンドウの中のテキストを比較してくれるという、洒落たコマンドである。これは、各々のウィンドウの中のポイントの位置から比較を開始して、両方が同じである所まで進む。私はこのコマンドをいつも利用している。

また、これを見れば、全てのモードに対する大域的なキーバインディングを設定する方法が分る。

コマンドは global-set-key だ。この後にキーバインディングが続く。‘.emacs’ ファイルではキーバインディングは上に示した通りの方法で書かれる。 \C-c は ‘control-c’, つまり、「control キーと c を同時に押す」ことを表わす。また、 w は「w を押す」ことを表わす。説明の中では、 C-c w のように書かれる。(CTL キーで

はなく M-c のような META キーを使ったバインドをしたい場合は、\M-c のように書く。詳しくは次を参照。Section “Rebinding Keys in Your Init File” in *The GNU Emacs Manual*.)

このキーで呼び出されるコマンドは compare-windows である。この時、compare-windows の先頭に引用符を付けることに注意しよう。そうしないと、Emacs は最初にこのシンボルの値を評価しようとしてしまう。

二重引用符と、‘C’ の前のバックスラッシュ、及び単独の引用符の三つはキーバインディングには必須であるにもかかわらず、私はよく忘れてしまう。幸いなことに、これらは ‘.emacs’ を見れば思い出すことが出来る。

C-c w というキーバインディングそのものについても書いておく。これは、C-c という前置キーと一文字 — 今の場合は w— からなっている。このような C-c に続けて一文字というキー設定は、特に個人的な用途に残されている。もしあなたが Emacs を拡張したい場合、これらのキーを公の用途に使わないで欲しい。その代わり、C-c C-w のようなキーを使うべきである。そうでないと個人用にバインドするキーがなくなってしまう。

以下にコメント付きで他のキーバインディングを書いておく。

```
;;; Keybinding for ‘occur’
; I use occur a lot, so let’s bind it to a key:
; (私は occur をよく使うので、キーにバインドしておく。)
(global-set-key "\C-co" ’occur)
```

occur コマンドは、現在のバッファの中で正規表現にマッチする部分を含む行を全て表示してくれるコマンドである。マッチした行は、‘*Occur*’というバッファに表示される。このバッファはその位置に簡単に飛ぶためにも使われる。

次に、キーバインディングの解除の仕方を示す。特定のキーバインディングを使えなくするわけである。

```
;;; Unbind ‘C-x f’
(global-unset-key "\C-xf")
```

このキーバインディングを解除するのには理由がある。私は、自分がよく C-x C-f とタイプしようとして、間違って C-x f とタイプしてしまうことに気が付いた。ファイルを読み込もうとして、偶然行詰めの幅を設定してしまうわけである。結果的に大抵は変な幅が設定されてしまう。私はこの幅を変更することは殆どないので、単にキーバインディングを解除することにしたのだ。

次は現在のキーをバインドし直すものである。

```
;;; Rebind ‘C-x C-b’ for ‘buffer-menu’
(global-set-key "\C-x\C-b" ’buffer-menu)
```

デフォルトでは、C-x C-b は list-buffer コマンドを起動するものである。このコマンドは現在のバッファのリストを別のウィンドウに表示する。私は大抵はそのウィンドウで何かをしたいことが多いので buffer-menu コマンドの方が好きである。これは単にバッファのリストを表示するだけでなく、ポイントをそのウィンドウに移動してくれる。

16.8 ファイルのロード

GNU Emacs のコミュニティの沢山の人々が Emacs の拡張を行ってきた。時が経つにつれ、これらの拡張が新しいリリースに含まれることも多くなった。例えばカレンダーや日記のパッケージは今や Emacs version 19 の配布の一部になっている。これらは標準の Emacs version 18 の配布には含まれていなかったものだ。

(私は Calc も Emacs の大切な一部だと考えている。パッケージが大きいために別配布の形を取ってはいるものの、これも標準の配布の一部と言えるだろう。)

load コマンドを使うと、ファイルの中身全てを評価することが出来る。従って、そのファイルの関数や変数を全て Emacs にインストールすることが可能だ。例えば、

```
(load "~/emacs/kfill")
```

というS式は、ホームディレクトリの ‘emacs’ というサブディレクトリの下の ‘kfill.el’ ファイルを評価、即ちロードする。(もしバイトコンパイルされた ‘kfill.elc’ があれば、そちらがロードされる。こっちの方が速くロード、実行出来る。)

(‘kfill.el’ は、Kyle E. Jones の ‘filladapt.el’ パッケージを Bob Weinere が改造したもので、「すっきりとして、こうるさいくない折り返し及びパラグラフの行詰めを提供し、ニュースやメール、それに Lisp, C++, Postscript やシェルコマンドなんかを含めてインデントしてくれる」パッケージである。私はこれをよく利用するので、標準の配布の中に含めてくれることを願っている。)

沢山の拡張パッケージをロードしたい場合は、上のようにそのファイルの位置を正確に指定するのではなく、あるディレクトリを Emacs の load-pathに指定するとよい。私はそうしている。この場合には、Emacs はファイルをロードする際にデフォルトのディレクトリのリストに加えてそのディレクトリも検索してくれるようになる。(デフォルトのディレクトリは Emacs を作成する際に ‘paths.h’ で指定される。)

次のコマンドで、‘~/emacs’ ディレクトリを現在のロードパスに含めることが出来る。

```
;;; Emacs Load Path
(setq load-path (cons "~/emacs" load-path))
```

ついでに言っておくと、load-library は関数をロードする際のインタラクティブなインターフェースも提供してくれる。この関数の完全なコードは次のようになっている。

```
(defun load-library (library)
  "Load the library named LIBRARY.
This is an interface to the function ‘load’."
  (interactive "sLoad library: ")
  (load library))
```

load-library という関数の名前では ‘library’ という言葉を便宜的に ‘file’ の同意語として見ている。load-library コマンドのソースは ‘files.el’ ライブラリから来ている。

これとちょっと違う動作をするインタラクティブなコマンドとして、load-file がある。これが load-library とどう違うかについてのより詳しい情報は次を参照。 Section “Libraries of Lisp Code for Emacs” in *The GNU Emacs Manual*.

16.9 オートロード

その関数を含むファイルをロードしたり、関数定義を直接評価したりしてインストールする代わりに、その関数が実際に最初に使われるまではインストールはしないが、いつでも呼び出せる状態にはしておくということも出来る。これはオートロード (*autoloading*) と呼ばれる。

オートロードされた関数を実行すると、Emacs は自動的にその定義を含むファイルを評価して、その関数を呼び出す。

オートロード関数を使うと Emacs をより速く起動出来る。というのも起動時にはそのライブラリはロードされないからだ。その代わり、最初にオートロードされた関数を使う際には、それを含むファイルを評価する間ちょっと待たされることになる。

めったに使われない関数はしばしばオートロードされる。‘loaddefs.el’ライブラリには bookmark-set から wordstar-mode まで何百ものオートロードされる関数が含まれている。勿論、個人的にこれらの「めったに使わない」関数を頻繁に使うことになることもあるかもしれない。この場合は、その関数のファイルを load 式を使って .emacs ファイルの中でロードしておくべきだろう。

私の Emacs version 19.23 用の ‘.emacs’ ファイルでは、元々オートロードされるような関数を含む 17 のライブラリがロードされている。(実際には、これらの関数を「dump」した Emacs を作るべきなのだが、忘れていたのだ。dump についての詳しい情報は、 Section “Building Emacs” in *The GNU Emacs Lisp Reference Manual*, や ‘INSTALL’ を見て欲しい。)

或いはまた ‘.emacs’ ファイルにオートロード式を書きたいこともあるだろう。 autoload は五つの引数を取る組込み関数であり、最後の三つの引数は省略可能である。最初の引数はオートロードする関数の名前、二番目の

引数はロードされるファイル名、三番目は関数の説明文、四番目はその関数をインタラクティブに呼び出せるかどうか、そして最後の五番目の引数は、オブジェクトのタイプ —autoload は関数だけでなく、キーマップやマクロも扱えるので — である。(デフォルトは関数である。)

以下に典型的な例を挙げる。

```
(autoload 'html-helper-mode
  "html-helper-mode" "Edit HTML documents" t)
```

この式は、html-helper-mode 関数を、‘html-helper-mode.el’ ファイルのからロードする。(或いは、もし ‘html-helper-mode.elc’ があれば、そちらから読み込む。) このファイルは load-path で指定されたディレクトリに置かれていなければならない。説明文を読むと、これは HyperText Markup Language で書かれた文書を編集するためのモードであることが分る。このモードは M-x html-helper-mode とタイプすることでインタラクティブに呼び出すことが出来る。(この場合、オートロード式の説明文には正規の説明文をそのまま複写する必要がある。正規の説明文はまだロードされていないので、まだ読めないわけである。)

詳しくは、次を参照。 Section “Autoload” in *The GNU Emacs Lisp Reference Manual*.

16.10 ちょっとした拡張: line-to-top-of-window

次に、ポイントのある行をウィンドウの最上段に移動する簡単な Emacs の拡張を書いておく。これを使うとテキストを簡単に見ることが出来るので、私は普段よく利用している。

次のコードを別ファイルにして、それを ‘.emacs’ でロードしても良いし、コードを直接 ‘.emacs’ の中に含めてしまうことも出来る。

以下が定義である。

```
;;; Line to top of window;
;;; replace three keystroke sequence  C-u 0 C-l
(defun line-to-top-of-window ()
  "Move the line point is on to top of window."
  (interactive)
  (recenter 0))
```

次にキーバインディングである。

大抵の Emacs version 18 用の ‘.emacs’ ファイルは、そのまま version 19 でも動くのだが、やはり幾つかの違いがある。(勿論 Emacs 19 での新しい機能もある。)

Version 19 の Emacs では、ファンクションキーを ‘[f6]’ のように書くことが出来る。 Version 18 ではファンクションキーを押した時にキーボードから送られるキーストロークを指定しなければならなかった。例えば Zenith 29 キーボードでは、6 番目のファンクションキーを押すと ESC P が送られるし、 Ann Aubor Ambassador キーボードでは ESC O F が送られる。これらのキーストロークは各々 ‘\eP’ 及び ‘\eOF’ のように書かれることになる。

私は version 18 の ‘.emacs’ では line-to-top-of-window を端末のタイプに依存するキーにバインドしている。

```
(defun z29-key-bindings ()
  "Function keybindings for Z29 terminal."
  ;; ...
  (global-set-key "\eP" 'line-to-top-of-window))

(defun aaa-key-bindings ()
  "Function keybindings for Ann Arbor Ambassador"
  ;; ...
  (global-set-key "\eOF" 'line-to-top-of-window))
```

(ファンクションキーを押した時にどういうキーストロークが送られるかは、`C-h l` (`view-lossage`) とタイプすることで知ることが出来る。これは、最後にタイプした 100 のキーストロークを表示してくれる。)

キーバインディングを指定した後は、端末タイプに従って、そのどちらかのキーバインディングを選択するＳ式を評価する。が、その前に以前定義されたデフォルトのキーバインディングを削除する。そうしないと、定義がぶつかった場合に‘`.emacs`’でのキーバインディングが上書きされる。

```
;;; Turn Off Predefined Terminal Keybindings

; The following turns off the predefined
; terminal-specific keybindings such as the
; vt100 keybindings in lisp/term/vt100.el.
; If there are no predefined terminal
; keybindings, or if you like them,
; comment this out.

(setq term-file-prefix nil)
```

以下が選択のためのＳ式である。

```
(let ((term (getenv "TERM")))
  (cond
   ((equal term "z29") (z29-key-bindings))
   ((equal term "aaa") (aaa-key-bindings))
   (t (message
       "No binding for terminal type %s."
       term))))
```

Emacs version 19 では、関数キーは (マウスボタンイベントや非アスキー文字と同様) 引用符無しの角括弧でくくって書くことが出来る。私は次のようにして `line-to-top-of-window` をファンクションキー `F6` にバインドしている。

```
(global-set-key [f6] 'line-to-top-of-window)
```

こっちの方がずっと単純だ！

より詳しいことについては、次を見て欲しい。 Section “Rebinding Keys in Your Init File” in *The GNU Emacs Manual*.

Emacs 18 と Emacs 19 の両方を走らせている場合、次のような条件分岐を使って、どちらのコードを使うかを選択することが出来る。

```
(if (string=
     (int-to-string 18)
     (substring (emacs-version) 10 12))
    ;; evaluate version 18 code
    (progn
      ... )
  ;; else evaluate version 19 code
  ...
```

16.11 キーマップ

Emacs はどのキーをどのコマンドにバインドするかを *keymaps* を使って記録する。 C mode や Text mode のような特定のモードは、それ自身のキーマップを持っている。モード独自のキーマップは、全てのバッファで共有される global map を上書きする。

global-set-key 関数は global keymap のキーをバインドしたり再バインドしたりする。例えば次のＳ式は、キー C-c C-l を関数 line-to-top-of-window にバインドする。

```
(global-set-key "\C-c\C-l" 'line-to-top-of-window)
```

モード独自のキーマップは define-key 関数を使ってバインドされる。これは、キーやコマンドと同時に特定のキーマップも引数に取る。例えば私の ‘.emacs’ ファイルでは、 texinfo-insert-@group コマンドを C-c C-c g にバインドするために、次のようなＳ式が書かれている。

```
(define-key texinfo-mode-map "\C-c\C-cg"
  'texinfo-insert-@group)
```

texinfo-insert-@group 関数そのものは、 Texinfo ファイルに ‘@group’ をインサートするためのちょっとした Texinfo mode の拡張である。私はこのコマンドをいつも使うので、六つのキーストローク @ g r o u p よりも三つのキーストロークである C-c C-c g を使う方がずっと便利なのだ。(‘@group’ と対応する ‘@end group’ は中のテキストを一つのページに収めるためのコマンドである。この本の中の、沢山の複数行に渡る例が ‘@grouip ... @end group’ で囲まれている。)

以下が texinfo-insert-@group の関数定義である。

```
(defun texinfo-insert-@group ()
  "Insert the string @group in a Texinfo buffer."
  (interactive)
  (beginning-of-line)
  (insert "@group\n"))
```

(タイプ数を節約するためには、勿論、単語を挿入する関数を定義したりする代わりに Abbrev mode を使うことも出来る。しかし、キーストロークが他の Texinfo mode のキーバインディングと調和が取れているという意味で、私は上の方法の方が好みである。)

‘loaddefs.el’ を見れば ‘c-mode.le’ や ‘lisp-mode.el’ のような様々なモードライブラリと同時に沢山の *define-key* 式が書かれているのが分る。

キーマップについての詳細は次を参照のこと。Section “Customizing Key Bindings” in *The GNU Emacs Manual*, 及び Section “Keymaps” in *The GNU Emacs Lisp Reference Manual*.

