GNU Emacs Lisp リファレンス・マニュアル

Unix ユーザ向け
GNU Emacs バージョン 19
原作 2.4 版、 1995 年 6 月
日本語ドラフト 1.0 版、 1995 年 12 月

**by Bil Lewis, Dan LaLiberte, Richard Stallman
and the GNU Manual Group**

Japanese DRAFT Revision 1.0, Dec 1995
for original Edition 2.4
Revised for Emacs Version 19.29,
June, 1995.

Cover art by Etienne Suvasa.

GNU 一般公有使用許諾

1991 年 6 月, バージョン 2

はじめに

ほとんどのソフトウェアの使用許諾は、ソフトウェアを共有し、変更するユーザの自由を奪うことを意図しています。それに対して、我々の GNU 一般公有使用許諾は、フリー・ソフトウェアを共有したり変更する自由をユーザに保証するためのもの、即ちフリー・ソフトウェアがそのユーザ全てにとってフリーであることを保証するためのものです。本使用許諾は、 Free Software Foundation のほとんど全てのソフトウェアに適用されるだけでなく、プログラムの作成者が本使用許諾に依るとした場合のそのプログラムにも適用することができます。 (その他の Free Software Foundation のソフトウェアのいくつかは、本許諾書ではなく、 GNU ライブラリ一般公有使用許諾で保護されます。) あなたは自分のプログラムにもこれを適用できます。

我々がフリー・ソフトウェアについて言う場合は自由のことに言及しているのであって、価格のことではありません。我々の一般公有使用許諾の各条項は、次の事柄を確実に実現することを目的として立案されています。

フリー・ソフトウェアの複製物を自由に頒布できること (そして、望むならあなたのこのサービスに対して対価を請求できること)。

ソース・コードを実際に受け取るか、あるいは、希望しさえすればそれを入手することが可能であること。

入手したソフトウェアを変更したり、新しいフリー・プログラムの一部として使用できること。

以上の各内容を行なうことができるということをユーザ自身が知っていること。

このようなユーザの権利を守るために、我々は、何人もこれらの権利を否定したり、あるいは放棄するようにユーザに求めることはできないという制限条項を設ける必要があります。これらの制限条項は、ユーザが、フリー・ソフトウェアの複製物を頒布したり変更しようとする場合には、そのユーザ自身が守るべき義務ともなります。

例えば、あなたがフリー・ソフトウェアの複製物を頒布する場合、有償か無償かにかかわらず、あなたは自分の持っている権利を全て相手に与えなければなりません。あなたは、相手もまたソース・コードを受け取ったり入手できるということを認めなければなりません。さらにあなたは、彼らが自分たちの権利を知るように、これらの条項を知らしめなければなりません。

我々は次の 2 つの方法でユーザの権利を守ります。 (1) ソフトウェアに著作権を主張し、 (2) 本使用許諾の条項の下でソフトウェアを複製・頒布・変更する権利をユーザに与えます。

また、各作成者や我々自身を守るために、本フリー・ソフトウェアが無保証であることを全ての人々が了解している必要があります。さらに、他の誰かによって変更されたソフトウェアが頒布された場合、受領者はそのソフトウェアがオリジナル・バージョンではないということを知らされる必要があります。それは、他人の関与によって原開発者に対する評価が影響されないようにするためです。

最後に、どのフリー・プログラムもソフトウェア特許に絶えず脅かされています。我々は、フリー・プログラムの再頒布者が個人的に特許権を取得し、事実上そのプログラムを自分の財産にしてしまうという危険を避けたいと願っています。これを防ぐために我々は、いずれの特許も、誰でも自由に使用できるように使用許諾されるべきか、あるいは何人に対しても全く使用させないかの、いずれかにすべきであることを明らかにしてきました。

複写・頒布・変更に対する正確な条項と条件を次に示します。

複製、頒布、変更に関する条項と条件

0. 本使用許諾は、本一般公有使用許諾の各条項に従って頒布されるという著作権者からの告知文が表示されているプログラムやその他の作成物に適用されます。以下において「プログラム」とは、そのようなプログラムや作成物を指すものとし、また、「プログラム生成物」とは、上述した「プログラム」自身、または、著作権法下における全ての派生物; すなわち、その「プログラム」の全部又は一部を、そのまま又は変更して、且つ / 又は他の言語に変換して、内部に組み込んだ作成物を意味します。(以下、言語変換は「変更」という用語の中に無条件に含まれるものとします。) 本使用許諾によって許諾を受ける者を「あなた」と呼びます。

   複製、頒布、変更以外の行為は本使用許諾の対象としません。それらは本使用許諾の範囲外です。「プログラム」を実行させる行為に関して制約はありません。「プログラム」の出力は、(「プログラム」を実行させて作成させたかどうかとは無関係に) その内容が「プログラム生成物」である場合に限り本使用許諾の対象となります。これが当てはまるかどうかは、「プログラム」が何をするものかに依ります。

1. あなたは、どのような媒体上へ複製しようとする場合であっても、入手した「プログラム」のソース・コードをそのままの内容で複写した上で適正な著作権表示と保証の放棄を明確、且つ適正に付記する場合に限り、複製又は頒布することができます。その場合、本使用許諾及び無保証に関する記載部分は、全て元のままの形で表示してください。また、「プログラム」の頒布先に対しては、「プログラム」と共に本使用許諾書の写しを渡してください。

   複製物の引き渡しに要する実費は請求することができます。また、あなた独自の保証を行なう場合はそれを有償とすることができます。

2. 次の各条件を全て満たしている限り、あなたは、「プログラム」又はその一部分を変更して「プログラム生成物」とすることができ、さらに、変更版や右作成物を上記第 1 項に従って複製又は頒布することもできます。

   a. ファイルを変更した旨とその変更日とを、変更したファイル上に明確に表示すること。

b. 変更したか否かを問わず、凡そ「プログラム」又はその一部分を内部に組み込んでいるか又はそれから派生した生成物を頒布する場合には、その全体を本使用許諾の条項に従って第三者へ無償で使用許諾すること。

c. 変更したプログラムが実行時に通常の対話的な方法でコマンドを読むようになっているとすれば、最も普通の方法で対話的にそのプログラムを実行する時に、次の内容を示す文言がプリンタへ印字されるか、或いは画面に表示されること。適切な著作権表示。無保証であること (あなたが独自に保証する場合は、その旨)。頒布を受ける者も、本使用許諾と同一の条項に従って「プログラム」を再頒布できること。頒布を受ける者が本使用許諾書の写しを参照する方法。(例外として、「プログラム」自体は対話的であっても起動時の文言を通常は印字しないのならば、あなたの「プログラム生成物」はこのような文言を印字する必要はありません。)

これらの要件は変更された作成物にも全て適用されます。その変更版の或る部分が「プログラム」の派生物ではなく、しかもそれ自体独立で異なる作成物だと合理的に考えられる場合、あなたがそれらを別の作成物として頒布した時は、本使用許諾とその条項はそれらの部分には適用されません。しかし、それらを「プログラム生成物」の一部として頒布する場合は、全体が本使用許諾の条項に従って頒布されなければならず、使用許諾を受ける他の全ての者に対する許諾もプログラム全体にわたって与えられなければならず、結果として、誰が書いたかにかかわらず、全ての部分に本使用許諾が適用されなければなりません。

このように、本条項の意図するところは、完全にあなたによって書かれた作成物について、権利を要求したり、あなたと権利関係を争うことではありません。むしろその目的は、作成物が「プログラム生成物」である場合にその派生物や集合物の頒布を規制することにあります。

さらに、「プログラム」(又は「プログラム生成物」) と「プログラム生成物」とはならない他のプログラムとを、単に保管や頒布のために同一の媒体上にまとめて記録したとしても、本使用許諾は他のプログラムには適用されません。

3. あなたは、以下のうちいずれか 1 つを満たす限り、上記第 1 項及び第 2 項に従って「プログラム」(又は、上記第 2 項で言及している「プログラム生成物」) をオブジェクト・コード又は実行可能な形式で複製及び頒布することができます。

a. 対応する機械読み取り可能なソース・コード一式を一緒に引き渡すこと。その場合、そのソース・コードの引き渡しは上記第 1 項及び第 2 項に従って、通常ソフトウェアの交換に用いられる媒体で行なわれること。

b. 少なくとも 3 年間の有効期間を定め、且つその期間内であれば対応する機械読み取り可能なソース・コード一式の複製を、ソース頒布に関わる実費以上の対価を要求せずに提供する旨、及びその場合には上記第 1 項及び第 2 項に従って、通常ソフトウェアの交換に用いられる媒体で提供される旨を記載した書面を、第三者に一緒に引き渡すこと。

c. 対応するソース・コード頒布の申し出に際して、あなたが得た情報を一緒に引き渡すこと 。(この選択肢は、営利を目的としない頒布であって、且つあなたが上記の (b) 項に基づいて、オブジェクト・コード或いは実行可能形式のプログラムしか入手していない場合に限り適用される選択項目です。)

なお、ソース・コードとは、変更作業に適した記述形式を指します。また、実行可能形式のファイルに対応するソース・コード一式とは、それに含まれる全モジュールに対応する全てのソース・コード、及びあらゆる関連のインタフェース定義ファイル、及び実行を可能にするコンパイルとインストールの制御に関する記述を指します。特別な例外として、実行可能なファイルが動作するオペレーティング・システムの主要な構成要素 (コンパイラ、カーネルなど) と共に (ソース・コード又はバイナリのどちらかで) 頒布されているものについては、その構成要素自体が実行形式に付随していない場合に限り、頒布されるソース・コードに含める必要はありません。

実行可能形式またはオブジェクト・コードの頒布が、指示された場所からの複製のためのアクセス権の賦与である場合、同じ場所からのソース・コードの複製のための同等なアクセス権を賦与すれば、たとえ第三者にオブジェクト・コードと共にソースの複製を強いなくとも、ソース・コードを頒布したものとみなします。

4. 本使用許諾が明示的に許諾している場合を除き、あなたは、「プログラム」を複製、変更、サブライセンス、頒布することができません。本使用許諾に従わずに「プログラム」を複製、変更、サブライセンス、頒布しようとする行為は、それ自体が無効であり、且つ、本使用許諾があなたに許諾している「プログラム」の権利を自動的に消滅させます。その場合、本使用許諾に従ってあなたから複製物やその権利を得ている第三者は、本使用許諾に完全に従っている場合に限り、引続き有効な使用権限を持つものとします。

5. あなたはまだ同意の印として署名していないので、本使用許諾を受け入れる必要はありません。しかし、あなたに「プログラム」又はその派生物を変更又は再頒布する許可を与えるものは本使用許諾以外にはありません。これらの行為は、あなたがもし本使用許諾を受け入れないのであれば、法律によって禁じられます。従って、あなたが「プログラム」(又は「プログラム生成物」) の変更又は頒布を行えば、それ自体であなたは本使用許諾を受け入れ、且つ、「プログラム」又はその「プログラム生成物」の複製、頒布、変更に関するこれらの条項と条件の全てを受け入れたことを示します。

6. あなたが「プログラム」(又はその「プログラム生成物」) を再頒布すると自動的に、その受領者は、元の使用許諾者から、本使用許諾の条項に従って「プログラム」を複製、頒布、変更することを内容とする使用許諾を受けたものとします。あなたは、受領者に許諾された権利の行使について、さらに制約を加えることはできません。あなたには、第三者に本使用許諾の受け入れを強いる責任はありません。

7. 裁判所の判決、又は特許侵害の申し立て、又は (特許問題に限らない) 何らかの理由の結果として、あなたに課せられた条件が本使用許諾と相入れないものであったとしても (裁判所の命令、契約、その他によるものであれ)、本使用許諾の条件が免除されるものではありません。本使用許諾による責務と、その他の何らかの関連責務を同時に満たす態様で頒布することができないならば、あなたは「プログラム」を全く頒布してはいけません。例えば、特許権の内容が、あなたから直接又は間接に複製を受け取った全ての人に使用料のないプログラムの再頒布を許さないものであれば、あなたがかかる特許上の要請と本使用許諾の両方を満足させる方法は、「プログラム」の頒布を完全に断念することだけです。

   本条項の或る部分が何らかの特別な状況下で無効または適用不可能になった場合、本条項のその他の残りの部分が適用されるように意図されており、また、本条項は全体としてその他の状況に当てはまるように意図されています。

本条項の目的は、特許やその他の財産権を侵害したり、そのような権利に基づく主張の妥当性を争うようにあなたに勧めることではありません。本条項の唯一の目的は、フリー・ソフトウェアの頒布システムの完全性を守ることで、それは公有使用許諾の実践によって履行されます。多くの人々が、このシステムの一貫した適用を信頼して、このシステムを通じて頒布されている幅広い範囲のソフトウェアに惜しみない貢献をしてくれました。作成者や寄贈者が他の何らかのシステムを通じてソフトウェアを頒布したいと決めることは彼らの自由意志であり、使用許諾を受ける者はその選択を強いることはできません。

本条項は、本使用許諾の他の条項の意味内容が何であるかを完全に明らかにすることを意図しています。

8. 「プログラム」の頒布・使用が、ある国において特許又は著作権で保護されたインタフェースのどちらかで制限される場合、「プログラム」を本使用許諾下においた原著作権保持者は、その国を除外する旨の明示的な頒布地域制限を加え、それ以外の (除外されない) 国に限定して頒布が許されるようにすることができます。そのような場合、その制限を本使用許諾の本文にあたかも書かれているかのように本使用許諾の中に組み入れられるものとします。

9. Free Software Foundation は随時、本一般公有使用許諾の改訂版、又は新版を公表することがあります。そのような新しいバージョンは、現行のバージョンと基本的に変わるところはありませんが、新しい問題や懸案事項に対応するために細部では異なるかもしれません。

   各バージョンは、バージョン番号によって区別します。「プログラム」中に本使用許諾のバージョン番号の指定がある場合は、その指定されたバージョンか、又はその後に Free Software Foundation から公表されているいずれかのバージョンから 1 つを選択して、その条項と条件に従ってください。「プログラム」中に本使用許諾のバージョン番号の指定がない場合は、 Free Software Foundation が公表したどのバージョンでも選択することができます。

10. 「プログラム」の一部を頒布条件の異なる他のフリー・プログラムに組み込みたい場合は、その開発者に書面で許可を求めてください。 Free Software Foundation が著作権を持っているソフトウェアについては、 Free Software Foundation へ書面を提出してください。このような場合に対応するために我々は例外的処理をすることもありますが、その判断基準となるのは、次の 2 つの目標の実現に合致するか否かという点です。即ち、 1 つは我々のフリー・ソフトウェアの全ての派生物をフリーな状態に保つことであり、もう 1 つはソフトウェアの共有と再利用とを広く促進させることです。

無保証

11. 「プログラム」は無償で使用許諾されますので、適用法令の範囲内で、「プログラム」の保証は一切ありません。著作権者やその他の第三者は全く無保証で「そのまま」の状態で、且つ、明示か暗黙であるかを問わず一切の保証をつけないで提供するものとします。ここでいう保証とは、市場性や特定目的適合性についての暗黙の保証も含まれますが、それに限定されるものではありません。「プログラム」の品質や性能に関する全てのリスクはあなたが負うものとします。「プログラム」に欠陥があるとわかった場合、それに伴う一切の派生費用や修理・訂正に要する費用は全てあなたの負担とします。

12. 適用法令の定め、又は書面による合意がある場合を除き、著作権者や上記許諾を受けて「プログラム」の変更・再頒布を為し得る第三者は、「プログラム」を使用したこと、または使用できないことに起因する一切の損害について何らの責任も負いません。著作権者や前記の第三者が、そのような損害の発生する可能性について知らされていた場合でも同様です。なお、ここでいう損害には通常損害、特別損害、偶発損害、間接損害が含まれます (データの消失、又はその正確さの喪失、あなたや第三者が被った損失、他のプログラムとのインタフェースの不適合化、等も含まれますが、これに限定されるものではありません)。

以上、条項と条件

注意

英文文書 (GNU General Public License) を正式文書とする。この和文文書は弁護士の意見を採り入れて、できるだけ正確に英文文書を翻訳したものであるが、法律的に有効な契約書ではない。

和文文書自体の再配布に関して

いかなる媒体でも次の条件がすべて満たされている場合に限り、本和文文書をそのまま複写し配布することを許可する。また、あなたは第三者に対して本許可告知と同一の許可を与える場合に限り、再配布することが許可されています。

受領、配布されたコピーに著作権表示および本許諾告知が前もって載せられていること。

コピーの受領者がさらに再配布する場合、その配布者が本告知と同じ許可を与えていること。

和文文書の本文を改変しないこと。

あなたの新しいプログラムにこれらの条項を適用する方法

あなたが新しくプログラムを作成し、それを公用に供したい場合は、プログラムをフリー・ソフトウェアにして、全ての人々が以上の各条項に従ってこれを再頒布や変更をすることができるようにするのが最良の方法です。

そうするためには、プログラムに以下の表示をしてください。その場合、無保証であるということを最も効果的に伝えるために、ソース・ファイルの冒頭にその全文を表示すれば最も安全ですが、その他の方法で表示する場合でも、「著作権表示」と全文を読み出す為のアドレスへのポインタだけはファイル上に表示しておいてください。

```
プログラム名とどんな動作をするものかについての簡単な説明の行
Copyright (C) 19 ○○  著作権者名

本プログラムはフリー・ソフトウェアです。あなたは、 Free Software
Foundation が公表した GNU 一般公有使用許諾の「バージョン 2」或いはそれ
以降の各バージョンの中からいずれかを選択し、そのバージョンが定める条項に
従って本プログラムを再頒布または変更することができます。

本プログラムは有用とは思いますが、頒布にあたっては、市場性及び特定目的適
合性についての暗黙の保証を含めて、いかなる保証も行ないません。詳細につい
ては GNU 一般公有使用許諾書をお読みください。

あなたは、本プログラムと一緒に GNU 一般公有使用許諾の写しを受け取ってい
るはずです。そうでない場合は、 Free Software Foundation, Inc., 675 Mass
Ave, Cambridge, MA 02139, USA へ手紙を書いてください。
```

また、ユーザが電子メイルや書信であなたと連絡をとる方法についての情報も書き添えてください。

プログラムが対話的に動作する場合は、対話モードで起動した時に次のような短い告知文が表示されるようにしてください。

```
Gnomovision バージョン 69, Copyright (C) 19 ○○ 著作権者名
Gnomovision は完全に無保証です。
詳細は‘show w’とタイプしてください。
これはフリー・ソフトウェアなので、特定の条件の下でこれを再頒布することがで
きます。詳細は‘show c’とタイプしてください。
```

上記の‘show w’や‘show c’は各々、本一般公有使用許諾の関連する部分を表示するコマンドを指します。もちろん、あなたが使うこれらのコマンドは‘show w’や‘show c’といった呼び名でなくても構いません。さらに、それらのコマンドはあなたのプログラムに合わせる為に、マウスでクリックしたりメニュー形式にすることもできます。

また、必要と認めた場合には、あなたの雇い主 (あなたがプログラマとして働いている場合) や在籍する学校から、そのプログラムに対する「著作権放棄」を認めた署名入りの書面を入手してください。ここにその文例を載せます。名前は変えてください。

```
Yoyodyne, Inc. は、 James Hacker が開発したプログラム‘Gnomovision’(コン
パイラにつなげるプログラム) についての著作権法上の全ての権利を放棄する。

Ty Coon の署名, 1 April 1989
Ty Coon, 副社長
```

本一般公有使用許諾は、あなたのプログラムを財産権の対象となっている他のプログラムに組み込むことは認めていません。あなたのプログラムがサブルーチン・ライブラリであって、あ

なたがそのライブラリを財産権の対象となっている他のアプリケーションとリンクさせることによって、さらに有用なものにしようとする場合には、本使用許諾書の代わりに、 GNU ライブラリ一般公有使用許諾書に従ってください。

訳者まえがき

Emacs Lisp バリバリのプログラマ、あるいはこれから Emacs Lisp を勉強しようという、Emacs Lisp 初心者の皆さんへ。

ここに、FSF 一行が日本にやって来る 95 年 12 月に間に合わせるべく作業を行なった、Emacs Lisp Reference Manual 2.4 版 (現時点での、最新版!) の、邦訳をお届けします。これは最新の Emacs バージョン 19(以下、Emacs19 と略) に対応しています。

我々訳者一同、心をこめて、この翻訳作業に携わりました。原文の不明瞭な点、誤りと思われる点は、可能なかぎり FSF に問い合わせて、修正しました (回答をくれた、FSF および RMS に感謝!)。しかし、いかんせん、翻訳素人の悲しさ、不十分なできの部分もあるかと思います。訳者同士の訳語の統一も、可能なかぎり努力しましたが、結局不十分に終りました。監訳者にお迎えした井田先生のご指摘も、FSF 来日に間に合わせるという時間的な制約もあり、満足に活かせなかった部分、多々あります (井田先生ごめんなさい!)。これらの点は非常に心苦しく思っています。

我々は今回の翻訳が不十分なものであると十分に認識していますから、もし、皆さんがこの翻訳に不備な点を見つけた場合、改良した方がいいと感じる点があったら、どうぞ遠慮なくご連絡ください (第 1 章「序文」p.11、1.1 節「警告」p.11 参照)。我々は歓迎します。

今回の翻訳には、おそらく欠陥があります。しかし、ここに、日本語で読める Emacs Lisp のマニュアルがあるのです!我々は、皆さんがこれを使って Emacs Lisp の理解を深め、よりよい Emacs Lisp アプリケーションの開発や改良、既存のアプリケーションの不具合の修正など、さまざまな活動を行なってくれることを期待します。みんなで FSF の GNU 活動を支援しましょう!

日本では、Mule-2 が Emacs19 をベースにしています。したがって、日本の皆さんがこのマニュアルの Emacs Lisp を使用するのは、Mule-2 の上で、ということになるかと思います。ここで、一つ、注意して頂きたいことがあります。

それは、Mule-2 と Emacs19 との相違点です。Mule-2 は、Emacs19 をベースに開発されたので、その Emacs Lisp のほとんどの機能は Emacs19 と同じです。しかし、違うところももちろんあります。多くは、多国語対応のためになされた機能拡張や機能の追加ですが、一部はやむなくなされた既存の Emacs19 の機能や概念の修正です。

Mule-2 と Emacs19 とで、最も異なっているのは「文字 (character)」に関する扱いです。Emacs19 では 'character' は、実はバイトのことを表しています。したがって、Emacs19 を解説した Emacs Lisp Reference Manual での 'character' とは、バイトについて述べていることになります。この翻訳でも、その流れを踏んでいるので、'character' の訳「文字」は、バイトのことを指しています。

しかし、Mule-2 では多国語文字に対応するため、「文字 (character)」は、「多国語文字」に概念拡張されました。これは、「多バイト文字」で表現されるため、Emacs19 が仮定していた「文字 == バイト」の図式は通用しなくなります。たとえば、`forward-char` という関数は、Emacs19 では「指定バイト数前方に、ポイントを進める」という機能ですが、Mule-2 では「指定文字数 (多バイト文字) 前方に、ポイントを進める」という機能に変わります。Mule-2 でバイト単位で進めるには、`forward-byte` を使わなくてはなりません。

よって、本マニュアルで「文字」についての議論が「文字 == バイト」を仮定している場合には、その部分の議論は Mule-2 には通用しなくなります。こういった点にだけは、ご注意ください。

なお、この Mule-2 と Emacs19 との相違については、 Mule-2 附属の、 Info を始めとするドキュメントに説明されています。詳しくはそちらをご覧ください。 Mule-2 で行なわれたさまざまな拡張についても、知ることができます。

さて、この辺で私の作業は終らせて貰うかな。ほかの訳者のみなさん、監訳の先生方、印刷担当の方達も、本当にお疲れ様でした (そろそろ夜が明けてきた)。

1995 年 11 月 5 日午前 6 時
訳者の一人より

1 序文

GNU Emacs テキスト・エディタの大部分は Emacs Lisp というプログラミング言語で書かれています。新しいコードを Emacs Lisp で書いてエディタの拡張部分としてインストールすることができます。しかし、 Emacs Lisp は単なる “拡張言語” 以上のものです。 Emacs Lisp はそれ自身でも完全なコンピュータ・プログラミング言語なのです。ほかのプログラミング言語と同じように使うことができるのです。

Emacs Lisp はエディタ内で使用するために設計されていますから、ファイルや、バッファ、ディスプレイ、サブプロセスなどを扱う機能だけでなく、テキストをスキャンしたりパースしたりするための特別な機能もあります。 Emacs Lisp は編集機能と密接に統合されていますから、編集コマンドは Lisp プログラムからも簡単に呼ぶことのできる関数ですし、カスタマイズ用のパラメータは普通の Lisp 変数です。

本マニュアルでは、 Emacs を編集作業に使用することにかなり精通していることを前提にして、 Emacs Lisp について説明します (このあたりの基本情報については *The GNU Emacs Manual* を参照のこと)。一般的に言って、前の方の章では多くのプログラミング言語にもあるような Emacs Lisp の機能について説明し、後ろの方の章では Emacs Lisp 独特の機能や、編集作業に特に関連する機能について説明します。

本版は第 2.4 版です。

1.1 警告

本マニュアルは数多くの草稿を経ています。だいたい完全ですが、完全無欠というわけではありません。 (たとえばほとんどの個々のモードのように)2 次的であると判断したか、あるいはまだ書かれるべきではないという理由で、カバーされていない話題も少しあります。そういったものを完全に取り扱うことはできないので、数箇所を意図的に手をつけないまま残してあります。これには、 VMS での使用法に関する多くの情報も含まれます。

このマニュアルはカバーしてある範囲内では完全に正確であるべきです。ですから、具体例や説明文から、章節の順番に至るまで、書かれているどのようなことに関しても批判を受け付けます。もし、何かわかりにくいことがあったり、マニュアルでカバーされてないことを知るために、ソースを見なければならなかったり、実験しなければならなかったりするなら、おそらく、マニュアルは修正した方がいいでしょう。ぜひとも教えてください。

マニュアルを使用する際は、後で調べて提出することができるように、訂正部分のあるページをマークするようにお願いします。関数に簡単で現実的な例を思い付いたら、お手数ですが、それを清書して提出してください。ページ番号や章節番号は変わりますから、該当する本文を見つけるのに手間取るかもしれませんので、コメントは適宜、章名、節名、関数名でリファレンス付けしてください。また、版番号も記載しておいてください。

コメントと訂正は下記へメールしてください。[1]

```
bug-lisp-manual@prep.ai.mit.edu
```

[1] 日本語版に関するコメントと訂正は `emacs19-info-j@sramhc.sra.co.jp` へ。

このリストへのメールは、誰かがその訂正を適用すると決定するまで、未読としてためておきます。アップデートするのに数ヵ月、ときには数年かかります。ですから、返答がないということに他意はないと思ってください。メールは、ちゃんと時が来れば作業します。もっと速く Emacs メンテナと連絡を取りたい人は、 `bug-gnu-emacs@prep.ai.mit.edu` にメールしてください。

–Bil Lewis, Dan LaLiberte, Richard Stallman

1.2 Lisp の歴史

Lisp (LISt Processing language リスト処理言語) は、人工知能の研究用にマサチューセッツ工科大学で 1950 年代後半にはじめて開発されました。 Lisp 言語は、編集コマンドの記述のようなほかの目的にも優れた能力を発揮します。

何年もの間に、数十もの Lisp 処理系が開発され、それぞれが独特の表現形式をもつに到りました。その多くは、 MIT の Project MAC で 1960 年代に書かれた Maclisp の影響を受けていました。とうとう、 Maclisp 子孫の処理系開発者たちが集まって Common Lisp と呼ばれる Lisp システムの標準を開発しました。

GNU Emacs Lisp は Maclisp に大きく影響を受けています。 Common Lisp にも少し影響を受けています。 Common Lisp を知っている人は、多くの類似性に気づくと思います。しかし、 GNU Emacs の要求メモリを削減するため、省略されたり単純化された Common Lisp の機能が多くあります。時には、単純化があまりに過激で、 Common Lisp ユーザは非常に混乱するかも知れません。時折り、 GNU Emacs Lisp が Common Lisp とどう違うか指摘することにします。 Common Lisp を知らない人は心配しないでください。本マニュアルは自己充足的ですから。

1.3 慣習

本節では、本マニュアルで使用される表記法の慣習について説明します。本節を飛ばして、後で戻って参照してもよいでしょう。

1.3.1 用語

本マニュアルの中では、 “Lisp リーダ”、 “Lisp プリンタ” という言葉を、 Lisp オブジェクトをテキスト表現から実 Lisp オブジェクトへ変換、逆変換する Lisp 内ルーチンを指すのに使います。詳細は 2.1 節「印字表現とリード構文」 p.19 を参照してください。本マニュアルを読んでいるあなたを、 “プログラマ” と見なし、 “あなた” と呼ぶことにします。 “ユーザ” とは、あなたが書くものも含めた Lisp プログラムを使う人とします。

Lisp コードの例はこのようなフォント、形式になっています:`(list 1 2 3)`。引数やメタ構文上の変数の名前はこのようなフォント、形式になっています: *first-number*。

1.3.2 nil と t

Lisp ではシンボル nil には、三つの別な意味があります。 'nil' という名のシンボルでもあり、論理真理値 *false* でもあり、空リスト (0 個の要素からなるリスト) でもあります。変数として使用される時は、 nil は常に値 nil を持ちます。

Lisp リーダが関与するかぎり、 '()' と 'nil' は同一です。同じオブジェクト、すなわちシンボル nil です。このシンボルにいくつかの異なる書き方があるのは、すべて人間の読み手のためです。 Lisp リーダが '()' か 'nil' を読んだ後は、実際にプログラマがどちらの表現で書いたかを決定する方法はありません。

本マニュアルでは、空リストとしての意味を強調したいときに () を使用し、真理値 *false* としての意味を強調したいときに nil を使用します。 Lisp プログラム内でも使えるよい慣習です。

```
(cons 'foo ())                ; 空リストであることを強調
(not nil)                     ; 真理値 false であることを強調
```

真理値が求められる文脈中では、 nil 以外の如何なる値も *true* と見なされます。しかし、 t を使うのが、真理値 *true* を表すのに好ましい方法です。 *true* を表す値を選ぶ必要があり、ほかに選択基準がないときは、 t を使いましょう。シンボル t は、常に値 t を持ちます。

Emacs Lisp では、 nil と t は、評価すると常に自分自身になる特別なシンボルです。これは、プログラム中で定数として使用する時、 quote する必要を無くすためです。値を変更しようとすると setting-constant エラーになります。 10.6 節「変数の値へのアクセス」 p.134 を参照してください.

1.3.3 評価の表記法

評価できる Lisp 式を『形式 (form)』と呼びます。形式を評価すると常に結果が生成されます。結果は Lisp オブジェクトです。本マニュアル中の例では、 '⇒' で示されます。

```
(car '(1 2))
     ⇒ 1
```

これは、 "(car '(1 2)) を評価すると 1 になる" と読むことができます。

形式がマクロ呼出であるときは、 Lisp が評価するため、新しい形式に展開します。展開の結果を '↦' で示すことにします。展開した形式の評価の実際の結果は、示すかもしれないし、示さないかも知れません。

```
(third '(a b c))
     ↦ (car (cdr (cdr '(a b c))))
     ⇒ c
```

時々、ある形式を説明しやすくするため、同一結果を生成する別の形式を示すことがあります。二つの形式の正確な等価性は '≡' で示します。

```
(make-sparse-keymap) ≡ (list 'keymap)
```

1.3.4 印字の表記法

本マニュアル内の例の多くは、評価されるとテキストを印字します。コード例を Lisp Interaction バッファ (たとえばバッファ '*scratch*') 内で実行すると、印字されるテキストはバッファ内に挿入されます。例をほかの方法で (たとえば関数 eval-region を評価することで) 実行すると、印字されるテキストはエコー領域に表示されます。エコー領域に表示されるテキストは、単一行に切り詰められているということを承知しておいてください。

本マニュアル中の例では、印字されるテキストを、それがどこに行くかには関係なく、'⊣' で示します。形式 (ここでは bar) を評価して返ってきた値は別の行に来ます。

```
(progn (print 'foo) (print 'bar))
     ⊣ foo
     ⊣ bar
     ⇒ bar
```

1.3.5 エラー・メッセージ

エラーをシグナルする例もあります。これは普通、エラー・メッセージをエコー領域内に表示します。エラー・メッセージは、'error' で始まる行に示すことにします。'error' 自身はエコー領域に現れないことに注意してください。

```
(+ 23 'x)
error  Wrong type argument: integer-or-marker-p, x
```

1.3.6 バッファ・テキストの表記法

"実行前" と "実行後" のテキストを用いて、バッファ内のテキストが変更されたことを示す例もあります。このような例では、バッファ名を含んだ破線 2 本の間に問題になっているバッファの内容を示します。加えて、'⋆' がポイントの位置を示します。(ポイント用のシンボルはもちろん、バッファのテキストの一部ではありません。ポイントが位置している 2 文字間の場所を指します。)

```
---------- Buffer: foo ----------
This is the ⋆contents of foo.
---------- Buffer: foo ----------

(insert "changed ")
     ⇒ nil
---------- Buffer: foo ----------
This is the changed ⋆contents of foo.
---------- Buffer: foo ----------
```

1.3.7 説明文の書式

関数、変数、マクロ、コマンド、ユーザ・オプション、特殊形式は、本マニュアルでは同一フォーマットで説明されます。説明文の最初の行は、その名前を含み、もしあれば、それに引数

が続きます。カテゴリ (関数、変数など何でも) は、右余白の隣に印刷されます。説明文は、時には例とともに、次の行に続きます。

1.3.7.1 関数説明のサンプル

関数の説明では、説明する関数の名前は最初に現れます。同一行にパラメータのリストが続きます。パラメータに使用された名前は説明文本文中でも使用されます。

パラメータ・リスト中にキーワード `&optional` が現れたら、それはそれ以降のパラメータへの引数が省略されるかもしれないことを示します。 (省略されたパラメータのデフォルトは `nil` です)。関数を呼ぶ時は `&optional` を書いてはいけません。

キーワード `&rest`(常に単一のパラメータが後に続きます) は、任意の数の引数を後に続けることができることを示します。後に続くこの単一のパラメータの値は、この引数すべてのリストとなります。関数を呼ぶ時は `&rest` を書いてはいけません。

以下は、架空の関数 `foo` の説明文です。

foo *integer1* &optional *integer2* &rest *integers* Function

関数 `foo` は *integer2* から *integer1* を引き、その後、その結果に残りの引数すべてを加えます。 *integer2* が与えられなかった場合、 19 という値がデフォルトとして使用されます。

```
(foo 1 5 3 9)
     ⇒ 16
(foo 5)
     ⇒ 14
```

より一般的に書くと、

```
(foo w x y...)
≡
(+ (- x w) y...)
```

型名を含んだ名前 (例、 *integer*、 *integer1*、 *buffer*) をもつパラメータは、その型であることが期待されています。型の複数形 (例、 *buffers*) はよく、その型のオブジェクトのリストを意味します。 *object* という名前のパラメータは任意の型でよいこと意味します (Emacs オブジェクト型のリストは第 2 章「Lisp のデータ型」p.19 を参照してください)。ほかの種類の名前を持ったパラメータは、関数の説明文の中で具体的に論じます。いくつかの関数のパラメータに共通するものを、節の冒頭で説明している場合もあります。

optional 引数や rest 引数のより完全な説明は11.2 節「ラムダ式」p.148 を参照してください。

コマンド、マクロ、特殊形式の説明文は、 ‘Function’ という言葉がそれぞれ、 ‘Command’、 ‘Macro’、 ‘Special Form’ という言葉に置き換えられることを除けば同じ書式です。コマンドとは単に、対話的に呼び出されることが許される関数のことです。マクロは、引数を関数とは違うやり方で処理しますが (引数は評価されない)、同様に与えられます。

特殊形式の説明文は、オプション引数や繰り返し引数を指定するために、もっと複雑な記述法を使用しています。なぜなら、特殊形式は、引数リストをもっと複雑な方法で別個の引数に展開することができるからです。'[*optional-arg*]' は、 *optional-arg* がオプションであることを意味し、'*repeated-args*...' は 0 個以上の引数であることを意味しています。いくつかの引数がさらに深いレベルのリスト構造にグループ化されているときは、括弧が使われます。例:

count-loop (*var* [*from to* [*inc*]]) *body*... Special Form

この架空の特殊形式は、 *body* 形式を実行するループを作り、繰り返しのたびに変数 *var* をインクリメントします。最初の繰り返しでは、変数は、値 *from* を持ち、以後の繰り返しでは、 1(*inc* が与えられていればその値) づつインクリメントされます。もし、 *body* を実行する前に *var* が *to* に等しければ、ループを抜けます。例:

```
(count-loop (i 0 10)
  (prin1 i) (princ " ")
  (prin1 (aref vector i)) (terpri))
```

from と *to* が省略された場合、 *var* はループが始まる前に `nil` に束縛されます。繰り返しの冒頭で、もし *var* が `nil` でない場合は、ループを抜けます。例:

```
(count-loop (done)
  (if (pending)
      (fixit)
    (setq done t)))
```

この特殊形式では、引数 *from* と *to* はオプションですが、両方ともあるか、両方ともないか、どちらかでなければなりません。もし両方ともあれば、 *inc* を指定することもできます。これらの引数は、引数 *var* があるリストにグループ化されます。これは *body* と区別するためで、 *body* は形式の残りの要素すべてを含みます。

1.3.7.2 変数説明のサンプル

『変数 (variable)』とは値をもつことができる名前のことです。どんな変数もユーザがセットすることができますが、ユーザが変更できることがはっきり指定されている変数がいくつかあります。それらを『ユーザ・オプション (user options)』と呼びます。普通の変数とユーザ・オプションは、引数がないことを除き、関数の場合と似た書式を使って説明されています。

架空の `electric-future-map` 変数の説明文。

electric-future-map Variable

この変数の値は Electric Command Future モードで使われる全キーマップです。このマップ内の関数で、まだ実行しようとは思っていないコマンドを編集することができます。

ユーザ・オプションの説明文も、'Variable' が 'User Option' に置き換えられる以外は、同じ書式です。

1.4 謝辞

本マニュアルは、数年間にわたる努力の末、 Robert Krawitz、 Bil Lewis、 Dan LaLiberte、 Richard M. Stallman、 Chris Welty、及び GNU manual group のボランティアによって書き上げられました。 Robert J. Chassell は、 Warren A. Hunt, Jr. of Computational Logic, Inc. の手配で、 Defense Advanced Research Projects Agency, ARPA Order 6082 のサポートを受け、マニュアルを監修、編集するのを手伝ってくれました。

訂正をしてくれた人々:Karl Berry, Jim Blandy, Bard Bloom, Stephane Boucher, David Boyes, Alan Carroll, Richard Davis, Lawrence R. Dodd, Peter Doornbosch, David A. Duff, Chris Eich, Beverly Erlebacher, David Eckelkamp, Ralf Fassel, Eirik Fuller, Stephen Gildea, Bob Glickstein, Eric Hanchrow, George Hartzell, Nathan Hess, Masayuki Ida, Dan Jacobson, Jak Kirman, Bob Knighten, Frederick M. Korz, Joe Lammens, Glenn M. Lewis, K. Richard Magill, Brian Marick, Roland McGrath, Skip Montanaro, John Gardiner Myers, Thomas A. Peterson, Francesco Potorti, Friedrich Pukelsheim, Arnold D. Robbins, Raul Rockwell, Per Starback, Shinichirou Sugou, Kimmo Suominen, Edward Tharp, Bill Trost, Rickard Westman, Jean White, Matthew Wilding, Carl Witty, Dale Worley, Rusty Wright, and David D. Zuhn.

2 Lisp のデータ型

Lisp『オブジェクト (object)』とは Lisp のプログラムによって使用されたり、処理されたりするデータです。私たちの世界では、『型 (type)』または『データ型 (data type)』は、あらゆるオブジェクトの集合を表します。

すべてのオブジェクトは少なくとも一つの型に属します。同じ型のオブジェクトは類似した構造をもち、通常は同じ文脈で使用できます。型はオーバラップでき、オブジェクトは二つ以上の型に属することができます。そのため、オブジェクトが特定の型に属しているかを尋ねることはできますが、オブジェクトの型が「何」であるかは尋ねられません。

いくつかの根本的なオブジェクト型は Emacs 内に組み込まれています。これら残りすべての型を構成する元になるものは『プリミティブ型 (primitive type)』と呼ばれます。個々のオブジェクトは、ただ一つのプリミティブ型に属しています。これらの型には『整数 (integer)』、『浮動小数点数 (float)』、『コンス (cons)』、『シンボル (symbol)』、『文字列 (string)』、『ベクタ (vector)』、『subr』、『バイトコード関数 (byte-code function)』、さらに、『バッファ (buffer)』のように編集と関係のある、いくつかの特殊な型 (2.4 節「編集型」 p.32 参照) が含まれます。

個々のプリミティブ型はそれぞれに、あるオブジェクトがその型のメンバーであるかどうかをチェックするための Lisp 関数を持っています。

Lisp のオブジェクトは『それ自身が型を持っている (self-typing)』という点において、他の多くの言語とは違っていることに注意してください。オブジェクトのプリミティブ型は、そのオブジェクト自身に暗黙に埋め込まれています。例えば、オブジェクトがベクタであるならば、どうやってもそれを数値として扱うことはできません。 Lisp は、それがベクタであり、数ではないことを知っています。

ほとんどの言語において、プログラマは個々の変数のデータ型を宣言しなければならず、型はコンパイラが知っているだけでデータの中には表されていません。そうした型の宣言というものは Emacs Lisp には存在しません。 Lisp 変数は、どのような型の値でももつことができ、どんな値を格納したとしても、型やすべてを憶えています。

この章では GNU Emacs Lisp の個々の標準型の目的、印字表現、リード構文について述べます。これらの型の使用方法の詳細は後々の章で述べられています。

2.1 印字表現とリード構文

オブジェクトの『印字表現 (printed representation)』とは、そのオブジェクトに対して Lisp プリンタ (関数 `prin1`) が生成する出力形式です。オブジェクトの『リード構文 (read syntax)』とは、そのオブジェクトとして Lisp リーダ (関数 `read`) が認識する入力形式です。ほとんどのオブジェクトは、一つ以上のリード構文をもっています。中にはリード構文をもってない型のオブジェクトもありますが、そのような場合を除いて、オブジェクトの印字表現は、そのオブジェクトのリード構文でもあります。

他の言語では、式はテキストです。それ以外の形式はまったくありません。 Lisp では、式はオブジェクトであることが根本であり、オブジェクトのリード構文であるテキストは従属的なも

のでしかありません。この区別を強調する必要はほとんどありませんが、このことは心の奥にとどめておくことが必要で、そうしなければ、非常に混乱することがあるでしょう。

すべての型は印字表現をもっています。いくつかの型はリード構文を持っていません。そうなっているのは Lisp プログラム中にその型のオブジェクトを直接に書くことに意味がないからです。例えば、バッファ型はリード構文をもっていません。そのような型のオブジェクトは『ハッシュ表記 (hash notation)』で印字されます。ハッシュ表記は文字 '`#<`' の後に説明の文字列 (通常、型名のあとにオブジェクトの名前を続けたもの) が続き、対応する '`>`' で終わります。ハッシュ表記を読み込むことはまったくできません。したがって Lisp リーダは '`#<`' に出くわすとエラー `invalid-read-syntax` を発します。

```
(current-buffer)
     ⇒ #<buffer objects.texi>
```

あなたが式を対話的に評価するときには、 Lisp インタプリタは最初、そのテキスト表現を読み、 Lisp オブジェクトを作成し、それから、評価を行ないます (第 8 章「評価」 p.101 参照)。しかし、評価と読込みは別々な動作です。読込みは、読み込まれたテキストが表現している Lisp オブジェクトを返します。そのオブジェクトは後で評価されることも、されないこともあります。オブジェクトの読み込みを行う基本関数 `read` の説明は 16.3 節「入力関数」 p.230 を参照してください。

2.2 コメント

『コメント (comment)』は、プログラムの読み手のためにだけ、それに書き込まれるテキストであり、プログラムの意味にはまったく影響を与えません。 Lisp において、セミコロン ('`;`') が文字列定数、または文字定数内にないならば、それがコメントの始まりになります。コメントは行末まで続きます。 Lisp リーダはコメントを捨てます。したがって、コメントは Lisp システムにおいてプログラムを表している Lisp オブジェクトの一部とはなりません。

次の *count* 文字を読み飛ばす '`#@`*count*' 構文要素はバイナリ・データを含んだ、プログラムによって生成されたコメントに有用です。 Emacs Lisp のバイトコンパイラは、これをその出力ファイルに使用します (第 14 章「バイトコンパイル」 p.181 参照)。しかし、それはソース・ファイルに使うものではありません。

コメントの形式の規則については、 A.4 節「コメントの書き方」 p.641 を参照してください。

2.3 プログラム型

Emacs Lisp の型には二つの一般的なカテゴリがあります。一方は Lisp のプログラミングで必要となるもので、もう一方は編集で必要となるものです。前者は同じ形式、または別の形式で、多くの Lisp の処理系に存在します。後者は Emacs Lisp 特有のものです。

2.3.1 整数型

Emacs Lisp の整数の値の範囲は、たいていのマシンでは -134217728 から 134217727 まで (28 ビット、すなわち -2^{27} から $2^{27}-1$) です。(マシンによってはより広い範囲を扱えます。)

Emacs Lisp の算術演算用の関数はオーバフローをチェックしないので注意が必要です。つまり、たいていのマシンでは `(1+ 134217727)` が -134217728 になってしまいます。

整数のリード構文は (基数 10 の) 数字の列であり、先頭に符号があってもよく、終端にピリオドがあってもよいです。 Lisp インタプリタによって作成される印字表現では '+' で始まったり、'.' で終わったりすることはまったくありません。

```
-1                 ; 整数 -1
1                  ; 整数 1
1.                 ; これも整数 1
+1                 ; これも整数 1
268435457          ; これも整数 1!
                   ;   (28 ビット実装の場合)
```

詳細は第 3 章「数値」 p.41 を参照してください。

2.3.2 浮動小数点型

Emacs バージョン 19 では浮動小数点数をサポートします (無効にするコンパイル・オプションもあります)。浮動小数点数の正確な範囲はマシン依存です。

浮動小数点数の印字表現は小数点 (最低一つの数字が後に続く)、指数、または両方を必要とします。例えば、 '`1500.0`', '`15e2`', '`15.0e2`', '`1.5e3`', '`.15e4`' は、値が 1500 である浮動小数点数を書く五つの方法です。それらはすべて等価です。

詳細は 第 3 章「数値」 p.41 を参照してください。

2.3.3 文字型

Emacs Lisp の『文字 (character)』は整数以外のなにものでもありません。すなわち、文字はそれらのキャラクタコードで表されています。例えば、文字 `A` は整数 65として表されます。

個々の文字がそれ単体でプログラムに使用されるのは、まれなことです。文字列 (文字から構成されているシーケンス) で扱うことのほうが、より一般的です。 2.3.8 節「文字列型」 p.28 を参照してください。

文字列、バッファ、およびファイル内の文字は現在のところ 0 から 255 までの範囲の間 (8 ビット) に制限されています。あなたが、文字列、バッファ、またはファイルに、それより大きな整数を格納しても、その範囲に切り捨てられます。キーボード入力を表している文字は、ずっと広い範囲をもっています。

文字はまさに整数であるので、文字の印字表現は十進数です。これはまた、文字として読み込み可能なリード構文でもありますが、 Lisp プログラムにおいて、そのような方法で文字を書くのは非常に悪い考えです。あなたは Emacs Lisp が文字のために提供する、特別なリード構文の形式を常に使用するべきです。この構文の形式は疑問符で始まります。

英数字として通常使われるリード構文は疑問符の後に文字が続いたものです。したがって、文字 `A` を表すのは '`?A`'、文字 `B` を表すのは '`?B`'、文字 `a` を表すのは '`?a`' です。

例えば:

```
?Q ⇒ 81      ?q ⇒ 113
```

句読点文字に対しても同一の構文が使用できますが、多くの場合 '\' をつけ加えておくのがよい考えです。これは Lisp コードを編集するための Emacs コマンドが誤動作しないようにするためです。例えば、'?\ ' はスペース文字を書く方法です。文字が '\' であるならば、それを引用するために二つ目の '\' を使用し '?\\' とする必要があります。

Control-g、バックスペース、タブ、改行、垂直タブ、改ページ、復帰、エスケープの文字は、それぞれ '?\a', '?\b', '?\t', '?\n', '?\v', '?\f', '?\r', '?\e' で表すことができます。それらの値は十進数で 7, 8, 9, 10, 11, 12, 13, 27 です。したがって、

```
?\a ⇒ 7                  ; C-g
?\b ⇒ 8                  ; バックスペース, BS, C-h
?\t ⇒ 9                  ; タブ, TAB, C-i
?\n ⇒ 10                 ; 改行, LFD, C-j
?\v ⇒ 11                 ; 垂直タブ, C-k
?\f ⇒ 12                 ; 改ページ文字, C-l
?\r ⇒ 13                 ; キャリッジ・リターン, RET, C-m
?\e ⇒ 27                 ; エスケープ文字, ESC, C-[
?\\ ⇒ 92                 ; バックスラッシュ文字, \
```

このような、バックスラッシュで始まるシーケンスはバックスラッシュがエスケープ文字の役割を果たすので、『エスケープ・シーケンス (escape sequence)』としても知られています。この用法は文字 ESC と何の関係もありません。

制御文字は、さらに別のリード構文を使用して表されることがあります。これは疑問符の後にバックスラッシュ、キャレット、対応する非制御文字 (英大文字または英小文字) が続いたものから成っています。例えば、'?\^I' と '?\^i' はどちらも文字 C-i(値が 9 である文字) を表す適正なリード構文です。

'^' の代わりに 'C-' を使用することができます。したがって、'?\C-i' は '?\^I' や '?\^i' と等価です。

```
?\^I ⇒ 9      ?\C-I ⇒ 9
```

文字列やバッファの中では、ASCII に存在する制御文字だけに使用が制限されますが、キーボード入力という目的では、あらゆる文字を 'C-' によって制御文字に変えることができます。ASCII でない制御文字の文字コードは対応する制御文字でないコードとともに、2^{26} ビットを含んでいます。通常の端末では ASCII でない制御文字を生成する方法をもちませんが、X 端末では通常の使い方でそれらを生成することができます。

歴史上の理由により、Emacs は DEL 文字を Control-? と同等に扱います。

```
?\^? ⇒ 127      ?\C-? ⇒ 127
```

結果として、X においては意味のある入力文字 Control-? を表現するのは現在のところ不可能です。この方法で DEL を参照している Lisp ファイルが多いので、これを変更するのは容易ではありません。

ファイルや文字列内の制御文字の表現には '^' 構文を勧めます。キーボード入力の制御文字には、私たちは 'C-' 構文を好みます。この使い分けはプログラムの意味には何も貢献しませんが、それを読んだ人の理解の助けになります。

『メタ文字 (meta character)』とは `META` 修飾キーとともにタイプされた文字です。そのような文字を表している整数は 2^{27} ビットがセットされています (ほとんどのマシンでは負の数で表されます)。私たちは基本文字コードの幅を広くとるために、このメタ修飾子や他の修飾子などに上位のビットを使用しています。

文字列では 2^7 ビットがメタ文字であることを示していています。したがって、文字列内に存在できるメタ文字は 128 から 255 までの範囲のコードをもち、それは通常の ASCII 文字のメタバージョンです。(Emacs 18 以前では、この規則が文字列以外に含まれる文字にも適用されていました。)

メタ文字のリード構文は '`\M-`' を使用します。例えば、'`?\M-A`' は `M-A` を意味します。あなたは 8 進文字コード (後を参照)、'`\C-`'、または他のあらゆる文字の構文に対して、'`\M-`' を使用できます。したがって、`M-A` を '`?\M-A`' とも '`?\M-\101`' とも記述することができます。同様に、`C-M-b` を '`?\M-\C-b`' とも '`?\C-\M-b`' とも '`?\M-\002`' とも記述することができます。

通常の英文字の大文字、小文字の区別は、その文字の ASCII 部分のコードよって示されますが、ASCII は制御文字が大文字であるか小文字であるかを表現する手段を持っていません。Emacs は制御文字をタイプする際にシフトキーが使用されたことを示すのに、2^{25} ビットを使用します。この区別は X 端末、または他の特殊な端末を使用するときにだけ可能です。通常の端末にはコンピュータにその区別を示す術がありません。

X ウィンドウ・システムは、文字に設定することのできる修飾子ビットとしてさらに三つ定義しています。即ち『ハイパー (hyper)』、『スーパー (super)』、『オルト (alt)』です。これらのビットの構文は、それぞれ '`\H-`', '`\s-`', '`\A-`' です。したがって '`?\H-\M-\A-x`' は `Alt-Hyper-Meta-x` を表します。数値の上ではビット値 2^{22} がオルト、2^{23} がスーパー、2^{24} がハイパーです。

結局のところ、最も一般的なリード構文は、疑問符の後に、バックスラッシュと 8 進数の文字コード (3 個まで) が続いたものから成っています。したがって、文字 `A` は '`?\101`'、文字 `C-a` は '`?\001`'、文字 `C-b` は`?\002` と表します。この構文は、あらゆる ASCII 文字を表すことができますが、明示的な 8 進数の値が ASCII 表現より重要なときだけに、使用するのがよいでしょう。

```
?\012 ⇒ 10        ?\n ⇒ 10        ?\C-j ⇒ 10
?\101 ⇒ 65        ?A ⇒ 65
```

特別なエスケープの意味をもたない、あらゆる文字に対しても、その前にバックスラッシュをつけることは許されていますが、その効果はありません。したがって、'`?\+`' は '`?+`' と等価です。ほとんどの文字に対しては、その前にバックスラッシュをつける必要性はまったくありません。しかし、Lisp コードの編集では、Emacs のコマンドが混同するのを避けるために、文字 '`()\|;'`"#.,`' の前にはバックスラッシュをつけておくべきです。また、スペース、タブ、改行、改ページのような空白文字の前にもバックスラッシュをつけてください。しかし、タブのようなものは実際の空白文字よりも '`\t`' のように読みやすいエスケープ・シーケンスを使用したほうがよりすっきりします。

2.3.4 シンボル型

GNU Emacs Lisp での『シンボル (symbol)』は名前をもつオブジェクトです。シンボルの名前はシンボルの印字表現でもあります。通常の使用においては、名前は一意であり、二つのシンボルが同じ名前をもつことはありません。

シンボルは変数としても関数名としても使用できますし、属性リストを保持するものとしても使用することができます。また、単にデータ構造中でその存在を確実に認識するために、他のどんな Lisp オブジェクトとも区別されるように使用することもあります。与えられた文脈においては、普通これらの使用目的のうちの一つだけが意図されています。しかし、一つのシンボルに対して、これらの方法のうちのすべてを、別個に使用することは可能です。

シンボル名は、あらゆる文字を含むことができます。ほとんどのシンボル名は英文字、数字、および句読点文字 '`-+=*/`' で記述されます。そのような名前は特殊な句読点文字を必要としません。名前に使用する文字は、その名前が数値と似ないようになっていれば十分です (それを行うのであれば、強制的にシンボルとして解釈されるように、名前の先頭に '`\`' を記述してください)。文字 '`_~!@$%^&:<>{}`' は、あまり使用されることはありませんが、使用するとしても特殊な句読点文字を必要としません。他のあらゆる文字はバックスラッシュでエスケープすることによって、シンボルの名前の中に含めることができます。文字列での使用とは異なり、シンボルの名前でのバックスラッシュは単にバックスラッシュに続く一つの文字を示すだけです。例えば、文字列において '`\t`' はタブ文字を表しています。しかし、シンボルの名前において '`\t`' は単に文字 `t` を示します。シンボルの名前の中にタブ文字を含めるためには実際のタブ文字 (前置するバックスラッシュが必要) を使用しなければなりません。しかし、そのようなことをするのはまれです。

> **Common Lisp 注意書き:** Common Lisp では明示的に回避されないかぎり、英小文字は常に英大文字に「変換して解釈」されます。 Emacs Lisp では、英大文字と英小文字は区別されます。

以下に、シンボル名の例をいくつか示します。 5 番目の例での '`+`' は数値として読み込まれないようにするために、エスケープされていることに注意してください。 6 番目の例にも '`+`' がありますが、名前の他の部分から数値としては無効であるとわかるのでエスケープする必要はありません

```
foo                 ; `foo' という名前のシンボル。
FOO                 ; `FOO' という名前のシンボル。
                    ;    `foo' とは異なる。
char-to-string      ; `char-to-string' という名前のシンボル。
1+                  ; `1+' という名前のシンボル
                    ;    (整数の `+1' ではありません)。
\+1                 ; `+1' という名前
                    ;    (あまり読みやすい名前ではありません)。
\(*\ 1\ 2\)         ; `(* 1 2)' という名前のシンボル (悪い名前)。
+-*/_~!@$%^&=:<>{}  ; `+-*/_~!@$%^&=:<>{}' という名前のシンボル。
                    ;    これらの文字はエスケープする必要はありません。
```

2.3.5 シーケンス型

『シーケンス (sequence)』とは要素の集合のうち順序のあるものを表す Lisp オブジェクトです。 Emacs Lisp でのシーケンスにはリストと配列の 2 種類があります。したがって、リスト型や配列型のオブジェクトもシーケンスと考えられます。

配列はさらに文字列とベクタに分類されます。ベクタはどのような型の要素でも保持できますが、文字列の要素は 0 から 255 までの範囲の文字でなければなりません。しかし、文字列内の文字はバッファ中の文字のようにテキスト属性をもつことができます (29.18 節「テキスト属性」 p.509 参照)。ベクタは、その要素が文字である場合でさえもテキスト属性をサポートしていません。

リスト、文字列、ベクタは違ったものですが、それらは重要な類似点をもっています。例えば、すべてのシーケンスは長さ l を持ち、 0 から $l-1$ で順序づけされた要素をもっています。また、シーケンス関数と呼ばれるいくつかの関数は、あらゆる種類のシーケンスを受け取ることができます。例えば、関数 `elt` は与えたインデックスに該当するシーケンスの要素を取り出すのに使用されます。第 6 章「シーケンス、配列、ベクタ」 p.85 を参照してください。

シーケンスは読み込まれる度に新たに作成されるので、同じシーケンスを二度読み込むのは不可能です。シーケンスのリード構文を二度読み込むと、等しい内容の二つのシーケンスを得ます。一つの例外があります。空リスト `()` は常に同じオブジェクト `nil` を意味します。

2.3.6 コンス・セル型とリスト型

『コンス・セル (cons cell)』は CAR と CDR という名前の二つのポインタから成るオブジェクトです。コンス・セルの二つのポインタは任意の Lisp オブジェクトを指すことができます。

『リスト (list)』はコンス・セルが連続したもので、個々のコンス・セルの CDR が別のコンス・セルあるいは空リストを指すようにリンクされています。リストに対する操作を行う関数については、第 5 章「リスト」 p.67 を参照してください。ほとんどのコンス・セルがリストの一部分として使用されているので、『リスト構造 (list structure)』という用語はコンス・セルで作られた任意の構造であることを指しています。

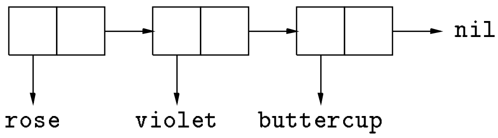
CAR や CDR という名前は今では歴史上の意味だけをもっています。オリジナルの Lisp 処理系は IBM 704 コンピュータの上で動作し、そのコンピュータでは、ワードが “address” 部と “decrement” 部の二つの部分に分けられていました。 CAR はレジスタの address 部の内容を取り出す命令で、 CDR は decrement 部の内容を取り出す命令でした。対照的に「コンス・セル」という名前はコンス・セルを作成する関数 `cons` からきたもので、その関数の目的であるセルの構築 (construction of cells) というところから名づけられています。

コンス・セルは Lisp では非常に中心的なものであるので、「コンス・セルではないオブジェクト」のための用語も用意されています。そのようなオブジェクトは『アトム (atom)』と呼ばれます。

リストのリード構文と印字表現は同一であり、左括弧、任意の個数の要素、右括弧から成っています。

読み込むとすぐに、括弧で囲まれた中の個々のオブジェクトはリストの要素になります。いいかえると、個々の要素ごとにコンス・セルが作成されます。コンス・セルの CAR は要素を指し、 CDR は、次のコンス・セルがリストの次の要素を保持しているなら、そのコンス・セルを指しています。最後のコンス・セルの CDR は NIL を指すように設定されています。

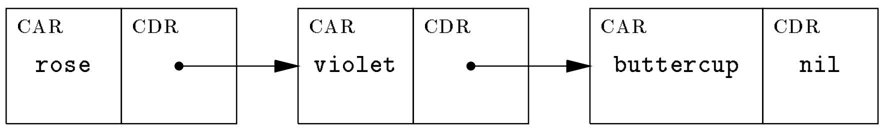
リストはコンス・セルを一対の箱で表したダイアグラムによって図解できます。(Lisp リーダはそのような図を読むことはできません。人間とコンピュータの両方が理解することのできるテキスト上の表記とは違い、箱の図は人間だけに理解できます。) 以下は三つの要素のリスト `(rose violet buttercup)` を表しています。



このダイアグラムにおいて、個々の箱は、どのような Lisp オブジェクトでも参照できるスロットを表しています。個々の箱の対はコンス・セルを表しています。個々の矢は Lisp オブジェクト、アトム、または別のコンス・セルへのリファレンスです。

この例において、最初のコンス・セルの CAR である最初の箱は (シンボル)`rose` を参照、または「含んで」います。最初のコンス・セルの CDR である 2 番目の箱は 2 番目のコンス・セルである次の箱の対を参照しています。 2 番目のコンス・セルの CAR は `violet` を参照し、 CDR は 3 番目のコンス・セルを参照します。 3 番目 (最後) のコンス・セルの CDR は `nil` を指しています。

以下に、同じリスト `(rose violet buttercup)` を違った方法で描いた別のダイアグラムを示します。

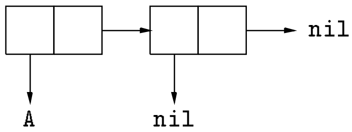


要素を持たないリストは『空リスト (empty list)』です。それはシンボル `nil` と同一のものです。いいかえると、 `nil` はシンボルでもあり、リストでもあります。

以下に Lisp の構文で記述されたリストの例を示します。

```
(A 2 "A")            ; 三つの要素をもつリスト。
()                   ; 要素のないリスト (空リスト)。
nil                  ; 要素のないリスト (空リスト)。
("A ()")             ; 一つの要素 (文字列 "A ()") をもつリスト。
(A ())               ; 二つの要素 (A と空リスト) をもつリスト。
(A nil)              ; 直前のものと同じ。
((A B C))            ; 一つの要素 (三つの要素をもつリスト) をもつリスト。
```

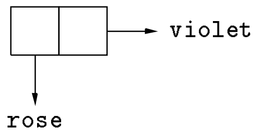
以下にリスト `(A ())`(または `(A nil)`) を箱と矢で描いたものを示します。



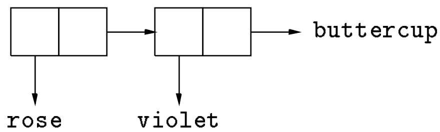
2.3.6.1 ドット対表記

ドット対表記は、最後の CDR が nil でないコンス・セルの連鎖を表すために、リスト表記と組み合わせて使うことができます。例えば、(rose violet . buttercup) は、オブジェクト a、 CDR がオブジェクト b であるコンス・セルを意味しています。したがって、ドット対表記はリストの構文よりも一般的です。リスト '(1 2 3)' はドット対表記では '(1 . (2 . (3 . nil)))' と記述されます。 nil で終わるリストに関しては、どちらの表記も同じ結果になりますが、使用できる場合には、リスト表記がのほうが普通は、よりすっきりしていて、より便利です。リストを印字するときには、セルの CDR がリストでない場合のみドット対表記が使用されます。

以下にドット対を箱を使った表記で図解します。この例は (rose . violet) の対を示しています。

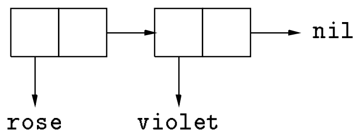


ドット対表記はリスト表記とあわせて使用することもでき、そのような表記は最後の CDR が nil でないコンス・セルの連鎖を表します。例えば、(rose violet . buttercup) は (rose . (violet . buttercup)) と同一のものです。そのオブジェクトは以下のようになります。



これらのダイアグラムは、なぜ(rose . violet . buttercup)が不正な構文であるかを明らかにしています。それは二つではなく、三つの部分をもつコンス・セルを必要としてしまうのです。

リスト (rose violet) は、(rose . (violet)) と同一のものであり、以下のようになります。



同様に、三つの要素をもつリスト `(rose violet buttercup)` は、`(rose . (violet . (buttercup)))` と同一のものです。

2.3.6.2 連想リスト型

『連想リスト (association list)』または『alist』は、要素がコンス・セルの特別に構成されたリストです。個々の要素において、 CAR は『キー (key)』と考えられ、 CDR は『連想値 (associated value)』と考えられます。 (場合によっては、連想値は CDR の CAR に格納されます。) 連想リストはリストの前に容易に連想を追加したり、リストの前から連想を取り除いたりできるので、よくスタックとして使用されます。

例えば、

```
(setq alist-of-colors
      '((rose . red) (lily . white)  (buttercup . yellow)))
```

は、三つの要素をもつ alist を変数 `alist-of-colors` に設定します。最初の要素では、 `rose` がキーであり、 `red` が値です。

alist の詳細と、 alist に対する関数については 5.8 節「連想リスト」 p.81 を参照してください。

2.3.7 配列型

『配列 (array)』は、他の Lisp オブジェクトを参照する任意個のスロットから構成され、メモリの連続したブロックに配置されます。配列のどの要素に対するアクセスも、かかる時間は同じです。逆に、リストの要素のアクセスではリストの要素の位置に比例した時間が必要となります。 (リストの終端の要素はリストの先頭にある要素よりもアクセスに長い時間がかかります。)

Emacs は二つの配列型 (文字列とベクタ) を定義します。文字列は文字の配列であり、ベクタは任意のオブジェクトの配列です。両方の配列とも一次元です (ほとんどの他のプログラミング言語では多次元配列をサポートしていますが、それは本質的なものではありません。配列の配列によって、同じ効果を得ることができます)。いずれの配列型も、それ自身のリード構文をもっています。 2.3.8 節「文字列型」 p.28 および 2.3.9 節「ベクタ型」 p.30 を参照してください。

配列は整数の最大値まで、どんな長さでももつことができます。しかし一度作成されると、その大きさは固定です。配列の先頭の要素の添字は 0 であり、 2 番目の要素は添字 1 を持ち、以下同様に添字が割り当てられます。これは『0 起源 (zero-origin)』添字付けと呼ばれます。例えば、四つの要素のある配列は添字 0, 1, 2, 3 を持ちます。

配列型はシーケンス型に含まれていて、文字列型とベクタ型を含んでいます。

2.3.8 文字列型

『文字列 (string)』は文字の配列です。文字列はテキスト・エディタにおいて予想されるように、 Emacs において多くの目的に使用されます。例えば、 Lisp のシンボルの名前として使用されたり、ユーザに対するメッセージとして使用されたり、バッファから取り出したテキストを表

すために使用されたりします。 Lisp での文字列は定数です: 文字列を評価すると同じ文字列を返します。

文字列のリード構文は `"like this"`のように、二重引用符、任意の数の文字、そして、もう一つの二重引用符から成っています。 Lisp リーダは文字列の文字の読み込みのために、単一の文字の読み込みに使用するものと同じ形式を受けつけます (文字定数の始まりを示す疑問符はいりません)。タブ文字や `C-a` や `M-C-A` のような表示できない文字は便利なエスケープ・シーケンスを使用して `"\t, \C-a, \M-\C-a"`のように入力できます。文字列内に二重引用符を含めるには、その前にバックスラッシュをつけます。したがって、 `"\""`は単一の二重引用符だけを含む文字列です。 (文字のリード構文の詳細は 2.3.3 節「文字型」 p.21 を参照してください。)

文字列定数内で、メタ文字であることを示すのに '`\M-`' を使用した場合、文字列内のその文字の 2^7 ビットがセットされます。これは (文字列内ではない) 文字それ自身のメタ修飾子と同じ表現ではありません。 2.3.3 節「文字型」 p.21 を参照してください。

文字列は、ハイパー、スーパー、オルト修飾子をもった文字を保持できません。文字列は、 ASCII 制御文字を保持できますが、それ以外の制御文字は保持できません。 ASCII 制御文字では、大文字、小文字の区別はされません。

文字列の印字表現は二重引用符、文字列がもっている文字の並び、そしてもう一つの二重引用符から成っています。しかし、文字列内にあるバックスラッシュまたは二重引用符は `"this \" is an embedded quote"` のようにバックスラッシュでエスケープしなければなりません。

改行文字は、文字列のリード構文においては、特殊なものではありません。二重引用符の間に改行を入れた場合、それは文字列内の文字になります。しかし、前に '\' をつけてエスケープされた改行は文字列の一部分にはなりません。すなわち、 Lisp リーダは文字列の読み込みで、エスケープされた改行を無視します。

```
"It is useful to include newlines
in documentation strings,
but the newline is \
ignored if escaped."
     ⇒ "It is useful to include newlines
in documentation strings,
but the newline is ignored if escaped."
```

文字列は文字自身だけでなく、文字列が含んでいるテキストの属性を保持できます。これによって、特殊な処理なしに、属性を維持した状態で文字列とバッファの間でテキストをコピーするプログラムが可能になります。 29.18 節「テキスト属性」 p.509 を参照してください。テキスト属性をもつ文字列は特殊なリード構文と印字表現を持っています。

`#("`*characters*`"` *property-data*`...)`

ここで、 *property-data* は 0 個以上の要素から成っていて、以下の三つのグループです。

beg end plist

要素 *beg* と *end* は整数で、その二つで文字列内の添字の範囲をを指定します。 *plist* は、その範囲に適用される属性リストです。

文字列に対する関数については、第 4 章「文字列と文字」 p.55 を参照してください。

2.3.9 ベクタ型

『ベクタ (vector)』は、任意の型の要素の一次元配列です。ベクタのどの要素に対するアクセスも、かかる時間は一定です。(リストでは、要素のアクセス時間は要素のリストの最初からの距離と比例しています。)

ベクタの印字表現は、左角括弧、要素の並び、そして右角括弧から成っています。これはリード構文でもあります。数値や文字列のように、ベクタは評価に関しては定数と考えられます。

```
[1 "two" (three)]      ; 三つの要素をもつベクタ。
     ⇒ [1 "two" (three)]
```

ベクタに関する関数については、 6.4 節「ベクタ」 p.89 を参照してください。

2.3.10 関数型

他のプログラミング言語の関数が実行可能なように、『Lisp 関数 (Lisp function)』オブジェクトは、実行コードの一部です。しかし、 Lisp の関数は、 Lisp オブジェクトであることが根本であり、それを表現するテキストは従属的なものでしかありません。ラムダ式、先頭要素がシンボル `lambda` であるリスト、は Lisp オブジェクトです。 (11.2 節「ラムダ式」 p.148 参照)。

ほとんどのプログラミング言語では、名前のない関数をもつことは不可能です。 Lisp では、関数は固有の名前をもちません。ラムダ式は『匿名関数 (anonymous function)』とも呼ばれます (11.7 節「匿名関数」 p.156 参照)。 Lisp における名前つきの関数は実際にはシンボルであり、その関数セルに有効な関数をもちます (11.4 節「関数の定義方法」 p.152 参照)。

ほとんどの場合、関数は、その名前が Lisp プログラム中の Lisp 式に記述されたときに呼ばれます。しかし、実行時に関数オブジェクトを構成したり、取得したりすることができ、プリミティブ関数 `funcall` や `apply` でそれを呼び出すことができます。 11.5 節「関数の呼出し」 p.154 を参照してください。

2.3.11 マクロ型

『Lisp マクロ (Lisp macro)』は、 Lisp 言語を拡張するユーザ定義の構文要素です。それは、ほとんど関数のようなオブジェクトとして表現されますが、パラメータ引き渡しの意味が異なっています。 Lisp マクロは、先頭要素がシンボル `macro` で、 CDR が `lambda` シンボルを含む Lisp 関数オブジェクトであるリストの形式を持っています。

Lisp マクロは、普通、組込み関数 `defmacro` で定義されますが、 `macro` で始まっていれば、どのようなリストも Emacs に関するかぎりはマクロです。マクロを記述する方法の説明は第 12 章「マクロ」 p.163 を参照してください。

2.3.12 プリミティブ関数型

『プリミティブ関数 (primitive function)』は、プログラミング言語 C で記述されているが Lisp から呼び出すことのできる関数です。プリミティブ関数は『subr』とも、『組込み関数 (built-in function)』とも呼ばれます。 (“subr” という用語は “subroutine” からきています。)

ほとんどのプリミティブ関数は、それが呼ばれたときに、その引数のすべてを評価します。すべてのその引数を評価するわけではないプリミティブ関数は、『特殊形式 (special form)』と呼ばれます (8.2.7 節「特殊形式」p.107 参照)。

関数がプリミティブであるかどうかは、関数を呼び出す側には重要ではありません。しかし、Lisp によって記述された関数で、同じ名前のプリミティブ関数を置き換えようとするならば、これは重要になります。その理由は、プリミティブ関数は C コードから直接呼ばれうることです。再定義された関数を Lisp から呼び出す場合には、その新しい定義が使用されますが、 C コードからの呼び出しは依然として組込みの定義が使用されるかもしれません。

『関数 (function)』という用語は Lisp で記述されたか、 C で記述されたかにかかわらず、すべての Emacs 関数を指します。 Lisp で記述された関数については、 2.3.10 節「関数型」p.30 を参照してください。

プリミティブ関数にはリード構文がなく、サブルーチンの名前をもつハッシュ表記で印字します。

```
(symbol-function 'car)          ; シンボルの関数セルのアクセス。
     ⇒ #<subr car>
(subrp (symbol-function 'car))  ; これはプリミティブ関数か?
     ⇒ t                        ; イエス。
```

2.3.13 バイトコード関数型

バイトコンパイラは『バイトコード関数オブジェクト (byte-code function object)』を作成します。内部的には、バイトコード関数オブジェクトは、ほとんどベクタのようなものです。しかし、それが関数として呼ばれれるようなときには、評価子はこのデータ型を特別に扱います。バイトコンパイラに関する情報は第 14 章「バイトコンパイル」p.181 を参照してください。

バイトコード関数オブジェクトの印字表現とリード構文はベクタに似ていて、始まりの ‘[’ の前に ‘#’ がついています。

2.3.14 自動ロード型

『自動ロード・オブジェクト (autoload object)』は先頭要素がシンボル `autoload` であるリストです。それは実際の定義の場所をもつシンボルの関数定義として記憶されます。それは実際の定義が、必要なときにロードされる Lisp コードのファイルの中にあることを示しています。自動ロード・オブジェクトはファイルの名前と、それに加えていくらかの実際の定義に関する情報を含んでいます。

ファイルがロードされると、そのシンボルは、自動ロード・オブジェクトではない新しい関数定義をもつでしょう。その後、最初からそれがあったかのように新しい定義が呼び出されます。ユーザーからみれば、ロードされたファイルに含まれる関数定義を使用した関数呼出しが期待したとおりに動作するわけです。

自動ロード・オブジェクトは普通、関数 `autoload` によって作成され、シンボルの関数セル内に記憶されます。詳細は 13.2 節「自動ロード」p.173 を参照してください。

2.4 編集型

前のセクションの型は多くの Lisp の方言で共通しています。 Emacs Lisp は編集に関連した目的のために、いくつかの追加のデータ型を提供します。

2.4.1 バッファ型

『バッファ (buffer)』は、編集可能なテキストを保持するオブジェクトです (第 24 章「バッファ」 p.393 参照)。ほとんどのバッファは、ディスク上のファイルの内容を編集できるように保持していていますが、他の目的で使用されるものもあります。ほとんどのバッファには、ユーザによって見られるという意義もあり、したがって、かなり長い間ウィンドウ上に表示されていることもあります (第 25 章「ウィンドウ」 p.407 参照)。しかし、バッファがどれかのウィンドウに表示されている必要はありません。

バッファの中身は、ほとんど文字列のようなものですが、バッファは Emacs Lisp では文字列のようには使用できませんし、適用できる操作も異なっています。例えば、テキストの “挿入” ということを考えてみると、文字列への場合は部分文字列の結合が必要となり、その結果は完全に新しい文字列オブジェクトになってしまうのに対し、バッファへのテキストの挿入は非常に効率的です。

個々のバッファは『ポイント (point)』と呼ばれる指定位置をもっています (第 27 章「位置」 p.457 参照)。どのようなときにでも、一つのバッファが『カレント・バッファ (current buffer)』になっています。ほとんどの編集コマンドはカレント・バッファのポイントの付近の内容に対して作用します。します。このマニュアルでは、一章を割いて、これらの関数の説明にあてています (第 29 章「テキスト」 p.479 参照)。

他のデータ構造のいくつかは個々のバッファと関連しています。

- ローカル構文テーブル (第 31 章「構文テーブル」 p.541 参照)。
- ローカル・キーマップ (第 19 章「キーマップ」 p.293 参照)。
- ローカル変数の束縛リスト (10.9 節「バッファローカルな変数」 p.140 参照)。
- オーバレイのリスト (35.9 節「オーバレイ」 p.609 参照)。
- バッファ中のテキストのテキスト属性 (29.18 節「テキスト属性」 p.509 参照)。

ローカル・キーマップとローカル変数のリストは、グローバルなバインドやグローバル変数の値に対し個別に優先するエントリを含んでいます。これらはプログラムを実際に変えずに、種々のバッファにおけるプログラムの振る舞いをカスタマイズするために使用されます。

バッファは『間接 (indirect)』にすることができ、それは別のバッファのテキストを共有することを意味します。 24.11 節「間接バッファ」 p.406 を参照してください。

バッファはリード構文をもちません。バッファは、バッファの名前を示すハッシュ表記で印字します。

```
(current-buffer)
     ⇒ #<buffer objects.texi>
```

2.4.2 マーカ型

『マーカ (marker)』は特定のバッファ内の位置を示します。したがって、マーカは二つの構成要素をもっています。一つはバッファを指すためのもので、もう一つは位置を指すためのものです。バッファのテキストの変化があっても、マーカが常にバッファ内の同じ二つの文字の間を指していることが保証されるように、位置の値は自動的に再配置されます。

マーカはリード構文をもちません。マーカはハッシュ表記で印字し、それには現在の文字の位置と、バッファの名前がつきます。

```
(point-marker)
     ⇒ #<marker at 10779 in objects.texi>
```

マーカのテスト、作成、コピー、移動の方法に関しては、第 28 章「マーカ」 p.471 を参照してください。

2.4.3 ウィンドウ型

『ウィンドウ (window)』は Emacs がバッファを表示するために使用する端末画面の一部を表します。すべてのウィンドウは、一つの対応したバッファをもち、その内容がウィンドウに表示されます。逆に、与えられたバッファは、一つのウィンドウに表示したり、ウィンドウに表示しなかったり、複数のウィンドウに表示するようにすることができます。

多くのウィンドウが同時に存在できますが、どのようなときでも、一つのウィンドウが『選択されたウィンドウ (selected window)』と称されます。これは、 Emacs がコマンドを受けつける状態のときに、カーソルが (普通) 表示されているウィンドウです。選択されたウィンドウは、普通、カレント・バッファを表示していますが、必ずしもそういうわけではありません。

ある画面上のウィンドウは、フレームにグループ化されます。個々のウィンドウは、ただ一つのフレームに属します。 2.4.4 節「フレーム型」 p.33 を参照してください。

ウィンドウはリード構文をもちません。ウィンドウは、ハッシュ表記で印字し、それにはウィンドウ番号と、表示しているバッファの名前がつきます。ウィンドウに表示されているバッファは頻繁に変わるので。ウィンドウを一意に識別するために、ウィンドウ番号が用意されています。

```
(selected-window)
     ⇒ #<window 1 on objects.texi>
```

ウィンドウに対する関数の説明は、第 25 章「ウィンドウ」 p.407 を参照してください。

2.4.4 フレーム型

『フレーム (frame)』は、一つ以上の Emacs ウィンドウを含む長方形の画面です。フレームは最初はメインとなるウィンドウを一つ含みます (ミニバッファ・ウィンドウもあるかもしれませんが)。そのウィンドウは、より小さなウィンドウに垂直または水平に再分割することができます。

フレームはリード構文をもちません。フレームはハッシュ表記で印字し、それにはフレームのタイトル、主記憶のアドレス (フレームを一意に識別するために有用) がつきます。

```
(selected-frame)
     ⇒ #<frame xemacs@mole.gnu.ai.mit.edu 0xdac80>
```

フレームに対する関数の説明は 第 26 章「フレーム」 p.435 を参照してください。

2.4.5 ウィンドウ構成型

『ウィンドウ構成 (window configuration)』は、フレーム内のウィンドウの位置、サイズ、内容に関する情報を記憶し、後でウィンドウを同じ配置で再生できるようにします。

ウィンドウ構成はリード構文をもちません。ウィンドウ構成は '#<window-configuration>' のように印字します。ウィンドウ構成に関係するいくつかの関数の説明は、 25.16 節「ウィンドウ構成」 p.432 を参照してください。

2.4.6 プロセス型

『プロセス (process)』という用語は普通、実行しているプログラムを意味します。 Emacs 自身は、この種のプロセスで動作しています。しかし、 Emacs Lisp においてプロセスとは、 Emacs プロセスによって作成されたサブプロセスを示している Lisp オブジェクトです。シェル、 GDB、 ftp、コンパイラなどのプログラムは、 Emacs のサブプロセスで動作し、 Emacs の機能を拡張します。

Emacs のサブプロセスは、 Emacs からのテキスト入力を受け取り、それ以降の操作のために Emacs にテキスト出力を返します。 Emacs はサブプロセスにシグナルも送ることができます。

プロセス・オブジェクトはリード構文を持ちません。プロセス・オブジェクトは、ハッシュ表記で印字し、それにはプロセスの名前がつきます。

```
(process-list)
     ⇒ (#<process shell>)
```

プロセスを作成する関数、プロセスを消去する関数、プロセスに関する情報を返す関数、入力やシグナルを送る関数、出力を受け取る関数についての情報は、第 33 章「プロセス」 p.559 を参照してください。

2.4.7 ストリーム型

『ストリーム (stream)』は文字の並びのソース、またはシンクとして使用するオブジェクトです。ソースは入力に文字を供給するためのものであり、シンクは出力として文字を受け取るためのものです。多くの異なる型 (マーカ、バッファ、ストリング、関数) は、この方法で使用されます。典型的には、入力ストリーム (文字ソース) は、キーボード、バッファ、またはファイルから文字を受け取り、出力ストリーム (文字シンク) は、バッファ (たとえば '*Help*' バッファのようなもの)、またはエコー領域に文字を送ります。