16.12 X11 でのカラー表示

X Window system を使うと、 Emacs version 19 をカラーで使うことが出来る。(これまでに挙げた全ての例は、 Emacs version 18 でも Emacs version 19 でも動くはずである。が、ここの例は Emacs version 19 でのみ動作する。)

私はデフォルトのカラーが気に入らないので、自分独自の色を指定している。

私の指定は、大抵の X の初期化ファイルにあるものである。やったことを備忘録として自分の ‘.emacs’ に記述している。

```
;; I use TWM for window manager;
;; my ~/.xsession file specifies:
;;    xsetroot -solid navyblue -fg white
```

実際は、 X window のルートは Emacs の一部などでは全くない。が、ともかく書いておくことにした。

```
;; My ~/.Xresources file specifies:
;;     XTerm*Background:    sky blue
;;     XTerm*Foreground:    white
;;     emacs*geometry:      =80x40+100+0
```

```
;     emacs*background:    blue
;     emacs*foreground:    grey97
;     emacs*cursorColor:   white
;     emacs*pointerColor:  white
```

次からが、‘.emacs’で値を設定しているＳ式の部分である。

```
;;; Set highlighting colors for isearch and drag
(set-face-foreground 'highlight "white" )
(set-face-background 'highlight "slate blue")
(set-face-background 'region    "slate blue")
(set-face-background
 'secondary-selection "turquoise")

;; Set calendar highlighting colors
(setq calendar-load-hook
      '(lambda ()
         (set-face-foreground 'diary-face   "skyblue")
         (set-face-background 'holiday-face "slate blue")
         (set-face-foreground 'holiday-face "white")))
```

色々な色合いの青が、スクリーンのちらつきから私の眼を守り、そして癒してくれる。

16.13 V19 での雑多な事柄

以下は、 Emacs version 19 用の細かい設定である。

- 自動的に必要なだけミニバッファの大きさを変更する。

```
(resize-minibuffer-mode 1)
(setq resize-minibuffer-mode t)
```

- 検索文字をハイライト表示する。

```
(setq search-highlight t)
```

- 全てのフレームでメニューバーを表示し、マウスをそのフレーム上に移動したときに上に来るようにする。

```
(setq default-frame-alist
      '((menu-bar-lines . 1)
        (auto-lower . t)
        (auto-raise . t)))
```

-

マウスカーソルの色と形の設定。

```
; Cursor shapes are defined in
; '/usr/include/X11/cursorfont.h';
; for example, the 'target' cursor is number 128;
; the 'top_left_arrow' cursor is number 132.

(let ((mpointer (x-get-resource "*mpointer"
                                "*emacs*mpointer")))
  ;; If you have not set your mouse pointer
  ;;     then sent it, otherwise leave as is:
  (if (eq mpointer nil)
      (setq mpointer "132")) ; top_left_arrow
  (setq x-pointer-shape (string-to-int mpointer))
  (set-mouse-color "white"))
```

16.14 モード行の修正

最後に、モード行の修正について述べる。これが変更出来るようになっているのは本当に嬉しい。

私は時々ネットワーク上で仕事をするので、通常モード行の左側に書かれている 'Emacs: ' をシステムの名前で置き換えている — そうしないと、私が現在どのマシンの上で仕事をしているか忘れてしまうからである。更に、どのディレクトリにいたかを忘れないようにデフォルトのディレクトリのリストも書いている。また、'Line' が溢れるために、line point を on にした。以上の設定をする '.emacs' の記述は以下の通りである。

```
(setq mode-line-system-identification
  (substring (system-name) 0
             (string-match "\\..+" (system-name))))
(setq default-mode-line-format
      (list ""
            'mode-line-modified
            "<"
            'mode-line-system-identification
            "> "
            "%14b"
            " "
            'default-directory
            " "
            "%[("
            'mode-name
            'minor-mode-alist
            "%n"
            'mode-line-process
            ")%]--"
             "Line %l--"
            '(-3 . "%P")
            "-%-"))
;; Start with new default.
(setq mode-line-format default-mode-line-format)
```

default のモード行のフォーマットは Info のような様々なモードが上書きすることを許すようにした。リストの中の多くの要素は、名前から役目が推測出来るようになっている。例えば `mode-line-modified` はバッファが修正された場合、それを教えてくれるようにする変数だし、 `mode-name`はモードの名前を教えてくれる、等々。

'"%14b"' は現在のバッファの名前を (お馴染みの `buffer-name` 関数を使って) 表示する。'14' は表示する文字数の最大の指定である。それよりも文字数が少ない場合は空白文字で埋めてくれる。'%[' と '%]' は各々の再帰編集レベルに応じて角括弧の組を表示するものである。'%n' はナローイングをしている時に 'Narrow' と表示してくれる。'%P' はバッファ全体でウィンドウの最下段より上にある部分が何パーセントあるか、もしくは 'Top' か 'Bottom' かを表示してくれる。(小文字の 'p' にすると、ウィンドウの最上段より上の部分がどのくらいかを表示する。) '%-' は行の残りをダッシュ (訳註 : '-' のこと) で埋めてくれる。

Emacs version 19.29 以降では、 `frame-title-format` を使って Emacs のフレームにタイトルを表示することが出来る。この変数は `mode-line-format` と同じ構造をしている。

モード行のフォーマットについては、次に説明がある。 Section "Mode Line Format" in *The GNU Emacs Lisp Reference Manual*.

(訳註 : Nemacs や Mule では日本語コードや多国語コードの状態を表示するために拡張が加えられている。`default-mode-line-format` を一度評価して値を調べ、それを元に修正するとよいだろう。)

もう一度「Emacs を好きになるために Emacs を好きになる必要はない。」という言葉を思い出して欲しい — あなた自身の Emacs はデフォルトの Emacs とは異なる色や異なるコマンド、そして異なるキーマップを持つことが出来るのだ。

一方で、もし「箱から取り出したままの」カスタマイズされていない素の Emacs を起動したい場合は、次のようにタイプしよう。

```
emacs -q
```

こうすると、Emacs はあなたの初期化ファイル‘~/.emacs’をロードしないで起動する。これが素のままのデフォルトの Emacs である。

17 デバッグ

GNU Emacs は二つのデバッガを持っている。 debug と edebug である。最初のものは、 Emacs の内部に組込まれていて、常に使える状態にある。二番目のものは拡張機能として与えられているもので、 version 19 では標準の配布の中に含まれている。

これらのデバッガは共に、以下の所に非常に詳しく説明されている。 Section "Debugging Lisp Program" in *The GNU Emacs Lisp Reference Manual.* この章では、各々の簡単な例を見ていくことにする。

17.1 debug

例えばあなたが、 1 から与えられた数までの和を計算するための関数の定義を書いたとしよう。 (これは以前に説明した triangle 関数である。次の所で詳しく議論されている。 Section 11.1.4, "減少するカウンタを使ったループ", p. 88.)

しかしながら、あなたの関数にはバグがあったとしよう。例えば '1-' と書くべきところを '1=' と書いてしまったとする。この間違いをしたプログラムは以下の通りである。

```
(defun triangle-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (setq number (1= number)))      ; ここでエラー
    total))
```

もし、この文章を Info で読んでいるなら、この関数をいつものように評価することが出来る。そうすると、エコー領域には triangle-bugged と表示されるはずだ。

さて、 triangle-bugged を引数 4 とともに評価してみよう。

```
(triangle-bugged 4)
```

すると、次のようなエラーメッセージを受け取るはずだ。

```
Symbol's function definition is void: 1=
```

実際は、このような単純なバグなら、エラーメッセージだけでどう修正すればよいか分るだろう。しかし、分らない場合はどうすればよいか？

debug-on-error の値を t とすることで、デバッグを開始することが出来る。

```
(setq debug-on-error t)
```

こうしておくと、 Emacs は次にエラーに遭遇した時にデバッグに入る。

逆に debug-on-error を nil にすると、デバッグには入らない。

```
(setq debug-on-error nil)
```

では、 debug-on-error を t にして、次を評価してみよう。

```
(triangle-bugged 4)
```

今度は Emacs は '*Backtrace*' と呼ばれるバッファを生成したはずだ。これは以下のようなものだ。

```
---------- Buffer: *Backtrace* ----------
Signalling: (void-function 1=)
  (1= number))
  (setq number (1= number)))
  (while (> number 0) (setq total (+ total number))
        (setq number (1= number))))
  (let ((total 0)) (while (> number 0) (setq total ...)
        (setq number ...)) total))
  triangle-bugged(4)
  eval((triangle-bugged 4))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

(上はちょっとばかり修正してある。実際はデバッガは長い行を折り返したりはしない。)

'*Backtrace*' を読む時は、下から上に向かって読んでいく。そうすると、Emacs がどのようにしてエラーを出すに至ったかが分る。今の場合、Emacs は C-x C-e (eval-last-sexp) をインタラクティブに呼び出し、その結果として triangle-bugged 式を評価している。上の各々の行から Lisp インタプリタが次に何を評価していったかが分る。

上から三行目は

```
(setq number (1= number))
```

である。Emacs はこのＳ式を評価しようとした。そのために、内部のＳ式を評価しようとした。それが上から二行目の式である。

```
(1= number)
```

エラーが生じたのは、ここである。一行目にはそう書かれている。

```
Signalling: (void-function 1=)
```

そこでエラーを修正し、再び関数定義を評価して、もう一度テストしてみることになる。

もし、Info でこれを読んでいるなら、ここで下の式を評価して debug-on-error を nil に戻しておこう。

```
(setq debug-on-error nil)
```

17.2 debug-on-entry

ある関数に対して debug を実行する二番目の方法は、関数を呼び出す時にデバッグに入る方法である。それには debug-on-entry を使う。

まず、次のようにタイプしてみよう。

```
M-x debug-on-entry RET triangle-bugged RET
```

そして、次を評価する。

```
(triangle-bugged 5)
```

Emacs は '*Backtrace*' バッファを作成し、triangle-bugged 関数の評価の開始を知らせてくる。

```
---------- Buffer: *Backtrace* ----------
Entering:
* triangle-bugged(5)
  eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

‘*Backtrace*’ バッファの中で、 d とタイプしてみよう。すると、 Emacs は triangle-bugged の最初のＳ式を評価する。このときバッファは次のようになる。

```
---------- Buffer: *Backtrace* ----------
Beginning evaluation of function call form:
* (let ((total 0)) (while (> number 0) (setq total ...)
        (setq number ...)) total))
  triangle-bugged(5)
* eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

ここで、ゆっくりと、更に 8 回 d をタイプしてみよう。 d とタイプするごとに、 Emacs は関数定義の中の別のＳ式を評価していく。結果として、このバッファは次のようになる。

```
---------- Buffer: *Backtrace* ----------
Beginning evaluation of function call form:
* (setq number (1= number)))
* (while (> number 0) (setq total (+ total number))
        (setq number (1= number))))
* (let ((total 0)) (while (> number 0)
        (setq total ...) (setq number ...)) total))
  triangle-bugged(5)
* eval((triangle-bugged 5))
  eval-last-sexp(nil)
* call-interactively(eval-last-sexp)
---------- Buffer: *Backtrace* ----------
```

最後に、更に 2 回 d をタイプすると Emacs はエラーの箇所まで辿り着く。その結果 ‘*Backtrace*’ の上から二行は次のようになる。

```
---------- Buffer: *Backtrace* ----------
Signalling: (void-function 1=)
* (1= number))
...
---------- Buffer: *Backtrace* ----------
```

d とタイプすることで、関数を一つずつ実行していくことが出来る。

‘*Backtrace*’ バッファを抜けるには q とタイプする。こうするとトレースは抜けるが、 debug-on-entry はキャンセルしない。

debug-on-entry そのものを無効にするには、 cancel-debug-on-entry に呼び出して関数名を入力すればよい。次のような感じだ。

```
M-x cancel-debug-on-entry RET triangle-bugged RET
```

(もしこれを Info で読んでいるなら、ここで debug-on-entry を無効にしておこう。)

17.3 debug-on-quit と (debug)

debug-on-error をセットしたり debug-on-entry を呼び出したりする以外にも二つ程、 debug を開始する方法がある。

一つは変数 debug-on-quit を t にセットすることで、こうすると C-g (keyboard-quit) をタイプすることで常に debugを開始することが出来る。これは無限ループをデバッグする際に有効である。

もう一つは、コードの中のデバッグを開始したい所に (debug) を呼び出す行を挿入する方法である。例えば次のような感じである。

```
(defun triangle-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive."
  (let ((total 0))
    (while (> number 0)
      (setq total (+ total number))
      (debug)                          ; デバッガ起動
      (setq number (1= number)))       ; ここでエラー
    total))
```

debug 関数は次の中で詳しく説明されている。 Section “The Lisp Debugger” in *The GNU Emacs Lisp Reference Manual*.

17.4 ソースレベルのデバッガ edebug

Edebug は通常デバッグをしているコードのソースを表示してくれる。そして、現在どの行を実行しているかを左側の矢印で示してくれる。

関数を一行ごとに実行することも出来るし、実行を止める *breakpoint*の所まで一気に走らせることも出来る。

Edebug については次に所に説明がある。 Section “Edebug” in *The GNU Emacs Lisp Reference Manual*.

次にバグを含んだ triangle-recursively の関数定義を挙げておく。これについて復習したい場合には、次を参照。 Section 11.2.2, “カウンタの代わりに再帰を使う”, p. 91. この例は、下に説明されているように defun の左のインデント無しで表示されている。

```
(defun triangle-recursively-bugged (number)
  "Return sum of numbers 1 through NUMBER inclusive.
Uses recursion."
  (if (= number 1)
      1
    (+ number
       (triangle-recursively-bugged
        (1= number)))))                ; ここでエラー
```

普通は、この関数定義をインストールするにはカーソルを関数の最後の閉じ括弧の後で C-x C-e (eval-last-sexp) とタイプするか、関数定義内にカーソルを置いて C-M-x (eval-defun) とタイプする。(デフォルトでは eval-defun コマンドは Emacs Lisp mode か Lisp Interaction mode でしか動作しない。)

しかしながら、 Edebug を使ってこの関数をデバッグする際には、まず最初に別の関数を使って、コードの 膳立て (*instrument*) をする必要がある。 Emacs version 19 では関数定義内で次のようにタイプする。

```
M-x edebug-defun RET
```

こうすることで、もし Edebug が Emacs にインストールされていなければ自動的にインストールし、関数を適切に膳立てする。(Edebug をロードした後は、 Edebug の標準的なキーバインディングが使えるようになる。例えば edebug-defun を C-u C-M-x (eval-defun の前置引数付きの実行) で実行出来る。)

Emacs version 18 では自分で Edebug をロードしなければならない。それには '.emacs' に適切な load コマンドを書いておけばよい。

もしこれを Info の中で読んでいるなら、ここで上の triangle-recursively-bugged 関数の膳立てをする。その際に関数定義の defun の行がインデントされていると、 edebug-defun が定義の範囲を特定出来なくなってしまうので、上に挙げた例では今まで入れていた defun の左のスペースを除いてある。

関数の膳立てをした後、カーソルを次のＳ式のすぐ後に持っていって C-x C-e (eval-last-sexp) とタイプしよう。

```
(triangle-recursively-bugged 3)
```

すると triangle-recursively-bugged のソースに戻ってカーソルがこの関数内の if の行の最初に移動する。また、'=>' という矢印がその行の左側に表示される。この矢印は関数が実行されている場所を指している。

```
=>⋆(if (= number 1)
```

例の中で、星印 '⋆' はポイントを表している (これは Info では '-!-' と表示される。)。

さて、ここで SPC を押すと、ポイントは次に実行されるＳ式に移動する。従って、この行が次のようになる。

```
=>(if ⋆(= number 1)
```

続けて SPC を押していくと、ポイントはＳ式からＳ式へと移動していく。同時にＳ式が値を返すごとにその値がエコー領域に表示される。例えばポイントが number の所を通過すると、次のように表示されるはずだ。

```
Result: 3 = C-c
```

これは number の値が 3、つまり三番目の ASCII コードである ASCII CTL-C であることを意味している。 (C はアルファベットの三番目の文字である。) (訳註：暇な人は（char-to-string 3）等を評価してみよう。)

こうしてエラーのある行まで移動することが出来る。これを表示する直前には、この行は次のように表示されている。

```
=>        ⋆(1= number)))))                ; ここでエラー
```

ここでもう一度 SPC を押すと次のようなエラーメッセージが表示される。

```
Symbol's function definition is void: 1=
```

これがバグである。

Edebug を終了するには 'q' を押せばよい。

この関数定義の膳立てを解くには、 単に膳立てしないようなコマンドでその関数を評価しなおせばよい。例えば、関数定義の最後の閉じ括弧の後で C-x C-e とタイプするだけでよい。

Edebug は単に関数を一つずつ実行していくだけではなく他にも沢山のことをしてくれる。自動的に関数を実行していってエラーが起きた所で止まるように設定することも出来るし、特定の場所で止まるようにすることも可能である。様々なＳ式の値の変化を表示させることも出来るし、ある関数が何回実行されたかも調べることも出来る。その他にもいろいろなことが出来る。

Edebug については次で説明されている。 Section "Edebug" in *The GNU Emacs Lisp Reference Manual*.

17.5 デバッグについての練習問題

- count-words-region 関数をインストールし、それを呼び出した時に内部デバッガが起動するようにしなさい。二つの単語を含むような region の上でこの関数を実行しなさい。 d をかなりの回数押す必要があるだろう。あなたの システムではこのコマンドが終了した時点で何かフックが呼ばれますか？ (フックについての情報は次を参照。 Section "Command Loop Overview" in *The GNU Emacs Lisp Reference Manual*.)

- `count-words-region` を ‘`*scratch*`’ バッファにコピーして、必要なら `defun` の先頭のスペースを取り除いてから、この関数を Edebug のために膳立てしなさい。そして、この関数を 1 ステップずつ実行しなさい。この場合、必ずしも関数にはバグがある必要はない。バグがあった方がよければバグを設定してもよい。もしこの関数にバグがなければ、問題なく最後まで実行出来るはずだ。
- Edebug を実行している間に、`?` とタイプして全ての Edebug のコマンドのリストを見なさい。(`global-edebug-prefix` は普通 `C-x X`、つまり `CTL-x` に続く大文字の `X` になっている。Edebug のデバッグバッファの外でのコマンドにはこの前置キーを使うようにする。)
- Edebug のデバッギングバッファで `p` (`edebug-bounce-point`) コマンドを使ってリージョンの何処で `count-words-region` が動作しているかを見なさい。
- ポイントを関数の下の方に持って行き、`h` (`edebug-goto-here`) コマンドをタイプすることでその場所にジャンプしなさい。
- `t` (`edebug-trace-mode`) コマンドを使って、Edebug が自分自身で関数の上を走るようにしなさい。同様に大文字の `T` を使って `edebug-Trace-fase-mode` に入りなさい。
- ブレークポイントを設定し、その位置まで Edebug を Trace mode で走らせなさい。

18 まとめ

この入門書はこれで終わりである。以上であなたは Emacs Lisp でのプログラミングについて十分学んだので、値をセットしたり、簡単な ‘.emacs’ ファイルを自分自身や友人のために書いたり、Emacs にちょっとしたカスタマイズや拡張を施すことが出来るようになっているはずである。

もっとも、これは一つの段階に過ぎない。望むなら、もっと先に進み、自分自身で学んでいくことが出来る。

ここからは GNU Emacs のソースや *The GNU Emacs Lisp Reference Manual* などが参考になるだろう。

Emacs Lisp のソースを見ることは一種の探検である。ソースを読んで見慣れない関数やＳ式に出逢った場合には、それが何であるかを解読したり調べたりしなければならない。

まずリファレンスマニュアルを開こう。これは徹底して書かれた、完全な、しかも読みやすい Emacs Lisp の説明書である。これは熟練者のためだけではなく、あなたが現在知っているような知識しかない人向けにも書かれている。(*Reference Manual* は標準的な GNU Emacs の配布に含まれている。この入門書と同様、これも Texinfo のソースファイルとして配布されている。従って、オンラインでも読めるし、タイプセットして印刷された本としても読むことも出来る。)

また、他の GNU Emacs 附属のオンラインヘルプも読んでみよう。全ての関数には説明文字列がついているし、`find-tags` というソースを見つけてくれるプログラムもある。

例として私がソースを探検する方法を紹介しよう。もう随分と昔のことになるが、最初に私は、その名前に魅かれて ‘`simple.el`’ というファイルを見た。往々にしてそんなものだが、‘`simple.el`’ の中の幾つかの関数は複雑だった。少なくとも、ぱっと見には複雑そうに見えた。例えば一番最初の `open-line` 関数からして、いかにも複雑という感じだ。

あなたは、この関数を `forward-sentence` 関数の時のように、ゆっくりと眺めたいかもしれない。(Section 12.3, “`forward-sentence`”, p. 96.) 或いはこの関数は飛ばして他のもっと簡単そうな `split-line` 等の関数を見たいかもしれない。これらの関数を全て読む必要はない。`count-words-in-defun` を使うと、`split-line` 関数は 27 個の単語とシンボルを含んでいることが分る。

`split-line` は、その短さにも関わらず、まだ我々が学んでいない四つのＳ式を含んでいる。`skip-chars-forward`、`indent-to`、`insert`、そして ‘`?\n`’ である。

例えば `insert` 関数を考えてみよう。(これについては復習のセクションである次の所でちょこっと触れられている。) Chapter 12, “正規表現の検索”, p. 95. Emacs 上で `C-h f` (`describe-function`) に続けて `insert`とタイプすれば、この関数についてより詳しいことを知ることが出来る。これは関数の説明文字列を表示してくれる関数である。ソースを見る場合は `find-tag` を使う。これは `M-.` にバインドされている。(もっとも今の場合にはあまり役に立たない。この関数は Lisp ではなく C で書かれたプリミティブだからだ。) 最後に、この関数についてのリファレンスマニュアルの記述は次のようにして探せる。まず Info を起動する。そして `i` (`Info-index`) に続けてこの関数の名前をタイプする。もしくは印刷されたマニュアルのインデックスから `insert` を探してもよい。

同様に、‘`?\n`’ の意味も調べることが出来る。`Info-index` で ‘`?\n`’ を探してみるとよい。やってみると分るがこれは失敗する。しかし諦めてはいけない。インデックスで ‘`?`’ 無しの ‘`\n`’ を探してみれば、マニュアルの中で関係するセクションを見つけることが出来る。(‘`?\n`’ が改行文字を表していることは次の所に出ている。Section “Character Type” in *The GNU Emacs Lisp Reference Manual*.)

`skip-chars-forward` と `indent-to` で何が行われているかは推察することが出来るだろう。勿論詳しく見ることだって出来る。(ついでだが、`describe-function` 関数そのものは ‘`help.el`’ にある。これは、長い割には解読しやすい関数の一つである。これの定義を見れば、どうやって標準の文字コードを使わずに `interactive` 式をカスタマイズ出来るかとか、どうやってテンポラリバッファを生成するかということが分る。

他の面白いソースとしては、‘`paragraphs.el`’ や ‘`loaddefs.el`’ それに ‘`loadup.el`’ などがあげられる。‘`paragraphs.el`’ ファイルには、長い関数だけでなく短くて簡単に理解出来る関数も含まれている。‘`loaddefs.el`’ ファイルには沢山の標準オートロード式や沢山のキーマップが含まれている。‘`loadup.el`’ は Emacs の標準部分をロードするファイルである。これを見れば、Emacs がどういうふうに作成されるかがよく

分る。(Emacs の作成については次を参照。 Section “Building Emacs” in *The GNU Emacs Lisp Reference Manual.*)

前にも言ったように、これまでに Emacs の部品についてはいくらか学習してきた。しかし、これは大切なことなのだが、プログラミングの多くの部分については触れることは出来ていない部分が多い。私は情報のソートについては既に定義された `sort` 関数を使うこと以外全く触れることが出来なかった。また、変数やリストを使うこと以外の事柄については、例えば情報を貯めておくにはどうしたらいいかといったことにも触れられなかった。プログラムを書くプログラムについても同様だ。これらの話題については、他の別の種類の本で本書とは異なる学習形態で学ぶべきであろう。

ともかく、以上で GNU Emacs を使って沢山の実際の仕事をする準備が整った。出発点に辿り着いたわけである。これは始まりの終わりなのだ。

Appendix A 関数 the-the

文章を書いていると、時々単語を二重に書いてしまうことがある — 例えば “you you” (訳註：原文では実際にこの文の先頭で you が二重になっている。) と書いてしまったりする。個人的には “the” をよく二重に書いてしまう。そこで、二重になっている単語を見つける関数のことを the-the と呼ぶことにする。

まず最初の段階として、取り敢えず次の正規表現を使えば、このような二重の単語を表現出来そうなことが分る。

```
\\(\\w+[ \t\n]+\\)\\1
```

この正規表現 (の前半) は一つ以上の単語構成文字に続いて一つ以上のスペース、タブないしは改行がくるものにマッチする。しかし、これでは二行に渡るものは見つけられない。というのも、最初の単語の終わりに改行が来た場合に、次の行の同じ単語の後にスペースが来ればマッチしないからである。(正規表現についての詳細は、Chapter 12, “正規表現の検索”, p. 95, や Section “Syntax of Regular Expressions” in *The GNU Emacs Manual*, それに Section “Regular Expressions” in *The GNU Emacs Lisp Reference Manual*, を参照。)

単に単語の部分だけを検索すればよいのではと考えるかもしれないが、これでは ‘with the’ の中の ‘th’ のようなものまで引っかけてしまう。

上とは別の正規表現検索で、単語構成文字に続いて非単語構成文字が続き、更に最初の単語が来るものにマッチするものがある。次の式で ‘\\w+’ は一つ以上の単語構成文字にマッチし、‘\\W*’ は零個以上の非単語構成文字にマッチする。

```
\\(\\(\\w+\\)\\W*\\)\\1
```

が、これも役に立たない。

次に私が使っているものを挙げる。これは完全ではないが、十分使いものになる。‘\\b’ は単語の始まりもしくは終わりの空文字列にマッチし、‘[^@ \n\t]+’ は一つ以上の @ マーク、空白、改行、もしくはタブのいずれでもない文字の連続にマッチする。

```
\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b
```

もっと複雑な式を書くことも出来るが、私自身はこの式で十分であると分ったので、これを使っている。

以下が the-the 関数である。これを使いやすいキーバインディングと共に ‘.emacs’ ファイルに記述している。

```
(defun the-the ()
  "Search forward for for a duplicated word."
  (interactive)
  (message "Searching for for duplicated words ...")
  (push-mark)
  ;; This regexp is not perfect
  ;; but is fairly good over all:
  (if (re-search-forward
       "\\b\\([^@ \n\t]+\\)[ \n\t]+\\1\\b" nil 'move)
      (message "Found duplicated word.")
    (message "End of buffer")))

;; Bind ‘the-the’ to  C-c \
(global-set-key "\C-c\\" 'the-the)
```

以下がテスト用の文章である。

```
one two two three four five
five six seven
```

関数定義の中の正規表現を上に挙げた他の正規表現で置き換えて、このリストの上で試してみることも出来る。

Appendix B Kill リングの扱い

kill リングは rotate-yank-pointer の働きによってリングの形態に変換されたリストである。 yank 及び yank-pop コマンドは rotate-yank-pointer 関数を利用している。この Appendix では、 yank や yank-pop と共に、 rotate-yank-pointer 関数も説明することにする。

B.1 関数 rotate-yank-pointer

rotate-yank-pointer 関数は kill-ring-yank-pointer が指す kill リングの要素を変更する。例えば、今まで kill-ring-yank-pointer が二番目の要素を指していたなら、それを三番目の要素に変更する。

これが rotate-yank-pointer のコードである。

```
(defun rotate-yank-pointer (arg)
  "Rotate the yanking point in the kill ring."
  (interactive "p")
  (let ((length (length kill-ring)))

    (if (zerop length)

        ;; then-part
        (error "Kill ring is empty")

      ;; else-part
      (setq kill-ring-yank-pointer
            (nthcdr (% (+ arg
                          (- length
                             (length
                              kill-ring-yank-pointer)))
                       length)
                    kill-ring)))))
```

この関数は複雑そうに見えるが、いつも通り幾つかの部分に分けて考えると、すっきりと理解出来る。最初に骨組みを見てみよう。

```
(defun rotate-yank-pointer (arg)
  "Rotate the yanking point in the kill ring."
  (interactive "p")
  (let varlist
    body...)
```

この関数は arg という引数を一つ取る。また、簡単な説明文字列がついており、小文字の 'p' によりインタラクティブ宣言されている。これは、引数が処理された前置引数であり、数として関数に渡されなければならないことを意味している。

関数定義の本体は let 式であり、これは変数リスト *varlist* と本体部分からなる。変数は length と呼ばれ、kill リングの要素の数にバインドされる。この数は、 length という関数によって調べられる。(この関数が、変数 length と同じ名前であることに注意しよう。にもかかわらず一方では関数名として使われ、もう片方は変数名として使われている。この二つはきちんと区別されている。同様に、英語を話す人々は 'ship'という単語の意味を "I must ship this package immediately." という時と "I must get aboard the ship immediately." という時で区別している。)

関数 length はリストの中の要素の数を教えてくれるものである。従って、(length kill-ring) は kill リングの中の要素の数を返すことになる。

B.1.1 rotate-yank-pointer の本体

rotate-yank-pointer は let 式であり、 let の本体は、 if 式である。

この if 式は、 kill リングに何か要素があるかどうかを見るためのものである。もし kill リングが空なら error 関数がこの関数の評価を止めて、エコー領域にメッセージを表示する。一方、もし kill リングが空でなかった場合は、この関数の残りの仕事が実行される。

以下が、この if 式の if-part と then-part である。

```
(if (zerop length)                      ; if-part
    (error "Kill ring is empty")        ; then-part
  ...
```

もし kill リングに何もなければその長さは零であるから、ユーザに対して 'Kill ring is empty' というエラーメッセージが表示される。上の if 式では、値が零かどうかの判定に zerop を使っている。これはテスト対象が零であれば真を返す関数である。もし zerop テストが真であれば、 if 式の then-part が評価される。 then-part は関数 error から始まるリストである。この関数は message 関数と似ていて (Section 1.8.5, "関数 message", p. 10, 参照)、一行メッセージをエコー領域に表示するのだが、それだけではなくて error が埋め込まれている関数の評価を停止する働きも持っている。今の場合、もし kill リングの長さが零であれば関数の残りの部分は評価されないということになる。

(個人的な意見では、少なくとも人間にとっては、この関数に 'error' という用語を使うのはちょっと誤解を招きやすい気がする。例えば 'cancel' という用語の方がよりしっくりくる。勿論厳密に言えば長さを持たないリストの要素を指すことは出来ないし、ましてや指す位置を巡回させることなど出来ない。だから、計算機の視点に立てば 'error' という言葉は全く正しい。しかし人間の立場からすると、こういったことをしたいのは kill リングが空かそうでないかを見る場合だけである。これは一種の調査でしかない。

人間の視点から見ると、調査や発見という行為は必ずしもエラーではない。従って、例え計算機の世界の奥底にいる場合でもエラーと呼ぶべきではない。もしそうだとしたら、 Emacs のコードは、正しい態度で自らの環境を調査している人がエラーを犯していると言っていることになってしまうからだ。これではまずい。いくら、計算機の方がエラーが起きた場合と同じステップを取ることになるとはいっても、 'cancel' の方がより正しい語感を持っているとは言えよう。)

if 式の else-part

if 式の else-part は kill リングが空でない場合の kill-ring-yank-pointer の設定に費やされている。コードは以下の通りである。