オブジェクト `nil` は、その他の意味に加えて、ストリームとしても使用できます。それは変数 `standard-input` または `standard-output` の値を表してします。また、ストリームとしてのオブジェクト `t` は、ミニバッファ (第 17 章「ミニバッファ」 p.237 参照) を使用した入力、または、エコー領域 (35.4 節「エコー領域」 p.603 参照) を指定した出力を指定します。

ストリームは、特殊な印字表現やリード構文を持たず、そのプリミティブ型がなんであれ、その型として印字されます。

解析や印字の関数など、ストリームに関係する関数の説明は、第 16 章「Lisp オブジェクトの読込みと印字」p.227 を参照してください。

2.4.8 キーマップ型

『キーマップ (keymap)』はユーザによってタイプされるキーをコマンドに割り当てます。この割り当てはユーザのコマンド入力がどのように実行されるかを制御します。キーマップは実際には CAR がシンボル `keymap` であるリストです。

キーマップの作成、前置キーの処理、グローバル・キーマップとローカル・キーマップ、キー・バインドの変更に関する情報は、第 19 章「キーマップ」p.293 を参照してください。

2.4.9 構文テーブル型

『構文テーブル (syntax table)』は 256 個の整数のベクタです。ベクタの個々の要素は、ある文字がバッファ上に現れた場合に、どのように解釈されるかを定義します。例えば、 C モード (20.1 節「主モード」p.317 参照) では文字 '+' は句読点ですが、 Lisp モードではシンボルに使用できる文字です。これらのモードは、構文テーブル内の添字 43 にある '+' のためのエントリを変更することによって、異なる解釈を指定します。

構文テーブルは、バッファ内のテキストをスキャンするためだけに使用され、 Lisp 式の読み込みには使用されません。式の読み込みに Lisp インタプリタが使用するテーブルは Emacs のソース・コードに組み込まれていて、変更することはできません。したがって、リストの範囲を定める '(' と ')' の代わりに '{' と '}' を使用できるようにすることはできません。

構文クラスと構文テーブルの更新方法に関する詳細は、第 31 章「構文テーブル」p.541 を参照してください。

2.4.10 表示テーブル型

『表示テーブル (display table)』は、個々の文字コードを表示する方法を指定します。個々のバッファおよび個々のウィンドウは、それ自身の表示テーブルをもつことができます。表示テーブルは実際には、長さ 262 のベクタです。 35.14 節「表示テーブル」p.618 を参照してください。

2.4.11 オーバレイ型

『オーバレイ (overlay)』は、バッファの一部分の表示上の外観の一時的な変更を指定します。それはバッファのある範囲の境界を定めるマーカを含み、加えて属性リスト (属性の名前と値を交互に要素としているリスト) も含んでいます。オーバレイは一時的に異なった表示スタイルでバッファの一部を表示するために使用されます。オーバレイは、リード構文をもたず、バッファ名と位置の範囲のついたハッシュ表記で印字します。

オーバレイの作成、使用の方法に関しては、 35.9 節「オーバレイ」 p.609 を参照してください。

2.5 型述語

Emacs Lisp インタプリタ自身は、関数が呼ばれたときに渡された実引数に対して型チェックを行ないません。型チェックが行えないのは、 Lisp の関数の引数に、他のプログラミング言語にあるようなデータ型の宣言がないためです。したがって、各実引数が、その関数が利用できる型に属しているかをテストするのは、個々の関数の責任です。

すべての組込み関数は、適当なときにその実引数の型をチェックし、引数が不正な型であれば、エラー `wrong-type-argument` を発します。例えば、以下は + が扱うことのできない引数を渡した場合に何が起きるかを示しています。

```
(+ 2 'a)
     error  Wrong type argument: integer-or-marker-p, a
```

型ごとに異なった処理を行うプログラムが欲しいのであれば、あなたは明示的に型チェックを行なわなければなりません。オブジェクトの型をチェックする最もありふれた方法は、『型述語 (type predicate)』関数を呼び出すことです。 Emacs は、型の組合せのための述語の他にも、個々の型のための型述語を持っています。

型述語関数は、一つの引数をとります。その引数が適切な型に属していれば `t` を返し、そうでなければ `nil` を返します。述語関数についての一般的な Lisp の慣例にしたがって、ほとんどの型述語の名前は 'p' で終ります。

以下は、リストであることをチェックする述語 `listp` と、シンボルであることをチェックする `symbolp` を使用した例です。

```
(defun add-on (x)
  (cond ((symbolp x)
         ;; x がシンボルであれば、それを list に加える。
         (setq list (cons x list)))
        ((listp x)
         ;; x がリストであれば、それを list に加える。
         (setq list (append x list)))
```

```
      (t
       ;; シンボルとリストだけを処理する。
       (error "Invalid argument %s in add-on" x))))
```

以下は、既定義の型述語をアルファベット順でならべた一覧で、詳細な情報の参照先を示しています。

`atom` 5.3 節「リストに関する述語」p.68 を参照してください。

`arrayp` 6.3 節「配列に関する関数」p.88 を参照してください。

`bufferp` 24.1 節「バッファの基本」p.393 を参照してください。

`byte-code-function-p`
2.3.13 節「バイトコード関数型」p.31 を参照してください。

`case-table-p`
4.8 節「大文字小文字変換のカスタマイズ」p.65 を参照してください。

`char-or-string-p`
4.2 節「文字列の述語」p.56 を参照してください。

`commandp` 18.3 節「対話的呼出し」p.262 を参照してください。

`consp` 5.3 節「リストに関する述語」p.68 を参照してください。

`floatp` 3.3 節「数値型にたいする述語」p.42 を参照してください。

`frame-live-p`
26.5 節「フレームの消去」p.443 を参照してください。

`framep` 第 26 章「フレーム」p.435 を参照してください。

`integer-or-marker-p`
28.2 節「マーカに対する述語」p.472 を参照してください。

`integerp` 3.3 節「数値型にたいする述語」p.42 を参照してください。

`keymapp` 19.3 節「キーマップの作成」p.295 を参照してください。

`listp` 5.3 節「リストに関する述語」p.68 を参照してください。

`markerp` 28.2 節「マーカに対する述語」p.472 を参照してください。

`wholenump` 3.3 節「数値型にたいする述語」p.42 を参照してください。

`nlistp` 5.3 節「リストに関する述語」p.68 を参照してください。

`numberp` 3.3 節「数値型にたいする述語」p.42 を参照してください。

`number-or-marker-p`
28.2 節「マーカに対する述語」p.472 を参照してください。

`overlayp` 35.9 節「オーバレイ」p.609 を参照してください。

`processp` 第 33 章「プロセス」p.559 を参照してください。

`sequencep` 6.1 節「シーケンス」p.85 を参照してください。

`stringp` 4.2 節「文字列の述語」p.56 を参照してください。

`subrp` 11.8 節「関数セルの内容へのアクセス」p.158 を参照してください。

`symbolp` 第 7 章「シンボル」p.93 を参照してください。

`syntax-table-p`
第 31 章「構文テーブル」p.541 を参照してください。

`user-variable-p`
10.5 節「グローバル変数の定義」p.131 を参照してください。

`vectorp` 6.4 節「ベクタ」p.89 を参照してください。

`window-configuration-p`
25.16 節「ウィンドウ構成」p.432 を参照してください。

`window-live-p`
25.3 節「ウィンドウの消去」p.411 を参照してください。

`windowp` 25.1 節「Emacs ウィンドウの基本概念」p.407 を参照してください。

オブジェクトの型をチェックする最も一般的な方法は、関数 `type-of` を呼び出すことです。個々のオブジェクトは、ただ一つのプリミティブ型に属していることを思い出してください。`type-of` は、その一つをあなたに知らせます (第 2 章「Lisp のデータ型」p.19 参照)。しかし、`type-of` は非プリミティブ型については何も知りません。たいていの場合、`type-of` よりも型述語を使用したほうが便利です。

type-of *object* Function

この関数は、 *object* のプリミティブ型の名前を表すシンボルを返します。その値は `symbol`, `integer`, `float`, `string`, `cons`, `vector`, `marker`, `overlay`, `window`, `buffer`, `subr`, `compiled-function`, `window-configuration`, or `process` のうちの一つです。

```
(type-of 1)
     ⇒ integer
(type-of 'nil)
     ⇒ symbol
(type-of '())    ; () is nil.
     ⇒ symbol
(type-of '(x))
     ⇒ cons
```

2.6 等価述語

ここでは、任意の二つのオブジェクトの等価性をテストする二つの関数について述べます。それとは別に、特定の型 (文字列など) のオブジェクトの等価性をテストする関数もあります。そのような述語に関しては、そのデータ型について述べられている適切な章を参照してください。

eq *object1 object2* Function

この関数は *object1* と *object2* が同じオブジェクトであれば、 t を返し、そうでなければ nil を返します。「同じオブジェクト」であるということは、一方を変更すると他方にそれと同じ変更が反映されることを意味しています。

eq は *object1* と *object2* が同じ値をもつ整数であれば、 t を返します。また、シンボル名は、普通一意であるので、引数が同じ名前をもつシンボルであれば、それらは eq です。その他の型 (リスト、ベクタ、文字列など) では、内容または要素が同じである二つ引数が、必ずしも互いに等価であるわけではありません。それらが同じオブジェクトである場合に限って、それらは eq です。

(make-symbol 関数は標準の obarray に intern されていないシンボルを返します。 intern されていないシンボルが使用されているときには、シンボル名はもう一意ではありません。同じ名前をもつ別々のシンボルは eq ではありません。 7.3 節「シンボルの作成と intern」p.95 を参照してください。)

```
(eq 'foo 'foo)
     ⇒ t

(eq 456 456)
     ⇒ t

(eq "asdf" "asdf")
     ⇒ nil

(eq '(1 (2 (3))) '(1 (2 (3))))
     ⇒ nil

(setq foo '(1 (2 (3))))
     ⇒ (1 (2 (3)))
(eq foo foo)
     ⇒ t
(eq foo '(1 (2 (3))))
     ⇒ nil

(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil

(eq (point-marker) (point-marker))
     ⇒ nil
```

equal *object1 object2* Function

この関数は *object1* と *object2* が同じ成分をもっているならば、 t を返し、そうでなければ nil を返します。 eq が、その引数が同一のオブジェクトであるかをテストするのに対し、 equal は、要素が同一であるかを判別するために、同一ではない引数の中身を調べていきます。したがって、もし二つのオブジェクトが eq であるならば、それらは equal ですが、逆は常に真とはかぎりません。

```
(equal 'foo 'foo)
     ⇒ t
```

```
(equal 456 456)
     ⇒ t
(equal "asdf" "asdf")
     ⇒ t
(eq "asdf" "asdf")
     ⇒ nil
(equal '(1 (2 (3))) '(1 (2 (3))))
     ⇒ t
(eq '(1 (2 (3))) '(1 (2 (3))))
     ⇒ nil
(equal [(1 2) 3] [(1 2) 3])
     ⇒ t
(eq [(1 2) 3] [(1 2) 3])
     ⇒ nil
(equal (point-marker) (point-marker))
     ⇒ t
(eq (point-marker) (point-marker))
     ⇒ nil
```

文字列の比較では大文字、小文字が区別され、文字列中の文字の他にもテキスト属性を考慮に入れます。テキスト属性を比較することなく、二つの文字列の文字を比較するためには `string=` を使用してください (4.4 節「文字・文字列の比較」p.58 参照)。

```
(equal "asdf" "ASDF")
     ⇒ nil
```

二つの別々のバッファは、たとえその内容が同じであっても、 `equal` になることはありません。

等価性のテストは再帰的に実装されているので、循環リストでは (エラーをもたらす) 無限の再帰を引き起こすかもしれません。

3 数値

GNU Emacs は『整数 (integer)』と『浮動小数点数 (floating point number)』の二つの数値データ型をサポートします。整数は、 -3、 0、 7、 13、 511 のような数全部です。整数の値は、きっちりとその表現の値を表します。浮動小数点数は、 -4.5、 0.0、 2.71828 のように小数部のある数です。浮動小数点数は、指数表記法で表現することもできます。 1.5e2 は 150 と等しいです。この例の ‘e2’ は 10 の自乗を表わし、それを 1.5 倍しています。浮動小数点の値は厳密ではありません。固定で有限量の精度をもっています。

浮動小数点数のサポートは Emacs 19 の新機能です。それは独立したコンパイル時オプションで制御されます。そのためサイトによっては、 Emacs が浮動小数点をサポートしていないこともあるかもしれません。

3.1 整数の基本事項

整数値の範囲は、マシンに依存します。最低でも、範囲は -134217728 から 134217727 まで (28 ビット、すなわち -2^{27} から $2^{27}-1$ まで) ですが、マシンによってはもっと広い範囲をとることもあります。この章でとりあげる多くの例では、整数が 28 ビットと仮定しています。

Lisp リーダは、先行する省略可能な符号と最後の省略可能なピリオドを含んだ数字の列として、整数を読み込みます。

```
 1               ; 整数の 1。
 1.              ; 整数の 1。
+1               ; これも整数の 1。
-1               ; 整数の -1。
 268435457       ; 桁あふれ (overflow) のため、これも整数の 1。
 0               ; 整数の 0。
-0               ; 整数の 0。
```

ビットごとの演算 (3.8 節「整数にたいするビットごとの演算」 p.49 参照) などは特にそうですが、いろいろな整数についての関数のはたらきを理解するには、数を 2 進形式でながめるのが便利です。

28 桁の 2 進数では、 10 進整数の 5 は以下のようになります。

```
0000  0000 0000  0000 0000  0000 0101
```

(2 進整数を読みやすくするため、 4 桁ごとにスペースを一つ、 8 桁ごとにスペースを二つあけてあります。)

整数の -1 は以下のようになります。

```
1111  1111 1111  1111 1111  1111 1111
```

28 個の 1 で -1 を表わします (これを『2 の補数 (two’s complement)』表記といいます)。

負の整数の -5 は -1 から 4 を引いてできます。 10 進整数の 4 は 2 進数では 100 です。したがって -5 は以下のようになります。

```
1111  1111 1111  1111 1111  1111 1011
```

この実装方法では、 28 桁 2 進整数の最大値は、 10 進整数の 134,217,727 になります。これは 2 進数では以下のようになります。

```
0111  1111 1111  1111 1111  1111 1111
```

算術関数は整数範囲外のチェックをしないので、 134,217,727 に 1 を足すと値は負数の $-134{,}217{,}728$ になります。

```
(+ 1 134217727)
     ⇒ -134217728
     ⇒ 1000  0000 0000  0000 0000  0000 0000
```

以下の多くの関数は、整数と同様にマーカも引数としてうけつけます (第 28 章「マーカ」 p.471 を参照してください)。より厳密には、そのような関数の実引数は、整数でもマーカでもかまいません。これがこのような引数の名前を *int-or-marker* とする理由です。引数の値がマーカのとき、マーカの位置の値が用いられ、マーカのバッファは無視されます。

3.2 浮動小数点の基本事項

マクロの `LISP_FLOAT_TYPE` を定義してコンパイルすると、 Emacs バージョン 19 は浮動小数点数をサポートするようになります。浮動小数点数の正確な範囲は、マシン個有です。そのマシンの C のデータ型 `double` と同じ範囲です。

浮動小数点数の印字表現は、 (少なくとも一つの数字が続いた) 小数点か指数、あるいはその両方を含んでいる必要があります。たとえば、‘`1500.0`’、‘`15e2`’、‘`15.0e2`’、‘`1.5e3`’、‘`.15e4`’ は、 1500 という値の浮動小数点数を、 5 とおりに書き表わしたものです。 ‘`-1.0`’ のような負の浮動小数点数を書くのに負符号が使えます。

最近の多くの計算機では、 IEEE の浮動小数点標準をサポートしています。この標準には、浮動小数点値に加えて正の無限大と負の無限大があります。また、 NaN あるいは「非数」 (“not-a-number”) という値のクラスがあり、数値関数は正解のないときこのような値を返します。たとえば、 `(sqrt -1.0)` は NaN を返します。実際の問題としては、 Emacs Lisp では異なる NaN 値間の違いは重要ではありません。どの場合にどの NaN 値が使われるべきか、ということに関する正確な規則はありません。そのためこのマニュアルではこれらを区別しないことにします。 Emacs Lisp には、 NaN や無限大に対するリード構文はありません。多分将来には、構文を作るべきでしょう。

`logb` を使って、浮動小数点数の 2 進指数部をとりだすことができます (整数の対数を見積もることができます)。

logb *number* Function

この関数は、 *number* の 2 進指数部を返します。正確に書くと、この値は、 2 を底にした *number* の対数を整数値に切り捨てたものです。

3.3 数値型にたいする述語

この節で説明する関数は、引数が数であるか、またそれがどんな種類の数であるかを調べるのに使います。関数 `integerp` と `floatp` には、引数としてどんな型の Lisp オブジェクトでも

与えることができます (そうでなければ述語というものはあまり使えないでしょう)。しかし、zerop 述語には、引数として数だけが許されます。28.2 節「マーカに対する述語」p.472 の、integer-or-marker-p と number-or-marker-p を参照してください。

floatp *object* Function

この述語は、引数が浮動小数点数であるか否か調べ、もしそうなら t を、そうでなければ nil を返します。

floatp はバージョン 18 やそれ以前の Emacs にはありません。

integerp *object* Function

この述語は、引数が整数であるか否かを調べ、もしそうなら t を返し、そうでなければ nil を返します。

numberp *object* Function

この述語は、引数が数 (整数か浮動小数点かどちらか) であるか否かを調べ、もしそうなら t を、そうでなければ nil を返します。

wholenump *object* Function

wholenump 述語 ("whole-number-p" という句から名づけられました) は、引数が非負の整数であるか否かを調べ、もしそうなら t を、そうでなければ nil を返します。 0 は非負とみなします。

natnump は wholenump の古い別称です。

zerop *number* Function

この述語は、引数が 0 であるか否かを調べ、もしそうなら t を、そうでなければ nil を返します。引数は数でなければなりません。

次の二つの形式は等価です。(zerop x) ≡ (= x 0)

3.4 数値の比較

数をその数値としての等価性について調べるには、 eq を使わず、普通は = を使います。同一の数値になる、多くの別々な浮動小数点数のオブジェクトがあり得ます。それらを eq で比べる場合、二つの値が同一のオブジェクトかどうかを調べることになりますが、それとは対照に、ただ数としての値を比べたいなら = を使います。

今のところ、 Emacs Lisp の中では、整数値は一意な Lisp オブジェクトです。したがって、整数を比較する場合には、 eq は = と等価です。 eq は、未知の値が数でなくともエラーをおこさず、すべての型の引数をうけつけるので、未知の値と整数を比較するのに eq を使うと便利なこともあります。反対に、 = は、引数が数でもマーカでもないとエラーを通知します。しかしながら、 Emacs の将来のバージョンでは、整数の内部表現を変える可能性があるので、整数をくらべるときにも、できるだけ = を使う方がいいでしょう。

ここで助言があります。浮動小数点の演算は厳密ではないので、二つの浮動小数点値の間で、等価性を調べるのは考えものです。多くの場合、近い値かどうかを調べる方がいいでしょう。以下に、これを実行する関数をあげます。

```
(defvar fuzz-factor 1.0e-6)
(defun approx-equal (x y)
  (or (and (= x 0) (= y 0))
      (< (/ (abs (- x y))
            (max (abs x) (abs y)))
         fuzz-factor)))
```

Common Lisp 注意書き: Common Lisp では、多語整数があるので、常に数の比較には `=` を使う必要があります。二つの異なる整数オブジェクトが同一の数値をもつことがありうるからです。 Emacs Lisp では整数値の範囲を制限しているため、与えられたすべての値にたいする整数オブジェクトが一意に定まります。

= *number-or-marker1 number-or-marker2* Function

この関数は、引数が同じ数値であるか否かを調べ、同じ数値である場合 `t` を返し、そうでない場合 `nil` を返します。

/= *number-or-marker1 number-or-marker2* Function

この関数は、引数が同じ数値であるか否かを調べ、同じ数値でない場合 `t` を返し、同じ数値である場合 `nil` を返します。

< *number-or-marker1 number-or-marker2* Function

この関数は、最初の引数が 2 番目の引数より厳密に小さいか否かを調べます。小さい場合 `t` を返し、そうでない場合 `nil` を返します。

<= *number-or-marker1 number-or-marker2* Function

この関数は、最初の引数が 2 番目の引数より小さいか等しいかを調べます。小さいか等しい場合 `t` を返し、そうでない場合 `nil` を返します。

> *number-or-marker1 number-or-marker2* Function

この関数は、最初の引数が 2 番目の引数より厳密に大きいか否かを調べます。大きい場合 `t` を返し、そうでない場合 `nil` を返します。

>= *number-or-marker1 number-or-marker2* Function

この関数は、最初の引数が 2 番目の引数より大きいか等しいかを調べます。大きいか等しい場合 `t` を返し、そうでない場合 `nil` を返します。

max *number-or-marker* &rest *numbers-or-markers* Function

この関数は、引数の最大値を返します。

```
(max 20)
     ⇒ 20
(max 1 2.5)
     ⇒ 2.5
(max 1 3 2.5)
     ⇒ 3
```

min *number-or-marker* &rest *numbers-or-markers* Function

この関数は、引数の最小値を返します。

```
(min -4 1)
     ⇒ -4
```

3.5 数の変換

整数を浮動小数点に変換するには、関数 `float` を使います。

float *number* Function

これは、浮動小数点に変換した *number* を返します。もし、 *number* がすでに浮動小数点数なら、 `float` は変更せずに返します。

浮動小数点数を整数に変換する四つの関数があります。丸め方が違っています。整数の引数を与えてこれらの関数を呼んでもかまいません。そのときは引数を変更しないで返します。

truncate *number* Function

これは、 *number* を 0 の方向へ丸めた整数に変換し、返します。

floor *number* &optional *divisor* Function

これは、 *number* を小さくなる (負の無限大の) 方向へ丸めた整数に変換し、返します。

divisor を指定すると、 *number* を *divisor* で割ってからその floor をとります。これは、 `mod` にたいする除算に相当します。 *divisor* が 0 なら `arith-error` を発生します。

ceiling *number* Function

これは、 *number* を大きくなる (正の無限大の) 方向へ丸めた整数に変換し、返します。

round *number* Function

これは、 *number* を一番近い整数へ丸めて整数に変換し、返します。となりあった整数の中間の値を丸めようとすると、マシンに依存して、 0 に近い整数か、偶数の方向へと丸めます。

3.6 算術演算

Emacs Lisp は伝統的な四則演算、つまり加減乗除を提供します。除算に加えて、剰余と法の関数も用意してあります。 Lisp では伝統的であり、またよく使われるため、 1 を加減算する関数もあります。

`%` 以外の関数は、引数の中に浮動小数点値のものがあると浮動小数点値を返します。

重要なことですが、 GNU Emacs Lisp の算術関数では桁あふれのチェックは行ないません。ハードウェアによっては、 `(1+ 134217727)` は、 -134217728 と評価されるでしょう。

1+ *number-or-marker* Function

この関数は、 *number-or-marker* 足す 1 を返します。たとえば、

```
(setq foo 4)
     ⇒ 4
```

```
(1+ foo)
     ⇒ 5
```

この関数は、 C の ++ 演算子とは違います。変数をインクリメントさせません。和を計算するだけです。したがって、その後でもこうなります。

```
foo
     ⇒ 4
```

もし、変数をインクリメントさせたいなら、次のように setq を使わなければなりません。

```
(setq foo (1+ foo))
     ⇒ 5
```

1- *number-or-marker* Function

この関数は、 *number-or-marker* 引く 1 を返します。

abs *number* Function

これは、 *number* の絶対値を返します。

+ &rest *numbers-or-markers* Function

この関数は、引数を足しあわせます。引数が与えられない場合、 + は、 0 を返します。

```
(+)
     ⇒ 0
(+ 1)
     ⇒ 1
(+ 1 2 3 4)
     ⇒ 10
```

- &optional *number-or-marker* &rest *other-numbers-or-markers* Function

関数 - には、二つの目的があります。反数 (はんすう) と減算です。 - に単一の引数を与えたとき、値は引数の反数です。複数の引数が存在する場合、 *number-or-marker* から全部の *other-numbers-or-markers* を次々に引きます。引数が存在しない場合、結果は 0 になります。

```
(- 10 1 2 3 4)
     ⇒ 0
(- 10)
     ⇒ -10
(-)
     ⇒ 0
```

***** &rest *numbers-or-markers* Function

この関数は、引数を掛けあわせて積を返します。引数が与えられない場合、 * は、 1 を返します。

```
(*)
     ⇒ 1
(* 1)
```

```
     ⇒ 1
(* 1 2 3 4)
     ⇒ 24
```

/ *dividend divisor* &rest *divisors* Function

この関数は、 *dividend* を *divisor*(除数) で割り、商を返します。もし省略可能な引数 *divisors* が存在する場合は、 *dividend* を各除数で次々に割っていきます。各引数は数でもマーカでもかまいません。

全部の引数が整数なら、結果も整数になります。つまり結果は丸められます。ほとんどのマシンでは、すべての除算の後、 0 の方向へ丸められます。しかし、あるマシンでは負の引数のとき異なる丸め方をすることがありえます。これは、 Lisp 関数 / が、マシン依存の丸め方を許す C の除算演算子を使って実装されているためです。実際にはよく知られたマシンでは標準的な方法で丸められます。

0 で割ると `arith-error` を発生します。 (9.5.3 節「エラー」 p.119 を参照してください。)

```
(/ 6 2)
     ⇒ 3
(/ 5 2)
     ⇒ 2
(/ 25 3 2)
     ⇒ 4
(/ -17 6)
     ⇒ -2
```

原理的には `(/ -17 6)` の結果は、あるマシンでは -3 になりえます。

% *dividend divisor* Function

この関数は、 *dividend* を *divisor* で割ったあとの整数剰余 (integer remainder) を返します。引数は、整数かマーカでなければなりません。

負の引数については、原理的には商がマシン依存のため、余りもマシン依存になります。しかし実際は、すべてのよく使われるマシンでは同じように振る舞います。

`divisor` が 0 なら `arith-error` になります。

```
(% 9 4)
     ⇒ 1
(% -9 4)
     ⇒ -1
(% 9 -4)
     ⇒ 1
(% -9 -4)
     ⇒ -1
```

任意の二つの整数 *dividend* と *divisor* について、

```
(+ (% dividend divisor)
   (* (/ dividend divisor) divisor))
```

は、常に *dividend* と等しいです。

mod *dividend divisor* Function

この関数は、 *divisor* を法 (ほう) とした *dividend* の値を返します。いいかえると、 *divisor* と同じ符号にした *dividend* の、 *divisor* による除算の剰余です。引数は数かマーカでなければなりません。

`%` とは異なり、 `mod` は、負の引数についても、よく定義された結果を返します。そして浮動小数点の引数も許されています。余りを計算するときに、商を整数へ小さくなる (負の無限大の) 方向へ丸めた値を用います。

divisor が 0 なら `arith-error` になります。

```
(mod 9 4)
     ⇒ 1
(mod -9 4)
     ⇒ 3
(mod 9 -4)
     ⇒ -3
(mod -9 -4)
     ⇒ -1
(mod 5.5 2.5)
     ⇒ .5
```

任意の二つの数 *dividend* と *divisor* について、以下の

```
(+ (mod dividend divisor)
   (* (floor dividend divisor) divisor))
```

は、引数が浮動小数点の場合の丸め誤差の範囲で、常に *dividend* に等しいです。 `floor` については、 3.5 節「数の変換」 p.45 を参照してください。

3.7 丸め演算

関数 `ffloor`、 `fceiling`、 `fround`、 `ftruncate` は、浮動小数点の引数をとり、整数に最も近い浮動小数点値の結果を返します。 `ffloor` は最も近い小さな整数を返します。 `fceiling` は最も近い大きな整数です。 `ftruncate` は 0 の方向にむかって最も近い整数です。 `fround` は最も近い整数です。

ffloor *float* Function

この関数は、 *float* の次に小さい整数へ丸めた値を、浮動小数点数として返します。

fceiling *float* Function

この関数は、 *float* の次に大きい整数に丸めた値を、浮動小数点数として返します。

ftruncate *float* Function

この関数は、 *float* を 0 の方向へ丸めた整数に変換した値を、浮動小数点数として返します。

fround *float* Function

この関数は、 *float* に一番近い整数へ丸めた値を、浮動小数点数として返します。

3.8 整数にたいするビットごとの演算

計算機内部では、整数は、 2 進数、すなわち (0 と 1 の数字になった) 『ビット (bit)』・シーケンスで表現され、ビットごとの演算は、このようなシーケンスの個々のビットに作用します。たとえば、『桁送り (shifting)』は、ビット・シーケンス全体を左右に一つ、あるいはもっと、移動させて、同じパターンを「移動した」のに等しいものを作りだします。

Emacs Lisp のビット演算は、整数に対してのみ適用します。

lsh *integer1 count* Function

`lsh` (『logical shift』の省略形) は、 *integer1* 中のビットを左 (*count* が負の場合は右) に、 *count* だけ桁送りし、あいたビットに 0 を入れます。 `count` が負の場合、 `lsh` は一番左 (最高位) のビットに 0 を入れるので、 *integer1* が負のときの結果は正になります。これは後述の `ash` とは異なっている点です。

次の二つの例では、ビットのパターンを一つ左に桁送りしています。ここでは低位の 8 桁だけを示しています。残りのビットはすべて 0 です。

```
(lsh 5 1)
     ⇒ 10
;; 10 進の 5 が 10 進の 10 になる。
00000101 ⇒ 00001010

(lsh 7 1)
     ⇒ 14
;; 10 進の 7 が 10 進の 14 になる。
00000111 ⇒ 00001110
```

例で説明したように、ビットのパターンを一つ左に桁送りすると、元の数の 2 倍の数になります。

ビットのパターンを二つ左に桁送りするとこうなります (8 桁の 2 進数)。

```
(lsh 3 2)
     ⇒ 12
;; 10 進の 3 が 10 進の 12 になる。
00000011 ⇒ 00001100
```

一方、一つ右に桁送りするとこうなります。

```
(lsh 6 -1)
     ⇒ 3
;; 10 進の 6 が 10 進の 3 になる。
00000110 ⇒ 00000011

(lsh 5 -1)
     ⇒ 2
;; 10 進の 5 が 10 進の 2 になる。
00000101 ⇒ 00000010
```

例で説明したように、一つ右に桁送りすると、正の整数を 2 で割り小さく丸めた値になります。

関数 `lsh` は、 Emacs Lisp の全算術関数と同様に、桁あふれのチェックは行ないません。したがって左への桁送りは高位ビットを捨て去り、数の符号を変えることがあります。28 ビット・マシンでは、134,217,727 を左に桁送りすると、 -2 になります。

```
(lsh 134217727 1)          ; 左に桁送りする
     ⇒ -2
```

28 ビットの実装では、引数の 2 進表示は、以下のようになります。

```
;; 10 進の 134,217,727
0111  1111 1111  1111 1111  1111 1111
```

これを左に桁送りしたのが次のものです。

```
;; 10 進の −2
1111  1111 1111  1111 1111  1111 1110
```

ash *integer1 count* Function

`ash` (『arithmetic shift』) は、 *integer1* 中のビットを左 (*count* が負の場合は右) に、*count* だけ桁送りします。

`ash` は、 *integer1* と *count* が両方とも負である場合をのぞき `lsh` と同じ結果を返します。この場合、 `lsh` は、桁送りを行なう際、その左側から 0 を入れますが、 `ash` は、桁送りを行なう際、そこに 1 を入れます。

したがって、あるビットパターンを `ash` で右に一つ桁送りすると、以下のようになります。

```
(ash -6 -1) ⇒ -3
;; 10 進の −6 が 10 進の −3 になる。
1111  1111 1111  1111 1111  1111 1010
     ⇒
1111  1111 1111  1111 1111  1111 1101
```

逆に、そのビットパターンを `lsh` で一つ右に桁送りすると、こうなります。

```
(lsh -6 -1) ⇒ 134217725
;; 10 進の −6 が 10 進の 134,217,725 になる。
1111  1111 1111  1111 1111  1111 1010
     ⇒
0111  1111 1111  1111 1111  1111 1101
```

ほかにも例をあげます。

```
                   ;           28 桁 2 進値

(lsh 5 2)          ;   5  =  0000  0000 0000  0000 0000  0000 0101
     ⇒ 20          ;      =  0000  0000 0000  0000 0000  0001 0100
(ash 5 2)
     ⇒ 20
(lsh -5 2)         ;  -5  =  1111  1111 1111  1111 1111  1111 1011
     ⇒ -20         ;      =  1111  1111 1111  1111 1111  1110 1100
(ash -5 2)
     ⇒ -20
(lsh 5 -2)         ;   5  =  0000  0000 0000  0000 0000  0000 0101
     ⇒ 1           ;      =  0000  0000 0000  0000 0000  0000 0001
```

```
(ash 5 -2)
     ⇒ 1
(lsh -5 -2)         ;  -5  =  1111  1111 1111  1111 1111  1111 1011
     ⇒ 4194302      ;      =  0011  1111 1111  1111 1111  1111 1110
(ash -5 -2)         ;  -5  =  1111  1111 1111  1111 1111  1111 1011
     ⇒ -2           ;      =  1111  1111 1111  1111 1111  1111 1110
```

logand &rest *ints-or-markers* Function

この関数は、引数の「論理積」を返します。結果の第 n ビットは、全ての引数で第 n ビットがセットされている場合にのみ、セットされます (「セット」とは、ビットの値が 0 でなく 1 にすることです)。

たとえば、 4 桁の 2 進数で、 13 と 12 の「論理積」は 12 です。 1101 と 1100 で 1100 になります。両方の 2 進数で左側の二ビットがセットされている (値が 1) ので、戻り値の左の二つのビットはセットされます。けれども、右側の二ビットでは、少なくともどちらかの引数が 0 なので、戻り値の右側の二ビットは 0 になります。

ですから、

```
(logand 13 12)
     ⇒ 12
```

引数が存在しない場合、結果は -1 になります。この数の 2 進表現は全ビットが 1 なので、 `logand` の単位元です。 `logand` にたいし一つしか引数が存在しない場合、その引数が結果となります。

```
                    ;          28 桁 2 進値

(logand 14 13)      ; 14  =  0000  0000 0000  0000 0000  0000 1110
                    ; 13  =  0000  0000 0000  0000 0000  0000 1101
     ⇒ 12           ; 12  =  0000  0000 0000  0000 0000  0000 1100
(logand 14 13 4)    ; 14  =  0000  0000 0000  0000 0000  0000 1110
                    ; 13  =  0000  0000 0000  0000 0000  0000 1101
                    ;  4  =  0000  0000 0000  0000 0000  0000 0100
     ⇒ 4            ;  4  =  0000  0000 0000  0000 0000  0000 0100
(logand)
     ⇒ -1           ; -1  =  1111  1111 1111  1111 1111  1111 1111
```

logior &rest *ints-or-markers* Function

この関数は、引数の「論理和」を返します。結果の第 n ビットは、少なくとも一つの引数で第 n ビットがセットされている場合にのみセットされます。引数が存在しない場合、結果はこの演算の単位元である 0 になります。 `logior` にたいし一つしか引数が存在しない場合、その引数が結果となります。

```
                    ;          28 桁 2 進値

(logior 12 5)       ; 12  =  0000  0000 0000  0000 0000  0000 1100
                    ;  5  =  0000  0000 0000  0000 0000  0000 0101
     ⇒ 13           ; 13  =  0000  0000 0000  0000 0000  0000 1101
(logior 12 5 7)     ; 12  =  0000  0000 0000  0000 0000  0000 1100
                    ;  5  =  0000  0000 0000  0000 0000  0000 0101
                    ;  7  =  0000  0000 0000  0000 0000  0000 0111
     ⇒ 15           ; 15  =  0000  0000 0000  0000 0000  0000 1111
```

logxor &rest *ints-or-markers* Function

この関数は、引数の「排他的論理和」を返します。結果の第 n ビットは、奇数個の引数で第 n ビットがセットされている場合にのみセットされます。引数が存在しない場合、結果はこの演算の単位元である 0 になります。 `logxor` にたいし一つしか引数が存在しない場合、その引数が結果となります。

```
                   ;        28 桁 2 進値

(logxor 12 5)      ; 12  =  0000  0000 0000  0000 0000  0000 1100
                   ;  5  =  0000  0000 0000  0000 0000  0000 0101
     ⇒ 9           ;  9  =  0000  0000 0000  0000 0000  0000 1001

(logxor 12 5 7)    ; 12  =  0000  0000 0000  0000 0000  0000 1100
                   ;  5  =  0000  0000 0000  0000 0000  0000 0101
                   ;  7  =  0000  0000 0000  0000 0000  0000 0111
     ⇒ 14          ; 14  =  0000  0000 0000  0000 0000  0000 1110
```

lognot *integer* Function

この関数は、引数の論理的補数 (logical complement) を返します。結果の第 n ビットは、 *integer* の第 n ビット目が 0 である場合にのみセットされます。逆もまた同様です。

```
(lognot 5)
     ⇒ -6
;;   5  =  0000  0000 0000  0000 0000  0000 0101
;; は
;; -6  =  1111  1111 1111  1111 1111  1111 1010
;; になる
```

3.9 標準的な数学関数

以下の数学関数は、浮動小数点をサポートしている場合に有効です。浮動小数点数だけでなく、整数を引数に与えることができます。

sin *arg* Function

cos *arg* Function

tan *arg* Function

これらは通常の三角関数で、引数の単位はラジアンです。

asin *arg* Function

(`asin` *arg*) の値は、正弦が *arg* となるような $-\pi/2$ から $\pi/2$ までの (境界を含む) 数です。しかし、もし *arg* が $[-1, 1]$ の範囲外なら、結果は NaN です。

acos *arg* Function

(`acos` *arg*) の値は、余弦が *arg* となるような 0 から π までの (境界を含む) 数です。しかし、もし *arg* が $[-1, 1]$ の範囲外なら、結果は NaN です。

atan *arg* Function

(`atan` *arg*) の値は、正接が *arg* となるような $-\pi/2$ から $\pi/2$ までの (境界を含まない) 数です。

exp *arg* Function

これは指数関数です。 e の *arg* 乗を返します。 e は数学の基本定数で自然対数の底ともいいます。

log *arg* &optional *base* Function

この関数は、 *base* を底にした *arg* の対数を返します。 *base* を指定しなければ底として e を用います。 *arg* が負なら、結果は NaN です。

log10 *arg* Function

この関数は、 10 を底にした対数を返します。 *arg* が負なら、結果は NaN です。少なくとも近似的には (`log10` *x*) $\equiv$ (`log` *x* `10`) です。

expt *x y* Function

この関数は、 *x* の *y* 乗を返します。もし、両方の引数が整数で、 *y* が正ならば、結果は整数です。この場合、表現可能な整数値の範囲に切り捨てられます。

sqrt *arg* Function

これは *arg* の平方根を返します。 *arg* が負なら結果は NaN です。

3.10 乱数

計算機プログラムは決定性があり、真の乱数を発生できません。ほとんどの目的のためには、『疑似乱数 (pseudo-random number)』で十分です。疑似乱数列は、決定性のある仕方で生成されます。そうした乱数列は真の乱数ではありませんが、乱数列に類似した特性をもっています。たとえば、すべてのとり得る値は、疑似乱数列中で等しい頻度で発生します。

Emacs は、「種」(seed) となる数から疑似乱数を生成します。与えられた種から始めて、`random` 関数は、常に同じ数列を生成します。 Emacs は常に同一の種値から始めるので、`random` の返す数の列は、実は Emacs を起動するたびに同じものになります ! たとえば、あるオペレーティング・システムでは、 Emacs を起動した後 (`random`) を呼び出すと、常に -1457731 を返し、もう一度呼び出すと -7692030 を返します。この復元性はデバッグを行なう際、役立ちます。

真に予見不能な乱数群を必要とする場合は、 (`random t`) を実行します。これは、現在の時刻と Emacs のプロセス ID 番号とをもとに新しい種の選択を行ないます。

random &optional *limit* Function

この関数は、整数の疑似乱数 (pseudo-random integer) を返します。繰り返して呼ぶと、疑似乱数の列を返します。

もし *limit* が正整数なら、値は非負かつ *limit* 未満になります。

limit が `t` なら、現在の時刻と Emacs のプロセス ID 番号とをもとに、新しい種の選択を行ないます。

あるマシンでは、 `random` の結果として Lisp で表現可能な、どの整数が返されることもあります。あるいは、マシンによっては、値は、ある最大値より大きくはなく、またある (負の) 最小値より小さくなることもありません。

4 文字列と文字

Emacs Lisp での文字列 (string) は、順序づけられた文字の並び (ordered sequence of character) を含む配列です。文字列は、シンボル、バッファ、ファイルなどの名前として、ユーザにメッセージを送るため、バッファ間でコピーするテキストを保持するため、その他多くの目的のために用いられます。文字列は非常に重要であるため、 Emacs Lisp は、非常に多くの文字列を操作する関数を持っています。 Emacs Lisp プログラムでは、一つ一つの文字よりも文字列の方が多く用いられます。

キーボード文字イベントの文字列のために特別に考慮すべきことについては、 18.5.14 節「文字列にキーボード・イベントを入れる」 p.278 を参照してください。

4.1 文字列・文字の基本的な性質

Emacs Lisp での文字列 (string) は、順序づけられた文字の並び (ordered sequence of character) からなる配列です。また、 Emacs Lisp では、文字は整数で表現されますが、整数が文字を意味するかどうかは、それがどのように使われるかによってのみ決まります。つまり文字列は実は整数の集まりなのです。

どんな配列でもそうなのですが、文字列の長さも文字列の内容にかかわらず固定され、変えることはできません。 Lisp での文字列は、特定の文字コードで終端されていません (これに対して、 C では ASCII code の 0 で終端されます)。つまり、 null 文字 (ASCII コードの 0) を含む任意の文字が、文字列の有効な要素となります。

文字列は配列の一種であるため、一般的な配列用の関数で操作することが可能です (第 6 章「シーケンス、配列、ベクタ」 p.85 を参照してください)。たとえば、文字列内の個々の文字を、 `aref` や `aset` を用いて取り出したり変更したりできます (6.3 節「配列に関する関数」 p.88 参照)。

文字列中のそれぞれの文字は、 1 バイトのデータとして格納されます。したがって、 0 から 255 の範囲に入らない数は、文字列に格納されるときには切り捨てられます。このため、同じ長さのベクタに比べて、文字列は、より少ないメモリでかまいません。

キー・シーケンスは、文字列として表現されるときがあります。このとき、文字列の 128 から 255 までの範囲の要素は、 128 から 255 の範囲のキーボード・イベントではなく、 (非常に大きな整数である) メタ文字を表わします。

文字列は、 hyper、 super、もしくは alt、の修飾子 (modifier) を持った文字を保持することはできません。また ASCII 制御文字を保持することはできますが、それ以外の制御文字を保持することはできません。文字列は、 ASCII 制御文字の大文字小文字 (case) を区別しません。[1]キーボード入力文字のメタや他の修飾子の表現に関しての詳細は、 2.3.3 節「文字型」 p.21 を参照してください。

文字列は、正規表現を成立させるのにも用いられます。文字列に対して、正規表現を用いた照合を行うことも可能です (30.3 節「正規表現の検索」 p.530 参照)。正規表現照合に基づいた文

---

[1] 訳注: X のイベントでは、 `Ctrl-X` と `Ctrl-x` が区別されるが、 ASCII では、両者が区別されないということ。

字列の分解および変更には、関数 `match-string` (30.6.1 節「単純な一致データのアクセス」p.534 参照) および `replace-match` (30.6.2 節「一致したテキストの置換」p.536 参照) が用いられます。

バッファと同様に、文字列は、個々の文字を持つと同時に、文字列内の文字のためのテキスト属性を持つことができます。29.18 節「テキスト属性」p.509 を参照してください。文字列から、バッファもしくは他の文字列へテキストをコピーする全ての Lisp プリミティブは、コピーされる文字のテキスト属性もいっしょにコピーします。

文字列を表示する関数もしくはバッファにコピーする関数に関しては、第 29 章「テキスト」p.479 を参照してください。文字と文字列の構文 (syntax) に関しては、 2.3.3 節「文字型」p.21 および 2.3.8 節「文字列型」p.28 を参照してください。

4.2 文字列の述語

一般的なシーケンスと配列の述語に関しての詳細は、第 6 章「シーケンス、配列、ベクタ」p.85 と 6.2 節「配列」p.87 を参照してください。

stringp *object* Function

この関数は、 *object* が文字列であったら `t` を返し、そうでない場合は `nil` を返します。

char-or-string-p *object* Function

この関数は、 *object* が文字列もしくは文字 (すなわち整数) であったら `t` を返し、そうでない場合には `nil` を返します。

4.3 文字列の生成

以下の関数は、文字列を最初から生成したり、いくつかの文字列を結合して生成したり、もしくは文字列の一部を取り出したりすることによって生成したりします。

make-string *count character* Function

この関数は、 *count* 個の文字 *character* からなる文字列を返します。 *count* が負のときはエラーとなります。

```
(make-string 5 ?x)
     ⇒ "xxxxx"
(make-string 0 ?x)
     ⇒ ""
```

この関数と比較されるものとして、 `char-to-string` (4.5 節「文字と文字列間の変換」p.60 参照) と `make-vector` (6.4 節「ベクタ」p.89 参照) と `make-list` とがあります (5.5 節「コンス・セルとリストの生成」p.71 参照)。

substring *string start* &optional *end* Function

この関数は、 *string* 中の *start* 番目の文字 (この文字を含む) から、 *end* 番目の文字 (この文字は含まない) までの文字列を生成し、返します。ここで最初の文字は 0 番目の文字とします。

```
(substring "abcdefg" 0 3)
     ⇒ "abc"
```

ここで、‘a’ のインデックスは 0、‘b’ のインデックスは 1、そして ‘c’ のインデックスは 2 となります。したがって、文字列 "abcdefg"から 3 文字の ‘abc’ がコピーされます。ここでインデックス 3 は、その文字までの部分文字列をコピーすることを表します。インデックス 3 で示される文字は、実際には文字列中の 4 番目の文字です。

負の数字は、文字列の終りから数えます。 -1 は文字列の最後の文字を示します。たとえば、

```
(substring "abcdefg" -3 -1)
     ⇒ "ef"
```

この例で ‘e’ のインデックスは -3、‘f’ のインデックスは -2、そして ‘g’ のインデックスは -1 となります。したがって ‘e’ と ‘f’ は含まれ、‘g’ は含まれません。

nil がインデックスとして用いられた場合は、文字列の長さを表わします。したがって、次のようになります。

```
(substring "abcdefg" -3 nil)
     ⇒ "efg"
```

引数 *end* を省略することは、 nil を指定することと等価です。よって (substring *string* 0) は *string* の全体のコピーを返します。

```
(substring "abcdefg" 0)
     ⇒ "abcdefg"
```

しかしこのような用途には、 copy-sequence を用いることを推奨します (6.1 節「シーケンス」p.85 参照)。

string からコピーされた文字がテキスト属性を持っている場合は、その属性も新たな文字列へコピーされます。 29.18 節「テキスト属性」p.509 を参照してください。

start もしくは *end* が整数でも nil でもない場合は、 wrong-type-argument エラーとなります。 *start* が *end* より後ろの文字を指している場合、もしくはいずれかの値が *string* の範囲外である場合は、 args-out-of-range エラーとなります。

この関数と、カレント・バッファのテキストの一部分の文字列を返す関数 buffer-substring (29.2 節「バッファ内容の調査」p.480 参照) との違いに注意してください。文字列の先頭は 0 とインデックスされるのに対し、バッファの先頭は 1 とインデックスされます。

concat &rest *sequences* Function

この関数は、渡された引数に含まれる文字からなる新しい文字列を返します (もしテキスト属性を持っている場合は、テキスト属性もコピーされます)。引数は、文字列、数のリスト、数のベクタのいずれかであり、引数自体は変更されません。引数が与えられなかった場合は、 concat は空文字列を返します。

```
(concat "abc" "-def")
     ⇒ "abc-def"
(concat "abc" (list 120 (+ 256 121)) [122])
     ⇒ "abcxyz"
```

```
;; nil は空の列です.
(concat "abc" nil "-def")
     ⇒ "abc-def"
(concat "The " "quick brown " "fox.")
     ⇒ "The quick brown fox."
(concat)
     ⇒ ""
```

この 2 番目の例は、文字が、 256 で割った余りとして文字列に格納される様子を示しています。つまり、文字列中のそれぞれの文字は、 1 バイトで格納されます。

関数 `concat` は、既存のいかなる文字列とも `eq` でない新しい文字列を常に生成します。

ある引数が、(整数の並びではなく) 一つの整数である場合は、その整数を 10 進数で表現した数字の文字列に変換されます。この機能を使用しないでください。我々はこれを削除する予定です。すでにこの機能を使っていたら、今すぐにあなたのプログラムを変更してください！ `format` (4.6 節「文字列の書式変換」p.61 参照) もしくは `number-to-string` (4.5 節「文字と文字列間の変換」p.60 参照) を用いるのが、整数を数字の文字列に変換する正しい方法です。

```
(concat 137)
     ⇒ "137"
(concat 54 321)
     ⇒ "54321"
```

他の連結関数に関しては、 11.6 節「マッピング関数」p.155 の `mapconcat`、 6.4 節「ベクタ」p.89 の `vconcat`、さらに 5.5 節「コンス・セルとリストの生成」p.71 の `append` の解説を参照してください。

4.4 文字・文字列の比較

char-equal *character1 character2* Function

この関数は、引数が同じ文字を表わす場合は `t` を返し、そうでない場合 `nil` を返します。`case-fold-search` が `nil` でない場合は、大文字と小文字を区別しません。[2]

```
(char-equal ?x ?x)
     ⇒ t
(char-to-string (+ 256 ?x))
     ⇒ "x"
(char-equal ?x  (+ 256 ?x))
     ⇒ t
```

string= *string1 string2* Function

この関数は、二つの文字列の文字が完全に一致した時、 `t` を返します。大文字と小文字は区別されます。

[2] 訳注: 最後の 2 例は Mule では結果が異なる。

```
(string= "abc" "abc")
     ⇒ t
(string= "abc" "ABC")
     ⇒ nil
(string= "ab" "ABC")
     ⇒ nil
```

関数 `string=` は、二つの文字列のテキスト属性を無視します。テキスト属性も含めて文字列を比較したい場合は、 `equal` を使用してください (2.6 節「等価述語」 p.38 参照)。

string-equal *string1 string2* Function

`string-equal` は `string=` の別の名前です。

string< *string1 string2* Function

この関数は、二つの文字列を一文字ずつ比較します。まず、それぞれの文字列の先頭から順に調べていき、最初に、対応する文字が一致しない最初の文字の組を見つけようとします。この一致しない文字の組で、小さい方の文字が *string1* のものであった場合は *string1* の方が小さいこととなり、この関数は `t` を返します。小さい方の文字が *strings2* のものであった場合は *string1* の方が大きいこととなり、この関数は `nil` を返します。二つの文字列が完全に一致した場合は `nil` となります。

それぞれの文字の組は ASCII コードで比較されます。 ASCII 文字集合では、ある文字の小文字はその大文字よりも大きい値をもつことと、数字や多くの句読点文字は、大文字よりも小さな値をもつことに気をつけてください。

```
(string< "abc" "abd")
     ⇒ t
(string< "abd" "abc")
     ⇒ nil
(string< "123" "abc")
     ⇒ t
```

異なる長さの文字列を比較するときに、 *string1* の長さまで一致した場合は `t` となります。 *string2* の長さまで一致したときは `nil` となります。文字を持たない文字列 (長さ 0 の文字列) は、いかなる文字列よりも小さいものとされます。

```
(string< "" "abc")
     ⇒ t
(string< "ab" "abc")
     ⇒ t
(string< "abc" "")
     ⇒ nil
(string< "abc" "ab")
     ⇒ nil
(string< "" "")
     ⇒ nil
```

string-lessp *string1 string2* Function

`string-lessp` は `string<`の別の名前です。

バッファ中のテキストを比較する方法に関しては、 29.3 節「テキストの比較」 p.482 の `compare-buffer-substrings` を参照してください。正規表現を文字列に対して照合する関数 `string-match` は、文字列比較の一手段としても用いることができます。 30.3 節「正規表現の検索」 p.530 参照。

4.5 文字と文字列間の変換

この節では、文字、文字列、および整数相互の変換を行う関数について説明します。 `format` と `prin1-to-string` (16.5 節「出力関数」 p.233 参照) は、 Lisp オブジェクトを文字列に変換します。 `read-from-string` (16.3 節「入力関数」 p.230 参照) は、 Lisp オブジェクトの文字列表現を Lisp オブジェクトに "変換" します。

テキスト文字と一般の入力イベントの解説 (textual description) を生成する関数 (`single-key-description` および `text-char-description`) に関しては、第 21 章「ヘルプ」 p.339 を参照してください。これらの関数はおもにヘルプ・メッセージの生成に用いられます。

char-to-string *character* Function

この関数は、 1 文字の文字列を新たに生成し、返します。 *character* の値を 256 で割った余りの値が用いられます。

この関数は、整数の引数を一つ取る関数 `make-string` に似ています (4.3 節「文字列の生成」 p.56 を参照してください)。この変換は、 '`%c`' 書式指定子を用いた `format` によって行うこともできます (4.6 節「文字列の書式変換」 p.61 を参照してください)。[3]

```
(char-to-string ?x)
     ⇒ "x"
(char-to-string (+ 256 ?x))
     ⇒ "x"
(make-string 1 ?x)
     ⇒ "x"
```

string-to-char *string* Function

この関数は、 *string* の最初の文字を返します。文字列が空文字列の場合は 0 を返します。また *string* が null 文字 (ASCII コードの 0) で始まっている場合も 0 を返します。

```
(string-to-char "ABC")
     ⇒ 65
(string-to-char "xyz")
     ⇒ 120
(string-to-char "")
     ⇒ 0
(string-to-char "\000")
     ⇒ 0
```

この関数は、将来、残す必要がないと判断された時点で削除されるかもしれません。

[3] 訳注: 最後の二例は Mule では結果が異なる。

number-to-string *number* Function

この関数は、整数もしくは浮動小数点数である *number* の印字表現の文字列を返します。引数が負であった場合は、返される文字列は負符号 '-' で始まります。

```
(number-to-string 256)
     ⇒ "256"
(number-to-string -23)
     ⇒ "-23"
(number-to-string -23.5)
     ⇒ "-23.5"
```

`int-to-string` は、この関数に対するほとんど陳腐化した別名です。[4]

4.6 節「文字列の書式変換」 p.61 の `format` も参照してください。

string-to-number *string* Function

この関数は、 *string* 中の文字を基数 10 の数字として読込み、その数値を返します。*string* の先頭のスペースとタブを読みとばした後、 *string* から数字として解釈できるかぎり読込みます (いくつかのシステムではスペースとタブ以外の空白文字 (whitespace) も読みとばします)。空白文字を除いた最初の文字が数字もしくは負符号 '-' でなかった場合は、この関数は 0 を返します。

```
(string-to-number "256")
     ⇒ 256
(string-to-number "25 is a perfect square.")
     ⇒ 25
(string-to-number "X256")
     ⇒ 0
(string-to-number "-4.5")
     ⇒ -4.5
```

`string-to-int` は、この関数に対してつけられている陳腐化した別名です。

4.6 文字列の書式変換

『書式変換 (formatting)』とは、与えられた文字列のいくつかの部分を、指示された処理の結果の値で、置き換え、結果の文字列を生成することを意味します。この文字列は、それ自身以外の引数の値がどこに表示されるかだけでなく、どのように印字されるかも指定します。この文字列を『書式変換文字列 (format string)』と呼びます。

書式変換は、メッセージの表示に便利です。事実 `message` と `error` という関数は、ここで説明するものと同じ書式変換機能を持ちます。書式変換の結果の使い方のみが `format` と異なります。

[4] 訳注: Emacs バージョン 18 までは浮動小数点数は扱えなかったため `int-to-string` という名前であった。

format *string* &rest *objects* Function

この関数は、 *string* をコピーし、その中の書式指定子 (format specification) を、対応する *objects* の演算結果を文字列に変換したものと置き換えた、新たな文字列を返します。

引数 *objects* は、書式変換に使われる値です。

書式指定子は、 '%' で始まる文字の並びです。 '%d' が *string* の中にある場合、 `format` 関数は、それを書式変換すべき値の一つ (引数 *objects* の一つ) の印字表現と置き換えます。たとえば、

```
(format "The value of fill-column is %d." fill-column)
     ⇒ "The value of fill-column is 72."
```

string が複数の書式指定子を含んでいる場合は、それぞれの書式指定は *objects* のそれぞれの値に対応づけられます。たとえば、 *string* の中の最初の書式指定子は、最初の値を用い、二つ目の書式指定子は二つ目の値を用います。余分な書式指定子 (対応する値がない書式指定子) があった場合の動作は不定です。余分な値は無視されます。

いくつかの書式指定子は、特定の型の値と対応づけられなければなりません。しかし、実際に与えられた値が期待された型ではなかったとしてもエラーとはなりません。そのかわり出力が意味のないものとなります。

以下は有効な書式指定子の表です。

'%s' 指定子を quote しないオブジェクトの印字表現と置き換えます。文字列は、 '"' 文字で括られず、特殊文字には '\' 文字を付加しません。

対応するオブジェクトがない場合は空文字列となります。

'%S' 指定子を quote されたオブジェクトの印字表現と置き換えます。文字列は、 '"' 文字で括られ、特殊文字の前には '\' 文字が付加されます。

対応するオブジェクトがない場合は、空文字列となります。

'%o' 指定子を整数の 8 進数表現と置き換えます。

'%d' 指定子を整数の 10 進数表現と置き換えます。

'%x' 指定子を整数の 16 進数表現と置き換えます。

'%c' 指定子を与えられた値の文字と置き換えます。

'%e' 指定子を浮動小数点数の指数表記と置き換えます。

'%f' 指定子を浮動小数点数の小数点表記と置き換えます。

'%g' 指定子を浮動小数点数の指数表記と小数点表記のうち短いものと置き換えます。

'%%' 1 文字の '%' となります。この書式指定は値を必要としない特殊なものです。たとえば `(format "%% %d" 30)` は `"% 30"`を返します。

これら以外の変換文字は 'Invalid format operation' エラーとなります。

以下にいくつかの例をあげます。

```
(format "The name of this buffer is %s." (buffer-name))
     ⇒ "The name of this buffer is strings.texi."

(format "The buffer object prints as %s." (current-buffer))
     ⇒ "The buffer object prints as strings.texi."

(format "The octal value of %d is %o,
         and the hex value is %x." 18 18 18)
     ⇒ "The octal value of 18 is 22,
         and the hex value is 12."
```

全ての書式指定子は、'%' と指定子の間に数値前置詞をいれることができます。数値前置詞は、オブジェクトの最小の幅を指定します。オブジェクトの印字表現がこれよりも少ない文字の場合は、空白文字もしくは文字 0 が付加されます。前置詞の値が正のときは (もしくは 0 で始まるときは) 右に、負のときは右に付加されます。通常は空白文字が付加されますが、数値前置詞が 0 で始まる場合は 0 が付加されます。

```
(format "%06d is padded on the left with zeros" 123)
     ⇒ "000123 is padded on the left with zeros"

(format "%-6d is padded on the right" 123)
     ⇒ "123    is padded on the right"
```

format は、幅を指定した場合でもオブジェクトの印字表現を切り詰めることはありません。したがって数値前置詞によって、情報を失うことなく、項目間の最小の間隔を指定することができます。

以下の三つの例で、'%7s' は最小幅 7 を指定します。最初の例では、'%7s' の位置に入れられる文字列は 3 文字なので四つの空白文字 (blank spaces) が加えられます。 2 番目の例では、文字列 "specification"の幅は 13 文字ですが切り詰められません。 3 番目の例では右に空白文字が加えられています。

```
(format "The word `%7s' actually has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word `    foo' actually has 3 letters in it."
(format "The word `%7s' actually has %d letters in it."
        "specification" (length "specification"))
     ⇒ "The word `specification' actually has 13 letters in it."
(format "The word `%-7s' actually has %d letters in it."
        "foo" (length "foo"))
     ⇒ "The word `foo    ' actually has 3 letters in it."
```

4.7 大文字・小文字変換

大文字・小文字変換関数は、単一の文字もしくは文字列の段 (case) の変換を行ないます。これらの関数は、アルファベット文字 ('A' から 'Z' と 'a' から 'z') のみを変換し、他の文字は変更しません。また引数として与えられた文字列自身は変更しません。

以下の例では文字 'X' (ASCII code 88) と 'x' (ASCII code 120) を用います。

downcase *string-or-char* Function

この関数は文字もしくは文字列を下段 (lower case) に変えます。

引数 downcase が文字列の場合は、引数の文字列中の大文字が小文字に変換された新たな文字列が返されます。

引数 downcase が文字の場合は、 downcase は対応する小文字を返します。この値は整数です。もとの文字が小文字もしくはアルファベット文字ではなかった場合、もとの文字が返されます。

```
(downcase "The cat in the hat")
     ⇒ "the cat in the hat"

(downcase ?X)
     ⇒ 120
```

upcase *string-or-char* Function

この関数は文字もしくは文字列を上段 (upper case) に変えます。

引数 upcase が文字列の場合は、引数の文字列中の小文字が大文字に変換された新たな文字列が返されます。

引数 upcase が文字の場合は、 upcase は対応する大文字を返します。この値は整数です。もとの文字が大文字もしくはアルファベット文字ではなかった場合、もとの文字が返されます。

```
(upcase "The cat in the hat")
     ⇒ "THE CAT IN THE HAT"

(upcase ?x)
     ⇒ 88
```

capitalize *string-or-char* Function

この関数は文字列や文字のキャピタル化 (capitalize) を行ないます。

引数 string-or-char が文字列の場合、引数の *string-or-char* のそれぞれの単語 (word) をキャピタル化した新たな文字列が返されます。つまり各単語の最初の文字を大文字に変換し、残りの文字を小文字に変換します。

単語 (word) の定義は、現在の構文テーブル (syntax table) で単語を構成する文字に分類される文字が連続したものです (31.2.1 節「構文クラスのテーブル」 p.542 を参照してください)。

capitalize の引数が文字の場合は、 capitalize は、 upcase と同じ値を返します。

```
(capitalize "The cat in the hat")
     ⇒ "The Cat In The Hat"

(capitalize "THE 77TH-HATTED CAT")
     ⇒ "The 77th-Hatted Cat"
```

```
(capitalize ?x)
     ⇒ 88
```

4.8 大文字小文字変換のカスタマイズ

特別な『段変換テーブル (case table)』を設定することで上段と下段の対応をカスタマイズすることが可能です。この変換テーブルは大文字と小文字の対応を規定します。文字列と文字の大文字小文字変換関数 (前節参照) とバッファ内のテキストに対する変換関数 (29.17 節「大文字 / 小文字の変換」 p.508 参照) の両方に効果があります。標準の ASCII 文字以外の文字を使う言語を用いているのであれば、この変換テーブルの設定が必要になります。

段変換テーブルは次の形式を取ります。

(*downcase upcase canonicalize equivalences*)

ここで、それぞれの項は、 `nil` もしくは長さ 256 の文字列です。項 *downcase* は、それぞれの文字をその下段文字に対応させる仕方を示します。 *upcase* は、それぞれの文字をその上段字に対応させる仕方を示します。上下段が 1 対 1 に対応する場合、 *upcase* を `nil` としてください。そうすれば、 Emacs が *downcase* から上段を類推します。

いくつかの言語では、上下段は 1 対 1 対応しません。一つの大文字に対して二つの異なる小文字がある場合もあります。このような場合は、双方向の対応表を指定しなければなりません。

項 *canonicalize* は、それぞれの文字を、標準等価 (canonical equivalent) な文字と対応づけます。大文字・小文字変換で関連づけられる 2 文字は同じ標準等価な文字を持ちます。

項 *equivalences* は、 (同じ標準等価な文字をもつ文字の) それぞれの等価なクラスを循環させて並べ変えた対応表です (通常の ASCII では、これは ‘`a`’ を ‘`A`’ に対応づけ、 ‘`A`’ を ‘`a`’ に対応づける、というようにそれぞれ等価な文字の組を同様に対応づけます)。[5]

canonicalize を `nil` として段変換テーブルを生成すると、 Emacs が *upcase* と *downcase* からそれを埋めます。また *equivalences* を `nil` とすれば、 Emacs は *canonicalize* から埋めます。実際に使われている変換テーブルでは、これらの項は `nil` ではありません。 *canonicalize* を指定せずに *equivalences* を指定しようとしないでください。

それぞれのバッファは、段変換テーブルを一つ持ちます。また、 Emacs は、バッファを生成するときにそれぞれのバッファにコピーされる『標準段変換テーブル (standard case table)』を持っています。標準段変換テーブルを変更しても、すでに存在しているバッファには影響を与えません。

以下に段変換テーブルを操作する関数を示します。

case-table-p *object* Function

この述語は、 *object* が有効な段変換テーブルであるとき non-`nil` を返します。

[5] 訳注: “循環させる” とは、たとえばある言語で文字 A, B, C が文字 X という文字と標準等価であるとき、 A を B に、 B を C に、 C を A にそれぞれ対応づけること。

set-standard-case-table *table* Function

この関数は、 *table* を標準段変換テーブルにします。この新しいテーブルは、それ以後生成されるバッファに適用されます。

standard-case-table Function

この関数は、標準段変換テーブルを返します。

current-case-table Function

この関数は、現在のバッファの段変換テーブルを返します。

set-case-table *table* Function

この関数は、現在のバッファの段変換テーブルを *table* とします。

以下の三つの関数は、非 ASCII 文字集合を定義するプログラムで便利なサブルーチンです。引数として文字列 *downcase-table* を与え、それを変更します。この文字列は、変換テーブルの *downcase* 部分に用いられる文字列となります。これらの関数は、標準構文テーブルも変更します。第 31 章「構文テーブル」p.541 を参照してください。

set-case-syntax-pair *uc lc downcase-table* Function

この関数は、対応する一組の大文字と小文字を指定します。

set-case-syntax-delims *l r downcase-table* Function

この関数は、文字 *l* から *r* までを、段変換を行なわない文字とします。

set-case-syntax *char syntax downcase-table* Function

この関数は、 *char* を構文 *syntax* では段変換を行なわない文字とします。

describe-buffer-case-table Command

このコマンドは、カレント・バッファの段変換テーブルの設定値を表示します。

‘iso-syntax’ ライブラリをロードすることで、 8-bit ISO Latin 1 文字集合のための標準構文テーブルの設定と段変換テーブルの定義を行うことができます。

5 リスト

『リスト (list)』は、零個もしくは数個の要素 (どんな Lisp オブジェクトでも構いません) の並び (sequence) です。リストとベクタとの重要な違いは、複数のリストの間でその一部を共有することが可能なことです。さらにリスト全体をコピーすることなしに要素を加えたり、削除したりすることも可能です。

5.1 リストとコンス・セル

Lisp では、リストはプリミティブなデータ型ではありません。リストは『コンス・セル (cons cell)』から構成されます。個々のコンス・セルは順序づけられた対のデータ・オブジェクトです。コンス・セルは、二つの Lisp オブジェクトを指します。一つは CAR と呼ばれ、もう一つは CDR と呼ばれます。これらの名前は伝統的なものです (2.3.6 節「コンス・セル型とリスト型」p.25 参照)。 CDR は “could-er” と発音します。

リストは、その各要素が一つのコンス・セルである、一継りのコンス・セルの集まりです。慣習により、そのそれぞれのコンス・セルの CAR はリストの要素を持ち、 CDR はコンス・セルをつなぐために用いられます。つまり、それぞれのコンス・セルの CDR は、次のコンス・セルを指します。最後のコンス・セルの CDR は `nil` です。この CAR と CDR との非対称性はただ単に慣習によるものです。コンス・セル自体は、 CAR も CDR も同じ性質を持っています。

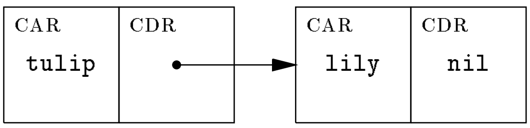
ほとんどのコンス・セルがリストの一部分として用いられることから、『リスト構造 (list structure)』という言葉は、コンス・セルから構成される全ての構造を意味します。

シンボル `nil` は、シンボルとしてだけではなく、要素を持たないリストとしても扱われます。便宜上シンボル `nil` は、その CDR に (またその CAR にも) `nil` をもつものとして扱われます。

空ではないリスト l の CDR は、 l の最初の要素以外の全ての要素を含むリストです。

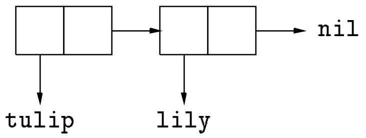
5.2 箱を用いたリストの表現

コンス・セルは、一組の箱として表現することができます。最初の箱が CAR を表わし、二つめの箱が CDR を表わします。二つのコンス・セルからなる、(`tulip lily`) という二つの要素からなるリストは、次のように表されます。

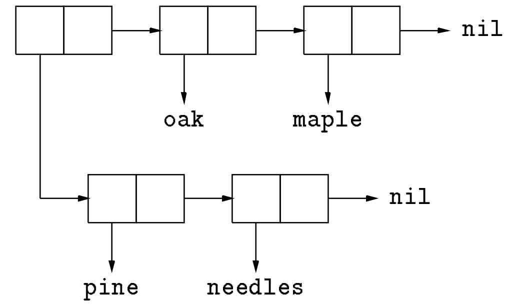


それぞれの箱の組は、コンス・セルを表わします。それぞれの箱がリスト・オブジェクトを “参照する (refer to)” とか、 “指し示す (point to)” とか、あるいは、 “含む (contain)” などと表現します (これらの言葉は同義です)。先頭のコンス・セルの CAR である最初の箱には、`tulip` というシンボルが入っています。その CDR から 2 番目のセルへの矢印は、最初のコンス・セルの CDR が 2 番目のコンス・セルを指し示していることを表します。

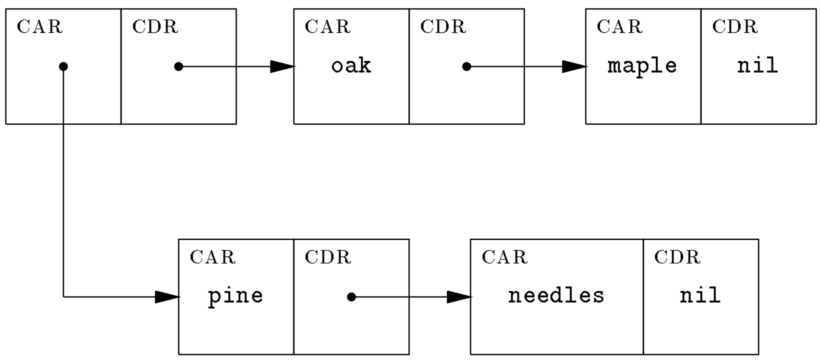
このリストは、次のような別の箱表記で表わすこともできます。



次はより複雑な例です。((pine needles) oak maple) という三つの要素からなるリストを表わし、その最初の要素は、二つの要素からなるリストです。



このリストを最初の箱表記で表すと、次のようになります。



コンス・セルやリストに対する、読込みと印字の構文、および、リストの “箱と矢印” を用いた表記に関するより詳しい解説は、 2.3.6 節「コンス・セル型とリスト型」p.25 を参照してください。

5.3 リストに関する述語

以下の述語 (predicate) は、リスト・オブジェクトがアトムであるか、コンス・セルであるか、リストであるか、もしくは nil であるかを調べるために用いられます (これらの述語の多くは、他のものを用いて定義できますが、非常によく使われるので、全部を用意しておくことに意味があるのです)。

consp *object* Function

この関数は *object* がコンス・セルであれば `t` を返し、そうでなければ `nil` を返します。`nil` はリストですが、コンス・セルではありません。

atom *object* Function

この関数は、 *object* がアトムであれば `t` を返し、そうでなければ `nil` を返します。コンス・セル以外の全てのオブジェクトはアトムです。 `nil` はアトムであり、かつリストでもあります。アトムであり、かつリストであるのは、唯一 `nil` だけです。

```
(atom object) ≡ (not (consp object))
```

listp *object* Function

この関数は、 *object* がコンス・セルもしくは `nil` であれば `t` を返し、そうでなければ `nil` を返します。

```
(listp '(1))
     ⇒ t
(listp '())
     ⇒ t
```

nlistp *object* Function

この関数は、 `listp` の反対の働きをします。つまり *object* がリストでなければ `t` を返し、そうでなければ `nil` を返します。

```
(listp object) ≡ (not (nlistp object))
```

null *object* Function

この関数は、 *object* が `nil` であれば `t` を返し、そうでなければ `nil` を返します。この関数は `not` と同じものですが、わかりやすいように、 *object* がリストである場合 `null` を、真か偽かを表わす場合には `not` を用います (9.3 節「条件を結合するための構文要素」p.114 の `not` を参照)。

```
(null '(1))
     ⇒ nil
(null '())
     ⇒ t
```

5.4 リストの要素の参照

car *cons-cell* Function

この関数は、コンス・セル *cons-cell* の最初のポインタによって指し示される値を返します。これを簡単に、この関数は *cons-cell* の CAR を返す、とも表現します。

特別な場合として、 *cons-cell* が `nil` の場合、 `car` は `nil` を返すと定義されます。したがって、いかなるリストも `car` の有効な引数となります。引数がコンス・セルでも `nil` でもない場合、エラーが通知されます。

```
(car '(a b c))
     ⇒ a
(car '())
     ⇒ nil
```

cdr *cons-cell* Function

この関数は、コンス・セル *cons-cell* の二つめの箱のポインタによって指し示される値を返します。この関数は *cons-cell* の CDR を返す、とも表現されます。

特別な場合として、 *cons-cell* が `nil` の場合、 `cdr` は `nil` を返すと定義されます。したがって、いかなるリストも `cdr` の有効な引数となります。引数がコンス・セルでも `nil` でもない場合、エラーが通知されます。

```
(cdr '(a b c))
     ⇒ (b c)
(cdr '())
     ⇒ nil
```

car-safe *object* Function

この関数は、データ型がコンス・セル以外である場合でもエラーとならず、 CAR を得ることができます。 *object* がコンス・セルの場合は *object* の CAR を返し、そうでなければ `nil` を返します。これは、 *object* がリストでない場合にエラーを通知する `car` と対照的です。

```
(car-safe object)
≡
(let ((x object))
  (if (consp x)
      (car x)
    nil))
```

cdr-safe *object* Function

この関数は、データ型がコンス・セル以外である場合でもエラーとならず、 CDR を得ることができます。 *object* のコンス・セルの場合は *object* の CDR を返し、そうでなければ `nil` を返します。これは、 *object* がリストでない場合にエラーを通知する `cdr` と対照的です。

```
(cdr-safe object)
≡
(let ((x object))
  (if (consp x)
      (cdr x)
    nil))
```

nth *n list* Function

この関数は、 *list* の n 番目の要素を返します。最初の要素は 0 番目とされます。つまり *list* の CAR は 0 番目の要素です。もし、 *list* の長さが n 以下の場合は `nth` は `nil` を返します。

n が負である場合、 `nth` は *list* の最初の要素を返します。

```
(nth 2 '(1 2 3 4))
     ⇒ 3
(nth 10 '(1 2 3 4))
     ⇒ nil
(nth -3 '(1 2 3 4))
     ⇒ 1

(nth n x) ≡ (car (nthcdr n x))
```

nthcdr *n list* Function

この関数は、 *list* の *n* 番目の CDR を返します。別の表現をすると、 *list* の最初の *n* 個のリンクを取り除き、残ったものを返します。

n が 0 以下の場合、 `nthcdr` は *list* 全体を返します。 *list* の長さが *n* 以下の場合は、 `nthcdr` は `nil` を返します。

```
(nthcdr 1 '(1 2 3 4))
     ⇒ (2 3 4)
(nthcdr 10 '(1 2 3 4))
     ⇒ nil
(nthcdr -3 '(1 2 3 4))
     ⇒ (1 2 3 4)
```

5.5 コンス・セルとリストの生成

リストは Lisp の中核となる概念であるため、多くの関数がリストを生成します。 `cons` が基本的なリストを生成する関数です。しかし、興味深いことに、 Emacs のためのソース・コードでは `cons` よりも `list` の方が多く使われています。

cons *object1 object2* Function

この関数は、新しいリスト構造を作るために用いられる基本的な関数です。新しいコンス・セルを作り、 *object1* を CAR に、さらに *object2* を CDR にします。そしてこの新しいコンス・セルを返します。引数 *object1* と *object2* はどんなリスト・オブジェクトでも構いません。しかし、ほとんどの場合 *object2* はリストです。

```
(cons 1 '(2))
     ⇒ (1 2)
(cons 1 '())
     ⇒ (1)
(cons 1 2)
     ⇒ (1 . 2)
```

`cons` は、しばしばリストの先頭に新しい要素を加えるために用いられます。これを『要素をリストに cons する』といいます。たとえば、

```
(setq list (cons newelt list))
```

この例で用いられた `list` という変数と、次に説明する `list` という関数との間で衝突が起きないことに注意してください。いかなるシンボルも変数名、関数名、両方の目的に使うことができます。

list &rest *objects* Function

この関数は、 *objects* をその要素とするリストを生成します。結果のリストは常に `nil` で終端されます。 *objects* が与えられなかった場合、空のリストが返されます。

```
(list 1 2 3 4 5)
     ⇒ (1 2 3 4 5)
(list 1 2 '(3 4 5) 'foo)
     ⇒ (1 2 (3 4 5) foo)
(list)
     ⇒ nil
```

make-list *length object* Function

この関数は、長さが *length* で、その全ての要素が *object* と同じ値をもつリストを生成します。 `make-list` と `make-string` を比較してください (4.3 節「文字列の生成」 p.56 参照)。

```
(make-list 3 'pigs)
     ⇒ (pigs pigs pigs)
(make-list 0 'pigs)
     ⇒ nil
```

append &rest *sequences* Function

この関数は、 *sequences* の全ての要素を含むリストを返します。 *sequences* はリストかベクタ、もしくは文字列のいずれでも構いませんが、最後の引数はリストであるべきです。最後のものを除く全ての引数は、コピーされるので、いずれも変更されません。

もっと一般化していえば、 `append` の最後の引数は、どんな Lisp オブジェクトでも構いません。最後の引数はコピーも変換もされません。新しいリストの最後のコンス・セルの CDR になります。最後の引数自身がリストである場合、その要素はすべてそのまま結果のリストの要素になります。最後の要素がリストではない場合は、最後の CDR が `nil` ではないため、真のリストにはならず、結果は “ドット・リスト (dotted list)” になります。

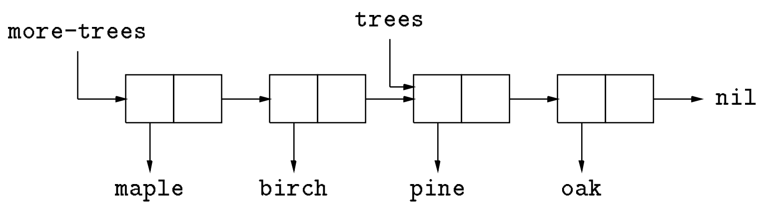
コピーをせずにリストを連結する方法は、 5.6.3 節「リストを再構成する関数」 p.77 の `nconc` を参照してください。

次に `append` を用いた例を示します。

```
(setq trees '(pine oak))
     ⇒ (pine oak)
(setq more-trees (append '(maple birch) trees))
     ⇒ (maple birch pine oak)

trees
     ⇒ (pine oak)
more-trees
     ⇒ (maple birch pine oak)
```

```
(eq trees (cdr (cdr more-trees)))
     ⇒ t
```

箱表記を見れば、 append がどのように実行されるかがわかります。変数 trees は、(pine oak) に設定され、次に変数 more-trees は、リスト (maple birch pine oak) となります。一方、変数 trees はもとのリストを参照したままです。



sequences の中に空の *sequence* があっても、 append が返す値には、何の影響も与えません。その結果、最後の引数を nil にすることで、そこまでの引数のコピーを生成することができます。

```
trees
     ⇒ (pine oak)
(setq wood (append trees ()))
     ⇒ (pine oak)
wood
     ⇒ (pine oak)
(eq wood trees)
     ⇒ nil
```

かつて、関数 copy-sequence が考案されるまでは、これがリストを生成する普通の方法でした。第 6 章「シーケンス、配列、ベクタ」 p.85 を参照してください。

apply を用いることで、リストの中のそれぞれのリストに含まれる全てのリストを append することができます。

```
(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)
```

append に *sequences* が与えられなかった場合、 nil が返されます。

```
(append)
     ⇒ nil
```

これらは最後の引数がリストでない場合の例です。

```
(append '(x y) 'z)
     ⇒ (x y . z)
(append '(x y) [z])
     ⇒ (x y . [z])
```

二つめの例は、最後の引数が列ではあるがリストではないとき、その列の各要素は結果のリストの要素にはならないことを示しています。その代わりその列は、リストではない最後の引数と同様に、最後の CDR となります。

append 関数は、引数に整数をとることもできます。その整数の 10 進印字表現を作って数字の文字列へ変換し、もとの整数のかわりにこの文字列を使用します。この機能を使わないでください。我々はこの機能を取り除く予定です。もしすでにあなたのプログラムでこの機能を使用しているのであれば、ただちに修正してください！整数を 10 進数に変換する適切な方法は format (4.6 節「文字列の書式変換」 p.61 参照) もしくは number-to-string (4.5 節「文字と文字列間の変換」 p.60 参照) を用いることです。

reverse *list* Function

この関数は、 *list* の要素を逆順に並べた新しいリストを生成します。もとの引数 *list* は変更されません。

```
(setq x '(1 2 3 4))
     ⇒ (1 2 3 4)
(reverse x)
     ⇒ (4 3 2 1)
x
     ⇒ (1 2 3 4)
```

5.6 リスト構造の変更

setcar と setcdr というプリミティブを用いて、コンス・セルの CAR と CDR の内容を変更することができます。

> **Common Lisp 注意書き:** Common Lisp には、リスト構造を変更する関数 rplaca と rplacd があります。これらは setcar や setcdr と同様な処理を行ないますが、setcar や setcdr が新たな CAR や CDR の値を返すのに対して Common Lisp の関数はコンス・セルを返します。