```
(setq kill-ring-yank-pointer
      (nthcdr (% (+ arg
                    (- length
                       (length kill-ring-yank-pointer)))
                 length)
              kill-ring)))))
```

これにはちょっと説明が必要だろう。まず kill-ring-yank-pointer が kill リングの何番目かの CDR に設定されるべきであるというのはいいだろう。これには以前説明した nthcdr 関数を使う。 (Section 8.5, "copy-region-as-kill", p. 71, 参照。) しかし、実際にはどうしたらいいのだろうか。

細かいコードを見る前に、まず rotate-yank-pointer の目的を考えてみよう。

rotate-yank-pointer 関数は、 kill-ring-yank-pointer が指す場所を変更する。もし kill-ring-yank-pointer がリストの最初の要素を指している状態から出発した場合、 rotate-yank-pointer を呼び出すと、これが二番目の要素を指すことになる。そして更に rotate-yank-pointer を呼び出すと、三番目に移る。 (また、 rotate-yank-pointer に 1 より大きな引数を与えて呼び出すと、その数だけ指す位置が移動される。)

rotate-yank-pointer 関数は setq を使って kill-ring-yank-pointer が指す位置をリセットしている。もし kill-ring-yank-pointer が kill リングの最初の要素を指していたなら、単純な場合、rotate-yank-pointer 関数は二番目の要素を指さなければならない。別の言い方をすると、kill-ring-yank-pointer は kill リングの CDR に等しい値に再設定されなければならないのである。

即ち、

```
(setq kill-ring-yank-pointer
   ("some text" "a different piece of text" "yet more text"))

(setq kill-ring
   ("some text" "a different piece of text" "yet more text"))
```

という状況では、コードは

```
(setq kill-ring-yank-pointer (cdr kill-ring))
```

となるべきである。結果として、kill-ring-yank-pointer は次のようになっていなければならない。

```
kill-ring-yank-pointer
     ⇒ ("a different piece of text" "yet more text"))
```

実際の setq 式では nthcdr 関数が使われている。

以前見たように (Section 7.3, “nthcdr”, p. 60, 参照)、nthcdr 関数はリストの CDR を繰り返し取ることによって動作する。これは CDR の CDR の ... の CDR を取る、ということである。

次の二つのＳ式は同じことである。

```
(setq kill-ring-yank-pointer (cdr kill-ring))

(setq kill-ring-yank-pointer (nthcdr 1 kill-ring))
```

しかし rotate-yank-pointer 関数では、nthcdr の最初の引数の所に何やら算数が使われていて、複雑そうに見える。

```
(% (+ arg
      (- length
         (length kill-ring-yank-pointer)))
   length)
```

これを理解するには、いつもの通りまずネストの一番深い所にあるＳ式から順に見ていく必要がある。

もっとも深い所にあるＳ式は (length kill-ring-yank-pointer) である。これは、現在の kill-ring-yank-pointer の長さを調べるものだ。(kill-ring-yank-pointer は値がリストであるような変数の名前であることを思い出そう。)

この長さを測定するＳ式は、次のＳ式の中に埋め込まれている。

```
(- length (length kill-ring-yank-pointer))
```

このＳ式の中の最初の length は、この関数の最初の let 式の局所変数の宣言の部分で kill リングの長さに設定された変数である。(この length という変数は length-of-kill-ring と名付けた方が解りやすいのではないかという人も居るかもしれない。しかし、ここでやっているように小さい部分に切り分けたりせずに、関数全体を眺めてみれば、短いが故に面倒でないということが実感出来るだろう。)

従って、(- length (length kill-ring-yank-pointer)) という行で、kill リングの長さと kill-ring-yank-pointer という名前のリストの長さの差を知ることが出来る。

これが rotate-yank-pointer 関数の中にどう組込まれるているかを見るために、まず kill-ring-yank-pointer が kill リングの最初の要素を指している場合、つまり、kill リング全体である場合に、rotate-yank-pointer を引数 1 で呼び出すとどうなるかを分析してみることにしよう。

この場合は、変数 length と (length kill-ring-yank-pointer)の長さは同じである。というのは、変数 length の値は kill リングの長さであり、 kill-ring-yank-pointer は kill リング全体を指しているからである。結果として、

```
(- length (length kill-ring-yank-pointer))
```

の値は零になる。引数として 1 を与えているので、

```
(+ arg (- length (length kill-ring-yank-pointer)))
```

全体の値は 1 になる。

そうすると、 nthcdr の引数としては次の式の値が渡されることになる。

```
(% 1 length)
```

剰余関数 %

(% 1 length) を理解するには、 % が何かを知らねばならない。説明文字列によると (これは C-h f % RET とタイプすると見ることが出来る)、 % 関数は最初の引数を二番目の引数で割った余りを返す関数であることが分る。例えば 5 を 2 で割った余りは 1 である。 (2 は 5 の中に二つ含まれていて、残りが 1 ということである。)

あまり算数を知らない人の中には、ある数をその数よりも大きな数で割ることが出来て、しかも余りまで求まると聞くとびっくりする人も多い。上の例では 5 を 2 で割ったのだが、これを逆にして、 2 を 5 で割るとどうなるだろうか。もし分数を使えれば、答えは明らかに 2/5 もしくは 0.4 である。が、ここでは整数しか使えないので、これとは違う答えにならなければならない。明らかに 5 は 2 の中に零個しか含まれていない。が、残りはいくつだろう。答えを知るために、子供の頃から親しんでいる場合を考えてみよう。

- 5 を 5 で割ると、商は 1 で、余りは 0 であり、
- 6 を 5 で割ると、商は 1 で、余りは 1 であり、
- 7 を 5 で割ると、商は 1 で、余りは 2 である。
- 同様に、 10 を 5 で割ると、商は 2 で、余りは 0 であり、
- 11 を 5 で割ると、商は 2 で、余りは 1 であり、
- 12 を 5 で割ると、商は 1 で、余りは 2 である。

これと全く同じように考えれば、

- 0 を 5 で割ると、余りは 0 であり、
- 1 を 5 で割ると、余りは 1 であり、
- 2 を 5 で割ると、余りは 2 であり、

という具合になる。

従って、今回のコードでは、もし length の値が 5 であれば、

```
(% 1 5)
```

を評価した結果は 1 になる。 (これは、カーソルを上の式の最後に持っていって C-x C-e とタイプすれば確かめられる。ちゃんとエコー領域に 1 が表示されるはずだ。)

rotate-yank-pointer の中での % の利用

kill-ring-yank-pointer が kill リングの最初を指しており、かつ rotate-yank-pointer の引数として 1 が渡された場合、 % 式が返す値は 1 になる。

```
(- length (length kill-ring-yank-pointer))
     ⇒ 0
```

故に、

```
(+ arg (- length (length kill-ring-yank-pointer)))
     ⇒ 1
```

であり、結果として length の値に関係無く

```
(% (+ arg (- length (length kill-ring-yank-pointer)))
   length)
     ⇒ 1
```

となる。

以上から、この場合の setq kill-ring-yank-pointer 式は、

```
(setq kill-ring-yank-pointer (nthcdr 1 kill-ring))
```

のようになる。これを見れば、やっていることは一目瞭然だ。kill-ring-yank-pointer の指す位置を kill リングの最初の要素ではなく二番目の要素にセットするのである。

また、rotate-yank-pointer に渡す引数を 2 にすれば kill-ring-yank-pointer は (nthcdr 2 kill-ring) にセットされることも明らかだろう。他の引数を与えた場合も同様である。

同じように、もし killl-ring-yank-pointer が kill リングの二番目の要素からスタートした場合、その長さ length は kill リングの長さより 1 少ない値になり、また余りの計算は (% (+ arg 1) length) である。従って、引数 1 で rotate-yank-pointer を評価すると、kill-ring-yank-pointer は kill リングの二番目の要素から三番目の要素を指すように変更される。

最後の要素を指している場合

最後の問題は、kill-ring-yank-pointer が kill リングの最後の要素を指していた場合にどうなるかである。kill リングにはもう何も残っていないので、rotate-yank-pointer は何も指さない状態になるのだろうか。答えは No である。実際にはもっと都合の良い結果になる。kill-ring-yank-pointer は kill リングの最初の要素を指すようにセットされるのである。

コードを見てどのように動作するのかを見てみることにしよう。取り敢えず kill リングの長さは 5 とし、rotate-yank-pointer に渡す引数は 1 であるとする。kill-ring-yank-pointer が kill リングの最後の要素を指している場合、その長さは 1 である。コードは次のようになっている。

```
(% (+ arg (- length (length kill-ring-yank-pointer))) length)
```

上の式の中の変数を実際の数値で置き換えると、式は次のようになる。

```
(% (+ 1 (- 5 1)) 5)
```

このＳ式をもっとも内部の式から外側に向かって見ていこう。(- 5 1)の値は 4 である。(+ 1 4) という和は 5 になる。そして、5 を 5 で割った余りは 0 である。従って rotate-yank-pointer では、

```
(setq kill-ring-yank-pointer (nthcdr 0 kill-ring))
```

というＳ式を評価することになる。これは、kill-ring-yank-pointer を kill リングの最初の要素を指すようにセットするものである。

従って、rotate-yank-pointer を続けて呼び出すと、kill-ring-yank-pointer を kill リングの最初の値から順に移動していき、最後の要素まで来たらまた最初に戻るというふうになる。そしてこれこそが kill リングがリングと呼ばれる所以である。最後まで来たらまた最初に戻るので、あたかもリストには終わりがないように見えるのだ！ (訳註：原文では最後に "And what is a ring, but an entity with no end?" という文が続く。これはどう訳したものだろう？)

B.2 yank

rotate-yank-pointer について学んだ後では、 yank 関数のコードは全く易しく見えるだろう。これには一箇所だけトリッキーな部分がある。 rotate-yank-pointer に渡す引数の計算の所だ。

コードは以下の通りである。

```
(defun yank (&optional arg)
  "Reinsert the last stretch of killed text.
More precisely, reinsert the stretch of killed text most
recently killed OR yanked.
With just C-u as argument, same but put point in front
(and mark at end).  With argument n, reinsert the nth
most recently killed stretch of killed text.
See also the command \\[yank-pop]."

  (interactive "*P")
  (rotate-yank-pointer (if (listp arg) 0
                         (if (eq arg '-) -1
                           (1- arg))))
  (push-mark (point))
  (insert (car kill-ring-yank-pointer))
  (if (consp arg)
      (exchange-point-and-mark)))
```

このコード全体を眺めてみると、最後の数行だけでもほぼ十分なことが分る。まずマークがプッシュされる。つまり、位置が記憶される。そして、 kill-ring-yank-pointer が指すリストの最初の要素 (即ち CAR) が挿入される。更に、もし関数に渡された引数がコンスセルであれば、ポイントとマークが交換され、ポイントは挿入されたテキストの最後ではなく最初の位置に置かれる。このオプションは説明文字列の中で説明されている。関数そのものはオプション "*P" でインタラクティブ宣言されている。これは、書き込み不可のバッファでは動作せず、未処理の前置引数が関数に渡されるという指定である。

引数の渡し方

yank で難しいのは rotate-yank-pointer に渡される引数の計算を理解する部分である。幸いなことに、これは見た目程難しいわけではない。

やっていることは、二つある if 式の片方ないしは両方を評価し、結果として得られた数値を rotate-yank-pointer に引数として渡すということである。

コードをコメント付きで見やすいように書くと、次のようになる。

```
(if (listp arg)                          ; if-part
    0                                    ; then-part
  (if (eq arg '-)                        ; else-part, 内部の if
      -1                                 ; 内部の if の then-part
    (1- arg))))                          ; 内部の if の else-part
```

このコードは二つの if 式からなる。一つはもう一方の else-part になっている。

最初の外側の if 式は yank に渡された引数がリストであるかどうかのテストである。奇妙なことに、 yank が引数無しで呼び出された場合にはこれは真になる。理由は、この場合省略可能な引数の値として nil が渡されるのだが、(listp nil) は真になるからである！従って、もし yank を引数無しで呼び出すと yank の内部の rotate-yank-pointer には引数として零が渡される。これは、 kill-ring-yank-pointer が指すリストは変化せず、その最初の要素が挿入されることを意味する。これは、期待通りの動作である。同様に、もし yank が引数 C-u で呼び出されたとすると、これもリストだと見做され、またもや rotate-yank-pointer

には零が渡される。(C-uは処理されない前置引数（4）を生成するため。) 同時に、この引数はこの関数の後の部分でコンスセルと見做されるので、ポイントは挿入されたテキストの始めに、マークは終わりに置かれる。(interactive の P という引数は、オプションの引数が渡されなかった場合や C-u が渡された場合に上で述べたような値を与えるように設計されている。)

ということで、引数無しの場合や引数が C-u の場合は、外側の if 式の then-part が扱う。そして、その他の場合は else-part が扱う。else-part はそれ自身が if 式になっている。

内部の if 式は、引数が負の符号かどうかを判定するテストである。(これは META キーと - キーを同時に押すか、ESC キーを押してから - キーを押すと発生する。) この場合、rotate-yank-pointer 関数に渡される引数は -1 になる。これは、kill-ring-yank-pointer の指す位置を一つ手前に戻すことになり、期待通りの動作となる。

もし内部の if 式の真偽テストが偽であったなら (つまり引数が負の符号でなければ)、このＳ式の else-part が評価される。これは、(1- arg) という式である。二つの if 式から、この場合になるのは引数が正の数であるか、もしくは (単に負の符号だけではない) 負の数である場合のみである。(1- arg) はその数から 1 を引いた値を返す。(1- は引数を 1 減らす関数である。) これは、もし rotate-yank-pointer の引数が 1 だったとすると、それが 0 になるということである。従って、kill-rotate-yank-pointer の最初の要素がヤンクされる。これも期待通りである。

負の引数を渡すとどうなるか

最後に、幾つかの疑問が湧く。もし「剰余関数」% や nthcdr関数に負の引数を与えたとするとどうなるだろうか。

答えはちょっと実験してみれば分る。(% -1 5) を評価すると、負の引数が返る。また、nthcdr が負の引数と共に呼び出されると、最初の引数が零であった場合と同じ値が返る。これは次のコードを評価すれば確かめられる。

以下では、'⇒' の後に書かれているものがその前のコードを評価した結果である。評価するには、いつものようにコードの直後にカーソルを置いて C-x C-e (eval-last-sexp) とタイプすると良い。これは、この文書を GNU Emacs の Info の中で読んでいる場合にはそのまま実行出来る。

```
(% -1 5)
     ⇒ -1

(setq animals '(cats dogs elephants))
     ⇒ (cats dogs elephants)

(nthcdr 1 animals)
     ⇒ (dogs elephants)

(nthcdr 0 animals)
     ⇒ (cats dogs elephants)

(nthcdr -1 animals)
     ⇒ (cats dogs elephants)
```

従って、負の符号や負の数が yank に渡された場合には、kill-rotate-yank-pointer は逆向きに周り、リストの最初に辿りつくと、そのまま止まるという結果になる。正の引数を与えた場合のように、一番最後まで行くとまた最初に戻るというふうにはならず、そこで止まってしまうわけである。これは合理的な動作である。ヤンクする際は、直前に切り取ったテキストをヤンクしたい場合が最も多く、普通は 30 回も前に kill したテキストを挿入しようとは思わないだろう。だから、リングの中を順に進んでいって一番最後まで行くことはあっても、後から最初の方に戻ろうとして、また一番最後に戻ってしまうなんてことは、あって欲しくないに違いない。

ついでにいっておくと、負の符号がついた数が yank に引数として渡される場合は、全て -1 として扱われる。これは、明らかにプログラムを単純にするためのものである。kill リングの中を最初の方に向かって一度に二つ以上もジャンプする必要はないし、負の数の絶対値を決定する関数を書くよりもこっちの方がずっと簡単である。

B.3 yank-pop

yank を理解してしまえば、yank-pop 関数は簡単に理解出来る。スペースの節約のために説明文字列を省くと、コードは次の通りである。