5.6.1 setcar によるリスト要素の変更

setcar によって、コンス・セルの CAR の変更が行なわれます。リストに対して用いられた場合、 setcar は、リストのある要素を別の要素で置き換えます。

setcar *cons object* Function

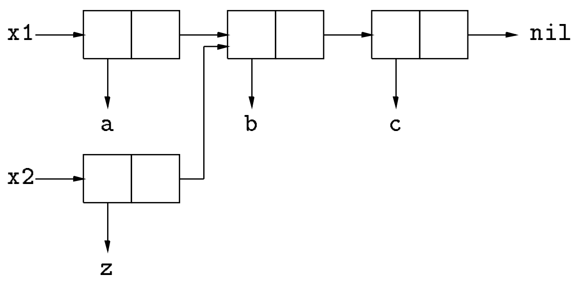
この関数は *cons* の CAR を *object* に置き換えます。 *object* の値を返します。たとえば、

```
(setq x '(1 2))
     ⇒ (1 2)
(setcar x 4)
     ⇒ 4
x
     ⇒ (4 2)
```

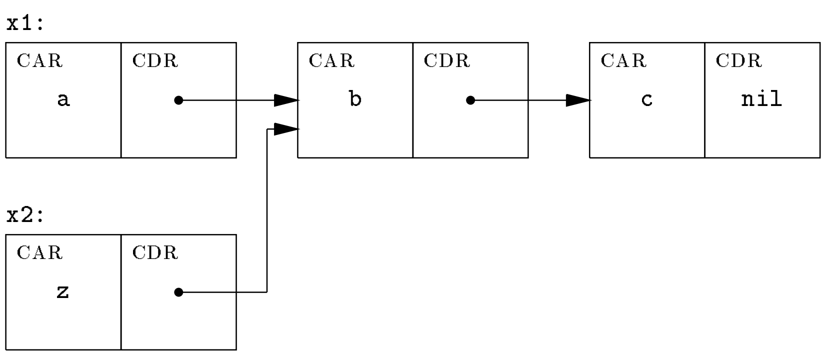
あるコンス・セルがいくつかのリストで共有される構造の一部分であるときは、そのコンス・セルに新しい CAR を設定することによって、これらのリストのそれぞれの要素が変更されます。次に例をあげます。

```
;; 一部分が共有された二つのリストを生成する。
(setq x1 '(a b c))
     ⇒ (a b c)
(setq x2 (cons 'z (cdr x1)))
     ⇒ (z b c)

;; 共有された CAR を変更する。
(setcar (cdr x1) 'foo)
     ⇒ foo
x1                           ; 両方のリストが変更される。
     ⇒ (a foo c)
x2
     ⇒ (z foo c)

;; 共有されていない CAR を変更する。
(setcar x1 'baz)
     ⇒ baz
x1                           ; 片方のリストのみが変更される。
     ⇒ (baz foo c)
x2
     ⇒ (z foo c)
```

次は、変数 x1 と x2 にある二つのリストの間での共有構造を図示したものです。なぜ b を変更することで両方の変数が変更されるかを示しています。



次は同じ関係を示す、別の箱表記です。



5.6.2 リストの CDR の変更

CDR を変更する最も低水準のプリミティブは `setcdr` です。

setcdr *cons object* Function

この関数は、 *cons* の CDR を *object* に置き換えます。 *object* の値を返します。

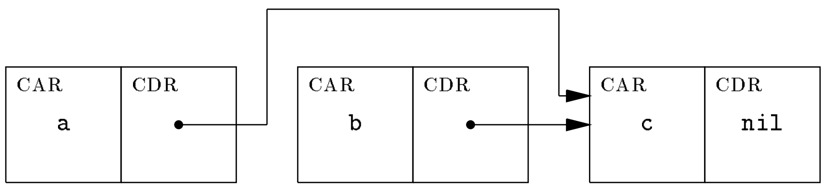
次は、あるリストの CDR を別のリストで置き換える例です。リストの最初の要素以外の要素は取り除かれ、要素の異なる並びに置き換えられます。最初の要素はリストの CAR に格納され、 CDR の一部ではないため、変更されません。

```
(setq x '(1 2 3))
     ⇒ (1 2 3)
(setcdr x '(4))
     ⇒ (4)
x
     ⇒ (1 4)
```

リスト中のコンス・セルの CDR を変更することで、リストの中程の要素を取り除くことができます。たとえばリスト `(a b c)` の 2 番目の要素 `b` を、最初のセルの CDR を変更することによって取り除くことができます。

```
(setq x1 '(a b c))
     ⇒ (a b c)
(setcdr x1 (cdr (cdr x1)))
     ⇒ (c)
x1
     ⇒ (a c)
```

これは、結果の箱表記です。

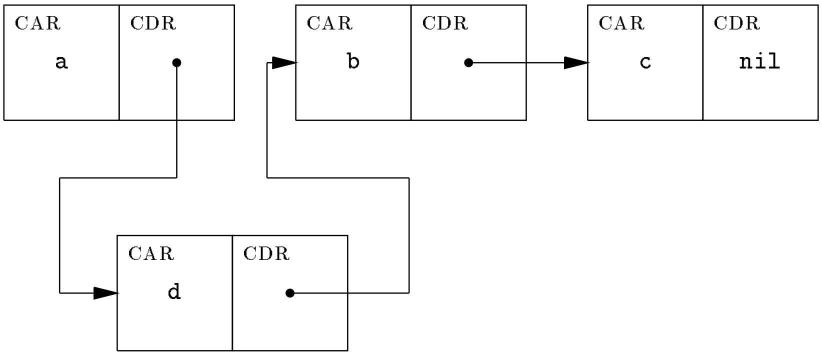


要素 `b` を持っていた 2 番目のコンス・セルは依然として存在し、その CAR は `b` のままですが、このリストの一部分ではありません。

同様に新しい要素を挿入することは、 CDR を変更することで容易に行うことができます。

```
(setq x1 '(a b c))
     ⇒ (a b c)
(setcdr x1 (cons 'd (cdr x1)))
     ⇒ (d b c)
x1
     ⇒ (a d b c)
```

次は、結果の箱表記です。



5.6.3 リストを再構成する関数

以下のものは、コンス・セルの CDR を変更することで、“破壊的に” リストを並び変える関数です。これらの関数が “破壊的” と呼ばれるのは、戻り値となる新しいリストを生成するために引数として与えられたもとのリストを分解するからです。

次節の関数 `delq` は、破壊的なリスト操作のもう一つの例です。

nconc &rest *lists* Function

この関数は、*lists* の全ての要素を含むリストを返します。`append` (5.5 節「コンス・セルとリストの生成」p.71 参照) と異なり、*lists* はコピーされません。そのかわり、*lists* のそれぞれの最後の CDR が次のリストを参照するように修正されます。*lists* の最後のものは変更されません。たとえば、

```
(setq x '(1 2 3))
     ⇒ (1 2 3)
(nconc x '(4 5))
     ⇒ (1 2 3 4 5)
x
     ⇒ (1 2 3 4 5)
```

`nconc` の最後の引数自身は修正されないので、上記の例のように、`'(4 5)` のような定数リストを用いるのは理にかなっています。同様な理由で、最後の引数はリストである必要はありません。

```
(setq x '(1 2 3))
     ⇒ (1 2 3)
(nconc x 'z)
     ⇒ (1 2 3 . z)
x
     ⇒ (1 2 3 . z)
```

`nconc` の最後の引数以外の引数に、quote された定数リストを用いることはよく陥る落し穴です。これを行うと、実行するたびにプログラムが修正されてしまいます。次のようなことが起きます。

```
(defun add-foo (x)            ; その引数の先頭に foo を加える
  (nconc '(foo) x))           ;   関数を定義したい。
```

```
(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc (quote (foo)) x))
(setq xx (add-foo '(1 2)))    ; うまくゆくように見える。
     ⇒ (foo 1 2)
(setq xy (add-foo '(3 4)))    ; 何が起きたか?
     ⇒ (foo 1 2 3 4)
(eq xx xy)
     ⇒ t
(symbol-function 'add-foo)
     ⇒ (lambda (x) (nconc (quote (foo 1 2 3 4) x)))
```

nreverse *list* Function

この関数は、 *list* の要素の順序を逆にします。 `reverse` と異なり、 `nreverse` はリストを構成するコンス・セルの CDR をひっくり返し、引数を修正します。 *list* の最後のコンス・セルであったものは、結果のリストの最初のセルになります。

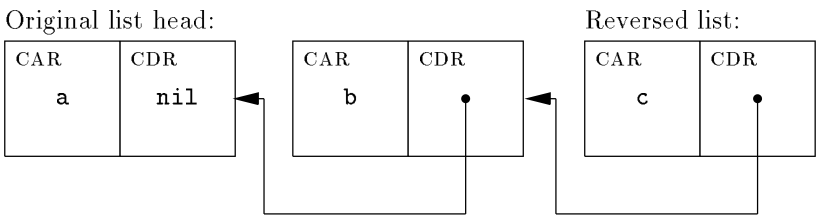
たとえば、

```
(setq x '(1 2 3 4))
     ⇒ (1 2 3 4)
x
     ⇒ (1 2 3 4)
(nreverse x)
     ⇒ (4 3 2 1)
;; 先頭だったセルは最後尾になっている。
x
     ⇒ (1)
```

普通、混乱を避けるため、 `nreverse` の結果をもとのリストをもっていた同じ変数に代入しなおします。

```
(setq x (nreverse x))
```

次に、お得意の例で、 `(a b c)` の `nreverse` を図示します。



sort *list predicate* Function

この関数は、 *list* を安定的に (stably) 並べ換え、並べ換えられたリストを返します。もとの *list* は破壊されます。要素を *predicate* を用いて比較します。安定的な並べ換えとは、並べ換えキーが等しい場合は相対的な順序が変わらない並べ換えのことです。この安定性は、異なる *predicate* を用いて、続けて並べ換えを行うときに重要です。

引数 *predicate* は、二つの引数をとる関数でなければなりません。この関数は、 *list* の二つの要素を引数にして呼ばれます。昇順に並べ換えるためには、 *predicate* は最初の要素

が 2 番目の要素より小さいときに t を、そうでないときには nil を返さなければなりません。

sort が破壊的であるということは、 CDR を書き換えて *list* を構成するコンス・セルを再構成するということです。非破壊的並べ換え関数であれば、要素をある順序で並べ換えて格納する、新しいコンス・セルを生成します。もとのリストを破壊することなく並べ換えを行ないたいのなら、最初に copy-sequence を用いてコピーした後、並べ換えを行ってください。

並べ換えは、 *list* のコンス・セルの CAR は変更しません。つまり、もともと要素 a を持っていた *list* の中のコンス・セルは、並べ換え後も a をその CAR に持っています。ただし CDR が変更されるため、リストの中の位置は変わっています。たとえば、

```
(setq nums '(1 3 2 6 5 4 0))
     ⇒ (1 3 2 6 5 4 0)
(sort nums '<)
     ⇒ (0 1 2 3 4 5 6)
nums
     ⇒ (1 2 3 4 5 6)
```

nums のリストの中に 0 が含まれていないことに注目してください。 nums の指すコンス・セルは並び換えを行う前と同じですが、もはやリストの最初のコンス・セルではありません。変数が並び換えられたリスト全体を持っていると考えてはなりません！ そのかわり sort の結果を保存してから使用してください。ほとんどの場合、結果をもとのリストを格納していた変数へ再度格納します。

```
(setq nums (sort nums '<))
```

並び換えを行うその他の関数に関しては、 29.14 節「整列」 p.499 を参照してください。 sort の有用な例は、 21.2 節「説明文字列へのアクセス」 p.340 の documentation を参照してください。

5.7 集合としてのリストの使用

リストは、順序を持たない数学的な集合を表わすことができます。ある値がリストの中にある場合、その値が集合の要素であるとします。二つの集合の和集合を生成するためには (同じ要素を 2 重にもつことを気にしないのであれば)、 append を使用してください。集合のためのその他の便利な関数として、 memq と delq と、これらの equal 版である member と delete があります。

> **Common Lisp 注意書き:** Common Lisp には集合の操作のために (要素が 2 重になることを避ける) 関数 union と intersection がありますが、 GNU Emacs Lisp はこれらを持っていません。必要であれば Lisp を用いて作ることができます。

memq *object list* Function

この関数は、 *object* が *list* の要素であるかどうかを調べます。もしそうであれば、 memq は *object* が最初に見つかったところから始まるリストを返します。そうでなければ nil

を返します。 `memq` の ‘q’ という文字はリストの要素と *object* を比較するときに `eq` を用いることを示します。たとえば、

```
(memq 'b '(a b c b a))
     ⇒ (b c b a)
(memq '(2) '((1) (2)))    ; (2) と (2) は eq ではない。
     ⇒ nil
```

delq *object list* Function

この関数は *list* から *object* と `eq` である全ての要素を破壊的に取り除きます。 `delq` の ‘q’ という文字は `memq` と同様に、リストの要素と *object* を比較するときに `eq` を用いることを示します。

`delq` がリストの先頭から要素を取り除くときは、単純にリストを順に調べていった後、これらの要素の後の部分のリストを返します。

```
(delq 'a '(a b c)) ≡ (cdr '(a b c))
```

リストの中間の要素が取り除かれる場合は CDR の書き換えが行なわれます (5.6.2 節「リストの CDR の変更」p.76 参照)。

```
(setq sample-list '(a b c (4)))
     ⇒ (a b c (4))
(delq 'a sample-list)
     ⇒ (b c (4))
sample-list
     ⇒ (a b c (4))
(delq 'c sample-list)
     ⇒ (a b (4))
sample-list
     ⇒ (a b (4))
```

`(delq 'c sample-list)` が 3 番目の要素を取り除くために `sample-list` を修正するのに対して、 `(delq 'a sample-list)` は何も取り除かないことに気をつけてください。後者は単により短いリストを返すだけです。引数 *list* を持っていた変数が、その後、より少ない個数の要素を持っているとも、まだ同じリストを持っているとも仮定してはいけません！ その代わりに、`delq` の結果を保存し、それを用いてください。通常結果をもとのリストを格納していた変数へ再度格納します。

```
(setq flowers (delq 'rose flowers))
```

次の例では、 `delq` が照合した `(4)` と `sample-list` の中の `(4)` は `eq` ではありません。

```
(delq '(4) sample-list)
     ⇒ (a c (4))
```

次の二つの関数は、 `memq` と `delq` に似ていますが、要素の比較に `eq` の代わりに `equal` を使用します。これらは Emacs バージョン 19 で新たに追加されました。

member *object list* Function

この関数 `member` は、 `equal` を使って *object* と *list* の要素を比較することで、 *object* が要素であるかどうかを調べます。もしそうであれば、 `member` は、 *object* が最初に見つかったところから始まるリストを返します。そうでなければ `nil` を返します。

この関数と `memq` を比較してください。

```
(member '(2) '((1) (2)))  ; (2) と (2) は equal。
     ⇒ ((2))
(memq '(2) '((1) (2)))    ; (2) と (2) は eq ではない。
     ⇒ nil
;; 同じ内容の二つの文字列は equal である。
(member "foo" '("foo" "bar"))
     ⇒ ("foo" "bar")
```

delete *object list* Function

この関数は、 *object* と `equal` である全ての要素を、 *list* から破壊的に取り除きます。`memq` が `member` と対応するのと同様に、この関数は `delq` に対応します。 `member` と同様に、 *object* と要素を比較するのに `equal` を用います。一致する要素を見つけたときには `delq` と同様にその要素を取り除きます。たとえば、

```
(delete '(2) '((2) (1) (2)))
     ⇒ ((1))
```

> **Comman Lisp 注意書き:** GNU Emacs Lisp の関数 `member` と `delete` は、 Common Lisp ではなく Maclisp に由来します。 Common Lisp では要素の比較に `equal` を用いません。

変数に格納されたリストに要素を加える他の方法として、 10.7 節「変数の値を変更する方法」 p.135 の関数 `add-to-list` も参照してください。

5.8 連想リスト

『連想リスト (association list)』 (略して『alist』) はキーから値への関連を記憶します。連想リストは、連想子 (association) と呼ぶコンス・セルのリストです。それぞれのセルの CAR は『キー (key)』であり、その CDR はそれに関連づけられた値、『連想値 (associated value)』です。[1]

次は alist の例です。 `pine` というキーは、 `cones` という値に関連づけられます。 `oak` というキーは `acorns` という値に関連づけられます。さらに `maple` というキーは `seeds` という値に関連づけられます。

```
'((pine . cones)
  (oak . acorns)
  (maple . seeds))
```

[1] ここでの “キー” は “キー・シーケンス” という用語とは関係ありません。ここでは表の中の項目を探すための値を意味します。この場合 alist が表であり、 alist の各連想がその項目です。

alist の連想値およびキーは、どんな Lisp オブジェクトでも構いません。たとえば、次の alist では、シンボル `a` と数 `1` が関連づけられ、文字列 `"b"` とリスト `(2 3)` が関連づけられます。つまり、`(2 3)` は、その alist 要素の CDR です。

```
((a . 1) ("b" 2 3))
```

連想値を、要素の CDR の CAR に格納するように alist を構成するのが適切な場合もあります。次はその例です。

```
'((rose red) (lily white) (buttercup yellow))
```

ここで、 `red` は `rose` の連想値であるとみなします。この方法の一つの利点は他の関連する情報を、他の項目のリストでさえ、 CDR の CDR の中に格納することが可能であることです。一つの欠点は、ある値をもつ要素を探すために `rassq` (次の項を参照) を使うことができないことです。これらの制約が問題でない場合、その alist に対して一貫性を保つのであれば、どう選択するかは好みの問題です。

上で示した同じ alist は、要素の CDR に連想値を持つとみなすこともできることに注意してください。その場合、 `rose` の連想値は、リスト `(red)` になります。

連想リストは、新しい連想子をリストの先頭に簡単に加えることができるため、連想リストを用いなければスタックを用いるような情報を記録するのに、しばしば使用されます。ある与えられたキーに関連する値を連想リストから検索するときで、複数の関連する連想子がある場合には、最初に見つかったものが返されます。

Emacs Lisp では、連想リストのある要素がコンス・セルではなくともエラーにはなりません。 alist に対する検索関数は、単にこれらの要素を無視するだけです。多くの他の Lisp の実装ではこのような場合はエラーが通知されます。

いくつかの点で、属性リスト (property list) が連想リストと類似していることに気をつけてください。属性リストは、それぞれのキーがただ一回のみ用いられる連想リストのように振舞います。属性リストと連想リストの比較のため 7.4 節「属性リスト」 p.98 を参照してください。

assoc *key alist* Function

この関数は、 *alist* の中の *key* を持つ最初の連想子を返します。 `equal` を用いて alist の要素と *key* を比較します (2.6 節「等価述語」 p.38 参照)。 *alist* の中に、 *key* と `equal` である CAR を持つ連想子がない場合、 `nil` を返します。

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
(assoc 'oak trees)
     ⇒ (oak . acorns)
(cdr (assoc 'oak trees))
     ⇒ acorns
(assoc 'birch trees)
     ⇒ nil
```

次はキーや値がシンボルではない別の例です。

```
(setq needles-per-cluster
      '((2 "Austrian Pine" "Red Pine")
        (3 "Pitch Pine")
        (5 "White Pine")))

(cdr (assoc 3 needles-per-cluster))
```

```
     ⇒ ("Pitch Pine")
(cdr (assoc 2 needles-per-cluster))
     ⇒ ("Austrian Pine" "Red Pine")
```

rassoc *value alist* Function

この関数は、 *alist* の中の *value* を値に持つ最初の連想子を返します。 *alist* の中に *value* と equal である CDR を持つものがない場合、 nil を返します。

rassoc は、それぞれの *alist* の CAR の代わりに、 CDR と比較することを除いて、 assoc に似ています。与えられた値に関連するキーを探す “逆 (reverse) assoc” と考えることができます。

assq *key alist* Function

この関数は、 *alist* の中の *key* を持つ最初の連想子を返す点で assoc に似ていますが、 equal のかわりに eq を用いて比較を行ないます。 *alist* の中に *key* と eq である CAR を持つものがない場合は assq は nil を返します。 eq は equal よりも高速であり、ほとんどの alist はキーとしてシンボルを用いるため、この関数は assoc よりもよく使われます。 2.6 節「等価述語」 p.38 を参照してください。

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))
     ⇒ ((pine . cones) (oak . acorns) (maple . seeds))
(assq 'pine trees)
     ⇒ (pine . cones)
```

一方、 assq は、キーがシンボルではない alist には通常不便な関数です。

```
(setq leaves
      '(("simple leaves" . oak)
        ("compound leaves" . horsechestnut)))

(assq "simple leaves" leaves)
     ⇒ nil
(assoc "simple leaves" leaves)
     ⇒ ("simple leaves" . oak)
```

rassq *value alist* Function

この関数は、 *alist* の中の *value* の値を持つ最初の連想子を返します。 *alist* の中に *valude* と eq である CDR を持つものがない場合、 nil を返します。

rassq は、それぞれの *alist* の CAR の代わりに CDR と比較することを除いて、 assq に似ています。与えられた値に関連するキーを探す “逆 (reverse) assq” と考えることができます。

たとえば、

```
(setq trees '((pine . cones) (oak . acorns) (maple . seeds)))

(rassq 'acorns trees)
     ⇒ (oak . acorns)
(rassq 'spores trees)
     ⇒ nil
```

rassq は、要素の CDR の CAR に格納された値を探すことはできないことに気をつけてください。

```
(setq colors '((rose red) (lily white) (buttercup yellow)))

(rassq 'white colors)
     ⇒ nil
```

この場合は、 (lily white) という連想子の CDR は、 white というシンボルではなく、 (white) というリストです。次のように連想子をドット対記法を用いて表現すれば、よりわかりやすくなります。

```
(lily white) ≡ (lily . (white))
```

copy-alist *alist* Function

この関数は、 *alist* の 2 段の深さのコピー (two-level deep copy) を返します。それぞれの連想の新しい組を作るので、昔の alist を変更することなく新しい alist の連想を変更することが可能となります。

```
(setq needles-per-cluster
      '((2 . ("Austrian Pine" "Red Pine"))
        (3 . ("Pitch Pine"))
        (5 . ("White Pine"))))
⇒
((2 "Austrian Pine" "Red Pine")
 (3 "Pitch Pine")
 (5 "White Pine"))

(setq copy (copy-alist needles-per-cluster))
⇒
((2 "Austrian Pine" "Red Pine")
 (3 "Pitch Pine")
 (5 "White Pine"))

(eq needles-per-cluster copy)
     ⇒ nil
(equal needles-per-cluster copy)
     ⇒ t
(eq (car needles-per-cluster) (car copy))
     ⇒ nil
(cdr (car (cdr needles-per-cluster)))
     ⇒ ("Pitch Pine")
(eq (cdr (car (cdr needles-per-cluster)))
    (cdr (car (cdr copy))))
     ⇒ t
```

この例は、 copy-alist を使っておけば、あるコピーの連想を、他に影響を与えることなく変更できることを示します。

```
(setcdr (assq 3 copy) '("Martian Vacuum Pine"))
(cdr (assq 3 needles-per-cluster))
     ⇒ ("Pitch Pine")
```

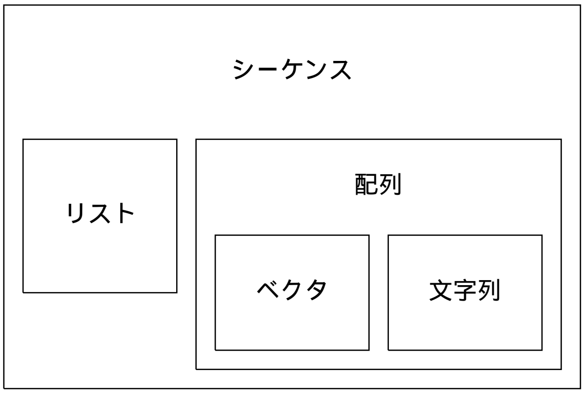
6 シーケンス、配列、ベクタ

『シーケンス (sequence)』型は、リスト、ベクタ、文字列というほかの三つの Lisp 型の和集合であることを思い出してください。いいかえれば、任意のリストはシーケンスであり、任意のベクタはシーケンスであり、任意の文字列はシーケンスです。全てのシーケンスが共通してもつ性質は、順序づけられた要素の集まりだということです。

『配列 (array)』は、プリミティブなオブジェクトの一つで、要素ごとにスロットを持ちます。全ての要素は、一定の時間でアクセス可能ですが、既存の配列の長さを変えることはできません。文字列とベクタは配列の二つの型です。

リストは要素の並びですが、プリミティブなオブジェクトの一つではありません。リストはコンス・セルからなり、要素ごとに一つのセルが割当てられています。 *n* 番目の要素を見つけるには、 *n* 番目のコンス・セルまで全て見なければならないので、リストの先頭から遠い要素ほどアクセスに時間がかかります。しかし、リストだと要素を追加することや取り除くことができます。

次の図式は、これらの型の間の関係を示しています:



ベクタとリストの要素は、任意の Lisp オブジェクトで構いません。文字列の要素は、全て文字です。

6.1 シーケンス

Emacs Lisp では、『シーケンス (sequence)』は、リスト、ベクタ、文字列のいずれかです。全てのシーケンスが共通してもつ性質は、順序づけられた要素の集まりだということです。本節では、これらのシーケンスを受けつける関数を説明します。

sequencep *object* Function

object がリスト、ベクタ、文字列ならば `t` を、そうでなければ `nil` を返します。

copy-sequence *sequence* Function

sequence のコピーを返します。コピーは元のシーケンスと同じ型のオブジェクトで、同じ要素を同じ順序で持ちます。

コピーに新たな要素を格納しても、元の *sequence* に影響しませんし、その逆も同じです。けれども、作られた方のシーケンスの要素はコピーではありません。それらは元の要素と同一 (eq) のものです。したがって、コピーされたシーケンスでこれらの要素内に行なった変更は、元のシーケンスでも見ることができます。

もしそのシーケンスがテキスト属性をもつ文字列であるならば、コピーでの属性リスト自身もコピーであり、元の属性リストを共有するものではありません。けれども、属性の実際の値は共有されています。 29.18 節「テキスト属性」 p.509 を参照してください。

シーケンスをコピーするほかの方法については、 5.5 節「コンス・セルとリストの生成」 p.71 の append、 4.3 節「文字列の生成」 p.56 の concat、 6.4 節「ベクタ」 p.89 の vconcat なども参照してください。

```
(setq bar '(1 2))
     ⇒ (1 2)
(setq x (vector 'foo bar))
     ⇒ [foo (1 2)]
(setq y (copy-sequence x))
     ⇒ [foo (1 2)]

(eq x y)
     ⇒ nil
(equal x y)
     ⇒ t
(eq (elt x 1) (elt y 1))
     ⇒ t

;; シーケンスの 1 要素を置き換える。
(aset x 0 'quux)
x ⇒ [quux (1 2)]
y ⇒ [foo (1 2)]

;; 共有要素の内部を修正する。
(setcar (aref x 1) 69)
x ⇒ [quux (69 2)]
y ⇒ [foo (69 2)]
```

length *sequence* Function

sequence 中の要素の個数を返します。 *sequence* がコンス・セルであっても、(最後の CDR が nil ではないので) それがリストではないならば、 wrong-type-argument エラーが通知されます。

```
(length '(1 2 3))
    ⇒ 3
(length ())
    ⇒ 0
```

```
(length "foobar")
    ⇒ 6
(length [1 2 3])
    ⇒ 3
```

elt *sequence index* Function

この関数は、 *sequence* の添字 *index* 番目の要素を返します。正当な *index* の値は整数で、範囲は 0 から *sequence* の長さよりも一つ小さい値まででです。範囲外の *index* 値を指定すると、 *sequence* がリストのときは `nil` を返します。それ以外の場合は、 `args-out-of-range` エラーを発します。

```
(elt [1 2 3 4] 2)
     ⇒ 3
(elt '(1 2 3 4) 2)
     ⇒ 3
(char-to-string (elt "1234" 2))
     ⇒ "3"
(elt [1 2 3 4] 4)
     error Args out of range: [1 2 3 4], 4
(elt [1 2 3 4] -1)
     error Args out of range: [1 2 3 4], -1
```

この関数は、 `aref`(6.3 節「配列に関する関数」 p.88 参照) と `nth`(5.4 節「リストの要素の参照」 p.69 参照) とを一般化したものです。

6.2 配列

『配列 (array)』オブジェクトは、スロットを持ち、そこにほかの Lisp オブジェクトを複数、格納できます。それらを配列要素と呼びます。任意の配列要素は一定の時間でアクセスできます。それに対しリストの要素は、そのリスト内の位置に比例したアクセス時間を要します。

配列作成の際は、それがいくつ要素をもつかを指定しなければなりません。割当てられる空間の大きさは要素数に依存します。したがって、いったん作成した配列の大きさを変更することは不可能です。要素の追加や削除はできません。しかし、要素を異なる値で置き換えることはできます。

Emacs は、ともに 1 次元である二つの配列型、『文字列 (string)』と『ベクタ (vector)』を定義しています。ベクタは一般化された配列です。つまり、その要素は任意の Lisp オブジェクトで構いません。文字列は、その要素として文字 (即ち、 0 から 255 までの整数) だけが許される、特化した配列です。いずれの配列型も固有のリード構文を持ちます。 2.3.8 節「文字列型」 p.28 と、 2.3.9 節「ベクタ型」 p.30 を参照してください。

いずれの配列も次の特性を持ちます:

- 配列の第 1 要素の添字は 0 で、第 2 要素の添字は 1 で、以下同様です。これを『0 起源 (zero-origin)』添字づけと呼びます。たとえば、 4 要素の配列は添字 0, 1, 2, 3 を持ちます。

配列の要素を関数 `aref` や `aset` で、それぞれ参照あるいは変更することができます (6.3 節「配列に関する関数」p.88 参照)。

原則的には、テキスト文字の配列が欲しい場合、文字列あるいはベクタのどちらでも構いません。しかし、実用的には、そのような場合は常に文字列を選んでいます。それには四つ理由があります:

- 文字列は、同じ要素のベクタの 1/4 の空間しか必要としません。
- 文字列は、内容がよりはっきりと文字であるとわかるように表示されます。
- 文字列は、テキスト属性をもつことができます。 29.18 節「テキスト属性」p.509 を参照してください。
- Emacs の機能のうち、編集と I/O に特化したもののほとんどが、文字列のみを受けつけます。たとえば、文字のベクタをバッファに挿入しようとしても、文字列のようには行なえません。第 4 章「文字列と文字」p.55 を参照してください。

対照的に、 (キー・シーケンスなどの) キーボード入力文字の配列に対しては、ベクタであることが必要な場合もあります。というのも、多くのキーボード入力文字は、文字列に収まるような範囲の外にあるためです。 18.6.1 節「キー・シーケンス入力」p.279 を参照してください。

6.3 配列に関する関数

本節では、文字列とベクタを受けつける関数を説明します。

arrayp *object* Function

この関数は、 *object* が配列 (即ち、ベクタあるいは文字列) であるならば、 `t` を返します。

```
(arrayp [a])
⇒ t
(arrayp "asdf")
⇒ t
```

aref *array index* Function

この関数は、 *array* の *index* 番目の要素を返します。第 1 要素の添字は 0 です。

```
(setq primes [2 3 5 7 11 13])
     ⇒ [2 3 5 7 11 13]
(aref primes 4)
     ⇒ 11
(elt primes 4)
     ⇒ 11

(aref "abcdefg" 1)
     ⇒ 98               ; ‘b’ は ASCII コード 98.
```

6.1 節「シーケンス」p.85 にある関数 `elt` も参照してください。

aset *array index object* Function

この関数は、 *array* の *index* 番目の要素に *object* を設定します。この関数は、 *object* を返します。

```
(setq w [foo bar baz])
     ⇒ [foo bar baz]
(aset w 0 'fu)
     ⇒ fu
w
     ⇒ [fu bar baz]

(setq x "asdfasfd")
     ⇒ "asdfasfd"
(aset x 3 ?Z)
     ⇒ 90
x
     ⇒ "asdZasfd"
```

array が文字列なのに *object* が文字ではないならば、結果は `wrong-type-argument` エラーとなります。

fillarray *array object* Function

この関数は、 *object* で *array* を埋めます。その結果、 *array* の各要素は *object* となります。この関数は、 *array* を返します。

```
(setq a [a b c d e f g])
     ⇒ [a b c d e f g]
(fillarray a 0)
     ⇒ [0 0 0 0 0 0 0]
a
     ⇒ [0 0 0 0 0 0 0]
(setq s "When in the course")
     ⇒ "When in the course"
(fillarray s ?-)
     ⇒ "------------------"
```

array が文字列なのに *object* が文字ではないならば、結果は `wrong-type-argument` エラーとなります。

シーケンス一般についての関数 `copy-sequence` や `length` は、配列であろうオブジェクトに対してもしばしば有用です。 6.1 節「シーケンス」 p.85 を参照してください。

6.4 ベクタ

Lisp における配列は、ほとんどの言語における配列同様、メモリのひとかたまりからなり、要素には一定の時間でアクセスできます。『ベクタ (vector)』は汎用の配列であり、その要素は任意の Lisp オブジェクトで構いません (これに対して、 Emacs Lisp でのほかの配列『文字列 (string)』では、その要素は文字でなければなりません)。ベクタは Emacs 内で、構文テーブル (整数のベクタ) として、 obarray(シンボルのベクタ) として、キーマップ (コマンドのベクタ)

として使われています。内部的にはさらに、バイトコンパイルされた関数表現の一部としても使われます。そのような関数を印字すると、ベクタが中にあるのが見えます。

Emacs Lisp では、ベクタ要素の添字は 0 からはじまり、そこから順に数えて行きます。

ベクタは要素を角括弧で囲んだものとして表示されます。よって、要素がシンボル `a`, `b`, `a` であるベクタは、 `[a b a]` と表示されます。 Lisp の入力でもベクタを同じように書くことができます。

ベクタは、文字列や数同様に、評価に際しては定数とみなされます。その評価結果は同じベクタです。評価もされないし、ベクタの要素を調べることさえされません。 8.2.1 節「自己評価型形式」p.104 を参照してください。

次に、これらの原則の例を示します。

```
(setq avector [1 two '(three) "four" [five]])
     ⇒ [1 two (quote (three)) "four" [five]]
(eval avector)
     ⇒ [1 two (quote (three)) "four" [five]]
(eq avector (eval avector))
     ⇒ t
```

6.5 ベクタを操作する関数

ベクタに関係するいくつかの関数を挙げます:

vectorp *object* Function

この関数は、 *object* がベクタなら `t` を返します。

```
(vectorp [a])
     ⇒ t
(vectorp "asdf")
     ⇒ nil
```

vector &rest *objects* Function

この関数は、引数 *objects* が要素になるようなベクタを作成し、返します。

```
(vector 'foo 23 [bar baz] "rats")
     ⇒ [foo 23 [bar baz] "rats"]
(vector)
     ⇒ []
```

make-vector *length object* Function

この関数は、 *length* 個の要素からなる新しいベクタを返します。各要素を *object* に初期化します。

```
(setq sleepy (make-vector 9 'Z))
     ⇒ [Z Z Z Z Z Z Z Z Z]
```

vconcat &rest *sequences* Function

この関数は、 *sequences* の全要素を含む新しいベクタを返します。引数 *sequences* はリストでもベクタでも文字列でも構いません。 *sequences* が何も与えられない場合、空ベクタを返します。

この値は新たに構築されたベクタであり、既存のいかなるベクタにも `eq` ではありません。

```
(setq a (vconcat '(A B C) '(D E F)))
     ⇒ [A B C D E F]
(eq a (vconcat a))
     ⇒ nil
(vconcat)
     ⇒ []
(vconcat [A B C] "aa" '(foo (6 7)))
     ⇒ [A B C 97 97 foo (6 7)]
```

関数 `vconcat` もまた、引数として整数を許します。引数をその整数の 10 進印字表現をなす数字文字列に変換し、それを元の整数の代わりに使用します。この機能は削除する予定なので、使ってはなりません。すでにこの機能を使っているならば、直ちにプログラムを変更してください！こういった方法で整数を数字に変換したいならば、 `format`(4.6 節「文字列の書式変換」 p.61 参照) あるいは `number-to-string`(4.5 節「文字と文字列間の変換」 p.60 参照) を使ってください。

これ以外の結合用関数については、 11.6 節「マッピング関数」 p.155 にある `mapconcat`、 4.3 節「文字列の生成」 p.56 にある `concat`、 5.5 節「コンス・セルとリストの生成」 p.71 にある `append` を参照してください。

`append` 関数でベクタを同じ要素をもつリストに変換する方法があります (5.5 節「コンス・セルとリストの生成」 p.71 参照):

```
(setq avector [1 two (quote (three)) "four" [five]])
     ⇒ [1 two (quote (three)) "four" [five]]
(append avector nil)
     ⇒ (1 two (quote (three)) "four" [five])
```

7 シンボル

『シンボル (symbol)』とは固有の名前をもつオブジェクトのことです。この章では、シンボルとそれを構成する要素、その属性リストについてお話しし、さらにそれらがどのようにして生成され、 intern されるかについて説明します。シンボルを変数名や関数名として利用することについては、章を改めて記述しますので、それらについては第 10 章「変数」p.127 と第 11 章「関数」p.147 を参照してください。シンボルの正確なリード構文については、 2.3.4 節「シンボル型」p.24 を御覧ください。

任意の Lisp オブジェクトがシンボルであるかどうかは、 `symbolp` で調べることができます:

symbolp *object* Function

この関数は、 *object* がシンボルならば `t` を返し、そうでなければ `nil` を返します。

7.1 シンボルの要素

各シンボルは四つの要素 (“セル” ともいう) を持っており、それぞれはさらに別のオブジェクトを参照します:

印字名　『印字名セル (print name cell)』は、読込みや印字の際にシンボルの名前を表す文字列を格納しています。 7.3 節「シンボルの作成と intern」p.95 の `symbol-name` を参照してください。

値　『値セル (value cell)』は、変数としてのシンボルの現在の値を格納します。シンボルが式として使われると、その式の値はシンボルの値セルの内容になります。 10.6 節「変数の値へのアクセス」p.134 の `symbol-value` を参照してください。

関数　『関数セル (function cell)』は、シンボルの関数定義を保持します。あるシンボルが関数として使われると、その関数定義がそこで使われます。このセルは、エディタのコマンド実行の際にシンボルにキーマップやキーボード・マクロを意味させるためにも用いられます。各シンボルは値と関数のセルを別に持ちますから、変数と関数の名前がぶつかることはありません。 11.8 節「関数セルの内容へのアクセス」p.158 の `symbol-function` を参照してください。

属性リスト　『属性リスト・セル (property list cell)』は、シンボルの属性リストを格納しています。 7.4 節「属性リスト」p.98 の `symbol-plist` を参照してください。

印字名セルは、常に文字列を格納し、これを変更することはできません。ほかの三つのセルには、与えられた任意の Lisp オブジェクトを個別にセットすることができます。

印字名セルは、そのシンボルの名前である文字列を保持しています。シンボルは、その名前によってテキスト的に表現されるため、同じ名前で二つのシンボルが存在しないことは重要です。これは Lisp リーダによって保証されます。というのは、シンボルを読み込むときには新しいシンボルを作成する前に、まず指定された名前を持つ既存のシンボルがないかどうか検索するからです (GNU Emacs Lisp では、この検索にはハッシュ・アルゴリズムと obarray を用います。 7.3 節「シンボルの作成と intern」p.95 を参照してください)。

通常の使い方では、関数セルは、 Lisp インタプリタがそこに期待するとおりの関数かマクロを保持します (第 8 章「評価」 p.101 参照)。また、ときどきは、キーボード・マクロ (18.13 節「キーボード・マクロ」 p.291 参照)、キーマップ (第 19 章「キーマップ」 p.293 参照) および自動ロード・オブジェクト (8.2.8 節「自動ロード」 p.109 参照) もシンボルの関数セルに格納されることもあります。しばしば、 "関数 `foo`" という言い方をしますが、実際にはシンボル `foo` の関数セルに格納された関数を意味します。これらは、必要なときにのみ使い分けられます。

属性リスト・セルは、通常正しい形式の属性リスト (7.4 節「属性リスト」 p.98 参照) を保持すべきものであり、多くの関数はそこに属性リストがあるものとみなしています。

関数セルや値セルは、『void』(無値) であるかも知れません。これは、そのセルが何のオブジェクトも参照していないことを示しています (これは、 `void` というシンボルを保持していることとは違いますし、 `nil` というシンボルを保持していることとも違います)。 void であるセルをいじると、 '`Symbol's value as variable is void`' というようなエラーになります。

`symbol-name`、 `symbol-value`、 `symbol-plist`、 `symbol-function` という四つの関数があり、これらは、シンボルの四つのセルの内容を返します。ここで、 `buffer-file-name` というシンボルの四つのセルの内容を例として示しましょう:

```
(symbol-name 'buffer-file-name)
     ⇒ "buffer-file-name"
(symbol-value 'buffer-file-name)
     ⇒ "/gnu/elisp/symbols.texi"
(symbol-plist 'buffer-file-name)
     ⇒ (variable-documentation 29529)
(symbol-function 'buffer-file-name)
     ⇒ #<subr buffer-file-name>
```

このシンボルは、カレント・バッファに対応づけられているファイルの名前を保持する変数ですから、今その値セルの内容は、 Emacs Lisp Manual のこの章のソース・ファイルの名前になっています。[1]属性リスト・セルは、 `buffer-file-name` という変数の説明文字列が '`DOC`' ファイルのどこで見つかるかを説明関数に伝える `(variable-documentation 29529)` というリストを格納しています (29529 とは、説明文字列が始まる場所の、 '`DOC`' ファイルの先頭からのオフセット値です)。関数セルは、このファイル名を返す関数を保持しています。 `buffer-file-name` が指すのはプリミティブ関数であり、これはリード構文を持たず、ハッシュ表記 (2.3.12 節「プリミティブ関数型」 p.30 参照) で印字されます。 Lisp 言語で記述された関数を指すシンボルの場合であれば、このセルはラムダ式 (あるいはバイトコード・オブジェクト) を持つはずです。

7.2 シンボルを定義する

Lisp において、『定義 (definition)』とは、あるシンボルを特定の用途で用いることを告知するための特殊形式です。 Emacs Lisp では、シンボルを変数として定義したり、関数 (またはマクロ) として定義したり、さらに両方を独立に行なったりできます。

---
[1] 訳注: これはもちろん、印刷されたマニュアルではなく、 Emacs 上でこの章を読んでいる場合の話です。

定義構文要素はふつう、シンボルのある一つの用法における値や意味と、それに加えてその用法におけるその意味に関する説明文も与えます。つまり、あるシンボルを変数として定義した場合には、その変数の初期値と、それに加えてその変数の説明文を与えることができます。

`defvar` と `defconst` は、シンボルをグローバル変数として定義する特殊形式です。それらは 10.5 節「グローバル変数の定義」p.131 に詳しく記述されています。

`defun` は、シンボルを関数として定義し、ラムダ式を作ってシンボルの関数セルに格納します。したがってこのラムダ式がそのシンボルの関数定義となります (“関数定義” という用語は、関数セルの内容ということを意味しますが、これは `defun` がシンボルに対して、関数としての定義を与えるのであるという発想からきたものです)。 `defsubst` と `defalias` は、関数を定義するほかの二とおりのやり方です。第 11 章「関数」p.147 を参照してください。

`defmacro` は、シンボルをマクロとして定義します。これはマクロ・オブジェクトを作成し、それをシンボルの関数セルに格納します。ここで注意していただきたいのは、ある一つのシンボルはマクロまたは関数のどちらか一方にはなることができますが、同時に両方にはなれません。というのも、マクロと関数のいずれもが関数セルに保持され、またどの時点でもセルはただ一つの Lisp オブジェクトしか保持できないからです。第 12 章「マクロ」p.163 を参照してください。

Emacs Lisp において、変数や関数としてシンボルを利用するにあたって、定義は必須のものではありません。このため、シンボルをグローバル変数とすることは `setq` で可能で、これは始めに定義を行なうかどうかには関係しません。定義の本当の目的は、プログラマとプログラミング・ツールをガイドすることです。定義を行なうことにより、コードを読むプログラマに、あるシンボルが変数とか関数として使われることを意図しているということを伝えるわけです。さらに、‘`etags`’ や ‘`make-docfile`’ のようなユーティリティは定義を認識し、適当な情報をタグ・テーブルや ‘`emacs/etc/DOC-`*version*’ ファイルに加えるわけです。 21.2 節「説明文字列へのアクセス」p.340 を参照してください。

7.3 シンボルの作成と intern

GNU Emacs Lisp においてシンボルがどのように作成されるかを理解するには、 Lisp がそれらをどう読み込むのかを理解しなければなりません。 Lisp は、同じひと揃いの文字を読み込んだときには、常に同じシンボルを見つけ出さなければなりません。これに失敗すると、まるっきりの混乱を招くはずです。

Lisp リーダがシンボルに出会うと、名前のすべての文字を読み込みます。次にそれらの文字に対する “ハッシュ” が行なわれ、『obarray』と呼ばれる表に対するインデクスを見つけます。ハッシュというのは何かを検索するのに効率的な手法です。たとえば電話帳で Jan Jones さんを探すときには、端から探してゆく代りにまず J のところを始点にしてそこから探し始めます。これは単純な形のハッシュ法です。 obarray の各要素は『バケツ (bucket)』であり、それはあるハッシュ・コードに対するすべてのシンボルを保持しています。与えられた名前を探すには、その名前のハッシュ・コードに対応するバケツに入っているすべてのシンボルを調べれば充分なのです。

所望の名前をもつシンボルが見つかると、リーダはそのシンボルを使います。 obarray にその名前をもつシンボルが含まれていなければ、リーダは新たにシンボルを作り、 obarray にそれを追加します。ある名前をもつシンボルを検索または追加することを、『intern する (interning)』といい、またそのシンボルは『intern されたシンボル (interned symbol)』と呼ばれます。

intern を行なうことで、それぞれの obarray が、どんな特定の名前に対してもただ一つのシンボルを持つことが保証されます。同じ名前のほかのシンボルも存在しえますが、同じ obarray の中には存在しません。そのため、同じ obarray を使って読込みを続けるかぎり、リーダは同じ名前に対しては同じシンボルを得るのです。

すべてのシンボルを保持する obarray は存在しません。それどころか、どの obarray にも属さないシンボルもあります。それらは、『intern されていないシンボル (uninterned symbols)』と呼ばれます。 intern されていないシンボルでも、ほかのシンボル同様に四つのセルを持っています。しかし、それにアクセスする唯一の方法は、ほかのオブジェクトの中から見つけるか、変数の値として探しだすかの方法しかありません。

Emacs Lisp では、 obarray は実はベクタです。ベクタの各要素はバケツであり、その値は、名前がそのバケツにハッシュされるようなシンボルであるか、あるいはバケツが空ならば 0 になります。 intern されたシンボルはそれぞれバケツ内での次のシンボルへの (ユーザには見えない) 内部リンクを持ちます。これらのリンクが見えないため、ある obarray 内のシンボルをすべて探し出すことは、 `mapatoms`(後述) を使う以外にはできません。バケツ内でのシンボルの順序には意味はありません。

空の obarray では、どの要素も 0 であり、 obarray を `(make-vector` *length* `0)` で作成することができます。これが **obarray** を作成する唯一の正当な方法です。 素数を length に使うと、よいハッシングになる傾向があります。また、 2 のべき乗から 1 を引いた値も length として適しています。

obarray にシンボルを勝手に入れようとしてはいけません。これは動作しません。 `intern` だけが obarry にシンボルを正しく入れることができます。一つのシンボルを二つの **obarry** に **intern** してはなりません。これをやると両方の obarray をグシャグシャにしてしまいます。というのも、シンボルは obarry バケツの中での次のシンボルを入れておくスロットを一つしか持たないためです。このようなことを行なったときの結果は予測できません。

二つの異なるシンボルが、別の obarray 内で同じ名前をもつこともあり得ます。その場合、これらのシンボルは `eq` でも `equal` でもありません。しかし、こうしたことはふつう省略形 (abbrev) のメカニズム (第 32 章「省略形とその展開」 p.553 参照) の一部として起こるだけです。

> **Common Lisp 注意書き:**Common Lisp の場合、一つのシンボルが複数の obarray 相当に intern され得ます。

以下の関数の大部分は引数として名前を取り、場合によってはそれに加えて obarray を取ることもあります。名前が文字列でないか、または obarray がベクタでないと、 `wrong-type-argument` エラーが通知されます。

symbol-name *symbol* Function

この関数は *symbol* の名前である文字列を返します。たとえば:

```
(symbol-name 'foo)
     ⇒ "foo"
```

文字を入れ換えてこの文字列を変更するとか、いろいろやればシンボルの名前を変えることになりますが、 obarray を更新できませんから、これはやらないでください。

make-symbol *name* Function

この関数は、新たに割り付けられた、 *name*(これは文字列であること) という名前の intern されていないシンボルを返します。その値と関数定義は存在せず (void であり)、その属性リストは `nil` です。以下の例において、 `sym` の値は `foo` と `eq` にはなりません。というのも、それは同じく '`foo`' という名前をもつ、 intern されていない別のシンボルになるためです。

```
(setq sym (make-symbol "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ nil
```

intern *name* &optional *obarray* Function

この関数は、名前が *name* である intern されたシンボルを返します。 obarray *obarray* にそのようなシンボルがない場合、 `intern` は新しくそれを作り、それをその obarray に加えたうえで返します。 *obarray* が省略された場合には、グローバル変数 `obarray` の値が用いられます。

```
(setq sym (intern "foo"))
     ⇒ foo
(eq sym 'foo)
     ⇒ t

(setq sym1 (intern "foo" other-obarray))
     ⇒ foo
(eq sym 'foo)
     ⇒ nil
```

intern-soft *name* &optional *obarray* Function

この関数は、 *obarray* の中にある、名前が *name* であるシンボルを返し、またもし *obarray* にそのような名前のシンボルがなければ `nil` を返します。したがって、 `intern-soft` を、与えられた名前をもつシンボルがすでに intern されているかどうかを調べるために用いることができます。 *obarray* が省略された場合には、グローバル変数 `obarray` の値が用いられます。

```
(intern-soft "frazzle")        ; そのようなシンボルは存在しない。
     ⇒ nil
(make-symbol "frazzle")        ; intern されていないものを作る。
     ⇒ frazzle
(intern-soft "frazzle")        ; これは見つけ出せない。
     ⇒ nil
(setq sym (intern "frazzle"))  ; intern されたものを作る。
     ⇒ frazzle
```

```
(intern-soft "frazzle")        ; これは見つけ出せる！
     ⇒ frazzle
(eq sym 'frazzle)              ; で、同じものになる。
     ⇒ t
```

obarray Variable

この変数は、 `intern` と `read` で用いられる標準の obarray です。

mapatoms *function* &optional *obarray* Function

この関数は、 obarray *obarray* にある各シンボルそれぞれに対して *function* を呼び出します。戻り値は `nil` です。 *obarray* が省略されると、デフォルト値として普通のシンボル用の標準の obarray である `obarray` が使われます。

```
(setq count 0)
     ⇒ 0
(defun count-syms (s)
  (setq count (1+ count)))
     ⇒ count-syms
(mapatoms 'count-syms)
     ⇒ nil
count
     ⇒ 1871
```

`mapatoms` を用いた別の例は、 21.2 節「説明文字列へのアクセス」 p.340 にある `documentation` を御覧ください。

unintern *symbol* &optional *obarray* Function

この関数は *symbol* を obarry *obarry* から消去します。もしも `symbol` が実際には obarry に入っていなかった場合、 `unintern` はなにもしません。 *obarray* が `nil` の場合には現在の obarray が使われます。

symbol にシンボルの代わりに文字列が与えられた場合、これはシンボル名であるとみなされます。そのうえで `unintern` は、その名前を保持している obarray から (もしあれば) そのシンボルを消去します。そのようなシンボルがないならば、 `unintern` はなにもしません。

`unintern` はシンボルの消去を行なった場合には `t` を返します。そうでなければ `nil` が返されます。

7.4 属性リスト

『属性リスト (property list)』 (縮めて『plist』) はシンボルの属性リスト・セルに格納された、要素を二つ単位でペアにしたリストのことです。それぞれのペアは、属性の名前 (通常はシンボル) と属性、いいかえれば値、を結び付けます。属性リストは、シンボルに関する情報を記録しておくために広く用いられ、たとえば変数としての説明やそのシンボルが定義されたファイル名、さらにはひょっとすると言語解釈システムの中で (単語を表している) シンボルの品詞を示すためにさえ利用されたりします。

文字列やバッファ内での文字の位置情報もまた属性リストをもつことが可能です。 29.18 節「テキスト属性」 p.509 を参照してください。

属性リストにおける属性の名前と値は、どんな Lisp オブジェクトであっても構わないのですが、名前はふつうシンボルです。それらは、 `eq` を使って比較されます。以下に属性リストの例がありますが、これはコンパイラがロードされたときに `progn` というシンボルにおいて見られるものです:

```
(lisp-indent-function 0 byte-compile byte-compile-progn)
```

ここで `lisp-indent-function` と `byte-compile` が属性名であり、そのほかの要素はそれぞれに対応する値です。

7.4.1 属性リストと連想リスト

連想リスト (5.8 節「連想リスト」 p.81 参照) は属性リストととても似通っています。連想リストとは対照的に、属性の名前は一意でなければならないため、属性リスト内でのペアの順序は重要ではありません。

さまざまな Lisp の関数名や変数に情報を付加するためには、属性リストの方が連想リストよりも優れています。もしも全ての連想情報が一つの連想リストに記録されていた場合、関数や変数が操作されようとする度に、そのプログラムはそのリスト全体を検索しなければなりません。対照的に、もしその情報がその関数名や変数そのものの属性リストに記録されていれば、一回の検索は一つの属性リストの長さだけスキャンするだけですし、通常その長さは短いのです。これが、変数に関する説明が `variable-documentation` という名前の属性リストに記録されていることの理由です。同様に、バイトコンパイラは特殊な取り扱いを必要とするような関数を記録するために属性リストを使います。

とはいうものの、連想リストにもそれなりの利点があります。アプリケーションによっては、連想リストの先頭に連想情報を追加する方が、属性を更新するよりも速い場合があります。一つのシンボルに対する全ての属性は同一の属性リストに格納されますから、一つの属性名に対して異なる用法がぶつかる可能性があります (このため、おそらく一意であるような属性名、すなわち属性名にライブラリの名前を含ませたようなものを選ぶのがよいでしょう)。連想リストは、連想情報がリストの先頭にプッシュされ、後で捨てられるスタックのようにして使われ得るものです。これは属性リストでは不可能です。

7.4.2 シンボルの属性リスト関数

symbol-plist *symbol* Function

この関数は *symbol* の属性リストを返します。

setplist *symbol plist* Function

この関数は *symbol* の属性リストを *plist* とします。通常、 *plist* はきちんとした形式の属性リストであるべきですが、これは強制はされません。

```
(setplist 'foo '(a 1 b (2 3) c nil))
     ⇒ (a 1 b (2 3) c nil)
(symbol-plist 'foo)
     ⇒ (a 1 b (2 3) c nil)
```

特殊な obarray で、それが普通の用途に使われないような場合、その中に置かれたシンボルに関しては、おかしな格好をした属性リスト・セルを用いることが意味を持つ場合もあるかも知れません。 実際、省略形 (abbrev) の機構ではそうしています (第 32 章「省略形とその展開」p.553 参照)。

get *symbol property* Function

この関数は *symbol* の属性リストの中で *property* という名前の属性の値を見つけ出します。もしもそうした属性がない場合には、 `nil` が返されます。したがって、 `nil` という値がある場合と属性が存在しない場合の区別はつきません。

property で与えられる名前は、存在している属性名と `eq` を使って比較されますから、どのようなオブジェクトでも合法的な属性になり得ます。

例として `put` を御覧ください。

put *symbol property value* Function

この関数は *value* を *symbol* の属性リストの中にある *property* という名前の属性に入れ、以前の属性値を置き換えます。 `put` 関数は *value* を返します。

```
(put 'fly 'verb 'transitive)
     ⇒'transitive
(put 'fly 'noun '(a buzzing little bug))
     ⇒ (a buzzing little bug)
(get 'fly 'verb)
     ⇒ transitive
(symbol-plist 'fly)
     ⇒ (verb transitive noun (a buzzing little bug))
```

7.4.3 シンボルの外での属性リスト

以下の二つの関数はシンボル以外の場所に格納された属性リストを扱うのに便利です:

plist-get *plist property* Function

これは属性リスト *plist* に格納されている属性 *property* の値を返します。たとえば、

```
(plist-get '(foo 4) 'foo)
     ⇒ 4
```

plist-put *plist property value* Function

これは属性リスト *plist* に、属性 *property* の値として *value* を格納します。これは *plist* を破壊的に修正するかも知れませんし、あるいは古いものを変更することなく新しいリスト構造を作成するかも知れません。この関数は変更された属性リストを返すので、 *plist* を得た場所にそれを書き込み戻すことができます。たとえば、

```
(setq my-plist '(bar t foo 4))
     ⇒ (bar t foo 4)
(setq my-plist (plist-put my-plist 'foo 69))
     ⇒ (bar t foo 69)
(setq my-plist (plist-put my-plist 'quux '(a)))
     ⇒ (quux (a) bar t foo 5)
```

8 評価

Emacs Lisp 式の『評価 (evaluation)』は、『Lisp インタプリタ (Lisp interpreter)』が行ないます。 Lisp インタプリタは、 Lisp オブジェクトを入力として受け取り、『式としての値 (value as an expression)』を計算するプログラムです。評価方法はオブジェクトのデータ型に依存します。本章に述べる規則にしたがいます。インタプリタはプログラムの一部分を評価するために自動的に実行されますが、 Lisp プリミティブ `eval` を明示的に呼び出すことでも実行できます。

評価を目的とする Lisp オブジェクトを『式 (expression)』あるいは『形式 (form)』と呼びます。式がデータ・オブジェクトであって、単なるテキストではないということは、 Lisp のような言語と典型的プログラミング言語との間の根本的な違いの一つです。どんな Lisp オブジェクトも評価できますが、実際には数、シンボル、リスト、文字列だけが非常に頻繁に評価されます。

Lisp 式を読み込み、その後その式を評価するということは、ごく普通に行なわれますが、読込みと評価は別の行為であり、いずれも単独で行なうことができます。読込みそれ自体は何も評価しません。それは、 Lisp オブジェクトの印字表現をオブジェクト自身に変換するものです。このオブジェクトが評価すべき形式であるかどうか、あるいは全く異なる目的に使うものなのかどうかということは、 `read` の呼び手次第です。 16.3 節「入力関数」 p.230 を参照してください。

評価とキー解釈とを混同しないでください。エディタのコマンド・ループは、アクティブなキーマップを使ってキーボード入力をコマンド (対話的に呼び出せる関数) に変換し、その後 `call-interactively` を使ってそのコマンドを起動します。コマンドが Lisp で書かれていたならばコマンドの実行自身が評価を含みますが、それはコマンドのキー解釈自身の一部ではありません。第 18 章「コマンド・ループ」 p.257 を参照してください。

評価は再帰的処理です。いいかえると、形式の評価は形式の一部分を評価するために `eval` を呼び出すかもしれないということです。たとえば、関数呼び出の評価では最初に関数呼出しの各引数の評価を行ない、その後で関数本体中の各形式を評価します。形式 `(car x)` の評価を考えてみると、部分形式 `x` を最初に再帰的に評価しなければ、その値を関数 `car` の引数として渡すことができません。

関数呼出しの評価は、最終的には、そこで指定された関数を呼び出します。第 11 章「関数」 p.147 を参照してください。関数の実行は、それ自体がその関数定義の評価によるものかも知れません。あるいは、関数は C で実装された Lisp のプリミティブかも知れませんし、バイトコンパイルされた関数 (第 14 章「バイトコンパイル」 p.181 参照) かも知れません。

形式の評価は、全ての Lisp 変数の現在の値と現在の束縛からなる『環境 (environment)』と呼ばれる状況の中で行なわれます。[1] その変数に新しい束縛を与える場合を除いて、形式が変数を参照するときは常に現在の環境中で束縛された値が使われます。第 10 章「変数」 p.127 を参照してください。

形式の評価は、変数の束縛によって、再帰的評価のための新たな環境を作ることがあります (10.3 節「ローカル変数」 p.128 参照)。これら環境は一時的なもので、形式の評価完了時には消

[1] この「環境」の定義はプログラムの結果に影響するような全てのデータを含むことを明確に意図してはいません。

えてしまいます。形式は永続的な変更を作ることもあります。このような変更を『副作用 (side effect)』と呼びます。副作用を起こす形式の例は、`(setq foo 1)` です。

各種の形式に対する評価が何であるかということの詳細を以下に説明します (8.2 節「形式の種類」p.104 参照)。

8.1 Eval

ほとんどの場合、実行中のプログラムの現れた場所で、形式は自動的に評価されます。まれに、実行時に計算される形式を評価するコードを書く必要に迫られるかも知れません。編集中のテキストから形式を読み取った後や、属性リストから形式を取り出した後のような場合です。このような場合に『eval』関数を使います。

注意 **!:** 一般的には、データ構造に格納された式を評価するよりも、データ構造に格納された関数を呼び出す方が、きれいかつ柔軟です。関数を使うと、情報を引数として渡すことができます。

この節で説明する関数と変数には、形式を評価するもの、評価処理の制限を指定するもの、最近返された値を記録するものがあります。ファイルをロードすると、やはり評価を行ないます (第 13 章「ロード」p.171 参照)。

eval *form* Function

評価を行なう基本の関数です。現在の環境の *form* を評価し、結果を返します。評価がどのように処理されるかは、オブジェクトの型に依存します (8.2 節「形式の種類」p.104 参照)。

`eval` は関数なので、 `eval` 呼出しに現れる引数の式は 2 回評価されます。そのうち 1 回は `eval` 呼出し前の準備として、もう 1 回は `eval` 関数自身によって行なわれます。例を挙げます:

```
(setq foo 'bar)
     ⇒ bar
(setq bar 'baz)
     ⇒ baz
;; eval は引数 bar(foo の値である) を受取る。
(eval foo)
     ⇒ baz
(eval 'foo)
     ⇒ bar
```

`eval` 呼出しの、ある時点での有効回数は、 `max-lisp-eval-depth`(以下参照) によって制限されます。

eval-region *start end* &optional *stream* Command

この関数は、カレント・バッファの位置 *start* と *end* の範囲で定められるリージョン内の全形式を評価します。リージョンから形式を読み込んで、その形式に対して `eval` を呼び出すことを、リージョンの終わりに達するかハンドルされないエラーが通知されるまで続けます。

stream を与えると、評価中は `standard-output` を *stream* に束縛します。

変数 `load-read-function` を使うと、 `eval-region` が式を読み込むときに `read` の代わりに使う関数を指定することもできます。 13.1 節「プログラムでロードする方法」p.171 を参照してください。

`eval-region` は常に `nil` を返します。

eval-current-buffer &optional *stream* — Command

この関数は、バッファ全体に対して実行することを除けば `eval-region` と同じです。

max-lisp-eval-depth — Variable

この変数は、 `eval`, `apply`, `funcall` 呼出しにおいて許されるネストの深さの最大値を定義します。これを超えると、 (エラー・メッセージ `"Lisp nesting exceeds max-lisp-eval-depth"`をともなう) エラーが通知されます。この数には、これらの関数の内部的使用や、関数呼出しの引数や関数本体の形式の再帰的評価も勘定に入れます。前者には、Lisp 式の形で書かれた関数の呼出しも含まれます。

この制限は、制限を超えたときにそれを表すエラーを出力することで、間違って定義された関数の無限再帰を Lisp で回避するための一つの方法です。

この変数のデフォルト値は 200 です。設定した値が 100 未満のときにその値に達した場合は、 Lisp は値を 100 に再設定します。

`max-specpdl-size` が別のネストの制限を与えます。 10.3 節「ローカル変数」p.128 を参照してください。

values — Variable

この変数の値は、 (ミニバッファを含む) バッファから読み込まれ、評価され、印字された全ての式が返した値のリストです。その要素は最近のものほど前に来るように並んでいます。

```
(setq x 1)
     ⇒ 1
(list 'A (1+ 2) auto-save-default)
     ⇒ (A 3 t)
values
     ⇒ ((A 3 t) 1 ...)
```

この変数は、最近評価された式の値を振り返って参照するのに便利です。 `values` 自身の値を印字するのは一般的にはよくないことです。なぜなら結果は非常に長くなり得るからです。代わりに、このように特定の要素を調べるとよいでしょう:

```
;; 一番新しい評価結果を参照する。
(nth 0 values)
     ⇒ (A 3 t)
;; これで新しい要素を一つ追加したことになるので、
;;    全ての要素は一つ後ろに動く。
(nth 1 values)
     ⇒ (A 3 t)
```

```
;; この例の実行前には 2 番目に新しかった
;;    要素を取り出す。
(nth 3 values)
     ⇒ 1
```

8.2 形式の種類

評価されることを目的とする Lisp オブジェクトを『形式 (form)』と呼びます。 Emacs がどのように形式を評価するかは、そのデータ型に依存します。 Emacs は評価方法の違いで形式を 3 種類に分けます。シンボル、リスト、および「ほかの全ての型」です。この節では、自己評価型形式である「ほかの全ての型」から始めて、これら 3 種すべてについて説明します。

8.2.1 自己評価型形式

『自己評価型形式 (self-evaluating form)』とは、リストやシンボルではない形式のことをいいます。自己評価型形式を評価するとそれら自身になります。その評価結果は評価したオブジェクトと同一になります。それゆえ、数 25 は 25 と評価され、文字列 `"foo"`は文字列 `"foo"`と評価されます。同様に、ベクタの評価ではベクタ要素の評価は起こりません。内容の変わらない同じベクタが返ります。

```
'123                 ; オブジェクトを評価抜きで見る。
     ⇒ 123
123                  ; 普通に評価する (結果は同じ)。
     ⇒ 123
(eval '123)          ; 「手作業で」評価する — 結果は同じ。
     ⇒ 123
(eval (eval '123))   ; 2 回評価しても何も変わらない。
     ⇒ 123
```

数、文字、文字列、ベクタさえもこれらが自己評価型である利点を活かして、 Lisp コードに当たり前に書いています。けれどもリード構文がない型に対してこのように書くということは普通は行ないません。なぜならそういったものをテキストで書く方法がないためです。こういった型を含む Lisp 式を Lisp プログラムという手段によって構築することは可能です。例を挙げます:

```
;; バッファ・オブジェクトを含む式を作る。
(setq buffer (list 'print (current-buffer)))
     ⇒ (print #<buffer eval.texi>)
;; それを評価する。
(eval buffer)
     ⊣ #<buffer eval.texi>
     ⇒ #<buffer eval.texi>
```

8.2.2 シンボル形式

シンボルが評価されるときは、変数として扱われます。結果は、あるとすれば変数の値です。値がない場合 (その値セルが void の場合) は、エラーが通知されます。変数の使い方についてこれ以上の話は、第 10 章「変数」p.127 を参照してください。

以下の例では、シンボルの値を setq で設定します。その後シンボルを評価し、 setq が格納した値を取り戻します。

```
(setq a 123)
     ⇒ 123
(eval 'a)
     ⇒ 123
a
     ⇒ 123
```

nil と t というシンボルは、特別な扱いを受けます。その結果、 nil の値は常に nil であり、 t の値は常に t になります。これらをほかの値に設定することも束縛することもできません。したがって、たとえ eval がこれらをほかのシンボルのように扱ったとしても、この二つのシンボルは自己評価型形式のように振舞います。

8.2.3 リスト形式の分類

リストという形式は、それが空でないならば、その最初の要素によって、関数呼出しか、マクロ呼出しか、特殊形式かのいずれかになります。これら 3 種の形式は、以下に述べるように異なる方法で評価されます。リストの残りの要素は、関数、マクロ、特殊形式いずれかの『引数 (argument)』を構成します。

空ではないリストの評価の最初の手順は、その第 1 要素の検査です。この要素だけが、そのリストがどの種類の形式であるのかということと、リストの残りがどのように処理されるかということを決定します。 Lisp の方言によっては、たとえば Scheme では評価するのですが、 Emacs Lisp では第 1 要素は評価しません。

8.2.4 シンボル関数間接参照

リストの第 1 要素がシンボルならば、評価においてはそのシンボルの関数セルを検査し、その中身を元のシンボルの代わりに使用します。その中身がほかのシンボルならば、この『シンボル関数間接参照 (symbol function indirection)』と呼ぶ処理を非シンボルを得るまで繰り返します。シンボルの関数セルに格納された関数の名前として、そのシンボルを使うことについてのこれ以上の話は、 11.3 節「関数の命名」p.152 を参照してください。

この処理の結果として無限ループになることもあります。それは、シンボルの関数セルが同じシンボルを参照しているときに起こります。あるいは、シンボルが void な関数セルを持ち、そのためにサブルーチン symbol-function が void-function エラーを通知するかも知れません。このいずれも起こらない場合は、最終的には非シンボルを得ます。その非シンボルは関数またはほかの適当なオブジェクトになっているはずです。

もっと精確に書くと、 Lisp 関数 (ラムダ式)、バイトコード関数、プリミティブ関数、 Lisp マクロ、特殊形式、自動ロード・オブジェクトのいずれかになっていないとおかしいでしょう。これらの型のそれぞれは以下のどれかの節で説明します。オブジェクトがこれら型のいずれでもない場合、 `invalid-function` エラーが通知されます。

以下の例は、シンボル間接参照処理を図解しています。 `fset` を使ってシンボルの関数セルを設定し、 `symbol-function` で先の関数セルの内容を取り出しています (11.8 節「関数セルの内容へのアクセス」 p.158 参照)。具体的には、シンボル `car` を `first` の関数セルに格納し、シンボル `first` を `erste` の関数セルに格納しています。

```
;; この関数セルのリンクを作る:
;;   -------------       -----        -------        -------
;;  | #<subr car> | <-- | car |  <-- | first |  <-- | erste |
;;   -------------       -----        -------        -------

(symbol-function 'car)
     ⇒ #<subr car>
(fset 'first 'car)
     ⇒ car
(fset 'erste 'first)
     ⇒ first
(erste '(1 2 3))   ; erste が参照する関数を呼び出す。
     ⇒ 1
```

対照的に、次の例では関数をシンボル関数間接参照なしで呼び出しています。というのも、最初の要素が匿名 Lisp 関数であり、シンボルではないからです。

```
((lambda (arg) (erste arg))
 '(1 2 3))
     ⇒ 1
```

この関数自身の実行がこの本体を評価します。これには、 `erste` を呼び出す時のシンボル関数間接参照がともないます。

組込み関数 `indirect-function` は、明示的に関数のシンボル関数間接参照を簡単に行なう方法を提供しています。

indirect-function *function* Function

この関数は、 *function* の関数としての意味を返します。 *function* がシンボルならば、 *function* の関数定義を見つけ、その値を用いて最初からやり直します。 *function* がシンボルではないならば、 *function* 自身を返します。

```
(defun indirect-function (function)
  (if (symbolp function)
      (indirect-function (symbol-function function))
    function))
```

8.2.5 関数形式の評価

評価しているリストの第 1 要素が Lisp 関数オブジェクトの場合、バイトコード・オブジェクトの場合、あるいはプリミティブ関数オブジェクトの場合は、そのリストは『関数呼出し (function call)』です。たとえば、これは関数 + の呼出しです:

```
(+ 1 x)
```

関数呼出しを評価する第 1 手順は、残りのリスト要素を左から右に評価することです。結果は実引数の値であり、一つの値がリスト要素一つに対応します。次の手順はこの引数のリストを持って関数を呼び出すことで、事実上関数 apply を使用します (11.5 節「関数の呼出し」p.154 参照)。この関数が Lisp で書かれていたならば、実引数で関数の仮引数を束縛します (11.2 節「ラムダ式」p.148 参照)。その後関数本体内の形式が順序どおりに評価され、最後の本体形式が関数呼出しの値になります。

8.2.6 Lisp マクロの評価

評価中のリストの第 1 要素がマクロ・オブジェクトならば、リストは『マクロ呼出し (macro call)』です。マクロ呼出しが評価されると、リストの残りの要素ははじめは評価されません。代わりに、これら要素自身がマクロの引数として使われます。マクロ定義はマクロの『展開 (expansion)』と呼ばれる置換形式を計算し、これが元の形式の代わりに評価されます。展開はあらゆる種類の形式となり得ます。自己評価型定数、シンボル、リストです。展開自身がマクロ呼出しならば、この展開処理をほかの種類の形式が結果となるまで繰り返します。

マクロ呼出しの評価は、普通はその展開を評価することで終了します。しかしマクロ展開をすぐに評価する必要はありませんし、全く評価しなくても構いません。なぜならほかのプログラムも同じようにマクロ呼出しを展開し、それらもその展開を評価することもしないこともあるからです。

普通、引数に現れる式は、マクロ展開の計算の一部としては評価されず、代わりにその展開の一部として現れ、その結果、それらはその展開を計算するときに計算されます。

たとえば、次のようにマクロを定義します:

```
(defmacro cadr (x)
  (list 'car (list 'cdr x)))
```

(cadr (assq 'handler list)) のような式は、マクロ呼出しで、その展開はこうなります:

```
(car (cdr (assq 'handler list)))
```

引数の (assq 'handler list) が展開の中に現れていることに注意しましょう。

Emacs Lisp マクロの完全な説明は、第 12 章「マクロ」p.163 を参照してください。

8.2.7 特殊形式

『特殊形式 (special form)』とはその引数の全てが評価されないように設計された関数です。特殊形式のほとんどが、制御構造を定義したり変数の束縛、関数がやれないこと、を行ないます。

特殊形式のそれぞれが固有の規則を持っていて、どの引数を評価し、どの引数は評価しないで使うかを決めます。特定の引数が評価されるか否かが、ほかの引数を評価した結果に依存することもあります。

`and` 9.3 節「条件を結合するための構文要素」p.114 参照

`catch` 9.5.1 節「明示的な非局所脱出: catch と throw」p.116 参照

`cond` 9.2 節「条件分岐」p.112 参照

`condition-case`
9.5.3.3 節「エラーをハンドルするコードの記述」p.121 参照

`defconst` 10.5 節「グローバル変数の定義」p.131 参照

`defmacro` 12.4 節「マクロの定義」p.165 参照

`defun` 11.4 節「関数の定義方法」p.152 参照

`defvar` 10.5 節「グローバル変数の定義」p.131 参照

`function` 11.7 節「匿名関数」p.156 参照

`if` 9.2 節「条件分岐」p.112 参照

`interactive`
18.3 節「対話的呼出し」p.262 参照

`let`
`let*` 10.3 節「ローカル変数」p.128 参照

`or` 9.3 節「条件を結合するための構文要素」p.114 参照

`prog1`
`prog2`
`progn` 9.1 節「順次実行」p.111 参照

`quote` 8.3 節「quote する」p.109 参照

`save-excursion`
27.3 節「脱線」p.466 参照

`save-restriction`
27.4 節「ナローイング」p.467 参照

`save-window-excursion`
25.16 節「ウィンドウ構成」p.432 参照

`setq` 10.7 節「変数の値を変更する方法」p.135 参照

`setq-default`
10.9.2 節「バッファローカルな束縛の作成と削除」p.141 参照

`track-mouse`
26.13 節「マウスの追跡」p.449 参照

`unwind-protect`
9.5 節「非局所脱出」p.116 参照

`while` 9.4 節「繰り返し」p.115 参照

`with-output-to-temp-buffer`
35.8 節「一時表示」p.608 参照

> **Common Lisp 注意書き:** GNU Emacs Lisp と Common Lisp との特殊形式の比較をします。 `setq`, `if`, `catch` は、 Emacs Lisp と Common Lisp との双方において特殊形式です。 `defun` は Emacs Lisp では特殊形式ですが、 Common Lisp ではマクロです。 `save-excursion` は Emacs Lisp においては特殊形式ですが、 Common Lisp には存在しません。 `throw` は Common Lisp では特殊形式 (複数の値を throw できるようにするため) ですが、 Emacs Lisp では関数です (複数の値は持たない)。

8.2.8 自動ロード

『自動ロード (autoload)』機能によって、まだ Emacs にその定義がロードされてないような関数やマクロを呼び出すことができます。これによって、どのファイルがその定義を含むかを指定します。自動ロード・オブジェクトがシンボルの関数定義として現れたとき、そのシンボルを関数として呼び出すことで自動的に指定されたファイルがロードされます。その後、ファイルからロードされた実際の定義を呼び出します。 13.2 節「自動ロード」p.173 を参照してください。

8.3 quote する

特殊形式 `quote` は、一つの引数を、書かれたとおり、評価をせずに返します。これによって、定数シンボルやリストなど、自己評価型ではないオブジェクトをプログラムに含める手段が用意されています (数、文字列、ベクタなど、自己評価型オブジェクトを quote する必要はありません)。

quote *object* Special Form

この特殊形式は *object* を評価せずに返します。

`quote` はしょっちゅうプログラムで使われるので、 Lisp はそれを便利にするリード構文を用意しています。アポストロフィ文字 (‘’’) に続く一つの Lisp オブジェクト (リード構文において) は、第 1 要素が `quote` で、 2 番目の要素が先のオブジェクトというリストに展開されます。したがって、リード構文`'x` は、`(quote x)` の省略形です。

`quote` を使う式の例を挙げます:

```
(quote (+ 1 2))
     ⇒ (+ 1 2)
(quote foo)
     ⇒ foo
'foo
     ⇒ foo
''foo
     ⇒ (quote foo)
'(quote foo)
     ⇒ (quote foo)
['foo]
     ⇒ [(quote foo)]
```

そのほかの quote を行なう構文要素には、 Lisp で書かれた匿名ラムダ式がコンパイルされるようになる `function`(11.7 節「匿名関数」 p.156 参照)、ほかの部分の計算や置換をする間、リストの一部だけを quote するために使用する ‘(12.5 節「逆引用符」 p.165 参照) があります。

9 制御構造

Lisp プログラムは式または『形式 (form)』(8.2 節「形式の種類」p.104 参照) から成っています。私たちは、それらを『制御構造 (control structure)』で囲むことによって、形式の実行の順番を制御することができます。制御構造とは、囲んでいる形式を、いつ実行するか、実行するかしないか、何回実行するかを制御する特殊形式のことです。

最も単純な実行順序は順次実行です。最初に形式 *a*、次に形式 *b* というように実行します。これは関数の本体、または Lisp コードのファイルの最上位レベルに、複数の形式を連続して記述したときの実行であって、それらの形式は記述された順番に実行されます。私たちはこれを『テキストどおりの順序 (textual order)』と呼んでいます。例えば、もし関数の本体が二つの形式 *a* と *b* から成っているなら、関数は最初に *a*、次に *b* を評価し、 *b* の値が関数の値になります。

明示的な制御構造は順次以外の実行順序を可能にします。

Emacs Lisp は、順次実行の変種、条件分岐、繰返し、および (制御された) ジャンプを含んだ数種類の制御構造を提供していて、そのすべてを以下に説明します。組込みの制御構造は、その部分形式が必ずしも評価されるわけではなく、また、順番に評価されるわけでもないので、特殊形式になっているのです。マクロを使って、独自の制御構造を定義することができます (第 12 章「マクロ」p.163 参照)。

9.1 順次実行

形式を出現順に評価することは、ある形式から他の形式へ制御を移す方法のなかで、もっともよく使われるものです。関数本体のような、いくつかの文脈においては自動的にこのようになります。その他の場所で、これを行うには制御構造の構文要素を使用しなければなりません。`progn` は Lisp の最も簡単な制御の構文要素です。

特殊形式 `progn` は以下のようなものです。

```
(progn a b c ...)
```

これは形式 *a*, *b*, *c* などを、その順番で実行することを意味しています。これらの形式は `progn` 形式の本体と呼ばれます。本体の最後の形式の値が `progn` 全体の値になります。

初期の Lisp では、 `progn` は、連続した二つ以上の形式を実行し、その最後の値を使用するための唯一の方法でした。しかし、プログラマは、 (その時点では) ただ一つの形式だけが許されていた関数の本体にも、 `progn` を使用することがしばしば必要であることに気づきました。そこで、関数の本体に「暗黙の `progn`」を作りました。その中の形式は実際の `progn` の本体と同様に扱われます。他の多くの制御構造も同様に暗黙の `progn` を含んでいます。結果として、`progn` は本来必要であるほどには使用されません。それは `unwind-protect`, `and`, `or`, または `if` の *then* 部分の中で最もよく使用されます。

progn *forms...* Special Form

この特殊形式は *forms* のすべてをテキストどおりの順序で評価し、最後の形式の結果を返します。

```
(progn (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The third form"
```

以下の二つの制御構造も同様に形式の並びを評価しますが、異なった値を返します。

prog1 *form1 forms...* Special Form

この特殊形式は、 *form1* と *forms* のすべてをテキストどおりの順序で評価し、 *form1* の結果を返します。

```
(prog1 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The first form"
```

次の例では変数 `x` に設定されたリストから先頭の要素を削除し、その削除した要素の値を返しています。

```
(prog1 (car x) (setq x (cdr x)))
```

prog2 *form1 form2 forms...* Special Form

この特殊形式は、 *form1, form2,* そしてその後の *forms* のすべてをテキストどおりの順序で評価し、 *form2* の結果を返します。

```
(prog2 (print "The first form")
       (print "The second form")
       (print "The third form"))
     ⊣ "The first form"
     ⊣ "The second form"
     ⊣ "The third form"
⇒ "The second form"
```

9.2 条件分岐

条件分岐の制御構造は選択肢の中から選び出すことをします。 Emacs Lisp は、二つの条件分岐の形式を持っています。他の言語とほとんど同じである `if` と、一般化された case 文である `cond` です。

if *condition then-form else-forms...* Special Form

`if` は *condition* の値を元に *then-form* か *else-forms* かを選択します。評価された *condition* が `nil` でないならば、 *then-form* が評価され、その結果が返されます。そうでなけ

れば、 *else-forms* がテキストどおりの順序で評価され、最後のものの値が返されます。(`if` の *else* 部は暗黙の `progn` の例です。 9.1 節「順次実行」 p.111 参照)

condition が値 `nil` を持ち、 *else-forms* が与えられていなければ、 `if` は `nil` を返します。

`if` は選択されない枝はまったく評価されない (無視される) ので、特殊形式になっています。したがって、以下の例で `print` は決して呼ばれないので `true` は印字されません。

```
(if nil
    (print 'true)
  'very-false)
⇒ very-false
```

cond *clause...* Special Form

`cond` は任意個の選択肢から選択します。 `cond` の個々の *clause* はリストでなければなりません。このリストの CAR は *condition* です。残りの要素は (もしあれば)*body-forms* です。したがって句 (clause) は次のようになります。

(*condition body-forms...*)

`cond` は個々の句の *condition* を評価していくことによって、その句の並びをテキストどおりの順序で試します。 *condition* の値が `nil` でなければ、その句は「成功」します。その場合、 `cond` はその *body-forms* を評価し、 *body-forms* の最後の値が `cond` の値となります。残りの句は無視されます。

condition の値が `nil` ならば、その句は「失敗」し、 `cond` は次の句に制御を移し、その *condition* を試していきます。

すべての *condition* が `nil` に評価された場合、すなわち、すべての句が失敗した場合、 `cond` は `nil` を返します。

句は次のようになっているかもしれません。

(*condition*)