```
(defun yank-pop (arg)
  (interactive "*p")
  (if (not (eq last-command 'yank))
      (error "Previous command was not a yank"))
  (setq this-command 'yank)
  (let ((before (< (point) (mark))))
    (delete-region (point) (mark))
    (rotate-yank-pointer arg)
    (set-mark (point))
    (insert (car kill-ring-yank-pointer))
    (if before (exchange-point-and-mark))))
```

この関数は、小文字の 'p' でインタラクティブ宣言されている。従って、前置引数は処理されてから関数に渡される。このコマンドは、直前にヤンクを行った場合にのみ有効である。そうでなければ、エラーメッセージが出される。このチェックは変数 last-command で行う。(これについては他の所で説明した。Section 8.5, "copy-region-as-kill", p. 71, 参照。)

let 式では、変数 before を、ポイントがマークの前にあるか後にあるかによって真もしくは偽にセットする。そして、ポイントとマークの間のリージョンを削除する。これは、直前のヤンクによって挿入されたリージョンであり、置き換えようとしている部分である。次に kill-ring-yank-pointer を回して直前に挿入したテキストがまた挿入されないようにする。マークは新しいテキストが挿入されるべき部分にセットされ、kill-ring-yank-pointer が指すリストの要素が挿入される。その結果、ポイントは新しく挿入されたテキストの直後に置かれる。ここで、もし前回のヤンクでポイントが挿入されたテキストの前に置かれていたならポイントとマークを交換して、ポイントを今回挿入したテキストの最初の位置に移動する。これがこの関数でやっていることの全てである！

Appendix C ラベルと軸が付いたグラフ

座標軸を描くと、グラフを理解する手助けになる。目盛もつけたい。以前の章では (Chapter 15, “グラフを描く準備”, p. 133, 参照) グラフの本体を描くコードを書いたのであった。ここではその本体にそって縦軸と横軸を表示し、ラベルをつけるコードを書くことにしよう。

ここでの挿入は、バッファのポイントの下方を右方向に向って埋めていくので、新しいグラフ表示関数では、まず Y 軸ないしは縦軸を表示し、ついで本体を、そして最後に X 軸ないしは横軸を表示するという順番になる。この流れから、関数の中身のレイアウトは次のような感じになる。

1. コードの下準備。
2. Y 軸の表示。
3. グラフ本体の表示。
4. X 軸の表示。

以下に出来上がったグラフの例がどうあるべきかを載せておく。

```
10 -
                *
                *  *
                *  **
                *  ***
 5 -       *    *******
         * *** *******
         ************
       ***************
 1 - ****************
     |    |     |     |
     1    5    10    15
```

このグラフでは、縦軸と横軸には数によるラベルがついている。しかしながら、グラフの種類によっては横軸が時間で、月によるラベルをつけた方が良い場合も多いだろう。次のような感じだ。

```
 5 -       *
         * ** *
         *******
       ********** **
 1 - **************
     |     ^       |
     Jan   June    Jan
```

実際には、ちょっと考えれば横軸や縦軸について様々な形式を思いつく。複雑な関数にすることも可能だ。しかし複雑さは混乱を招く。それよりも、最初は単純な形式を選んで、後でそれを修正ないしは置き換えるのが良いだろう。

以上のことを考慮すると、次のようなアウトラインで print-graph 関数を書いていくのがいいだろう。

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

この print-graph 関数の雛型を元にして、各々の部分を順に書いていくことにしよう。

C.1 print-graph の変数リスト

print-graph 関数を書く際にまずやらねばならないことは、let式の変数リストを書くことである。(取り敢えず、インタラクティブ式や説明文字列のことは置いておく。)

変数リストでは幾つかの値を設定しなければならない。縦軸のラベルの一番上は少なくともグラフの高さでなくてはいけないので、まずこれについての情報が必要になる。print-graph-body 関数でもこの情報が必要だったことを思い出そう。同じ計算を別の場所で二度行う理由はどこにもないので、以前定義した print-graph-body を書換えてこの計算を一度だけにするべきである。

同様に、X 軸のラベルを表示する関数と print-graph-body 関数のどちらにおいても記号の幅の値を知る必要がある。これも以前の章の print-graph-body の定義を書換えて、最初に計算をすませてしまうべきであろう。

横軸のラベルの長さは少なくともグラフの長さの分はなければならない。しかし、この情報は横軸を表示する関数だけに必要なものである。従って、ここでやらなくとも良い。

以上のことから、print-graph の let 式の変数リストは次のような感じになる。

```
(let ((height (apply 'max numbers-list)) ; 最初のバージョン
      (symbol-width (length graph-blank)))
```

が、後で見るように、これではちょっとまずい。

C.2 関数 print-Y-axis

print-Y-axis 関数の仕事は、縦軸として次のようなラベルを表示することである。

```
        10 -



         5 -



         1 -
```

この関数には、グラフの高さを渡してやる必要がある。それを元に適切な目盛と数を構成し、挿入するわけである。

図で見ると、Y 軸のラベルがどうなるかべきか一目瞭然である。しかし、これを言葉で表しその仕事をする関数を書くとなると、ちょっとややこしくなる。単に 5 行ごとに数や目盛をつければよいというのは間違いである。'1' と '5' の間には三行しかない (2、3、4 行)。しかし、'5' と '10'の間には四行ある (6、7、8、9 行)。まずは一行目にベースとなる行として数 1 と目盛をつけ、その他に最下行から 5 行ごとに目盛と数をつけていくといった方がよいだろう。

次の問題は、ラベルの高さをどうするかである。例えば、グラフの縦の列の中で最も高いものの高さが 7 だったとしよう。この場合、最も上のラベルは '5-' にして、グラフを上に突き出させるべきだろうか。それとも 7 以上の最大の 5 の倍数として '10 -' の所までラベルを付けるべきだろうか。

これは後者の方がよいだろう。大抵のグラフは長方形のなかに描かれており、側面は刻み幅の整数倍になっている。今の場合なら、5、10、15 というふうな 5 の倍数である。しかし、縦軸の仕様としてこのようなものを採用するとなると、以前の関数の変数リストでの、単純に高さを計算するＳ式ではまずいことに気付く。これは (apply 'max numbers-list) というものだったが、これでは単に正確なグラフの縦の列の高さの最大値が出るだけで、5 の倍数になるような調整はしてはくれない。もっと複雑な関数が必要なわけである。

いつものことだが、複雑な問題も幾つかの小さな問題に分割して考えれば単純な問題になることが多い。

最初に、いつグラフの高さが 5 の整数倍になるかを考えてみよう。これは 5、 10、 15 等の 5 の倍数の時である。この場合は、この値を即 Y 軸の高さとしてよい。

ある数が 5 の倍数になるかどうかを見るには、この数を 5 で割ってみて、余りがどうなるかを見るのが早い。もし余りが無ければ、その数は 5 の倍数である。 7 の場合は余りが 2 になるので、これは 5 の倍数ではない。ちょっと言い方を変えて小学校ふうに説明するなら、 5 は 7 の中に一回だけ含まれて、残りは 2 になる。一方、 5 は 10 の中には 2 回含まれ、残りはない。従って、 10 は 5 の倍数ということになる。

C.2.1 寄り道: 剰余の計算

Lisp では余りを計算する関数は % である。この関数は最初の引数を二番目の引数で割った時の余りを返す。あいにく、 % という Emacs Lisp 関数は apropos では見つけられない。つまり、 M-x apropos RET remainder RET とすると、何も見つけられない。 % という関数の存在を知るためにはこの文書などの本を見るか、 Emacs Lisp のソースを眺めるしか方法がない。 % 関数は rotate-yank-pointer の中で使われており、これは次の所で説明されている。 (Section B.1.1, “rotate-yank-pointer の本体”, p. 166, を参照。)

% 関数を試してみるには、次の二つの S 式を評価してみると良い。

```
(% 7 5)

(% 10 5)
```

最初の S 式は 2 を返すし、二番目の S 式は 0 を返す。

返された値が 0 かそうでないかのテストには zerop 関数が使える。この関数は引数として数値を取り、それが零なら t を返す。

```
(zerop (% 7 5))
     ⇒ nil

(zerop (% 10 5))
     ⇒ t
```

というわけで、次のような S 式を書けば、高さが 5 で割切れる場合には tが返される。

```
(zerop (% height 5))
```

(height の値は勿論 (apply 'max numbers-list). で分る。)

一方、もし height が 5 の倍数でない場合、値が 5 の倍数になるように再設定する必要がある。これは既にお馴染みの関数を使えば単純な算数にすぎない。まずは height を 5 で割ってこの中に何回 5 が含まれるかを調べる。例えば 12 なら 2 回含まれる。この値に 1 加えて 5 倍すれば、棒グラフの最大の高さよりも大きい最初の 5 の倍数が得られる。 5 は 12 の中には 2 回含まれるので、これに 1 を足して 5 倍すると 15 が求まる。これが 12 を越える最初の 5 の倍数であ。この作業を行う S 式は次の通りである。

```
(* (1+ (/ height 5)) 5)
```

例えばもし次の S 式を評価したなら、結果は 15 になる。

```
(* (1+ (/ 12 5)) 5)
```

これまでの議論では「5」を Y 軸についての目盛幅としてきたわけだが、ここは他の値を使っても構わない。一般性のために、 5 を他の値も設定出来るように変数で置き換えよう。私としては、この変数は Y-axis-label-spacing と名付けるのが良いと思う。この変数と if 式を使うと、次の S 式が出来る。

```
(if (zerop (% height Y-axis-label-spacing))
    height
  ;; else
  (* (1+ (/ height Y-axis-label-spacing))
     Y-axis-label-spacing))
```

このＳ式は、もし height が Y-axis-label-spacing の値の倍数なら height の値そのものを返し、そうでない場合は、それより大きい最小の Y-axis-label-spacing の倍数の値を計算して返す。

それでは、このＳ式を print-graph 関数の中の let 式の中に (Y-axis-label-spacing の設定の後で) 埋め込んでみよう。

```
(defvar Y-axis-label-spacing 5
  "Number of lines from one Y axis label to next.")

...
(let* ((height (apply 'max numbers-list))
       (height-of-top-line
        (if (zerop (% height Y-axis-label-spacing))
            height
          ;; else
          (* (1+ (/ height Y-axis-label-spacing))
             Y-axis-label-spacing)))
       (symbol-width (length graph-blank))))
...
```

(let* 関数を使っていることに注意。始めに変数 height の初期値が (apply 'max numbers-list) 式で計算され、その値を使って他の変数の値を計算しているためである。 let* についての詳細は、次を参照。 “let* 式”, p. 100.)

C.2.2 Y 軸の要素の構成

縦軸を表示する際には、‘5 -’ とか ‘10 -’ 等を 5 行ごとに挿入したい。更に、数は下の部分を揃えたい。つまり、小さな桁の数は、前に空白を置くことになる。もし二桁の数が出てきた時には、一桁の数の先頭には一つの空白を置くことになるわけである。

数の長さを求めるには、 length 関数が使われる。しかし、 length 関数は文字列に対してしかうまく動作せず、数値は扱えない。そこでまず、数を数値から文字列に変換する必要がある。これは、 int-to-string 関数を使って行うことが出来る。例えば、次のような感じである。

```
(length (int-to-string 35))
     ⇒ 2

(length (int-to-string 100))
     ⇒ 3
```

更に、各ラベルにおいて、各々の数には ‘ - ’ のような文字列を付け加えたい。これを、 Y-axis-tic マーカと呼ぶことにしよう。この変数は、 defvar を使って定義する。

```
(defvar Y-axis-tic " - "
   "String that follows number in a Y axis label.")
```

Y ラベルの長さは、 Y axis tick mark とグラフの一番上の数の桁数の和になる。

```
(length (concat (int-to-string height) Y-axis-tic)))
```

この値は、 print-graph 関数の中の変数リストの所で、 full-Y-label-width の値として計算される。(我々が最初にこの関数を考えた時には、この関数は変数リストの中に入れることは考えていなかったことに注意しよう。)

縦軸を完成させるには、目盛記号を数と結合する必要がある。そして、その前には数の桁数によって空白が付いたりする。というわけで、ラベルには三つのパートがあることになる。先頭の空白 (無い場合もある)、数、そして目盛記号である。この関数には、指定された行に対する数の値と一番上の行の幅が渡される。これらの値は print-graph によって (一回だけ) 計算される。

```
(defun Y-axis-element (number full-Y-label-width)
  "Construct a NUMBERed label element.
A numbered element looks like this ‘  5 - ’,
and is padded as needed so all line up with
the element for the largest number."
  (let* ((leading-spaces
          (- full-Y-label-width
             (length
              (concat (int-to-string number)
                      Y-axis-tic)))))
    (concat
     (make-string leading-spaces ? )
     (int-to-string number)
     Y-axis-tic)))
```

Y-axis-element 関数は、(もしあれば) 先頭の空白と、文字列としての数と、目盛記号を結合するものである。

先頭にいくつ空白をつければよいかは、ラベルの実質部分の長さ — 数の長さと目盛記号の長さの和 — を望まれるラベルの長さから引くことで求められる。

空白は make-string 関数を使って挿入される。この関数は、二つの引数を取る。一つは文字列の長さ、もう一つは特定の形式で書かれた挿入する文字のシンボルである。今の場合、この形式は ‘? ’ のように、疑問符に続く空白という形をしている。このような文字の表し方についての説明は、次を参照。 Section “Character Type” in *The GNU Emacs Lisp Reference Manual*.

int-to-string 関数は、結合式の中で数を文字列に変換するために使われている。文字列に変換してから先頭の空白と目盛記号と結合するのである。

C.2.3 Y 軸全体の構成

前節までの関数で、縦軸のラベルとして挿入する番号と空白のついた文字列のリストを生成する関数を構成するのに必要なツールが全て揃う。

```
(defun Y-axis-column (height width-of-label)
  "Construct list of Y axis labels and blank strings.
For HEIGHT of line above base and WIDTH-OF-LABEL."
  (let (Y-axis)
    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; ラベル挿入
          (setq Y-axis
                (cons
                 (Y-axis-element height width-of-label)
                 Y-axis))
        ;; そうでなければ空白挿入
        (setq Y-axis
              (cons
               (make-string width-of-label ? )
               Y-axis)))
      (setq height (1- height)))
    ;; ベースライン挿入
    (setq Y-axis
          (cons (Y-axis-element 1 width-of-label) Y-axis))
    (nreverse Y-axis)))
```

この関数では、まず height の値から出発してその値を一つずつ減らしていき、各々の引き算が終わった所でその値が Y-axis-label-spacingの整数倍になっているかどうかを判定する。そして、もしそうなっていれば、Y-axis-element 関数を使って番号付きのラベルを作成し、そうでなければ make-string 関数を使って空白のラベルを作成する。最下行は、番号 1 と目盛記号からなっている。

C.2.4 print-Y-axis 最終版

(訳註：題名に最終版と書いてあるが、実際には Section C.4, “グラフ全体の表示”, p. 183 に出ているものが真の最終版である。原文ではここに最終版が載っていたのだが、そうすると、中で使っている y-axis-column 関数が最終版でないために、その次に述べられているテストでエラーが出てしまう。)

Y-axis-column 関数によって構成されたリストは print-Y-axis関数に渡される。これが実際にそのリストを挿入する。

```
(defun print-Y-axis
  (height full-Y-label-width)
  "Insert Y axis using HEIGHT and FULL-Y-LABEL-WIDTH.
Height must be the  maximum height of the graph.
Full width is the width of the highest label element."
;; Value of height and full-Y-label-width
;; are passed by ‘print-graph’.
  (let ((start (point)))
    (insert-rectangle
     (Y-axis-column height full-Y-label-width))
    ;; グラフ挿入の準備としてポイントを移動
    (goto-char start)
    ;; full-Y-label-width の値の分だけポイントを前方に移動
    (forward-char full-Y-label-width)))
```

print-Y-axis は Y-axis-column によって作成された Y 軸のラベルを挿入するのに insert-rectangle を使っている。更に、グラフの本体部分を挿入するための適切な位置にポイントを移動している。

以下のようにして print-Y-axis をテストしてみることが出来る。

1. まず次の変数や関数をインストールする。

```
Y-axis-label-spacing
Y-axis-tic
Y-axis-element
Y-axis-column
print-Y-axis
```

2. 次のＳ式をコピーする。

```
(print-Y-axis 12 5)
```

3. '*scratch*' バッファに移り、カーソルを縦軸のラベルを挿入したい位置まで移動する。
4. M-ESC (eval-expression) とタイプする。
5. print-Y-axis 式をミニバッファに C-y (yank) を使ってヤンクする。
6. このＳ式を評価するために RET を押す。

Emacs はラベルを縦に表示する。一番上は '10 - ' である。(print-graph 関数は、height-of-top-line の値を渡すのだが、今の場合これは 15 である。) (訳註：ここは原文も混乱しているみたいである。print-Y-axis は print-graph の中で使われている。その中では print-Y-axis に第一引数として渡されるのは height-of-top-line の値なのであるが、print-graph 関数の中の局所変数としての *height* の値が 12 で *Y-axis-label-spacing* の値が 5 だったとすると、この height-of-top-line の値が 15 になるということを言いたかったのであろう。詳しくは最終版 print-graph を参照のこと。)

C.3 関数 print-X-axis

X 軸のラベルは Y 軸のラベルと大変似ている。違うのは、目盛記号が数の上にのっていることである。ラベルは次のようになる。

```
|    |     |     |
1    5    10    15
```

最初の目盛はグラフの最初の桁の下の部分にきており、その前には幾つかの空白がある。これらの空白は、Y 軸のラベルが上に来る部分である。二番目、三番目、四番目、或いはそれに続く目盛は全て等間隔で並んでいる。この間隔は X-axis-label-spacing の値で決まる。

X 軸の二行目は、空白に続く数からなる。この数字の間隔もまた X-axis-label-spacing の値による。

変数 X-axis-label-spacing の値そのものは symbol-width という単位を元に決められるべきである。棒グラフを表示するためのシンボルの幅を変更する時に、ラベルの付け方まで変えなければならないなんて事態は避けたいだろう。

print-X-axis 関数は多かれ少なかれ print-Y-axis 関数と同じ形をしている。違うのは、目盛の行と数字の行の二行あるということである。この二つは各々別の関数で書いて、print-X-axis の中で一緒にすることにする。

以下が、これから行う三段階のステップである。

1. X 軸の目盛記号を表示する関数 print-X-axis-tic-line を書く。
2. X 軸の数字を表示する関数 print-X-axis-numbered-line を書く。
3. この print-X-axis-tic-line と print-X-axis-numbered-line を使って、上の二つの行を両方とも表示する関数 print-X-axis を書く。

C.3.1 X 軸の目盛記号

最初の関数では、X 軸の目盛記号を表示する。まずは、目盛記号とその間の記号を指定しないといけない。

```
(defvar X-axis-label-spacing
  (if (boundp 'graph-blank)
      (* 5 (length graph-blank)) 5)
  "Number of units from one X axis label to next.")
```

(graph-blank の値はまた別の defvar で設定されることに注意しよう。boundp という述語は、既に何かの値がセットされているかどうかを判定するものである。もし何もセットされていなければ、boundp は nil を返す。もし graph-blank が設定されず、また、この条件分岐式がなかったとすると、Symbol's value as variable is void というエラーメッセージを受けとることになる。

```
(defvar X-axis-tic-symbol "|"
  "String to insert to point to a column in X axis.")
```

目標は、次のような行を作成することである。

```
          |   |    |    |
```

最初の目盛は、Y 軸のラベルのためのスペースを空けるためにインデントされた最初の棒グラフの開始の桁と同じだけインデントしないといけない。

目盛の要素は、ある目盛から次の目盛までのスペースと目盛記号からなる。空白の数は目盛記号の幅と X-axis-label-spacing から決まる。

コードは次の通りである。

```
;;; X-axis-tic-element
...
(concat
 (make-string
  ;; 空白の文字列を作成
  (-  (* symbol-width X-axis-label-spacing)
      (length X-axis-tic-symbol))
  ? )
 ;; 空白を目盛記号と結合
 X-axis-tic-symbol)
...
```

次に、最初の目盛をグラフの開始位置までインデントするために、どれだけ空白が要るかを数える必要がある。これには、print-graph 関数によって渡された full-Y-label-width の値を利用する。

X-axis-leading-spaces を作るためのコードは次の通りである。

```
;; X-axis-leading-spaces
...
(make-string full-Y-label-width ? )
...
```

また、横軸の長さも決めなければならない。これは、数のリストの長である。更に、横軸の目盛の数も決める必要がある。

```
;; X-length
...
(length numbers-list)
;; tic-width
...
(* symbol-width X-axis-label-spacing)
```

```
;; number-of-X-tics
(if (zerop (% (X-length tic-width)))
    (/ (X-length tic-width))
  (1+ (/ (X-length tic-width))))
```

これらを使うと、X 軸の目盛の行を表示する関数は次のようになる。

```
(defun print-X-axis-tic-line
  (number-of-X-tics X-axis-leading-spaces X-axis-tic-element)
  "Print tics for X axis."
    (insert X-axis-leading-spaces)
    (insert X-axis-tic-symbol)  ; 最初の桁の下に
    ;; 右の位置に二番目の目盛記号を挿入
    (insert (concat
             (make-string
              (-  (* symbol-width X-axis-label-spacing)
                  ;; 二番目の目盛記号まで空白を挿入
                  (* 2 (length X-axis-tic-symbol)))
              ? )
             X-axis-tic-symbol))
    ;; 残りの目盛記号を挿入
    (while (> number-of-X-tics 1)
      (insert X-axis-tic-element)
      (setq number-of-X-tics (1- number-of-X-tics))))
```

数字の行も同様にして簡単に書ける。

まずは前に空白の付いた番号を作成する。

```
(defun X-axis-element (number)
  "Construct a numbered X axis element."
  (let ((leading-spaces
         (-  (* symbol-width X-axis-label-spacing)
             (length (int-to-string number)))))
    (concat (make-string leading-spaces ? )
            (int-to-string number))))
```

次に、数字のついた行を表示するための関数を作る。これはまず最初の桁に数 “1” を付けるところから始まる。

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces)
  "Print line of X-axis numbers"
  (let ((number X-axis-label-spacing))
    (insert X-axis-leading-spaces)
    (insert "1")
    (insert (concat
             (make-string
              ;; 次の数字の所まで空白を挿入
              (-  (* symbol-width X-axis-label-spacing) 2)
              ? )
             (int-to-string number)))
```

```
;; 残りの数字を挿入
(setq number (+ number X-axis-label-spacing))
(while (> number-of-X-tics 1)
  (insert (X-axis-element number))
  (setq number (+ number X-axis-label-spacing))
  (setq number-of-X-tics (1- number-of-X-tics)))))
```

最後に、print-X-axis-tic-line と print-X-axis-numbered-line を使って、print-X-axis を書く。

この関数では print-X-axis-tic-line と print-X-axis-numbered-line の両方で使われている局所変数の値を決定してから、これらの関数を呼び出さなければならない。また、二つの行の区切りに復帰コードを表示する必要もある。

この関数は、5 つの局所変数を指定する変数リストと二つの行を表示する関数の呼び出しからなる。

```
(defun print-X-axis (numbers-list)
  "Print X axis labels to length of NUMBERS-LIST."
  (let* ((leading-spaces
          (make-string full-Y-label-width ? ))
         ;; symbol-width は graph-body-print で与えられる。
         (tic-width (* symbol-width X-axis-label-spacing))
         (X-length (length numbers-list))
         (X-tic
          (concat
           (make-string
            ;; 空白の文字列を作成
            (-  (* symbol-width X-axis-label-spacing)
                (length X-axis-tic-symbol))
            ? )
           ;; 空白を目盛記号と結合
           X-axis-tic-symbol))
         (tic-number
          (if (zerop (% X-length tic-width))
              (/ X-length tic-width)
            (1+ (/ X-length tic-width)))))
    (print-X-axis-tic-line tic-number leading-spaces X-tic)
    (insert "\n")
    (print-X-axis-numbered-line tic-number leading-spaces)))
```

これで、print-X-axis のテストが出来る。

1. まずは X-axis-tic-symbol, X-axis-label-spacing, print-X-axis-tic-line, などの変数とともに X-axis-element, print-X-axis-numbered-line, print-X-axis という関数をインストールする。
2. 次のＳ式をコピーする。

```
(progn
 (let ((full-Y-label-width 5)
       (symbol-width 1))
   (print-X-axis
    '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16))))
```

3. '*scratch*' バッファに移り、カーソルを軸のラベルを表示したい位置にまで動かす。
4. M-ESC (eval-expression) とタイプする。
5. C-y (yank) を使って、テストのＳ式をヤンクする。

6. その式を評価するために、 RET を押す。

これで Emacs は次のような横軸を表示してくれるはずだ。

```
        |    |     |     |     |
        1    5    10    15    20
```

C.4 グラフ全体の表示

さて、これでグラフ全体を表示する準備が整った。

きちんとしたラベルが付いたグラフを表示する関数を書く際には、以前作成したアウトラインに従うわけだが (Appendix C, “ラベルと軸の付いたグラフ”, p. 173, 参照)、幾つか追加することがある。

アウトラインは以下のようなものだった。

```
(defun print-graph (numbers-list)
  "documentation..."
  (let ((height  ...
         ...))
    (print-Y-axis height ... )
    (graph-body-print numbers-list)
    (print-X-axis ... )))
```

最終的なバージョンは、上の計画とは二つの点で違っている。まず、変数リストの中で追加して計算するものがある。次に、ラベルの増加のさせ方を指定するオプションがある。後者の特徴はかなり本質的である。そうしないことには、画面や紙面に合わなくなってしまう可能性があるからだ。

この新しい特徴を実現するには、 Y-axis-column 関数にいくらか変更を加えて vertical-step という変数を加える必要がある。同時に、 print-Y-axis も書換える。新しい関数は次のようになる。

```
;;; 最終バージョン。
(defun Y-axis-column
  (height width-of-label &optional vertical-step)
  "Construct list of labels for Y axis.
HEIGHT is maximum height of graph.
WIDTH-OF-LABEL is maximum width of label.
VERTICAL-STEP, an option, is a positive integer
that specifies how much a Y axis label increments
for each line.  For example, a step of 5 means
that each line is five units of the graph."
  (let (Y-axis
        (number-per-line (or vertical-step 1)))
    (while (> height 1)
      (if (zerop (% height Y-axis-label-spacing))
          ;; ラベルの挿入
          (setq Y-axis
                (cons
                 (Y-axis-element
                  (* height number-per-line)
                  width-of-label)
                 Y-axis))
```

```
          ;; そうでない場合は, 空白の挿入
          (setq Y-axis
                (cons
                 (make-string width-of-label ? )
                 Y-axis)))
        (setq height (1- height)))
      ;; ベースラインの挿入
      (setq Y-axis (cons (Y-axis-element
                          (or vertical-step 1)
                          width-of-label)
                         Y-axis))
      (nreverse Y-axis)))

;;; 最終バージョン。
(defun print-Y-axis
  (height full-Y-label-width &optional vertical-step)
  "Insert Y axis using HEIGHT and FULL-Y-LABEL-WIDTH.
Height must be the  maximum height of the graph.
Full width is the width of the highest label element.
Optionally, print according to VERTICAL-STEP."
;; Value of height and full-Y-label-width
;; are passed by 'print-graph'.
  (let ((start (point)))
    (insert-rectangle
     (Y-axis-column height full-Y-label-width vertical-step))
    ;; Place point ready for inserting graph.
    (goto-char start)
    ;; Move point forward by value of full-Y-label-width
    (forward-char full-Y-label-width)))
```

グラフの最大の高さの値とシンボルの幅は print-graph の中の let 式によって計算される。従って、graph-body-print はそれらを受け取ることが出来るように変更しなければならない。

```
;;; 最終バージョン。
(defun graph-body-print (numbers-list height symbol-width)
  "Print a bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values.
HEIGHT is maximum height of graph.
SYMBOL-WIDTH is number of each column."
  (let (from-position)
    (while numbers-list
      (setq from-position (point))
      (insert-rectangle
       (column-of-graph height (car numbers-list)))
      (goto-char from-position)
      (forward-char symbol-width)
      ;; 各桁ごとにグラフを描写
      (sit-for 0)
      (setq numbers-list (cdr numbers-list)))
    ;; X 軸のラベルのためにポイントを移動
    (forward-line height)
    (insert "\n")))
```

最後に、 print-graph 関数のコードを載せておこう。

```
;;; 最終バージョン。
(defun print-graph
  (numbers-list &optional vertical-step)
  "Print labelled bar graph of the NUMBERS-LIST.
The numbers-list consists of the Y-axis values.

Optionally, VERTICAL-STEP, a positive integer,
specifies how much a Y axis label increments for
each line.  For example, a step of 5 means that
each row is five units."
  (let* ((symbol-width (length graph-blank))
         ;; height は最大の数でもあり、
         ;; 最大桁の数でもある。
         (height (apply 'max numbers-list))
         (height-of-top-line
          (if (zerop (% height Y-axis-label-spacing))
              height
            ;; else
            (* (1+ (/ height Y-axis-label-spacing))
               Y-axis-label-spacing)))
         (vertical-step (or vertical-step 1))
         (full-Y-label-width
          (length
           (concat
            (int-to-string
             (* height-of-top-line vertical-step))
            Y-axis-tic))))

    (print-Y-axis
     height-of-top-line full-Y-label-width vertical-step)
    (graph-body-print
     numbers-list height-of-top-line symbol-width)
    (print-X-axis numbers-list)))
```

C.4.1 print-graph のテスト

print-graph 関数を短い数のリストで試してみよう。

1. (他のコードに加えて) Y-axis-column、 print-Y-axis、 graph-body-print、そして print-graph の各最終バージョンをインストールする。
2. 次のＳ式をコピーする。

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1))
```

3. '*scratch*' バッファに移り、カーソルをグラフを表示したい位置にまで移動する。
4. M-ESC (eval-expression) とタイプする。
5. C-y (yank) を使って、上のＳ式をミニバッファにヤンクする。
6. このＳ式を評価するために RET を押す。

Emacs は次のようなグラフを表示する。

```
10 -


          *
         **   *
 5 -    ****  *
        **** ***
      * *********
      ************
 1 - ************

      |    |     |     |
      1    5    10    15
```

一方、次のＳ式を評価して print-graph に vertical-step として 2 を与えてみたとする。