その場合、 *condition* のテスト結果が `nil` でなければ、 *condition* の値が `cond` 形式の値になります。

以下の例では四つの句を持っていて、それぞれ `x` の値が数値、文字列、バッファ、シンボルのどれかであるかをテストします。

```
(cond ((numberp x) x)
      ((stringp x) x)
      ((bufferp x)
       (setq temporary-hack x) ; 一つの句に
       (buffer-name x))        ; 複数の body-forms
      ((symbolp x) (symbol-value x)))
```

最後の句より前の句がどれも成功しなかった場合には、その最後の句を実行したいということがよくあります。これを行うために私たちは次のように最後の句の *condition* として `t` を使用します。 (`t` *body-forms*)。形式 `t` は `t` に評価され、決して `nil` にはならず、したがって、この句は決して失敗せず、最悪でもそこには到達する `cond` が提供されます。

例えば、

```
(cond ((eq a 'hack) 'foo)
      (t "default"))
⇒ "default"
```

この式は a の値が hack ならば foo を返し、そうでなければ文字列 "default"を返す cond です。

あらゆる条件分岐の構造は cond でも if でも表現することができます。したがって、どちらを使うかの選択はスタイルの問題です。例えば:

(if *a* *b* *c*)
≡
(cond (*a* *b*) (t *c*))

9.3 条件を結合するための構文要素

このセクションでは複雑な条件を表現するために if や cond とともによく使用される三つの構文要素を説明します。構文要素 and と or は、ある種の複合した条件を組み合わせるものとして別個に使用されることもあります。

not *condition* Function

この関数は *condition* が偽であるかをテストします。 *condition* が nil であれば t を返し、そうでなければ nil を返します。関数 not は null と同一ですが、空リストのテストには null の方を使用することを私たちは勧めます。

and *conditions...* Special Form

特殊形式 and は、すべての *conditions* が真であるかをテストします。それは *conditions* を書かれた順序で一つずつ評価していくことによって行なわれます。

conditions のうちのどれかが nil に評価されれば、残りの *conditions* がどうであろうとも and の結果は nil になるはずです。だから、 and は残りの *conditions* を無視して、ただちに帰ります。

すべての *conditions* が nil でなければ、その最後のものの値が形式 and の値になります。

以下に例をあげます。最初の条件は整数 1(nil ではない) を返します。同様に、 2 番目の条件は整数 2(nil ではない) を返します。 3 番目の条件は nil であり、残りの条件は決して評価されません。

```
(and (print 1) (print 2) nil (print 3))
     ⊣ 1
     ⊣ 2
⇒ nil
```

以下は and を使用した、より現実的な例です。

```
(if (and (consp foo) (eq (car foo) 'x))
    (message "foo は x で始まるリスト"))
```

(consp foo) が nil を返すのであれば、(car foo) は実行されないことに注意してください。これによってエラーが回避されます。

and は、 if または cond で表現することができます。例えば:

```
(and arg1 arg2 arg3)
≡
(if arg1 (if arg2 arg3))
≡
(cond (arg1 (cond (arg2 arg3))))
```

or *conditions...* Special Form

特殊形式 or は少なくとも *conditions* の一つが真であるかをテストします。それは *conditions* を書かれた順序で一つずつ評価していくことによって行なわれます。

conditions のうちのどれかが nil でないものに評価されれば、 or の結果は nil 以外になるはずです。それなので、 or は残りの *conditions* を無視して、ただちに帰ります。その返す値は返却の直前に評価された条件の nil でない値です。

すべての *conditions* が nil であれば、 or は nil を返します。

例えば、以下の式は x が 0 または nil のどちらかであるかをテストします。

```
(or (eq x nil) (eq x 0))
```

構文要素 and のように、 or は cond で書き表すことができます。例えば:

```
(or arg1 arg2 arg3)
≡
(cond (arg1)
      (arg2)
      (arg3))
```

if を使っても or とほとんど同じものを書き表すことができますが、完全ではありません。

```
(if arg1 arg1
  (if arg2 arg2
    arg3))
```

これは *arg1* または *arg2* を 2 回評価するので、完全には等しくありません。一方、(or *arg1 arg2 arg3*) では、何度も評価されることは決してありません。

9.4 繰り返し

繰り返しはプログラムの一部を反復して実行することを意味します。例えば、あなたはリストの個々の要素について一度ずつ、または 0 から n までの個々の整数について一度ずつ、ある計算を行ないたいとします。あなたは特殊形式 while を使用して Emacs Lisp でこれを行なうことができます。

while *condition forms...* Special Form

while は最初に *condition* を評価します。その結果が nil でなければ、テキストどおりの順序で *forms* を評価します。そして、 *condition* を再評価し、その結果が nil でなけれ

ば *forms* を再び評価します。この処理は *condition* が `nil` に評価されるまで繰り返します。

繰り返しの回数には制限がありません。その繰り返しは、 *condition* が `nil` に評価されるか、エラーまたは `throw` による脱出 (9.5 節「非局所脱出」 p.116 参照) が発生するまで継続します。

形式 `while` の値は常に `nil` です。

```
(setq num 0)
     ⇒ 0
(while (< num 4)
  (princ (format "Iteration %d." num))
  (setq num (1+ num)))
     ⊣ Iteration 0.
     ⊣ Iteration 1.
     ⊣ Iteration 2.
     ⊣ Iteration 3.
     ⇒ nil
```

個々の繰り返しで終了テストの直前に毎回何かを実行したいならば、以下に示すように `while` の最初の引数として `progn` を置き、その中に、終了テストと一緒に置きます。

```
(while (progn
         (forward-line 1)
         (not (looking-at "^$"))))
```

これは前方に 1 ライン移動し、空行に達するまで行移動を継続します。 `while` が本体を持たず、終了テストだけ (ポイントの移動という実際の処理も行うが) という点で、まれなものです。

9.5 非局所脱出

『非局所脱出 (nonlocal exit)』はプログラム中のある位置から、別の離れた位置への制御の移行です。非局所脱出は Emacs Lisp では、エラーの結果として生じることがあります。また、明示的な制御のもとで、それを使用することもできます。非局所脱出は脱出しようとしている構文要素にある全ての変数の束縛を解きます。

9.5.1 明示的な非局所脱出: `catch` と `throw`

たいていの制御構造は構造自身内での制御の流れにのみ影響します。関数 `throw` は、この通常のプログラム実行の規則の例外です。それは要求により非局所脱出を実行します (他の例外もありますが、それはエラーをハンドルするだけのためのものです)。 `throw` は `catch` の内側で使用され、その `catch` に飛びます。例えば:

```
(catch 'foo
  (progn
    ...
    (throw 'foo t)
    ...))
```

throw は該当する catch の直後に制御を移します (該当する catch から直ちにリターンする)。throw 以降のコードは実行されません。 throw の 2 番目の引数は catch の戻り値として使用されます。

throw と catch は最初の引数によって対応がつけられます。 throw は、最初の引数が throw で指定されたものと eq である catch を検索します。したがって上記の例では、 throw は、 foo を指しており、 catch も同じシンボルを指定しているので、その catch が適用されます。二つ以上の適用可能な catch がある場合は、最も内側のものが優先されます。

throw を実行すると、対応する catch までの、すべての Lisp 構文要素 (関数呼出しも含む) から脱出します。 let または関数呼出しのような束縛をつくる構文要素からこのようにして脱出した場合、それらの構文要素が普通に終了するのと同様に、その束縛は解除されます (10.3 節「ローカル変数」 p.128 参照)。同様に、 throw は save-excursion (27.3 節「脱線」 p.466 参照) によって保存されていたバッファとポジション、 save-restriction によって保存されていたナローイング状態、 save-window-excursion によって保存されていたウィンドウ・セレクション (25.16 節「ウィンドウ構成」 p.432 参照) を復元します。その形式を脱出するときには、unwind-protect 特殊形式によって設定されたあらゆるクリーンアップも行ないます (9.5.4 節「非局所脱出のクリーンアップ」 p.125 参照)。

throw は、それがジャンプする先の catch がレキシカルに存在していることを必要としません。それは catch の中で呼び出される別の関数からも同様に呼び出されることができます。 throw が評価順で catch の後であり、それから抜けるのが評価順で前であるかぎり、その catch へのアクセスを持ちます。これがエディタのコマンド・ループに戻る exit-recursive-edit (18.10 節「再帰編集」 p.288 参照) のようなコマンドで throw が使用できる理由です。

> **Common Lisp 注意書き:** Common Lisp を含む、ほとんどの他のバージョンの Lisp は非順次的に制御を移す方法をいくつか持っています。例えば、 return, return-from, go があります。 Emacs Lisp は throw だけを持ちます。

catch *tag body...* Special Form

catch は throw 関数のための復帰位置を確立します。復帰位置は *tag*(任意の Lisp オブジェクトを指定可能) によって他の復帰位置と区別されます。普通は復帰位置が確立される前に、引数 tag が評価されます。

有効な復帰位置を持って、 catch は、テキストどおりの順序で *body* の形式を評価します。エラーまたは非局所脱出なしに、普通にその形式が実行されるならば、 *body* の最後の形式の値が catch から返されます。

throw が同じ値 *tag* を指定して *body* の中で行なわれるならば、直ちに catch を抜けます。それが返す値は throw の 2 番目の引数として指定されたものです。

throw *tag value* Function

throw の目的は前もって catch によって設定された復帰位置に戻ることです。引数 *tag* は、いくつかの存在する復帰位置の中から選択するために使用されます。それは catch に指定された値と eq にならなければなりません。複数の復帰位置が *tag* と一致する場合、最も内側のものが使用されます。

引数 *value* は、その catch から返す値として使用されます。

実効的にタグ *tag* に一致する復帰位置がない場合には、 no-catch エラーがデータ (*tag value*) をともなって発行されます。

9.5.2 catch と throw の例

catch と throw を使用する一つの方法は 2 重に入れ子になったループから抜けることです。(たいていの言語では、これは “go to” で行なわれます。) 以下は *i* と *j* をそれぞれ 0 から 9 まで値を変えていきながら (foo *i j*) を計算します。

```
(defun search-foo ()
  (catch 'loop
    (let ((i 0))
      (while (< i 10)
        (let ((j 0))
          (while (< j 10)
            (if (foo i j)
                (throw 'loop (list i j)))
            (setq j (1+ j))))
        (setq i (1+ i))))))
```

foo が仮にに nil でない値を返したならば、直ちに中断して、 *i* と *j* のリストを返します。 foo が常に nil を返すならば、 catch は普通に復帰し、その値は nil になります (それが while の結果であるので)。

次に、二つのトリッキーな例を示します。それらは、同時に二つの復帰位置を示していますが、わずかに異なっています。

最初のものには同じタグ hack をもつ二つの復帰位置があります。

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
⇒ catch2

(catch 'hack
  (print (catch2 'hack))
  'no)
⊣ yes
⇒ no
```

両方の復帰位置が throw と一致するタグを持っているので、それは内側の catch2 で設定されたものに行きます。したがって catch2 は普通に値 yes を返し、この値が印字されます。最後には外側の catch の 2 番目の形式 ('no) が評価され、外側の catch から復帰します。

ここで、 catch2 に与えれている引数を変えてみましょう。

```
(defun catch2 (tag)
  (catch tag
    (throw 'hack 'yes)))
⇒ catch2

(catch 'hack
  (print (catch2 'quux))
  'no)
⇒ yes
```

この例も二つの復帰位置を持ちますが、外側のものだけがタグ `hack` を持ちます。内側のものは、その代わりにタグ `quux` を持っています。したがって、 `throw` により、外側の `catch` が値 `yes` を返します。関数 `print` は決して呼び出されず、本体の形式 `'no` は決して評価されません。

9.5.3 エラー

Emacs Lisp は何かの理由によって評価できない形式を評価しようとした場合、『エラー (error)』を『発行 (signal)』します。

エラーが発行された場合、 Emacs のデフォルトの反応はエラーメッセージを印字し、現在のコマンドの実行を終了することです。これは、ほとんどの場合適切なものです。例えば、バッファの最後で `C-f` をタイプするような場合です。

複雑なプログラムでは、単純な終了が望むものではないかもしれません。例えば、そのプログラムはデータ構造の一時的な変更を行っているかもしれないし、プログラムが終了する前に消去されるべき、一時的なバッファを作成しているかもしれません。そのような場合にはエラーの場合に評価される『クリーンアップ式 (cleanup expression)』を設定する `unwind-protect` を使用します (9.5.4 節「非局所脱出のクリーンアップ」 p.125 参照)。ときにはサブルーチンでエラーがあっても、実行の継続を望むかもしれません。このような場合には、エラーの場合に制御を回復するための『エラー・ハンドラ (error handler)』を設定する `condition-case` を使用します。

プログラムのある場所から別の場所に制御を移すためにエラーのハンドルを使用する、という誘惑には抵抗してください。その代わりには `catch` と `throw` を使用してください。 9.5.1 節「明示的な非局所脱出: `catch` と `throw`」 p.116 を参照してください。

9.5.3.1 エラーを発行する方法

ほとんどのエラーは、他の目的で呼び出している Lisp プリミティブ (例えば、整数の CAR を取ろうとしたり、バッファの最後で前方に移動しようとすること) によって「自動的に」発行されます。一方、関数 `error` と `signal` によって、明示的にエラーを発行することもできます。

とりやめ (ユーザが `C-g` をタイプしたときに起きること) はエラーとは考えられませんが、ほとんどエラーと同様にハンドルされます。 18.8 節「とりやめ」 p.284 を参照してください。

error *format-string* &rest *args* Function

この関数は、 `format` (4.5 節「文字と文字列間の変換」p.60 参照) を *format-string* と *args* に適用しエラー・メッセージを作り、エラーを発行します。

以下は `error` の典型的な使用例を示しています。

```
(error "You have committed an error.
        Try something else.")
     error  You have committed an error.
        Try something else.

(error "You have committed %d errors." 10)
     error  You have committed 10 errors.
```

`error` は、二つの引数 (エラー・シンボル `error` と `format` によって返される文字列を含んだリスト) とともに `signal` を呼び出すことをします。

文字列をエラー・メッセージとしてそのまま使用したい場合、(`error` *string*) とだけ書いてはいけません。 *string* が '%' を含んでいる場合、それは整形指定として解釈され、不正な結果になってしまいます。その代わりに、(`error "%s"` *string*) としてください。

signal *error-symbol data* Function

この関数は *error-symbol* によって名づけられたエラーを発行します。引数 *data* はエラーの状況に関連する追加の Lisp オブジェクトです。

引数 *error-symbol* は、『エラー・シンボル (error symbol)』 (値が条件名のリストである属性 `error-conditions` をもつシンボル) でなければなりません。これが、 Emacs Lisp が種々のエラーを分類するやり方です。

data 中のオブジェクトの数と意味は *error-symbol* に依存します。例えば、 `wrong-type-arg` エラーでは、リストに二つのオブジェクトを入れます。期待されている型を記述する述語と、その型との適合に失敗したオブジェクトです。エラー・シンボルの説明は、 9.5.3.4 節「エラーシンボルと条件名」p.124 を参照してください。

error-symbol と *data* の両方を、エラーをハンドルする任意のエラー・ハンドラは利用できます。 `condition-case` はローカル変数に形式 (*error-symbol* . *data*) のリストを束縛します (9.5.3.3 節「エラーをハンドルするコードの記述」p.121 参照)。エラーがハンドルされなければ、これら二つの値がエラー・メッセージの印字に使用されます。

関数 `signal` から戻ることはありません。 (しかし、古いバージョンの Emacs では戻れることもありました)。

```
(signal 'wrong-number-of-arguments '(x y))
     error  Wrong number of arguments: x, y
(signal 'no-such-error '("My unknown error condition."))
     error  peculiar error: "My unknown error condition."
```

Common Lisp 注意書き: Emacs Lisp には Common Lisp の continuable エラーの概念に相当するものはありません。

9.5.3.2 Emacs がエラーをハンドルする方法

エラーが発行された場合、 `signal` は、そのエラーについて有効な『ハンドラ (handler)』を探します。ハンドラは、 Lisp プログラムの一部でエラーが起きた場合に実行されるように指定された Lisp 式のならびです。そのエラーが適切なハンドラを持っていれば、そのハンドラが実行され、そのハンドラの後から、制御が再開されます。そのハンドラは、それを、設定された `condition-case` の環境で実行されます。 `condition-case` で呼ばれるすべての関数は、すでに脱出しているので、ハンドラはそれらに戻れません。

そのエラーに対して適切なハンドラがなければ、現在のコマンドは終了させられ、制御はエディタのコマンド・ループに戻ります。コマンド・ループには、あらゆる種類のエラーを扱う暗黙のハンドラがあるからです。コマンド・ループのハンドラはそのエラー・シンボルと、エラー・メッセージの印字に関連したデータを使用します。

明示的なハンドラを持たないエラーは Lisp デバッガを呼び出すことができます。デバッガは変数 `debug-on-error` (15.1.1 節「エラーでデバッガにはいる」 p.193 参照) が `nil` でない場合に有効にされます。エラー・ハンドラとは違い、デバッガはエラーの環境で動作するので、あなたはエラーの時点での変数の値を正確に調べることができます。

9.5.3.3 エラーをハンドルするコードの記述

エラーを発行することの通常の効果は動作しているコマンドを終了し、直ちに Emacs エディタのコマンド・ループに戻ることです。特殊形式 `condition-case` でエラー・ハンドラを設定すると、プログラムのある部分で起こるエラーを捕捉するようにすることができます。以下に簡単な例を示します。

```
(condition-case nil
    (delete-file filename)
  (error nil))
```

これは、 *filename* という名前のファイルを消去し、どんなエラーでもそれが起きたときには捕捉し、 `nil` を返します。

`condition-case` の 2 番目の引数は『保護形式 (protected form)』です (先の例での保護形式は `delete-file` を呼び出すことです)。この形式の実行が始まるときにエラー・ハンドラが有効になり、この形式が戻るときに無効化されます。それは介在する時間のすべてにおいて有効状態にあります。特に、この形式、そのサブルーチン、その他によって呼び出される関数の実行中には有効状態にあります。厳密な話、エラーは保護形式によって呼び出される Lisp プリミティブ (`signal` と `error` を含む) によって発行されるのであって、保護形式自身によって発行されるものではないので、これはよいことです。

引数の中で、保護形式の後にはハンドラが続きます。個々のハンドラには、ハンドルするエラーを指定する、一つ以上の (シンボルである) 『状態名 (condition name)』のリストを置きます。エラーを発行するときに指定するエラーシンボルも、状態名のリストを定義します。それらが共通の状態名をもつならば、そのハンドラがエラーに適合します。先の例では一つのハンドラがあり、それはすべてのエラーをカバーする一つの状態名 `error` を指定しています。

適用可能なハンドラの探索は最も最近に設定されたハンドラから始め、すべての設定されたハンドラをチェックします。したがって、二重にネストしている `condition-case` 形式が同じエラーをハンドルするようになっている場合には、二つのうちの内側のものが実際にそれをハンドルすることになります。

エラーがハンドルされるときには、制御がハンドラに渡ります。これが起こる前に、 Emacs は抜けようとしている束縛構文要素によって束縛されている全ての変数の束縛を解除し、抜けたすべての `unwind-protect` 形式のクリーンアップを実行します。いったん、制御がハンドラに到達すると、そのハンドラの本体が実行されます。

ハンドラの本体の実行の後、実行は `condition-case` 形式から戻ることで続行します。保護形式はハンドラの実行前に完全に抜けているので、ハンドラはエラー時点の実行を再開することはできず、保護形式で束縛していた変数を調べることもできません。行えることのすべてはクリーンアップと続行です。

`condition-case` は、 `insert-file-contents` の呼び出しでファイルのオープンに失敗するといった、予想できるエラーを捕捉するのによく使用されます。また、プログラムがユーザから読み込んだ式を評価するときに起きるような、まったく予測不可能なエラーを捕捉するのにも使用されます。

エラーの発行とハンドルは、 `throw` と `catch` と類似している点がありますが、まったく異なった機能です。エラーは `catch` で捕まえることはできませんし、 `throw` はエラー・ハンドラでハンドルすることができません (しかし、ふさわしい `catch` がない `throw` を使うと、ハンドルできるエラーを発行します)。

condition-case *var protected-form handlers...* Special Form

この特殊形式は、 *protected-form* の実行の間に有効になるエラー・ハンドラ *handlers* を設定します。 *protected-form* がエラーなしで実行されると、その戻り値が、 `condition-case` 形式の値になります。この場合、 `condition-case` はまったく効果を持ちません。 *protected-form* の実行中にエラーが発生した場合、 `condition-case` は違う動作を行ないます。

handlers のそれぞれは、形式 (*conditions body...*) のリストです。 *conditions* はハンドルされるエラーの状態名、または状態名のリストです。 *body* は、このハンドラがエラーをハンドルするときに実行される一つ以上の Lisp 式です。以下はハンドラの例です。

```
(error nil)

(arith-error (message "Division by zero"))

((arith-error file-error)
 (message
  "Either division by zero or failure to open a file"))
```

起こる個々のエラーは、それが何の種類のエラーであるかを示す『エラー・シンボル (error symbol)』を持ちます。そのシンボルの `error-conditions` 属性は状態名のリストです (9.5.3.4 節「エラーシンボルと条件名」 p.124 参照)。 Emacs は、これらの状態名のうちの一つ以上を指定するハンドラをすべての有効な `condition-case` 形式から検索します。最も内側で一致する `condition-case` がエラーをハンドルします。この `condition-case` において、最初の適用可能なハンドラがエラーをハンドルします。

ハンドラの本体の実行の後、 condition-case からは普通に戻り、ハンドラの本体の中で最後の形式の値を、その全体の値として使用します。

引数 *var* は変数です。 condition-case は *protected-form* を実行しているとき、この変数を束縛せず、エラーをハンドルするときにのみ束縛します。その時点では変数 *var* を形式 (*error-symbol* . *data*) に局所的に束縛し、エラーの詳細を与えます。ハンドラは何を行うかを決定するために、このリストを参照することができます。例えば、エラーが、ファイルのオープンに失敗したというものであれば、ファイル名は *data* の 2 番目の要素 (*var* の 3 番目の要素) になっています。

var が nil ならば、変数を束縛する手段はありません。ハンドラは、エラー・シンボルおよび関連するデータを使えません。

以下は 0 による除算から起きるエラーをハンドルするために condition-case を使用した例です。ハンドラは警告メッセージを印字し、非常に大きな数を返します。

```
(defun safe-divide (dividend divisor)
  (condition-case err
      ;; 保護形式。
      (/ dividend divisor)
    ;; ハンドラ。
    (arith-error                        ; 状態。
     (princ (format "Arithmetic error: %s" err))
     1000000)))
⇒ safe-divide

(safe-divide 5 0)
     ⊣ Arithmetic error: (arith-error)
⇒ 1000000
```

ハンドラは状態名 arith-error を指定しているので、 0 による除算だけをハンドルします。他の種類のエラーはハンドルされません。少なくとも、この condition-case によってはハンドルされません。したがって、

```
(safe-divide nil 3)
     error  Wrong type argument: integer-or-marker-p, nil
```

以下は、あらゆる種類のエラー (error で発行されたものも含む) を捕らえる condition-case です。

```
(setq baz 34)
     ⇒ 34

(condition-case err
    (if (eq baz 35)
        t
      ;; これは関数 error の呼び出しです。
      (error "Rats!  The variable %s was %s, not 35" 'baz baz))
  ;; これはハンドラです; 形式ではありません。
  (error (princ (format "The error was: %s" err))
         2))
⊣ The error was: (error "Rats!  The variable baz was 34, not 35")
⇒ 2
```

9.5.3.4 エラーシンボルと条件名

エラーを発行するときには、あなたが念頭に置いているエラーの種類を示す『エラーシンボル (error symbol)』を指定します。個々のエラーは、それを分類するために、唯一無二のエラーシンボル持っています。これは Emacs Lisp 言語で定義された、最も優れたエラーの分類法です。

この狭い分類法は『状態名 (condition name)』で識別される『エラー状態 (error condition)』と呼ばれる、より幅のある種別の階層にグループ化されています。そのような種別の中で最も狭いものが、エラー・シンボルそのものです。個々のエラー・シンボルは状態名でもあります。より広い種別の状態名もあり、それはあらゆる種類のエラーをとる状態名 `error` にまで至ります。したがって、個々のエラーは一つ以上の状態名を持っています。すなわち、 `error`、`error` からより明瞭にする場合のエラー・シンボル、そして場合によっては中間の分類によるもの。

シンボルがエラー・シンボルであるためには、状態名のリストを与える `error-conditions` 属性を持たなければなりません。このリストは、その種類のエラーが属する状態を定義します (エラー・シンボル自身とシンボル `error` は、常にこのリストのメンバーであるべきです)。したがって、状態名の階層はエラー・シンボルの `error-conditions` 属性によって定義されます。

`error-conditions` リストに加えて、エラー・シンボルは `error-message` 属性をもつべきであり、その値は、そのエラーが発行されたけれどもハンドルされない場合に印字される文字列です。 `error-message` 属性が存在するけれども文字列でない場合にはエラー・メッセージ '`peculiar error`' が使用されます。

以下はエラー・シンボル `new-error` を定義する方法を示しています。

```
(put 'new-error
     'error-conditions
     '(error my-own-errors new-error))
⇒ (error my-own-errors new-error)
(put 'new-error 'error-message "A new error")
⇒ "A new error"
```

このエラーは三つの状態名を持っています。最も狭い分類である `new-error`、それよりは広い分類であろう `my-own-errors`、そしてすべてに渡る最も広いものである `error` です。

エラーの文字列は英大文字で始まるべきですが、ピリオドで終るべきではありません。これは Emacs の他のものとの一貫性のためです。

もちろん Emacs は、それ自身では `new-error` を発行しません。自分のコードで `signal` (9.5.3.1 節「エラーを発行する方法」 p.119 参照) を明示的に呼び出すことによってのみ、これを行うことができます。

```
(signal 'new-error '(x y))
     error  A new error: x, y
```

このエラーは三つの状態名のうちのどれかでハンドルすることができます。以下の例は、`new-error` と `my-own-errors` に分類される任意のエラーをハンドルします。

```
(condition-case foo
    (bar nil t)
  (my-own-errors nil))
```

エラーを分類する有意義な方法は、状態名 (エラーをハンドラと一致させるために使用する名前) によることです。エラーシンボルは意図したエラー・メッセージと状態名のリストとを指定するのに便利な方法としてのみ役立ちます。一つのエラー・シンボルではなく、状態名のリストを `signal` に与えるのは面倒なことです。

逆に、状態名ではなく、エラー・シンボルを使用することは `condition-case` の能力を明らかに減少させます。状態名は、あなたがエラー・ハンドラを書くときに、普遍性のあるさまざまなレベルにエラーを分類することを可能にします。エラー・シンボル単独で使用することは最も狭いレベルの分類以外をすべてふるい落します。

すべての標準のエラー・シンボルとその状態の一覧は、付録 C「標準のエラー」p.659 を参照してください。

9.5.4 非局所脱出のクリーンアップ

構文要素 `unwind-protect` は、一時的に、あるデータ構造を一貫性のない状態に置くときには常に必須となります。つまり、エラーまたは throw の場合に、データが一貫していることを保証することができます。

unwind-protect *body cleanup-forms. . .* Special Form

`unwind-protect` は、制御が *body* から離れた場合、そこでどのようなことが起ころうとも *cleanup-forms* が評価される、という保証の上で *body* を実行します。 *body* が普通に完了する場合、 `unwind-protect` の外に `throw` が実行される場合、エラーが起きた場合など、あらゆる場合に *cleanup-forms* が評価されます。

body 形式が普通に終る場合、 `unwind-protect` は *cleanup-forms* を評価した後に、最後の *body* 形式の値を返します。 *body* 形式が終了しなかった場合、 `unwind-protect` は普通の感覚においては何も値を返しません。

実際には *body* だけが `unwind-protect` によって保護されます。 *cleanup-forms* 自身のどこかで、非局所的な脱出が起きた場合 (例えば、 `throw` またはエラー)、 `unwind-protect` は、その後の部分の評価を保証しません。 *cleanup-forms* のうちの一つの失敗に、問題を引き起こす可能性がある場合には、その形式を別の `unwind-protect` で保護してください。

現在有効な `unwind-protect` 形式の個数は、ローカル変数の束縛の数とともに、上限 `max-specpdl-size` に対してカウントされます (10.3 節「ローカル変数」p.128 参照)。

例えば、以下では一時的な利用のために不可視のバッファを作り、終る前にかならずそれを削除するようにしています。

```
(save-excursion
  (let ((buffer (get-buffer-create " *temp*")))
    (set-buffer buffer)
    (unwind-protect
        body
      (kill-buffer buffer))))
```

`(kill-buffer (current-buffer))` と書き、変数 `buffer` なしですますことができると考えるかもしれません。しかし、違うバッファに切り替わった後に *body* がエラーを受け取ることが

起こり得るならば、上に示した方法の方が安全です！ (代わりに、本体のまわりに別の `save-excursion` を書き、一時的なバッファが、それを削除するときにカレントになることを保証するようにすることができます。)

以下はファイル '`ftp.el`' から取ってきた実際の例です。それはリモート・マシンへの接続を設定しようとするプロセスを作ります (第 33 章「プロセス」p.559 参照)。関数 `ftp-login` は、関数の作者が予期できない多くの問題に直面する確率が高いので、失敗の場合にプロセスの消去を保証する形式で保護します。そうしなければ、 Emacs は無駄なサブプロセスで満たされるかもしれません。

```
(let ((win nil))
  (unwind-protect
      (progn
        (setq process (ftp-setup-buffer host file))
        (if (setq win (ftp-login process host user password))
            (message "Logged in")
          (error "Ftp login failed")))
    (or win (and process (delete-process process)))))
```

この例は実際には小さなバグを持っています。ユーザがとりやめようとして `C-g` をタイプし、関数 `ftp-setup-buffer` から戻った直後で、変数 `process` が設定される前に中断が起きると、そのプロセスは削除されません。このバグを修正する容易な方法はありませんが、とにかく、それは非常にありそうもないことです。

以下は一時的なバッファを必ず削除するようにするために `unwind-protect` を使用する別の例です。この例では `unwind-protect` によって返却される値が使用されています。

```
(defun shell-command-string (cmd)
  "Return the output of the shell command CMD, as a string."
  (save-excursion
    (set-buffer (generate-new-buffer " OS*cmd"))
    (shell-command cmd t)
    (unwind-protect
        (buffer-string)
      (kill-buffer (current-buffer)))))
```

10 変数

『変数 (variable)』とはプログラム内で使用される名前で、ある値を表します。ほとんどすべてのプログラミング言語にはこのような変数があります。 Lisp プログラムのテキストでは、変数はシンボルの構文で記述します。

通常のプログラミング言語と異なり、 Lisp のプログラムは主に Lisp オブジェクトとして表され、テキストとしては二次的にしか表されません。変数に使用される Lisp オブジェクトはシンボルです。シンボル名が変数名で、変数の値はシンボルの値セルに格納されます。シンボルは、変数として使うこととと関数名として使うこととは独立です。 7.1 節「シンボルの要素」p.93 を参照してください。

Lisp プログラムを構成する Lisp オブジェクトは、そのプログラムのテキスト形式を決定します。単に、その Lisp オブジェクトのリード構文であるということです。このため、たとえばテキストによる Lisp プログラムの変数の記述には、その変数を表すシンボルのリード構文を使用します。

10.1 グローバル変数

変数のもっとも簡単な使用方法は『グローバルに (globally)』使うことです。すなわち変数の値は一度に一つだけで、この値は (少なくともこの時点では)Lisp システム全域に対して有効であるということです。値は、新しく指定するまで有効です。値が指定されて元の値が置き換えられた場合、元の値に対する痕跡は変数内に存在しなくなります。

シンボルに対する値は setq を用いて指定します。例を示します。

```
(setq x '(a b))
```

上の例では、変数 x に値 (a b) が指定されます。 setq は最初の引数 (変数名) は評価しませんが、二番目の引数 (新規の値) は評価します。

変数がいったん値を持つと、シンボルそれ自体を式として用いた参照が行なえます。すなわち、

```
x ⇒ (a b)
```

ここでは、上述したような setq 形式がすでに実行されたことを前提としています。

再度 setq を行なうと、元の値が新しい値に置き換わります。

```
x
     ⇒ (a b)
(setq x 4)
     ⇒ 4
x
     ⇒ 4
```

10.2 変更されない変数

Emacs Lisp には、 `nil` および `t` という特別なシンボルがあり、この二つの評価結果は常に自分自身になります。これらのシンボルは、再束縛したり値セルを変更したりすることはできません。`nil` や `t` を変更しようとすると、 `setting-constant` エラーが発生します。

```
nil ≡ 'nil
     ⇒ nil
(setq nil 500)
[error]  Attempt to set constant symbol: nil
```

10.3 ローカル変数

グローバル変数の値は、明示的に書き換えるまで保持されます。ただし、一時的にしか存在しない、すなわちプログラムの中で局所的 (ローカル) にしか存在しない変数の値を作成した方が便利な場合もあります。このような値は『ローカルである (local)』といい、その場合に使用する変数を『ローカル変数 (local variables)』といいます。

たとえば、関数が呼出された場合、仮引数はローカルな値を新しく受け取り、関数を抜けるまで保持します。 `let` 特殊形式は、指定された変数に対し新規のローカルな値を明示的に確立します。これらの値は `let` 形式から抜けるまで存在しています。

ローカルな値を確立すると、その変数の直前の値 (または値が存在しないという情報) が保存されます。ローカルな値のライフタイムが終わると、直前の値が復元されます。その間、直前の値は『隠されて (shadowed)』おり、『不可視 (not visible)』になっていたということができます。グローバル変数とローカル変数はどちらも隠しておくことができます (10.8.1 節「スコープ」 p.137 参照)。

変数がローカルであるときにセットすると (`setq` などを用いて)、ローカルの値が置き換わります。グローバルの値や、隠されているローカル値の直前の値は変更されません。この動作を示すために、『局所的な束縛 (local binding)』について、ローカル値と同じように説明しましょう。

局所的な束縛とは、局所的な値を保持している概念的な場所です。関数や、 `let` のような特殊形式を開始すると、局所的な束縛が作成されます。関数を抜けると、 `let` は局所的な束縛を解除します。局所的な束縛が続いている間、変数の値はその中に格納されています。局所的な束縛があるときに `setq` や `set` を使用すると、その局所的な束縛には別の値が格納されます。新規に束縛が作成されるわけではありません。

『グローバル束縛 (global binding)』 についても述べておきましょう。ここには、 (概念的に) グローバル変数が保持されます。

変数には、局所的な束縛を同時に複数持たせることもできます (たとえば、ネストした `let` 形式が束縛しているような場合)。このような場合、存在している最新の局所的な束縛が、その変数の『現在の束縛 (current binding)』になります。これを『動的スコーピング (dynamic scoping)』といいます。 10.8 節「変数束縛のスコーピングに関する規則」 p.137 を参照してください。局所的な束縛がない場合、その変数のグローバル束縛が現在の束縛になります。また、

現在の束縛のことを『既存の最も局所的な束縛 (most-local existing binding)』と呼んで強調することもあります。シンボルを通常に評価すると、かならずその現在の束縛の値を返します。

特殊形式 `let` および `let*` は、局所的な束縛を作成するためのものです。

let (*bindings...*) *forms...* Special Form

この特殊形式は、 *bindings* に従って変数を束縛し、 *forms* をすべて記述された順序で評価します。 `let`- 形式は *forms* にある末尾の形式の値を返します。

bindings はそれぞれ、(i) aシンボルまたは (ii) 形式のリスト (*symbol value-form*) になります。前者の場合シンボルは `nil` に束縛され、後者の場合 *symbol* は *value-form* の評価結果に束縛されます。 *value-form* が省略された場合は `nil` が使用されます。

bindings の *value-form* はすべて現れた順に、どのシンボルが束縛されるより前に評価されます。例を示します。 `Z` は `Y` の元の値、つまり新規の値 1 ではなく 2 に束縛されます。

```
(setq Y 2)
     ⇒ 2
(let ((Y 1)
      (Z Y))
  (list Y Z))
     ⇒ (1 2)
```

let* (*bindings...*) *forms...* Special Form

この特殊形式は `let` のようなものですが、この場合、束縛するのはローカル値を計算した直後で、次の変数のローカル値を計算する前になります。そのため、 *bindings* の式は、この `let*` 形式で先行するシンボルを適切に参照できるようになります。下に示す例を、上の `let` の例と比べてみてください。

```
(setq Y 2)
     ⇒ 2
(let* ((Y 1)
       (Z Y))    ; Y の確立されたばかりの値を使う
  (list Y Z))
     ⇒ (1 1)
```

その他、局所的な束縛を作成する機能をすべて下に示します。

- 関数の呼出し (第 11 章「関数」 p.147 参照)。
- マクロの呼出し (第 12 章「マクロ」 p.163 参照)。
- `condition-case`(9.5.3 節「エラー」 p.119 参照)

変数ではバッファローカルな束縛を持つことも可能です (10.9 節「バッファローカルな変数」 p.140 参照)。端末ローカルな束縛を持つ変数もわずかですがあります (26.2 節「複数のディスプレイ」 p.435 参照)。このような種類の束縛処理は、通常の局所的な束縛と多少類似していますが、それは、ローカライズした時点ではなく、 Emacs の「どこに」いるかということに依存します。

max-specpdl-size Variable

この変数は、ローカル変数束縛と `unwind-protect` のクリーンアップの総数の上限を定義します (9.5 節「非局所脱出」 p.116 参照)。これらは、エラーが通知されるまで (データ `"Variable binding depth exceeds max-specpdl-size"`による) 行なわれます。

この制限を超えると、それに対応するエラーが発生します。これは、適切に定義されていない関数への無限再帰を防ぐためのものです。

デフォルトの値は 600 です。

`max-lisp-eval-depth` はネスティングに対する別の制限です。 8.1 節「Eval」 p.102 を参照してください。

10.4 変数が "Void" である場合

シンボルにグローバル変数としての値をまったく指定していない場合、そのシンボルのグローバル値は『void』であるといいます。つまり、そのシンボルのセルの中には Lisp オブジェクトがないということです。そのシンボルを評価しようとすると、値を取得することはできず、`void-variable` エラーが発生します。

`nil` は void とは別であることに注意してください。シンボル `nil` は Lisp オブジェクトで、ほかのオブジェクトと同じように変数の値にすることもできます。ただしこれは値です。 void 変数には値はありません。

変数に値を代入した後で、 `makunbound` を使用すると、変数は再度 void になります。

makunbound *symbol* Function

この関数は、 *symbol* の現在の束縛を void にします。続けて、このシンボルの値を変数として使用しようとすると、再度セットするまで `void-variable` エラーになります。

`makunbound` は *symbol* を返します。

```
(makunbound 'x)         ; xvoid の
                        ;     グローバル値を作成
     ⇒ x
x
error  Symbol's value as variable is void: x
```

symbol が局所的に束縛されている場合、 `makunbound` は最も局所的な束縛に作用します。シンボルの局所的な束縛を void にする方法はこれだけです。局所的な束縛を作成する構文要素は、値とともに作成するためです。この場合、 void であるという状態は、長くても束縛と同じ期間しか継続しません。構文要素を抜けるなどで束縛が削除された場合、直前の束縛またはグローバルな束縛が通常どおり再度現れ、新規に現れた束縛が void である場合を除き、変数は void ではなくなります。

```
(setq x 1)               ; グローバル束縛に値を設定
     ⇒ 1
(let ((x 2))             ; 局所的に束縛
  (makunbound 'x)        ; 局所的な束縛を void にする
  x)
error  Symbol's value as variable is void: x
```

```
x                          ; グローバル束縛は変更されていない
     ⇒ 1

(let ((x 2))               ; 局所的に束縛
  (let ((x 3))             ; くり返し
    (makunbound 'x)        ; もっとも内側の局所的な束縛を void にする
    x))                    ; 参照: void
error  Symbol's value as variable is void: x
(let ((x 2))
  (let ((x 3))
    (makunbound 'x))       ; 内側の束縛を void にしてから削除
  x)                       ; 外側の let 束縛が可視になる
     ⇒ 2
```

makunbound を用いて void にされた変数と、値が与えられなかったために void だった変数は、区別できません。

boundp 関数を使うと、変数が現在 void かどうかが調べられます。

boundp *variable* Function

boundp は、 *variable*(シンボル) が void でない場合 (正確には、変数の現在の束縛が void でない場合) は t を返します。 void である場合は、 nil を返します。

```
(boundp 'abracadabra)          ; void から開始
     ⇒ nil
(let ((abracadabra 5))         ; 局所的に束縛
  (boundp 'abracadabra))
     ⇒ t
(boundp 'abracadabra)          ; グローバルでは void のまま
     ⇒ nil
(setq abracadabra 5)           ; グローバルでも非 void にする
     ⇒ 5
(boundp 'abracadabra)
     ⇒ t
```

10.5 グローバル変数の定義

シンボルをグローバル変数として使用するということは、『変数定義 (variable definition)』によってアナウンスできます。これは特殊形式で、 defconst または defvar のいずれかになります。

Emacs Lisp では、定義により 3 種類の目的が達成できます。まず、コードを読む人間に対して、特定のシンボルが特定の方法で (変数として) 使用されることを意図されているということが伝わります。二番目に、このような情報が Lisp システムにも伝わり、値や説明が与えられます。三番目に、 etags や make-docfile など、プログラム内の関数や変数のデータベースを作成するユーティリティにも情報が伝わります。

defconst と defvar の主な違いは、プログラムが変数を変更するかどうかを人間の読者に対して伝えるという意図にあります。 Emacs Lisp は、変数を使用する方法に関して、 defconst で宣言したか defvar で宣言したかという点を区別しません。ただし、初期化には違いがあります。 defconst は無条件で変数を初期化しますが、 defvar は void の場合にかぎり初期化を行ないます。

「ユーザ・オプションの変数は、プログラムによって値が変更されることがないので、defconst で定義する方がいい」と思われることもありますが、プリロードされないライブラリで定義されている場合は、悪い結果を生じるでしょう。ライブラリがロードされる際、defconst は前の値より優先します。ユーザの初期化ファイルを使用してユーザ・オプションをセットし、定義に対するデフォルト値を書き換えることもできます。このため、ユーザ・オプションは defvar で定義する必要があります。

defvar *symbol* [*value* [*doc-string*]] Special Form

この特殊形式は *symbol* を value に定義して、初期化します。この定義により、 *symbol* が変数として使用されており、したがってプログラムがセットや変更を行うこともあるのだということが、コードを読む人にわかるようになります。これは、ユーザ・オプション変数 (プリロードされた Emacs の一部であるものを除く) にも使用されます。 *symbol* は評価されないので注意してください。定義されるシンボルは、 defvar に明示されている必要があります。

symbol に値がすでにある場合 (すなわち、 void ではない場合)、 *value* は評価されず、 *symbol* の値は変更されません。 *symbol* が void で *value* が指定されている場合、defvar は評価を行ない、結果に *symbol* ををセットします。 (*value* を省略すると、*symbol* の値が変更されることはありません。)

Eamcs Lisp モードで C-M-x を用いてトップレベルの defvar 形式を評価する (eval-defun) と、 eval-defun の特殊機能により defconst と評価されます。この動作の目的は、個別の指示があった場合に、変数の値の再初期化を確実に行なうということです。

symbol のカレント・バッファに、バッファローカルな束縛があれば、 defvar はローカルな値ではなくデフォルトの値をセットします。 10.9 節「バッファローカルな変数」p.140 を参照してください。

doc-string 引数があれば、これが変数の説明を指定します。 (文書化を指定できるということが、変数定義の主な利点のひとつです。) 文書はシンボルの variable-documentation 属性に格納されます。 Emacs のヘルプ機能 (第 21 章「ヘルプ」p.339 参照) はこの属性を検索します。

doc-string の最初の文字が ‘*’ の場合、この変数はユーザ・オプションであると考えられます。これにより、ユーザが set-variable および edit-options で変数をセットすることが便利になります。

たとえば、この形式は foo を定義しますが、その値は定義しません。

```
(defvar foo)
     ⇒ foo
```

次の例では、 bar の値を 23 にセットし、説明文字列も指定します。

```
(defvar bar 23
  "The normal weight of a bar.")
     ⇒ bar
```

次の形式は、 bar の説明文字列を変更し、変数をユーザ・オプションにしますが、値は変更しません。 bar にはすでに値があるためです。 (追加されている (1+23) は実行されません。)

```
(defvar bar (1+ 23)
  "*The normal weight of a bar.")
     ⇒ bar
bar
     ⇒ 23
```

defvar 特殊形式と等価である式の例を次に示します。

```
(defvar symbol value doc-string)
≡
(progn
  (if (not (boundp 'symbol))
      (setq symbol value))
  (put 'symbol 'variable-documentation 'doc-string)
  'symbol)
```

defvar 形式は *symbol* を返しますが、通常はファイル中のトップレベルで使用されるので、値は問題になりません。

defconst *symbol* [*value* [*doc-string*]] Special Form

この特殊形式は *symbol* を value に定義して、初期化します。これにより、 *symbol* にグローバルな値があり、それがここで確立され、通常はプログラムの実行によって変更されることも局所的に束縛されることもないということが、コードを読む人にわかるようになります。ただし、ユーザが変更することもできます。 *symbol* は評価されないので注意してください。定義されるシンボルは、 defconst に明示されている必要があります。

defconst は、かならず *value* を評価して、その結果にグローバルな値 *symbol* をセットします。これには *value* が指定されているということが条件です。 *symbol* に、バッファローカルな束縛がカレント・バッファにある場合、 defconst はローカルな値ではなくデフォルトの値をセットします。

注意: 標準的なプリロードを行なっていないライブラリでは、 defconst をユーザ・オプション変数に使用しないでください。このような変数にはファイル '.emacs' でユーザによる値の指定ができるようにしておき、ライブラリを後からロードしても効果があるようにしてください。

ここで、 pi はおそらく誰にも変更されない (インディアナ州議会はやろうとしたが) 定数です。ただし、二番目の形式でそれは単に勧告であるということを示しています。

```
(defconst pi 3.1415 "Pi to five places.")
     ⇒ pi
(setq pi 3)
     ⇒ pi
pi
     ⇒ 3
```

user-variable-p *variable* Function

variable がユーザ・オプション (ユーザによるカスタマイズのためのセットを意図した変数) であれば、この関数は t を返し、それ以外の場合は nil を返します。 (ユーザ・オプション以外の変数も、 Lisp プログラム内で内部的に使用されており、ユーザはそれを知る必要はありません。)

ユーザ・オプション変数は、 variable-documentation 属性の最初の文字によって、ほかの変数から区別されています。 '*' で始まる文字列の属性があれば、その変数はユーザ・オプションです。

ユーザ・オプションの変数に variable-interactive 属性があれば、 set-variable コマンドはその値を使用して、変数に対する新しい値の読込みを制御します。属性の値は、interactive の場合と同じように使用されます。

警告 !: 変数に局所的な束縛がある場合に、特殊形式 defconst および defvar を使用すると、局所的な束縛の値がセットされます。グローバルな束縛は変更されません。これは本来意図したところではないので、これを防ぐために、このような特殊形式はファイルのトップレベルで使用してください。ここでは、通常局所的な束縛は有効ではないからです。また、かならずファイルをロードしてから変数に対する局所的な束縛を作成するようにしてください。

10.6 変数の値へのアクセス

変数を参照する場合、最も一般的な方法は、その変数を特定するシンボルを書くことです (8.2.2 節「シンボル形式」 p.105 参照)。この場合、プログラムを記述する際に変数名を指定する必要があります。これが通常のやり方でしょう。ただし、どの変数を参照するかを実行中に選択しなければならないこともあります。これには、 symbol-value を使用します。

symbol-value *symbol* Function

この関数は、 *symbol* の値を返します。これは、シンボルの局所的な束縛の中で最も内側にあるものの値です。また、局所的な束縛がない場合はグローバルな値になります。

```
(setq abracadabra 5)
     ⇒ 5
(setq foo 9)
     ⇒ 9

;; ここでは、シンボル abracadabra
;;    の値が調べられる。
(let ((abracadabra 'foo))
  (symbol-value 'abracadabra))
     ⇒ foo

;; ここでは、 abracadabra
;;    すなわち foo の
;;    値が調べられる
(let ((abracadabra 'foo))
  (symbol-value abracadabra))
     ⇒ 9
```

```
(symbol-value 'abracadabra)
     ⇒ 5
```

エラー void-variable は、 *symbol* に局所的な束縛もグローバルな値もないということを示します。

10.7 変数の値を変更する方法

変数の値を変更する場合、通常は特殊形式 setq を使用します。実行時に変数の選択を計算する必要がある場合は、関数 set を使用します。

setq [*symbol form*]. . . Special Form

この特殊形式は、変数の値を変更するための最も一般的な方法です。 *symbol* には、それぞれ対応する *form* の評価結果である新規の値が与えられます。シンボルに対して存在する束縛の中で、最もローカルなものが変更されます。

setq は *symbol* を評価せず、記述したシンボルをセットします。この引数は『自動的に quote(automatically quoted)』されます。 'q' は「quoted」の略です。

setq 形式の値は、最後にある *form* の値になります。

```
(setq x (1+ 2))
     ⇒ 3
x                     ; x がグローバルな値を取得
     ⇒ 3
(let ((x 5))
  (setq x 6)          ; x の局所的な束縛をセット
  x)
     ⇒ 6
x                     ; グローバルな値は変更されない
     ⇒ 3
```

最初の *form* が評価され、それから最初の *symbol* がセットされ、次に二番目の *form* が評価され、二番目の *symbol* がセットされるという順序で実行されることに注意してください。

```
(setq x 10            ; x は y の値を計算する前に
      y (1+ x))       ;    セットされているということに注意
     ⇒ 11
```

set *symbol value* Function

この関数は、 *symbol* の値を *value* にセットし、 *value* を返します。 codeset は関数なので、 *symbol* に対して記述した式が評価されて、セットするシンボルを取得します。

変数の既存の束縛で最も局所的な束縛がセットされます。隠されている束縛は影響されません。

```
(set one 1)
error  Symbol's value as variable is void: one
```

```
(set 'one 1)
     ⇒ 1
(set 'two 'one)
     ⇒ one
(set two 2)         ; two はシンボル one に評価される
     ⇒ 2
one                 ; したがって、 one がセットされる
     ⇒ 2
(let ((one 1))      ; one のこの束縛をセット
  (set 'one 3)      ;    グローバルな値ではない
  one)
     ⇒ 3
one
     ⇒ 2
```

symbol が実際のシンボルではない場合、 `wrong-type-argument` エラーが発生します。

```
(set '(x y) 'z)
error  Wrong type argument: symbolp, (x y)
```

論理的にいうと、 `set` は `setq` よりもプリミティブなものです。 `setq` を使っている箇所はすべて、 `set` を使うように書き換えることが簡単にできます。 `set` が使える場合は、 `setq` をマクロで定義することもできます。ただし、 `set` 自身が使用されることはほとんどなく、初心者が知る必要はないでしょう。この操作が便利になるのは、実行時にどの変数をセットするかを選択するような場合に限られます。たとえば、 `set-variable` コマンドはユーザから変数名を読込んで変数をセットするので、 `set` を使用する必要があります。

> **CL 注意書き:** Common Lisp では、 `set` はかならずシンボルの特殊な値を変更し、辞書的な束縛をすべて無視します。 Emacs Lisp では、変数と束縛はすべて (実質上) 特殊なので、 `set` は常に、最も局所的な束縛に影響を与えます。

変数をセットする別の関数には、リストに入っていない要素をリストに追加するように設計されているものがあります。

add-to-list *symbol element* Function

この関数は、 *element* がその値のメンバになっていない場合に、 *element* を従来の値に cons することで、変数 *symbol* の値をセットします。この関数は、リストが更新されてもされなくてもリストを返します。 *symbol* の値は、呼び出す前にリストにしておくといいでしょう。

symbol を引用符で囲むということは含意されていません。 `add-to-list` は `set` と同じような通常の関数で、 `setq` とは異なっています。必要に応じて、引数を引用符で囲んでください。

`add-to-list` の使用例を示します。

```
(setq foo '(a b))
     ⇒ (a b)
```

```
(add-to-list 'foo 'c)     ;; c を追加
     ⇒ (c a b)

(add-to-list 'foo 'b)     ;; 効果なし
     ⇒ (c a b)

foo                       ;; foo を変更
     ⇒ (c a b)
```

(add-to-list '*var value*) は、次のような式に相当します。

```
(or (member value var)
    (setq var (cons value var)))
```

10.8 変数束縛のスコーピングに関する規則

指定されたシンボル foo には、 Lisp プログラムの各所で確立されたいくつかのローカルな変数束縛と、グローバルな束縛が存在することがあります。最も新しく確立された束縛は、ほかのものより優先されます。

Emacs Lisp の局所的な束縛には『無限スコープ (indefinite scope)』と『動的エクステント (dynamic extent)』があります。『スコープ (Scope)』とは、束縛がソースコードの本文中のどの部分にアクセスできるかということを表しています。無限スコープとは、変数の束縛がプログラムのどの部分にもアクセスできる可能性があるということです。『エクステント (Extent)』は、プログラムの実行中、どの時点で束縛が存在するかということを表します。動的エクステントとは、その束縛を確立した構文要素がアクティブである間は、かならず束縛が継続して存在するということを表します。

動的エクステントと無限スコープを組み合わせを『動的スコーピング (dynamic scoping)』といいます。それに対して、ほとんどのプログラミング言語では、『辞書的スコーピング (lexical scoping)』を用いています。この場合、ローカル変数への参照は、関数の内部か、またはその変数を束縛するブロックの内部に存在している必要があります。

> **Common Lisp に関する注意:** Common Lisp で “special” として宣言された変数は、Emacs Lisp の変数と同じようにスコープは動的になります。

10.8.1 スコープ

Emacs Lisp は、ローカルな変数束縛に『無限スコープ (indefinite scope)』を使用します。これは、プログラムのテキスト内にある関数はすべて、指定された変数束縛へのアクセスが可能であることを表します。次のような関数定義を考えてみてください。

```
(defun binder (x)   ; x は binder で束縛される
   (foo 5))         ; foo はほかの関数

(defun user ()      ; x は user で使用される
  (list x))
```

辞書的にスコープしている言語では、 binder の中の x に user でアクセスすることはできません。 user が関数 binder に含まれていないためです。ただし、動的スコープの Emacs Lisp では、 user は、 binder で確立される x の束縛を参照する場合と参照しない場合 (状況による) があります。

- binder をまったく呼出さずに user を直接呼出すと、 x の束縛があっても、それは binder からきたものではありません。
- foo を以下のように定義して binder を呼出すと、 binder で作成された束縛は、 user で可視になります。

```
(defun foo (lose)
  (user))
```

- foo を以下のように定義して binder を呼出すと、 binder で作成された束縛は、 user で不可視になります。

```
(defun foo (x)
  (user))
```

  ここで、 foo が binder から呼出されると、 x が束縛されます。 (foo で束縛を行なうことは binder で作成されたものを『隠す (shadow)』といいます。) したがって、 user は、 binder ではなく foo で束縛された x にアクセスできるようになります。

10.8.2 エクステント

『エクステント (extent)』とは、プログラムを実行する際に変数名が有効である時間を表します。 Emacs Lisp では、変数が有効になるのは、その変数を束縛する形式の実行中に限られます。これを、『動的エクステント (dynamic extent)』といいます。 C や Pascal など、ほとんどの言語では「ローカル」または「自動」変数に動的エクステントがあります。

別の動的エクステントに『無限エクステント (indefinite extent)』があります。これは、変数の束縛が、その束縛を作成した形式から抜けた後も存続するという意味です。たとえば、 Common Lisp や Scheme ではこれをサポートしていますが、 Emacs Lisp ではサポートしていません。

これを図示すると、以下の関数で make-add は、独自の引数 m に n を追加する関数を返します。これは Common Lisp では動作しますが、 Emacs Lisp を意図した場合には動作しません。 make-add への呼出しを抜けた後、変数 n は実引数 2 には束縛されていないためです。

```
(defun make-add (n)
    (function (lambda (m) (+ n m))))  ; 関数を返す
     ⇒ make-add
(fset 'add2 (make-add 2))  ; (make-add 2) で
                           ;    関数 add2 を定義
     ⇒ (lambda (m) (+ n m))
(add2 4)                   ; 2 と 4 を加算
error  Symbol's value as variable is void: n
```

Lisp の方言には “closure” を持つものがあります。これは関数と同じようなオブジェクトですが、変数束縛も記録できます。 Emacs Lisp には closure はありません。

10.8.3 動的スコープの実装

簡単な実装例 (Emacs Lisp の実際の動作というわけではありません) により、動的束縛が理解しやすくなるでしょう。この手法は『深い束縛 (deep binding)』と呼ばれており、初期の Lisp システムで使用されていました。

束縛のスタック (変数と値の対) があるとします。関数または `let` 形式の開始地点で、引数またはそこで作成されるローカル変数に対して束縛をスタックにプッシュします。束縛構造を抜けた時点で、スタックから束縛をポップすることができます。

変数の値を検索する場合は、スタックを先頭から末尾まで調べて、その値に対する束縛を検索します。その束縛からの値は変数の値です。変数をセットするには、まず現在の束縛を検索し、新規の値をその束縛に格納します。

みればわかるはずですが、関数の束縛は実行中であればかならず有効になっています。これは、ほかの関数を呼出している場合でも変わりません。束縛のエクステントが動的であるというのは、このためです。ほかの関数からでも、その束縛が有効である時点で同じ変数を使用すると、この束縛を参照できます。スコープが無限であるというのは、このためです。

変数スコーピングを実装する場合、 GNU Emacs Lisp では『浅い束縛 (shallow binding)』という手法を使用します。各変数には標準的な位置があり、現在の値はすべてその場所にあります。すなわち、シンボルの値セルです。

浅い束縛では、変数のセットは、値を値セルに格納することで行ないます。新規の束縛は、従来の値 (直前の束縛に属する) をスタックにプッシュし、ローカルな値を値セルに格納することで作成されます。束縛の除去は、従来の値をスタックから値セルにポップすることで行ないます。

浅い束縛を使用する理由は、深い束縛と同じ結果が得られて高速に実行できるためです。これは、束縛を検索する必要がないためです。

10.8.4 動的スコーピングの標準的な使い方

ある関数で変数を束縛してほかの関数で使用することは、高度な手法ですが、乱用するとプログラムが判読しにくくなります。この手法をわかりやすく使用する方法は二とおりあります。

- 変数を束縛して使用する関数を、同じファイルの近い位置に記述された関連のある少数の関数に限定します。このような変数は、同じプログラムの内部で伝達のために使用されます。

  コメントを記述して、ほかのプログラマにも使用前に値がすべてわかるようにし、ほかの場所で使用を追加しないように忠告してください。

- 変数の定義を注意深く行ない、意味を文書に記載し、適切な関数がすべて、意味が関連している場合に参照する (束縛やセットは行なわない) ようにします。たとえば、変数 `case-fold-search` を「非 `nil` の場合、検索時に大文字 / 小文字を区別しない」と定義するとします。検索・置換の関数はすべて直接またはサブルーチンを介してその変数を参照しますが、束縛やセットは行ないません。

  これで、その変数の効果が確実にわかり、ほかのプログラムでその変数が束縛できるようになります。

どちらの場合でも、変数は defvar で定義してください。これにより、ほかのプログラマたちも、関数間の使用方法をみればプログラムがわかるようになります。また、バイトコンパイラからの警告も避けられます。変数名が重複しないように注意してください —x などの短い名称は避けてください。

10.9 バッファローカルな変数

グローバル変数とローカル変数の束縛は、ほとんどのプログラミング言語でさまざまな形式で行なわれています。 Emacs もあまり一般的でない種類の変数束縛をサポートしています。これは『バッファローカル (buffer-local)』な束縛といい、適用されるバッファはひとつに限定されます。 Emacs Lisp は編集コマンドをプログラムするためのものであり、別々のバッファに別々の値が入るということが、カスタマイズの重要な方法なのです。 (指定された X 端末に対して局所的な束縛を持つ変数も多少あります。 26.2 節「複数のディスプレイ」 p.435 を参照してください。)

10.9.1 バッファローカルな変数 (紹介)

バッファローカルな変数には、バッファローカルな束縛があり、特定のバッファに対応しています。束縛は、そのバッファがカレント・バッファであれば有効です。カレント・バッファでなければ有効ではありません。バッファローカルな束縛が有効な時点で変数をセットすると、新規の値はその束縛にはいるので、グローバルな束縛は変更されません。すなわち、変更がわかるのはそのバッファのみになるということです。

変数には、バッファローカルな束縛が一部のバッファにしかないというものがあります。グローバルな束縛は、独自の束縛を持たないバッファすべてに共有されます。したがって、バッファローカルな束縛を持たないバッファにある変数をセットすると、新規の値は、バッファローカルな束縛を持たないバッファすべてで可視になります。 (ここでは、問題を複雑にする let 形式の局所的な束縛がないことを前提にしています。)

バッファローカルな束縛を使用する場合、もっとも一般的なものは主モードのもので、コマンドの動作を制御する変数を変更するという方法です。たとえば、 C モードと Lisp モードは、どちらも変数 paragraph-start をセットすることにより、空行のみがパラグラフを分割するという指定を行ないます。これは、 C モードや Lisp モードになったバッファローカルな変数を作成し、そのモードに対する新規の値をセットすることで行ないます。

バッファローカルな束縛は、通常、主モード・コマンドが使用する make-local-variable で作成します。これはカレント・バッファのみに影響します。ほかのバッファ (まだ作成されていないものも含む) はグローバルな値の共有を継続します。

さらに高度な操作として、 make-variable-buffer-local を呼出して変数を『automatically buffer-local』と印づけるという方法があります。これは、すべてのバッファで変数をローカルにすると考えることができます。さらに正確にいうと、この効果は、変数をセットした場合、その変数がカレント・バッファに対してバッファローカルでなければ、自動的にバッファローカルになるということです。すべてのバッファは、最初は変数のグローバルな値を共有しますが、 setq があれば、カレント・バッファに対するバッファローカルな束縛を作成します。新

規の値はバッファローカルな束縛を格納し、 (デフォルトの) グローバルな束縛は変更されません。グローバルな値は setq で変更されることがなくなるので、変更には setq-default を使用する必要があります。

警告 !: 変数が一つまたは複数のバッファにローカル変数を持っていた場合、変数を let に束縛し、ほかの束縛が有効であるほかのカレント・バッファに変更して let を抜けるので Emacs はひじょうに混乱します。これにより、グローバル束縛および局所的な束縛の値がスクランブルします。

正常を保つためには、このような一連の動作を避けるようにしてください。カレント・バッファを変更する各コードの周囲で save-excursion を使用すると、このような問題が発生します。何を避けるかという例をここに示します。

```
(setq foo 'b)
(set-buffer "a")
(make-local-variable 'foo)
(setq foo 'a)
(let ((foo 'temp))
  (set-buffer "b")
  body...)
foo ⇒ 'a        ; バッファ 'a' に対する従来のローカルな値が
                ;     デフォルト値になる
(set-buffer "a")
foo ⇒ 'temp     ; 消去すべきローカルな値が
                ;     バッファ 'a' に対するローカルな値になる
```

ただし、ここで示すように save-excursion が問題を回避します。

```
(let ((foo 'temp))
  (save-excursion
    (set-buffer "b")
    body...))
```

body 内の foo への参照が、バッファ 'b' に対してバッファローカルな束縛にアクセスするという点に注意してください。

ファイルがローカルな値を指定する場合、これらの値はファイルを visit した時点でバッファローカルになります。 20.1.3 節「主モードの自動選択の仕組み」 p.323 を参照してください。

10.9.2 バッファローカルな束縛の作成と削除

make-local-variable *variable* Command

この関数は、バッファローカルな束縛を、 *variable*(シンボル) に対するカレント・バッファに作成します。ほかのバッファには影響しません。返される値は *variable* になります。

バッファローカルな *variable* の値は、最初は、 *variable* の直前の値と同じになります。 *variable* が void の場合は void のままです。

```
;; In buffer `b1':
(setq foo 5)                 ; 全バッファに影響
     ⇒ 5
(make-local-variable 'foo)  ; `b1' 内でローカルになる
     ⇒ foo
foo                          ; 値を変更
     ⇒ 5                     ;    しなかった
(setq foo 6)                 ; `b1' の値を
     ⇒ 6                     ;    変更
foo
     ⇒ 6

;; バッファ `b2' では、値は変更されていません。
(save-excursion
  (set-buffer "b2")
  foo)
     ⇒ 5
```

バッファローカルな変数を作成する場合、その変数に対する `let` 束縛の内部で行なっても動作しません。これは、 `let` が異なる種類の束縛を区別しないためです。束縛がどの変数に対して作成されたかということのみを識別します。

変数が端末ローカルであれば、この関数はエラーを通知します。このような変数には、バッファローカルな束縛も使用できません。 26.2 節「複数のディスプレイ」 p.435 を参照してください。

注意 !: フック変数には `make-local-variable` ではなく `make-local-hook` を使用してください。 20.4 節「フック」 p.335 を参照してください。

make-variable-buffer-local *variable* Command

この関数は *variable*(シンボル) が自動的にバッファローカルになるように印づけるので、その後でセットしようとすると、その時点のカレント・バッファに対してローカルになります。

戻り値は *variable* です。

local-variable-p *variable* &optional *buffer* Function

バッファ *buffer*(デフォルトではカレント・バッファ) の中で *variable* がバッファローカルであれば、 `t` を返します。それ以外の場合は `nil` を返します。

buffer-local-variables &optional *buffer* Function

この関数は、 *buffer* の中の、バッファローカルな変数の連想リストを返します。これは、連想リスト (5.8 節「連想リスト」 p.81 参照) を返します。このリストでは、それぞれ対応する項目にバッファローカルな変数がひとつと、その値が含まれています。 *buffer* の中の、バッファローカルな変数が void であれば、結果のリストに直接はいります。 *buffer* が省略されると、カレント・バッファが使用されます。

```
(make-local-variable 'foobar)
(makunbound 'foobar)
(make-local-variable 'bind-me)
(setq bind-me 69)
(setq lcl (buffer-local-variables))
    ;; 最初は、すべてのバッファでローカルな組込み変数
⇒ ((mark-active . nil)
    (buffer-undo-list nil)
    (mode-name . "Fundamental")
    ...
    ;; 次に、非組込みのローカル変数
    ;; これはローカルで void になっている
    foobar
    ;; これはローカルで void になっていない
    (bind-me . 69))
```

このリストにあるコンス・セルの CDR に新規の値を格納しても、変数のローカルな値は変更されないので注意してください。

kill-local-variable *variable* Command

この関数は、カレント・バッファにある *variable*(シンボル) に関して、バッファローカルな束縛があればそれを消去します。結果として、 *variable* のグローバル (デフォルト) 束縛がこのバッファで可視になります。通常、このために *variable* の値が変更されます。これは、グローバルな値が、削除されたばかりの、バッファローカルな値と通常は違うためです。

セットされると自動的にローカルになる変数の局所的な束縛を kill すると、グローバルな値はカレント・バッファで可視になります。ただし、変数を再度セットすると、局所的な束縛が再度作成されます。

`kill-local-variable` は *variable* を返します。

この関数はコマンドです。これは、バッファローカルな変数の対話的な作成が便利であり、同様にバッファローカルな変数を対話的に kill することも便利であるためです。

kill-all-local-variables Function

この関数は、カレント・バッファに関するバッファローカルな変数の束縛をすべて除去しますが、 “permanent” と印づけられた変数は例外です。結果として、バッファにはほとんどの変数に対するデフォルトの値がみえるようになります。

また、この関数はバッファに対するほかの特定情報のリセットも行ないます。これは、ローカルなキーマップを `nil` に、構文テーブルを `standard-syntax-table` の値に、略称テーブルを `fundamental-mode-abbrev-table` の値にセットします。

主モードのコマンドはすべて、この関数の呼出しで開始されます。これには、 Fundamental モードに切り替え、直前の主モードの効力をほとんど削除するという効力があります。この作業を確実に行なうには、主モードがセットする変数に permanent の印をつけないようにしてください。

`kill-all-local-variables` returns `nil`.

ローカル変数は、その変数名 (シンボル) の `permanent-local` の属性が `nil` でない場合に『permanent』になります。常時ローカルな変数は、そのファイルの内容の編集方法ではなく、ファイルの場所や保存方法に属しているデータに対して適切なものになります。

10.9.3 バッファローカルな変数のデフォルト値

バッファローカルな束縛のある変数のグローバルな値は『デフォルト (default)』値ともいいます。この値は、明示して無視しないかぎり有効になるためです。

関数 `default-value` および `setq-default` は、変数のデフォルト値にアクセスしてこの値を変更します。これには、カレント・バッファにバッファローカルな束縛があるかどうかは無関係です。たとえば、 `setq-default` を用いて、ほとんどのバッファに対する `paragraph-start` のデフォルトセットを変更することができます。これは、 C または Lisp モードのバッファにいて、この変数に対してバッファローカルな値がある場合でも同様です。

また、特殊形式 `defvar` および `defconst` は、ローカル値ではなくデフォルト値 (変数をセットすることがあれば) のセットも行ないます。

default-value *symbol* Function

この関数は、 *symbol* のデフォルト値を返します。この値は、この変数に関して独自の値を持たないバッファにある値です。 *symbol* がバッファローカルでなければ、この値は `symbol-value` と等価になります (10.6 節「変数の値へのアクセス」 p.134 参照)。

default-boundp *symbol* Function

関数 `default-boundp` は、 *symbol* のデフォルト値が void 以外であるかどうかを示します。 `(default-boundp 'foo)` が `nil` を返す場合、 `(default-value 'foo)` はエラーになります。

`boundp` が `symbol-value` に対応することと同じように、 `default-boundp` は `default-value` に対応します。

setq-default *symbol value* Special Form

これは、 *symbol* のデフォルト値を *value* にセットします。 *symbol* の評価は行ないませんが、 *value* は評価します。 `setq-default` 形式の値は *value* になります。

symbol がカレント・バッファに対してバッファローカルでなく、自動的にバッファローカルになるという印もついていない場合、 `setq-default` の効果は `setq` と同じになります。 *symbol* がカレント・バッファに対してバッファローカルであれば、これにより、ほかのバッファの値は変更されますが (バッファローカルな値を持たないバッファの場合)、カレント・バッファが認識する値は変更されません。

```
;; In buffer ‘foo’:
(make-local-variable 'local)
     ⇒ local
(setq local 'value-in-foo)
     ⇒ value-in-foo
(setq-default local 'new-default)
     ⇒ new-default
```

```
local
     ⇒ value-in-foo
(default-value 'local)
     ⇒ new-default
;; ‘bar’ バッファ (新規) で:
local
     ⇒ new-default
(default-value 'local)
     ⇒ new-default
(setq local 'another-default)
     ⇒ another-default
(default-value 'local)
     ⇒ another-default
;; ‘foo’ バッファに戻る:
local
     ⇒ value-in-foo
(default-value 'local)
     ⇒ another-default
```

set-default *symbol value* Function

この関数は `setq-default` と同じようなものですが、 *symbol* が評価されるという点が異なります。

```
(set-default (car '(a b c)) 23)
     ⇒ 23
(default-value 'a)
     ⇒ 23
```

11 関数

Lisp プログラムは、主に Lisp 関数で構成されます。この章では、関数という概念、引数の処理方法、関数の定義方法について説明します。

11.1 関数とは

一般に、関数とは、『引数 (argument)』と呼ばれる与えられたいくつかの値に対して計算を実行するための規則です。計算の結果は、関数の値と呼ばれます。計算には、副作用がともなう場合もあります。副作用には、変数の値の変化や、データ構造の内容の変化があります。

Emacs Lisp の関数や関数に準ずるオブジェクトに関する重要な用語をいくつか説明します。

『関数』
: Emacs Lisp では、『関数 (function)』とは、 Lisp プログラムにおいて引数に適用可能なものすべてを指します。場合によっては、 Lisp で書かれた関数を特に意味する場合もあります。特殊形式やマクロは関数ではありません。

『プリミティブ』
: 『プリミティブ (primitive)』とは、 `car` や `append` のように、 C で書かれた、Lisp から呼び出せる関数です。プリミティブは、『組込み (built-in)』関数または『subr』とも呼ばれます (特殊形式もプリミティブとみなされます)。

  通常、関数をプリミティブにする理由には、関数が基本的なものであること、オペレーティング・システムのサービスへの低レベル・インターフェースを実現するものであること、高速に実行する必要があること、などがあります。プリミティブを修正したり追加するには、 C のソースを変更して、エディタをコンパイルしなおさなければなりません。 B.4 節「Emacs のプリミティブを書く」p.650 を参照してください。

『ラムダ式』
: 『ラムダ式 (lambda expression)』とは、 Lisp で書かれた関数です。ラムダ式については、次に示す節で説明しています。

『特殊形式』
: 『特殊形式 (special form)』は、関数に似ていますが、すべての引数を評価する方法が通常の形式とは異なります。特殊形式は、一部の引数だけを評価したり、通常とは異なる順番で引数を評価したり、同じ引数を何度も評価する場合があります。第 9 章「制御構造」p.111 では、たくさんの特殊形式について説明しています。

『マクロ』
: 『マクロ (macro)』とは、プログラマが Lisp で定義する構文要素です。マクロが関数とは異なるのは、マクロは、プログラマが書いた Lisp 式を評価せずに、そのまま等価な式に変換する点です。マクロを使えば、特殊形式と同様のことを Lisp プログラムで実現できます。マクロの定義方法や使い方については、第 12 章「マクロ」p.163 を参照してください。

『コマンド』
: 『コマンド (command)』とは、 `command-execute` が起動できるオブジェクトであり、通常はキー操作の定義内容として使われます。一部の関数はコマンドです。

Lisp で書かれた関数は、対話的コマンド宣言を含んでいればコマンドになります (18.2 節「コマンドの定義」p.258 参照)。コマンド関数は、ほかの関数と同様に Lisp 式から呼び出せます。その場合、その関数がコマンドであることに意味はありません。

キーボード・マクロ (文字列とベクタ) も、関数ではありませんがコマンドです。コマンドを関数定義にもつシンボルもコマンドです。このようなシンボルは、 `M-x` で起動できます。同様に、シンボルの定義が関数ならば、そのシンボルは関数です。 18.1 節「コマンド・ループ概観」p.257 を参照してください。

『キーストローク・コマンド』

『キーストローク・コマンド (keystroke command)』とは、キー・シーケンス (通常は一つから三つのキーストローク) にバインドされたコマンドです。キーストローク・コマンドとコマンドの違いをここで説明しているのは、 Emacs 以外のエディタでの “コマンド” と混同しないようにするためだけです。通常、 Lisp プログラムでは、キーストローク・コマンドとコマンドの違いは重要ではありません。

『バイトコード関数』

『バイトコード関数 (byte-code function)』とは、バイトコンパイラでコンパイルされた関数です。 2.3.13 節「バイトコード関数型」p.31 を参照してください。

subrp *object* Function

この関数は、 *object* が組込み関数 (Lisp プリミティブ) ならば `t` を返します。

```
(subrp 'message)              ; message はシンボルです。
     ⇒ nil                    ;   subr オブジェクトではありません。
(subrp (symbol-function 'message))
     ⇒ t
```

byte-code-function-p *object* Function

この関数は、 *object* がバイトコード関数ならば `t` を返します。例を次に示します。

```
(byte-code-function-p (symbol-function 'next-line))
     ⇒ t
```

11.2 ラムダ式

Lisp で書かれた関数は、次のようなリストです。

```
(lambda (arg-variables...)
  [documentation-string]
  [interactive-declaration]
  body-forms...)
```

このようなリストは、『ラムダ式 (lambda expression)』と呼ばれています。 Emacs Lisp では、ラムダ式は実際には式として有効であり、評価されるとそれ自体になります。一部の Lisp 方言では、ラムダ式は有効な式ではありません。どちらの場合も、ラムダ式は式として評価するためではなく、関数として呼び出すために主に使われます。

11.2.1 ラムダ式の構成要素

ラムダ式の最初の要素は、つねにシンボル `lambda` です。 `lambda` は、リストが関数を表現していることを示します。関数の定義を `lambda` で始めるのは、ほかの用途に使われるリストを誤って関数として有効にしてしまわないためです。

2 番目の要素は、シンボル (引数変数の名前) のリストです。このリストは、『ラムダ・リスト (lambda list)』と呼ばれます。 Lisp 関数が呼び出されると、ラムダ・リスト内の各変数に対して引数の値が対応つけられます。そして、各変数には、渡された値がローカルに束縛されます。 10.3 節「ローカル変数」p.128 を参照してください。

説明文字列は、関数定義内に置かれた Lisp 文字列オブジェクトで、 Emacs ヘルプ機能での関数の説明として使われます。 11.2.4 節「関数の説明文字列」p.151 を参照してください。

interactive 宣言は、 `(interactive` *code-string*`)` という形式のリストです。 interactive 宣言は、関数が対話的に使われたときの引数の供給方法を宣言します。この宣言をもつ関数は、『コマンド (command)』と呼ばれます。コマンドは、 `M-x` を使って呼び出したり、キーにバインドできます。このような方法で (対話的に) 呼び出すべきでない関数には、 interactive 宣言を入れないでください。 interactive 宣言の書き方については、 18.2 節「コマンドの定義」p.258 を参照してください。

残りの要素は、関数の『本体 (body)』です。関数の本体とは、関数の実際の作業を行う Lisp コードです (Lisp プログラミングの観点からは、 "評価される Lisp 形式のリスト" ともいえます)。関数が返す値は、本体の最後の要素が返す値です。

11.2.2 簡単なラムダ式の例

次のような関数を例にとって考えます。

```
(lambda (a b c) (+ a b c))
```

この関数を呼び出すには、次のように、式の CAR にこの関数を書きます。

```
((lambda (a b c) (+ a b c))
 1 2 3)
```

この呼出しは、変数 `a` を 1 に、変数 `b` を 2 に、変数 `c` を 3 に束縛して、ラムダ式の本体を評価します。本体が評価されると、この三つの数が加算され、結果は 6 になります。このため、この関数の呼出しは 6 を返します。

また、次の例のように、ある関数呼出しの結果をさらに別の関数の引数にすることもできます。

```
((lambda (a b c) (+ a b c))
 1 (* 2 3) (- 5 4))
```

この式は、引数の `1`、 `(* 2 3)`、 `(- 5 4)` を左から右に評価します。次に、引数の 1 と 6 と 1 にラムダ式が適用され、結果は 8 になります。

通常は、このように形式の CAR としてラムダ式を書くのは不便です。特殊形式 `let` を使ってローカル変数を作成し、その変数に値を設定すれば、同じ結果を得ることができます (10.3 節「ローカル変数」p.128 参照)。また、 `let` の方がわかりやすく、使うのも簡単です。実際に

は、ラムダ式は、シンボルの関数定義として格納して名前をもつ関数を作成したり、ほかの関数の引数として渡すために使われます (11.7 節「匿名関数」p.156 参照)。

しかし、特殊形式 `let` が発明されるまでの以前の Lisp では、明示的なラムダ式への呼出しはよく使われていました。当時は、ローカル変数に値を束縛して初期化するには、この方法を使うしかありませんでした。

11.2.3 引数リストの高度な機能

この例の `(lambda (a b c) (+ a b c))` という簡単な関数では、三つの引数変数を指定しているため、この関数は三つの引数を指定して呼び出さなければなりません。引数を二つだけしか指定せずに、あるいは、四つも指定してこの関数を呼び出そうとすると、 `wrong-number-of-arguments` エラーが発生します。

一部の引数を省略できる関数を書いた方が便利な場合もよくあります。たとえば、 `substring` 関数は、文字列、開始インデックス、終了インデックスの三つの引数を受けつけます。しかし、3 番目の引数は、省略すると文字列の *length* になります。また、 `list` 関数や `+` 関数のように、関数が無限個の引数を受けつけるようにすると便利な場合もあります。

関数を呼び出す時に省略できる引数を指定するには、 `&optional` キーワードをオプション引数の前につけます。余っている 0 個以上の引数を要素とするリストを指定するには、最後の引数の前に `&rest` キーワードをつけます。

したがって、引数リストの完全な構文は次のようになります。

(*required-vars*...
 [`&optional` *optional-vars*...]
 [`&rest` *rest-var*])

角括弧は、 `&optional` 節と `&rest` 節、およびそれらの後の変数が省略可能であることを示しています。

関数を呼び出す時は、 *required-vars* のそれぞれについて実引数を一つずつ指定しなければなりません。 *optional-vars* には実引数を指定しなくてもかまいません。また、ラムダ・リストで `&rest` が使われていなければ、仮引数の個数を超える実引数を指定することはできません。 `&rest` が使われている場合は、余分な実引数をいくつでも指定できます。

オプション引数や残余引数に対する実引数を省略した場合のデフォルト値は、つねに `nil` です。明示的に `nil` が引数に指定されたのか、それとも引数が省略されたのかを呼び出された関数から区別することはできません。ただし、呼び出された関数の本体は、 `nil` を別の意味を示す省略値として扱うこともできます。たとえば、 `substring` は、 3 番目の引数に `nil` が指定されている場合、指定されている文字列の長さを使います。

> **Common Lisp 注意書き:** Common Lisp では、オプション引数が省略されている場合に使われるデフォルト値を関数に指定できます。 Emacs Lisp では、常に `nil` が使われます。

たとえば、次のような引数リストがあるとします。

```
(a b &optional c d &rest e)
```

この引数リストは、 a と b を最初の二つの引数に束縛します。この二つの引数は必須です。引数がさらに一つまたは二つ指定されている場合は、 c と d はそれぞれ対応する引数に束縛されます。最初の四つの引数の後の引数はすべて一つのリストにまとめられ、 e はそのリストに束縛されます。引数が二つしか指定されていない場合は、 c は nil になります。引数が二つまたは三つの場合は、 d は nil になります。引数が四つ以下の場合は、 e は nil になります。

オプション引数の後に必須引数を置くことはできません (無意味です)。その理由を考えるため、上記の例の c が省略可能で、 d が必須であると仮定してみます。実引数を三つ与えたとすると、三つめの引数はどの変数に束縛されるでしょうか? 同様に、 &rest 引数の後にさらに (必須やオプションの) 引数を置くのも無意味です。

引数リストと正しい呼出しの例をいくつか示します。

```
((lambda (n) (1+ n))                ; 必須引数が一つ:
 1)                                 ; 引数は一つだけで、省略不可能
     ⇒ 2
((lambda (n &optional n1)           ; 必須引数が一つ、オプション引数が一つ:
         (if n1 (+ n n1) (1+ n)))   ; 引数は 1 個または 2 個
 1 2)
     ⇒ 3
((lambda (n &rest ns)               ; 必須引数が一つ、残余引数が一つ:
         (+ n (apply '+ ns)))       ; 引数は 1 個以上
 1 2 3 4 5)
     ⇒ 15
```

11.2.4 関数の説明文字列

ラムダ式のラムダ・リストの直後には、『説明文字列 (documentation string)』を置くこともできます。この文字列は、関数の実行には影響を与えません。この文字列は一種のコメントですが、 Lisp 環境内に実際に存在する系統だったコメントであり、 Emacs のヘルプ機能によって利用できます。 *documentation-string* のアクセス方法については、第 21 章「ヘルプ」 p.339 を参照してください。

たとえ、自分のプログラムからしか呼び出されない関数でも、プログラムのすべての関数には説明文字列を用意しておくのがよいでしょう。説明文字列はコメントのようなものですが、コメントよりも調べやすいという特徴があります。

apropos は説明文字列の 1 行目だけを表示します。このため、説明文字列の 1 行目はそれだけで完結していなければなりません。説明文字列の 1 行目には、関数の用途を要約した一つか二つの文を置いてください。

通常、説明文字列の先頭はソース・ファイル内で字下げしますが、この字下げのための空白は開き二重引用符の前であるため、説明文字列には含まれません。テキストがプログラム・ソースと揃うように、残りの行を字下げする人もいますが、それは間違いです。 2 行目以降の字下げは、説明文字列内に含まれます。ソース・コードではきれいに見えても、ヘルプ・コマンドで表示すると乱れてしまいます。

説明文字列の後には関数の必須要素 (本体) があるのに、説明文字列を省略可能にできるのには理由があります。文字列を評価すると、副作用なしでその文字列自体が返されるため、文字列が関数の本体の最後の要素でなければ問題はありません。このように、実際には、本体の 1 番目の

部分と説明文字列が混同される恐れはありません。本体が文字列だけならば、その文字列は戻り値と説明の両方になります。

11.3 関数の命名

ほとんどのコンピュータ言語では、関数にはかならず名前があり、名前を持たない関数というのは無意味です。 Lisp では、厳密な意味での関数には名前はありません。関数は、最初の要素が `lambda` であるリストか、プリミティブな subr オブジェクトです。

しかし、関数の名前としてシンボルを使うこともできます。この場合、シンボルの『関数セル (function cell)』 (7.1 節「シンボルの要素」 p.93 参照) に関数を置きます。すると、シンボル自体が有効で呼出し可能な関数になり、関数セルが参照するリストまたは subr オブジェクトに等価になります。関数セルの内容は、シンボルの『関数定義 (function definition)』とも呼ばれます。シンボルのかわりにシンボルの関数定義を使う方法として、『シンボル関数間接参照 (symbol function indirection)』という方法があります。 8.2.4 節「シンボル関数間接参照」 p.105 を参照してください。

実際には、ほとんどすべての関数にこのように名前が与えられ、その名前によって参照されます。たとえば、シンボル `car` の関数セルにはプリミティブ subr オブジェクトの `#<subr car>` が格納されているため、シンボル `car` は、関数と同様に機能します。

関数に名前を与えるのは、 Lisp 式で関数を参照しやすくするためです。 `#<subr car>` などのプリミティブ subr オブジェクトの場合、名前でしか参照できません。このようなオブジェクトには、可読の構文はありません。 Lisp で書かれた関数を呼び出す場合も、明示的なラムダ式を記述するより名前を使ったほうが便利です。また、名前をもつ関数は、自分自身を参照できます (再帰呼出しが可能です)。関数の定義にその関数の名前を記述するほうが、関数定義が自分自身を指すようにするよりも簡単です (関数定義が自分自身を指すようにすることは不可能ではありませんが、実際にはさまざまな短所があります)。

関数を示すのに、その関数の名前であるシンボルを使うことがよくあります。たとえば、シンボル `car` とその関数定義に格納されているプリミティブ subr オブジェクトを区別せずに、 “関数 `car`” と表現することがよくあります。ほとんどの場合、この二つを区別する必要はありません。

ただし、関数に一意な名前を与える必要はないことを覚えておいてください。ある特定の関数オブジェクトは、通常は一つのシンボルの関数セルだけに格納されていますが、単に便宜上そうなっているだけです。 `fset` を使えば、複数のシンボルに一つの関数オブジェクトを簡単に格納できます。この場合、どのシンボルも、同じ関数の名前として同様に通用します。

また、関数名として使われているシンボルを変数として使うこともできます。シンボルのこの二つの使い方は独立しており、衝突しません。

11.4 関数の定義方法

通常、関数をはじめて作成するときは、関数に名前を与えます。『関数を定義する (defining a function)』とは、このことであり、関数を定義するには `defun` 特殊形式を使います。

defun *name argument-list body-forms* Special Form

通常、新しい Lisp 関数を定義するには、 `defun` を使います。 `defun` は、シンボル *name* を次のような関数として定義します。

```
(lambda argument-list . body-forms)
```

`defun` は、このラムダ式を *name* の関数セルに格納します。返される値は *name* ですが、通常はこの値を使うことはありません。

以前に説明したように (11.2 節「ラムダ式」p.148 参照)、 *argument-list* は引数の名前のリストであり、キーワードの `&optional` と `&rest` が使えます。また、 *body-forms* の最初の二つの部分は、説明文字列と interactive 宣言にすることもできます。

シンボルの値セルと関数セルは独立しているため、同じシンボル *name* を変数として使っても衝突は発生しません (7.1 節「シンボルの要素」p.93 を参照してください)。

例をいくつか示します。

```
(defun foo () 5)
     ⇒ foo
(foo)
     ⇒ 5

(defun bar (a &optional b &rest c)
    (list a b c))
     ⇒ bar
(bar 1 2 3 4 5)
     ⇒ (1 2 (3 4 5))
(bar 1)
     ⇒ (1 nil nil)
(bar)
error  Wrong number of arguments.

(defun capitalize-backwards ()
  "単語の最後の文字を大文字にします。"
  (interactive)
  (backward-word 1)
  (forward-word 1)
  (backward-char 1)
  (capitalize-word 1))
     ⇒ capitalize-backwards
```

既存の関数を誤って再定義してしまわないように注意してください。 `defun` は、 `car` などのプリミティブ関数でも何の忠告もなしに再定義してしまいます。 Lisp では不注意な再定義と意図的な再定義を区別できないため、すでに定義されている関数の再定義は慎重に行ってください。

defalias *name definition* Function

この特殊形式は、シンボル *name* を、定義 *definition* (任意の有効な Lisp 関数が使えます) をもつ関数として定義します。

`defalias` は、特定の関数名が定義されている箇所 (とくに、ロードされるソース・ファイルに関数名が明示的に現れる箇所) で使ってください。これは、 `defalias` が `defun` と同様に関数を定義しているファイルを記録するためです (13.5 節「アンロード」 p.178 参照)。

これに対して、ほかの目的で関数の定義を操作するプログラムでは、定義されているファイルを記録しない `fset` を使ってください。

`defun` と同様に、関数を定義し、コードを展開するように Lisp コンパイラに指示する `defsubst` も参照してください。 11.9 節「インライン関数」 p.160 を参照してください。

11.5 関数の呼出し

関数の定義は、作業の半分でしかありません。関数は、『呼び出す (call)』 (動作するように指示する) までは何もしません。関数の呼出しは、『起動 (invocation)』ともよばれます。

関数を起動するための最も一般的な方法は、リストを評価することです。たとえば、`(concat "a" "b")` というリストを評価することは、 `"a"`と `"b"`を引数として関数 `concat` を呼び出すことです。評価の説明については、第 8 章「評価」 p.101 を参照してください。

プログラム中の式としてこのリストを書くと、関数の名前はプログラムの一部になります。つまり、呼び出される関数とその関数に与える引数を、プログラムを書く時に選択することになります。通常は、これで十分です。しかし、呼び出される関数を実行時に決定しなければならない場合もあります。そのためには、 `funcall` 関数と `apply` 関数を使います。

funcall *function* &rest *arguments* Function

`funcall` は、 *arguments* を引数として *function* を呼び出し、 *function* が返した値を返します。

`funcall` は関数であるため、 *function* を含めたすべての引数は、 `funcall` が呼び出される前に評価されます。つまり、任意の式を使って、呼び出される関数を指定できます。また、 `funcall` は、 *arguments* に書かれている式ではなく、その値だけを使います。これらの値は、 *function* が呼び出されるとき、再び評価されることはありません。 `funcall` は、引数が評価された時点で、関数を呼び出すための通常の手順に入ります。

function 引数は、 Lisp の関数か、プリミティブ関数でなければなりません。特殊形式とマクロは "評価されていない" 引数式が与えられた場合だけ意味をもつため、ここでは使えません。上記のように、 `funcall` は未評価の引数を知ることができないためです。

```
(setq f 'list)
     ⇒ list
(funcall f 'x 'y 'z)
     ⇒ (x y z)
(funcall f 'x 'y '(z))
     ⇒ (x y (z))
(funcall 'and t nil)
error  Invalid function: #<subr and>
```

上記の例と `apply` の例を比較してください。

apply *function* &rest *arguments* Function

`apply` は、 `funcall` と同様に *arguments* を引数として *function* を呼び出しますが、一つだけ違いがあります。 *arguments* の最後の要素が、単一の引数ではなく、 *function* に渡される引数のリストになっています。また、 `apply` は、このリストの各要素がそれぞれ一つの引数になるように、リストを『散開 (spread)』します。

`apply` は、 *function* を呼び出した結果を返します。 `apply` でも、 `funcall` と同様に、 *function* は Lisp の関数またはプリミティブ関数でなければなりません。 `apply` では特殊形式とマクロは無効です。

```
(setq f 'list)
     ⇒ list
(apply f 'x 'y 'z)
error  Wrong type argument: listp, z
(apply '+ 1 2 '(3 4))
     ⇒ 10
(apply '+ '(1 2 3 4))
     ⇒ 10

(apply 'append '((a b c) nil (x y z) nil))
     ⇒ (a b c x y z)
```

`apply` のおもしろい使い方に関する例は、 11.6 節「マッピング関数」 p.155 の `mapcar` の説明を参照してください。

Lisp 関数の場合、 (フック変数や属性リストのように) 引数として関数を受け取ったり、データ構造内に関数を置き、 `funcall` または `apply` を使ってその関数を呼び出すことがよくあります。関数引数を受けつける関数は、『ファンクショナル (functional)』とも呼ばれます。

ファンクショナルを呼び出す場合、何もしない関数を引数として指定できると便利な場合があります。何もしない関数を二つ紹介します。

identity *arg* Function

この関数は *arg* を返すだけで、副作用はありません。

ignore &rest *args* Function

この関数は、引数を無視して `nil` を返します。

11.6 マッピング関数

『マッピング関数 (mapping function)』は、ある関数をリストなどの各要素に適用します。 Emacs Lisp には、マッピング関数が三つあります。ここで説明する `mapcar` と `mapconcat` は、リストをスキャンします。もう一つのマッピング関数である `mapatoms` については、 7.3 節「シンボルの作成と intern」 p.95 を参照してください。

mapcar *function sequence* Function

mapcar は、 *sequence* の各要素に *function* を順番に適用し、その結果のリストを返します。

sequence 引数には、リスト、ベクタ、または文字列を指定できます。結果は常にリストになります。結果のリストの要素数は、 *sequence* の要素数と同じです。

例:

```
(mapcar 'car '((a b) (c d) (e f)))
     ⇒ (a c e)
(mapcar '1+ [1 2 3])
     ⇒ (2 3 4)
(mapcar 'char-to-string "abc")
     ⇒ ("a" "b" "c")

;; my-hooks に含まれる各関数を呼び出します。
(mapcar 'funcall my-hooks)

(defun mapcar* (f &rest args)
  "すべての引数の car に関数を順番に適用し、
結果のリストを返します。 "
  ;; リストの要素がまだ残っていれば、
  (if (not (memq 'nil args))
      ;; それぞれの CAR に関数を適用します。
      (cons (apply f (mapcar 'car args))
            (apply 'mapcar* f
                   ;; 残りの要素について再帰実行します。
                   (mapcar 'cdr args)))))

(mapcar* 'cons '(a b c) '(1 2 3 4))
     ⇒ ((a . 1) (b . 2) (c . 3))
```

mapconcat *function sequence separator* Function

mapconcat は、 *sequence* の各要素に *function* を適用します。 *function* が返す値は文字列でなければならず、結果は連結されます。 mapconcat は、各要素について返された文字列の間に、文字列 *separator* を挿入します。通常、 *separator* には、スペース、カンマほかの適当な句読点を指定します。

function 引数は、引数を一つ取り、文字列を一つ返す関数でなければなりません。

```
(mapconcat 'symbol-name
           '(The cat in the hat)
           " ")
     ⇒ "The cat in the hat"

(mapconcat (function (lambda (x) (format "%c" (1+ x))))
           "HAL-8000"
           "")
     ⇒ "IBM.9111"
```

11.7 匿名関数

Lisp では、関数は、 lambda で始まるリストか、そのようなリストからコンパイルされたバイトコード関数か、プリミティブな subr オブジェクトです。名前は “おまけ” です。通常、関数は defun を使って定義し、同時に名前も与えますが、明示的にラムダ式 (匿名関数) を使ったほう

がわかりやすい場合もあります。ラムダ式であるリストは、間数名が使えるところならばどこでも使えます。

ラムダ式リストは、どのような手段で作成しても有効な関数になります。たとえば、次のような例も有効です。

```
(setq silly (append '(lambda (x)) (list (list '+ (* 3 4) 'x))))
⇒ (lambda (x) (+ 12 x))
```

上記の例は、(lambda (x) (+ 12 x)) のようなリストを評価し、それを silly の (関数定義ではなく) 値にします。

この関数を呼び出す方法を次に示します。

```
(funcall silly 1)
⇒ 13
```

(silly は silly の関数定義ではないため、(silly 1) のように書いても動作しません。silly には関数定義は設定されておらず、変数としての値が与えられているだけです。)

ほとんどの場合、匿名関数は、プログラム中に固定的に書かれます。たとえば、リストの各要素に関数を適用する mapcar 関数に匿名関数を引数として渡すとします。数値を 2 倍する匿名関数を渡す例を次に示します。

```
(defun double-each (list)
  (mapcar '(lambda (x) (* 2 x)) list))
⇒ double-each
(double-each '(2 11))
⇒ (4 22)
```

このような場合、通常は、単なるクォートを使って匿名関数をクォートせずに、特殊形式 function を使います。

function *function-object* Special Form

この特殊形式は、 *function-object* を評価せずに返します。上記の例では、 quote と等価です。しかし、この特殊形式は. *function-object* が関数としてだけ使われることを Emacs Lisp コンパイラに伝えるため、コンパイルが安全に行なわれるようになります。 8.3 節「quote する」 p.109 の quote と比較してみてください。

quote の代わりに function を使うと、コンパイルしようとする関数またはマクロの内部で違いがあります。例を示しましょう。

```
(defun double-each (list)
  (mapcar (function (lambda (x) (* 2 x))) list))
⇒ double-each
(double-each '(2 11))
⇒ (4 22)
```

この double-each の定義をコンパイルすると、匿名関数もコンパイルされます。一方、通常の quote を使った前の例の場合、 mapcar に渡される引数は、次に示すリストになります。

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

このリストは関数の形をしていますが、 Lisp コンパイラは、 mapcar がリストに対して行う処理を知らないため、このリストが関数であることを仮定できません。第 3 要素の CAR がシンボ

ル * であることを mapcar が調べる可能性もあるためです。 function の利点は、変化しない関数をコンパイルしても問題がないことをコンパイラに伝えられることにあります。

関数の名前をクォートするときに、 quote ではなく function と書く場合もありますが、これは単にわかりやすくするためだけです。

(function *symbol*) ≡ (quote *symbol*) ≡ '*symbol*

function と匿名関数を使った実例については、 21.2 節「説明文字列へのアクセス」p.340 の documentation を参照してください。

11.8 関数セルの内容へのアクセス

シンボルの『関数定義 (function definition)』は、シンボルの関数セルに格納されているオブジェクトです。ここでは、シンボルの関数セルをアクセス、テスト、設定する関数について説明します。

8.2.4 節「シンボル関数間接参照」 p.105 の indirect-function 関数も参照してください。

symbol-function *symbol* Function

この関数は、 *symbol* の関数セルに格納されているオブジェクトを返します。シンボルの関数セルが void ならば、 void-function エラーが発生します。

この関数は、返されるオブジェクトが有効な関数かどうかはチェックしません。

```
(defun bar (n) (+ n 2))
     ⇒ bar
(symbol-function 'bar)
     ⇒ (lambda (n) (+ n 2))
(fset 'baz 'bar)
     ⇒ bar
(symbol-function 'baz)
     ⇒ bar
```

シンボルに関数定義をまったく与えていない場合、そのシンボルの関数セルが『void』であるといいます。つまり、関数セルには Lisp オブジェクトが入っていない状態です。このようなシンボルを呼び出そうとすると、 void-function エラーが発生します。

void は、 nil やシンボル void とは異なります。シンボルの nil と void は Lisp オブジェクトであり、ほかのオブジェクトと同様、関数セルに格納できます (さらに、 defun を使って定義すれば有効な関数になります)。 void な関数セルには、オブジェクトがまったく入っていません。

シンボルの関数定義が void かどうかを調べるには、 fboundp を使います。いったん関数定義を与えたシンボルをもう一度 void にするには、 fmakunbound を使います。

fboundp *symbol* Function

この関数は、シンボルの関数セルにオブジェクトがあれば t を返し、そうでなければ nil を返します。オブジェクトが有効な関数かどうかはチェックしません。

fmakunbound *symbol* Function

この関数は、 *symbol* の関数セルを void にします。この関数を呼び出した後、関数セルにアクセスしようとすると、 `void-function` エラーが発生します (10.3 節「ローカル変数」p.128 の `makunbound` を参照してください)。

```
(defun foo (x) x)
     ⇒ x
(foo 1)
     ⇒1
(fmakunbound 'foo)
     ⇒ x
(foo 1)
error  Symbol's function definition is void: foo
```

fset *symbol object* Function

この関数は、 *symbol* の関数セルに *object* を格納し、 *object* を返します。通常、 *object* は関数または関数名ですが、チェックは行なわれません。

この関数には、主に三つの使い方があります:

- あるシンボルの関数定義を別のシンボルにコピーする場合 (つまり、関数の別名を作る場合)
- リストではないため `defun` で作成できない関数定義をシンボルに与える場合。たとえば、 `fset` を使って、シンボル `s1` に別のシンボル `s2` を関数定義として与えると、 `s1` は、 `s2` の現在の定義とまったく同じものの別名になります。
- 関数を定義したり変更する構文要素内。 `defun` がプリミティブでないとしたら、 `fset` を使って Lisp で (マクロとして) 書くことになるはずです。

最初の二つの場合の例を示します:

```
;; car と同じ定義を first に与えます。
(fset 'first (symbol-function 'car))
     ⇒ #<subr car>
(first '(1 2 3))
     ⇒ 1

;; シンボル car を xfirst の関数定義にします。
(fset 'xfirst 'car)
     ⇒ car
(xfirst '(1 2 3))
     ⇒ 1
(symbol-function 'xfirst)
     ⇒ car
(symbol-function (symbol-function 'xfirst))
     ⇒ #<subr car>

;; 名前つきのキーボード・マクロを定義します。
(fset 'kill-two-lines "\^u2\^k")
     ⇒ "\^u2\^k"
```

11.4 節「関数の定義方法」 p.152 の defalias 関数も参照してください。

すでに定義されている関数を拡張する関数を作成する場合、次に示す常套手段がときどき使われます。

```
(fset 'old-foo (symbol-function 'foo))
(defun foo ()
  "more-so を除いて old-foo と同じ。 "
  (old-foo)
  (more-so))
```

foo が自動ロードされるように定義されている場合、この方法はうまくいきません。そのような場合、 foo が old-foo を呼び出すと、 Lisp システムはファイルをロードして old-foo を定義しようとします。ファイルをロードすると、おそらく old-foo ではなく foo が定義されてしまい、適切な結果が得られません。この問題を避けるには、ファイルをロードしてから foo の以前の定義を置き換えるしかありません。

しかし、ほかの箇所で定義されている関数を Lisp ファイルで再定義するのは、どちらにしろモジュール化に反し、きれいではありません。

11.9 インライン関数

『インライン関数 (inline function)』を定義するには、 defun ではなく defsubst を使います。インライン関数は普通の関数と同様に動作します。ただし、インライン関数への呼出しをコンパイルすると、関数の定義が呼び出し側に展開されます。

関数をインラインにすると、明示的な呼出しが高速に実行されるようになります。しかし、欠点もあります。まず、柔軟性が低下します。関数の定義を変更しても、すでにインライン展開されている呼出しには再コンパイルするまで古い定義が使われます。関数の再定義の柔軟性は Emacs の重要な特徴であるため、実行速度が本当に重要でなければ、関数をインラインにしないでください。

もう一つの欠点は、大きな関数をインラインにすると、コンパイルされたコードのファイル・サイズとメモリ・サイズが増加することです。インライン関数の速度面での利点は小さな関数の場合にもっとも発揮されるので、通常は大きな関数をインラインにしないでください。

インライン関数が実行するのと同じコードに展開されるマクロを定義することもできます。しかし、マクロには限界があります。マクロは明示的にしか使うことができず、 apply や mapcar などで呼び出すことはできません。また、通常の関数をマクロに書き換えるには手間がかかります (第 12 章「マクロ」 p.163 を参照してください)。通常の関数をインライン関数に書き換えるのは非常に簡単で、 defun を defsubst で置き換えるだけです。インライン関数に書く引数は一度だけ評価されるため、マクロのように本体内での引数の使用回数を気にする必要はありません (12.6.1 節「マクロ引数の反復的評価」 p.166 を参照してください)。

インライン関数を使うと、マクロと同様に、その定義にしたがって同じファイル内のその関数の呼出しが展開されます。

11.10 関数に関するほかの項目

関数の呼出しと定義に関する機能をもつ関数をいくつか次の表に示します。これらの関数は別のところでも触れられていますが、クロスリファレンスを示しておきます。

`apply` 11.5 節「関数の呼出し」p.154 を参照。

`autoload` 13.2 節「自動ロード」p.173 を参照。

`call-interactively`
18.3 節「対話的呼出し」p.262 を参照。

`commandp` 18.3 節「対話的呼出し」p.262 を参照。

`documentation`
21.2 節「説明文字列へのアクセス」p.340 を参照。

`eval` 8.1 節「Eval」p.102 を参照。

`funcall` 11.5 節「関数の呼出し」p.154 を参照。

`ignore` 11.5 節「関数の呼出し」p.154 を参照。

`indirect-function`
8.2.4 節「シンボル関数間接参照」p.105 を参照。

`interactive`
18.2.1 節「`interactive` を使う」p.258 を参照。

`interactive-p`
18.3 節「対話的呼出し」p.262 を参照。

`mapatoms` 7.3 節「シンボルの作成と intern」p.95 を参照。

`mapcar` 11.6 節「マッピング関数」p.155 を参照。

`mapconcat` 11.6 節「マッピング関数」p.155 を参照。

`undefined` 19.7 節「キー検索」p.301 を参照。

12 マクロ

『マクロ (macro)』を使うと新しい制御構文要素などの言語機能を定義することができます。マクロは関数と同じように定義できますが、値を計算する方法を定義するのではなく、値を計算するための別の Lisp 式を計算する方法を定義します。この Lisp 式をマクロの『展開 (expansion)』といいます。

マクロでこのようなことができるのは、関数のように引数の値を処理するのではなく、未評価の引数式を処理するからです。そのため、未評価の引数式やその一部を含んだ式を展開させることができます。

通常の関数でできることを、効率だけを考えてマクロを使っているのであれば、かわりにインライン関数を使うことを検討してください。 11.9 節「インライン関数」 p.160 を参照してください。

12.1 簡単なマクロの例

C の `++` 演算子のような、変数値をインクリメントする Lisp 構文要素を定義したい場合を考えてみてください。`(inc x)` と書いて、`(setq x (1+ x))` の効果をもたせたいわけです。これを行なうためのマクロを以下に示します。

```
(defmacro inc (var)
   (list 'setq var (list '1+ var)))
```

これが `(inc x)` のように呼ばれると、引数 `var` の値は、 `x` の値ではなく、 `x` となります。マクロの本体は展開、すなわち `(setq x (1+ x))` を構築するのにこれを使います。マクロ定義がこの展開を返すと、この式を Lisp が評価し、変数 `x` がインクリメントされます。

12.2 マクロ呼出しの展開

マクロ呼出しは、マクロの名前で始まるリストである点で、ちょうど関数呼出しと似ています。リストの残りの要素はマクロの引数です。

マクロ呼出しの評価は、関数呼出しの評価のように始まりますが、一つだけ決定的に違う点があります。それは、マクロ引数はマクロ呼出しに現われる式そのものであることです。その式はマクロ定義に渡される前には評価されません。逆に、関数の引数には関数呼出しのリストの要素を評価した結果が渡されます。

引数を得ると、 Lisp は関数を起動するようにマクロ定義を起動します。マクロの仮引数はマクロ呼出しの引数値に束縛されます。ただし、 `&rest` 引数のあるときは、マクロ呼出しの引数値のリストに束縛されます。ちょうど関数本体が実行をして値を返すように、マクロ本体も実行をして値を返します。

マクロと関数の決定的な違いの二つめは、マクロ本体の返す値はマクロ呼出しの値ではないことです。そのかわりにマクロ本体の返す値は、マクロ呼出しの値を計算するための別な式で、マクロ『展開 (expansion)』といいます。 Lisp インタプリタは、マクロから帰ると、すぐに返された式を評価します。

展開された式は、通常の方法で評価されるため、別のマクロ呼出しを含んでいてもかまいません。普通は使われませんが、再帰的なマクロ呼出しも誤りではありません。

`macroexpand` を使うと、引数に与えたマクロ呼出しの展開を見ることができます。

macroexpand *form* &optional *environment* Function

この関数は、 *form* がマクロの場合、それを展開します。結果が別なマクロ呼出しの場合、それはマクロ呼出しでない何かになるまで、次々に展開されます。展開した結果が `macroexpand` の戻り値です。 *form* がマクロ呼出しでなければそれをそのまま返します。

`macroexpand` は、 *form* の部分式 (訳注: *form* への引数) を見ません (そういうものを見るマクロ定義はあるかもしれませんが)。たとえそれがマクロ呼出しであっても、`macroexpand` はそういうものを展開しません。

関数 `macroexpand` は、インライン関数の呼出しを展開しません。インライン関数は通常の関数にくらべても理解するのはそれほど難しくないので、普通はその必要がありません。

environment を使って現在定義されているマクロをシャドウするマクロ定義の alist を指定することもできます。バイトコンパイルでこの機能を使います。

```
(defmacro inc (var)
    (list 'setq var (list '1+ var)))
     ⇒ inc

(macroexpand '(inc r))
     ⇒ (setq r (1+ r))

(defmacro inc2 (var1 var2)
    (list 'progn (list 'inc var1) (list 'inc var2)))
     ⇒ inc2

(macroexpand '(inc2 r s))
     ⇒ (progn (inc r) (inc s))  ; ここでは、 inc は展開されない。
```

12.3 マクロとバイトコンパイル

わざわざマクロの展開を計算してから展開を評価するのはどうしてなのか、不思議に思うかもしれません。マクロ本体が必要な結果を直接つくらないのはどうしてなのか? そのわけはコンパイルに関係があります。

コンパイルされる Lisp プログラムの中にマクロ呼出しがあると、 Lisp コンパイラはインタプリタのようにマクロ定義を呼んで、マクロが展開をした結果を受け取ります。しかし、展開された式を評価するのではなく、展開されたコードがプログラム中に直接現われたかのように、コンパイルを行ないます。結果として、コンパイルされたコードは、マクロが想定した値を返すと同時に副作用を生み出しますが、実行速度が損なわれることはありません。もしマクロ本体が戻り値を計算したり、副作用自身を起したりするとしたならば、マクロはうまく動作しません。そのような場合、マクロはコンパイル時に計算されるため、便利ではないでしょう。

コンパイルしたマクロ呼出しがうまくはたらくためには、マクロへの呼出しがコンパイルされるときには、あらかじめ Lisp に定義されていなければなりません。コンパイラにはこれを助ける特別な機能があります。コンパイルされるファイルに `defmacro` があるときには、ファイルの

残りの部分をコンパイルしている間、一時的にこのマクロが定義されます。この機能を使うためには、マクロを使うのと同じファイルでマクロを使う前に定義する必要があります。

ファイルのバイトコンパイル時にはファイルのトップレベルにある `require` 呼出しをすべて実行します。これはファイルを正しくコンパイルするために、 require されたパッケージが必要な場合があるためです (13.4 節「機能」 p.177 参照)。コンパイル中に必要なマクロ定義を有効にする方法の一つに、それらを定義するファイルを require するという方法があります。コンパイルしたプログラムを誰かが作動させたとき、マクロの定義ファイルのロードをさせずにすますには、 `require` 呼出しのまわりに `eval-when-compile` を書きます (14.5 節「コンパイル時に行なわれる評価」 p.186 参照)。

12.4 マクロの定義

Lisp マクロとは CAR が `macro` のリストのことです。 CDR は関数でなければなりません。マクロの展開は、マクロ呼出しへの未評価の引数式のリストに、その関数 (`apply` で) 適用して得ます。

匿名の関数のように、匿名の Lisp マクロというのもあり得ますが、実際には使われません。というのも、 `mapcar` のようなファンクショナルに匿名マクロを渡しても意味がないためです。実際のところ、すべての Lisp マクロが名前をもっていて、通常 `defmacro` 特殊形式で定義されます。

defmacro *name argument-list body-forms...* Special Form

`defmacro` は、 *name* を以下のようなマクロとして定義します。

`(macro lambda` *argument-list* `.` *body-forms*`)`

name の関数セルにこのマクロ・オブジェクトを格納します。 `defmacro` 形式を評価したとき、戻される値は *name* です。しかし、多くの場合はこの値を無視します。

argument-list の形式と意味は関数の場合と同じです。キーワード `&rest` や `&optional` も使えます (11.2.3 節「引数リストの高度な機能」 p.150 参照)。マクロに説明文字列はあってもかまいませんが、マクロを対話的に呼ぶことができないため、 `interactive` 宣言は無視されます。

12.5 逆引用符

マクロを使って、定数と非定数の部分が混ぜたものから大きなリスト構造を構築する必要があることがよくあります。これを書くのをたやすくするため、マクロ '`' (『逆引用符 (backquote)』と呼ぶことが多い) を使います。

逆引用符ではリストを quote しますが、さらに、リストの要素を選択的に評価することができます。最も単純な使い方をした場合、逆引用符は特殊形式 `quote` と同じはたらきをします (8.3 節「quote する」 p.109 参照)。たとえば、以下に示す二つの形式は、同じ結果になります。

```
`(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
```

```
'(a list of (+ 2 3) elements)
     ⇒ (a list of (+ 2 3) elements)
```

逆引用符の引数の中に特別な標識 ',' を入れると、その値は定数でないことを表わします。逆引用符は ',' の引数を評価し、リスト構造にその値を入れます。

```
(list 'a 'list 'of (+ 2 3) 'elements)
     ⇒ (a list of 5 elements)
`(a list of ,(+ 2 3) elements)
     ⇒ (a list of 5 elements)
```

特別な標識 ',@' を用いることで、評価した値を結果のリストに『組み込む (splice)』こともできます。組み込まれたリストの要素は、結果のリストのほかの要素と同じレベルの要素になります。'`' を使わない等価なコードは、往々にして読みにくくなります。例をあげます。

```
(setq some-list '(2 3))
     ⇒ (2 3)
(cons 1 (append some-list '(4) some-list))
     ⇒ (1 2 3 4 2 3)
`(1 ,@some-list 4 ,@some-list)
     ⇒ (1 2 3 4 2 3)

(setq list '(hack foo bar))
     ⇒ (hack foo bar)
(cons 'use
  (cons 'the
    (cons 'words (append (cdr list) '(as elements)))))
     ⇒ (use the words foo bar as elements)
`(use the words ,@(cdr list) as elements)
     ⇒ (use the words foo bar as elements)
```

バージョン 19.29 より前の Emacs の '`' では、逆引用符構文要素全体のまわりに余分な括弧を必要とする異なった構文を使っていました。同様に、各 ',' と ',@' 置換も ',' や ',@' とそれに後続する式のまわりに余分な括弧を必要としていました。旧構文では '`'、','、',@' と後続する式の間に、空白が必要でした。

旧構文も受けつけられますが、旧バージョンの Emacs との互換性を例外にすれば、これはもはや推奨されません。

12.6 マクロの利用でよく見られる問題点

マクロの展開は直感的でない結果を引き起こす、という基本的事実があります。この節では、トラブルを引き起こす重要な結果と、トラブルを避けるためのルールについて述べます。

12.6.1 マクロ引数の反復的評価

マクロを定義する際は、展開が実行されるときに、引数が評価される回数について気をつけないといけません。(これを使うと繰返しに便利な) 次のマクロで問題を説明します。このマクロは、 Pascal に見られるような単純な "for" ループを書けるようにします。

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  (list 'let (list (list var init))
        (cons 'while (cons (list '<= var final)
                           (append body (list (list 'inc var)))))))
⇒ for

(for i from 1 to 3 do
   (setq square (* i i))
   (princ (format "\n%d %d" i square)))
↦
(let ((i 1))
  (while (<= i 3)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))

     ⊣1       1
     ⊣2       4
     ⊣3       9
⇒ nil
```

(マクロの引数 `from`、 `to`、 `do` は「シンタックス・シュガー」 "syntactic sugar" であり、すべて無視されます。これはマクロの呼出し時に (`from`、 `to`、 `do` のような) 余計な単語 (noise words) を、その引数の位置に書くことができるようにするための工夫です。)

逆引用符を使用して単純にした等価な定義を示します。

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple \"for\" loop.
For example, (for i from 1 to 10 do (print i))."
  `(let ((,var ,init))
     (while (<= ,var ,final)
       ,@body
       (inc ,var))))
```

形式の定義の両方とも (逆引用符の有無にかかわらず)、繰り返しごとに `final` を評価するという欠点をもっています。 `final` が定数であれば問題はありません。より複雑な形式の場合、たとえば、 `(long-complex-calculation x)` のような場合、これはかなり実行を遅らせてしまう可能性があります。もし `final` が副作用をもつなら、 2 回以上実行するのは多分誤りです。

評価の繰り返しがマクロの意図した目的である場合を除き、引数式を正確に一度だけ評価するような展開を生成することで、よくできたマクロ定義はこの問題を回避しています。 `for` マクロにたいする正しい展開を以下に示します。

```
(let ((i 1)
      (max 3))
  (while (<= i max)
    (setq square (* i i))
    (princ (format "%d      %d" i square))
    (inc i)))
```

この展開を生成するマクロ定義です。

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  `(let ((,var ,init)
         (max ,final))
     (while (<= ,var max)
       ,@body
       (inc ,var))))
```

残念ながら、これによりまた、別の問題が発生します。

12.6.2 マクロ展開内の局所変数

新しい `for` の定義には、新しい問題があります。ユーザが予期しない `max` という局所変数を導入しています。そのため以下に示す例でトラブルが発生します。

```
(let ((max 0))
  (for x from 0 to 10 do
    (let ((this (frob x)))
      (if (< max this)
          (setq max this)))))
```

`for` 本体の中からの `max` への参照は、ユーザによる `max` の束縛を参照することが想定されますが、実際は `for` による束縛をアクセスします。

これを直すには、 `max` のかわりに intern されていない (unintern されている) シンボルを使います (7.3 節「シンボルの作成と intern」p.95 参照)。 intern されてない (unintern されている) シンボルは、ほかのシンボルのように束縛、参照できますが、 `for` がこれを作ったので、ユーザのプログラムにもとから現われていることはありえません。 intern されてない (unintern されている) ので、ユーザが後でプログラムの中に入れる方法はありません。 `for` によって置かれた場所以外でそれが現われることはありません。このようにはたらく `for` の定義を示します。

```
(defmacro for (var from init to final do &rest body)
  "Execute a simple for loop: (for i from 1 to 10 do (print i))."
  (let ((tempvar (make-symbol "max")))
    `(let ((,var ,init)
           (,tempvar ,final))
       (while (<= ,var ,tempvar)
         ,@body
         (inc ,var)))))
```

通常の展開に現われる `max` という intern されたシンボルのかわりに、 `max` という intern されてない (unintern されている) シンボルを作り、展開に入れています。

12.6.3 展開内のマクロ引数の評価

展開を計算しているときに `eval` (8.1 節「Eval」p.102 参照) を呼ぶなどしてマクロ引数式を評価すると、別な問題がおこります。もし引数がユーザの変数を参照しようとすると、ユーザがたまたまマクロ引数の一つと同じ名前の変数を使おうとするときに、トラブルになります。マクロ本体の内側では、マクロ引数の束縛がこの変数のもっとも局所的な束縛になるので、評価されている形式の内側の参照は、それを参照します。例をあげます。

```
(defmacro foo (a)
  (list 'setq (eval a) t))
     ⇒ foo
```

```
(setq x 'b)
(foo x) ↦ (setq b t)
     ⇒ t                        ; で b を設定した。
;; ところが
(setq a 'c)
(foo a) ↦ (setq a t)
     ⇒ t                        ; しかしこれは c でなく
;; a のセットをした。
```

ユーザの変数が a という名か x という名かによって違いがでます。 a はマクロの仮引数の a と衝突しているからです。

マクロ定義で eval を呼ばないもう一つの理由は、プログラムをコンパイルするとおそらくあなたの思うとおりにはうごかないだろうということです。バイトコンパイラはプログラムのコンパイル中にマクロ定義を実行しますが、この時点では (eval を通じてアクセスしようとした) プログラム中で行なわれるはずの計算が行なわれないため、ローカルな変数への束縛は行なわれていません。

式の実行時における値で安全に動作させるためにはマクロ展開に式を代入してください。そうすると、展開の実行の一部として値が計算されます。

12.6.4 何回マクロが展開されるか?

インタープリトされる関数では、マクロは評価されるたびにマクロ呼出しが展開されるのにたいし、コンパイルされる関数では (コンパイル時に) 1 回だけ展開されるため、問題が発生することがあります。マクロ定義が副作用をもつなら、マクロが何回展開されるかによって動作が異なってきます。

とりわけオブジェクトを構築するのは副作用の一つです。マクロが 1 回呼び出されるとオブジェクトは 1 回だけ構築されます。いいかえると、マクロが実行されるたびに同じオブジェクトの構造が使われます。インタープリトされた操作では、マクロは毎回展開されマクロは毎回新しいオブジェクトの集まりを生成します。通常はそれで何の問題も起こりません。オブジェクトは共有されるかいなかにかかわらず、同じ内容をもっています。しかし、もし前後のプログラムがオブジェクトに副作用をあたえるならば、共有されているかどうかによって動作が異なってきます。例をあげます。

```
(defmacro empty-object ()
  (list 'quote (cons nil nil)))

(defun initialize (condition)
  (let ((object (empty-object)))
    (if condition
        (setcar object condition))
    object))
```

initialize がインタープリトされるなら、新しいリスト (nil) が、 initialize が呼び出されるたびに構築されます。よって、複数の呼出しの間に副作用が残ることはありません。

initialize がコンパイルされるなら、マクロ empty-object はコンパイル中に展開され、一つの「定数」(nil) を生成し、 initialize が呼ばれるごとに再利用されます。

このような病的な例を避ける一つの方法は、 empty-object をメモリに割りつける構文要素ではなく、おもしろい種類の定数とみなすことです。'(nil) のような定数に setcar を使うことはないでしょう。そうすれば empty-object でそれを使うことも当然ないでしょう。

13 ロード

Lisp コードのファイルをロードするとは、その内容を Lisp 環境に Lisp オブジェクトの形式で持ち込むことを意味します。 Emacs はそのファイルを見つけて開き、テキストを読み込み、各形式を評価し、その後ファイルを閉じます。

ロード関数群は、 `eval-current-buffer` 関数がバッファ内の全ての式を評価するのと同じように、ファイル内の全ての式を評価します。違いは、ロード関数群が読み込んで評価するのは、ディスク上にあるファイル中のテキストであり、 Emacs バッファ内のテキストではないことです。

ロードされたファイルは、ソース・コードかバイトコンパイル・コードかのいずれかの形の Lisp 式を含ねばなりません。ファイル中の各形式は、『トップレベルの形式 (top-level form)』と呼ばれます。ロード可能なファイル中の形式に特別なフォーマットは存在しません。ファイル中のどの形式も、同じように、直接バッファにタイプしてそこで評価ができます (実際、ほとんどのコードはこの方法で検査されます)。ほとんどの場合、それら形式は関数定義と変数定義です。

Lisp コードを含むファイルはしばしば『ライブラリ (library)』と呼ばれます。それゆえ、「Rmail ライブラリ」は Rmail モードのためのコードを含むファイルであるといえます。同様に、「Lisp ライブラリ・ディレクトリ」は、 Lisp コードを含むファイルのディレクトリであるといえます。

13.1 プログラムでロードする方法

Emacs Lisp にはロードのためのインタフェースがいくつかあります。たとえば、 `autoload` はあるファイル中の関数のために、名札オブジェクトを作成します。自動ロード関数を呼び出そうとすると、その関数の実際の定義を得るために、そのファイルがロードされます (13.2 節「自動ロード」 p.173 参照)。 `require` はそのファイルがまだロードされてないならば、ロードします (13.4 節「機能」 p.177 参照)。結局、これらの機能全ては `load` 関数を呼び出してその作業を行ないます。

load *filename* &optional *missing-ok nomessage nosuffix* Function

この関数は、 Lisp コード・ファイルを見つけて開き、ファイル中の全ての形式を評価し、ファイルを閉じます。

ファイルを見つけるのに、 `load` は最初に ‘*filename*`.elc`’ という名前のファイルを探します。つまり、 *filename* に ‘`.elc`’ を追加した名前のファイルを探します。その名前のファイルがあれば、それがロードされます。その名前のファイルがないならば、 `load` は ‘*filename*`.el`’ という名前のファイルを探します。そのファイルがあれば、それがロードされます。最後に、これらいずれの名前も見つからなかった場合に、 `load` は *filename* に何もつけない名前のファイルを探し、あればそれをロードします (`load` 関数の *filename* の探し方は、そんなに賢くありません。 ‘`foo.el.el`’ という意地悪な名前のファイルがあるときに、 `(load "foo.el")` を評価すると、まさにこれを見つけてしまいます)。

オプション引数 *nosuffix* が非 `nil` ならば、接尾辞 '`.elc`' や '`.el`' は試しません。この場合は、欲しいファイル名を正確に指定する必要があります。

filename が、たとえば '`foo`' や '`baz/foo.bar`' のように相対ファイル名ならば、`load` は変数 *load-path* を使ってそのファイルを探します。`load-path` のリストの各ディレクトリに *filename* を追加して、名前が一致した最初のファイルをロードします。現在のデフォルト・ディレクトリは、`load-path` に指定されたときにかぎり試されます。`nil` が現在のデフォルト・ディレクトリを表します。`load` は三つの接尾辞候補全てについて、`load-path` の最初のディレクトリで試し、次に 2 番目のディレクトリで試し、次のディレクトリで試し、としていきます。

'`foo.elc`' が '`foo.el`' よりも古いという警告を受け取ったなら、'`foo.el`' のリコンパイルを考えるべきでです。第 14 章「バイトコンパイル」p.181 を参照してください。

nomessage が `nil` ならば、ロード中、'`Loading foo...`' と '`Loading foo...done`' といったメッセージがエコー領域に表れます。

ファイル・ロード中にハンドルされないエラーが起こると、ロードは打ち切られます。ロードが `autoload` のためになされたものであった場合、そのロード中に行なわれた関数定義はいずれも取り消されます。

`load` がロードすべきファイルを見つけられなかった場合、`file-error` エラーを ('`Cannot open load file` *filename*' とともに) 通知するのが普通です。しかし、*missing-ok* が非 `nil` ならば、`load` は単に `nil` を返します。

変数 `load-read-function` を使って `load` が式を読み込むために `read` の代わりに使う関数を指定することもできます。以下を参照してください。

`load` はファイルのロードが成功した場合、`t` を返します。

load-path User Option

この変数の値は、`load` でファイルをロードするときに検索を行なうディレクトリのリストです。各要素は文字列 (ディレクトリ名であること) あるいは `nil`(現在の作業ディレクトリを表す) です。`load-path` の値は、環境変数 `EMACSLOADPATH` が設定されていればその値で初期化されます。さもなくば、Emacs 構築時に '`emacs/src/paths.h`' で指定したデフォルト値になります。

`EMACSLOADPATH` の構文は、`PATH` に使うものと同じです。'`:`'(オペレーティング・システムによっては '`;`') でディレクトリ名を分離し、'`.`' を現在のデフォルト・ディレクトリに使用します。`EMACSLOADPATH` 変数を `csh` の '`.login`' ファイルで設定する例を挙げます:

```
setenv EMACSLOADPATH .:/user/bil/emacs:/usr/lib/emacs/lisp
```

`sh` で設定する方法を挙げます:

```
export EMACSLOADPATH
EMACSLOADPATH=.:/user/bil/emacs:/usr/local/lib/emacs/lisp
```

'`.emacs`' ファイルで、デフォルトの `load-path` の前にディレクトリをいくつか追加するコードを挙げます:

```
(setq load-path
      (append (list nil "/user/bil/emacs"
                    "/usr/local/lisplib"
                    "~/emacs")
              load-path))
```

本例では、パスは現在の作業ディレクトリを最初に、続いて '/user/bil/emacs' ディレクトリで、次に '/usr/local/lisplib' ディレクトリで、次に個人の '~/emacs' ディレクトリで、それらの後で Lisp コードの標準ディレクトリを探します。

コマンド行オプション '-l' あるいは '-load' で、 Emacs 起動の一部としてロードする Lisp ライブラリを指定します。このファイルは現在のディレクトリにあるかもしれないので、 Emacs 18 は一時的に現在のディレクトリを load-path の前に追加して、そのファイルを見つけられるようにしていました。新しい Emacs の版では、 load-path を変更せずに、こういった現在のディレクトリのファイルも見つけることができます。

Emacs のダンプでは、 load-path に特別な値を使用します。ダンプ終了時点で load-path の値が変更されない (即ち、同じ特殊な値のまま) ならば、ダンプされた Emacs は通常の load-path の値に起動時に切替えます。しかし、ダンプ終了時点で load-path がそれ以外の値だったら、その値がダンプされた Emacs の実行時にも使用されます。

そのため、 'site-init.el' や 'site-load.el' 内でいくつかのライブラリをロードするために一時的に load-path を変更したいならば、 load-path を load 呼出し周辺で let によって、ローカルに束縛しなければなりません。

load-in-progress Variable

この変数は、 Emacs がファイルのロード中ならば非 nil で、それ以外ならば nil となります。

load-read-function Variable

この変数は、 load と eval-region について、 read の代わりに使用する代替式読込み関数を指定します。関数は引数を一つ、 read と全く同じように受け取らねばなりません。

通常この変数の値は nil で、ロード関数群が read を使わなければならないことを意味します。

load が Emacs 構築時にどのように使われるかについて知るには、 B.1 節「Emacs を作る」 p.645 を参照してください。

13.2 自動ロード

『自動ロード (autoload)』機能によって、関数やマクロを、 Lisp 内で既知ではあるが、それが定義されているファイルのロードは延期する、ということができます。その関数への最初の呼出しで、適切なファイルを自動的に読み込んで、実定義とほかの関連コードをインストールし、その後、実定義をあたかも元々からあったかのように実行します。

自動ロード関数を設定するには、二つの方法があります。 autoload を呼ぶ方法と、特殊な「魔法」のコメントをソースの実定義の前に書く方法です。自動ロードのための autoload は低

水準のプリミティブですが、任意の Lisp プログラムが `autoload` を任意の時点で呼び出すことができます。魔法のコメントは、それ自体は何もしません。しかしそれは、 Emacs 構築時に実行されるように `autoload` 呼出しを構築し調整する、コマンド `update-file-autoloads` の案内役として働きます。魔法のコメントは関数を自動ロード化する最も簡便な方法ですが、 Emacs とともにインストールされるパッケージにしか使えません。

autoload *function filename* &optional *docstring interactive type* Function

この関数は、 *function* という名前の関数 (あるいはマクロ) を、 *filename* から自動的にロードするように定義します。文字列 *filename* は *function* の実定義を取り出すためにロードするファイルを指定します。

引数 *docstring* はその関数の説明文字列です。通常、これは関数定義自身の説明文字列と同じです。 `autoload` 呼出しで説明文字列を指定することで、関数の実定義をロードしなくても説明文字列を見ることができるようになります。

interactive が非 `nil` ならば、関数は対話的に呼び出すことができます。これでその関数の実定義をロードしなくても、 `M-x` の補完が働くようになります。完全な対話引数の指定はここでは与えなくて構いません。ユーザが実際に *function* を呼び出すまで、引数は不要であり、もし呼び出したなら、実定義をロードするまでです。

マクロやキーマップも普通の関数同様に自動ロードできます。 *function* が実際はマクロならば、 *type* を `macro` として指定してください。 *function* が実際はキーマップならば、 *type* を `keymap` として指定してください。 Emacs のさまざまな部分で、実定義をロードせずにこの情報を知る必要があります。

自動ロード・キーマップは、キー検索の際に、前置キーのバインドがシンボル *function* であったならば、自動的にロードします。それ以外の種類のキーマップへのアクセスでは、自動ロードは起こりません。特に、 Lisp プログラムが変数の値からキーマップを取り出し、 `define-key` を呼び出す時には、起こりません。変数名がシンボル *function* と同一であっても起こりません。

訳注: キーマップの指定の仕方の例。

```
;; (autoload 'hoge-prefix "hoge.el" "Hoge map" t 'keymap)
;; (local-set-key "\C-c\C-c" 'hoge-prefix)

;; contents of file hoge.el follows.
(defvar hoge-map nil)
(if hoge-map
    nil
  (setq hoge-map (make-keymap))
  (define-key hoge-map "\C-c"
    (lambda () (interactive) (message "hoge!"))))
(fset 'hoge-prefix hoge-map)
```

function がすでに非 void な関数定義を持ち、それが自動ロード・オブジェクトではない場合、 `autoload` は何もしないで `nil` を返します。 *function* の関数セルが void な場合、あるいはそれがすでに自動ロード・オブジェクトならば、このような自動ロード・オブジェクトとして定義されます:

(autoload *filename docstring interactive type*)

たとえば、

```
(symbol-function 'run-prolog)
     ⇒ (autoload "prolog" 169681 t nil)
```

この場合、"prolog"はロードするファイルの名前で、169681 は'emacs/etc/DOC' ファイル中の説明文字列を参照し (21.1 節「説明文字列の基礎」p.339 参照)、t は関数が対話的であることを意味し、nil でそれがマクロでもキーマップでもないことを示します。

自動ロード・ファイルは普通ほかの定義も含み、一つ以上の機能を要求 / 提供します。ファイルが完全にロードされなかった場合 (その内容の評価で出たエラーのため)、ロード中に行なった、いかなる関数定義も provide の呼出しも、取り消されます。こうして、このファイルから自動ロードされるどの関数でも次の呼出しでは、ファイルのロードをやり直すことを保証します。もしこうなっていなかったならば、ファイル中のいくつかの関数が、定義されたようには見えるかもしれないが、ファイルの後ろの方にあったためにうまくロードされなかったサブルーチンの欠如により、それらは正しく動作しない、ということが起こり得ます。

自動ロードされたファイルが望みの Lisp 関数やマクロの定義に失敗すると、"Autoloading failed to define function *function-name*"というデータでエラーが通知されます。

魔法の自動ロード・コメントは、自動ロード可能なソース・ファイル中の関数の実定義の直前に';;;###autoload' のように行に単独で書きます。コマンド M-x update-file-autoloads は、対応する autoload 呼出しを'loaddefs.el' に書き込みます。'loaddefs.el' は Emacs の構築中にロードされるので、autoload が呼び出されます。M-x update-directory-autoloads はさらに強力です。これは現在のディレクトリ中の全てのファイルについて、自動ロードを更新します。

同じ魔法のコメントで任意の形式を'loaddefs.el' にコピーできます。魔法のコメントに続く形式が関数定義でないならば、それはそのままコピーされます。魔法のコメントを、そのファイルがロードされるときには実行せずに、構築時に実行するという形式を作るために使うこともできます。こうするには、形式を魔法のコメントと同じ行に書きます。それはコメント中にあるので、ファイルをロードしたときには何もしません。しかし、update-directory-autoloads はそれを'loaddefs.el' にコピーするので、Emacs 構築中に実行されます。

以下の例は、魔法のコメントで doctor の自動ロードがどのように準備されるかを示します:

```
;;;###autoload
(defun doctor ()
  "Switch to *doctor* buffer and start giving psychotherapy."
  (interactive)
  (switch-to-buffer "*doctor*")
  (doctor-mode))
```

これが'loaddefs.el' に生成されます:

```
(autoload 'doctor "doctor"
  "\
Switch to *doctor* buffer and start giving psychotherapy."
  t)
```

二重引用符直後に続くバックスラッシュと改行は、'loaddefs.el' のようにプリロードされる Lisp ファイル中だけで使う、お決まりの手法です。これによって make-docfile に、この説明

文字列は 'etc/DOC' ファイルに置くとのだということを示します。 B.1 節「Emacs を作る」p.645 を参照してください。

13.3 ロードの反復

1 回の Emacs セッション中で、一つのファイルを複数回ロードしても構いません。たとえば、バッファで編集して、関数定義を書き直して再インストールした後に、元の版に戻りたいと思うかもしれません。そうするには、その関数があったファイルを再ロードすればよいのです。

ファイルをロードあるいは再ロードするときは、 load や load-library 関数がコンパイルされてないファイルではなく、類似の名前のバイトコンパイルされたファイルを自動的にロードするということを覚えておいてください。保存して再インストールしたいと思ったファイルを書き直した場合には、必要ならそれをバイトコンパイルしてください。さもないと不注意にも古いバイトコンパイルされたファイルを、新しいコンパイルされてないファイルの代わりに再ロードしてしまうかも知れません！

Lisp ライブラリ・ファイル中に形式を書く時は、そのファイルが複数回ロードされるかもしれないことを念頭に置いてください。たとえば、変数定義が defvar か defconst かというのは、ライブラリが再ロードされたときにその変数を再初期化するのが望ましいか否かに依存します。つまり defconst は再初期化しますが、 defvar はこれを行ないません (10.5 節「グローバル変数の定義」p.131 を参照してください)。

要素を alist に追加する最も簡単な方法は、こうなります:

```
(setq minor-mode-alist
      (cons '(leif-mode " Leif") minor-mode-alist))
```

しかしこれではライブラリが再ロードされた場合、要素が重複して追加されます。この問題を避けるには、次のように書きます:

```
(or (assq 'leif-mode minor-mode-alist)
    (setq minor-mode-alist
          (cons '(leif-mode " Leif") minor-mode-alist)))
```

リストに要素を重複せずに追加するには、 add-to-list を使います。 (10.7 節「変数の値を変更する方法」p.135 参照)。

ライブラリがすでにロードされたか否かを明示的に検査したくなることもあるでしょう。ライブラリ中で、それが以前ロードされたか否かを検査する一つの方法をここに示します:

```
(defvar foo-was-loaded)

(if (not (boundp 'foo-was-loaded))
    execute-first-time-only)

(setq foo-was-loaded t)
```

ライブラリが provide を使って名前つき機能を提供しているならば、 featurep でそのライブラリがロードされたか否かを検査できます。

13.4 機能

provide と require とはファイルを自動的にロードする autoload の代わりになります。これらは、名前つき『機能 (feature)』に基づいて働きます。自動ロードは特定の関数の呼出しによって起動されますが、機能はほかのプログラムがはじめてそれを名前で尋ねたときにロードされます。

機能名は関数や変数などをまとめたものを代表するシンボルです。それらを定義するファイルはその機能を『提供 (provide)』しなくてはなりません。それらを使うほかのプログラムはその機能を『要求 (require)』することで、定義されていることを保証するかも知れません。要求した機能がまだロードされてなければ、定義ファイルをロードします。

機能が存在することを要求するには、機能名を引数として require を呼び出します。require はグローバル変数 features を覗いて、望みの機能がすでに提供されているか調べます。なければ、その機能を適切なファイルからロードします。ロードされるファイルは features にその機能を追加するために、 provide をトップレベルで呼び出す必要があります。それに失敗すると、 require はエラーを通知します。

機能は普通それを提供するファイルに因んで名づけられます。そうすると、 require にファイル名を与える必要がなくなります。

たとえば、 'emacs/lisp/prolog.el' においては、 run-prolog の定義が次のコードを含みます:

```
(defun run-prolog ()
  "Run an inferior Prolog process, input and output via buffer *prolog*."
  (interactive)
  (require 'comint)
  (switch-to-buffer (make-comint "prolog" prolog-program-name))
  (inferior-prolog-mode))
```

式 (require 'comint) で、 'comint.el' がまだロードされてなければ、それをロードします。これによって、 make-comint が定義されていることを保証します。

```
(provide 'comint)
```

これは comint をグローバルな features リストに追加し、そうすることで (require 'comint) に、もう何もしなくてよいことを知らせます。

require をファイルのトップレベルで使うと、ロードしたときと同様、そのファイルのバイトコンパイル時 (第 14 章「バイトコンパイル」 p.181 参照) にも効果を発揮します。要求されたパッケージがバイトコンパイラが知るべきマクロを含む時には、このようにします。

require のトップレベルでの呼出しはバイトコンパイル時に評価されますが、 provide 呼出しは違います。それゆえ、定義ファイルに同じ機能についての provide に続く require を含めることで、それがバイトコンパイル以前にロードされることを保証できます。次の例のようになります。

```
(provide 'my-feature)   ; バイトコンパイラは無視し、
                        ;    load は評価する。
(require 'my-feature)   ; バイトコンパイラが評価する。
```

コンパイラは `provide` を無視し、 `require` を処理することで問題のファイルをロードします。問題のファイルをロードすると、 `provide` 呼出しが実行されるので、続く `require` 呼出しはロード中何もしなくてよくなります。

provide *feature* Function

この関数は、 *feature* が今ロードされたことを、あるいはロードされつつあることを現在の Emacs セッションに告知します。 *feature* に関連する機能がほかの Lisp プログラムにも利用可能になった、あるいはなる、という意味です。

`provide` 呼出しの直接の効果は、まだ入ってないならば *feature* を、リスト `feature` の先頭に追加することです。引数 *feature* は実際シンボルでなければなりません。 `provide` は *feature* を返します。

```
features
     ⇒ (bar bish)

(provide 'foo)
     ⇒ foo
features
     ⇒ (foo bar bish)
```

ファイルが自動ロードを満足するためにロードされ、それが内容の評価中にエラーでストップした場合、そのロード中に発生した関数定義や `provide` 呼出しは、全て取り消されます。 13.2 節「自動ロード」p.173 を参照してください。

require *feature* &optional *filename* Function

この関数は、 *feature* が現在の Emacs セッションに存在するか否かをチェックします ((`featurep` *featurep*) を使います。以下参照)。存在しないならば、 `require` は *filename* を `load` でロードします。 *filename* が与えられないならば、シンボル *feature* の名前をロードのファイル名として使います。

ファイルのロードで *feature* の提供に失敗すると、 `require` は '`Required feature` *feature* `was not provided`' というエラーを通知します。

featurep *feature* Function

この関数は、 *feature* が現在の Emacs セッションに提供されている (即ち、 *feature* が `features` のメンバである) ならば、 `t` を返します。

features Variable

この変数の値は現在の Emacs セッションにロードされた機能シンボルのリストです。各シンボルは `provide` 呼出しによって本リストに置かれます。 `features` リスト内の要素順序に意味はありません。

13.5 アンロード

ライブラリによってロードされた関数や変数を捨てて、ほかの Lisp オブジェクトのためにメモリを再生することができます。そうするには、関数 `unload-feature` を使います:

unload-feature *feature* &optional *force* Command

本コマンドは機能 *feature* を提供したライブラリをアンロードします。このコマンドは、ライブラリによって `defconst`, `defvar`, `defun`, `defmacro`, `defsubst`, `defalias` で定義された全ての関数、マクロ、変数を未定義化します。その後、以前これらシンボルに関連づけられていた自動ロードを全て復元します (ロードはこれらをシンボルの `autoload` 属性に保存しています)。

通常 `unload-feature` は、ほかのロードされたライブラリが依存するようなライブラリのアンロードは拒否します (ライブラリ *a* がライブラリ *b* に依存するのは、 *a* が *b* の `require` を含む時です)。オプション引数 *force* が非 `nil` ならば、依存性を無視して任意のライブラリをアンロードできます。

`unload-feature` 関数は Lisp で書かれています。そしてその動作は、変数 `load-history` に基づいています。

load-history Variable

この変数の値はライブラリ名に、それが定義する関数と変数の名前、それが提供する機能、それが要求する機能、を結びつけた alist です。

各要素がリストで、一つのライブラリを表します。そのリストの CAR はライブラリの名前の文字列です。リストの残りは以下のオブジェクトから成ります:

- このライブラリが定義したシンボル。
- `(require . features)` の形式のリストで、要求された機能を表します。
- `(provide . features)` の形式のリストで、提供された機能を表します。

`load-history` の値に CAR が `nil` である要素が一つあるかも知れません。この要素は、ファイルに対応づいていないバッファで行なった `eval-buffer` による定義であることを表しています。

コマンド `eval-region` は `load-history` を更新しますが、読み込んだファイルに関しての要素を置き換えるのではなく、その要素に定義されたシンボルを追加する、というやり方をします。

13.6 ロードのためのフック

特定のライブラリがロードされたときに、あるコードを実行するように、 `eval-after-load` を呼び出すことで要求することができます。

eval-after-load *library form* Function

この関数は、ライブラリ *library* がロードされたら、ロード終了時に *form* を評価するように調整します。ライブラリ *library* がすでにロードされていたならば、 *form* を直ちに評価します。

ライブラリ名 *library* は `load` の引数と完全に一致しなければなりません。 `load-path` を検索してインストールされたライブラリが見つかったときに正しい結果を得るには、*library* 内にいかなるディレクトリ名も含めてはなりません。

form 内でエラーが起きてもロードは取り消されませんが、 *form* の残り部分は実行されません。

一般的には、よく設計された Lisp プログラムは本機能を使ってないはずです。 Lisp ライブラリとの、きれいでモジュール式の対話法は、 (1) ライブラリの変数 (ライブラリ外での使用を意味する) を検査し設定する、 (2) ライブラリの関数を呼び出す、です。 (1) をやりたいときは、直ちに行なうことができます (ライブラリがロードされるのを待つ必要がない)。 (2) を行うには、ライブラリをロードする必要があります (`require` が望ましい)。

しかし、リリース用プログラムの設計標準に合わせる必要はないと感じるのであれば、 `eval-after-load` を自分の個人的カスタマイズに使用しても構いません

after-load-alist Variable

特定のライブラリがロードされたならば、そのとき、評価すべき式の alist です。各要素は、こんな感じになります:

(*filename forms*...)

関数 `load` は、 `eval-after-load` を実現するために `after-load-alist` をチェックします。

14 バイトコンパイル

GNU Emacs Lisp には『コンパイラ (compiler)』があります。これは、 Lisp で書いた関数を『バイトコード (byte-code)』と呼ばれる、より効率的に実行できる特殊な表現に変換します。コンパイラは Lisp 関数定義をバイトコードで置き換えます。バイトコードになった関数が呼び出されると、その関数定義は『バイトコード・インタプリタ (byte-code interpreter)』によって評価されます。

バイトコンパイルされたコードはマシンのハードウェアによって直接実行されるのではなく、バイトコード・インタプリタによって評価されるため (本当のコンパイルされたコードはハードウェアによって直接実行されるのですが)、バイトコードは再コンパイルなしに、あるマシンから別のマシンに移して動作させることが可能です。しかし、バイトコードは本当のコンパイルされたコードほどは速くありません。

一般的にどのバージョンの Emacs においても、それよりも少し前のバージョンの Emacs で作ったバイトコンパイル・コードであれば実行することができます。ただし、その逆は成り立ちません。特に Emacs バージョン 19.29 でコンパイルされたプログラムはそれ以前のバージョンでは動作しません。 14.3 節「説明文字列とコンパイル」p.184 を参照してください。 19.29 より前のバージョンでコンパイルしたファイルは、修飾ビット (modifier bits) の立った文字定数が入っている場合、動作しないことがあります。これは、 Emacs 19.29 になって修飾ビットの番号がつけ直されたためです。

バイトコンパイル中に起きたエラーを調べる方法は 15.3 節「コンパイル時の問題のデバッグ」p.202 を参照してください。

14.1 バイトコンパイルされたコードの性能

バイトコンパイルされた関数は、 C 言語で書かれているプリミティブ関数ほどは効率的ではありませんが、 Lisp で書いたままのものよりはかなり速く実行されます。例を挙げましょう。

```
(defun silly-loop (n)
  "N 回繰り返して、ループの前後の時間を返す。 "
  (let ((t1 (current-time-string)))
    (while (> (setq n (1- n))
              0))
    (list t1 (current-time-string))))
⇒ silly-loop

(silly-loop 100000)
⇒ ("Fri Mar 18 17:25:57 1994"
    "Fri Mar 18 17:26:28 1994")  ; 31 秒

(byte-compile 'silly-loop)
⇒ [コンパイルされたコードは省略]

(silly-loop 100000)
⇒ ("Fri Mar 18 17:26:52 1994"
    "Fri Mar 18 17:26:58 1994")  ; 6 秒
```

この例では、インタプリタで実行したコードは実行に 31 秒かかっていますが、バイトコンパイルしたコードでは 6 秒しかかかっていません。この結果は代表例にすぎませんので、実際に実行してみるとかなり違ったものになるでしょう。

14.2 バイトコンパイルを行なう関数

個々の関数やマクロ定義のコンパイルは、 `byte-compile` という関数で行なうことができます。ファイル全体のコンパイルは `byte-compile-file` で行ない、複数のファイルのコンパイルは `byte-recompile-directory` や `batch-byte-compile` で行ないます。

バイトコンパイルを実行すると、 ‘`*Compile-Log*`’ というバッファに警告メッセージが表示されることがあります。これは、プログラム中で問題となりそうなものを指摘するもので、必ずしもエラーではありません。

マクロを使っているコードをバイトコンパイルするときは注意してください。マクロの呼出しはコンパイルされるときに展開されてしまいます。そこで、正しくコンパイルするためには、マクロはコンパイル時にすでに定義されていなければいけません。詳細は 12.3 節「マクロとバイトコンパイル」p.164 を参照してください。

通常、ファイルをコンパイルしても、そのファイルの内容が評価されたり、ファイルがロードされたりすることはありません。しかし、ファイルのトップ・レベルにある `require` はいずれも実行されます。必要なマクロ定義をコンパイル時に使えるようにする確実な方法の一つはマクロを定義しているファイルを require しておくことです (13.4 節「機能」 p.177 参照)。コンパイルしたプログラムを動作させる際にマクロ定義ファイルをロードしないようにするには、`require` の呼出しを `eval-when-compile` で囲んでプログラムを書いてください (14.5 節「コンパイル時に行なわれる評価」 p.186 参照)。

byte-compile *symbol* Function

この関数は *symbol* の関数定義をバイトコンパイルし、その元の定義をコンパイルしたものに置き換えます。 *symbol* の関数定義はその関数の実際のコードでなくてはいけません。つまり、コンパイラはほかのシンボルへ間接的に関数定義をたどることはしません。`byte-compile` は新たにコンパイルされた *symbol* の関数定義を返します。

symbol の関数定義がバイトコード関数オブジェクトの場合、 `byte-compile` は何もせず `nil` を返します。 Lisp では、すべてのシンボルはそれぞれただ一つの関数定義しか保持しません。そして、それがすでにコンパイルされている場合、コンパイルする前のコードはどこからも得られません。そのため “同じ関数定義を 2 度コンパイルする” 方法はありません。

```
(defun factorial (integer)
  "INTEGER の階乗を計算する。"
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
⇒ factorial
```

```
(byte-compile 'factorial)
⇒
#[(integer)
  "^H\301U\203^H^@\301\207\302^H\303^HS!\"\207"
  [integer 1 * factorial]
  4 "INTEGERの階乗を計算する。"]
```

この結果はバイトコード関数オブジェクトです。そのオブジェクトに含まれる文字列が実際のバイトコードで、その中の各文字が命令や命令の作用対象 (operand) です。また、ベクタは、その関数で使われているすべての定数、変数名、関数名を含みます。ただし、特別な命令として作られている特定のプリミティブは現れません。

compile-defun Command

このコマンドはポイントを含む defun を読み込んで、コンパイルし、結果を評価します。[1] このコマンドを、実際に関数を定義する defun で実行した場合は、結果としてその関数のコンパイルされたバージョンが使われるようになります。

byte-compile-file *filename* Command

この関数は *filename* という名前の Lisp コード・ファイルをバイトコード・ファイルにコンパイルします。出力ファイル名は *filename* の最後に ‘c’ をつけたものになります。

コンパイル時には、入力ファイルを読み込みながら一度に一つの式が処理されます。その式が関数やマクロの定義であれば、コンパイルした関数やマクロの定義がファイルに書き出されます。それ以外の式はまとめられ、それぞれはコンパイルされ、そのファイルが読まれたとき、コンパイル・コードが実行されるように書き込まれます。入力ファイルが読み込まれるとき、すべてのコメントは無視されます。

このコマンドは t を返します。対話的に呼ばれた場合はファイル名の入力を促します。

```
% ls -l push*
-rw-r--r--  1 lewis     791 Oct  5 20:31 push.el

(byte-compile-file "~/emacs/push.el")
     ⇒ t

% ls -l push*
-rw-r--r--  1 lewis     791 Oct  5 20:31 push.el
-rw-rw-rw-  1 lewis     638 Oct  8 20:25 push.elc
```

byte-recompile-directory *directory flag* Command

この関数は、 *directory* にあって再コンパイルが必要な ‘.el’ ファイルすべてを再コンパイルします。 ‘.elc’ ファイルが存在しても ‘.el’ ファイルより古い場合、そのファイルは再コンパイルが必要とされます。

‘.el’ ファイルが対応する ‘.elc’ ファイルを持たない場合には、 *flag* によりどうするかが決まります。この値が nil の場合はそのファイルは無視されます。もし非 nil の場合、ユーザに対しそれぞれのファイルをコンパイルするかどうかを問い合わせます。

---

[1] 訳注: ここでいう defun はトップレベルにある S 式一般を指します。これに関しては Emacs マニュアルの Defuns を参照してください。

このコマンドの返り値は不定です。

batch-byte-compile Function

この関数はコマンド行に指定されたファイルに対して、 `byte-compile-file` を実行します。この関数は Emacs のバッチ走行でのみ使ってください。関数の終了時に Emacs を終了させせてしまうからです。一つのファイルでエラーが起こっても、それに続くファイルは処理され続けます。 (エラーの起きたファイルからは、もちろんコンパイル・コードは作成されません。)

```
% emacs -batch -f batch-byte-compile *.el
```

byte-code *code-string data-vector max-stack* Function

この関数が実際にバイトコードを解釈 (interpret) します。バイトコンパイルされた関数は実際には `byte-code` を呼び出している関数本体で定義されます。この関数を自分で呼び出すのはやめてください。この関数を正しく呼び出す方法はバイトコンパイラにしかわからないからです。

新しい Emacs (バージョン 19 とそれ以降) では、バイトコードは通常、バイトコード関数オブジェクトの一部として実行されます。 `byte-code` を明示的に呼び出すことで実行されることは本当にまれです。

14.3 説明文字列とコンパイル

バイトコンパイルされたファイルからロードされた関数と変数は、必要になったときに、その説明文字列をファイルから動的に取ってきます。これにより Emacs 内部の領域を節約できます。また、ファイルをロードするときに説明文字列自体を処理する必要がなくなるので、ロードも早くなります。この結果、説明文字列を実際に参照するのが遅くなりますが、多くの場合、ユーザをいらいらさせるほどのものではありません。

説明文字列を動的に参照することには以下のような問題点があります。

- コンパイルしたファイルをロードした後で削除したり名前を変えたりすると、そのファイルの関数や変数はもう Emacs から参照できなくなります。
- コンパイルしたファイルを変更してしまうと (たとえば新しいバージョンのファイルをコンパイルすると)、それ以後、このファイルの説明文字列を参照して得られた結果は意味のないものとなります。

ただし、 Emacs のインストールに通常の手段を用いていればこれらの問題はまず起りません。新しいバージョンをインストールするときには今までと違う名前の新しいディレクトリを使います。古いバージョンがインストールされているかぎり、古いファイルは期待されているところに変更されずに残っているはずです。

しかし、もし Emacs を自分自身で作成し、自分が作成したディレクトリで使用している場合には、 Lisp ファイルを編集し再コンパイルした場合、時々、この問題に出くわすでしょう。この問題が起きたなら再コンパイル後、ファイルを再ロードすれば解決できます。

Emacs 19.29 で作成したバイトコンパイル・ファイルは、それよりも古いバージョンの Emacs ではロードできません。古いバージョンではこの機能をサポートしていないからです。

そのため、 `byte-compile-dynamic-docstrings` を `nil` に設定することでこの機能を使わないようにできます。一度これを行うと、コンパイルしたファイルは、古いバージョンの Emacs でもロードできるようになります。これはグローバルに設定することもできますし、変数をファイルにローカル束縛するように指定することで、一つのソース・ファイルに対してこの設定ができます。その一つのやり方が以下のものです。

```
-*-byte-compile-dynamic-docstrings: nil;-*-
```

byte-compile-dynamic-docstrings Variable

この値が非 `nil` の場合には、バイトコンパイラが生成するコンパイル済みファイルが、説明文字列の動的なロードを行うように設定されます。

動的な説明文字列機能により、コンパイルされたファイルは、特別な Lisp リーダ構文要素 (construct)、 '`#@`*count*' を使用して書かれます。この構文要素は次の *count* 個の文字を読み飛ばします。また、 '`#$`' という構文要素も使われるようになります。これは "文字列にしたこのファイルの名前" を表します。 Lisp ソース・ファイルにはこれらの構文要素を使わないのが一番よいでしょう。

14.4 個々の関数の動的なロード

コンパイルするときに『動的な関数ロード (dynamic function loading)』という機能 (『遅延ロード (lazy loading)』としても知られています) を有効にするように選択できます。動的な関数ロードにより、ファイルをロードしてもそのファイルの関数定義の全部が読み込まれることはなくなります。そのかわりに、それぞれの関数定義にはファイルを参照する「名札」が入ります。それぞれの関数が最初に呼び出されるとき、ファイルから完全な関数定義が読み込まれ、「名札」と置き換わります。

動的な関数ロードの利点はファイルのロードがとても速くなることです。たくさんのコマンドが独立して納められているファイルで、そのうちの一つを使っても、すぐには (もしくは、その後ずっと) それ以外のコマンドを使うかどうかわからないような場合には便利です。たくさんのキーボード・コマンドをもつ専門化したモードの場合、コマンドは特定のパターンで使用されることが多いものです。そのような場合、ユーザはモードを呼び出したとしても、そのモードのもつごく少数のコマンドしか使いません。

動的なローディング機能は以下のような欠点を持っています。

- コンパイルしたファイルをロードした後で削除したり名前を変えたりすると、 Emacs はまだロードされていない残りの関数定義をもうロードできなくなります。
- コンパイルしたファイルを変更してしまうと (たとえば新しいバージョンのファイルをコンパイルすると)、まだロードされていない関数をロードしようとしても得られた結果は意味のないものとなります。

もし、ファイルの新しいバージョンをコンパイルしたのなら、すぐに新しくコンパイルしたファイルをロードするのが最もよいでしょう。これにより将来起りうる問題を防ぐことができるからです。

バイトコンパイラが動的な関数ロード機能を使うのは、コンパイル時に変数 `byte-compile-dynamic` が非 `nil` である場合です。ただし、この変数をグローバルに設定しないでください。特定のファイルにしか動的なロード機能は望ましくありません。そのかわり、ファイルローカルな変数束縛を行うことで特定のソース・ファイルでのみ機能を有効にしてください。

```
-*-byte-compile-dynamic: t;-*-
```

byte-compile-dynamic Variable

この値が非 `nil` の場合には、バイトコンパイラが生成するコンパイル済みファイルが動的な関数ロードを実行するように設定されます。

fetch-bytecode *function* Function

この関数は、 *function* の関数定義がまだ完全にはロードされていない場合、関数を含むバイトコンパイルされたファイルからこの関数を即座にロードします。引数 *function* はバイトコード関数オブジェクトでも関数名でも構いません。

14.5 コンパイル時に行なわれる評価

ここで挙げる機能により、プログラムをコンパイルするときに実行されるコードを書くことができます。

eval-and-compile *body* Special Form

この形式で *body* を囲むことにより、その中のコードがコンパイル時にも実行時にも (コンパイルされている、されていないに関わらず)、評価されるようにします。

同様の結果を得るには *body* を別のファイルに納め、そのファイルを `require` で参照します。このように実行したいコードがかなりな量ある場合は、 `require` を使う方がよいでしょう。

eval-when-compile *body* Special Form

この形式で *body* を囲むことにより、そのコードはコンパイル時に評価され、コンパイルされたプログラムがロードされるときには評価されないようになります。コンパイラによる評価結果は定数になってコンパイルされたプログラムに現れます。プログラムがコンパイルされていなければ、インタプリタで解釈されるとき *body* は通常どおり評価されます。

トップ・レベルでみると、これは Common Lisp での語法である `(eval-when (compile eval) ...)` と似ています。それ以外の場所では、 Common Lisp の ‘`#.`’ リーダ・マクロ (こちらはインタプリタによる解釈時に実行されるわけではありません) が `eval-when-compile` の行うものに近くなります。

14.6 バイトコード関数オブジェクト

バイトコード関数は特別なデータ・タイプになります。『バイトコード関数オブジェクト (byte-code function objects)』というタイプです。

内部的には、バイトコード関数オブジェクトはベクタにとてもよく似ています。しかし、関数として呼ばれるべきところに現れたとき、評価子はこのデータ・タイプを特別に扱います。バイトコード関数オブジェクトの印字表現は、最初の '[' の前に '#' が追加される以外はベクタの表現と似ています。

Emacs バージョン 18 にはバイトコード関数オブジェクトというデータ・タイプはなく、コンパイルされた関数はバイトコードを実行するために `byte-code` という関数を使っていました。

バイトコード関数オブジェクトは少なくとも 4 個の要素を持たなくてはなりません。要素数にかぎりはありませんが最初の 6 個の要素のみが実際には使用されます。その 6 要素は以下のものです。

引数リスト　引数シンボルのリスト。

バイトコード
: バイトコード命令を含む文字列。

定数　バイトコードが参照する Lisp オブジェクトのベクタ。これは関数名や変数名として使われるシンボルを含みます。

スタックの大きさ
: この関数が必要とするスタックの大きさの最大値。

説明文字列　説明文字列。もしなければ `nil` になります。これは数やリストの場合もあります。この場合、説明文字列はファイルにしまわれています。本当の説明文字列を得るには関数 `documentation` を使います (21.2 節「説明文字列へのアクセス」p.340 参照).