```
(print-graph '(3 2 5 6 7 5 3 4 6 4 3 2 1) 2)
```

するとグラフは次のようになるはずだ。

```
20 -


          *
         **   *
10 -    ****  *
        **** ***
      * *********
      ************
 2 - ************

      |    |     |     |
      1    5    10    15
```

(疑問：横軸の左の '2' はバグだろうか、仕様だろうか。もしバグであると思うのなら、そして '2' ではなく '1' であるべきだと思うのなら (或いは '0' であるべきだと思うのなら)、ソースを修正すればよい。)

C.4.2 単語やシンボルの数のグラフ化

さて、やっと目的のグラフの所まで来た。いよいよ単語やシンボルの数が 10 より少ない関数定義がどれだけあるか、 10 から 19 までの間だとどれだけか、 20 から 29 までだとどうか、といったことを示してくれるグラフを描く関数を書くわけである。

これは幾つかのプロセスに分けて行う。まずは、必要なコードを全てロードしてあることを確認しよう。

異なる値を設定してしまった場合に備えて、 top-of-ranges の値を再設定しておく方がよいだろう。それには以下を評価すればよい。

```
(setq top-of-ranges
 '(10  20  30  40  50
   60  70  80  90 100
  110 120 130 140 150
  160 170 180 190 200
  210 220 230 240 250
  260 270 280 290 300)
```

次に、各々の範囲に入っている単語やシンボルの数のリストを作成しよう。

まず次を評価する。

```
(setq list-for-graph
       (defuns-per-range
         (sort
          (recursive-lengths-list-many-files
           (directory-files "/usr/local/emacs/lisp"
                            t ".+el$"))
          '<)
         top-of-ranges))
```

私の機械では、これに一時間程かかった。私の持っている Emacs version 19.23 だと 303 の Lisp ファイルを見ていることになる。この計算の後では、 list-for-graph の値は次のようになる。

```
(537 1027 955 785 594 483 349 292 224 199 166 120 116 99
90 80 67 48 52 45 41 33 28 26 25 20 12 28 11 13 220)
```

これは、私の Emacs には 10 以下の単語やシンボルしか持たない関数定義が 537 個あり、 10 以上 19 未満の単語やシンボルを持つものが 1,027 個、 20 から 29 までだと 955 個、などというふうになっていることを示している。

このリストから明らかに見て取れることは、殆どの関数定義では、中に含まれる単語やシンボルの数は、 10 から 30 までだということである。

さて、これをグラフに表示することにしよう。我々は、 1,030 行もの高さのグラフは描きたくない。そうではなく、 25 行以下程度のグラフを描きたいのである。この高さのグラフであれば、大抵のモニタで表示出来るだろう。また、紙にも印刷しやすい。

これは、今の場合、 list-for-graph の各々の値を 50 分の 1 に縮小しないといけないということである。

この操作をやってくれる簡単な関数を、以下に挙げる。この関数には、今までに出てこなかった二つの関数 mapcar と lambda が出てくる。

```
(defun one-fiftieth (full-range)
  "Return list, each number one-fiftieth of previous."
 (mapcar '(lambda (arg) (/ arg 50)) full-range))
```

lambda 式

lambda は無名関数、即ち名前のない関数のシンボルである。無名関数を使う場合には毎回本体を全て含めて書かなければならない。

従って、

```
(lambda (arg) (/ arg 50))
```

であれば、常に arg として渡されたものを 50 で割った値を返す、という関数定義になる。

例えば、以前、 multiply-by-seven 関数というのを説明したことがある。これは引数を 7 倍するものであった。今回の関数も同様である。異なる点は、引数を 50 で割るということと、名前がないということだけである。 multiply-by-seven を無名関数で書き直すと次のようになる。

```
(lambda (number) (* 7 number))
```

(Section 3.1, “特殊形式 defun”, p. 19, を参照。)

もし 3 を 7 倍したいなら、次のように書くことが出来る。

```
(multiply-by-seven 3)
 \_______________/ ^
         |         |
      function  argument
```

このＳ式は 21 を返す。

同様に、次のようにも書ける。

```
((lambda (number) (* 7 number)) 3)
 \____________________________/ ^
               |                |
      anonymous function     argument
```

100 を 50 で割る場合は次のように書く。

```
((lambda (arg) (/ arg 50)) 100)
 \_____________________/  \_/
            |              |
   anonymous function   argument
```

このＳ式は 2 を返す。上の 100 は関数に渡されるもので、その数を関数が 50 で割っているのである。

lambda についてのより詳しいことは、 Section “Lambda Expressions” in *The GNU Emacs Lisp Reference Manual*, を参照せよ。 Lisp や lambda 式は、 Lambda 計算から派生したものである。

関数 mapcar

mapcar は最初の引数を二番目の引数の各々の要素にともに、順に呼び出す関数である。二番目の引数はシーケンス (sequence) でないといけない。

例えば、

```
(mapcar '1+ '(2 4 6))
     ⇒ (3 5 7)
```

という感じである。関数 1+ は引数に 1 を加える関数であるが、これがその後のリストの各要素について実行され、新しいリストを返すわけである。

これを apply と比較してみよう。こちらは最初の要素に対し、二番目以降の全ての引数を引数として渡して実行させるものだった。 (apply の説明については Chapter 15, “グラフを描く準備”, p. 133, を参照。)

one-fiftieth の定義では、最初の引数は、

```
(lambda (arg) (/ arg 50))
```

という無名関数である。また、二番目の引数は full-range であり、これは list-for-graph にバインドされる。

S式全体は次の通りである。

```
(mapcar '(lambda (arg) (/ arg 50)) full-range))
```

mapcar についての詳細は Section “Mapping Functions” in *The GNU Emacs Lisp Reference Manual*, を参照。

one-fiftieth 関数を使うと、各々の要素が、対応する list-for-graph の要素の 50 分の 1 であるようなリストを生成することが出来る。

```
(setq fiftieth-list-for-graph
      (one-fiftieth list-for-graph))
```

結果として出来るリストは次のようになる。

```
(10  20  19  15  11  9  6  5  4  3  3  2  2
1  1  1  1  0  1  0  0  0  0  0  0  0  0  0  0  0  4)
```

これは、今すぐにでも表示出来そうである！ (もっともここで情報が失われていることにも気付く。上の方の範囲の多くの要素は 0 である。これは、その範囲の単語やシンボルを持つ関数定義の数が 50 未満であることを示しているだけであって、そういう関数定義が全くないといっているわけではない。)

まだバグがある . . .

私は、「今すぐにでも表示出来そう」と書いた。勿論、 print-graph にはバグがあるのだ . . . これには vertical-step オプションがあるが、 horizontal-step オプションはない。 top-of-range の範囲は、10 ごとに 10 から 300 まである。しかし、 print-graph は 1 ずつしか表示出来ない。

これは、ある人々が最も見つけにくいタイプのバグだと考えている、考慮不足によるバグ (訳註：原文では bug of omission. これはどう訳すべきか) の古典的な例である。これは、コードを調べることで見つかるようなバグではない。コード自体に誤りはないからだ。これは単に何かが欠落しているということである。このようなバグに対するもっとも正しい態度は、プログラムをなるべく早く、何回もテストしてみることである。そして、コードを出来るだけ理解しやすく簡単に変更出来るようにアレンジしていくのである。また、一度書いたものは、すぐにではなくともいつかは書換えられるものだということを頭に入れておくようにしよう。これはなかなか実行が困難な格言ではあるが。

作業対象となるのは print-X-axis-numbered-line 関数である。また、 print-X-axis と print-graph もそれに合わせて書換える必要がある。もっともやることはそれ程多いわけではない。一つだけ面倒な部分がある。数字が、目盛記号の下に綺麗に並ばなければならないということである。この点はちょっとばかし頭をひねる必要がある。

以下が修正した print-X-axis-numbered-line である。

```
(defun print-X-axis-numbered-line
  (number-of-X-tics X-axis-leading-spaces
   &optional horizontal-step)
  "Print line of X-axis numbers"
  (let ((number X-axis-label-spacing)
        (horizontal-step (or horizontal-step 1)))
```

```
    (insert X-axis-leading-spaces)
    ;; 頭の余計な空白を削除
    (delete-char
     (- (1-
         (length (int-to-string horizontal-step)))))
    (insert (concat
             (make-string
              ;; 空白を挿入
              (-  (* symbol-width
                     X-axis-label-spacing)
                  (1-
                   (length
                    (int-to-string horizontal-step)))
                  2)
              ? )
             (int-to-string
              (* number horizontal-step))))
    ;; 残りの数を挿入
    (setq number (+ number X-axis-label-spacing))
    (while (> number-of-X-tics 1)
      (insert (X-axis-element
               (* number horizontal-step)))
      (setq number (+ number X-axis-label-spacing))
      (setq number-of-X-tics (1- number-of-X-tics)))))
```

もしあなたがこれを Info の中で読んでいるなら、print-X-axis と print-graph の新しいバージョンを見ることが出来る。もしこれを印刷された本で読んでいるとしたら、変更する行が下に書かれているはずだ。(印刷するにはコードがちょっと多過ぎるので。)

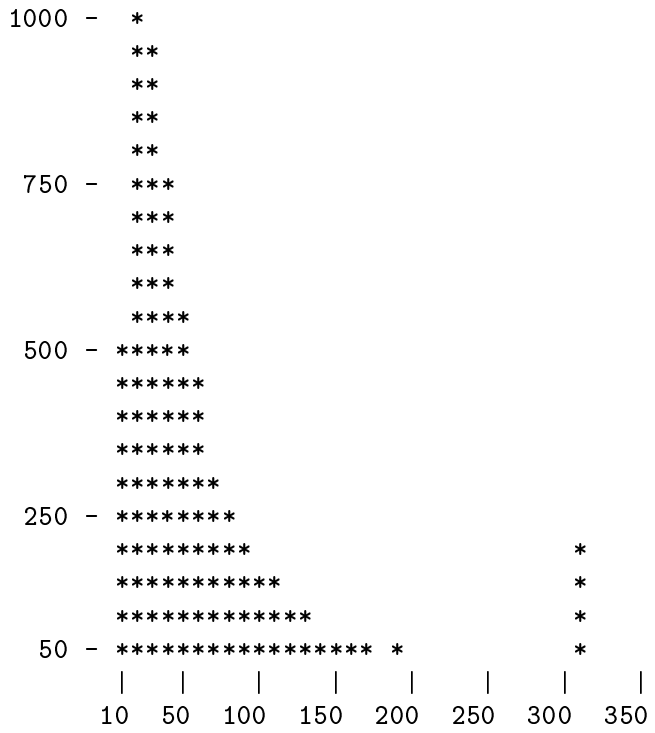
```
(defun print-X-axis (numbers-list horizontal-step)
  ...
    (print-X-axis-numbered-line
     tic-number leading-spaces horizontal-step))

(defun print-graph
  (numbers-list
   &optional vertical-step horizontal-step)
  ...
    (print-X-axis numbers-list horizontal-step))
```

C.4.3 表示されたグラフ

全てをインストールし終ったら、 print-graph コマンドを次のようにして呼び出そう。