対話引数　もしあれば、対話引数の指定。これは文字列か Lisp 式のどちらかです。対話的に使われる関数でなければこれは `nil` になります。

ここで印字表現によるバイトコード関数オブジェクトの例を示します。コマンド `backward-sexp` の定義です。

```
#[(&optional arg)
  "^H\204^F^@\301^P\302^H[!\207"
  [arg 1 forward-sexp]
  2
  254435
  "p"]
```

バイトコード・オブジェクトを生成する基本的な方法は `make-byte-code` を用いることです。

make-byte-code &rest *elements* 　　Function

この関数は *elements* をその要素とするバイトコード関数オブジェクトを構築し、返します。

関数 byte-code に与える要素を自分でなんとかしようなどとしてはいけません。要素に不整合があると、その関数を呼んだとき Emacs がクラッシュしてしまいかねないからです。このオブ

ジェクトを生成するのは常にバイトコンパイラに任せてください。バイトコンパイラは要素に矛盾を生じさせません (というのが我々の希望なのですが)。

バイトコード・オブジェクトの要素は `aref` を使えばアクセスできます。また、 `vconcat` で同様の要素をもつベクタを生成できます。

14.7 逆アセンブルされたバイトコード

人間はバイトコードを書きません。その仕事はバイトコンパイラに任されています。しかし、猫のような好奇心を満足させるために逆アセンブラが提供されています。逆アセンブラはバイトコンパイルされたコードを人間が読める形式に変換します。

バイトコード・インタプリタは単純なスタック・マシンとしてインプリメントされています。これは、インタプリタがそれ自身でもつスタックに値をプッシュしていき、それをポップして計算に用い、その結果自身をスタックにプッシュし戻します。

バイトコード関数は返るときスタックから値をポップし、それを関数の値として返します。

このスタック操作に加え、変数とスタックの間で値を移すことで、通常の Lisp 変数の値を使い、束縛し、設定することが可能になります。

disassemble *object* &optional *stream* Command

この関数は *object* の逆アセンブル・コードを印字します。もし *stream* 引数があれば、そこに出力されます。さもなくば、逆アセンブルされたコードはストリーム `standard-output` に印字されます。引数 *object* は関数名でも lambda 式でも構いません。

特に例外として、この関数は対話的に使われると '`*Disassemble*`' というバッファに出力します。

ここで `disassemble` 関数を用いた例を二つあげます。バイトコードと Lisp ソースを関連づけやすいように説明のためのコメントを付加しました。このコメントは `disassemble` の出力には現れません。この例では最適化されていないバイトコードを示します。最近ではバイトコードはたいてい最適化されていますが、説明のためにはこれで十分ですのでこの例を書き直すつもりはありません。

```
(defun factorial (integer)
  "整数 integer の階乗を計算する。"
  (if (= 1 integer) 1
    (* integer (factorial (1- integer)))))
     ⇒ factorial

(factorial 4)
     ⇒ 24

(disassemble 'factorial)
     ⊣ byte-code for factorial:
 doc: 整数 integer の階乗を計算する。
 args: (integer)
```

```
0   constant 1              ; 1 をスタックにプッシュする。

1   varref   integer        ; 環境から integer の
                            ;    値を取得し、
                            ;    その値をスタックにプッシュする。
2   eqlsign                 ; スタックから一番上の
                            ;    二つの値をポップし、
                            ;    それを比較し、
                            ;    結果をスタックにプッシュする。
3   goto-if-nil 10          ; スタックの一番上からポップし、
                            ;    値を調べる
                            ;    nil ならば 10 へ飛び、
                            ;    それ以外は次に続く。
6   constant 1              ; スタックの一番上に 1 をプッシュする。

7   goto     17             ; 17 へ飛ぶ。(この場合、1 が
                            ;    この関数から返される。)
10  constant *              ; シンボル * をスタックにプッシュする。

11  varref   integer        ; integer の値を
                            ;    スタックにプッシュする。
12  constant factorial      ; factorial をスタックにプッシュする。

13  varref   integer        ; integer の値を
                            ;    スタックにプッシュする。

14  sub1                    ; integer をポップし、値から 1 を引き、
                            ;    新しく得た値をスタックに
                            ;    プッシュする。
                            ; スタックの現在の値は:
                            ;    — integer から 1 引いた値
                            ;    — factorial
                            ;    — integer の値
                            ;    — *
15  call     1              ; 関数 factorial を呼び出す。
                            ;    このとき、スタックの
                            ;    最初 (すなわち一番上) の
                            ;    要素を引数として用いる。
                            ;    返ってきた値をスタックに
                            ;    プッシュする。
```

```
                                      ; スタックの現在の値は:
                                      ;    — factorial の
                                      ;         再帰呼出しの結果
                                      ;    — integer の値
                                      ;    — *
  16   call     2                     ; スタックの最初の二つ
                                      ;    (すなわち、一番上の二つ)
                                      ;    の要素を引数として、
                                      ;    関数 * を呼び出し、
                                      ;    その結果をスタックにプッシュする。
  17   return                         ; スタックの一番上の要素を返す。
      ⇒ nil
```

silly-loop 関数はもう少し複雑です。

```
  (defun silly-loop (n)
    "N 回繰り返して、ループの前後の時間を返す。"
    (let ((t1 (current-time-string)))
      (while (> (setq n (1- n))
                0))
      (list t1 (current-time-string))))
      ⇒ silly-loop

  (disassemble 'silly-loop)
       ⊣ byte-code for silly-loop:
   doc: N 回繰り返して、ループの前後の時間を返す。
   args: (n)

  0    constant current-time-string   ; current-time-string を
                                      ;    スタックの一番上に
                                      ;    プッシュする。
  1    call     0                     ; current-time-string を
                                      ;    引数なしで呼び出し、
                                      ;    結果をスタックにプッシュする。
  2    varbind  t1                    ; スタックからポップし、ポップした
                                      ;    値を t1 に束縛する。
  3    varref   n                     ; 環境から n の値を取得し、
                                      ;    その値をスタックにプッシュする。
  4    sub1                           ; スタックの一番上から 1 を引く。
  5    dup                            ; スタックの一番上を複製する。
                                      ;    すなわち、スタックの一番上を
                                      ;    コピーし、そのコピーを
                                      ;    スタックにプッシュする。
```

```
6   varset   n              ; スタックの一番からポップし、
                            ;   n にその値を束縛する。

                            ; 要するに、 dup varset を
                            ;   続けて行うことで、スタックの
                            ;   一番上の値を、ポップせずに
                            ;    n にコピーする。
7   constant 0              ; スタックに 0 をプッシュする。

8   gtr                     ; スタックから一番上にある
                            ;   二つの値をポップし、
                            ;   n の値が 0 よりも大きいか調べ
                            ;   その結果をスタックにプッシュする。
9   goto-if-nil-else-pop 17 ; n <= 0 ならば、 17 へ飛び
                            ;   (これによりループから抜ける)、
                            ;   それ以外ならば、スタックの
                            ;   一番上をポップして次に続く。
12  constant nil            ; nil をスタックにプッシュする
                            ;   (これがループの本体である)。
13  discard                 ; ループ本体の結果を捨てる
                            ;   (while ループは常に、
                            ;   副作用を使うために評価される)。
14  goto     3              ; while ループの最初に戻る。

17  discard                 ; スタックの一番上をポップすることで、
                            ;   while ループの結果を捨てる。
                            ;   この結果とは、 9 の goto 文では
                            ;   ポップされなかった値 nil である。
18  varref   t1             ; t1 の値をスタックにプッシュする。

19  constant current-time-string  ; current-time-string を
                                  ;   スタックの一番上に
                                  ;   プッシュする。
20  call     0              ; 再度 current-time-string を
                            ;   呼び出す。
21  list2                   ; スタックの一番上の二つの要素を
                            ;   ポップし、そのリストを作成し、
                            ;   リストをスタックにプッシュする。
22  unbind   1              ; 局所環境での t1 の束縛を外す。

23  return                  ; スタックの一番上の値を返す。

    ⇒ nil
```

15 Lisp プログラムのデバッグ

Emacs Lisp のプログラムに問題がおきたとき、プログラムで何をしていたかによって、問題を調べるための方法が三つあります。

- プログラムの実行中に問題がおきたのなら、 Lisp デバッガ (デフォルトのデバッガか Edebug) で実行時に何がおきたのか調べられます。
- 構文にたいする問題点ならば、 Lisp はプログラムを読み込むことすらできないはずなので、 Emacs の Lisp 編集機能を使って局所化できるでしょう。
- バイトコンパイラでプログラムをコンパイルしようとしていたのであれば、コンパイラの入力バッファを調べる方法を知る必要があります。

そのほかに便利なデバッグ用ツールとしてドリブル・ファイルがあります。ドリブル・ファイルを開けると、キーボードからの入力文字はすべてそのファイルにコピーされます。後々そのファイルを調べてどんな入力があったか、確かめることができます。 34.8 節「端末入力」 p.592 を参照してください。

端末記述の問題のデバッグには、 `open-termscript` 関数が便利です。 34.9 節「端末出力」 p.596 を参照してください。

15.1 Lisp のデバッガ

『Lisp デバッガ (Lisp debugger)』は、形式の評価を途中で止めることができるようにします。 (『ブレーク (break)』という) 評価が止められている状況では、実行時スタックを調べたり、ローカルもしくはグローバル変数の値を調べたり、その値を変更したりできます。ブレークは再帰編集であるため、通常の Emacs の編集機能が有効です。再帰的にデバッガにはいるプログラムを実行することさえ可能です。 18.10 節「再帰編集」 p.288 を参照してください。

15.1.1 エラーでデバッガにはいる

デバッガにはいる一番重要な場合は Lisp エラーのおきたときです。これでエラーによる直接的な結果を調べることができます。

普段はエラーの結果としてデバッガにはいることはありません。多くのコマンドは、不適切なところで実行されると (バッファの最後での `C-f`) よく Lisp エラーになるので、通常の編集の最中にデバッガを呼び出すのは非常に困ります。エラーのたびにデバッガにはいらせるには、 `debug-on-error` を非 `nil` にします。

debug-on-error User Option

この変数は、ハンドルされないエラーが通知された場合、デバッガを呼ぶべきか否かを定めます。もし `debug-on-error` が `t` なら、すべてのエラーでデバッガを呼びます。 `nil` の場合、デバッガは呼びません。

値はデバッガを呼ぶエラー・コンディションのリストでもかまいません。たとえば、これをリスト `(void-variable)` に設定すると、値のない変数についてのエラーだけがデバッガを呼びます。

この変数が非 `nil` のとき、プロセス・フィルターと見張りの処理で発生したエラーを Emacs が捕捉することはありません。したがって、このようなエラーでもデバッガを起動します。第 33 章「プロセス」p.559 を参照してください。

'`.emacs`' ファイルのロード中におきるエラーをデバッグするには、オプション '`-debug-init`' を使います。これは '`.emacs`' をロードしている間 `debug-on-error` を `t` に束縛し、`condition-case` で初期化ファイルのエラーをとらまえるのを禁止します。

もし '`.emacs`' ファイルで `debug-on-error` を設定しても、その効果は '`.emacs`' のロードが終わるまでしかありません (これは '`-debug-init`' コマンド行オプションを実装する際に生じた望ましくない副産物です)。'`.emacs`' ファイルに `debug-on-error` を永久的に設定させるには、次のように `after-init-hook` を使います。

```
(add-hook 'after-init-hook
          '(lambda () (setq debug-on-error t)))
```

15.1.2 無限ループのデバッグ

プログラムが無限にループして、戻れなくなっているとき、最初の問題はループを止めることです。ほとんどのオペレーティング・システムでは、とりやめをする `C-g` でこれができます。

とりやめでは通常、なぜプログラムがループしたのかについての情報を教えてはくれません。より詳しい情報を得るには、変数 `debug-on-quit` を非 `nil` に設定します。`C-g` によるとりやめはエラーとはみなしません。`debug-on-error` は `C-g` のハンドリングについての効果がありません。同様に `debug-on-quit` はエラーについての効果がありません。

無限ループの途中でデバッガを動かすと、ステップ・コマンドを使ってデバッガからの続行が可能です。ループ全体をステップすれば、おそらく問題を解くのに十分な情報が得らえるでしょう。

debug-on-quit User Option

この変数は、`quit` が通知されハンドルされなかった場合、デバッガを呼ぶべきか否かを定めます。`debug-on-quit` が非 `nil` の場合、とりやめる (つまり `C-g` を押下する) ごとにデバッガを呼びます。`debug-on-quit` が `nil` の場合、とりやめてもデバッガは呼ばれません。18.8 節「とりやめ」p.284 を参照してください。

15.1.3 関数呼出しでデバッガにはいる

プログラムの途中でおきる問題を調べるときに便利な手法として、特定の関数を呼び出したときには必ずデバッガにはいる、というのがあります。問題のおきる関数や問題のおきるところより前の関数をこう設定しておいて、その関数を呼ぶまでは素速くステップしたり、呼出し元に戻るまでステップしたりできます。

debug-on-entry *function-name* Command

この関数は、*function-name* が呼ばれるたびにデバッガを起動するようにします。これは形式 `(debug 'debug)` を関数定義の最初の形式として挿入することで行ないます。

Lisp コードで定義された関数は、解釈されるコードであれコンパイルされたコードであれ、どれでも (訳注: 関数に) はいるときのブレークを設定できます。関数がコマンドなら、 Lisp から呼ばれた場合も、対話的に呼ばれた場合も (引数を読み込んだ後) デバッガにはいります。プリミティブ関数 (すなわち C で書かれた関数) はこの方法でのデバッグはできません。

`debug-on-entry` が対話的に呼ばれた場合、 *function-name* を求めミニバッファにプロンプトを出します。

すでに関数へはいるときにデバッガを起動するよう設定してある場合、 `debug-on-entry` は何も行ないません。

注意 !: `debug-on-entry` の後で関数を再定義すると、デバッガにはいるコードはなくなります。

`debug-on-entry` は *function-name* を返します。

```
(defun fact (n)
  (if (zerop n) 1
      (* n (fact (1- n)))))
     ⇒ fact
(debug-on-entry 'fact)
     ⇒ fact
(fact 3)

------ Buffer: *Backtrace* ------
Entering:
* fact(3)
  eval-region(4870 4878 t)
  byte-code("...")
  eval-last-sexp(nil)
  (let ...)
  eval-insert-last-sexp(nil)
* call-interactively(eval-insert-last-sexp)
------ Buffer: *Backtrace* ------

(symbol-function 'fact)
     ⇒ (lambda (n)
          (debug (quote debug))
          (if (zerop n) 1 (* n (fact (1- n)))))
```

cancel-debug-on-entry *function-name* Command

この関数は、 *function-name* にたいする `debug-on-entry` を取り消します。対話的に呼ばれた場合、 *function-name* を求めミニバッファにプロンプトを出します。もし *function-name* が `nil` や空文字列の場合は、全関数のデバッグを取り消します。

同じ関数にたいし `debug-on-entry` を複数回呼んだ場合、 2 回目の呼出しは何も行ないません。 `cancel-debug-on-entry` は *function-name* を返します。

15.1.4 陽にデバッガへはいる

プログラムに式 (debug) を書いてデバッガを呼ぶこともできます。そうするにはソース・ファイルを読み込み、テキスト'(debug)' を挿入し C-M-x を押下します。セーブする前に挿入を取り消すのを忘れないでください!

'(debug)' を入れるのは、付加的な形式が評価されかつその値が無視されるような場所でなければなりません。((debug) の値が無視されなければ、プログラムの実行を変えることになります!) 適切で、ありがちな場所は progn や暗黙 progn の内側です (9.1 節「順次実行」p.111 参照)。

15.1.5 デバッガを使う

デバッガにはいると、以前に選択していたバッファを片方のウィンドウに表示し、そして'*Backtrace*' という名前のバッファを別のウィンドウに表示します。バックトレース・バッファには、1 行ごとに現在の Lisp 関数実行の各レベルが示されます。このバッファの最初に、デバッガが呼び出された理由を示したメッセージがあります (たとえば、エラーによって呼び出された場合は、エラー・メッセージとそれに関連するデータです)。

バックトレース・バッファは読出し専用で、Debugger モードという特別の主モードになっています。そこでは、文字がデバッガ・コマンドとして定義されています。普通の Emacs 編集コマンドも使えます。エラーの際に編集していたバッファを調べるためにウィンドウを切替えたり、バッファを切替えたり、ファイルを読み込んだりなどの編集もできます。しかし、デバッガは再帰編集レベルにあります (18.10 節「再帰編集」p.288 参照)。デバッガを使う必要がなくなったときは、バックトレース・バッファに戻って (q コマンドで) 正式にデバッガを抜けなければなりません。デバッガから抜けると、再帰編集を終わらせてバックトレース・バッファを kill します。

バックトレース・バッファには、実行している関数とそれに与えられた引数の値が表示されます。また、ポイントを動かして、その行が示しているスタック・フレームを指定する目的にも使えます (スタック・フレームとは、Lisp インタプリタが関数の実行についての情報を記録しておく場所です)。ポイントがある行のフレームを『現在のフレーム (current frame)』とみなします。デバッガ・コマンドのいくつかは現在のフレームに作用します。

デバッガ自身にスタック・フレームをいくつ作るかという仮定をするため、デバッガ自身はバイトコンパイルした上で動作させる必要があります。解釈されて動作しているときにはこの仮定がくずれてしまいます。

15.1.6 デバッガ・コマンド

デバッガ (Debugger モード) の中では、通常のカーソル移動コマンドのほかに特殊なコマンドが有効です (Emacs の普通の機能、たとえば、ウィンドウやバッファを切替えたりすることは、ずっと有効なことを忘れないでください)。

デバッガ・コマンドはコードにそってステップ実行するのに主に使われます。これでどのように制御が流れるのかを見ることができます。デバッガは解釈された関数の制御構造にそってステップできますが、バイトコンパイルされた関数ではできません。バイトコンパイルされた関数でそうしたいなら、同じ関数を解釈されるような再定義に置き換えてください (それには関数のソース・ファイルを読み込み、その定義のあるところで `C-M-x` を押下します)。

以下は Debugger モードのコマンドの一覧です。

`c`
: デバッガを抜けて実行を再開します。再開が可能ならば、あたかもデバッガにはいらなかったかのように (デバッガ内で変更した変数やデータ構造の作用は別として) プログラムの実行を続けます。

  再開が可能なのは、関数の入口、出口、陽に (訳注: デバッガを) 呼び出したとき、 `C-g` によるとりやめ、などがあります。エラーのせいでデバッガにはいったときは続行ができません。

`d`
: 実行を再開しますが、次に Lisp 関数が呼び出されるとまたデバッガにはいります。これによって、一つの式の中の各式をステップ実行することができ、それらの式が計算する値など実行状況を見ることができます。

  このようにしてデバッガにはいった関数呼出しのスタック・フレームには、そのフレームから抜けるときにもデバッガにはいるように自動的に印がつけられます。この印を消すには `u` コマンドを使います。

`b`
: 現在のフレームから抜けるときにデバッガにはいるようフレームに印をつけます。このように印をつけたフレームには、バックトレース・バッファで星を表示します。

`u`
: 現在のフレームから抜けるときにはデバッガにはいらないようにします。これはフレームでの `b` コマンドの取り消しになります

`e`
: Lisp 式をミニバッファから読み込んで評価し、その値をエコー領域に表示します。デバッガは実行中にある重要な変数群を変更し、操作の一部分としてカレント・バッファも変更します。 `e` は一時的にデバッガの外での値に復帰するので、その変数の値も調べることができます。これでデバッガがかなり透過的になります。一方 `M-:` は、デバッガの中でも特殊なことをしません。デバッガの中での変数の値を表示します。

`q`
: デバッグしているプログラムを終了します。 Emacs のコマンド実行のトップレベルに戻ります。

C-g によってデバッガにはいったが、デバッグではなく本当にとりやめをしたいときに、 q コマンドを使います。

r
: デバッガから値を返します。その値はミニバッファで式を読み込み、評価して求めます。

  r コマンドは (b を指定して) Lisp の呼出しフレームから抜けたために呼び出されたデバッガでは便利です。また、 debug を呼んだときの値を使用するというときにも便利です。それ以外の場合では、 r と c は同じで、指定した値は無視されます。

  エラーのせいでデバッガにはいったとき r は使えません。

15.1.7 デバッガの起動

ここではデバッガを起動するための関数について詳しく述べます。

debug &rest *debugger-args* Function

この関数は、デバッガにはいります。バッファを '*Backtrace*' (2 番目の再帰編集の場合 '*Backtrace*<2>' など) という名前のバッファに切替え、 Lisp 関数呼出しのスタックに関する情報を出力します。この後、バックトレース・バッファを Debugger モードにして、再帰編集にはいります。

Debugger モードの c と r コマンドは再帰編集を抜けます。 debug は前のバッファに戻し、 debug の呼び出したところに返します。これは関数 debug が呼出し元に返る唯一の方法です。

debug に渡した *debugger-args* の先頭が nil の場合 (またはそれが後述する表にある特殊な値でない場合)、残りの引数は '*Backtrace*' バッファの先頭に表示されます。このしくみはユーザへのメッセージを表示するのに使います。

しかし、 debug の最初の引数が以下の特殊な値の一つである場合、特別な意味をもちます。通常、デバッガと Emacs の内部だけがそれらの値を debug に渡し、 debug を呼んでいるプログラマが用いることはありません。

特別な値は、

lambda
: 最初の引数が lambda の場合は、 debug-on-next-call が nil でなかったときに関数を呼んだため、 debug にはいったということを意味します。デバッガはバッファの先頭行に 'Entering:' を表示します。

debug
: 最初の引数が debug のときは、関数にはいったときにデバッグするよう設定した関数に、はいったことを示します。 lambda の場合と同様に、デバッガは 'Entering:' を表示します。この関数のスタック・フレームに印をつけて、この関数から抜けたときにデバッガを起動するようにします。

t
: 最初の引数が t の場合は、 debug-on-next-call が非 nil のときにリスト形式を評価したために、 debug を呼んだことを意味します。デバッガはバッファの先頭行に

```
Beginning evaluation of function call form:
```

を表示します。

`exit` 最初の引数が `exit` の場合は、フレームから抜けるときにデバッガを呼ぶようにあらかじめ印をつけておいたスタック・フレームが、抜けたことを示すのに用います。この場合の `debug` の次の引数は、フレームの返した値です。バッファの先頭行に '`Return value:`' を表示し、返された値をこれに続けて表示します。

`error` 第 1 引数が `error` の場合、エラーか `quit` が通知されハンドルされなかったため、デバッガにはいったことを示すのに用います。'`Signalling:`' を表示し、通知されたエラーと `signal` にたいする引数とを後につけて表示します。たとえば、

```
(let ((debug-on-error t))
  (/ 1 0))

------ Buffer: *Backtrace* ------
Signaling: (arith-error)
  /(1 0)
...
------ Buffer: *Backtrace* ------
```

エラーが通知されたのなら、当然変数 `debug-on-error` は非 `nil` です。`quit` が通知されたのなら、当然変数 `debug-on-quit` は非 `nil` です。

`nil` あなたが陽にデバッガにはいりたいときは、 *debugger-args* の最初に `nil` を使います。バッファの先頭行に *debugger-args* の残りを表示します。たとえば、 `debug` を呼んだ条件を思い出すためメッセージを表示するのに、この機能を使うことができます。

15.1.8 デバッガの内部

この節ではデバッガ内部で使っている関数と変数について述べます。

debugger Variable

この変数の値は、デバッガを起動するために呼ぶ関数です。この値は、任意個の引数をもちうる関数 (もしくは、より一般的には関数の名前) でなくてはいけません。この関数は、何らかのデバッガにはいるものとされます。この変数のデフォルト値は `debug` です。

Lisp がこの関数に渡す最初の引数は、どのように呼ばれたかを示します。引数に関する慣例の詳細は、 `debug` の記述で述べられています。

backtrace Command

この関数は、今現在アクティブになっている Lisp 関数呼出しのトレースを印字します。これは `debug` で '`*Backtrace*`' バッファを作る際、用いる関数です。どの関数呼出しが

アクティブなっているかを調べるのにスタックをアクセスしなくてはいけないため、この関数は、 C で書かれています。常に `nil` を返します。

次は Lisp 式で陽に `backtrace` を呼んだ例です。これはバックトレースをストリーム `standard-output` に印字します。この場合は ‘`backtrace-output`’ です。バックトレースの各行が関数の一つの呼出しを表わします。関数の引数の値がわかっているときは、各行に表示します。計算中の場合はそのように表わします。 (この例では) 特殊形式の引数が省略されています。

```
(with-output-to-temp-buffer "backtrace-output"
  (let ((var 1))
    (save-excursion
      (setq var (eval '(progn
                         (1+ var)
                         (list 'testing (backtrace))))))))

     ⇒ nil
----------- Buffer: backtrace-output ------------
  backtrace()
  (list ...computing arguments...)
  (progn ...)
  eval((progn (1+ var) (list (quote testing) (backtrace))))
  (setq ...)
  (save-excursion ...)
  (let ...)
  (with-output-to-temp-buffer ...)
  eval-region(1973 2142 #<buffer *scratch*>)
  byte-code("...  for eval-print-last-sexp ...")
  eval-print-last-sexp(nil)
* call-interactively(eval-print-last-sexp)
----------- Buffer: backtrace-output -----------
```

文字 ‘`*`’ は、フレームの debug-on-exit フラグが設定されていることを表わします。

debug-on-next-call Variable

この変数が非 `nil` のとき、次の `eval`、 `apply`、 `funcall` の前にデバッガを呼ぶべきであることを示します。デバッガが `nil` にします。

デバッガの `d` コマンドの動きはこの変数の設定によります。

backtrace-debug *level flag* Function

この関数は、 *level* レベル下のスタック・フレームの debug-on-exit フラグを *flag* に設定します。 *flag* が非 `nil` の場合、そのフレームを抜ける際、デバッガにはいるようにします。フレームから局所的でない抜けかたをしてもデバッガにはいります。

この関数は、デバッガからのみ呼ばれます。

command-debug-status Variable

この変数は、現在の対話的なコマンドのデバッグ状態を記録します。コマンドが対話的に呼ばれるたびにこの変数は、 `nil` に束縛されます。デバッガは同じコマンドの実行中にあとで別なデバッガを起動するときのために情報を残しておくのにこの変数を設定しておくことがあります。

別のグローバル変数でなくこの変数を使うことの、デバッガにとっての利点は、ひき続くコマンドの呼出しにデータを渡さないようにすることです。

backtrace-frame *frame-number* Function

関数 `backtrace-frame` は Lisp デバッガが使うことを想定しています。 *frame-number* レベル下のスタック・フレームでおきている計算についての情報を返します。

そのフレームがまだ引数を評価していない (あるいは、特殊形式である) 場合は、値は (`nil` *function arg-forms...*) です。

そのフレームが引数を評価し、すでに関数を呼んでしまっているときは、値は (`t` *function arg-values...*) です。

戻り値における *function* は、計算中のリストの CAR として与えられているもの、あるいはマクロ呼出しの場合は `lambda` 式となります。関数に `&rest` 引数があるときは、リスト *arg-values* の最後になります。

frame-number が範囲外のとき、 `backtrace-frame` は `nil` を返します。

15.2 誤った Lisp 構文のデバッグ

Lisp リーダは不正な構文について知らせてくれますが、どこに真の問題があるのかについては、何も教えてはくれません。たとえば、エラー “End of file during parsing” (訳注: 構文解析中にファイル終了) が式の評価中におこったら、開き括弧 (あるいは角括弧) が多すぎることを示します。リーダはファイルの最後で不釣合いを検出しますが、閉じ括弧をどこに置くべきかはわかりません。同様に “Invalid read syntax: `")"`” は、閉じ括弧が多すぎるのか開き括弧が少なすぎることを示しますが、対応する括弧がどこなのかはわかりません。このときにどうやって変えるところを見つけるのでしょうか?

問題が単に括弧の不釣合いでないとしても、各 defun の先頭で `C-M-e` をやってみて、 defun の最後にゆくかどうかを見るのは、便利な手法です。もし最後に行かないのなら、問題はその defun にあります。

しかしながら、括弧の不対応は Lisp では一番ありがちな構文エラーなので、これに対処するための助言はさらに続きます。

15.2.1 余計な開き括弧

最初のステップは不釣合いな defun を見つけることです。開き括弧が多いときは、ファイルの最後に閉じ括弧を置いて `C-M-b` (`backward-sexp`) を実行します。こうすると不釣合いな defun の先頭に来ることになります (そして `C-SPC` `C-_` `C-u` `C-SPC` で、マークを設定し閉じ括弧の挿入を取り消し、マークの場所に戻ることができます)。

次のステップはどこが誤っているのか決定することです。プログラムから学ぶこと以外には、これについて確かなことをいうことはできませんが、往々にして、既存の字下げが、括弧がどこにあるべきかについての手がかりになることがあります。この手がかりを使う一番簡単な方法は、 `C-M-q` で字下げしてどこが動くかを見ることです。

これをやる前には、 defun に十分な数の閉じ括弧があることを確認しておいてください。さもないと、 C-M-q はエラーになるか、ファイルの最後までを全部字下げしてしまうことになります。ですから、 defun の最後に行って閉じ括弧をいれます。 defun が釣合っていなければだめなので、ここで C-M-e を使ってはいけません。

さてここで defun の先頭に行き C-M-q を押下します。普通はある点以下のすべての行が右にずれるはずです。多分その点の近くになくなった閉じ括弧か余計な開き括弧があるはずです (しかしながら常にそうとはかぎりません。コードから学んで確かめてください)。くいちがいがみつかったら C-M-q を C-_ で取り消します。なぜなら前の字下げの方が、意図していた括弧にたいしておそらく適切なはずだからです。

問題が直ったと考えたら、ふたたび C-M-q を使います。前の字下げが意図した括弧のネストと実際に合っていて、その括弧を戻したのであれば、 C-M-q では何も変わらないはずです。

15.2.2 余計な閉じ括弧

余計な閉じ括弧を扱うには、まずファイルの先頭に開き括弧を置いて 1 文字戻り、 C-M-f を押下し、不釣合いな defun の最後に行きます (そして C-SPC C-_ C-u C-SPC で、マークを設定し開き括弧の挿入を取り消し、マークの場所に戻ることができます)。

そして実際に対応する閉じ括弧を見つけるには、 defun の先頭で C-M-f を押下します。 defun が終るべき場所より手前に来ることがあるかもしれませんが、おそらくそのへんで余計な閉じ括弧を見つけることができるでしょう。

そこで問題がみえてこなければ、次にすべきことは defun の先頭で C-M-q を押下することです。何行かは左にずれるはずです。そのときに、なくなった開き括弧や余計な閉じ括弧が、ずれた行の前の方にあるはずです (しかしながら常にそうとはかぎりません。コードから学んで確かめてください)。くいちがいがみつかったら C-M-q を C-_ で取り消します。なぜなら前の字下げの方が、意図していた括弧にたいしておそらく適切なはずだからです。

問題が直ったと考えたら、ふたたび C-M-q を使います。前の字下げが意図した括弧のネストと実際に合っていて、その括弧を戻したのであれば、 C-M-q では何も変わらないはずです。

15.3 コンパイル時の問題のデバッグ

バイトコンパイル中にエラーがおきたときは、通常はコンパイルしているプログラム中の構文エラーが原因です。コンパイラは適切なエラー・メッセージを '*Compile-Log*' バッファに印字して、止まります。メッセージの中にはエラーがみつかった関数名があることもありますが、ないこともあります。いずれにせよ、以下ではファイルのどこでエラーがおきたのかを見つける方法について述べます。

まずすべきことはバッファ' *Compiler Input*'に切替えることです。 (バッファ名がスペースで始まっているため、 M-x list-buffers では表示されません。) このバッファにはコンパイル中のプログラムがあり、ポイントはバイトコンパイラがどこまで読み込んだかを示します。

エラーが不正な Lisp 構文によるものならば、ポイントは不正な構文を検出したまさにその場所を示します。エラーの原因はその近くとはかぎりません。前節の手法を使ってエラーを見つけてください。

もしエラーが、首尾よく読み込めた形式のコンパイル中に検出されたのなら、ポイントはその形式の後にあるはずです。この場合、今の手法ではエラーを正確に局所化することはできませんが、どの関数を調べればいいのかはわかります。

15.4 Edebug

Edebug は Emacs Lisp プログラムのソース・レベル・デバッガで、以下のようなことができます。

- 評価をステップ実行し、各式の前後で止まること。
- 条件つきまたは無条件のブレークポイントを設定すること。
- 指定した条件が成立したときに停止すること (グローバル・ブレーク条件)。[1]
- 遅くあるいは速くトレースすること。各ストップ・ポイントや各ブレークポイントでしばらく停止すること。
- 式の結果を表示し、あたかも Edebug の外にいるかのように式を評価すること。
- 式のリストを自動的に再評価し、 Edebug が表示を更新するごとにその結果を表示すること。
- 関数の入口と出口でトレース情報を出力すること。
- エラーがおきたときに停止すること。
- Edebug 自身のフレームを除いたバックトレースを表示すること。
- 定義形式とマクロの引数評価を指定すること。
- おおまかな網羅率検査と頻度計数を得ること。

最初の三つの節を読めば十分に Edebug を使うことができるようになるでしょう。

15.4.1 Edebug 利用入門

Edebug で Lisp プログラムをデバッグするには、まずあなたがデバッグしたい Lisp コードの『膳立て (instrument)』をしなければなりません。これを簡単に行なう方法は、まず、ポイントをマクロや関数定義の中に移動して、 `C-u C-M-x` (前置引数をつけた `eval-defun`) をすることです。コードの膳立てをするほかの方法については 15.4.2 節「Edebug のために膳立てをする」p.204。

いったん関数が膳立てをされると、その関数のどのような呼出しも Edebug をアクティブにします。 Edebug をアクティブにすると、あなたの選択した Edebug 実行モードによって、実行を停止して関数をずっとステップさせたり、表示を更新しデバッグ用コマンドを調べながら実行を

[1] 原著ではここだけグローバル・ブレーク・イベント (the global break event) としています。

継続させたりすることができます。デフォルトの実行モードは step で、これは実行を停止させます。 15.4.3 節「Edebug 実行モード」p.205 を参照してください。

Edebug の中にいるときは、デバッグ中の Lisp コードのソースを示している Emacs のバッファを、通常ながめることになります。これを『ソース・コード・バッファ (source code buffer)』といいます。このバッファは一時的に読出し専用です。

左はしの矢印は関数の実行している箇所を示します。ポイントははじめ関数が行のどこを実行しているのかを示していますが、あなたが自分でポイントを動かしてしまうとそうでなくなります。

もし (以下に示す)`fac` の定義の膳立てをして `(fac 3)` を実行すると、以下のようなものが見られるはずです。ポイントは `if` の前の開き括弧にあります。

```
(defun fac (n)
=>⋆(if (< 0 n)
      (* n (fac (1- n)))
    1))
```

Edebug が関数中で実行を止められる場所を『ストップ・ポイント (stop point)』といいます。これはリストである部分式の前後と、変数参照の後に起きます。以下に関数 `fac` のストップ・ポイントを終止符で示します。

```
(defun fac (n)
  .(if .(< 0 n.).
      .(* n. .(fac .(1- n.).).).
    1).)
; (訳注: 原文には 1-の前にストップ・ポイントがなかった。)
```

ソース・コード・バッファでは、 Emacs Lisp モードのコマンドに加えて、 Edebug の特殊なコマンドが有効です。たとえば、 Edebug コマンドの `SPC` を押下して、次のストップ・ポイントまで実行することができます。 `fac` にはいった後、 `SPC` を 1 回押下した直後の様子を以下に示します。

```
(defun fac (n)
=>(if ⋆(< 0 n)
      (* n (fac (1- n)))
    1))
```

Edebug が式の後で実行を止めたときは、その式の値をエコー領域に表示します。

ほかによく使うコマンドとして、 `b` でストップ・ポイントにブレークポイントを設定したり、 `g` でブレークポイントまで実行したり、 `q` で Edebug を抜けてトップレベルのコマンド・ループに戻ったりすることができます。`?` で全 Edebug のコマンドの一覧を見ることができます。

15.4.2 Edebug のために膳立てをする

Lisp コードをデバッグするのに Edebug を使うためには、まずはじめにコードの『膳立て (instrument)』をすることが必要です。コードの膳立てをすると適切な場所で Edebug を起動するコードを追加します。

いったん Edebug をロードすると、コマンド C-M-x (eval-defun) は再定義されるので、前置引数をつけて定義上で起動すると、その定義を評価する前に膳立てをするようになります (ソース・コード自身は修正しません)。もし変数 edebug-all-defs が非 nil のとき、前置引数の意味を反転させます。このとき C-M-x は前置引数がないとき、定義の膳立てをします。 edebug-all-defs のデフォルト値は nil です。コマンド M-x edebug-all-defs は、変数 edebug-all-defs の値をトグルします。

edebug-all-defs が非 nil のとき、コマンド eval-region、 eval-current-buffer、eval-buffer も、評価する定義の膳立てをします。同様に、 edebug-all-forms は、 eval-region が定義をしない形式も含んだ全形式の膳立てをすべきかどうかを制御します。これはロードや、ミニバッファでの評価には関係ありません。コマンド M-x edebug-all-forms は、このオプションをトグルします。

別なコマンド M-x edebug-top-level-form は、 edebug-all-defs や edebug-all-forms の値にかかわらず全トップレベルの形式の膳立てをします。

Edebug がセッションではじめてのコードの膳立てをしようとするとき、フック edebug-setup-hook を作動させてからこれを nil にします。使用中のパッケージにまつわる Edebug 仕様指定を、 Edebug を使っているときだけロードするのに、これを使えます。

Edebug がアクティブな間、コマンド I (edebug-instrument-callee) は、もしされていなければポイントの後のリスト形式に呼ばれているマクロや関数定義の膳立てをします。これは Edebug がその関数のソースの場所がどこかわかっているときだけ可能です。 Edebug をロードすると、 eval-region は評価する全定義の位置を記録するようになります (膳立てをしないときでも)。関数の膳立てをしてからステップする i コマンド (15.4.4 節「ジャンプ」p.207 参照) を参照してください。

Edebug は、標準の特殊形式、式を引数にもつ対話的形式、匿名ラムダ式、ほかの定義形式について知っています。ユーザが定義したマクロが、マクロ呼出しの引数をどう処理するかは Edebug にはわからないので、あなたが Edebug にそれを教えなければなりません。詳細は 15.4.16 節「マクロ呼出しの膳立て」p.217 を参照してください。

膳立てを定義から除くには、単に膳立てをしないよう定義を再評価します。膳立てをしないで形式を評価する二つの方法があります。 load でファイルから評価する方法とミニバッファから eval-expression (M-:) で評価する方法です。

Edebug が膳立てをしている間に構文エラーを検出したときは、誤ったコードにポイントを置き、 invalid-read-syntax エラーを通知します。

Edebug 内で有効なほかの評価関数については、 15.4.9 節「評価」p.211 を参照してください。

15.4.3 Edebug 実行モード

Edebug にはあなたがデバッグしているプログラムを作動させているときの実行モードがいくつかあります。これらの選択肢を『Edebug 実行モード (Edebug execution mode)』といいます。主モードや副モードと勘違いしないでください。現在の Edebug 実行モードは、 Edebug が

(訳注: 実行を) 停止させるまでにどれだけ実行を継続させるか、そして、停止させる前に評価の進行状況をどの程度 Edebug に表示させるかを、決定します。

通常は、どのモードでプログラムを継続させるかというコマンドを押下することで、 Edebug 実行モードを指定します。これらのコマンドの一覧を示します。 S 以外は少なくとも少しの間プログラムの実行を再開します。

`S` 停止 (stop): これ以上プログラムを実行しない。 Edebug のコマンドを待つ (`edebug-stop`)。

`SPC` ステップ (step): 次のストップ・ポイントにきたら止める (`edebug-step-mode`)。

`n` 次 (next): 式の後にある次のストップ・ポイントで止める (`edebug-next-mode`)。 15.4.5 節「いろいろな Edebug のコマンド」 p.208 の `edebug-forward-sexp` を参照。

`t` トレース (trace): 各 Edebug ストップ・ポイントで 1 秒休む (`edebug-trace-mode`)。

`T` 高速トレース (rapid trace): 各ストップ・ポイントで表示を更新するが、休みはしない (`edebug-Trace-fast-mode`)。

`g` 進める (go): 次のブレークポイントまで作動させる (`edebug-go-mode`)。 15.4.6 節「ブレークポイント」 p.208 を参照してください。

`c` 続行 (continue): 各ブレークポイントで 1 秒休み続行する (`edebug-continue-mode`)。

`C` 高速続行 (rapid continue): 各ブレークポイントへポイントを移動するが、休まない (`edebug-Continue-fast-mode`)。

`G` ノンストップで進める (go non-stop): ブレークポイントを無視する (`edebug-Go-nonstop-mode`)。引き続き S や任意の編集コマンドを押下すると、プログラムを止めることができます。

一般に上の表の上の方にある実行モードは、表の下にあるモードよりプログラムをゆっくり動かしたり、すぐに止めたりします。

実行中やトレース中に Edebug コマンドを押下して実行に割り込むことができます。次のストップ・ポイントで Edebug はプログラムを止め、あなたの押下したコマンドを実行します。たとえば、実行中の `t` は、次のストップ・ポイントから trace モードに切り替えます。実行を止め何もさせないようにするには S を使います。

関数がたまたま入力を読み込むものであったとき、実行を割り込むつもりで押下した文字は、関数によって代わりに読み込まれます。プログラムが入力を要求するときについて注意を払っていれば、そのような予期せぬ結果を避けることができます。

この節のコマンドを含むキーボード・マクロは、うまく働きません。プログラムに戻ろうとして Edebug を抜けてもキーボード・マクロの追跡に失敗します。これを直すのは大変です。また、 Edebug の外で定義したり実行したりするキーボード・マクロは、 Edebug のなかのコマン

ドには関係ありません。通常これは長所です。ですが `edebug-continue-kbd-macro` オプションを参照してください (15.4.17 節「Edebug のオプション」 p.222 参照)。

新しい Edebug レベルにはいったときは最初の実行は、変数 `edebug-initial-mode` の値となります。デフォルトでは step モードを指定します。たとえば、膳立てをされた関数が一つのコマンドから複数回呼ばれた場合、同じ Edebug レベルに複数回はいることもあることに注意してください。

15.4.4 ジャンプ

この節で述べられているコマンドは、指定した場所に到達するまで実行します。 `i` 以外のすべては、停止する場所を確立するために一時的なブレークポイントを作り、そして go モードにはいります。想定したストップ・ポイントの前のブレークポイントも実行を止めます。ブレークポイントの詳細については 15.4.6 節「ブレークポイント」 p.208 を参照してください。

これらのコマンドは非局所脱出のときは期待どおりには働きません。なぜなら、非局所脱出はプログラムを止めようと期待した一時的なブレークポイントを、迂回するからです。

`h` ポイントの近くにあるストップ・ポイントまで進める (`edebug-goto-here`)。

`f` 前方に式一つ分、すすめる (`edebug-forward-sexp`)。

`o` 含んでいる式が終わるまでプログラムを作動させる。

`i` ポイントの後の形式で呼ばれる関数やマクロをステップする。

`h` コマンドは一時的なブレークポイントを使って、ポイントの現在の位置の近くにあるストップ・ポイントまで続行します。ブレークポイントの詳細については 15.4.6 節「ブレークポイント」 p.208 を見てください。

`f` コマンドは、プログラムの順方向へ式一つ分だけ実行します。より正確にいえば、 `C-M-f` で到達する場所に一時的なブレークポイントを置き、ブレークポイントで停止するように go モードで実行します。

前置引数 n をつけると一時的なブレークポイントは S 式 n 個分向こうに置かれます。もし含んでいるリストが、 n 個の要素の前で終わっている場合、止まる場所は含んでいる式の後になります。

`C-M-f` で到達する位置がプログラムが本当に実行をする場所であることを注意してください。たとえば、 `cond` の中では、これが正しくないこともあります。

`f` コマンドでは柔軟性のため、ストップ・ポイントからではなくポイントから始めた `forward-sexp` を実行します。現在のストップ・ポイントから、式一つ分を実行したいときは、まず `w` を押下しポイントを動かしてから、 `f` を押下します。

`o` コマンドは、式から「抜け出るまで」続行します。これはポイントを含む S 式の終わりに一時的なブレークポイントを置きます。含んでいる S 式が関数定義自身であるときは、 `o` は定義中の最後の S 式の前まで続行します。今もしそこにいるなら関数から抜けて止まります。いいかえると、最後の S 式の後にいなければ、このコマンドは現在実行中の関数を抜けません。

`i` コマンドはポイントの後のリスト形式で呼ばれる関数やマクロをステップし、最初のストップ・ポイントで停止させます。形式は評価しようとしているものでなくともかまわないことに注意してください。しかし、形式が評価しようとしている関数呼出しであるなら、まだ引数の評価が始まらないうちにこのコマンドを使うようにしてください。評価の始まった後では遅すぎます。

`i` コマンドはステップしようとしている関数あるいはマクロの膳立てを (してなければ) します。これは便利ですが、陽に膳立てをやめるまではずっと膳立てをされたままになっている、ということに、気をつけてください。

15.4.5 いろいろな Edebug のコマンド

Edebug のそのほかのいろいろなコマンドについて解説します。

`?` Edebug のヘルプ・メッセージを表示します (`edebug-help`)。

`C-]` 一つ前のレベルのコマンド・レベルに戻します (`abort-recursive-edit`)。

`q` トップレベルのエディタ・コマンド・ループに戻ります (`top-level`)。これは Edebug の全レベルも含めすべての再帰編集レベルを抜けます。しかしながら、膳立てをしたコードで `unwind-protect` や `condition-case` 形式で保護されたもののせいで、デバッグに戻されることがあります。

`Q` `q` に似ていますが、保護されたコードがあっても止めません (`top-level-nonstop`)。

`r` わかっている一番最後の式の結果をエコー領域に再表示します (`edebug-previous-result`)。

`d` Edebug 自身の関数のそれを明確さのためにのぞいて、バックトレースを表示します (`edebug-backtrace`)。

標準のデバッガのときとは違って、 Edebug のバックトレース・バッファの中では、デバッガのコマンドは使えません。

実行を継続すると、バックトレース・バッファは自動的に kill されます。

Edebug の再帰編集から Edebug を再帰的にアクティブにするコマンドを起動してもかまいません。 Edebug がアクティブなとき、 `q` でトップレベルまでとりやめたり、 `C-]` で再帰編集レベルを一つ中断したりできます。 `d` で評価を保留しているすべてのバックトレースを表示できます。

15.4.6 ブレークポイント

Edebug の step モードでは次のストップポイントに到達すると実行を止めます。いったん動き始めたら Edebug が実行を止める方法は三つあります。それはブレークポイント、グローバル・ブレーク条件、ソースのブレークポイントです。

Edebug を使っている間は、テスト中のプログラムに『ブレークポイント (breakpoint)』、実行を止めるべき点を、指定できます。 15.4.1 節「Edebug 利用入門」 p.203 で定義された任意のストップ・ポイントでブレークポイントを設定できます。ブレークポイントを設定したり解除したりする際、影響をうけるストップ・ポイントは最初のものかあるいはソース・コード・バッファのポイントの後にあるものです。ブレークポイントにまつわる Edebug コマンドは以下のものがあります。

`b` ストップ・ポイント、あるいはポイントの後に、ブレークポイントを置く (`edebug-set-breakpoint`)。前置引数を与えるとブレークポイントは一時的になります (最初にそこでプログラムを止めたときに消えます)。

`u` ストップ・ポイントあるいはポイントの後にあるブレークポイント (もしあれば) を解除します (`edebug-unset-breakpoint`)。

`x` *condition* `RET`
condition が非 `nil` の値に評価したときにかぎって止める、条件つきのブレークポイントを設定します (`edebug-set-condition-breakpoint`)。前置引数を与えると、ブレークポイントは一時的になります。

`B` 定義中の次のブレークポイントにポイントを動かします (`edebug-next-breakpoint`)。

Edebug にいる間ブレークポイントを `b` で設定したり、 `u` で解除したりすることができます。まずポイントをあなたの好きな Edebug のストップ・ポイントに動かし、 `b` や `u` を押下して設定あるいは解除します。ブレークポイントのないところで解除しても何の効果もありません。

定義の膳立てをしたり再評価をしたりすると、その定義内のすべてのブレークポイントはなくなります。

『条件つきブレークポイント (conditional breakpoint)』は、プログラムがそこにくるたびに条件を検査します。条件の評価の結果起きたエラーはすべて無視し、結果は `nil` とみなします。条件つきブレークポイントを設定するには、 `x` を使ってミニバッファから条件式を指定します。以前に確立した条件つきブレークポイントのあるストップ・ポイントに条件つきブレークポイントを確立すると、編集ができるようにミニバッファに前の条件式を置きます。

条件つきや無条件のブレークポイントを『一時的 (temporary)』にするには、設定するコマンドに前置引数を与えます。一時的なブレークポイントがプログラムを止めたとき、自動的に解除されます。

Edebug モードが Go-nonstop のとき以外は、 Edebug はブレークポイントで休んだり止まったりします。 Go-nonstop モードのときはすべて無視します。

ブレークポイントの場所を見つけるには、 `B` コマンドを使います。これはポイントを、同じ関数でポイントの後ろにある次のブレークポイントか、もしそれがなければ最初のブレークポイントに動かします。このコマンドは実行の継続はしません。バッファのポイントを動かすだけです。

15.4.6.1 グローバル・ブレーク条件

『グローバル・ブレーク条件 (global break condition)』は、どこを実行していても、指定した条件が成立したときに実行を止めます。 Edebug は各ストップ・ポイントでグローバル・ブレーク条件を評価します。もしこれが非 `nil` の値に評価したら、ブレークポイントがあったかのように実行モードにしたがって止まったり休んだりします。評価でエラーが起きても実行を止めません。

`X` コマンド (`edebug-set-global-break-condition`) を使って、 `edebug-global-break-condition` に格納した条件式を、設定したり編集したりできます。

グローバル・ブレーク条件は、あるイベントがコードのどこで起こったかをさがすための最も単純な方法ですが、これによりコードの実行は非常に遅くなります。ですから使わないときは条件を `nil` にするべきです。

15.4.6.2 ソースのブレークポイント

定義中の全ブレークポイントは、あなたが再び膳立てをするたびに、なくなります。ブレークポイントがなくならないようにするには、単にソース・コードから関数 `edebug` を呼ぶだけの『ソースのブレークポイント (source breakpoint)』を書くことができます。もちろん条件をつけることもできます。たとえば、 `fac` 関数では、以下に示すように 1 行目に引数が 0 になったときに止まるようにしています。

```
(defun fac (n)
  (if (= n 0) (edebug))
  (if (< 0 n)
      (* n (fac (1- n)))
    1))
```

`fac` の定義が膳立てをされ関数が呼び出されると、 `edebug` の呼出しはブレークポイントのように作用します。実行モードによって Edebug はそこで止まったり休んだりします。

膳立てをされた関数を実行していないときに `edebug` を呼び出すと、その関数は `debug` を呼び出します。

15.4.7 エラーのトラップ

通常 Emacs は `condition-case` でハンドルされないエラーが通知されると、エラー・メッセージを表示します。通常 Edebug がアクティブな間はハンドルされていないすべてのエラーに反応します。あなたはオプション `edebug-on-error` や `edebug-on-quit` で、これをカスタマイズすることができます。 15.4.17 節「Edebug のオプション」 p.222 を参照してください。

Edebug がエラーに反応するとき、エラーの前にあった最後のストップ・ポイントを表示します。この場所は実際にエラーが起きたのとは別の、膳立てをされていない関数を呼び出している箇所であることがあります。束縛されていない変数のエラーの場合、前のブレークポイントは変数を参照しているところからは、とても離れているかもしれません。この場合は全バックトレース (15.4.5 節「いろいろな Edebug のコマンド」 p.208 参照) を表示したくなるでしょう。

Edebug がアクティブなとき debug-on-error や debug-on-quit を変更しても、 Edebug がアクティブでなくなると変更は失われます。さらに Edebug の再帰編集の間、これらの変数は、 Edebug の外の値に束縛します。

15.4.8 Edebug でのながめ

以下の Edebug のコマンドは、 Edebug にはいる前に取得したバッファとウィンドウの状態の外観を見せてくれます。外側のウィンドウ構成は、 Edebug の外側で有効になるウィンドウと (訳注: その) 内容の集まりです。

v
: 一時的に外側のウィンドウ構成をながめる (edebug-view-outside)。

p
: 一時的にポイントを外側の位置に移し、カレント・バッファの外側を表示します (edebug-bounce-point)。前置引数 *n* をつけると *n* 秒間休止します。

w
: ソース・コード・バッファのポイントを、現在のストップ・ポイントに動かします (edebug-where)。さらに、同じバッファを表示している別なウィンドウからこのコマンドを使った場合、それ以降では、現在の定義の表示にそのウィンドウを使います。

W
: Edebug が外側のウィンドウ構成を保存したり復帰したりするかどうかをトグルします (edebug-toggle-save-windows)。

  前置引数を与えると W は選択されたウィンドウの保存復帰についてだけトグルします。ソース・コード・バッファを表示していないウィンドウを指定するには、 C-x X W をグローバル・キーマップから使わなければなりません。

v で外側のウィンドウ構成をながめたり、また、通常には表示されていない場合にも p でただカレント・バッファのポイントへ跳ね返したりすることができます。ポイントを動かした後は、ソース・コード・バッファから w でストップ・ポイントへ戻りたくなるでしょう。

W で保存をやめるときはいつでも、 Edebug は保存した外側のウィンドウ構成を忘れます。したがって、保存を再開した場合でも、次に (プログラムを継続して)Edebug を抜けたときに、現在のウィンドウ構成は変わりません。しかしながら、ウィンドウを十分オープンしていないときは、 ‘*edebug*’ と ‘*edebug-trace*’ の自動再表示が、あなたの見たいバッファとかちあうことがあります。

15.4.9 評価

Edebug の中で、「あたかも」 Edebug が走っていないかのように式を評価できます。 Edebug は式の評価と印字にたいして透過的になるようにしています。副作用を起こす式の評価は Edebug が陽に保存復帰したものをのぞいて期待どおりに働きます。この過程についての詳細は、 15.4.14 節「外側の文脈」 p.215 を参照してください。

e *exp* RET
: 式 *exp* を Edebug の外側の文脈で評価します (edebug-eval-expression)。つまり Edebug は評価への影響を最少化しようとします。

`M-: exp RET`
: 式 *exp* を Edebug 自身の文脈で評価します。

`C-x C-e`
: ポイントの前の式を Edebug の外側の文脈で評価します (`edebug-eval-last-sexp`)。

Edebug は、'`cl.el`' (バージョン 2.03 かそれ以降) の `lexical-let` `macrolet` `symbol-macrolet` 構文要素で生成された、辞書的に束縛されたシンボルへの参照を含む、式の評価をサポートします。

15.4.10 評価リスト・バッファ

'`*edebug*`' という『評価リスト・バッファ (evaluation list buffer)』を使って対話的に式を評価することができます。また、 Edebug が表示を更新するたびに自動的に評価される式の『評価リスト (evaluation list)』も使えます。

`E`
: 評価リスト・バッファ '`*edebug*`' に切り替えます (`edebug-visit-eval-list`)。

'`*edebug*`' バッファでは Lisp Interaction モード (*The GNU Emacs Manual* の "Lisp Interaction" 節参照) のコマンドのほかに以下の特殊なコマンドが使えます。

`LFD`
: 外側の文脈でポイントの前の式を評価し、バッファに値を挿入します (`edebug-eval-print-last-sexp`)。

`C-x C-e`
: ポイントの前の式を Edebug の外側の文脈で評価します (`edebug-eval-last-sexp`)。

`C-c C-u`
: バッファの内容から新しい評価リストを作ります (`edebug-update-eval-list`)。

`C-c C-d`
: ポイントがある評価リスト群を消去します (`edebug-delete-eval-item`)。

`C-c C-w`
: 現在のストップ・ポイントのソース・コード・バッファに戻ります (`edebug-where`)。

`LFD` や `C-x C-e` で、 '`*scratch*`' でのように、評価リスト・ウィンドウの式を評価することができますが、これらは Edebug の外側の文脈で評価されます。

あなたが対話的に入力した式 (とその結果) は実行を継続したときに失われます。しかし、実行が止まるたびに評価をする式を含む『評価リスト (evaluation list)』を設定することができます。

こうするには評価リスト・バッファの中に単数または複数の『評価リスト群 (evaluation list group)』を書きます。評価リスト群には単数または複数の Lisp 式があります。各群はコメント行で分割されます。

コマンド `C-c C-u` (`edebug-update-eval-list`) は、バッファをスキャンして各群の 1 番目の式を用い評価リストを再構築します (なぜこうするかというと群の 2 番目の式は前に計算され表示された値だからです)。

無限ループになるので、膳立てをしたコードを実行する式をつけ加えないように注意してください。

Edebug への各呼出しは、後ろに現在の値をつけた各式をバッファに挿入して、評価リストの再表示を行ないます。コメント行も挿入するので各式はそれぞれに群をなします。したがってバッファ・テキストを変えずに C-c C-u をまた押下した場合、評価リストは実質的には変化しません。

評価リストの評価でエラーが起きたとき、それが結果であるかのように、エラー・メッセージが文字列として表示されます。したがって現在無効な変数を使用する式も、デバッグに割り込むことはありません。

以下は、式をいくつか追加した後の評価リスト・ウィンドウの例です。

```
(current-buffer)
#<buffer *scratch*>
;---------------------------------------------------------------
(selected-window)
#<window 16 on *scratch*>
;---------------------------------------------------------------
(point)
196
;---------------------------------------------------------------
bad-var
"Symbol's value as variable is void: bad-var"
;---------------------------------------------------------------
(recursion-depth)
0
;---------------------------------------------------------------
this-command
eval-last-sexp
;---------------------------------------------------------------
```

群を消去するにはポイントを中にもってきて C-c C-d を押下するか、単に群のテキストを消去してから C-c C-u で評価リストを更新するかします。新しい式を評価リストに追加するには、適切な場所に式と新しいコメント行を挿入します (コメント行にダッシュを挿入する必要はありません — 内容は関係ありません)。そして C-c C-u を押下します。

'*edebug*' を選択した後、 C-c C-w でソース・コード・バッファに戻ることができます。'*edebug*' バッファは実行を継続したときに抹消され、次に必要になったとき再生成されます。

15.4.11 Edebug での印字

もしプログラムの中の式が環状リスト構造を含む値を生成したら、 Edebug がそれを印字しようとしたときにエラーになることがありえます。

環状構造を処理する一つの方法は、印字の切り捨てをする print-length や print-level を設定することです。これは Edebug の方でやってくれます。もし nil なら print-length と print-level を 50 に束縛します (実際は edebug-print-length と edebug-print-level の値にします)。 16.6 節「出力に影響を与える変数」 p.235 を参照してください。

環状構造や要素を共有している構造をよりはっきりと印字するのに、'cust-print' パッケージを利用してもかまいません。

'cust-print' をロードして Edebug 専用にカスタム印字をアクティブにするには、単にコマンド M-x edebug-install-custom-print を使います。標準の印字関数を復帰するには、 M-x edebug-uninstall-custom-print を使います。

以下に環状構造を生成するコードの例をあげます。

```
(setq a '(x y))
(setcar a a))
```

カスタム印字はこれを 'Result: #1=(#1# y)' のように印字します。'#1=' という表記は、それ以降に続く構造に '1' というラベルをつけて、'#1#' という表記は以前につけたラベルの構造を参照します。この表記はリストやベクタの共有要素を表わすのに使います。

ほかのプログラムもカスタム印字を利用可能です。詳しくは 'cust-print.el' を見てください。

15.4.12 トレース・バッファ

Edebug は、実行トレースを '*edebug-trace*' という名前のバッファに格納して、記録しておくことができます。これは関数の呼出しと戻りのログで、関数名、引数、値を示すものです。トレース記録を有効にするには、 edebug-trace を非 nil の値に設定します。

トレース・バッファを作ることは、 trace 実行モードを使うこととは違います (15.4.3 節「Edebug 実行モード」 p.205 参照)。

トレース記録を有効にしてあるとき、各関数の呼出しと終了はトレース・バッファに行を追加します。関数の呼出し記録は '::::{' に関数名と引数値がついたような形になっています。関数の終了記録は '::::}' に関数名と関数の結果がついたような形になっています。

項目の中の ':' の数は再帰の深さを示します。トレース・バッファで、対応する関数呼出しの最初や最後を探すのに、波括弧を使うことができます。

関数 edebug-print-trace-before と、 edebug-print-trace-after を再定義して、関数の呼出しと終了のトレース記録をカスタマイズすることができます。

edebug-tracing *string body...* Macro

このマクロは *body* 形式の実行の前後で追加のトレース情報を要求します。引数 *string* はトレース・バッファに置くテキストを指定します。全引数を評価します。 edebug-tracing は *body* の最後の形式の値を返します。

edebug-trace *format-string* &rest *format-args* Function

この関数は、トレース・バッファにテキストを挿入します。テキストは (apply 'format *format-string format-args*) で算出します。項目をわけるため改行を追加します。

edebug-tracing と edebug-trace は、 Edebug がアクティブでないときでもトレース・バッファに行を追加します。

トレース・バッファへのテキストの追加では、最後に挿入した行が見えるようにウィンドウのスクロールもします。

15.4.13 網羅率検査

Edebug にはおおざっぱな網羅率検査と実行頻度の表示ができます。膳立てをされた式の評価の前後では、実行モードが Go-nonstop のときでも、みな頻度計数を累積してゆきます。網羅率検査はかなり高くつくので、 `edebug-test-coverage` が非 `nil` のときにだけ行ないます。コマンド `M-x edebug-display-freq-count` は頻度データと網羅率データを (記録していれば) 表示します。

edebug-display-freq-count Command

このコマンドは頻度計数データを、現在の定義の中の各行に表示します。

頻度計数は各行の後ろにコメントとして現われます。これは `undo` コマンドで取り消すことができます。計数は式の前の (、式の後の)、シンボルの最後の文字の下に現われます。同じ行の中では前の計数と同じ値の表示はしません。

文字 ‘`=`’ に式の計数がついたものは、式が評価されるたびにいつも同じ値を返していることを示します。これは Edebug が記録する唯一の網羅率情報です。

定義の頻度計数と網羅率情報を消すには、再び膳立てをします。

たとえば、ソースにブレークポイントをつけて `(fac 5)` を評価し、 `edebug-test-coverage` を `t` にした後で、ブレークポイントに到達すると、計数データは次のようになります。

```
(defun fac (n)
  (if (= n 0) (edebug))
;#6           1      0 =5
  (if (< 0 n)
;#5          =
      (* n (fac (1- n)))
;#    5               0
    1))
;#   0
```

コメント行は `fac` が 6 回呼ばれたことを示しています。最初の `if` ステートメントは同じ結果を 5 回返しています。 2 番目の `if` の条件についても同じことがいえます。 `fac` の再帰呼びだしはまったく返っていません。

15.4.14 外側の文脈

Edebug はデバッグ中のプログラムにたいして透過的になるように努めていますが、そうすることに完全には成功してません。 Edebug は `e` や評価リスト・バッファで式を評価するとき、一時的に外側の文脈を復帰して、透過的になるように努めています。この節では Edebug が復帰する文脈と、 Edebug が完全に透過的とはなりきれていないのはどういうところなのかを、正確に説明します。

15.4.14.1 いつ止めるかのチェック

トレース情報を作ったりプログラムを止めたりするかどうかを決める前であっても、 Edebugにはいるときは常に、あるデータを保存、復帰する必要があります。

- `max-lisp-eval-depth` と `max-specpdl-size` は、スタック上の Edebug の影響のために 1 増加させておきます。しかしながら、 Edebug 使用中でもスタックが不足することはありえます。
- キーボード・マクロの実行状態も保存復帰されます。 Edebug がアクティブな間、 `executing-macro` は `edebug-continue-kbd-macro` に束縛されます。

15.4.14.2 Edebug の表示更新

Edebug が何かを表示する必要のあるとき (たとえば、 trace モード中)、現在のウィンドウ構成 (25.16 節「ウィンドウ構成」 p.432 参照) を Edebug の「外側」から保存します。 (プログラムを継続して)Edebug から抜けたとき、前のウィンドウ構成を復帰します。

Emacs は休んでいるときだけ再表示を行ないます。通常、実行を継続したときに、ブレークポイントにおいてか、または、休まずにステップをした後か、または、途中で入力を読み込もうとして、プログラムは Edebug に戻ってきます。このような場合、 Emacs は「外側」の構成を再表示する機会を失います。割り込みがなければ、最後に Edebug がアクティブなときと同じウィンドウ構成が見えます。

何かを表示するため Edebug にはいると以下のデータの保存と復帰を生じますが、このうちいくつかはあえて、エラーやとりやめのシグナルが起きても復帰しません。

- どのバッファがカレントか、そしてカレント・バッファのポイントとマークの位置は、保存と復帰をします。
- `edebug-save-windows` が非 `nil` なら (15.4.14.2 節「Edebug の表示更新」 p.216 参照) 外側のウィンドウ構成は保存と復帰をします。

  エラーやとりやめではウィンドウ構成の復帰をしませんが、外側で選択されたウィンドウは `save-excursion` がアクティブでも、エラーやとりやめで再び選択されます。

  ウィンドウの開始位置とソース・コード・バッファの水平スクロールは復帰しませんが、 Edebug の中で表示は一貫性をもっています。
- `edebug-save-displayed-buffer-points` が非 `nil` なら、表示される各バッファのポイントの値は、保存、復帰されます。
- 変数 `overlay-arrow-position` と `overlay-arrow-string` は保存と、復帰をします。したがって再帰編集中に同じバッファのどこからでも安全に Edebug を起動できます。
- `cursor-in-echo-area` は局所的に `nil` に束縛し、カーソルがウィンドウに表示されるようにします。

15.4.15 Edebug の再帰編集

Edebug にはいってユーザからコマンドを実際に読み込んだとき、以下のデータも保存を (そして後で復帰を) します。

- 現在の一致データ。30.6 節「一致データ」p.534 を参照してください。
- `last-command`、 `this-command`、 `last-command-char`、 `last-input-char`、 `last-input-event`、 `last-command-event`、 `last-event-frame`、 `last-nonmenu-event`、 `track-mouse` についてですが、 Edebug 中で使用したコマンドは Edebug の外ではこれらの変数に影響を与えません。

  `this-command-keys` が返すキー・シーケンスは、 Edebug の中でコマンドを実行してしまうと変わってしまって、 Lisp ではキー・シーケンスを戻すことができません。

  Edebug は `unread-command-events` の値を保存、復帰することができません。この変数が自明でない値をもつ間に Edebug にはいると、デバッグ中のプログラムの実行に影響をあたえます。
- Edebug 中に複雑なコマンドを実行すると変数 `command-history` に追加されます。まれにこれが実行を変えることがあります。
- Edebug の中では、再帰の深さが Edebug の外側にくらべて一つ深くなります。これは自動的に更新される評価リストのウィンドウでは成立しません。
- `standard-output` と `standard-input` は、 `recursive-edit` によって `nil` に束縛されますが、 Edebug は評価の間、一時的にこれらを復帰します。
- キーボード・マクロ定義の状態は保存、復帰されます。 Edebug がアクティブな間、 `defining-kbd-macro` は `edebug-continue-kbd-macro` に束縛されます。

15.4.16 マクロ呼出しの膳立て

Edebug が Lisp マクロを呼び出す式の膳立てをするとき、処理を正しく行なうために付加的な情報が必要です。これは、マクロ呼出しのどの部分式が評価すべき形式なのかを、決めることができないためです (評価はマクロ本体の中に明示されているときや、結果の式が評価されるときあるいはそれ以降に起こることがあります)。 Edebug がこれを正しく処理できるよう、各マクロの呼出し形式を指定しなければなりません。それには、 `def-edebug-spec` を使い、与えたマクロへの呼出し形式を定義する必要があります。

def-edebug-spec *macro specification* Macro

> マクロ *macro* の呼出しのどの式が評価すべき形式なのかを指定します。単純なマクロにたいして、 *specification* はいつもマクロ引数の仮引数とよく似ていますが、指定方法はもっとマクロ引数よりもさらに一般的です。
>
> *macro* 引数はマクロ名だけでなく実際には任意のシンボルでかまいません。

以下の例では、この Emacs Lisp リファレンス・マニュアル中に登場するマクロ `for` の仕様を、 2 とおりの等価な仕様指定で定義しています。

```
(def-edebug-spec for
  (symbolp "from" form "to" form "do" &rest form))

(def-edebug-spec for
  (symbolp ['from form] ['to form] ['do body]))
```

次の表に *specification* のとりうる値と、それによって引数の処理がどう変わるかをまとめておきます。

`t` 全引数について評価の膳立てをします。

`0` 膳立てをする引数はありません。

シンボル シンボルは、代用する Edebug 仕様指定をもたなければなりません。この間接的な仕様指定は別種の仕様指定が見つかるまで続きます。別のマクロの仕様指定を継承することができます。

リスト リストの要素は呼出し形式の引数の型を指定します。仕様指定リストの要素になりえるものは次の節で解説します。

15.4.16.1 仕様指定リスト

マクロ呼出しの、ある引数は評価され、ある引数は評価されないという場合には、『仕様指定リスト (specification list)』が Edebug への指定として必要です。仕様指定リスト中のある要素群は、単数あるいは複数の引数と一致し、別な要素群はそれ以降の要素の処理を修飾します。後者の方は『仕様指定キーワード (specification keyword)』といいます。これは (`&optional` のように)'`&`' で始まるシンボルです。

仕様指定リストは、それ自身がリストであるような引数に対応する部分リストや、仲間わけに使うベクタを含んでいることがあります。このように部分リストやグループは仕様指定リストをレベルの階層構造に分割します。仕様指定キーワードは、それを含むグループや部分リストの残りの部分だけに適用します。

仕様指定リストが選択肢や繰り返しになるとき、実際のマクロ呼出しへの照合にはバックトラックが必要です。詳しくは、15.4.16.2 節「バックトラック」p.221 を参照してください。

Edebug 仕様指定は正規表現の照合能力にくわえて、文脈自由文法としての構文要素をもっています。すなわち、釣り合いのとれた括弧をもつ部分リスト、形式の再帰的処理、間接仕様指定を経由した再帰といったものです。

次の表はとりうる仕様指定の要素とその意味です。

`sexp` 評価されない単一の Lisp オブジェクト。

`form` 評価され膳立てをされる単一の式。

`place` Common Lisp の `setf` 構文要素のように、値を格納するところ。

`body` `&rest form` を簡単にしたもの。以下の `&rest` を見てください。

`function-form`
: 関数形式。すなわち quote された関数シンボル、 quote されたラムダ式、あるいは (関数シンボルやラムダ式へ評価する) 形式。これはラムダ式を渡す引数が、`function` ではなく `quote` で quote されることもある場合に、便利です。というのはどちらの場合でもラムダ式本体の膳立てをするからです。

`lambda-expr`
: quote しないラムダ式。

`&optional` 仕様指定リストでそれ以降の引数が省略可能であることをあらわします。一つでも一致しなくなるやいなや、 Edebug はこのレベルで照合を止めます。

省略不能な要素に続いていくつかの省略可能な要素がある場合、 `[&optional` *specs...*`]` を使います。いくつかの要素がすべて一致するかまったく一致しないかということを指定するには、 `&optional [`*specs...*`]` を使います。以下の `defun` の例を見てください。

`&rest` 仕様指定リスト中のそれ以降の要素は 0 個以上繰り返します。最後の繰り返しについてはすべてと照合する必要はありません。

いくつかの要素だけを繰り返すには、 `[&rest` *specs...*`]` を使います。すべての繰り返しで全部が一致する必要のあるときは、 `&rest [`*specs...*`]` を使います。

`&or` 仕様指定リスト中の後続する全要素が選択肢です。選択肢の一つが照合する必要があります。さもなくば `&or` 仕様指定は失敗します。

`&or` に続く各リスト要素は、単一の選択肢です。二つ以上のリスト要素を単一の選択肢とするには、 `[...]` でくくります。

`&not` 後続の各要素は `&or` を使ったときの選択肢のように照合されますが、一致するものがあると指定は失敗します。一致するものがないとき、照合するものは空ですが、このときに `&not` 仕様指定は成功します。

`&define` 仕様指定が定義を行なう形式であることを示します。定義を行なう形式自身は膳立てをされません (つまり Edebug は定義を行なう形式の前後では止まりません) が、内側の典型的な形式は膳立てをされます。 `&define` キーワードはリスト仕様指定の最初の要素でなければなりません。

`nil` これは現在の引数リスト・レベルで照合すべき引数がないときに成功します。後述する部分リスト仕様指定と逆引用符の例を見てください。

`gate` 引数とは対応しませんが、このレベルでの残った仕様指定の照合で gate をとおったバックトラックを禁止します。これは主に仕様指定の構文エラー・メッセージの生成に使用されます。詳しくは 15.4.16.2 節「バックトラック」 p.221 を見てください。また、後述の `let` の例も見てください。

other-symbol
: 仕様指定リスト中のほかのすべてのシンボルは、述語または間接的な仕様指定のはずです。

シンボルに Edebug 仕様指定があるなら、この『間接的仕様指定 (indirect specification)』はシンボルの代わりに使われる仕様指定リストであるか、あるいは引数を処理するために呼び出される関数です。仕様指定はマクロのように `def-edebug-spec` で定義されます。後述の `defun` の例を見てください。

それ以外の場合、シンボルは述語でなければなりません。述語は引数をともなって呼び出され、この述語が `nil` を返すと仕様指定は失敗します。いずれの場合も、引数の膳立てはされません。

適切な述語には、`symbolp`、`integerp`、`stringp`、`vectorp`、`atom` があります。

`[`*elements...*`]`

要素のベクタは、複数の要素を単一の『グループ仕様指定 (group specification)』にまとめます。この意味はベクタとは関係ありません。

`"`*string*`"` 引数は *string* という名前のシンボルでなければなりません。この仕様指定は、*string* という名前のシンボル *symbol* を quote した `'`*symbol* というものと等価ですが、文字列形式の方がお薦めです。

`(vector` *elements...*`)`

引数は、その要素が仕様指定中の *elements* と一致するベクタでなければなりません。後述の逆引用符の例を見てください。

`(`*elements...*`)`

それ以外のリストは『部分リスト仕様指定 (sublist specification)』です。この引数はリストで、その要素が仕様指定 *elements* と一致していなければなりません。

部分リスト仕様指定がドットつきリストであって、対応するリスト引数がドットつきリストであってもかまいません。また、ドットつきリスト仕様指定の最後の CDR が、別な部分リスト仕様指定 (仲間わけや間接仕様指定、すなわち、`(spec . [(more specs...)])`) で、その要素がドットのないリスト引数と一致するものであったりしても、かまいません。これは、後述の逆引用符の例にあるような再帰的仕様指定で、役に立ちます。このような再帰を終了させる `nil` 仕様指定については、前述しました。

`(specs . nil)` のように書かれる部分リスト仕様指定は、`(specs)` と等価であること、`(specs . (sublist-elements...))` は、`(specs sublist-elements...)` と等価であることに注意してください。

以下は `&define` の後に現われるだけの追加の仕様指定の一覧です。後述する `defun` の例を見てください。

`name` 引数のシンボルは定義形式中の名前です。

定義形式には名前のフィールドがなくともかまいません。複数の名前フィールドがあることもあります。

`:name` この構文要素は実際の引数とは対応しません。`:name` に続く要素はシンボルでなければなりません。これは定義にたいして、付加的な名前の成分に使われます。一

意で静的な成分を定義中の名前に追加することができます。1 回以上使うこともできます。

`arg` 引数のシンボルは定義形式の引数の名前です。しかし、ラムダ・リスト・キーワード (‘`&`’ で始まるシンボル) は許されていません。

`lambda-list`
これはラムダ・リスト、すなわちラムダ式の引数リストと一致します。

`def-body` 引数は定義のコードの本体です。これは前述の `body` に似ていますが、定義の本体は、定義にまつわる情報を参照する別な Edebug 呼出しで、膳立てをされていなければなりません。`def-body` は、定義内の形式の最高位のリストに使用します。

`def-form` 引数は定義内で最高位の単一の形式です。`def-body` に似ていますが、形式のリストではなく、単一の形式と対応します。特例として `def-form` では、形式を実行するとき追跡情報を出力しません。後述の `interactive` の例を見てください。

15.4.16.2 バックトラック

仕様指定がどこかで照合に失敗しても、それだけでは構文エラーを通知されるとはかぎりません。こういう場合、選択肢がなくなるまで『バックトラック (backtracking)』を行ないます。結局のところ、引数リストの各要素は仕様指定中のいずれかの要素と一致しなければなりませんし、仕様指定中で必須と指定した要素にはいずれかの引数と一致しなければなりません。

後述するある条件の成立した場合、バックトラックは部分リストやグループの余りの部分にたいして禁止されます。バックトラックは新しい選択肢が `&optional`、`&rest` や `&or` で確立されたときに、再び有効になります。部分リスト、グループ仕様指定、間接仕様指定を処理する前にも有効になります。

Edebug がもっと細かな構文エラー・メッセージを出せるように、バックトラックを禁止し別な選択肢をえらぶようにしたいかもしれません。通常、一致する選択肢がないとき、Edebug は一致したものがなかったということを知らせてきますが、一つしかない選択肢の場合には、どうして一致しなかったのかまで知らせることができます。

まず第一に形式の仕様指定のどれか (つまり `form`、`body`、`def-form`、`def-body`) に照合しているときは、バックトラックが禁止されます。これらの仕様指定はどんな形式にも一致するため、どんなエラーも高い水準でではなく形式それ自身の中で起こります。

そして第二に quote されたシンボルや文字列仕様指定とうまく一致した後にも、認識された構文要素を示しているということで、バックトラックは禁止されます。もし同じシンボルで始まる選択肢の構文要素の集まりがあるなら、`["foo" &or [first case] [second case] ...]` のようにシンボルを選択肢からくくり出すことにより、この制約をうまく回避することができます。

第三に `gate` 仕様指定を使って陽にバックトラックを禁止することもできます。これ以上高い選択肢がないとわかっているときには便利です。

15.4.16.3 仕様指定の例

以下の例で Edebug 仕様指定について学習すれば、理解が深まることでしょう。

let 特殊形式には束縛の列と本体があります。各束縛はシンボルであるか、あるいはシンボルと省略可能な値をともなった部分リストです。以下の仕様指定でいったん部分リストが見つかると、部分リストの内側の gate がバックトラックを妨げます。

```
(def-edebug-spec let
  ((&rest
    &or symbolp (gate symbolp &optional form))
   body))
```

defun、 defmacro、そして関連づけられた引数リストと interactive 仕様指定で Edebug が使用する仕様指定は、以下のとおりです。対話的な形式は、実際に関数本体の外側で評価される引数式であるため、特別扱いすることが必要です。

```
(def-edebug-spec defmacro defun) ; defun 仕様指定への間接参照。
(def-edebug-spec defun
  (&define name lambda-list
           [&optional stringp]   ; もしあれば、説明文字列と一致する。
           [&optional ("interactive" interactive)]
           def-body))

(def-edebug-spec lambda-list
  (([&rest arg]
    [&optional ["&optional" arg &rest arg]]
    &optional ["&rest" arg]
    )))

(def-edebug-spec interactive
  (&optional &or stringp def-form))    ; 注意: def-form
```

以下の逆引用符の仕様指定ではドットつきリストや再帰を終結させる nil の使い方が、説明されています。また、ベクタの成分と照合させる方法も説明します (再帰が深くなりすぎてだめなので、 Edebug で実際に定義している仕様指定では、ドットつきリストをサポートしていません)。

```
(def-edebug-spec ` (backquote-form))   ; 簡単のための別名。

(def-edebug-spec backquote-form
  (&or ([&or "," ",@"] &or ("quote" backquote-form) form)
       (backquote-form . [&or nil backquote-form])
       (vector &rest backquote-form)
       sexp))
```

15.4.17 Edebug のオプション

Edebug の振舞いを変えるオプションは以下のとおりです。

edebug-setup-hook User Option

Edebug を使う前に呼び出される関数群。新しい値が設定されるたびに Edebug はこれらの諸関数を 1 回ずつ評価し、その後 edebug-setup-hook を nil にします。使っているパッケージにまつわる Edebug 仕様指定を、 Edebug とともに使うときにだけロードするようなときに使います。 15.4.2 節「Edebug のために膳立てをする」 p.204 を参照してください。

edebug-all-defs User Option

もし非 nil なら、 defun や defmacro のような通常の定義形式の評価で、 Edebug への膳立てをします。これは eval-defun、 eval-region、 eval-buffer、 eval-current-buffer に適用されます。

コマンド M-x edebug-all-defs は、このオプションのトグルを行ないます。 15.4.2 節「Edebug のために膳立てをする」p.204 を参照してください。

edebug-all-forms User Option

もし非 nil なら、コマンド eval-defun、 eval-region、 eval-buffer、 eval-current-buffer は、定義をしない場合でも、全形式の膳立てを行ないます。ロードとミニバッファでの評価には適用しません。

コマンド M-x edebug-all-forms は、このオプションのトグルを行ないます。 15.4.2 節「Edebug のために膳立てをする」p.204 を参照してください。

edebug-save-windows User Option

もし非 nil ならば、 Edebug はウィンドウ構成の保存と復帰をします。これには時間がかかりますから、もしウィンドウ構成を無視するようなプログラムでは、この変数を nil にするとよいでしょう。

値がリストのとき、リスト中のウィンドウだけが保存復帰されます。

Edebug の W コマンドは、この変数を対話的に変更します。 15.4.14.2 節「Edebug の表示更新」p.216 を参照してください。

edebug-save-displayed-buffer-points User Option

これが非 nil のとき Edebug は、表示中の全バッファのポイントの保存と復帰をします。

別なバッファのポイントを保存復帰することは、選択されていないウィンドウに表示されているバッファのポイントを変更するコードをデバッグするときに必要になります。 Edebug あるいはユーザがウィンドウを選択すると、バッファのポイントはウィンドウのポイント値に移動します。

全バッファでポイントの保存と復帰をするのには、各ウィンドウを 2 回選択する必要があるので、高くつきます。ですから必要なときだけ使うようにしてください。 15.4.14.2 節「Edebug の表示更新」p.216 を参照してください。

edebug-initial-mode User Option

この変数が非 nil のとき、 Edebug が最初にアクティブになったときの実行モードをとります。とりうる値は、 step、 next、 go、 Go-nonstop、 trace、 Trace-fast、 continue、 Continue-fast のいずれかです。

デフォルト値は step です。

15.4.3 節「Edebug 実行モード」p.205 を参照してください。

edebug-trace User Option

非 nil のとき関数の呼出しと終了でトレースを表示します。トレース出力は '*edebug-trace*' という名前のバッファに 1 関数の呼出し終了ごとに 1 行ずつ、再帰レベルに応じた字下げをして表示されます。

デフォルト値は nil です。

15.4.12 節「トレース・バッファ」p.214 を参照してくださいの edebug-tracing も参照してください。

edebug-test-coverage User Option

もし非 nil なら、 Edebug はデバッグ中のすべての式の網羅率検査をします。各式の結果を前の結果と比較することによります。二つの異なる値が見つかったとき網羅率は良好とみなされます。したがってあなたのコードの網羅率を十分に検査するには、すべての式を (訳注: 少なくとも)1 回以上は評価するような、また、各式が異なる結果を生成するような、条件下で実行するようにしてください。

頻度計数と定義の網羅率情報を表示するには M-x edebug-display-freq-count を使います。 15.4.13 節「網羅率検査」p.215 を参照してください。

edebug-continue-kbd-macro User Option

もし非 nil なら、 Edebug の外側で実行しているキーボード・マクロの実行や、定義を続行します。 15.4.3 節「Edebug 実行モード」p.205 を参照してください。

edebug-print-length User Option

もし非 nil なら、 Edebug で結果を印字している間 print-length をこの値に束縛します。デフォルト値は 50 です。 15.4.11 節「Edebug での印字」p.213 を参照してください。

edebug-print-level User Option

もし非 nil なら、 Edebug で結果を印字している間 print-level をこの値に束縛します。デフォルト値は 50 です。

edebug-print-circle User Option

もし非 nil なら、 Edebug で結果を印字している間 print-circle をこの値に束縛します。デフォルト値は nil です。

edebug-on-error User Option

debug-on-error の値が nil のとき、 Edebug は debug-on-error をこの値に束縛します。 15.4.7 節「エラーのトラップ」p.210 を参照してください。

edebug-on-quit User Option

debug-on-quit の値が nil のとき、 Edebug は debug-on-quit をこの値に束縛します。 15.4.7 節「エラーのトラップ」p.210 を参照してください。

Edebug がアクティブのときに、 edebug-on-error や edebug-on-quit の値を変えても、次に Edebug が別なコマンドから起動されるまでは無効です。

edebug-global-break-condition User Option

もし非 nil なら各ストップ・ポイントで検査される式です。結果が非 nil ならブレークします。エラーは無視します。 15.4.6.1 節「グローバル・ブレーク条件」 p.210 を参照してください。

16 Lisp オブジェクトの読込みと印字

『印字 (printing)』と『読込み (reading)』は、 Lisp オブジェクトをテキスト形式へ、またその逆に変換するための操作です。これらは、第 2 章「Lisp のデータ型」p.19 で述べられている印字表現とリード構文を扱うものです。

この章では、読込みと印字のための Lisp 関数について述べます。また、どこからテキストを得るか (読込みの場合)、あるいはどこへ出力するか (印字の場合) を指定する、『ストリーム (stream)』についても述べます。

16.1 読込みと印字の概論

Lisp オブジェクトの『読込み (reading)』とは、テキスト形式の Lisp 式を解析して、対応する Lisp オブジェクトを生成することを意味します。これは、 Lisp プログラムが、 Lisp コードのファイルから Lisp の世界に入り込むための方法です。このテキストを、そのオブジェクトの『リード構文 (read syntax)』と呼びます。たとえばテキスト '`(a . 5)`' は、 CAR が `a` で CDR が数値 5 であるコンス・セルのリード構文です。

Lisp オブジェクトの『印字 (printing)』とは、オブジェクトを表現するテキストの生成、つまりオブジェクトからその印字表現への変換を意味します。上述のコンス・セルの印字では、 '`(a . 5)`' というテキストが生成されます。

読込みと印字は、多かれ少なかれ逆の操作であるといえます。与えられた一片のテキストの読込みによって生成されるオブジェクトの印字は、たいてい同じテキストを生成し、あるオブジェクトの印字によって生成されるテキストの読込みは、通常、同様なオブジェクトを生成します。たとえば、シンボル `foo` の印字はテキスト '`foo`' を生成し、そのテキストの読込みはシンボル `foo` を返します。要素が `a` と `b` であるリストの印字はテキスト '`(a b)`' を生成し、そのテキストの読込みは、 `a` と `b` を要素とするリスト (同じリストそのものではありませんが) を生成します。