```
(print-graph fiftieth-list-for-graph 50 10)
```

次のようなグラフが表示されるはずだ。



最も大きな関数のグループは、 10 から 19 までの単語やシンボルを含んでいるもののグループである。

Index

%

% (remainder function) ... 175

(

(debug) in code ... 158

*

* (multiplication) ... 20
* for read-only buffer ... 44
'*Backtrace*' buffer ... 156
'*scratch*' buffer ... 84

.

'.emacs' file ... 141
'.emacs' file, beginning of ... 142

/

/ (division) ... 49

>

> (greater than) ... 27

<

<= (less than or equal) ... 86

A

add-hook ... 144
and ... 100
Anonymous function ... 187
append-to-buffer ... 38
apply ... 134
Argument as local variable ... 89
'argument' defined ... 8
'argument list' defined ... 20
Argument, wrong type of ... 9
Arguments ... 8
Arguments' data types ... 8
Arguments, variable number of ... 9
Asterisk for read-only buffer ... 44
Auto Fill mode turned on ... 144
autoload ... 147
Automatic mode selection ... 144
Axis, print horizontal ... 179
Axis, print vertical ... 174

B

Backtrace buffer for debugging ... 156
beginning-of-buffer ... 47
'bind' defined ... 11
'body' defined ... 19
Body of graph ... 133
Buffer size ... 18
Buffer, history of word ... 16
buffer-file-name ... 15
buffer-menu, bound to key ... 146
buffer-name ... 15
Bug, most insidious type ... 189
Byte compiling ... 5

C

C language primitives ... 19
C, a digression into ... 69
'call' defined ... 17
cancel-debug-on-entry ... 157
car, introduced ... 57
cdr, introduced ... 57
Changing a function definition ... 21
Clipping text ... 63
Code installation ... 24
'command' defined ... 15
Comments in Lisp code ... 21
Common Lisp ... x
compare-windows ... 145
concat ... 8
cond ... 93
Conditional 'twixt Emacs 18 and 19 ... 149
Conditional with if ... 26
cons, introduced ... 58
copy-region-as-kill ... 71
copy-to-buffer ... 43
Count words recursively ... 113
count-words-in-defun ... 122
count-words-region ... 109
Counting ... 13
Counting words in a defun ... 119, 121
current-buffer ... 16
Customizing your '.emacs' file ... 141
Cutting and storing text ... 63

D

Data types ... 8
debug ... 155
debug-on-entry ... 156
debug-on-error ... 155
debug-on-quit ... 158
debugging ... 155
default-mode-line-format ... 152
'default.el' init file ... 141
Definition installation ... 20
Definition writing ... 19
Definition, how to change ... 21
defun ... 19
defvar ... 70
delete-region ... 69
Deleting text ... 63

`describe-function` ... 36
`describe-function`, introduced ... 35
Digression into C ... 69
`directory-files` ... 129
Division ... 49
Duplicated words function ... 163

E

`edebug` ... 158
`edit-options` ... 142
`edit-options`, introduced ... 70
Else ... 28
Emacs version, choosing ... 149
`eobp` ... 102
`eq` ... 32
`eq` (example of use) ... 72
`equal` ... 32
Erasing text ... 63
`error` ... 166
Error for symbol without value ... 7
Error message generation ... 3
`etags` ... 105
'`evaluate`' defined ... 3
Evaluating inner lists ... 6
Evaluation ... 5
Evaluation practice ... 15
'`expression`' defined ... 1

F

Falsehood and truth in Lisp ... 29
Find a File ... 123
Find function documentation ... 35
Find source of function ... 35
`find-tags` ... 35
Flowers in a field ... 1
Focusing attention (narrowing) ... 53
'`form`' defined ... 1
Formatting convention ... 39
Formatting help ... 2
`forward-paragraph` ... 99
`forward-sentence` ... 96
'`function`' defined ... 4
'`function definition`' defined ... 19
Function definition installation ... 20
Function definition writing ... 19
Function definition, how to change ... 21
Functions, primitive ... 19

G

Generate an error message ... 3
Getting a buffer ... 16
Global set key ... 145
`global-set-key` ... 145
`global-unset-key` ... 146
Graph prototype ... 133
Graph, printing all ... 183
`graph-body-print` ... 137
`graph-body-print` Final version ... 184

H

Handling the kill ring ... 165
Help typing lists ... 2
Horizontal axis printing ... 179

I

`if` ... 26
'`if-part`' defined ... 26
`indent-tabs-mode` ... 145
Indentation for formatting ... 39
Initialization file ... 141
Initializing a variable ... 70
Inner list evaluation ... 6
`insert-buffer` ... 44
`insert-buffer-substring` ... 38
Insidious type of bug ... 189
Install a Function Definition ... 20
Install code permanently ... 24
`int-to-string` ... 176
`interactive` ... 21
'`interactive function`' defined ... 15
Interactive functions ... 21
Interactive options ... 23
`interactive`, example use of ... 44
Interpreter, Lisp, explained ... 3
Interpreter, what it does ... 4

K

Key setting globally ... 145
Key unbinding ... 146
Keymaps ... 149
Keyword ... 48
Kill ring handling ... 165
Kill ring overview ... 81
`kill-append` ... 72
`kill-region` ... 68
Killing text ... 63

L

`lambda` ... 187
`length` ... 59
`lengths-list-many-files` ... 126
`let` ... 24
`let` expression sample ... 25
`let` expression, parts of ... 25
`let` variables uninitialized ... 26
Library, as term for 'file' ... 35
`line-to-top-of-window` ... 148
Lisp Atoms ... 1
Lisp history ... x
Lisp interpreter, explained ... 3
Lisp interpreter, what it does ... 4
Lisp Lists ... 1
`list-buffers`, rebound ... 146
`list-options` ... 142

Lists in a computer 77
`load-library` 147
`load-path` 147
Loading files 146
'`local variable`' defined 24
Local variables list, per-buffer, ... 144
Location of point 18
`looking-at` 102
Loops 83
Loops and recursion 83

M

Maclisp x
Mail aliases 145
`make-string` 177
`mapcar` 188
`mark` 30
`mark-whole-buffer` 37
`match-beginning` 103
`max` 134
`message` 10
`min` 134
Mode line format 152
Mode selection, automatic 144
Motion by sentence and paragraph 95

N

Narrowing 53
'`narrowing`' defined 18
`nil` 29
`nil`, history of word 16
`nreverse` 130
`nthcdr` 60, 71
'`null list`' defined 1

O

`occur` 146
`optional` 48
Optional arguments 48
Options for `interactive` 23
Options introduced 70
Options, easily settable 142
`or` 46
`other-buffer` 16

P

Paragraphs, movement by 95
Parts of `let` expression 25
Passing information to functions 8
Pasting text 81
Patterns, searching for 95
Per-buffer, local variables list 144
Permanent code installation 24
`point` 30
'`point`' defined 18
Point location 18
Point, mark, buffer preservation 30
Practicing evaluation 15
Preserving point, mark, and buffer 30
Primitive functions 19
Primitives written in C 19
Print horizontal axis 179
Print vertical axis 174
`print-elements-of-list` 84
`print-elements-recursively` 91
`print-graph` Final version 185
`print-graph` varlist 174
`print-X-axis` 182
`print-X-axis-numbered-line` 181
`print-X-axis-tic-line` 181
`print-Y-axis` 178
`print-Y-axis` Final version 184
Printing the whole graph 183
`prog1` 102
`progn` 65
Program, running one 3
Prototype graph 133

R

`re-search-forward` 96
Read-only buffer 44
Readying a graph 133
Rebinding keys 149
Recursion 90
Recursion and loops 83
`recursive-count-words` 117
`recursive-graph-body-print` 138
`recursive-lengths-list-many-files` 127
Recursively counting words 113
`regexp-quote` 101
Region, what it is 30
Regular expression searches 95
Regular expressions for word counting 109
Remainder function, `%` 175
Repetition (loops) 83
Repetition for word counting 109
Retrieving text 81
`reverse` 131
Ring, making a list like a 165
`rotate-yank-pointer` 81, 165
Run a program 3

S

Sample `let` expression 25
`save-excursion` 30
`save-restriction` 53
`search-forward` 65
Searches, illustrating 95
`sentence-end` 95
Sentences, movement by 95
`set` 11
`set-buffer` 17
`setcar` 61
`setcdr` 61

`setq` ... 12
Setting a key globally ... 145
Setting value of variable ... 11
'`side effect`' defined ... 5
Simple extension in '`.emacs`' file ... 148
`simplified-beginning-of-buffer` ... 35
'`site-init.el`' init file ... 141
'`site-load.el`' init file ... 141
Size of buffer ... 18
`sort` ... 128
Source level debugger ... 158
Special form ... 5
Special form of `defun` ... 19
Storing and cutting text ... 63
'`string`' defined ... 2
`switch-to-buffer` ... 17
Switching to a buffer ... 17
Symbol names ... 4
Symbol without value error ... 7
Symbolic expressions, introduced ... 1
Syntax categories and tables ... 120

T

Tabs, preventing ... 145
'`TAGS`' file, create own ... 105
TAGS table, specifying ... 35
Text between double quotation marks ... 2
Text Mode turned on ... 144
Text retrieval ... 81
`the-the` ... 163
'`then-part`' defined ... 26
`top-of-ranges` ... 129
`triangle-bugged` ... 155
`triangle-recursively` ... 91
Truth and falsehood in Lisp ... 29
Types of data ... 8

U

Unbinding key ... 146
Uninitialized `let` variables ... 26

V

Variable initialization ... 70
Variable number of arguments ... 9
Variable, setting value ... 11
Variables ... 6
'`varlist`' defined ... 25
Version of Emacs, choosing ... 149
Vertical axis printing ... 174

W

`what-line` ... 54
`while` ... 83
Whitespace in lists ... 2
Whole graph printing ... 183
Widening ... 53
Widening, example of ... 54
Word counting in a `defun` ... 119
Words and symbols in defun ... 119
Words, counted recursively ... 113
Words, duplicated ... 163
Writing a function definition ... 19
Wrong type of argument ... 9

X

X axis printing ... 179
`X-axis-element` ... 181

Y

Y axis printing ... 174
`Y-axis-column` ... 177
`Y-axis-column` Final version ... 183
`Y-axis-label-spacing` ... 176
`Y-axis-tic` ... 176
`yank` ... 81, 170
`yank-pop` ... 172

Z

`zap-to-char` ... 63
`zerop` ... 166

About the Author

(訳註：以下は原文をそのまま載せておくことにする。)

Robert J. Chassell has worked with GNU Emacs since 1985. He writes and edits, teaches Emacs and Emacs Lisp, and is a director and the Secretary/Treasurer of the Free Software Foundation, Inc. He has an abiding interest in social and economic history and flies his own airplane.

Short Contents

序文 ix
1 List 処理 1
2 実際の評価の仕方 15
3 関数定義の書き方 19
4 バッファに関する幾つかの関数 35
5 もう少し複雑な関数 43
6 ナローイングとワイドニング 53
7 `car`, `cdr`, `cons`: 基本関数 57
8 テキストの切取りと保存 63
9 リストはどのように実装されているか 77
10 テキストのヤンク 81
11 ループと再帰 83
12 正規表現の検索 95
13 カウント: 繰り返しと正規表現 109
14 `defun` 内の単語のカウント 119
15 グラフを描く準備 133
16 ‘`.emacs`’ ファイル 141
17 デバッグ 155
18 まとめ 161
Appendix A 関数 `the-the` 163
Appendix B Kill リングの扱い 165
Appendix C ラベルと軸が付いたグラフ 173
Index 193

Table of Contents

序文 ix
これは誰のために書かれたものか ix
Lisp の歴史 x
初心者の人へ x
謝辞 xi
訳者まえがき xi

1 List 処理 1
1.1 Lisp のリスト 1
1.1.1 Lisp のアトム 1
1.1.2 リストの中の空白 2
1.1.3 GNU Emacs によるリストのタイプの支援 2
1.2 プログラムの実行 3
1.3 エラーメッセージの出力 3
1.4 シンボルの名前と関数定義 4
1.5 Lisp インタプリタ 4
1.5.1 バイトコンパイル 5
1.6 評価 5
1.6.1 内部のリストの評価 6
1.7 変数 6
1.7.1 値のないシンボルに対するエラーメッセージ 7
1.8 引数 8
1.8.1 引数のデータ型 8
1.8.2 引数には変数の値やリストも使える 9
1.8.3 可変な数の引数 9
1.8.4 関数に間違った型の引数を与えると 9
1.8.5 関数 message 10
1.9 変数の値の設定 11
1.9.1 set の利用 11
1.9.2 setq の利用 12
1.9.3 カウント 13
1.10 まとめ 13
1.11 練習問題 14

2 実際の評価の仕方 15
2.1 バッファの名前 15
2.2 バッファの獲得 16
2.3 バッファ間の移動 17
2.4 バッファのサイズとポイントの位置 18
2.5 練習問題 18

3 関数定義の書き方 19
3.1 特殊形式 defun 19
3.2 関数定義のインストール 20
3.2.1 関数定義の変更 21
3.3 関数をインタラクティブにする 21
3.3.1 インタラクティブな multiply-by-seven 22
3.4 interactive の他のオプション 23
3.5 コードをずっとインストールしておくには 24
3.6 let 24
3.6.1 let 式の構成部分 25
3.6.2 let 式の例 25
3.6.3 let 式の変数宣言の中で初期値を設定しなかった場合 26
3.7 特殊形式 if 26

3.7.1 関数 type-of-animal の詳細 27
3.8 If–then–else 式 28
3.9 Lisp における真と偽 29
3.10 save-excursion 30
3.10.1 save-excursion 式のテンプレート 31
3.11 復習 31
3.12 練習問題 33

4 バッファに関する幾つかの関数 35
4.1 情報の探し方 35
4.2 簡略版 beginning-of-buffer の定義 35
4.3 mark-whole-buffer の定義 37
4.3.1 mark-whole-buffer の本体 37
4.4 append-to-buffer の定義 38
4.4.1 インタラクティブ式 append-to-buffer 38
4.4.2 append-to-buffer の本体 39
4.4.3 append-to-buffer の中の save-excursion 39
4.5 復習 41
4.6 練習問題 42

5 もう少し複雑な関数 43
5.1 copy-to-buffer の定義 43
5.2 insert-buffer の定義 44
5.2.1 insert-buffer の中のインタラクティブ式 44
書き込み不可のバッファ 44
インタラクティブ式の中の 'b' 44
5.2.2 関数 insert-buffer の本体部分 45
5.2.3 or の代わりに if を使った insert-buffer 45
5.2.4 本体部分の or 46
5.2.5 insert-buffer の中の let 式 46
5.3 beginning-of-buffer の完全な定義 47
5.3.1 省略可能な引数 48
5.3.2 引数付きで呼び出された場合の beginning-of-buffer 48
大きなバッファの場合 49
小さなバッファの場合 49
5.3.3 完全版 beginning-of-buffer 50
5.4 復習 51
5.5 &optional 引数の練習問題 51

6 ナローイングとワイドニング 53
6.1 特殊形式 save-restriction 53
6.2 what-line 54
6.3 ナローイングの練習問題 55

7 car, cdr, cons: 基本関数 57
7.1 car と cdr 57
7.2 cons 58
7.2.1 リストの長さを調べる: length 59
7.3 nthcdr 60
7.4 setcar 61
7.5 setcdr 61
7.6 練習問題 62

8 テキストの切取りと保存 63
8.1 zap-to-char 63
8.1.1 interactive 式 64
8.1.2 zap-to-char の本体部分 64
8.1.3 関数 search-forward 65
8.1.4 関数 progn 65

8.1.5 zap-to-char についての総括 66
8.1.6 Version 18 での実装 66
progn 式の本体 67
8.2 kill-region 68
8.3 delete-region: ちょっと脱線して C の話を 69
8.4 defvar を用いた変数の初期化 70
8.5 copy-region-as-kill 71
8.5.1 copy-region-as-kill の本体部分 71
関数 kill-append 72
copy-region-as-kill の else-part 73
8.6 復習 75
8.7 検索についての練習問題 76

9 リストはどのように実装されているか 77
9.1 練習問題 80

10 テキストのヤンク 81
10.1 Kill リングについての概観 81
10.2 変数 kill-ring-yank-pointer 81
10.3 yank と nthcdr についての練習問題 82

11 ループと再帰 83
11.1 while 83
11.1.1 while ループとリスト 83
11.1.2 リストを使ったループの例: print-elements-of-list 84
11.1.3 増加するカウンタを使ったループ 85
増加カウンタの例 85
関数定義の各部分 86
各部分の総合 87
11.1.4 減少するカウンタを使ったループ 88
減少するカウンタを使った例 88
関数定義の各部分 89
各部分の総合 89
11.2 再帰 90
11.2.1 List を使った再帰 90
11.2.2 カウンタの代わりに再帰を使う 91
引数 3 の場合 92
11.2.3 cond を使った再帰の例 93
11.3 ループについての練習問題 94

12 正規表現の検索 95
12.1 sentence-end の正規表現 95
12.2 関数 re-search-forward 96
12.3 forward-sentence 96
while ループ 98
正規表現の検索 99
12.4 forward-paragraph: 関数の金脈 99
let* 式 100
前方に移動する場合の while ループ 101
パラグラフとパラグラフの間での動作 102
パラグラフの内部での動作 103
Fill prefix が無い場合 103
Fill prefix がある場合 104
まとめ 104
12.5 自分自身の ‘TAGS’ ファイルの作成 105
12.6 復習 106
12.7 re-search-forward についての練習問題 107

13 カウント: 繰り返しと正規表現 109
13.1 関数 count-words-region 109
13.1.1 count-words-region の空白文字に関するバグ 111
13.2 再帰を使った単語数のカウント 113
13.3 練習問題: 句読点のカウント 118

14 defun 内の単語のカウント 119
14.1 何を数えればよいか？ 119
14.2 単語やシンボルは何から構成されているか 120
14.3 関数 count-words-in-defun 121
14.4 一つのファイルにある複数の defun を数える 123
14.5 ファイルを見つける 123
14.6 lengths-list-file についての詳細 124
14.7 異なるファイルの中の定義を数える 125
14.7.1 関数 append 126
14.8 異なるファイルの定義を再帰を使って数える 127
14.9 データをグラフに表示するための準備 128
14.9.1 リストのソート 128
14.9.2 ファイルのリストの作成 129

15 グラフを描く準備 133
15.1 関数 graph-body-print 137
15.2 関数 recursive-graph-body-print 138
15.3 軸を表示する 139
15.4 練習問題 140

16 '.emacs' ファイル 141
16.1 サイトごとの初期化ファイル 141
16.2 あるセッションでの変数の設定 142
16.3 '.emacs' の書き方 142
16.4 Text モードと Auto Fill モード 143
16.5 メールのエイリアス 144
16.6 Indent Tabs モード 145
16.7 幾つかのキーバインディング 145
16.8 ファイルのロード 146
16.9 オートロード 147
16.10 ちょっとした拡張: line-to-top-of-window 148
16.11 キーマップ 149
16.12 X11 でのカラー表示 150
16.13 V19 での雑多な事柄 151
16.14 モード行の修正 152

17 デバッグ 155
17.1 debug 155
17.2 debug-on-entry 156
17.3 debug-on-quit と (debug) 158
17.4 ソースレベルのデバッガ edebug 158
17.5 デバッグについての練習問題 159

18 まとめ 161

Appendix A 関数 the-the 163

Appendix B　Kill リングの扱い 165
B.1 関数 rotate-yank-pointer 165
B.1.1 rotate-yank-pointer の本体 166
if 式の else-part 166
剰余関数 % 168
rotate-yank-pointer の中での % の利用 168
最後の要素を指している場合 169
B.2 yank 170
引数の渡し方 170
負の引数を渡すとどうなるか 171
B.3 yank-pop 172

Appendix C　ラベルと軸が付いたグラフ 173
C.1 print-graph の変数リスト 174
C.2 関数 print-Y-axis 174
C.2.1 寄り道: 剰余の計算 175
C.2.2 Y 軸の要素の構成 176
C.2.3 Y 軸全体の構成 177
C.2.4 print-Y-axis 最終版 178
C.3 関数 print-X-axis 179
C.3.1 X 軸の目盛記号 179
C.4 グラフ全体の表示 183
C.4.1 print-graph のテスト 185
C.4.2 単語やシンボルの数のグラフ化 186
lambda 式 187
関数 mapcar 188
まだバグがある 189
C.4.3 表示されたグラフ 191

Index 193