しかしながら、厳密にはこれら二つの操作は逆ではありません。三種類の例外があります:

- まず、印字は、読み込むことのできないテキストを生成することがあります。たとえば、バッファ、ウィンドウ、フレーム、子プロセス、マーカは、'`#`' で始まるテキストとして印字されますが、このテキストを読み込もうとしてもエラーとなります。これらのデータ型を読み込む方法はありません。
- 次に、一つのオブジェクトは、複数のテキスト表現をもつこともあります。たとえば、 '`1`' と '`01`' は同じ整数を表わし、 '`(a b)`' と '`(a . (b))`' は同じリストを表わします。読込みではいずれの表現も受け入れますが、印字ではそれらのうち一つを選択しなくてはなりません。
- さらに、あるオブジェクトを読み込んでいる途中の位置に、その読込みの結果に影響を与えることなく、コメントを入れることができます。

16.2 入力ストリーム

テキスト読込み用の Lisp 関数のほとんどは、引数として『入力ストリーム (input stream)』を取ります。入力ストリームは、読み込むべきテキスト文字群をどこから、あるいはどのように得るのかを指定するものです。入力ストリームとして使用可能な型には、以下のようなものがあります:

バッファ
: 入力文字群は、バッファから読み込まれます。ポイント直後の文字から読み始められます。ポイントは、読み込まれた文字数分だけ進みます。

マーカ
: 入力文字群は、マーカがあるバッファから読み込まれます。そのマーカの直後の文字から読み始められます。マーカの位置は、読み込まれた文字数分だけ進みます。ストリームがマーカである場合、そのバッファのポイント値は何ら影響を受けません。

文字列
: 入力文字群は、文字列から取られます。その文字列の先頭の文字から読み始められ、必要なだけの文字が使われます。

関数
: 入力文字群は、関数によって、その呼出しごとに一文字ずつ生成されます。普通、その関数は、引数なしで呼び出されて一文字を返すものです。

  時折、関数は、一つの引数 (常に一文字) を以て呼び出されることがあります。その場合、関数はその引数を保存し、次回の呼出しでそれを返すように準備します。これは、文字の『読戻し (unreading)』と呼ばれます。 Lisp リーダが一文字余分に読み込んでしまい、“それを読込み元へ戻したい” というときに行なわれます。

`t`
: ストリームとして用いられる `t` は、入力がミニバッファから読み込まれることを意味します。実際には、一度ミニバッファが呼び出され、そこでユーザによって与えられたテキストが、後で入力ストリームとして用いられる文字列へと変換されます。

`nil`
: 入力ストリームとして与えられた `nil` は、その代わりに変数 `standard-input` の値を使用することを意味します。その値は、『デフォルト入力ストリーム (default input stream)』であり、非 `nil` 入力ストリームでなくてはなりません。

シンボル
: 入力ストリームとしてのシンボルは、そのシンボルの関数定義 (もしあれば) と等価です。

バッファ・ストリームからの読込みの例を、その前後でポイントがどこに位置するかを記しながら、以下に示します:

```
---------- Buffer: foo ----------
This★ is the contents of foo.
---------- Buffer: foo ----------

(read (get-buffer "foo"))
     ⇒ is
(read (get-buffer "foo"))
     ⇒ the
```

```
---------- Buffer: foo ----------
This is the⋆ contents of foo.
---------- Buffer: foo ----------
```

最初の読込みでスペースをスキップしていることに注意してください。読込みは、意味のあるテキストに先行するどんな量の空白もスキップします。

Emacs 18 においては、シンボルの読込みは、そのシンボルを終結しているデリミタを捨てていました。つまりポイントは、'the' の後ろではなく 'contents' の先頭に位置したでしょう。Emacs 19 の挙動は、あるオブジェクトの終りとなる開括弧が別のオブジェクトの先頭として必要となる、'bar(foo)' のような入力を正しく扱えるので、より優れています。

以下に、バッファの先頭に配置されたマーカ・ストリームからの読込みの例を示します。read の値は、シンボル This です。

```
---------- Buffer: foo ----------
This is the contents of foo.
---------- Buffer: foo ----------

(setq m (set-marker (make-marker) 1 (get-buffer "foo")))
     ⇒ #<marker at 1 in foo>
(read m)
     ⇒ This
m
     ⇒ #<marker at 5 in foo>   ;; 最初のスペースの前。
```

文字列の内容から読み込んでみましょう:

```
(read "(When in) the course")
     ⇒ (When in)
```

以下の例では、ミニバッファから読み込んでいます。プロンプトは'Lisp expression: 'です (ストリーム t から読み込むときは、常にこのプロンプトが用いられます)。ユーザの入力は、そのプロンプトに続けて表示されます。

```
(read t)
     ⇒ 23
---------- Buffer: Minibuffer ----------
Lisp expression: 23 RET
---------- Buffer: Minibuffer ----------
```

最後に、useless-stream と名づけられた関数ストリームの例を示します。このストリームを使用する前には、変数 useless-list を文字のリストで初期化します。その後、関数 useless-stream を呼び出すごとに、このリスト中の次の文字を獲得するか、あるいはリストの先頭に文字をつけ加えることによりその文字を読み戻します。

```
(setq useless-list (append "XY()" nil))
     ⇒ (88 89 40 41)
```

```
(defun useless-stream (&optional unread)
  (if unread
      (setq useless-list (cons unread useless-list))
    (prog1 (car useless-list)
           (setq useless-list (cdr useless-list)))))
     ⇒ useless-stream
```

では、このように構築されたストリームを使用して読み込んでみます:

```
(read 'useless-stream)
     ⇒ XY

useless-list
     ⇒ (40 41)
```

リスト中に開括弧と閉括弧が残っていることに注意してください。Lisp リーダは開括弧に出くわし、それが入力の終りであると判断して開括弧を読み戻したのです。ここで、もう一度このストリームからの読込みを試みると、'()' を読み込み、 `nil` を返すでしょう。

get-file-char Function

この関数は、 `load` 関数でオープンされた入力ファイルからの読込みを行なう入力ストリームとして、内部的に使用されます。自分自身では、この関数を使用しないでください。

16.3 入力関数

この節では、読込みに関する Lisp 関数と変数について述べます。

以下の関数群において、 *stream* は、入力ストリーム (前節参照) を表わしています。 *stream* が `nil` であるか、あるいは省略された場合、変数 `standard-input` の値がそのデフォルトとなります。

読込みが、終結されていないリスト、ベクタ、あるいは文字列に出会うと、 `end-of-file` エラーが通知されます。

read &optional *stream* Function

この関数は、 *stream* からテキスト形式の Lisp 式を一つ読み込み、それを Lisp オブジェクトとして返します。これは、基本的な Lisp 入力関数です。

read-from-string *string* &optional *start end* Function

この関数は、 *string* 内のテキストから最初の Lisp 式を読み込みます。そして、 CAR がその式で、 CDR が *string* 内の次の文字 (まだ読み込まれていない最初の文字) の位置を示す整数である、コンス・セルを返します。

start が与えられた場合、読込みは、 *string* 内の添字 *start* の位置から始められます (最初の文字が添字 0)。また、 *end* が与えられた場合、読込みは、文字列の残りがなかったかのように、その添字の位置の直前で停止します。

例:

```
(read-from-string "(setq x 55) (setq y 5)")
     ⇒ ((setq x 55) . 11)
(read-from-string "\"A short string\"")
     ⇒ ("A short string" . 16)
;; 最初の文字から読み始める。
(read-from-string "(list 112)" 0)
     ⇒ ((list 112) . 10)
;; 2 番目の文字から読み始める。
(read-from-string "(list 112)" 1)
     ⇒ (list . 5)
;; 7 番目の文字から読み始めて、
;;    9 番目で停止する。
(read-from-string "(list 112)" 6 8)
     ⇒ (11 . 8)
```

standard-input Variable

この変数は、 `read` が、 *stream* 引数が `nil` のときに用いる、デフォルト入力ストリームを持っています。

16.4 出力ストリーム

出力ストリームは、印字によって生成される文字で以て何をするのかを指定します。ほとんどの印字関数は、オプション引数として出力ストリームを受け付けます。出力ストリームとして使用可能な型には、以下のようなものがあります:

バッファ
: 出力文字群は、バッファのポイントへ挿入されます。ポイントは、挿入された文字数分だけ進みます。

マーカ
: 出力文字群は、マーカが指しているバッファの、そのマーカの位置へ挿入されます。マーカの位置は、挿入された文字数分だけ進みます。ストリームがマーカである場合、そのバッファのポイント値は何ら影響を受けません。

関数
: 出力文字群は、それらを別の場所へストアする責任をもつ関数に渡されます。その関数は、引数として一文字を取り、出力されるべき文字数分の回数呼ばれます。受け取った文字で何をするかはまったく自由です。

`t`
: 出力文字群は、エコー領域に表示されます。

`nil`
: 出力ストリームとして指定された `nil` は、その代わりに変数 `standard-output` の値を使用することを意味します。その値は、『デフォルト出力ストリーム (default output stream)』であり、非 `nil` 出力ストリームでなくてはなりません。

シンボル
: 出力ストリームとしてのシンボルは、そのシンボルの関数定義 (もしあれば) と等価です。

正当な出力ストリームの多くは、また入力ストリームとしても正当です。したがって入力と出力ストリームの違いは、主として Lisp オブジェクトをどう使用するかの違いであり、オブジェクトの型の区別ではありません。

出力ストリームとして使用されるバッファの例を以下に示します。見てのとおり、ポイントは最初、'the' の 'h' の直前に位置しています。しかし最後にも、ポイントは同じ 'h' の直前に位置しています。

```
---------- Buffer: foo ----------
This is t⋆he contents of foo.
---------- Buffer: foo ----------

(print "This is the output" (get-buffer "foo"))
     ⇒ "This is the output"

---------- Buffer: foo ----------
This is t
"This is the output"
⋆he contents of foo.
---------- Buffer: foo ----------
```

では、出力ストリームとしてのマーカの使用を見てみます。マーカは最初、バッファ foo 中の単語 'the' の 't' と 'h' の間にあります。そして最後には、そのマーカは、同じ 'h' の前に位置し続けるよう、挿入されたテキストを跨いで進みました。いつものように示されたポイントの位置に注意してください。その影響を受けていません。

```
---------- Buffer: foo ----------
"This is the ⋆output"
---------- Buffer: foo ----------

m
     ⇒ #<marker at 11 in foo>

(print "More output for foo." m)
     ⇒ "More output for foo."

---------- Buffer: foo ----------
"This is t
"More output for foo."
he ⋆output"
---------- Buffer: foo ----------

m
     ⇒ #<marker at 35 in foo>
```

以下の例は、エコー領域への出力を示しています:

```
(print "Echo Area output" t)
     ⇒ "Echo Area output"
---------- Echo Area ----------
"Echo Area output"
---------- Echo Area ----------
```

最後に、出力ストリームとしての関数の使用を見てみます。関数 eat-output は、与えられた各文字を取り、それをリスト last-output の先頭へ cons します (5.5 節「コンス・セルとリストの生成」p.71 参照)。そして最後には、そのリストは出力されたすべての文字を含みますが、その内容は逆順になっています。

```
(setq last-output nil)
     ⇒ nil

(defun eat-output (c)
  (setq last-output (cons c last-output)))
     ⇒ eat-output

(print "This is the output" 'eat-output)
     ⇒ "This is the output"

last-output
     ⇒ (10 34 116 117 112 116 117 111 32 101 104
    116 32 115 105 32 115 105 104 84 34 10)
```

このリストを逆にすることによって、正しい順番で出力を出せるようになります:

```
(concat (nreverse last-output))
     ⇒ "
\"This is the output\"
"
```

concat の呼出しは、リストを文字列へと変換するため、より明瞭にその内容を見ることができます。

16.5 出力関数

この節では、 Lisp オブジェクトの印字に用いる Lisp 関数について述べます。

一部の Emacs 印字関数は、その出力が正しく読み込まれ得るよう、必要に応じて出力に quote 文字を付加します。 Quote 文字は '"' と '\' です。これらは文字列とシンボルとを区別し、また読込み時に文字列中の句読点文字がデリミタとして取られるのを防ぎます。完全な詳細については、 2.1 節「印字表現とリード構文」p.19 を参照してください。 Quote するかしないかの指定は、印字関数の選択によって行ないます。

もし、そのテキストが再び Lisp 中に読み込まれるものであるなら、曖昧さを避けるため、quote 文字をともなって印字するのが一番です。 Lisp プログラマのために明確に Lisp オブジェクトを描写するのが目的なら、その場合も同様です。しかし、出力の目的が人間にとって見やすくすることであるなら、 quote なしで印字するほうがよいでしょう。

自己参照型 Lisp オブジェクトの印字は、無限量のテキストを必要とします。場合によっては、そのようなテキストの生成の試みは、スタック・オーバーフローを引き起こします。 Emacs はそのような再帰を検出し、すでに印字されているオブジェクトを再帰的に (訳注: 繰り返し) 印字する代わりに、 '#*level*' と印字します。たとえば以下での '#0' は、現在の印字操作のレベル 0 におけるオブジェクトの再帰的参照を示しています:

```
(setq foo (list nil))
     ⇒ (nil)
(setcar foo foo)
     ⇒ (#0)
```

以下の関数群において、 *stream* は、出力ストリームを表わしています (出力ストリームの解説については前節を参照)。 *stream* が `nil` であるか、あるいは省略された場合、変数 `standard-output` の値がそのデフォルトとなります。

print *object* &optional *stream* Function

`print` 関数は、印字のための便利な手段です。これは、 *object* の前の改行に加えて、後ろにも改行を付加しつつ、 *object* の印字表現を *stream* へ出力します。 Quote 文字が用いられます。 `print` は *object* を返します。以下がその例です:

```
(progn (print 'The\ cat\ in)
       (print "the hat")
       (print " came back"))
     ⊣
     ⊣ The\ cat\ in
     ⊣
     ⊣ "the hat"
     ⊣
     ⊣ " came back"
     ⊣
     ⇒ " came back"
```

prin1 *object* &optional *stream* Function

この関数は、 *object* の印字表現を *stream* へ出力します。 `print` のような出力を区切るための改行は印字しませんが、 quote 文字は `print` 同様に用います。 *object* を返します。

```
(progn (prin1 'The\ cat\ in)
       (prin1 "the hat")
       (prin1 " came back"))
     ⊣ The\ cat\ in"the hat"" came back"
     ⇒ " came back"
```

princ *object* &optional *stream* Function

この関数は、 *object* の印字表現を *stream* へ出力します。 *object* を返します。

この関数は、 `read` ではなく人間が読めるような出力を生成することを意図しており、そのため quote 文字は挿入せず、文字列の前後に二重引用符もつけません。呼出しごとの字空けも行ないません。

```
(progn
  (princ 'The\ cat)
  (princ " in the \"hat\""))
     ⊣ The cat in the "hat"
     ⇒ " in the \"hat\""
```

terpri &optional *stream* Function

この関数は、 *stream* に改行を出力します。この名前は、 “terminate print” を表わしています。

write-char *character* &optional *stream* Function

この関数は、 *character* を *stream* に出力します。 *character* を返します。

prin1-to-string *object* &optional *noescape* Function

この関数は、 `prin1` に同じ引数を与えたときに印字されるテキストを内容とする、文字列を返します。

```
(prin1-to-string 'foo)
     ⇒ "foo"
(prin1-to-string (mark-marker))
     ⇒ "#<marker at 2773 in strings.texi>"
```

noescape が非 `nil` の場合、出力における quote 文字の使用が抑制されます (この引数は、 Emacs バージョン 19 以降でサポートされています)。

```
(prin1-to-string "foo")
     ⇒ "\"foo\""
(prin1-to-string "foo" t)
     ⇒ "foo"
```

Lisp オブジェクトの印字表現を文字列として得るための他の方法については、 4.5 節「文字と文字列間の変換」 p.60 の `format` を参照してください。

16.6 出力に影響を与える変数

standard-output Variable

この変数の値は、印字関数が、 *stream* 引数が `nil` のときに用いる、デフォルト出力ストリームです。

print-escape-newlines Variable

この変数が非 `nil` の場合、文字列中の改行文字は '`\n`' として印字され、改ページ文字は '`\f`' として印字されます。普通、これらの文字は、実際の改行、改ページとして印字されます。

この変数は、関数 `prin1` と `print`、またそれらを使用するすべてに影響を与えます。 `princ` には影響を与えません。以下に、 `prin1` を使用する例を示します:

```
(prin1 "a\nb")
     ⊣ "a
     ⊣ b"
     ⇒ "a
b"

(let ((print-escape-newlines t))
  (prin1 "a\nb"))
     ⊣ "a\nb"
     ⇒ "a
b"
```

二つ目の式において、 `prin1` の呼出しの間は `print-escape-newlines` の局所的な束縛が効いていますが、その結果の印字のときには効いていません。

print-length Variable

この変数の値は、印字されるリストの要素の最大数です。印字されるリストが、これよりも多くの要素をもっている場合、省略符号 (訳注: '...') によって短縮されます。

この値が `nil` の場合 (デフォルト)、制限はありません。

```
(setq print-length 2)
     ⇒ 2
(print '(1 2 3 4 5))
     ⊣ (1 2 ...)
     ⇒ (1 2 ...)
```

print-level Variable

この変数の値は、丸括弧と角括弧が印字される際のネストの最大の深さです。この制限を越える深さのリスト、あるいはベクタは、省略符号 (訳注: '...') によって短縮されます。`nil` の値 (デフォルト) は、制限しないことを意味します。

この変数は、バージョン 19 以降に存在します。

17 ミニバッファ

『ミニバッファ (minibuffer)』は Emacs のコマンドが使う特殊なバッファで、一つの数値前置引数よりも複雑な複数の引数を読み込むのに使います。これら引数はファイル名、バッファ名、そして (`M-x` での) コマンド名を含みます。ミニバッファはエコー領域と同じ場所である、スクリーンの最下行に、一つの引数を読み込むために使う間だけ表示されます。

17.1 ミニバッファの紹介

ほとんどの部分において、ミニバッファは普通の Emacs のバッファです。バッファ内で使うほとんどの操作、編集コマンドなどは、ミニバッファ内で普通に動作します。けれどもバッファ自体を扱う操作のほとんどは、ミニバッファに対しては適用されません。ミニバッファの名前は常に'`*Minibuf-`*number*'の形式を持ち、変更はできません。ミニバッファはミニバッファだけに使われる専用のウィンドウだけに表示されます。これらウィンドウは常にフレームの下部に現れます (時にはフレームがミニバッファ・ウィンドウを持たない場合や、時には特別な種類のフレームがミニバッファ・ウィンドウだけしか含んでないこともあります。 26.8 節「ミニバッファとフレーム」p.445 参照)。

ミニバッファ・ウィンドウは通常 1 行です。一時的にウィンドウ・サイズ変更コマンドで大きさを変えることができます。この場合、ミニバッファを抜けると普通の大きさに戻ります。ミニバッファがアクティブでないときは、ウィンドウ・サイズ変更コマンドをフレームのほかのウィンドウに使うことで、恒久的にその大きさを変えることもできます。フレームがミニバッファだけを含む場合、フレームの大きさを変更することでミニバッファの大きさを変更できます。

コマンドが、アクティブなミニバッファがあるのに別のミニバッファを使うと、これを『再帰的ミニバッファ (recursive minibuffer)』と呼びます。最初のミニバッファは'` *Minibuf-0*`'と名づけられます。再帰的ミニバッファはこの名前の終りにある数を増やすことで名づけられます (名前が一つの空白ではじまることで、通常のバッファ・リストに現れないようになります)。複数の再帰的ミニバッファのうち、最も内側の (即ち最も最近に入った) ミニバッファがアクティブなミニバッファとなります。普通ミニバッファといえば、これを指します。再帰的ミニバッファは、変数 `enable-recursive-minibuffers` を設定する、あるいはコマンド・シンボルにこの名前の属性を置くことで、許可あるいは禁止することができます (17.8 節「ミニバッファそのほか」p.254 参照)。

ほかのバッファ同様、ミニバッファはいくつかあるローカル・キーマップのいずれかをたいていは使います (第 19 章「キーマップ」p.293 参照)。これらはさまざまなミニバッファを抜けるコマンドや、場合によっては補完コマンドを含みます (17.5 節「補完」p.242 参照)。

- `minibuffer-local-map` は、通常入力 (補完なし) に使います。
- `minibuffer-local-ns-map` は同様ですが、 `SPC` で丁度 `RET` のように抜けてしまう点が異なっています。主に Mocklisp 互換のために使われます。
- `minibuffer-local-completion-map` は、任意補完に使います。
- `minibuffer-local-must-match-map` は、厳密補完と注意補完に使います。

17.2 ミニバッファからテキスト文字列を読みとる

ほとんどの場合、ミニバッファはテキストを文字列として読むために使います。 Lisp オブジェクトをテキスト形式で読み込むためにも使います。ミニバッファ入力の最も基本的なプリミティブは、 `read-from-minibuffer` です。これでそのいずれかを行なうことができます。

ほとんどの場合、 Lisp 関数の途中でミニバッファ入力関数を呼ぶべきではありません。代わりに、全てのミニバッファ入力をコマンド引数の読込みの一部とし、 `interactive` 指定の中で済ませます。 18.2 節「コマンドの定義」 p.258 を参照してください。

read-from-minibuffer *prompt-string* &optional *initial-contents* *keymap read hist* Function

この関数は、ミニバッファから入力を取る、最も一般的な方法です。デフォルトでは、任意のテキストを受けつけそれを文字列として返します。しかし *read* が非 `nil` ならば、 `read` を使ってそのテキストを Lisp オブジェクトに変換します (16.3 節「入力関数」 p.230 参照)。

この関数が最初にやるのは、ミニバッファをアクティブにし、 *prompt-string* をプロンプトとして表示することです。この変数は、文字列でなければなりません。

次に、 *initial-contents* が文字列ならば、 `read-from-minibuffer` がそれをミニバッファに入れ、ポイントを末尾に置きます。ミニバッファはこのテキストをその内容として現れます。

initial-contents の値は、 (*string* . *position*) の形式のコンス・セルであっても構いません。これは *string* をミニバッファに入れますが、末尾ではなく、先頭から *position* 文字目にポイントを置きます。

keymap が非 `nil` ならば、そのキーマップはミニバッファで使うローカル・キーマップです。 *keymap* が省略されたあるいは `nil` ならば、 `minibuffer-local-map` の値をキーマップとして使います。キーマップを指定することは、ミニバッファをさまざまな用途 (たとえば補完) にカスタマイズするための最も重要な方法です。

引数 *hist* で、入力を保存するためとミニバッファで使う履歴コマンドのために、どの履歴リスト変数を使うかを指定します。デフォルトは `minibuffer-history` です。 17.4 節「ミニバッファ履歴」 p.241 を参照してください。

ユーザが、ミニバッファを抜けるためのコマンドをタイプすると、 `read-from-minibuffer` はミニバッファ内のテキストからその戻り値を作ります。普通はそのテキストを内容とする文字列を作るだけです。しかし、 *read* が非 `nil` ならば、 `read-from-minibuffer` はテキストを読み込み、結果の Lisp オブジェクトを評価せずに返します (読込みに関することがらは、 16.3 節「入力関数」 p.230 を参照してください)。

read-string *prompt* &optional *initial* Function

この関数は、ミニバッファから文字列を読んで、それを返します。引数 *prompt* と *initial* は、 `read-from-minibuffer` と同じように使います。使用するキーマップは `minibuffer-local-map` です。

これは `read-from-minibuffer` 関数への単純化したインタフェースです:

```
(read-string prompt initial)
≡
(read-from-minibuffer prompt initial nil nil nil)
```

minibuffer-local-map Variable

これがミニバッファから読み込むときのデフォルト・ローカル・キーマップです。デフォルトでは、次のバインドになります:

`LFD` `exit-minibuffer`

`RET` `exit-minibuffer`

`C-g` `abort-recursive-edit`

`M-n` `next-history-element`

`M-p` `previous-history-element`

`M-r` `next-matching-history-element`

`M-s` `previous-matching-history-element`

read-no-blanks-input *prompt* &optional *initial* Function

この関数は、ミニバッファから文字列を読みますが、空白文字が入力の一部にあってはいけません。その代わり、それら文字は入力を終らせます。引数 *prompt* と *initial* は、`read-from-minibuffer` と同じように使います。

これは、 `read-from-minibuffer` 関数に対する単純化したインタフェースであり、`minibuffer-local-ns-map` キーマップの値を、この関数の *keymap* 引数として渡します。キーマップ `minibuffer-local-ns-map` は `C-q` を再バインドしないので、 quote すれば、スペースを文字列に置くことが可能です。

```
(read-no-blanks-input prompt initial)
≡
(read-from-minibuffer prompt initial minibuffer-local-ns-map)
```

minibuffer-local-ns-map Variable

本組込み変数は、関数 `read-no-blanks-input` がミニバッファ・ローカル・キーマップとして使うキーマップです。デフォルトでは、 `minibuffer-local-map` のデフォルトに加えて、次のバインドになります:

`SPC` `exit-minibuffer`

`TAB` `exit-minibuffer`

`?` `self-insert-and-exit`

17.3 ミニバッファから Lisp オブジェクトを読み込みます

本節ではミニバッファから Lisp オブジェクトを読み込む関数を説明します。

read-minibuffer *prompt* &optional *initial* Function

この関数は、 Lisp オブジェクトをミニバッファで読み込み、それを評価せずに返します。引数 *prompt* と *initial* は、 `read-from-minibuffer` と同じように使います。

これは`read-from-minibuffer` 関数への単純化したインタフェースです:

```
(read-minibuffer prompt initial)
≡
(read-from-minibuffer prompt initial nil t)
```

文字列 `"(testing)"`を初期入力として与える例:

```
(read-minibuffer
 "Enter an expression: " (format "%s" '(testing)))

;; ミニバッファはこのように表示される:

---------- Buffer: Minibuffer ----------
Enter an expression: (testing)⋆
---------- Buffer: Minibuffer ----------
```

ユーザは `RET` をタイプして、直ちに初期入力をデフォルトとして使ってもいいです。あるいは、その入力を編集してもいいです。

eval-minibuffer *prompt* &optional *initial* Function

この関数は、ミニバッファから Lisp 式を読み込み、評価して、結果を返します。引数 *prompt* と *initial* は、 `read-from-minibuffer` と同じように使います。

この関数は、 `read-minibuffer` 呼出しの結果を単純に評価します:

```
(eval-minibuffer prompt initial)
≡
(eval (read-minibuffer prompt initial))
```

edit-and-eval-command *prompt form* Function

この関数は、 Lisp 式をミニバッファから読み込み、評価します。このコマンドと `eval-minibuffer` との違いは、ここでの初期 *form* が必須となり、テキスト文字列としてではなく、印字表現に変換される Lisp オブジェクトとして扱われることです。これは `prin1` で印字されるので、文字列ならば、ダブル・クォート文字 (‘"’) が初期テキスト中に現れます。 16.5 節「出力関数」 p.233 を参照してください。

`edit-and-eval-command` が最初にやることは、 *prompt* をプロンプトとするミニバッファをアクティブにすることです。次に *form* の印字表現をミニバッファに挿入し、ユーザに編集させます。ユーザがミニバッファを抜けると、編集されたテキストを `read` で読み込み、評価します。結果の値が `edit-and-eval-command` の値となります。

次の例では、すでに形式として有効である式を初期テキストとしてユーザに提供しています:

```
(edit-and-eval-command "Please edit: " '(forward-word 1))

;; この式を評価すると、
;;    次のようにミニバッファに現れる:

---------- Buffer: Minibuffer ----------
Please edit: (forward-word 1)⋆
---------- Buffer: Minibuffer ----------
```

直ちに RET をタイプすると、ミニバッファを抜け式を評価するので、ポイントが 1 語前に進みます。本例では、 `edit-and-eval-command` は `nil` を返します。

17.4 ミニバッファ履歴

『ミニバッファ履歴リスト (minibuffer history list)』は以前のミニバッファ入力を記録し、ユーザがそれを便利に再利用できるようにします。履歴リストは実はシンボルであり、リストではありません。これは、その値が新しいものほど先に来る文字列 (以前の入力) のリストになっている変数です。

異なる種類の入力に使う、別個の履歴リストが多数あります。ミニバッファを使う度に正しい履歴リストを指定するのは、 Lisp プログラマの仕事です。

ミニバッファ入力の基本関数 `read-from-minibuffer` も `completing-read` も双方共オプション引数 *hist* を受けつけ、これによって履歴リストを指定します。以下は、指定可能な値です:

variable *variable*(シンボルである) を履歴リストとして使います。

(*variable* . *startpos*)
variable(シンボルである) を履歴リストとして使い、初期履歴位置を *startpos*(履歴の最新の要素を 0 として数えた整数) であると仮定します。

startpos を指定したならば、一貫性のために、それが指す履歴要素を初期ミニバッファ内容として指定することも必要になります。

hist を指定しなかったならば、デフォルト履歴リスト `minibuffer-history` が使用されます。そのほかの標準履歴リストについては以下を参照。自分個人の履歴リスト変数を作ることもできます。最初に使う前に `nil` に初期化するだけでいいです。

`read-from-minibuffer` と `completing-read` は、新しい要素を履歴リストに自動的に追加し、ユーザがリスト上の項目を再使用するためのコマンドを提供しています。自作のプログラムが履歴リストを使うには、使いたいときにそれを初期化し、その名前を入力関数に渡すだけです。ただし、ミニバッファ入力関数が使ってないときには、そのリストを手で修正しても平気です。

minibuffer-history Variable

ミニバッファ履歴入力のための、デフォルトの履歴リスト。

query-replace-history Variable

`query-replace` の引数 (と、ほかのコマンドの似た引数) のための履歴リスト。

file-name-history Variable

ファイル名引数のための、履歴リスト。

regexp-history Variable

正規表現引数のための、履歴リスト。

extended-command-history Variable

拡張コマンドの名前 (これ自体が引数) のための、履歴リスト。

shell-command-history Variable

シェル・コマンド (これ自体が引数) のための、履歴リスト。

read-expression-history Variable

評価すべき Lisp 式 (これ自体が引数) のための、履歴リスト。

17.5 補完

『補完 (completion)』は、省略形からはじめて名前の残りを埋める機能です。補完は、ユーザの入力を有効な名前のリストと比較し、名前のどこまでが一意に決められるかをユーザがタイプしたもので判別することによって、働きます。たとえば、 `C-x b`(`switch-to-buffer`) とタイプし、続いて切替えたいバッファ名の最初の何文字かをタイプし、続いて `TAB`(`minibuffer-complete`) をタイプすると、 Emacs はその名前を可能なかぎり長いところまで展開します。

標準的 Emacs コマンドはシンボル名、ファイル名、バッファ名、プロセス名の補完を提供しています。本章の関数を使えば、ほかの種類の名前の補完を実装することができます。

`try-completion` 関数は補完の基本的なプリミティブです。これは、与えられた文字列の集合に対して一致する、与えられた初期文字列の最長の補完を判別して返します。

関数 `completing-read` は補完のための、より高い水準のインタフェースを提供します。`completing-read` の呼出しで、有効な名前のリストの判別法を指定します。この関数は次に、補完に便利なコマンドをいくつかのキーにバインドしたローカル・キーマップをもつミニバッファをアクティブにします。ほかの関数は、ある種の名前を補完で読み込む便利で単純なインタフェースを提供します。

17.5.1 基本的補完関数

二つの関数 `try-completion` と `all-completion` は、それら自身はミニバッファとは何の関係ありません。これらを本章で説明したのは、これらをミニバッファを使う高水準補完機能の近くに置いておきたかったからです。

try-completion *string collection* &optional *predicate* Function

この関数は、 *collection* 中から、 *string* の可能な補完全ての最長共通部分文字列を返します。 *collection* の値は、 alist、 obarray、文字列の仮想集合を実現する関数 (以下参照) のいずれかを取る必要です。

補完では、 *string* を *collection* で指定された任意補完のそれぞれに対して比較します。任意補完の冒頭部分が *string* に等しければ、一致です。任意補完に一致するものがなければ、 `try-completion` は `nil` を返します。ただ一つの任意補完が一致した場合で、それが完全一致ならば、 `try-completion` は `t` を返します。さもなくば、その値は一致した全ての任意補完に共通な最長の始めの部分[1]になります。

[1] 訳注: initial sequence

collection が alist(5.8 節「連想リスト」p.81 参照) ならば、 alist 要素の CAR が任意補完集合を成します。

collection が obarray(7.3 節「シンボルの作成と intern」p.95 参照) ならば、 obarray 中の全てのシンボルの名前が任意補完集合を成します。グローバル変数 `obarray` は全ての intern された Lisp シンボルの名前を含む obarray を保持しています。

新たな obarray を作る唯一の有効な方法は方法は、それを空にしてからシンボルを一つずつ `intern` によって追加してゆくことであることに注意してください。与えられたシンボルを複数の obarray に intern することもできません。

引数 *predicate* が非 `nil` ならば、それは引数一つの関数でなければなりません。これは各一致候補を検査するために使われ、 *predicate* が非 `nil` を返したときにかぎりその候補が受け入れられます。 *predicate* に与えられる引数は、 alist からのコンス・セル (その CAR は文字列) であるか、そうでなければ、 obarray からのシンボル (シンボル名ではない) です。

collection として、関数であるシンボルを使うこともできます。この場合関数は補完の実行に対して一切の責任を持ちます。ここで `try-completion` は、この関数が何を返そうともそれを返します。この関数は、三つの引数を持って呼ばれます。それらは *string*、*predicate*、 `nil` です (3 番目の引数の理由は、同じ関数が `all-completions` で使えるようにし、なおかつ、どちらの場合でも適切に働くようにするためです)。 17.5.6 節「プログラム補完」p.250 を参照してください。

以下の例の 1 番目では、文字列 '`foo`' が alist の CAR の三つに一致しています。全ての一致は文字列 '`fooba`' ではじまっているので、それが結果となります。 2 番目の例では、一致候補はただ一つで、それが完全なので、値が `t` となります。

```
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4)))
     ⇒ "fooba"
(try-completion "foo" '(("barfoo" 2) ("foo" 3)))
     ⇒ t
```

以下の例では、おびただしいシンボルが文字列 '`forw`' で始まり、それらの全てが単語 '`forward`' で始まっています。シンボルのほとんどにおいて、これに '`-`' が続いていますが、全てではないので、 '`forward`' 以上補完することはできません。

```
(try-completion "forw" obarray)
     ⇒ "forward"
```

最後に以下の例では、三つの一致候補のうち、二つだけが述語 `test` をパスしました (文字列 '`foobaz`' は短か過ぎました)。これらのいずれも文字列 '`foobar`' で始まります。

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test
(try-completion
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ "foobar"
```

all-completions *string collection* &optional *predicate nospace* Function

この関数は、 *string* の可能な補完の全てをリストで返します。この関数へのパラメータは `try-completion` に対するものと同じです。

collection が関数ならば、それが三つの引数をもって呼び出されます。それらは *string*、*predicate*、 `t` です。そして `all-completions` は、この関数が何を返そうともそれを返します。 17.5.6 節「プログラム補完」 p.250 を参照してください。

変数 *nospace* が非 `nil` ならば、スペースで始まる補完は、 `string` も同様にスペースで始まっているのでないかぎり、無視されます。

`try-completion` の例で示した関数 `test` を使った例を挙げます:

```
(defun test (s)
  (> (length (car s)) 6))
     ⇒ test
(all-completions
 "foo"
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 'test)
     ⇒ ("foobar1" "foobar2")
```

completion-ignore-case Variable

この変数の値が非 `nil` ならば、 Emacs は補完において大文字 / 小文字の区別をしません。

17.5.2 補完とミニバッファ

本節では補完を使ってミニバッファから読み込む基本的なインタフェースを説明します。

completing-read *prompt collection* &optional *predicate require-match initial hist* Function

この関数は、ユーザを補完で助けながら、ミニバッファで文字列を読み取ります。この関数は、プロンプト *prompt* でミニバッファをアクティブにします。 *prompt* は文字列でなければならない。 *initial* が非 `nil` ならば、 `completing-read` はそれを入力の一部として、ミニバッファに挿入します。続いて、いくつかの補完を試みるコマンドを用意して、ユーザが入力を編集できるようにします。

実際の補完は *collection* と *predicate* を関数 `try-completion` に渡すことで行なわれます。これはローカル・キーマップにバインドされた補完のためのある種のコマンドにおいて実行されます。

require-match が `t` ならば、通常のミニバッファを抜けるコマンドでは、入力が *collection* の要素に補完されないかぎり抜けることはありません。 *require-match* が `nil` でも `t` でもないならば、タイプされた入力が *collection* の要素自身とならないかぎり、ミニバッファを抜けるコマンドで抜けることができません。 *require-match* が `nil` ならば、ミニバッファの入力に関わらず、ミニバッファを抜けるコマンドは使えます。

ユーザはミニバッファが空のまま `RET` をタイプして、何も入力しないで抜けることができます。この場合、 `completing-read` は `nil` を返します。これによってユーザは、コマ

ンドが今読み込んでいる値に対するデフォルトがどんなものであっても、それを指定することができます。ユーザは、 *require-match* の値に関わらず、このように RET を使って戻ることができます。

関数 completing-read は read-minibuffer を呼び出すことで働きます。 *require-match* が nil ならば minibuffer-local-completion-map をキーマップとして使いますし、 require-match が非 nil ならば minibuffer-local-must-match-map を使います。 17.5.3 節「補完を行なうミニバッファ・コマンド」 p.245 を参照してください。

引数 *hist* で、入力の保存とミニバッファ履歴コマンドにどの履歴リスト変数を使うのかを指定します。デフォルトは minibuffer-history です。 17.4 節「ミニバッファ履歴」 p.241 を参照してください。

補完は、組込み変数 completion-ignore-case が非 nil ならば、一致候補に対して入力を比較するときに、大文字 / 小文字を無視します。 17.5.1 節「基本的補完関数」 p.242 を参照してください。

```
(completing-read
 "Complete a foo: "
 '(("foobar1" 1) ("barfoo" 2) ("foobaz" 3) ("foobar2" 4))
 nil t "fo")

;; この式を評価すると、
;;    次のようにミニバッファに現れる:

---------- Buffer: Minibuffer ----------
Complete a foo: fo⋆
---------- Buffer: Minibuffer ----------
```

続いてユーザが DEL DEL b RET とタイプすると、 completing-read は barfoo を返します。

completion-read 関数は、実際に補完を行なうコマンドに情報を渡すために、三つの変数を束縛します。これら変数は、 minibuffer-completion-table、 minibuffer-completion-predicate、 minibuffer-completion-confirm です。これらについて詳しいことは、 17.5.3 節「補完を行なうミニバッファ・コマンド」 p.245 を見てください。

17.5.3 補完を行なうミニバッファ・コマンド

本節では、ミニバッファで補完を行なうために使うキーマップ、コマンド、ユーザ・オプションを説明します。

minibuffer-local-completion-map Variable

completing-read は補完の各候補との完全一致が要求されてないときには、この値をローカル・キーマップとして使います。デフォルトでは、このキーマップは次のバインドを成します:

?	minibuffer-completion-help
SPC	minibuffer-complete-word

`TAB`	`minibuffer-complete`

ほかの文字については、 `minibuffer-local-map` 同様にバインドされます。

minibuffer-local-must-match-map Variable

`completing-read` は補完の各候補との完全一致が要求されたときには、この値をローカル・キーマップとして使います。それゆえ、無条件にミニバッファを抜けるコマンド `exit-minibuffer` はどのキーにもバインドされていません。デフォルトではこのキーマップは次のバインドを成します:

`?`	`minibuffer-completion-help`
`SPC`	`minibuffer-complete-word`
`TAB`	`minibuffer-complete`
`LFD`	`minibuffer-complete-and-exit`
`RET`	`minibuffer-complete-and-exit`

ほかの文字については、 `minibuffer-local-map` 同様にバインドされます。

minibuffer-completion-table Variable

この変数の値は、ミニバッファでの補完に使われる alist あるいは obarray です。これはグローバル変数で、 `completing-read` が `try-completion` に渡すものを内容としています。 `minibuffer-complete-word` のようなミニバッファ補完コマンドで使われます。

minibuffer-completion-predicate Variable

この変数の値は `completing-read` が `try-completion` に渡す述語です。この変数は、ほかのミニバッファ補完関数でも使われます。

minibuffer-complete-word Command

この関数は、ミニバッファの内容を高々 1 単語まで、補完します。ミニバッファの内容が一つの補完しか持たなかったとしても、 `minibuffer-complete-word` は語構成可能ではない最初の文字を超えて文字を追加することはありません。第 31 章「構文テーブル」 p.541 を参照してください。

minibuffer-complete Command

この関数は、ミニバッファの内容を可能なかぎり最大に補完します。

minibuffer-complete-and-exit Command

この関数は、ミニバッファの内容を補完し、確認が要求されてなければ抜けます。即ち、 `minibuffer-completion-confirm` が `nil` ならば、そうします。確認が要求されているならば、 2 度続けて実行すると、 — 本コマンドを直ちに繰り返すことで、そうなりますが — このコマンドは確認抜きで働くよう作られています。

minibuffer-completion-confirm Variable

この変数の値が非 `nil` の場合、 Emacs はミニバッファを抜ける前に補完の確認を促します。関数 `minibuffer-complete-and-exit` はこの変数の値を抜ける前にチェックします。

minibuffer-completion-help Command

この関数は、現在のミニバッファの内容の可能な補完のリストを作ります。これは、変数 `minibuffer-completion-table` の値を *collection* 引数とし、 `minibuffer-completion-predicate` の値を *predicate* 引数として、 `all-completions` を呼び出すことで行ないます。補完のリストは、 '`*Completions*`' という名前のバッファにテキストで表示されます。

display-completion-list *completions* Function

この関数は、 *completions* を `standard-output` のストリーム、通常はバッファ、に表示します (ストリームについてのこれ以上のことがらは、第 16 章「Lisp オブジェクトの読込みと印字」 p.227 を参照してください)。引数 *completions* は通常は `all-completions` が返した補完のリストそのままですが、そうである必要ありません。シンボルあるいは文字列であるような各要素はいずれも単に表示しますが、二つの文字列からなるリストはその文字列を結合したかのように表示します。

この関数は、 `minibuffer-completion-help` から呼び出されます。最もよく使う方法は、 `with-output-to-temp-buffer` とともに使うやり方で、このようにします:

```
(with-output-to-temp-buffer "*Completions*"
  (display-completion-list
    (all-completions (buffer-string) my-alist)))
```

completion-auto-help User Option

この変数が非 `nil` ならば、補完コマンドが、次の文字が一意に決められないという理由で補完できるものがないという場合はいつでも、可能な補完のリストを自動的に表示します。

17.5.4 高水準補完関数

本節ではある種の名前を補完で読み込むための、高水準の便利な関数を説明します。

ほとんどの場合、 Lisp 関数の途中でこれら関数を呼ぶべきではありません。可能なときは、全てのミニバッファ入力をコマンド引数の読込みの一部とし、 `interactive` 指定の中で済ませます。 18.2 節「コマンドの定義」 p.258 を参照してください。

read-buffer *prompt* &optional *default existing* Function

この関数は、バッファ名を読み込み、それを文字列として返します。引数 *default* はデフォルト名として使い、ユーザがミニバッファに何も入れずに抜けたときにはその値が返ります。非 `nil` ならば、文字列あるいはバッファでなければなりません。これはプロンプトには表示されるが、初期入力としてミニバッファに挿入されることはありません。

existing が非 `nil` ならば、指定された名前は存在するバッファのものでなければなりません。ミニバッファを抜ける普通のコマンドでは、テキストが有効でないかぎり、抜けることはできません。さらに `RET` は有効な名前を見つけるように補完を行ないます (けれども、 *default* の有効性はチェックしません。ユーザがミニバッファに何も入れずに抜けたときは、値が何であっても、これを返します)。

次の例では、ユーザが 'minibuffer.t' を入力し、続いて RET をタイプする。引数 *existing* が t で、与えられた入力で始まるバッファ名は 'minibuffer.texi' だけなので、その名前が値となります。

```
(read-buffer "Buffer name? " "foo" t)
;; この式を評価すると、
;;    次のプロンプトが
;;    空のミニバッファとともに現れる:

---------- Buffer: Minibuffer ----------
Buffer name? (default foo) ⋆
---------- Buffer: Minibuffer ----------

;; ユーザが minibuffer.t RET とタイプする。
     ⇒ "minibuffer.texi"
```

read-command *prompt* Function

この関数は、コマンド名を読み込み、それを Lisp シンボルとして返します。引数 *prompt* は read-from-minibuffer と同様に使われます。コマンドは commandp が t を返すものならば何でもコマンドですが、コマンド名は commandp が t を返すシンボルであることを思い出してください。 18.3 節「対話的呼出し」p.262 を参照してください。

```
(read-command "Command name? ")

;; この式を評価すると、
;;    次のプロンプトが空のミニバッファとともに現れる:

---------- Buffer: Minibuffer ----------
Command name?
---------- Buffer: Minibuffer ----------
```

ユーザが forward-c RET とタイプすると、この関数は、 forward-char を返します。

read-command 関数は、関数 completing-read への単純化したインタフェースです。この関数は、現存する Lisp シンボルの集合から補完するために変数 obarray を使い、コマンド名だけを受けつけるために commandp 述語を使います:

```
(read-command prompt)
≡
(intern (completing-read prompt obarray
                         'commandp t nil))
```

read-variable *prompt* Function

この関数は、ユーザ変数の名前を読み込み、それをシンボルとして返します。

```
(read-variable "Variable name? ")

;; この式を評価すると、
;;    次のプロンプトが
;;    空のミニバッファとともに現れる:
```

```
---------- Buffer: Minibuffer ----------
Variable name? ⋆
---------- Buffer: Minibuffer ----------
```

ユーザが続いて fill-p RET とタイプすると、 read-variable は fill-prefix を返す。

この関数は、 read-command に似ていますが、 commandp の代わりに述語 user-variable-p を使います:

```
(read-variable prompt)
≡
(intern
 (completing-read prompt obarray
                  'user-variable-p t nil))
```

17.5.5 ファイル名の読み取り

ここでファイル名を読み取るために設計された、もう一つの高水準補完関数を挙げます。これにはデフォルト・ディレクトリの自動挿入という特殊機能も用意されています。

read-file-name *prompt* &optional *directory default existing initial* Function

この関数は、ミニバッファからファイル名を読み取ります。 *prompt* をプロンプトし、補完を用意しています。 *default* が非 nil ならば、ユーザが RET だけをタイプしたときには *default* を返します。 *default* の有効性のチェックは行ないません。ユーザがミニバッファに何も入れずに抜けたときは、値が何であっても、これを返します。

existing が非 nil ならば、ユーザは存在するファイルの名前を指定しなければなりません。 RET は可能なら有効な名前になるように補完を行ない、それが有効でないかぎりミニバッファを抜けません。 *existing* の値が nil でも t でもない場合、 RET は補完の後に確認も要求します。 existing が nil ならば、存在しないファイルの名前も受けつけます。

引数 *directory* は相対ファイル名の補完に使うディレクトリを指定します。 insert-default-directory が非 nil ならば、 *directory* はミニバッファにも初期入力として挿入します。デフォルトはカレント・バッファの default-directory の値です。

initial を指定すると、それが初期ファイル名としてミニバッファに (*directory* が挿入されるなら、その後ろに) 挿入されます。この場合、ポイントは *initial* の先頭に行きます。 *initial* のデフォルトは nil です — ファイル名は一切挿入しません。 *initial* が何をするのか見たい場合は、コマンド C-x C-v を試してください。

```
(read-file-name "The file is ")

;; この式を評価すると、
;;    次のようにミニバッファに現れる:

---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/⋆
---------- Buffer: Minibuffer ----------
```

manual TAB をタイプすると、次の結果になります:

```
---------- Buffer: Minibuffer ----------
The file is /gp/gnu/elisp/manual.texi⋆
---------- Buffer: Minibuffer ----------
```

ユーザが RET をタイプすると、 `read-file-name` はファイル名を文字列 `"/gp/gnu/elisp/manual.texi"`として返します。

insert-default-directory User Option

この変数は、 `read-file-name` で使われます。その値は、 `read-file-name` が (もしあれば初期ファイル名を加えた) デフォルト・ディレクトリ名を、ミニバッファに置いて処理を開始するか否かを制御します。この変数の値が `nil` ならば、 `read-file-name` はミニバッファに何の初期入力も置きません (初期入力を *initial* 引数で指定した場合を除く)。この場合、デフォルト・ディレクトリは相対ファイル名の補完に相変わらず使われるが、表示はされません。

```
;; この場合はミニバッファはデフォルト・ディレクトリから開始する。
(let ((insert-default-directory t))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ~lewis/manual/⋆
---------- Buffer: Minibuffer ----------

;; この場合はミニバッファは空で
;;    プロンプトのみがその行に現れる。
(let ((insert-default-directory nil))
  (read-file-name "The file is "))

---------- Buffer: Minibuffer ----------
The file is ⋆
---------- Buffer: Minibuffer ----------
```

17.5.6 プログラム補完

時には意図する可能な補完の全てを含む alist や obarray を作ることができない場合もあります。このような場合、与えられた文字列の補完を計算するような自分のための関数を用意することができます。これを『プログラム補完 (programmed completion)』と呼びます。

本機能を使うには、 `completing-read` の *collection* 引数として、関数定義をもつシンボルを渡せばいいです。関数 `completing-read` は自作の補完関数を `try-completion` や `all-completions` に知らせて、自作の関数が作業の全てを行なえるように調整します。

自作の補完関数は、三つの引数を受けつけなければなりません:

- 補完される文字列。
- 一致候補をフィルタする述語関数、なければ `nil`。自作の関数はこの述語を各一致候補に対して呼び出し、述語が `nil` を返したならその一致候補を無視しなければなりません。
- 操作のタイプを指定するフラグ。

フラグの値としては三つあり、三つの操作に対応します:

- `nil` は、 `try-completion` を指定します。自作補完関数は指定文字列の補完を返すか、文字列がすでに完全一致であったなら `t` を返すか、文字列に一致する候補がなければ `nil` を返すかでなければなりません。
- `t` は、 `all-completions` を指定します。自作補完関数は、指定文字列の可能な補完全てのリストを返さなければなりません。
- `lambda` は、完全一致の検査を指定します。自作補完関数は、指定文字列が完全一致の候補をもつならば `t` を返さなければなりません。さもなくば `nil` です。

補完関数群の *collection* に、 lamda 式 (関数であるリスト) を関数シンボル同様に使えるようにすれば、一貫性もありきれいにもなるでしょう。しかし、これは不可能です。補完テーブルとしてのリストがすでに違う意味、 alist として、割り当てられています。失敗しないように alist を正常に操作しようとすることは信頼できないと思われます。なぜならそれが関数の可能性があるからです。そこで、補完に使いたい自作関数は全てシンボルにカプセル化するように調整しなければなりません。

Emacs はプログラム補完をファイル名補完に使っています。 22.8.6 節「ファイル名補完」 p.372 を参照してください。

17.6 Yes か No かの質問

本節ではユーザに yes か no かの質問を訊くために使う関数を説明します。関数 `y-or-n-p` は 1 文字で答えることができます。これは不注意で誤って答えても重大な結果を起こさないような質問のときに便利です。 `yes-or-no-p` はもっと重大な質問に適当です。なぜなら、 3 文字ないし 4 文字の答えを要求するからです。

これら関数いずれも、マウスを使って起動したコマンドから呼ばれた場合は、 — より精確には、 `last-nonmenu-event`(18.4 節「コマンド・ループからの情報」 p.265 参照) が `nil` あるいはリストならば — ダイアログ・ボックスあるいはポップアップ・メニューを使って質問をします。それ以外の場合は、キーボード入力を使います。呼出し周辺で `last-nonmenu-event` を適当な値に束縛することで、マウスあるいはキーボード入力を使用することを強制できます。

厳密にいえば、 `yes-or-no-p` はミニバッファを使いますが、 `y-or-n-p` は使いません。しかし、どちらもここで説明するのが一番でしょう。

y-or-n-p *prompt* Function

この関数は、ユーザに質問をしてエコー領域からの入力を待ちます。ユーザが `y` をタイプすれば `t` を返し、 `n` をタイプすれば `nil` を返します。この関数は、 yes の意味の `SPC` と no の意味の `DEL` も受けつけます。 `C-g` のような「とりやめ」の意味の `C-]` も受けつけます。というのも、質問がミニバッファのように見えるかもしれないので、ユーザが `C-]` を使って抜けようとするだろうという理由によります。答えは 1 文字で、 `RET` で終了する必要はありません。大文字も小文字も区別しません。

「質問をする」というのは、 *prompt* に文字列‘`y or n`’を続けてエコー領域に表示することを意味します。入力が期待している答えの一つ (`y`、 `n`、 `SPC`、 `DEL`、あるいは何かとり

やめるもの) ではない場合、関数は ‘Please answer y or n.’ で応じて、要求を繰り返します。

この関数は、実際にはミニバッファを使いません。なぜなら答えの編集を許さないからです。実際エコー領域は使いますし (35.4 節「エコー領域」 p.603 参照)、それはミニバッファと同じ画面領域を使っています。質問がなされている間、カーソルはエコー領域に行っています。

答えとその意味は、 ‘y’ や ‘n’ であっても、変えられないものではありません。キーマップ query-replace-map がそれらを指定します。 30.5 節「検索と置換」 p.533 を参照してください。

以下の例では、ユーザははじめに無効である q をタイプします。次のプロンプトで、ユーザは y をタイプします。

```
(y-or-n-p "Do you need a lift? ")

;; この式を評価すると、
;;    次のプロンプトがエコー領域に現れる:

---------- Echo area ----------
Do you need a lift? (y or n)
---------- Echo area ----------

;; ユーザが q をタイプすると、次のように現れる:

---------- Echo area ----------
Please answer y or n.  Do you need a lift? (y or n)
---------- Echo area ----------

;; ユーザが有効な答えをタイプすると、
;;    それは質問の後ろに表示される:

---------- Echo area ----------
Do you need a lift? (y or n) y
---------- Echo area ----------
```

連続していくつかのエコー領域のメッセージを示しましたが、画面に実際に表示されるのは 1 度に一つだけです。

yes-or-no-p *prompt* Function

この関数は、ユーザに質問をしてミニバッファからの入力を待ちます。ユーザが ‘yes’ を入力すれば t を返し、 ‘no’ を入力すれば nil を返します。ユーザは応答を終了するために、 RET をタイプする必要があります。大文字も小文字も区別しません。

yes-or-no-p は、 *prompt* に文字列‘yes or no’を続けてエコー領域に表示することからはじめます。ユーザは期待されている応答の一つをタイプする必要があります。さもなくば、関数は ‘Please answer yes or no.’ で応答し、 2 秒ほど待ってから、要求を繰り返します。

yes-or-no-p はユーザに y-or-n-p よりも多くの作業を要求するので、より重大な決定に適切です。

例を挙げます:

```
(yes-or-no-p "Do you really want to remove everything? ")

;; この式を評価すると、
;;    次のプロンプトが
;;    空のミニバッファとともに現れる:

---------- Buffer: minibuffer ----------
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

ユーザがはじめに `y RET` をタイプした場合、 — この関数は、 ‘`yes`’ という単語全体を要求しているので無効です — 関数はこれらプロンプトをちょっと間をおいて表示することで応じます:

```
---------- Buffer: minibuffer ----------
Please answer yes or no.
Do you really want to remove everything? (yes or no)
---------- Buffer: minibuffer ----------
```

17.7 複数のYかNかの質問をする

一連の似通った質問、たとえば「このバッファを保存しますか」と各バッファについて順番に訊く場合、各質問をバラバラに行なう代わりに、まとまった質問を尋ねるための `map-y-or-n-p` を使うべきです。これによって、ユーザはある種の便利な機能、たとえば一連の問い全てに 1 度に答えるという機能を使うことができます。

map-y-or-n-p *prompter actor list* &optional *help action-alist* Function

この関数は、 Emacs 19 の新機能で、ユーザに一連の質問をし、各質問に対して 1 文字の答えをエコー領域から読み取ります。

list の値で指定するオブジェクト群についての質問が行なわれます。この値はオブジェクトのリストあるいは生成関数のいずれかでなければなりません。関数ならば、引数を期待してはなりません。そして、質問される次のオブジェクトか、質問の終りを意味する `nil` のいずれかを返さなければなりません。

引数 *prompter* はどのように各質問を訊くかを指定します。 *prompter* が文字列ならば、質問テキストはこのように計算されます:

(`format` *prompter object*)

object は、次に質問される (*list* から得た) オブジェクトです。

文字列ではないならば、 *prompter* は引数一つ (質問される次のオブジェクト) の関数でなければならず、質問テキストを返さなければなりません。値が文字列ならば、それがユーザに訊く質問です。この関数は、このオブジェクトに作用する (そしてユーザに訊かない) ことを意味します `t`、あるいはこのオブジェクトは無視する (そしてユーザに訊かない) ことを意味する `nil` を返すこともできます。

引数 *actor* はユーザの答えに対してどのように作用するかを指定します。これは引数一つの関数でなければならず、ユーザが yes と答えた各オブジェクトについて呼び出されます。この引数は必ず *list* から得たものになっています。

引数 *help* を与えるなら、それはこの形式のリストでなければなりません:

(*singular plural action*)

ここで *singular* は概念的に作用を受けるオブジェクトの名詞単数形で、 *plural* はそれに対応する名詞複数形です。 `action` は *actor* がどんな作用をするのかを説明する他動詞です。

help を指定しなかった場合、デフォルトは `"object"` `"objects"` `"act on"`です。

質問がなされる度に、ユーザが `y`, `Y`, `SPC` のいずれかを入力したならば、そのオブジェクトに対して作用します。 `n`, `N`, `DEL` ならば、そのオブジェクトは飛ばします。 `!` は以降全てのオブジェクトに対して作用します。 `ESC` あるいは `q` はミニバッファを抜けます (以降全てを飛ばす)。 `.`(ピリオド) は現オブジェクトに作用して、その後抜けます。 `C-h` はヘルプを出します。これらは、 `query-replace` が受けつけるのと同じ答えです。キーマップ `query-replace-map` がこれらの意味を `query-replace` に対するのと同様に `map-y-or-n-p` に対しても定義しています。 30.5 節「検索と置換」 p.533 を参照してください。

action-alist を使って、可能な答えとそれらが何を意味するかを追加指定することができます。これは (*char function help*) という形式が要素である alist で、各要素が追加の答えを一つ定義します。この要素においては、 *char* が文字 (追加の答え) で、 *function* が引数 (*list* からのオブジェクト) 一つの関数、 *help* が文字列です。

ユーザが *char* で応じると、 `map-y-or-n-p` は *function* を呼び出します。それが非 `nil` を返したならば、オブジェクトは「作用された」と判断します。そして、 `map-y-or-n-p` は *list* の次のオブジェクトに進みます。 `nil` を返したならば、同一のオブジェクトに対してプロンプトが繰り返されます。

`map-y-or-n-p` が、マウスを使って起動したコマンドから呼ばれた場合は、 — より精確には、 `last-nonmenu-event`(18.4 節「コマンド・ループからの情報」 p.265 参照) が `nil` あるいはリストならば — ダイアログ・ボックスあるいはポップアップ・メニューを使って質問をします。この場合、キーボード入力やエコー領域は使いません。呼出し周辺で `last-nonmenu-event` を適当な値に束縛することで、マウスあるいはキーボード入力を使用することを強制できます。

`map-y-or-n-p` の戻り値は、作用を受けたオブジェクトの個数です。

17.8 ミニバッファそのほか

本節では、ミニバッファに関連する基本的な関数と変数とを説明します。

exit-minibuffer Command

本コマンドは、アクティブなミニバッファを抜けます。普通はミニバッファ・ローカル・キーマップのキーにバインドされています。

self-insert-and-exit Command

本コマンドは、キーボードでタイプされた最後の文字 (`last-command-char` で見つかります。 18.4 節「コマンド・ループからの情報」 p.265 参照) を挿入した後に、アクティブなミニバッファを抜けます。

previous-history-element *n* Command

本コマンドは、ミニバッファの中身を *n* 番目の以前の (より古い) 履歴要素で置き換えます。

next-history-element *n* Command

本コマンドは、ミニバッファの中身を *n* 番目の以後の (より新しい) 履歴要素で置き換えます。

previous-matching-history-element *pattern* Command

本コマンドは、ミニバッファの中身を以前の (より古い) 履歴要素のうち、 *pattern*(正規表現) に一致するもので置き換えます。

next-matching-history-element *pattern* Command

本コマンドは、ミニバッファの中身を以後の (より新しい) 履歴要素のうち、 *pattern*(正規表現) に一致するもので置き換えます。

minibuffer-prompt Function

この関数は、現在アクティブなミニバッファのプロンプト文字列を返します。アクティブなミニバッファがない場合、 `nil` を返します。

minibuffer-prompt-width Function

この関数は、現在アクティブなミニバッファのプロンプト文字列の表示幅を返します。アクティブなミニバッファがない場合、 0 を返します。

minibuffer-setup-hook Variable

これは正規フックで、ミニバッファに入るときはいつでも実行されます。 20.4 節「フック」p.335 を参照してください。

minibuffer-exit-hook Variable

これは正規フックで、ミニバッファから抜けるときはいつでも実行されます。 20.4 節「フック」p.335 を参照してください。

minibuffer-help-form Variable

この変数の現在の値は、 `help-form` をミニバッファ内ローカルに再束縛するために使います (21.5 節「ヘルプ関数」p.344 参照)。

active-minibuffer-window Function

この関数は、現在アクティブなミニバッファ・ウィンドウあるいは、現在アクティブなものがなければ `nil` を返します。

minibuffer-window &optional *frame* Function

この関数は、フレーム *frame* が使うミニバッファ・ウィンドウを返します。 *frame* が `nil` ならば、現フレームを意味します。フレームが使うミニバッファ・ウィンドウは、そのフレームの一部である必要がないことに注意してください — 自身のミニバッファを持っていないフレームは、必然的にほかのフレームのミニバッファ・ウィンドウを使うことになります。

window-minibuffer-p *window* Function

この関数は、 *window* がミニバッファ・ウィンドウならば非 `nil` を返します。

与えられたウィンドウがミニバッファか否かを判別するために、それを `(minibuffer-window)` の結果との比較によるのは、正しくありません。なぜなら、フレームが沢山あれば、ミニバッファ・ウィンドウも複数存在できるからです。

minibuffer-window-active-p *window* Function

この関数は、 *window*(ミニバッファ・ウィンドウと仮定) が現在アクティブならば、非 `nil` を返します。

minibuffer-scroll-window Variable

この変数の値が非 `nil` ならば、ウィンドウ・オブジェクトでなければなりません。関数 `scroll-other-window` がミニバッファから呼ばれると、このウィンドウをスクロールします。

最後に、再帰的ミニバッファを扱ういくつかの関数と変数を挙げます (18.10 節「再帰編集」 p.288 参照):

minibuffer-depth Function

この関数は、現在のアクティブなミニバッファの深さを返します (非負整数)。アクティブなミニバッファがない場合、 0 を返します。

enable-recursive-minibuffers User Option

この変数が非 `nil` ならば、ミニバッファ・ウィンドウにいるときであってもミニバッファを使うようなコマンド (たとえば `find-file`) を起動できます。こういった起動が、新しいミニバッファに対する再帰編集レベルを引き起こします。外側レベルのミニバッファは内側を編集している間不可視となります。

この変数は、ミニバッファ・ウィンドウを選択した状態でミニバッファを起動するときだけ影響します。ミニバッファにいるときにウィンドウを切替えたならば、そのほかのウィンドウを選択している間はいつでもミニバッファ・コマンドを起動できます。

コマンド名が非 `nil` の属性 `enable-recursive-minibuffers` をもつならば、そのコマンドがミニバッファから起動された場合であっても、それはミニバッファを使って引数を読むことができます。ミニバッファ・コマンド `next-matching-history-element` (通常ミニバッファでは `M-s`) はこの機能を使っています。

18 コマンド・ループ

Emacs を実行すると、ほぼ直ちに『エディタ・コマンド・ループ (editor command loop)』に入ります。このループはキー・シーケンスを読み取り、それらの定義を実行し、結果を表示します。本章では、これがどのように行なわれるかと、 Lisp プログラムでそれを行なうことができるサブルーチンを説明します。

18.1 コマンド・ループ概観

コマンド・ループが最初にやらなければならないことは、キー・シーケンスを読み込むことです。キー・シーケンスはイベントのシーケンスであり、コマンドに翻訳されます。コマンド・ループは、これを行なうのに関数 `read-key-sequence` を呼びます。自作の Lisp コードでもこの関数を呼ぶことができます (18.6.1 節「キー・シーケンス入力」 p.279 参照)。 Lisp プログラムでは `read-event` (18.6.2 節「イベントを一つ読み込む」 p.281 参照) による低水準な入力や、 `discard-input`(18.6.4 節「そのほかのイベント入力機能」 p.282 参照) によって未処理の入力を捨てることもできます。

キー・シーケンスは現在アクティブなキーマップを通してコマンドに変換されます。これがどうなされるかを知るためには、 19.7 節「キー検索」 p.301 を参照してください。結果は、キーボード・マクロあるいは対話的に呼出し可能な関数です。キーが `M-x` ならば、これは別のコマンドの名前を読み込み、その後それを呼びます。これは、コマンド `execute-extended-command` (18.3 節「対話的呼出し」 p.262 参照) が行ないます。

コマンドを実行するには、まずその引数を読み込む必要があります。これには `command-execute`(18.3 節「対話的呼出し」 p.262 参照) を呼び出します。 Lisp で書かれたコマンドでは、 `interactive` 指定にどのように引数を読むのかが書かれています。これによって、前置引数 (18.9 節「前置コマンド引数」 p.286 参照) を使ったり、ミニバッファ (第 17 章「ミニバッファ」 p.237 参照) にプロンプトを出して読み取ったりします。たとえば、コマンド `find-file` はミニバッファを使ってファイル名を読むよう書かれた `interactive` 指定を持っています。コマンドの関数本体はミニバッファを使いません。つまり、このコマンドを Lisp コードから関数として呼び出した場合、ファイル名文字列を普通の Lisp 関数引数として与えなければなりません。

コマンドが文字列あるいはベクタ (即ち、キーボード・マクロ) ならば、 `execute-kbd-macro` を使ってそれを実行します。この関数を自分自身で呼んでも構いません (18.13 節「キーボード・マクロ」 p.291 参照)。

実行中のコマンドを終らせるには、 `C-g` をタイプします。この文字によって『とりやめる (quitting)』 (18.8 節「とりやめ」 p.284 参照) ことができます。

pre-command-hook Variable

エディタ・コマンド・ループは、この正規フックを各コマンドの前に実行します。この時点で、 `this-command` の内容はこれから実行しようとしているコマンドになっています。そして、 `last-command` は直前のコマンドを表しています。 20.4 節「フック」 p.335 を参照してください。

post-command-hook Variable

エディタ・コマンド・ループは、この正規フックを各コマンドの後 (とりやめあるいはエラーによって、途中で終ったコマンドも含む) に実行します。それから、最初にコマンド・ループに入ったときにも実行します。その時点では、 `this-command` はたった今実行したコマンドを表し、 `last-command` はその前のものを表します。 20.4 節「フック」 p.335 を参照してください。

`pre-command-hook` と `post-command-hook` との実行中、とりやめは抑止されます。これらフック中でエラーが起こった場合には、フックの実行だけを終らせ、それ以外は何もしません。

18.2 コマンドの定義

Lisp 関数は、その本体のトップレベルに特殊形式 `interactive` の呼出しを行なう形式があるならば、コマンドになります。この形式は、実際の実行のときには何もしませんが、その存在は対話的呼出しが許されているという意味のフラグとして働きます。その引数は、対話的呼出しの際に引数をどのように読み込むかを制御します。

18.2.1 `interactive` を使う

本節では、 Lisp 関数を対話的呼出し可能なコマンドにする `interactive` 形式を、どのように書くのかを説明します。

interactive *arg-descriptor* Special Form

この特殊形式が関数中にあると、それがコマンドであり、それゆえ対話的に (`M-x` やこのコマンドにバインドされたキー・シーケンスの入力によって) 呼び出され得ることを宣言します。引数 *arg-descriptor* は、コマンドが対話的に呼び出されたときにコマンドへの引数をどのように計算するかを宣言します。

コマンドはほかの関数同様 Lisp プログラムから呼び出されるかもしれませんが、その場合は呼び手が引数を用意し、 *arg-descriptor* は効果を持ちません。

`interactive` 形式が効果をもつのは、関数を呼び出す前に、コマンド・ループ (実際にはそのサブルーチン `call-interactively`) がその関数定義をスキャンして `interactive` を探すからです。一旦関数が呼び出されると、その本体の `interactive` 形式を含む全ての形式が実行されますが、この際は `interactive` は引数の評価さえ行なわずに単に `nil` を返します。

引数 *arg-descriptor* には三つの可能性があります:

- 省略あるいは `nil`。この場合、コマンドは引数なしで呼ばれます。この結果、コマンドが一つ以上の引数を要求している場合は即座にエラーとなります。
- 文字列以外の Lisp 式。この場合、評価の結果がコマンドに引き渡す引数のリストになるような形式であることが必要です。

  この式がキーボード入力 (ミニバッファの使用を含む) を読み込むならば、入力読込み前のポイントあるいはマークを表す整数値は入力読込み後には誤っているかもしれないというこ

とを忘れてはいけません。理由はカレント・バッファがサブプロセスからの出力を受け取っているかもしれないからです。コマンドが入力を待つ間にサブプロセス出力が来ると、それがポイントとマークを再配置するかもしれません。

やってはいけないことの例を挙げます:

```
(interactive
 (list (region-beginning) (region-end)
       (read-string "Foo: " nil 'my-history)))
```

先の問題を回避するには、キーボード入力を読み込んだ後にのみポイントとマークを調べればよいのです:

```
(interactive
 (let ((string (read-string "Foo: " nil 'my-history)))
   (list (region-beginning) (region-end) string)))
```

- 文字列。この場合、その中身はコード文字にプロンプトを続けたもので構成する必要があります (プロンプトはコード文字によって、使うことも無視することもあります)。文字列自体の終りか改行までが、プロンプトの終りです。以下に簡単な例を示します:

  ```
  (interactive "bFrobnicate buffer: ")
  ```

  コード文字 'b' で、補完を使って既存のバッファ名を読み込むことを表します。バッファ名がコマンドに渡される唯一の引数で、文字列の残りはプロンプトです。

  文字列中に改行文字があった場合、それがプロンプトの終りです。文字列がそこで終ってなかった場合、文字列の残りは別の引数を指定する別のコード文字とプロンプトを含んでなければなりません。このようにして、任意個数の引数を指定できます。

  プロンプト文字列に '%' を使って、直前の引数の値 (第 1 引数以降) をプロンプト中に含めることができます。これは format (4.6 節「文字列の書式変換」p.61 参照) によって行なわれます。たとえば、既存のバッファ名に続いて、そのバッファにつける新しい名前を読み込むには、こうします:

  ```
  (interactive "bBuffer to rename: \nsRename buffer %s to: ")
  ```

  文字列の最初の文字が '*' の場合、バッファが読みだし専用バッファならばエラーを通知します。

  文字列の最初の文字が '@' の場合、そしてそのコマンドを起動するのに使われたキー・シーケンスがマウス・イベントを含むならば、その最初のイベントに関連づけられたウィンドウがコマンド実行前に選択されます。

  '*' と '@' とを一緒に使うこともできます。その順序は関係ありません。実際の引数の読込みは、('*' や '@' 以外の文字から始まる) プロンプト文字列の残りの部分によって制御されます。

18.2.2 interactive 用のコード文字

以下のコード文字の記述は、ここに定義するキー・ワードを含んでいます:

補完
: 補完が使えます。TAB、SPC、RET は名前の補完を行ないます。引数が completing-read(17.5 節「補完」p.242 参照) で読み込まれるためです。? で可能な補完のリストを表示します。

既存　既存のオブジェクトの名前を要求します。無効な名前は受けつけません。つまり、ミニバッファを抜けるコマンドでも、現在の入力が有効にならないかぎり抜けません。

デフォルト　ユーザが何のテキストもミニバッファに入力しないならば、ある種のデフォルト値が使われます。デフォルトはコード文字次第です。

入出力なし　コード文字は何ら入力を読み込むことなしに、引数を計算します。そのために、プロンプト文字列を使いませんし、プロンプト文字列を与えても無視します。

コード文字がプロンプト文字列を使わなくとも、それが文字列中の最後のコード文字でないかぎり、これに続く改行が必要です。

プロンプト　プロンプトがコード文字直後に続きます。プロンプトは文字列の終りか改行で終ります。

特殊　このコード文字は interactive 文字列の先頭においてのみ意味を持ちます。これにはプロンプトや改行は不要です。これは単独の孤立した文字です。

`interactive` に使用するコード文字記述子を挙げます:

'`*`'　カレント・バッファが読みだし専用ならば、エラーを通知します。特殊。

'`@`'　このコマンドを起動したキー・シーケンス中の最初のマウス・イベントについてのウィンドウを選択します。特殊。

'a'　関数名 (即ち、 `fboundp` を満たすシンボル)。既存、補完、プロンプト。

'b'　既存のバッファ名。デフォルトには、カレント・バッファ (第 24 章「バッファ」 p.393 参照) の名前を使います。既存、補完、デフォルト。

'B'　バッファ名。バッファが存在する必要はありません。デフォルトには、カレント・バッファ以外の最近使われたバッファ名を使います。補完、デフォルト、プロンプト。

'c'　文字。カーソルはエコー領域に移動しません。プロンプト。

'C'　コマンド名 (即ち、 `commandp` を満たすシンボル)。既存、補完、プロンプト。

'd'　整数でのポイント位置 (27.1 節「ポイント」 p.457 参照)。入出力なし。

'D'　ディレクトリ名。デフォルトはカレント・バッファの現在のデフォルト・ディレクトリ、 `default-directory`(34.3 節「OS 環境」 p.584 参照)。既存、補完、デフォルト、プロンプト。

'e'　コマンドを起動したキー・シーケンス中の、最初あるいは次のマウス・イベント。さらに詳しくいうと、 '`e`' によりリストでイベントを取れるので、リスト中のデータを見ることができます。 18.5 節「入力イベント」 p.266 を参照してください。入出力なし。

'`e`' は一つのコマンドの interactive 指定で何度でも使えます。コマンドを起動したキー・シーケンスにリストが n イベント分あった場合、 n 番目の '`e`' は n 番目のそ

ういったイベントを与えます。リストではないイベント、ファンクション・キーや ASCII 文字などは、'e' に関するかぎり勘定に入れません。

'f' 既存ファイル名 (22.8 節「ファイル名」p.366 参照)。デフォルト・ディレクトリは default-directory。既存、補完、デフォルト、プロンプト。

'F' ファイル名。ファイルが存在する必要はありません。補完、デフォルト、プロンプト。

'k' キー・シーケンス (19.1 節「キーマップの用語」p.293 参照)。これによりイベントが現在のキーマップ中にコマンド (あるいは未定義コマンド) として見つかるまで、イベントを読み込み続けます。キー・シーケンス引数は文字列あるいはベクタで表現されます。カーソルはエコー領域に移動しません。プロンプト。

この種の入力は describe-key や global-set-key のようなコマンドで使います。

'K' 定義を変更しようと考えているキー・シーケンス。これは 'k' のように働きますが、キー・シーケンスの最後の入力イベントについて、普通 (必要ならば) 使用する未定義のキーを定義キーに変換するための変換を抑止する点が異なります。

'm' 整数でのマーク位置。入出力なし。

'n' ミニバッファから読み込む数。入力が数でなければ、ユーザは再度入力を乞われます。前置引数がもしあっても、使いません。プロンプト。

'N' そのままの前置引数。前置引数が nil ならば、 n のときのように数を読み込みます。数を要求します。 18.9 節「前置コマンド引数」p.286 を参照してください。プロンプト。

'p' 数値前置引数 (この 'p' が小文字であることに注意)。入出力なし。

'P' そのままの前置引数 (この 'p' が大文字であることに注意)。入出力なし。

'r' ポイントとマークを二つの数値引数として、小さい方が先。これだけが一つではなく、二つの連続した引数を指定するコード文字です。入出力なし。

's' 任意テキスト、ミニバッファで読み込み、文字列として返します (17.2 節「ミニバッファからテキスト文字列を読みとる」p.238 参照)。 LFD、 RET のいずれでも入力を打ち切ります (これらの文字を入力に含めるためには C-q を使います)。プロンプト。

'S' ミニバッファで intern されたシンボルの名前を読み込みます。いずれの空白文字でも入力を打ち切ります (空白文字を文字列に含めるには C-q を使います)。普通ならシンボルを打ち切るようなほかの文字 (たとえば括弧や角括弧) も、ここではそうなりません。プロンプト。

'v' ユーザ・オプションとして宣言された変数 (即ち、述語 user-variable-p を満たす)。 17.5.4 節「高水準補完関数」p.247 を参照してください。既存、補完、プロンプト。

'x' リード構文を指定した Lisp オブジェクト。LFD あるいは RET で打ち切ります。オブジェクトは評価されません。17.3 節「ミニバッファから Lisp オブジェクトを読み込みます」p.239 を参照してください。プロンプト。

'X' x のときのように読み込む Lisp 形式ですが、その後評価してその値がコマンドの引数になるようにしたもの。プロンプト。

18.2.3 interactive 使用例

interactive の例をいくつか挙げます:

```
(defun foo1 ()              ; foo1 は引数を取らない、
    (interactive)           ;   単に 2 語前に進むだけ。
    (forward-word 2))
     ⇒ foo1

(defun foo2 (n)             ; foo2 は引数を一つ取り、
    (interactive "p")       ;   それは数値前置引数。
    (forward-word (* 2 n)))
     ⇒ foo2

(defun foo3 (n)             ; foo3 は引数を一つ取り、
    (interactive "nCount:") ;   それはミニバッファから読み込まれる。
    (forward-word (* 2 n)))
     ⇒ foo3

(defun three-b (b1 b2 b3)
  "Select three existing buffers.
Put them into three windows, selecting the last one."
    (interactive "bBuffer1:\nbBuffer2:\nbBuffer3:")
    (delete-other-windows)
    (split-window (selected-window) 8)
    (switch-to-buffer b1)
    (other-window 1)
    (split-window (selected-window) 8)
    (switch-to-buffer b2)
    (other-window 1)
    (switch-to-buffer b3))
     ⇒ three-b
(three-b "*scratch*" "declarations.texi" "*mail*")
     ⇒ nil
```

18.3 対話的呼出し

コマンド・ループはキー・シーケンスを定義に変換した後、その定義を関数 command-execute を使って起動します。定義がコマンドとしての関数ならば、command-execute は call-interactively を呼び出し、それが引数を読み込んでコマンドを呼び出します。自分自身でこれら関数を呼んでも構いません。

commandp *object* Function

object が対話的呼出しに適しているならば、 t を返します。即ち、 *object* がコマンドであれば、そうします。さもなくば、 nil を返します。

対話的に呼出し可能なオブジェクトには文字列とベクタ (キーボード・マクロとして扱われます)、トップレベルの interactive 呼出しを含むラムダ式、そういったラムダ式から得られたバイトコード関数オブジェクト、対話的と宣言された (autoload への第 4 引数が非 nil) 自動ロード・オブジェクト、いくつかのプリミティブ関数を含みます。

シンボルはその関数定義が commandp ならば、 commandp です。

キーやキーマップはコマンドではありません。それどころか、コマンドを検索するために使うものです (第 19 章「キーマップ」 p.293 参照)。

commandp の実際の使用例については、 21.2 節「説明文字列へのアクセス」 p.340 の documentation を参照してください。

call-interactively *command* &optional *record-flag* Function

この関数は、対話的呼出し可能な関数 *command* を、その interactive 呼出し指定に基づいて引数を読み込んで、呼び出します。 *command* が関数や対話的に呼出し可能ではない場合 (即ちコマンドではない)、エラーを通知します。キーボード・マクロ (文字列とベクタ) は、たとえそれらがコマンドとみなされる場合でも、受けつけません。これらは関数ではないからです。

record-flag が非 nil ならば、本コマンドとその引数は、無条件にリスト command-history に追加されます。さもなくば、コマンドはそれが引数読込みのためにミニバッファを使った場合にかぎり、追加されます。 18.12 節「コマンド履歴」 p.291 を参照してください。

command-execute *command* &optional *record-flag* Function

この関数は、編集コマンド *command* を実行します。引数 *command* は commandp 述語を満足する必要があります。即ち、対話的に呼出し可能な関数かキーボード・マクロでなければなりません。

command の文字列あるいはベクタは execute-kbd-macro が実行します。関数は call-interactively にオプション *record-flag* とともに渡します。

シンボルはシンボルの場所内の関数定義を使うことで処理します。 autoload 定義によるシンボルは、それが対話的呼出し可能な関数として宣言されていたならば、コマンドとして扱います。こういった定義の処理は、指定のライブラリをロードしてから、そのシンボルの定義を再検査することで行ないます。

execute-extended-command *prefix-argument* Command

この関数は、コマンド名をミニバッファから completing-read (17.5 節「補完」 p.242 参照) を使って読み込みます。続いて指定のコマンドを呼び出すために、 command-execute を使います。そのコマンドが何を返そうとも、それが execute-extended-command の値になります。

コマンドが前置引数を求めると、 *prefix-argument* の値を受け取ることになります。`execute-extended-command` が対話的に呼び出されたならば、現在の前置引数がそのまま *prefix-argument* に使われます。そのため、これが実行されるどのようなコマンドにも渡ります。

`execute-extended-command` は `M-x` の普通の定義なので、文字列'`M-x`'をプロンプトとして使用します (`execute-extended-command` の起動に使われたイベントからプロンプトを取る方がよいのですが、これは実装するのが辛いのです)。前置引数があれば、その値の記述も同じくプロンプトの一部となります。

```
(execute-extended-command 1)
---------- Buffer: Minibuffer ----------
1 M-x forward-word RET
---------- Buffer: Minibuffer ----------
     ⇒ t
```

interactive-p Function

この関数は、これを含む関数 (`interactive-p` を呼んだ関数) が `call-interactively` で対話的に呼び出されたのであるならば、 `t` を返します (`call-interactively` が Lisp から呼ばれようと、エディタ・コマンド・ループから直接呼ばれようとも、何ら違いはありません)。これを含む関数が Lisp の評価 (あるいは `apply` や `funcall`) で呼ばれたならば、対話的に呼び出されたことになりません。

`interactive-p` の最も普通の使い方は、案内メッセージを表示するかしないか決めるために使うことです。特殊な例外として、 `interactive-p` はキーボード・マクロ実行中はいつでも `nil` を返すというのがあります。これは、案内メッセージを抑制して、マクロの実行を速くするためです。

例:

```
(defun foo ()
  (interactive)
  (and (interactive-p)
       (message "foo")))
     ⇒ foo

(defun bar ()
  (interactive)
  (setq foobar (list (foo) (interactive-p))))
     ⇒ bar

;; M-x foo とタイプする。
     ⊣ foo

;; M-x bar とタイプする。
;; こっちは何も表示しない。

foobar
     ⇒ (nil t)
```

18.4 コマンド・ループからの情報

エディタ・コマンド・ループは、いくつかの Lisp 変数を設定して自身のためと実行されるコマンドのために、状態の記録を保持しています。

last-command Variable

この変数は、コマンド・ループで実行された直前のコマンド (現在のコマンドの一つ前) の名前を記録しています。普通、値は関数定義をともなうシンボルですが、そうである保証はありません。

この値は `this-command` から、コマンドがコマンド・ループに帰るときにコピーされます。そのコマンドが後続のコマンドのために前置引数を指定している場合は、別です。

この変数は、常に現在の端末に対してローカルなのであって、バッファローカルにはなりません。 26.2 節「複数のディスプレイ」p.435 を参照してください。

this-command Variable

この変数は、エディタ・コマンド・ループで今実行しているコマンドの名前を記録しています。 `last-command` のように、普通は関数定義をともなうシンボルです。

コマンド・ループはこの変数をコマンド実行の直前に設定し、その値をコマンド実行終了時に `last-command` にコピーします (コマンドが後続のコマンドのために前置引数を指定する場合を除きます)。

いくつかのコマンドではこの変数をコマンド実行中に設定して、次に実行するコマンドへのフラグとします。特に、テキスト集合を kill する関数群は、 `this-command` を `kill-region` に設定して、直後に続く任意の kill コマンドが kill したテキストを直前の kill に追加すべきであることがわかるようにします。

エラー時に特定のコマンドを直前のコマンドとみなされたくない場合には、そのコマンドをこれを防ぐように書かなければなりません。方法の一つは、そのコマンドの冒頭で `this-command` を `t` に設定し、末尾で `this-command` を適切な値に戻すことです。このようにします:

```
(defun foo (args...)
  (interactive ...)
  (let ((old-this-command this-command))
    (setq this-command t)
    ...do the work...
    (setq this-command old-this-command)))
```

this-command-keys Function

この関数は、現在のコマンドを起動したキー・シーケンスと、そのコマンドに対する前置引数を生成した直前のコマンドを含む文字列あるいはベクタを返します。これらイベントが全て文字ならば、値は文字列となります。 18.5 節「入力イベント」p.266 を参照してください。

```
(this-command-keys)
;; ここで C-u C-x C-e を使って、評価する。
     ⇒ "^U^X^E"
```

last-nonmenu-event Variable

この変数は、キー・シーケンスの一部として読み込んだ、最後に入力されたイベントを保持します。このキー・シーケンスには、マウス・メニューからのイベントは含みません。

この変数の使い方の一つは、メニューを次にポップ・アップするのにいいデフォルト位置を見つけることです。

last-command-event Variable
last-command-char Variable

この変数には、コマンドの一部としてコマンド・ループによって読み込まれた、最後に入力されたイベントが設定されます。この変数の本質的使い方は、 `self-insert-command` 中にあります。そこでは、どの文字を挿入するかを決定するために使っています。

```
last-command-event
;; ここで C-u C-x C-e を使って、評価する。
     ⇒ 5
```

値は 5 となり、理由はそれが `C-e` の ASCII コードであるからです。

別名 `last-command-char` が Emacs バージョン 18 との互換のために存在します。

last-event-frame Variable

この変数は、最後に入力したイベントがどのフレームに向けられたものかを記録します。普通これはイベント発生時に選択されていたフレームですが、そのフレームが入力フォーカスを別フレームにリダイレクトしていた場合、この値はイベントのリダイレクト先のフレームになります。 26.9 節「インプット・フォーカス」 p.446 を参照してください。

18.5 入力イベント

Emacs コマンド・ループはキーボードあるいはマウスの動きを表す『入力イベント (input event)』のシーケンスを読み込みます。キーボードの動きのためのイベントは、文字あるいはシンボルです。しかし、マウス・イベントは常にリストです。本節では入力イベントの表現と意味を詳細に説明します。

eventp *object* Function

この関数は、 *object* が入力イベントならば非 `nil` を返します。

18.5.1 キーボード・イベント

キーボードから取ることができる入力には 2 種類あります。それらは、通常キーとファンクション・キーです。通常キーは文字に対応します。これらが生成するイベントを Lisp は文字で表現します。 Emacs バージョン 18 以前では、文字のみがイベントでした。文字イベントのイベント・タイプはその文字自身 (整数) です。イベント・タイプについては、 18.5.12 節「イベントの分類」 p.275 を参照してください。

入力文字イベントは 0 ~ 255 までの『基本コード (basic code)』にいくつかあるいは全ての『修飾ビット (modifier bit)』を加えたものから成ります:

メタ　文字コードの 2^27 ビットは、メタ・キー押下状態でタイプされた文字であることを表します。

制御　文字コードの 2^26 ビットは、非 ASCII 制御文字であることを表します。

C-a のような ASCII 制御文字はそれ自身が特別な基本コードをもつので、Emacs はそれらに特別なビットを与える必要がありません。それゆえ、 C-a のコードは単なる 1 です。

しかし ASCII にない制御の組合せをタイプした場合、たとえば % を制御キーとともにタイプしたような場合、得られる数値は % に 2^26 を加えたものです (端末が非 ASCII 制御文字をサポートしていると仮定)。

シフト　文字コードの 2^25 ビットは、シフト・キー押下状態でタイプされた ASCII 制御文字であることを表します。

英文字については、小文字に対する大文字の基本コードを表します。けれども数字と句読点については、シフト・キーは異なる基本コードの全く別な文字を選択します。可能なときには ASCII 文字集合の範囲内に収まるようにするために、Emacs はこれらの文字については 2^25 ビットを使わないようにします。

けれども、 ASCII は C-A を C-a と区別する手段を持たないので、 Emacs は 2^25 ビットを C-A に使用して、 C-a には使いません。

ハイパー　文字コードの 2^24 ビットは、ハイパー・キー押下状態でタイプされた文字であることを表します。

スーパー　文字コードの 2^23 ビットは、スーパー・キー押下状態でタイプされた文字であることを表します。

オルト　文字コードの 2^22 ビットは、オルト・キー押下状態でタイプされた文字であることを表します (端末によっては、 ALT と書かれたキーが実際のメタ・キーです)。

自作プログラム内では、特定のビット番号は書かないのが一番です。文字の修飾ビットを検査するには、関数 `event-modifiers` を使ってください (18.5.12 節「イベントの分類」 p.275 参照)。キー・バインドを作る場合、修飾ビットつき文字のリード構文 (‘\C-’, ‘\M-’, などなど) を使うことができます。 `define-key` を使ってキー・バインドを作る場合は、文字の指定に `(control hyper ?x)` といったリストを使うこともできます (19.9 節「キー・バインドの変更」 p.305 参照)。関数 `event-convert-list` はそういったリストをイベント・タイプ (18.5.12 節「イベントの分類」 p.275 参照) に変換します。

18.5.2 ファンクション・キー

ほとんどのキーボードは『ファンクション・キー (function key)』(文字ではなく名前や記号をもつキー) も備えています。ファンクション・キーは Lisp ではシンボルとして表現されます。そしてシンボルの名前はファンクション・キーのラベルを小文字にしたものです。たとえば、 F1 と書かれたキーを押下すると、シンボル `f1` が入力ストリームに置かれます。

ファンクション・キー・イベントのイベント・タイプは、イベント・シンボル自身です。 18.5.12 節「イベントの分類」 p.275 を参照してください。

以下に、ファンクション・キーについてのシンボル命名法の特例をいくつか挙げます:

`backspace`, `tab`, `newline`, `return`, `delete`
: これらのキーは、ほとんどのキーボードに専用のキーをもつ共通の ASCII 制御文字に対応します。

  ASCII では、 `C-i` と `TAB` は同じ文字です。端末がこれらを区別できるならば、Emacs はその違いを前者を整数の 9 として、後者をシンボル `tab` として、 Lisp プログラムに伝えます。

  ほとんどの場合、これら二つを区別することは有用ではありません。そのため普通は `function-key-map` が準備されていて、 `tab` を 9 に対応させます。そのため文字コード 9(文字 `C-i`) のキー・バインドが `tab` にも適用されます。このグループのほかのシンボルについても同様です。関数 `read-char` も同様にこれらイベントを文字に変換します。

  ASCII では、 `BS` は `C-h` そのものです。しかし `backspace` は文字コード 127(`DEL`) に変換されます。 8(`BS`) ではありません。ほとんどのユーザがこちらの方を好みます。

`left`, `up`, `right`, `down`
: カーソル矢印キー。

`kp-add`, `kp-decimal`, `kp-divide`, ...
: キーパッド・キー (通常のキーボードの右側にある)。

`kp-0`, `kp-1`, ...
: キーパッド・キーの数字。

`kp-f1`, `kp-f2`, `kp-f3`, `kp-f4`
: キーパッドの PF キー。

`kp-home`, `kp-left`, `kp-up`, `kp-right`, `kp-down`
: キーパッドの矢印キー。 Emacs は普通これらを非キーパッド・キー `home`, `left`, ... に変換します。

`kp-prior`, `kp-next`, `kp-end`, `kp-begin`, `kp-insert`, `kp-delete`
: 付加的なキーパッド・キーで、普通はどこかほかの場所にも見つかるものと重複します。 Emacs は普通これらを似た名前の非キーパッド・キーに変換します。

ファンクション・キーとともに、修飾キー `ALT`, `CTRL`, `HYPER`, `META`, `SHIFT`, `SUPER` を使うことができます。これらの表現方法は、シンボル名に前置をつけることです:

'`A-`' オルト修飾子。

'`C-`' 制御修飾子。

'`H-`' ハイパー修飾子。

‘M-’ メタ修飾子。

‘S-’ シフト修飾子。

‘s-’ スーパー修飾子。

よって、 META を押下した F3 キーのシンボルは、 M-f3 です。一つ以上の前置を使う場合、それらを英文字順に並べることを勧めます。しかしその順序は、キー・バインドの検索と修正の関数への引数に使うときは影響しません。

18.5.3 マウス・イベント

Emacs は 4 種類のマウス・イベントをサポートしています。それらはクリック・イベント、ドラッグ・イベント、ボタン押下イベント、移動イベントです。全てのマウス・イベントはリストで表現されます。リストの CAR はイベント・タイプです。これはどのマウス・ボタンが関係し、どの修飾キーがともに使われたかを示します。イベント・タイプによって 2 連続、 3 連続のボタン押下も区別できます (18.5.7 節「繰り返しイベント」 p.271 参照)。リストの残りの要素は、位置と時間の情報を与えます。

キー検索には、イベント・タイプのみが関わります。ここで二つのイベントが同じイベント・タイプならば同じコマンドを実行しなければなりません。コマンドはこれらイベントの完全な値に ‘e’ 対話コードを使ってアクセスできます。 18.2.2 節「interactive 用のコード文字」 p.259 を参照してください。

マウス・イベントで始まるキー・シーケンスは、マウスが入っていたウィンドウ中のバッファのキーマップを使って読み込まれます。カレント・バッファではありません。これは、ウィンドウ中のクリックがそのウィンドウ即ちそのバッファを選択するという意味ではありません。キー・シーケンスのコマンドのバインド制御についてのみの話です。

18.5.4 クリック・イベント

ユーザが同じ位置でマウス・ボタンを押して離したときに、『クリック (click)』・イベントが発生します。マウス・クリック・イベントは次の形式を持ちます:

(*event-type*
 (*window* *buffer-pos* (*x* . *y*) *timestamp*)
 click-count)

これら要素が普通何を意味するかを説明します:

event-type どのマウス・ボタンが使われたかを示すシンボルです。ボタンの左から右に向かって、 mouse-1,mouse-2, ... と番号づけられたシンボルのうちの一つになります。

前置 ‘A-’, ‘C-’, ‘H-’, ‘M-’, ‘S-’, ‘s-’ を修飾子オルト、制御、ハイパー、メタ、シフト、スーパーに対して、ファンクション・キーのときと同様に使うことができます。

このシンボルはイベントのイベント・タイプとしても働きます。キー・バインドはそのイベントをイベント・タイプで記述します。それゆえ、 mouse-1 に対する

キー・バインドがあれば、そのバインドは *event-type* が `mouse-1` である全てのイベントに対して適用されます。

window クリックを行なったときのウィンドウです。

x, *y* これらはクリックのピクセル座標で、 *window* の左上隅 `(0 . 0)` からの相対で表します。

buffer-pos クリックされた文字のバッファ位置です。

timestamp イベント発生の時刻で、ミリ秒単位です (この値は Emacs Lisp 整数の全範囲を、約 5 時間で一周するので、時間的に近いイベントに関してのみ有効です)。

click-count 同じマウス・ボタンを素早く続けて押した回数です。 18.5.7 節「繰り返しイベント」p.271 を参照してください。

buffer-pos, *x*, *y* の意味は、イベント位置がモード行やスクロール・バーのような画面の特別な部分内にあるときに若干異なってきます。

位置がスクロール・バー内にあるときは、 *buffer-pos* はシンボル `vertical-scroll-bar` か `horizontal-scroll-bar` になり、 `(`*x* `.` *y*`)` の対は *portion* `.` *whole* の対で置き換えられます。ここで *portion* は、スクロール・バーの上端あるいは左端からのクリックの距離であり、 *whole* はスクロール・バーの全長です。

位置がモード行あるいは *window* を右隣から分離する垂直線にあるときは、 *buffer-pos* がシンボル `mode-line` あるいは `vertical-line` になります。モード行の場合は、 *y* は意味のないデータです。垂直線の場合は、 *x* が意味のないデータとなります。

一つの特例として、 *buffer-pos* が単なるシンボルの代わりに、シンボル (上に挙げたものの一つ) を含むリストになっている場合があります。これは、イベントに対する架空の前置キーが入力ストリームに挿入された場合に起こります。 18.6.1 節「キー・シーケンス入力」p.279 を参照してください。

18.5.5 ドラッグ・イベント

Emacs では、着替えもせずにドラッグ・イベント[1]を行なえます。『ドラッグ・イベント (drag event)』はユーザがマウス・ボタンを押して、それを離す前に異なる文字位置へマウスを動かしたとき常に起こります。ほかの全てのマウス・イベント同様、ドラッグ・イベントは Lisp のリストで表現されます。このリストはマウスの移動の開始位置と最終位置との両方を記録しています。こんな感じです:

```
(event-type
 (window1 buffer-pos1 (x1 . y1) timestamp1)
 (window2 buffer-pos2 (x2 . y2) timestamp2)
 click-count)
```

ドラッグ・イベントは、シンボル名 *event-type* に前置 ‘`drag-`’ がついています。イベントの 2 番目 3 番目の要素がドラッグの開始と終了の位置を与えます。これを別にすれば、データはク

[1] 訳注: drag event で、女装パーティのことらしい

リック・イベント (18.5.4 節「クリック・イベント」 p.269 参照) と同じ意味合いを持ちます。マウス・イベントの 2 番目の要素は、皆同じ方法でアクセスできます。ドラッグ・イベントをほかと区別する必要はありません。

'drag-' 前置は、 'C-' や 'M-' のような修飾キー前置の後に来ます。

`read-key-sequence` がキー・バインドのないドラッグ・イベントを受け取り、対応するクリック・イベントの方がバインドをもつならば、 `read-key-sequence` はドラッグ・イベントをドラッグ開始位置でのクリック・イベントに変換します。つまり、そうしたくないかぎり、クリックとドラッグの両イベントを区別する必要はない、ということです。

18.5.6 ボタン押下イベント

クリックとドラッグのイベントは、ユーザがマウス・ボタンを離したときに起きます。これより早く起こせないのは、ボタンが離されるまでクリックをドラッグと区別できないからです。

ボタンが押されると同時に動作を起こしたいならば、『ボタン押下 (button-down)』イベント[2]を操作する必要があります。これはボタンが押されるやいなやに発生します。これはクリック・イベント (18.5.4 節「クリック・イベント」 p.269 参照) と全く見かけが同じリストで表現されます。違うのは、 *event-type* シンボル名が前置 'down-' を含むことです。 'down-' 前置は 'C-' や 'M-' のような修飾キー前置の後に来ます。

関数 `read-key-sequence` は、コマンドにバインドされていないボタン押下イベントは全て無視します。その結果 Emacs コマンド・ループもそうすることになります。つまりボタン押下イベントで何かしようと思わないかぎり、その定義を気にする必要がないという意味です。ボタン押下イベントを定義する理由としてあり得そうなのは、そうすることでボタンが離されるまでのマウスの動きを (移動イベントを読み込むことで) 追跡できるというのがあります。 18.5.8 節「移動イベント」 p.273 を参照してください。

18.5.7 繰り返しイベント

マウスを動かさずに素早く続けて同じマウス・ボタンを何度も押した場合、 Emacs では特別な『繰り返し (repeat)』マウス・イベントを 2 回目以降の押下に対して発生します。

もっともありふれた繰り返しイベントは、『ダブルクリック (double-click)』・イベントです。 Emacs ではボタンを 2 度クリックしたときにダブルクリックを発生します。このイベントはボタンを離したときに起きます (それがクリック・イベント全てで普通)。

ダブルクリック・イベントのイベント・タイプには前置 'double-' がついています。そのため、 `META` を押下した 2 番目のマウス・ボタンのダブルクリックは、 `M-double-mouse-2` として Lisp プログラムに渡ります。ダブルクリック・イベントにバインドがない場合は、対応する通常のクリック・イベントのバインドが使われて、実行されます。そのため、気にしたくなければダブルクリック機能に注意を払わなくて構いません。

---

[2] ボタンダウン (シャツ) は保守的で、女装とは対象的です。

訳注: drag event が女装パーティなのに対し、 button-down(ボタン押下)event はボタンダウン・シャツを着たパーティ、といいたいらしい。

ユーザがダブルクリックをすると、 Emacs は最初に通常のクリック・イベントを発生し、次にダブルクリック・イベントを発生します。それゆえ、ダブルクリック・イベントのコマンド・バインドは、シングルクリックのコマンドがすでに実行されていることを仮定して設計しないとなりません。設計は、ダブルクリックが望みどおりの結果になるように、シングルクリックの結果からやらないといけません。

ダブルクリックの意味がシングルクリックの意味の上にともかく「増築されている」ならば、都合いいです — これがダブルクリックに対するお薦めのユーザ・インタフェース設計の習慣です。

ボタンをクリックして、再びそれを押下し、ボタンを押下したままでマウスを動かしはじめると、最後にボタンを離したときに『ダブルドラッグ (double-drag)』・イベントを得ることになります。そのイベント・タイプには、単なる ‘`drag`’ の代わりに ‘`double-drag`’ がつきます。ダブルドラッグ・イベントがバインドを持たない場合、 Emacs はイベントが通常のドラッグだったかのように代替バインドを探します。

ダブルクリックやダブルドラッグ・イベントよりも前に、ユーザが 2 度目にボタンを押下したときに、 Emacs は『ダブル押下 (double-down)』イベントを発生します。そのイベント・タイプには、単なる ‘`down`’ の代わりに ‘`double-down`’ がつきます。ダブル押下イベントがバインドを持たない場合、 Emacs はイベントが通常のボタン押下イベントだったかのように代替バインドを探します。それでもバインドが見つからない場合は、ダブル押下イベントは無視されます。

要約すると、ボタンをクリックして、直ちに再びそれを押下すると、 Emacs は押下イベントとクリック・イベントを最初のクリックに対して、ダブル押下イベントを 2 回目の押下に対して、最後にダブルクリックあるいはダブルドラッグ・イベントのいずれかを生成します。

ボタンを 2 度クリックして、さらにもう 1 度押下する、これらを素早く続けて行うと、 Emacs は『トリプル押下 (triple-down)』イベントを、続けて『トリプルクリック (triple-click)』あるいは『トリプルドラッグ (triple-drag)』のいずれかを発生します。これらのイベントのイベント・タイプには ‘`double`’ の代わりに ‘`triple`’ がつきます。どのトリプル・イベントでも、バインドを持たない場合、 Emacs は対応するダブル・イベントに使われるであろうバインドを使います。

ボタンを 3 回以上クリックし、再び押下した場合、 3 回目以降の押下に対するイベントは全てトリプル・イベントとなります。 Emacs は 4 重、 5 重、などなどのイベントを区別するイベント・タイプを持ちません。それでも、イベント・リストを見て、何回ボタンが押下されたかを正確に知ることができます。

event-click-count *event* Function

この関数は、 *event* を導くまでに到った、連続したボタン押下の回数を返します。 *event* がダブル押下、ダブルクリック、ダブルドラッグならば、値は 2 です。 *event* がトリプル・イベントならば、値は 3 以上です。 *event* が (繰り返しイベントではない) 通常のマウス・イベントならば、値は 1 です。

double-click-time Variable

繰り返しイベントを発生させるには、同じ画面位置で連続してマウス・ボタンを押下し、連続するボタン押下の間隔がミリ秒で `double-click-time` の値よりも小さくなければな

りません。 `double-click-time` を `nil` に設定すると、マルチ・クリックの検出を完全に禁止します。 `t` に設定すると、時間の制限を取り除きます。つまり Emacs はマルチ・クリックを場所のみで検出するようになります。

18.5.8 移動イベント

Emacs は、マウス・ボタンの動きがなくてもマウス移動を表すために『マウス移動 (mouse motion)』イベントを時折発生します。

```
(mouse-movement
 (window buffer-pos (x . y) timestamp))
```

リスト 2 番目の要素は、丁度クリック・イベント (18.5.4 節「クリック・イベント」 p.269 参照) のときのように、マウスの現在の位置を表します。

特殊形式 `track-mouse` 本体内では、移動イベントの発生が可能になっています。 `track-mouse` 形式外では、 Emacs は単なるマウスの移動についてのイベントは発生しません。よって、これらイベントは現れません。

track-mouse *body...* Special Form

この特殊形式は *body* を実行し、そこではマウス移動イベントの発生が可能になっています。典型的な *body* は、 `read-event` を使って移動イベントを読み込み、それにしたがって表示を修正します。

ユーザがボタンを離すと、クリック・イベントが発生します。典型的には、 *body* はクリック・イベントを認識した時点で帰り、なおかつそのイベントは捨てるべきです。

18.5.9 フォーカス・イベント

ウィンドウ・システムはどのウィンドウがキーボード入力を受け取るかをユーザが制御する一般的方法を備えています。このウィンドウの選択を『フォーカス (focus)』と呼びます。ユーザが Emacs のフレームを切替えるために何かをすると、『フォーカス・イベント (focus event)』が発生します。フォーカス・イベントの通常の定義はグローバル・キーマップにあり、ユーザが期待するとおりに Emacs 内で新たなフレームを選択するようになっています。 26.9 節「インプット・フォーカス」 p.446 を参照してください。

フォーカス・イベントは Lisp ではこんな感じのリストで表現されます:

```
(switch-frame new-frame)
```

ここで *new-frame* は切替え先のフレームです。

ほとんどの X ウィンドウ・マネージャは、単にマウスをウィンドウの中に移動するだけでフォーカスをそこに置けるようになっています。 Emacs もこうなっているように見えます。新たなフレーム内で、カーソルがはっきりするからです。けれども、 Lisp プログラムにとってはほかの種類の入力が来ないかぎり、フォーカスが変わったことを知る必要がありません。そこで、 Emacs は実際にユーザがキーボード・キーをタイプするかマウス・ボタンを押下するかを新たなフレームで行なった場合にかぎり、フォーカス・イベントを発生します。つまりフレーム間でマウスを単に移動しても、フォーカス・イベントは発生しません。

キー・シーケンスの途中でフォーカス・イベントが発生すると、キー・シーケンスがグシャグシャになってしまうことでしょう。そのため Emacs ではキー・シーケンス途中でフォーカス・イベントが発生することは絶対にありません。ユーザがキー・シーケンスの途中 (つまり、前置キーの後) でフォーカスを変えたならば、 Emacs はそのイベントを並べ替えて、フォーカス・イベントが多重イベントのキー・シーケンスの前か後に来るように、途中には来ないようにします。

18.5.10 そのほかのウィンドウ・システム・イベント

ウィンドウ・システム内で発生したことを表す、そのほかのイベント・タイプ。

`(delete-frame (`*frame*`))`
: この種のイベントはユーザがウィンドウ・マネージャに特定のウィンドウを消去するように命令したこと、それが Emacs のフレームだったことを示します。

  `delete-frame` イベントの標準定義は、 *frame* の消去です。

`(iconify-frame (`*frame*`))`
: この種のイベントはユーザがウィンドウ・マネージャを使って *frame* をアイコン化したことを示します。この標準定義は `ignore` です。というのも、フレームはすでにアイコン化されていたので、 Emacs はすることがないのです。このイベント・タイプの目的は、必要ならこういったイベントを追跡できるようにすることです。

`(make-frame-visible (`*frame*`))`
: この種のイベントはユーザがウィンドウ・マネージャを使って *frame* をアイコン化解除したことを示します。この標準定義は `ignore` です。というのも、フレームはすでにアイコン化解除されていたので、 Emacs はすることがないのです。

これらイベントのいずれかがキー・シーケンスの途中にきた場合 (つまり、前置キーの後) は、Emacs はこれらイベントを並べ替えて、このイベントが多重イベントのキー・シーケンスの前か後に来るように、途中には来ないようにします。

18.5.11 イベントの例

ユーザが同じ場所で左マウス・ボタンを押下して離したならば、このようなイベント列が発生します:

```
(down-mouse-1 (#<window 18 on NEWS> 2613 (0 . 38) -864320))
(mouse-1      (#<window 18 on NEWS> 2613 (0 . 38) -864180))
```

制御キーを押下したまま、ユーザが 2 番目のマウス・ボタンを押下し、マウスをある行から次の行へドラッグしたとします。これによって、以下に示すような二つのイベントが発生します:

```
(C-down-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219))
(C-drag-mouse-2 (#<window 18 on NEWS> 3440 (0 . 27) -731219)
                (#<window 18 on NEWS> 3510 (0 . 28) -729648))
```

メタとシフト・キーを押下したまま、ユーザが 2 番目のマウス・ボタンをウィンドウのモード行で押下し、マウスを別のウィンドウへドラッグしたとします。これによって、このようなイベントの組が発生します:

```
(M-S-down-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844))
(M-S-drag-mouse-2 (#<window 18 on NEWS> mode-line (33 . 31) -457844)
                  (#<window 20 on carlton-sanskrit.tex> 161 (33 . 3)
                   -453816))
```

18.5.12 イベントの分類

全てイベントは『イベント・タイプ (event type)』を持ち、キー・バインドでの使用では、それによってイベントを分類しています。キーボード・イベントについては、イベント・タイプはイベント値に同じです。そのため文字のイベント・タイプは文字で、ファンクション・キー・シンボルのイベント・タイプはシンボル自身です。リストであるようなイベントについては、イベント・タイプはリストの CAR にあるシンボルです。よって、イベント・タイプは常にシンボルか文字となります。

同じ種の二つのイベントはキー・バインドに関するかぎり等価です。そのため、どちらも常に同じコマンドを実行します。これは必ずしもこれらが同一のことを行なうことを意味するわけではありません。というのも、コマンドによってはイベント全体を見て何をするのか決めるからです。たとえば、コマンドによってはマウス・イベントの位置を使って、どのバッファに作用するかを決めます。

ときにはイベントをさらに詳しく分類することが便利です。たとえば、イベントが `META` キーとともに使われたかどうかを確かめたい場合があるかもしれません (ほかのキーやマウス・ボタンの使用は無視する)。

関数 `event-modifiers` と `event-basic-type` が、こういった情報を都合よく得るために用意されています。

event-modifiers *event* Function

この関数は、 *event* がもつ修飾子のリストを返します。修飾子はシンボルです。これらは `shift`, `control`, `meta`, `alt`, `hyper`, `super` を含みます。加えて、マウス・イベント・シンボルの修飾子リストは、常に `click`, `drag`, `down` のいずれかを含みます。

引数 *event* は完全なイベント・オブジェクトであるか、単なるイベント・タイプかのいずれかです。

以下に例を挙げます:

```
(event-modifiers ?a)
     ⇒ nil
(event-modifiers ?\C-a)
     ⇒ (control)
(event-modifiers ?\C-%)
     ⇒ (control)
(event-modifiers ?\C-\S-a)
     ⇒ (control shift)
(event-modifiers 'f5)
```

```
     ⇒ nil
(event-modifiers 's-f5)
     ⇒ (super)
(event-modifiers 'M-S-f5)
     ⇒ (meta shift)
(event-modifiers 'mouse-1)
     ⇒ (click)
(event-modifiers 'down-mouse-1)
     ⇒ (down)
```

クリック・イベントについての修飾子のリストは、明示的に `click` を含みますが、イベント・シンボル名自身は ‘`clisk`’ を含みません。

event-basic-type *event* Function

この関数は、 *event* が表す全ての修飾子を取り除いたキーあるいはマウス・ボタンを返します。例を挙げます:

```
(event-basic-type ?a)
     ⇒ 97
(event-basic-type ?A)
     ⇒ 97
(event-basic-type ?\C-a)
     ⇒ 97
(event-basic-type ?\C-\S-a)
     ⇒ 97
(event-basic-type 'f5)
     ⇒ f5
(event-basic-type 's-f5)
     ⇒ f5
(event-basic-type 'M-S-f5)
     ⇒ f5
(event-basic-type 'down-mouse-1)
     ⇒ mouse-1
```

mouse-movement-p *object* Function

この関数は、 *object* がマウス移動イベントならば、非 `nil` を返します。

event-convert-list *list* Function

この関数は、修飾子名のリストと基本イベント・タイプをそれら全てを指定するイベント・タイプに変換します。例を挙げます:

```
(event-convert-list '(control ?a))
     ⇒ 1
(event-convert-list '(control meta ?a))
     ⇒ -134217727
(event-convert-list '(control super f1))
     ⇒ C-s-f1
```

18.5.13 イベントへのアクセス

本節ではマウス・ボタンや移動イベント中のデータをアクセスするのに都合がいい関数を説明します。

これら二つの関数はマウス・ボタン・イベントの開始あるいは終了位置を返します。位置はこの形式のリストです:

(*window* *buffer-position* (*x* . *y*) *timestamp*)

event-start *event* Function

これは *event* の開始位置を返します。

event がクリックあるいはボタン押下イベントならば、これがイベントの位置を返します。 *event* がドラッグ・イベントならば、これがドラッグの開始位置を返します。

event-end *event* Function

これは *event* の終了位置を返します。

event がドラッグ・イベントならば、これがユーザがボタンを離した位置を返します。*evnet* がクリックあるいはボタン押下イベントならば、その値は実際には開始位置で、こういったイベントが唯一もつ位置です。

これら四つの関数は上に挙げたような位置を取り、それのさまざまな部分を返します。

posn-window *position* Function

position が入っているウィンドウを返します。

posn-point *position* Function

position のバッファ位置を返します。これは整数です。

posn-x-y *position* Function

position 内のピクセル・ベースの x, y 座標を (*x* . *y*) というコンス・セルで返します。

posn-col-row *position* Function

position の (文字単位の) 行と桁を (*x* . *y*) というコンス・セルで返します。これらは *position* から実際に取り出した *x*, *y* の値より計算します。

posn-timestamp *position* Function

position のタイムスタンプを返します。

scroll-bar-event-ratio *event* Function

この関数は、スクロール・バー・イベントのスクロール・バー中の垂直位置を割合で返します。値は (*portion* . *whole*) というコンス・セルで、その比が位置の割合となるような二つの整数を含みます。

scroll-bar-scale *ratio total* Function

この関数は、(事実上)*ratio* に *total* を乗じて、結果を整数に丸めます。引数 *ratio* は数ではなく、(*num* . *denom*) の組です。典型的には `scroll-bar-event-ratio` が返した値です。

この関数は、スクロール・バー上の位置をバッファ位置に変換するのに便利です。こうやります:

```
(+ (point-min)
   (scroll-bar-scale
      (posn-x-y (event-start event))
      (- (point-max) (point-min))))
```

スクロール・バー・イベントが、 x, y 座標の組の代わりに、二つの比をなす整数をもつことに注意してください。

18.5.14 文字列にキーボード・イベントを入れる

文字列が使われるほとんどの部分において、私たちはその文字列をテキスト文字を含むものとして概念化しています (同じ種類の文字がバッファあるいはファイルにもあるのです)。時折 Lisp プログラムは文字列をキーボード文字を含むものとして扱います。たとえば、それらはキー・シーケンスやキーボード・マクロ定義であるかもしれません。そこで、キーボード文字を文字列中に置くための特別な規則が存在します。というのも、それらはテキスト文字のように 0 から 255 の範囲に限定されないからです。

META を使ってタイプされたキーボード文字を『メタ文字 (meta character)』と呼びます。このようなイベントに対する数値コードは 2^27 ビットを含みます。けれどもこれでは全然文字列に収まりません。しかし初期の Emacs の版においては、これら文字に対して異なる表現を使っていました。これらに 128 から 255 の範囲のコードを与えるのです。こうすれば文字列に収まり、多くの Lisp プログラムが、特に `define-key` や類似の関数の引数において、'`M-`' によって表現されたメタ文字を文字列定数に含めていました。

キーボード文字イベントを文字列に入れる特別な規則を使っているこれらプログラムを実行するために、後方互換性を提供しています。こういう規則です:

- キーボード文字の値が 0 から 127 の範囲にあるならば、そのまま文字列に入る。
- これら文字のメタ類のもの、コードが 2^27 から $2^27 + 127$, の範囲にあるものについては、同じく文字列に入る。ただし、その数値は変更する必要がある。 2^7 ビットを 2^27 ビットの代わりに設定する必要がある。こうして、値が 128 から 255 の間になるようにする。
- ほかのキーボード文字イベントは文字列に収まらない。これは、 128 から 255 の範囲のキーボード・イベントを含む。

`read-key-sequence` のような関数は、これら規則にしたがったキーボード入力文字の文字列を構築することができます。これら関数は、イベントが文字列に収まらない場合には文字列の代わりにベクタを構築します。

リード構文 '`\M-`' を文字列中で使用した場合、 128 から 255 の範囲内のコードが作られます (対応するキーボード・イベントを文字列に入れるために修正したときに得られるコードと同じものです)。そのため、文字列中のメタ・イベントは、それらがどのように文字列に入ってくるかに関わらず、一貫して機能します。

キーボード・イベントとしてのメタ文字の表現を変更した理由は、127 を超える基本文字コードのために空間を空けるためと、そういった大きな文字コードのメタ類をサポートするためです。

新たなプログラムでは、これらの特別な互換性規則の使用を避けることができます。メタ文字を含む可能性があるキー・シーケンスに対しては、文字列の代わりにベクタを使い、イベントの文字列をアクセスするには `listify-key-sequence` を使用すればいいのです。

listify-key-sequence *key* Function

この関数は、文字列あるいはベクタ *key* を、 `unread-command-events` に入れることができるイベントのリストに変換します。ベクタの変換は簡単ですが、文字列の変換は文字列中のメタ文字に使用する特別な表現のために込み入ってます。

18.6 入力の読込み

エディタ・コマンド・ループはキーボード入力を関数 `read-key-sequence` で読み込み、その関数は `read-event` を使っています。これらのおよびほかのキーボード入力のための関数も同様に、 Lisp プログラムで使用できるようになっています。 35.8 節「一時表示」 p.608 の `momentary-string-display` や、 18.7 節「時間の経過あるいは入力を待つ」 p.283 の `sit-for` を参照してください。端末入力モードの制御と端末入力のデバッグのための関数や変数については、 34.8 節「端末入力」 p.592 を参照してください。

高水準入力機能については、第 17 章「ミニバッファ」 p.237 を参照してください。

18.6.1 キー・シーケンス入力

コマンド・ループはキー・シーケンス入力を 1 回に一つ、 `read-key-sequence` を呼び出すことで読み込みます。 Lisp プログラムで同じようにこの関数を使えます。たとえば、 `describe-key` はこれを使って説明するキーを読み込みます。

read-key-sequence *prompt* Function

この関数は、キー・シーケンスを一つ読み込んで、それを文字列あるいはベクタで返します。イベントを読み込み続け、それが完全なキー・シーケンスになったとき止めます。即ち、現在アクティブなキーマップで、何らかの非前置コマンドを指定するに十分となるまでです。

イベントが全て文字で、その全てが文字列に収まるならば、 `read-key-sequence` は文字列を返します (18.5.14 節「文字列にキーボード・イベントを入れる」 p.278 参照)。それ以外ではベクタを返します。ベクタは全種類のイベント (文字、シンボル、リスト) を収めることができるからです。文字列やベクタの要素はキー・シーケンス中のイベントです。

関数 `read-key-sequence` は、とりやめを抑止します。つまりこの関数で読込み中にタイプされた `C-g` は、ほかのなんらかの文字のように働き、 `quit-flag` を設定しません。 18.8 節「とりやめ」 p.284 を参照してください。

引数 *prompt* は、プロンプトとしてエコー領域に表示されるべき文字列であるか、プロンプトを表示しないことを意味する `nil` であるか、のどちらかです。

以下の例では、プロンプト '?' がエコー領域に表示され、ユーザは `C-x C-f` をタイプします。

```
(read-key-sequence "?")

---------- Echo Area ----------
?C-x C-f
---------- Echo Area ----------

     ⇒ "^X^F"
```

num-input-keys Variable

この変数の値は、現在の Emacs セッションにおいてこれまで処理されたキー・シーケンスの個数です。これは端末から読み込んだキー・シーケンスと実行されたキーボード・マクロ自身を読み込んだときのキー・シーケンスとを含みます。

入力文字が大文字でキー・バインドはないが、その小文字のバインドならある場合、 `read-key-sequence` はその文字を小文字に変換します。 `lookup-key` は大文字小文字変換をこのように行なわないことに注意してください。

関数 `read-key-sequence` はマウス・イベントも同じように変形します。バインドされてないドラッグ・イベントをクリック・イベントに変換し、バインドされてないボタン押下イベントを完全に捨てます。同じようにフォーカス・イベントとそのほかのウィンドウ・イベントを並べ替えて、キー・シーケンス中にほかの種類のイベントが絶対現れないようにします。

マウス・イベントがウィンドウの特別な部分、たとえばモード行やスクロール・バーで発生した場合、イベント・タイプは何も特別なことを示しません。マウス・ボタンと修飾キーの組合せが、いつもの表現と同じシンボルになっています。ウィンドウ部分に関する情報は、イベントの違う場所、座標の中にあります。でも、 `read-key-sequence` はこの情報を架空の前置キーに変換します。これらは全てシンボルで、 `mode-line`, `vertical-line`, `horizontal-scroll-bar`, `vertical-scroll-bar` です。

これら架空の前置キーを使ったキー・シーケンスを定義することで、特別なウィンドウ部分内でのマウス・クリックに対する意味を定義できます。

たとえば、 `read-key-sequence` を呼び、マウスをウィンドウのモード行でクリックした場合、このような二つのイベントを得ます:

```
(read-key-sequence "Click on the mode line: ")
     ⇒ [mode-line
         (mouse-1
          (#<window 6 on NEWS> mode-line
           (40 . 63) 5959987))]
```

18.6.2 イベントを一つ読み込む

コマンド入力の最低水準関数は、これら単一のイベントを読み込むものです。

read-event Function

この関数は、コマンド入力の次のイベントを読み込んで返します。必要ならばイベントが来るまで待ちます。イベントはユーザから直接あるいはキーボード・マクロから来ます。

関数 `read-event` は入力待ちであることを示すメッセージは何も表示しません。もしそうしたいならば、 `message` をはじめに使ってください。メッセージを表示しなかったならば、 `read-event` はエコーすることでプロンプトを出します。つまり、現在のコマンドを導いた、または、現在のコマンドによって読み込まれたイベントを表す表示をします。35.4 節「エコー領域」 p.603 を参照してください。

`cursor-in-echo-area` が非 `nil` ならば、 `read-event` はカーソルを一時的にエコー領域に移動し、そこに表示される何らかのメッセージが終るまでそうしています。それ以外では、 `read-event` はカーソル移動しません。

`read-event` を呼んで右矢印ファンクション・キーを押下したとすると、こうなります:

```
(read-event)
     ⇒ right
```

read-char Function

この関数は、コマンド入力の 1 文字を読み込んで返します。文字を拾うまで、文字以外のいかなるイベントも捨てます。

最初の例では、ユーザが文字 `1`(ASCII コード 49) をタイプします。 2 番目の例はキーボード・マクロ定義で、ミニバッファから `eval-expression` を使って `read-char` を呼んでいます。 `read-char` がキーボード・マクロの直後の文字 `1` を読み込んでいます。次に `eval-expression` はその戻り値をエコー領域に表示します。

```
(read-char)
     ⇒ 49

;; これを評価するために M-: を使うと仮定。
(symbol-function 'foo)
     ⇒ "^[:(read-char)^M1"
(execute-kbd-macro 'foo)
     ⊣ 49
     ⇒ nil
```

18.6.3 Quote した文字入力

関数 `read-quoted-char` を使って、ユーザに文字を指定するよういい、ユーザが制御やメタ文字を簡便に指定できるようにすることができます。文字どおりでも、 8 進文字コードでもどちらでもできます。コマンド `quoted-insert` はこの関数を使っています。

read-quoted-char &optional *prompt* Function

この関数は、 `read-char` に似てますが、最初の文字が 8 進数字 (0-7) だったときにあと二つ 8 進数字を読み込んで (ただし、非 8 進数字が見つかれば止める)、これら 8 進数字で表現される文字を返すというところが異なります。

とりやめは最初の文字の読込みでは抑止されているので、ユーザは `C-g` を入力できます。18.8 節「とりやめ」 p.284 を参照してください。

prompt を与えると、それはユーザへのプロンプト文字列の指定となります。プロンプト文字列は常にエコー領域に表示され、 '-' がそれに続きます。

次の例では、ユーザが 8 進数 177(10 進で 127) をタイプしています。

```
(read-quoted-char "What character")

---------- Echo Area ----------
What character-177
---------- Echo Area ----------

     ⇒ 127
```

18.6.4 そのほかのイベント入力機能

本節ではイベントを取り出さずに「あらかじめ覗く」方法、未処理の入力を検査する方法、保留中の入力を捨てる方法を説明します。

unread-command-events Variable

この変数は、コマンド入力として読み込まれるのを待っているイベントのリストを保持しています。これらイベントはリストに現れた順に使われ、使われるにしたがって一つづつ取り除かれます。

この変数が必要だったわけは、場合によっては関数がイベントを読み込んで、それを使わないと決めることがあるからです。そのイベントをこの変数に格納すると、コマンド・ループやコマンド入力を読み込む関数が、それを普通に処理します。

例として、数値前置引数を実現する任意桁数の数字を読み込む関数があります。これが非数字イベントを見つけると、コマンド・ループが普通に読み込むように、それを読み戻さなければなりません。同様に、段階的検索はこの機能を使って検索時に特別な意味を持たないイベントを読み戻すようにします。というのも、これらイベントは検索を抜けてから、普通に実行しなければならないためです。

`unread-command-events` 中に置けるようにするために、キー・シーケンスからイベントを抜き出す信頼できる簡単な方法は、 `listify-key-sequence` を使うことです (18.5.14 節「文字列にキーボード・イベントを入れる」 p.278 参照)。

unread-command-char Variable

この変数は、コマンド入力として読み込まれるべき文字を保持します。値 -1 の意味は、「空」です。

この変数は、今やほとんど obsolete で、それは `unread-command-events` を代わりに使えるからです。これは Emacs バージョン 18 以前のために書かれたプログラムをサポートするためだけに存在します。

input-pending-p Function

この関数は、現在読込み可能なコマンド入力があるかないかを判別します。読込み可能な入力があれば `t` の値で、そうでなければ `nil` の値で、直ちに戻ります。ごくまれに読込み可能な入力がないのに `t` で戻ることもあります。

last-input-event Variable

この変数は、読み込んだ最新の端末入力イベントを記録します。コマンドの一部でも、Lisp プログラムによる明示的なものでも関係ありません。

次の例では、 Lisp プログラムが文字 `1`、 ASCII コード 49 を読み込みます。これは `last-input-event` の値になります。一方、 `C-e`(`C-x C-e` コマンドを使ってこの式を評価したと仮定) の方は `last-command-event` の値として残ります。

```
(progn (print (read-char))
       (print last-command-event)
       last-input-event)
     ⊣ 49
     ⊣ 5
     ⇒ 49
```

別名 `last-input-char` が、 Emacs バージョン 18 との互換のために存在します。

discard-input Function

この関数は、端末入力バッファの中身を捨てて、定義の最中のキーボード・マクロがあったならばキャンセルします。 `nil` を返します。

次の例では、この形式の評価が始まり次第、ユーザは沢山の文字をタイプすることもできます。 `sleep-for` がスリープを終り次第、 `discard-input` がスリープ中にタイプされた全ての文字を捨て去ります。

```
(progn (sleep-for 2)
       (discard-input))
     ⇒ nil
```

18.7 時間の経過あるいは入力を待つ

待ち関数は、一定の時間が経つか入力があるまで待つために設計されました。たとえば、計算の途中でポーズを置いて、ユーザが表示を眺めることができるようにしたいことがあるかもしれません。 `sit-for` はポーズを置いて画面を更新し、入力があれば直ちに戻ります。一方、 `sleep-for` は画面の更新はしません。

sit-for *seconds* &optional *millisec nodisp* Function

この関数は、 (ユーザの未処理の入力がないならば) 再表示を実行し、 *seconds* 秒か入力があるまで待ちます。値が `t` となるのは、 `sit-for` が入力が来ないでフルに待った場合

です (18.6.4 節「そのほかのイベント入力機能」p.282 の `input-pending-p` 参照)。それ以外のときは、値は `nil` です。

引数 *seconds* は整数でなくても構いません。浮動小数点数ならば、 `sit-for` は小数で指定した秒を待ちます。いくつかのシステムでは秒未満をサポートしていません。このようなシステムでは、 *seconds* は切り捨てられます。

オプション引数 *millisec* はミリ秒単位の追加待ち時間を指定します。これは *seconds* で指定した時間に追加されます。システムが小数秒をサポートしない場合の非 0 の *millisec* 指定は、エラーとなります。

再表示は入力が来ると常に preempt されるので、再表示が始まる前に入力が来ると、再表示は全く行なわれなくなります。そのため、未処理の入力がある場合に画面の更新を強制する方法はありません。けれども、未処理の入力がなければ `(sit-for 0)` を使って遅延なしに更新を強制できます。

nodisp が非 `nil` ならば、 `sit-for` は再表示をしません。しかし相変わらず、入力があり次第 (または時間経過後すぐに) 戻ります。

フレームのアイコン化あるいはアイコン化解除があると `sit-for` から戻ります。というのも、これらはイベントを発生するからです。 18.5.10 節「そのほかのウィンドウ・システム・イベント」p.274 を参照してください。

`sit-for` の普通の目的は、表示したテキストを読む時間をユーザに与えることです。

sleep-for *seconds* &optional *millisec* Function

この関数は、単純に *seconds* 秒ポーズを置きます。表示は更新しません。入力も全く気にしません。 `nil` を返します。

引数 *seconds* は整数でなくても構いません。浮動小数点数ならば、 `sleep-for` は小数で指定した秒を待ちます。いくつかのシステムでは秒未満をサポートしていません。このようなシステムでは、 *seconds* は切り捨てられます。

オプション引数 *millisec* はミリ秒単位の追加待ち時間を指定します。これは *seconds* で指定した時間に追加されます。システムが小数秒をサポートしない場合には、非 0 の *millisec* 指定はエラーとなります。

確実に遅延を取りたい場合は `sleep-for` を使ってください。

現在時刻を得るための関数については、 34.5 節「1 日の時刻」p.588 を参照してください。

18.8 とりやめ

Lisp 関数実行中の `C-g` のタイプは、 Emacs がやっていることを何であれ『とりやめ (quit)』ます。これは制御が最も内側のアクティブなコマンド・ループに帰ることを意味します。

コマンド・ループがキーボード入力を待っているときに `C-g` をタイプしてもとりやめになりません。これは普通の入力文字同様に働きます。最も単純な例では、違いを見分けることはできません。なぜなら、 `C-g` は普通はコマンド `keyboard-quit` を実行し、その効果はとりやめだから

です。けれども、前置キーに続く `C-g` の場合、結果は未定義キーとなります。この効果は、前置引数もろとも前置キーをキャンセルします。

ミニバッファ中では、 `C-g` は異なる定義を持ちます。即ち、ミニバッファを中断して外へ出るのです。つまり、事実上、ミニバッファを抜けてとりやめます (単にとりやめると、ミニバッファ中のコマンド・ループに戻る)。コマンド・リーダが入力を読み込んでいるときに、直に `C-g` でとりやめないのは、こうするとミニバッファ中で `C-g` の意味を再定義できるからです。前置キーに続く `C-g` はミニバッファ中で再定義されていないので、前置キーと前置引数とをキャンセルするという普通の効果を持ちます。これも、 `C-g` が常に直にとりやめるのだったら、できなかったことです。

`C-g` が直にとりやめるときは、変数 `quit-flag` を `t` に設定することでそうします。 Emacs はこの変数を適切なときに検査して、値が非 `nil` ならばとりやめます。そのためどんな方法でも、`quit-flag` を非 `nil` に設定すると、とりやめを引き起こします。

C コードのレベルでは、とりやめはどこでも起こるというわけではありません。 `quit-flag` を検査している特別な場所でのみ起こります。この理由は、ほかの場所でとりやめると Emacs の内部状態を一貫性のない状態にしてしまうかもしれないからです。とりやめが安全な場所まで遅延されるので、とりやめで Emacs がクラッシュすることはありません。

ある種の `read-key-sequence` や `read-quoted-char` のような関数は入力待ち中だったとしてもとりやめを完全に防ぎます。とりやめの代わりに、 `C-g` は要求された入力として働きます。`read-key-sequence` の場合なら、コマンド・ループ中の `C-g` の特別な振舞をするように働きます。 `read-quoted-char` の場合なら、 `C-q` で `C-g` を quote できるようにするために、こうなってます。

Lisp 関数内のある部分でとりやめを防ぎたいならば、変数 `inhibit-quit` を非 `nil` に束縛すればできます。こうすると、 `C-g` が `quit-flag` を `t` にいつもどおり設定しても、普通なら起こるはずのとりやめが起きません。そのうち、 `inhibit-quit` が再び `nil` になったとします。たとえば `let` 形式の終りでその束縛が解かれるなどです。そのとき、 `quit-flag` がいまだ非 `nil` だったならば、要求されていたとりやめが直ちに起こります。この振舞は、プログラムの「クリティカル・セクション」内でとりやめが絶対起こらないようにしたい場合にふさわしいものです。

(`read-quoted-char` のような) 関数によっては、 `C-g` は特別な方法で扱われてとりやめを起こしません。 `inhibit-quit` を `t` に束縛して入力を読み込み、 `inhibit-quit` を `nil` にする前に `quit-flag` を `nil` に再び設定すると、こうなります。 `read-quoted-char` の定義からの引用が、これをどう行なうかを示しています。これは通常のとりやめが最初の文字の入力後は許されていることも示しています。

```
(defun read-quoted-char (&optional prompt)
  "...documentation..."
  (let ((count 0) (code 0) char)
    (while (< count 3)
      (let ((inhibit-quit (zerop count))
            (help-form nil))
        (and prompt (message "%s-" prompt))
        (setq char (read-char))
```

```
        (if inhibit-quit (setq quit-flag nil)))
      ...)
    (logand 255 code)))
```

quit-flag Variable

この変数が非 nil ならば、 inhibit-quit が非 nil でないかぎり、 Emacs は直ちにとりやめます。 C-g のタイプで、普通 quit-flag は inhibit-quit の値に関わらず、非 nil に設定されます。

inhibit-quit Variable

この変数は、 quit-flag が nil 以外の値に設定されているときに、 Emacs がとりやめるべきか否かを決定します。 inhibit-quit が非 nil ならば、 quit-flag は特別な効果を持ちません。

keyboard-quit Command

この関数は、 quit 条件を (signal 'quit nil) によって通知します。これは、とりやめと同じことを行ないます (9.5.3 節「エラー」 p.119 の signal を参照)。

C-g 以外の文字をとりやめのために指定できます。関数 34.8 節「端末入力」 p.592 の set-input-mode を参照してください。

18.9 前置コマンド引数

ほとんどの Emacs コマンドは、コマンド自身に先立って指定される数値である『前置引数 (prefix argument)』を使えます (前置引数と前置キー (prefix key) を混同しないように)。前置引数はいつでも一つの値で表現され、それは現在前置引数はないという意味の nil かもしれません。 Emacs コマンドは前置引数を使うかもしれないし、無視するかもしれません。

前置引数には二つの表現があります。それらは『そのまま (raw)』と『数値 (numeric)』です。エディタ・コマンド・ループはそのままの表現を内部的に使用し、その情報を格納する Lisp 変数もそのようにしますが、コマンドはいずれの表現も要求できます。

以下がそのままの前置引数の取り得る値です:

- nil、前置引数がないことを意味します。この数値は 1 ですが、多くのコマンドが nil と整数 1 とを区別します。
- 整数、それ自身を表します。
- 要素一つのリスト、この要素は整数です。この形式の前置引数は、数値なしで 1 回以上 C-u の連打した結果です。数値はリスト中の整数で、コマンドによってはこういったリストと数字単体とを区別します。
- シンボル -。これは、数値なしで M-- あるいは C-u - をタイプしたことを示します。これと等価な数値は -1 ですが、コマンドによっては整数 -1 とシンボル -とを区別します。

これら可能性は、次の関数をさまざまな前置で呼び出すことで例示できます。

```
(defun display-prefix (arg)
  "Display the value of the raw prefix arg."
  (interactive "P")
  (message "%s" arg))
```

以下がさまざまなそのままの前置引数で display-prefix を呼び出した結果です:

```
        M-x display-prefix  ⊣ nil

C-u     M-x display-prefix  ⊣ (4)

C-u C-u M-x display-prefix  ⊣ (16)

C-u 3   M-x display-prefix  ⊣ 3

M-3     M-x display-prefix  ⊣ 3      ; (C-u 3 と同じ。)

C-u -   M-x display-prefix  ⊣ -

M--     M-x display-prefix  ⊣ -      ; (C-u - と同じ。)

C-u - 7 M-x display-prefix  ⊣ -7

M-- 7   M-x display-prefix  ⊣ -7     ; (C-u -7 と同じ。)
```

Emacs は前置引数を格納するために二つの変数を使用します。それらは prefix-arg と current-prefix-arg です。 universal-argument のようなほかのコマンドのために前置引数を準備するコマンドが、値を prefix-arg に格納します。対照的に、 current-prefix-arg は現在のコマンドに前置引数を伝えます。これに値を設定してもこれ以降のコマンドの前置引数には何の影響もありません。

普通コマンドは、前置引数に数値あるいはそのまま、どちらの表現を使うかを interactive 宣言で指定します (18.2.1 節「interactive を使う」p.258 を参照してください)。その一方で、関数は前置引数の値を直に変数 current-prefix-arg で見るかも知れません。しかしこれは若干汚いやり方です。

prefix-numeric-value *arg* Function

この関数は、有効なそのままの前置引数 *arg* の数としての意味を返します。引数はシンボル、数、リストです。 nil ならば、値 1 が返ります。 -ならば、 -1 が返ります。数ならば、その数が返ります。リストならば、リストの CAR(数であるはず) が返ります。

current-prefix-arg Variable

この変数は、現在のコマンドのそのままの前置引数を保持します。コマンドは直接それを調べても構いませんが、普通は (interactive "P") でアクセスします。

prefix-arg Variable

この変数の値は、次の編集コマンドのそのままの前置引数です。次のコマンドのために前置引数を指定するコマンドは、この変数を設定する働きがあります。

関数 `universal-argument`, `digit-argument`, `negative-argument` は、ユーザに次のコマンドのための前置引数を入力させようというのでないかぎり、呼び出さないでください。

universal-argument Command

本コマンドは入力を読み込み、次のコマンドのための前置引数を指定します。本コマンドを自分で呼び出す場合は、自分が何をしているのかを理解の上でやってください。

digit-argument *arg* Command

本コマンドは、次のコマンドのために前置引数を追加します。引数 *arg* は、本コマンド前までのそのままの前置引数です。これは、前置引数を更新するために使います。本コマンドを自分で呼び出す場合は、自分が何をしているのかを理解の上でやってください。

negative-argument *arg* Command

本コマンドは、次のコマンドのための数値引数を追加します。引数 *arg* は本コマンド前までのそのままの前置引数です。この値は新たな前置引数を作るために負数にされます。本コマンドを自分で呼び出す場合は、自分が何をしているのかを理解の上でやってください。

18.10 再帰編集

Emacs のコマンド・ループには、 Emacs 起動時に自動的に入ります。このトップレベルで起動したコマンド・ループを抜けることはありません。 Emacs が実行を続けるかぎり、これも実行を続けます。 Lisp プログラムもやはりコマンド・ループを起動できます。これはコマンド・ループを複数回アクティブにするので、『再帰編集 (recursive edit)』と呼びます。再帰編集レベルは、それを起動したいかなるコマンドをもサスペンドし、そのコマンドに戻るまではユーザに任意の編集を許します。

再帰編集の間に利用可能なコマンドは、トップレベルの編集ループで利用可能なものと同一であり、キーマップに定義されています。いくつかの特殊なコマンドのみが再帰編集レベルを抜けます。それ以外は終了時にその再帰編集レベルに戻ります (抜けるための特殊なコマンドは常に利用可能ですが、それらは再帰編集中以外は何もしません)。

再帰的なものを含む、全てのコマンド・ループは汎用のエラー・ハンドラを準備します。そうすることでコマンド・ループから実行したコマンド中でエラーが起きても、コマンド・ループを抜けることがなくなります。

ミニバッファ入力は特別な種類の再帰編集です。いくつかの特徴があり、ミニバッファとミニバッファ・ウィンドウの表示を可能にしたりできますが、思ったほど多くはありません。いくつかのキーはミニバッファ中では異なる振舞をします。しかし、それはミニバッファのローカル・マップのせいでしかありません。したがってウィンドウを切替えれば、普通の Emacs コマンドが使えます。

再帰編集レベルを起動するには、関数 `recursive-edit` を呼び出します。この関数は、コマンド・ループを含みます。タグ `exit` での `catch` 呼出しも含むので、 `exit` を throw することで再帰編集レベルを抜けることが可能になります (9.5.1 節「明示的な非局所脱出: `catch` と `throw`」p.116 参照)。 `t` 以外の値を throw したならば、 `recursive-edit` は普通にそれを呼

び出した関数に戻ります。コマンド C-M-c(exit-recursive-edit) は exit の throw を行ないます。 t 値を throw すると recursive-edit をとりやめ、その結果制御は一つ上のコマンド・ループに戻ります。これを『中断 (abort)』と呼び、 C-] で行ないます (abort-recursive-edit)。

ほとんどのアプリケーションでは、ミニバッファの使用の一部を除いて再帰編集を使うべきではありません。普通、カレント・バッファの主モードを一時的に特別な主モードに変更すると、ユーザにとって使い易くなります。この特別なモードは、直前のモードに戻るためのコマンドを持たなければなりません (Rmail の e コマンドはこのテクニックを使っています)。あるいは特別なモードの新しいバッファを作成して、それを選択し、その中でユーザに「再帰的に」編集する別のテキストを与えたいと思うことがあるかもしれません。このモードでは、処理を完結させ、直前のバッファに戻るコマンドを定義しておきます (Rmail の m コマンドはこれを行ないます)。

再帰編集はデバッグに便利です。関数定義にブレークポイントの一種として debug 呼出しを挿入しておき、関数がそこへきたら調べられるようにすることができます。 debug は再帰編集の起動だけでなく、デバッガのほかの機能も備えています。

再帰編集レベルは、 query-replace で C-r をタイプしたときや C-x q (kbd-macro-query) を使ったときにも使われます。

recursive-edit Function

この関数は、エディタ・コマンド・ループを起動します。 Emacs 初期化では自動的に呼び出され、ユーザが編集を開始できるようにします。 Lisp プログラムから呼び出した場合、再帰編集レベルに入ります。

次の例では、関数 simple-rec がはじめにポイントを 1 語進め、再帰編集に入り、エコー領域にメッセージを表示します。そこでユーザは望みの編集を行なうことができます。それから C-M-c のタイプでそれを抜けて、 simple-rec の実行を継続できます。

```
(defun simple-rec ()
  (forward-word 1)
  (message "Recursive edit in progress")
  (recursive-edit)
  (forward-word 1))
     ⇒ simple-rec
(simple-rec)
     ⇒ nil
```

exit-recursive-edit Command

この関数は、最も内側の (ミニバッファ入力を含む) 再帰編集から抜けます。この定義は事実上 (throw 'exit nil) です。

abort-recursive-edit Command

この関数は、最も内側の (ミニバッファ入力を含む) 再帰編集を抜けた後、その再帰編集を要求したコマンドを quit の通知によって中断します。この定義は事実上 (throw 'exit t) です。 18.8 節「とりやめ」 p.284 を参照してください。

top-level Command

この関数は、全ての再帰編集レベルを抜けます。その結果いかなる計算からも完全に飛び出してメインのコマンド・ループに直接戻ってしまうので、値を返しません。

recursion-depth Function

この関数は、現在の再帰編集の深さを返します。再帰編集にない場合、 0 を返します。

18.11 コマンドの禁止

『コマンドの禁止 (disabling a command)』は、その実行の前にユーザの確認を要求するようにコマンドに印をつけます。禁止は、初心者が混乱しやすいコマンドを、彼らが間違って使うのを防ぐために使います。

コマンドを禁止する低水準の機構は、そのコマンドの Lisp シンボルの `disabled` 属性に非 `nil` を置くことです。これら属性は普通ユーザの '`.emacs`' ファイルでこのような Lisp 式によって設定されます。

```
(put 'upcase-region 'disabled t)
```

いくつかのコマンドは、これら属性がデフォルトで存在し、 '`.emacs`' ファイルでそれを取り除いているかも知れません。

`disabled` 属性の値が文字列なら、このコマンドが禁止されている旨を告げるメッセージがその文字列を含んでいます。こんなふうに:

```
(put 'delete-region 'disabled
     "Text deleted this way cannot be yanked back!\n")
```

禁止されたコマンドを対話的に起動した場合に何が起こるかの詳細は、 *The GNU Emacs Manual* の "Disabling" 節を参照してください。コマンドの禁止は Lisp プログラムからそれを関数として呼び出す場合には何の効果も持ちません。

enable-command *command* Command

これ以後、 *command* の実行に特別な確認が不要になります。そして、 (ユーザがいいといえば) ユーザの '`.emacs`' ファイルを書き換えて、今後のセッションでもそうなるようにします。

disable-command *command* Command

これ以後、 *command* の実行には特別な確認が必要になります。そして、 (ユーザがいいといえば) ユーザの '`.emacs`' ファイルを書き換えて、今後のセッションでもそうなるようにします。

disabled-command-hook Variable

本正規フックは、ユーザが禁止されたコマンドを対話的に起動したときに、そのコマンドの代わりに実行されます。フック関数は `this-command-keys` を使ってユーザがこのコマンドを実行するために何をタイプしたかを判別することができます。そのため、何のコマンドだったかを知ることができます。 20.4 節「フック」 p.335 を参照してください。

デフォルトでは、 `disabled-command-hook` はユーザに続行するか否かを尋ねる関数を含んでいます。

18.12 コマンド履歴

コマンド・ループはそれまでに実行された複雑なコマンドの履歴を持っています。こうすることで、これらコマンドの繰り返しの便宜を図ります。『複雑なコマンド (complex command)』とは、対話引数の読込みにミニバッファを使うものをいいます。これは、 `M-x` コマンドの全て、 `M-:` コマンドの全て、そして `interactive` 指定でミニバッファから引数を読み込むコマンド全てを含みます。コマンド自身の実行中にミニバッファを明示的に使用しても、そのコマンドを複雑とは考えません。

command-history Variable

この変数の値は、最近の複雑なコマンドのリストで、各要素が評価されるべき形式で表現されています。編集セッションの間中、この変数に全ての複雑なコマンドが記録し続けますが、ガベージ・コレクションが発生すると、最初 (最近) の 30 個以外の要素は削除されます (B.3 節「ガベージ・コレクション」p.647 参照)。

```
command-history
⇒ ((switch-to-buffer "chistory.texi")
    (describe-key "^X^[")
    (visit-tags-table "~/emacs/src/")
    (find-tag "repeat-complex-command"))
```

この履歴リストは実際はミニバッファ履歴の特別な場合ですが (17.4 節「ミニバッファ履歴」p.241 参照)、一つだけ捻りが加えられています。つまり、要素は文字列ではなく式となってます。

沢山のコマンドが以前のコマンドの編集と呼出しのために用意されています。コマンド `repeat-complex-command`, `list-command-history` はユーザ・マニュアル (*The GNU Emacs Manual* の “Repetition” 節参照) に説明してあります。ミニバッファ中では、使用された履歴コマンドは任意のミニバッファで利用可能なものと同じです。

18.13 キーボード・マクロ

『キーボード・マクロ (keyboard macro)』は、入力イベント・シーケンスの缶詰で、コマンドとみなすこともできるし、キー定義をなすこともできます。キーボード・マクロの Lisp 表現は、文字列あるいはイベントを含むベクタです。 Lisp のマクロとキーボード・マクロとを混同しないようにしてください (第 12 章「マクロ」p.163 参照)。

execute-kbd-macro *macro* &optional *count* Function

この関数は、 *macro* をイベントのシーケンスとして実行します。 *macro* が文字列あるいはベクタなら、その中のイベントはユーザによって入力されたのと全く同じように実行されます。シーケンスが単一のキー・シーケンスであると思ってはいけません。つまり、普通キーボード・マクロ定義は複数のキー・シーケンスを連結したものです。

macro がシンボルならば、その関数定義を *macro* の代わりに使います。それがまた別のシンボルならば、この処理を繰り返します。最終的には結果が文字列あるいはベクタになるはずです。結果がシンボル、文字列、ベクタのいずれでもない場合は、エラーが通知されます。

引数 *count* は繰り返し回数です。ここでは *macro* がその回数だけ実行されます。 *count* が省略されるか、 `nil` だった場合、 *macro* は 1 回だけ実行されます。 0 ならば、 *macro* はエラーとなるか検索に失敗するまで、延々と実行されます。

executing-macro Variable

この変数の内容は、現在実行中のキーボード・マクロを定義している文字列あるいはベクタです。その時点で何のマクロも実行していないなら、値は `nil` です。コマンドを実行中のマクロから実行した場合、コマンドはこの変数を検査して振舞を変えることができます。自分自身でこの変数を設定するのは止めてください。

defining-kbd-macro Variable

この変数は、キーボード・マクロの定義中であるか否かを示します。コマンドはこの変数を検査して、マクロ定義中ならば振舞を変えることができます。コマンド `start-kbd-macro` と `end-kbd-macro` とは、この変数を設定します (自分自身でこの変数を設定するのは止めてください)。

この変数は、常に現在の端末に関してローカルであり、バッファローカルにはできません。 26.2 節「複数のディスプレイ」p.435 を参照してください。

last-kbd-macro Variable

この変数は、最新のキーボード・マクロを定義したものです。その値は文字列あるいはベクタ、あるいは `nil` です。

この変数は、常に現在の端末に対してローカルであり、バッファローカルにはできません。 26.2 節「複数のディスプレイ」p.435 を参照してください。

19 キーマップ

入力イベントとコマンドとのバインドは『キーマップ (keymap)』と呼ばれるデータ構造に記録されています。キーマップの中のそれぞれのバインドは、個々のイベント・タイプを別のキーマップやコマンドに関連づけ (または『バインド (bind)』[1]し) ます。あるイベントがキーマップにバインドされたときは、次の入力イベントを調べるのにそのキーマップを使います。これはコマンドが見つかるまで続けられます。この処理全体のことを『キー検索 (key lookup)』と呼びます。

19.1 キーマップの用語

『キーマップ (keymap)』はイベント・タイプと定義 (どんな Lisp オブジェクトでも構いませんが、コマンド・ループで実行することができる型でなければなりません) を対応づける表です。イベント (またはイベント・タイプ) とキーマップが与えられれば、 Emacs はそのイベントの定義を得ることができます。イベントには通常 ASCII 文字、ファンクション・キー、マウスの動作が含まれます (18.5 節「入力イベント」p.266 参照)。

一つの単位を構成する一連の入力イベントを『キー・シーケンス (key sequence)』、または略して『キー (key)』と呼びます。一つのイベントは常にキー・シーケンスですし、ある複数のイベントのまとまりもそうです。

キーマップはどんなキー・シーケンスに対してもバインドや定義を決定します。キー・シーケンスが一つのイベントの場合、そのイベントの定義は最初のキーマップのバインドです。複数のイベントからなるキー・シーケンスのバインドは、反復処理で求めます。まず、最初のイベントのバインドを探します。それはキーマップのはずです。次に 2 番目のイベントのバインドをそのキーマップの中から探します。これをキー・シーケンスの全てのイベントを使い切るまで繰り返します。

あるキー・シーケンスのバインドがキーマップであるとき、そのキー・シーケンスを『前置キー (prefix key)』と呼びます。そうでなければ、『完結キー (complete key)』と呼びます (つけ加えるイベントがないので)。バインドが `nil` のときは『未定義の (undefined)』キーと呼びます。前置キーの例としては、 `C-c`、 `C-x`、 `C-x 4` があります。定義済みの完結キーの例としては、 `X`、 `RET`、 `C-x 4 C-f` があります。未定義の完結キーの例としては、 `C-x C-g`、 `C-c 3` があります。詳細は 19.5 節「前置キー」p.297 を参照してください。

キー・シーケンスのバインドを探す規則により、中間のバインド (最後のイベントより前のイベントのバインド) は全てキーマップだとみなします。そうでないならその一連のイベントは一つの単位ではありません。つまり本当のキー・シーケンスではありません。言い替えると、正しいキーの後ろの方からいくつかのイベントを取り除くと、必ず前置キーになります。たとえば、 `C-f C-n` はキーではありません。 `C-f` は前置キーではないので、 `C-f` ではじまる、より長いシーケンスもキーになることはありません。

[1] bind はほかの章では「束縛」と訳されていますが、この章では key binding を「キー束縛」とはしたくなかったので、「バインド」と訳します。

複数のイベントからなる可能なキー・シーケンスの集合は、前置キーのバインドに依存していることに注意してください。したがって、違うキーマップでは違う集合になり得ますし、バインドを変更すれば変わり得ます。しかし、単独のイベントが正しいキーかどうかはどの前置キーにも依存しないので、常にキー・シーケンスです。

どんなときでも、いくつかの基本キーマップが『アクティブ (active)』になっています — すなわち、キー・バインドを探すのに使われます。それらは全てのバッファで共有される『グローバル・マップ (global map)』と、通常指定されている主モードの『ローカル・キーマップ (local keymap)』と、現在使用している副モードに付随する『副モード・キーマップ (minor mode keymap)』です (全ての副モードがキーマップをもつわけではありません)。副モード・キーマップはないかもしれませんし、複数あるかもしれません。ローカル・キーマップのバインドは対応するグローバル・マップのバインドを隠します (すなわち優先されます)。副モード・キーマップはローカルとグローバルのキーマップの両方を隠します。詳しくは 19.6 節「アクティブ・キーマップ」p.298 を参照してください。

19.2 キーマップのフォーマット

キーマップはその CAR がシンボル `keymap` であるリストです。リストの残りの要素がそのキーマップのキー・バインドを定義します。あるオブジェクトがキーマップかどうかをテストするには、関数 `keymapp`(以下を見てください) を使ってください。

それぞれの普通のバインドは特定の『イベント・タイプ (event type)』(たいてい文字やシンボルです) のイベントに適用されます。 18.5.12 節「イベントの分類」p.275 を参照してください。

キーマップの普通の要素は (*type* . *binding*) という形式のコンス・セルです。これはイベント・タイプ *type* に対するバインドを一つ指定します。

CAR が `t` であるようなコンス・セルは『デフォルト・キー・バインド (default key binding)』です。キーマップのほかの要素でバインドされない全てのイベントは、与えられた *binding* にバインドされます。デフォルト・バインドを使うと、起こりうる全てのイベント・タイプをいちいち列挙しないでもバインドすることができます。デフォルト・バインドがあるキーマップは、より低い優先度のキーマップを完全に隠してしまいます。

キーマップのある要素がベクタならば、そのベクタは全ての ASCII 文字のバインドとみなされます。ベクタの要素 n はコード n の文字のバインドになります。これはたくさんのバインドを記録するコンパクトなやり方です。このようなベクタを含むキーマップを『密なキーマップ (full keymap)』と呼びます。そうでないキーマップは『疎なキーマップ (sparse keymap)』といいます。

キーマップがベクタを含んでいる場合は、そのベクタは、もし要素が `nil` だったとしても常に全ての ASCII 文字のバインドを定義します。そのような `nil` のバインドは、そのキーマップのどんなデフォルト・バインドも無効にします。しかし、デフォルト・バインドは ASCII 文字以外のイベントに対しては依然として有効です。 `nil` のバインドは優先度の低いキーマップを無効にしません。したがって、ローカル・マップに `nil` のバインドがある場合、 Emacs はグローバル・マップのバインドを使います。

バインドはさておき、キーマップは文字列も要素としてもつことができます。これを『見出し文字列 (overall prompt string)』と呼び、キーマップをメニューとして使えるようにしています。19.12 節「メニュー・キーマップ」p.311 を参照してください。

キーマップにはメタ文字 (コードが 128 から 255 の文字) のバインドを直接記録することはできません。その代わり、キー検索のためにメタ文字を最初の文字が `ESC`(または、現在の `meta-prefix-char` の値) である 2 文字のシーケンスとみなします。したがって、`M-a` というキーは実際には `ESC a` と表され、そのグローバル・バインドは `esc-map` の中の `a` の位置に見い出されます (19.5 節「前置キー」p.297 参照)。

次に示すのは Lisp モードのローカル・キーマップの例です。これは疎なキーマップで、 `DEL` と `TAB` に加えて `C-c C-l`、 `M-C-q`、 `M-C-x` のバインドを定義しています。

```
lisp-mode-map
⇒
(keymap
 ;; TAB
 (9 . lisp-indent-line)
 ;; DEL
 (127 . backward-delete-char-untabify)
 (3 keymap
    ;; C-c C-l
    (12 . run-lisp))
 (27 keymap
     ;; M-C-q、ESC C-q として扱われる
     (17 . indent-sexp)
     ;; M-C-x、ESC C-x として扱われる
     (24 . lisp-send-defun)))
```

keymapp *object* Function

この関数は *object* がキーマップのときは `t` を、そうでないときは `nil` を返します。もっと正確にいうと、この関数はリストの CAR が `keymap` であるかどうかテストをします。

```
(keymapp '(keymap))
    ⇒ t
(keymapp (current-global-map))
    ⇒ t
```

19.3 キーマップの作成

ここではキーマップを作成するための関数について説明します。

make-keymap &optional *prompt* Function

この関数は、新しい密な (すなわち、全ての ASCII 文字を定義するための長さが 128 のベクタの含む) キーマップを作成し、返します。新しいキーマップの全ての ASCII 文字のバインドの初期値は `nil` で、それ以外のどの種類のイベントもバインドしません。

```
(make-keymap)
    ⇒ (keymap [nil nil nil ... nil nil])
```

prompt を指定すると、そのキーマップの見出し文字列になります。見出し文字列はメニュー・キーマップのために役立ちます。(19.12 節「メニュー・キーマップ」p.311 参照)。

make-sparse-keymap &optional *prompt* Function

この関数は何も登録されていない新しい疎なキーマップを作成し、返します。新しいキーマップはどんなイベントもバインドしません。引数 *prompt* には `make-keymap` と同様に見出し文字列を指定します。

```
(make-sparse-keymap)
    ⇒ (keymap)
```

copy-keymap *keymap* Function

この関数は *keymap* のコピーを返します。 *keymap* にバインドとして直接現れるキーマップは、全て再帰的にどの深さまででもコピーされます。ですが、文字の定義がキーマップを返す関数定義をもったシンボルであるときには、そのキーマップはコピーされず、新しいコピーには同じそのシンボルが現れます。

```
(setq map (copy-keymap (current-local-map)))
⇒ (keymap
     ;; (メタ文字の実装です。)
     (27 keymap
         (83 . center-paragraph)
         (115 . center-line))
     (9 . tab-to-tab-stop))

(eq map (current-local-map))
    ⇒ nil
(equal map (current-local-map))
    ⇒ t
```

19.4 継承とキーマップ

キーマップは、別のキーマップのバインドを継承することができます。これを行なうには、キーマップの “最後” を別に存在する継承したいキーマップにします。キーマップはこのようになります:

(`keymap` *bindings*... . *other-keymap*)

このキーマップは、キーを検索するときに *other-keymap* に存在するすべてのバインドを受け継ぎますが、 *bindings* でそれらのバインドを上書きしたりつけ加えたりすることができます。

other-keymap のバインドを `define-key` やほかのキー・バインド関数を使って変更すると、その変更が *bindings* で隠されていないかぎり継承しているキーマップにも反映されます。逆は真ではありません。 `define-key` を使って継承しているキーマップを変更した場合は *bindings* に作用し、 *other-keymap* には何の影響もありません。

text-mode-map を継承するキーマップの作り方の例を次に示します:

```
(setq my-mode-map (cons 'keymap text-mode-map))
```

19.5 前置キー

『前置キー (prefix key)』は、その前置キーではじまるキー・シーケンスが何をするのかを定義するキーマップと結びつけられています。たとえば、 C-x は前置キーで、変数 ctl-x-map に入っているキーマップを使います。 Emacs の標準の前置キーとそれらのキーマップの一覧を示します:

- esc-map は、 ESC に続くイベントに使われます。したがって、全てのメタ文字のグローバル定義は実際にはここにあります。このマップは ESC-prefix の関数定義でもあります。
- help-map は、 C-h に続くイベントに使われます。
- mode-specific-map は、 C-c に続くイベントに使われます。このマップは実際にはモード固有 (mode specific) ではありませんが、 C-c 前置キーのおもな利用方法を説明する C-h b (display-bindings) でユーザに情報を与えるためにこの名前がつけられています。
- ctl-x-map は、 C-x に続くイベントに使われます。このマップは Control-X-prefix の関数定義でもあります。
- ctl-x-4-map は、 C-x 4 に続くイベントに使われます。
- ctl-x-5-map は、 C-x 5 に続くイベントに使われます。
- 前置キー C-x n、 C-x r、 C-x a は名前がついていないキーマップを使います。

前置キーのバインドは、その前置キーに続くイベントを検索するために使うキーマップです (キーマップの代わりに、関数定義がキーマップであるようなシンボルでも構いません。効果は同じですが、シンボルは前置キーの名前になります)。そういうわけで、 C-x のバインドは、関数定義が C-x コマンドのキーマップであるシンボル Control-X-prefix となります (ctl-x-map の値も同じキーマップになります)。

前置キーの定義はどのアクティブなキーマップにも存在することができます。 C-c、 C-x、 C-h、 ESC の前置キーとしての定義はグローバル・マップに存在しますので、これらの前置キーは常に使えます。主モードと副モードは、ローカル・マップまたは副モードのマップで前置キーとして定義することによって、あるキーを前置キーとして定義し直すことができます。 19.6 節「アクティブ・キーマップ」p.298 を参照してください。

あるキーが複数のアクティブなマップで前置キーとして定義されている場合は、実際にはそのさまざまな定義は合成されます。副モード・キーマップで定義されたコマンドが最初に来て、次にローカル・キーマップの、最後にグローバル・マップのが来ます。

下記の例では、 C-p を C-x と等価にすることによってローカル・キーマップで C-p を前置キーにしています。 C-p C-f のバインドは C-x C-f と同様に関数 find-file になっています。キー・シーケンス C-p 6 はどのアクティブ・キーマップにも存在しません。

```
(use-local-map (make-sparse-keymap))
    ⇒ nil
```

```
(local-set-key "\C-p" ctl-x-map)
     ⇒ nil
(key-binding "\C-p\C-f")
     ⇒ find-file

(key-binding "\C-p6")
     ⇒ nil
```

define-prefix-command *symbol* Function

この関数は *symbol* を前置コマンドとして定義します。すなわち、密なキーマップを作成して、それを *symbol* の関数定義とします。あるキーのバインドをシンボルにすると、そのキーは名前つきの前置キーになります。また、この関数は *symbol* をそのキーマップを値とする変数にします。返り値は *symbol* です。

Emacs バージョン 18 では、この関数は *symbol* の関数定義をセットするだけで、変数の値はセットしません。

19.6 アクティブ・キーマップ

通常、 Emacs はたくさんのキーマップを持っていますが、任意の時点でその中のいくつかが『アクティブ (active)』、すなわちユーザの入力を解釈するのに関係しています。そのいくつかとは、グローバル・キーマップ、カレント・バッファのローカル・キーマップ、使用中の全ての副モードのキーマップです。

『グローバル・キーマップ (global keymap)』には、 `C-f` のような、カレント・バッファに関係なしに定義されるキーのバインドが入っています。変数 `global-map` にこのキーマップが納められています。このキーマップは常にアクティブです。

バッファはそれぞれ『ローカル・キーマップ (local keymap)』という、キーの新たな定義や更新を行なう別のマップをもつことができます。カレント・バッファのローカル・キーマップは、 `overriding-local-map` によって無効にされないかぎり常にアクティブです。テキスト属性によって、バッファの特定部分に対して別のローカル・マップを指定することができます。 29.18.4 節「特殊な意味をもつ属性」 p.513 参照。

それぞれの副モードはキーマップをもつことができます。ある副モードが使用されると、そのモードがキーマップを持っていればアクティブになります。

変数 `overriding-local-map` が非 `nil` で、別のローカル・キーマップを指定している場合、バッファのローカル・マップと全ての副モード・キーマップは無効になり、その指定されたキーマップが使われます。

キーが入力されたとき、どのコマンドを実行すればいいかを決定するために、全てのアクティブなキーマップが一緒に使われます。 Emacs は、どれかのマップでバインドが見つかるまで、優先順位にしたがってこれらのマップを順々に探していきます。

通常、 Emacs は最初に副モード・マップ群 (一度には一つのマップ) でキーを探します。そのキーのバインドがその中では見つからないときは、ローカル・マップで探します。そこでも見つからなければ、グローバル・マップで探します。ですが、 `overriding-local-map` が非 `nil` のときは、最初にそのマップで探して、次にグローバル・マップで探します。

単独のキーマップで探す手続きのことを『キー検索 (key lookup)』といいます。 19.7 節「キー検索」p.301 参照。

同じ主モードを使っている全てのバッファは、通常同じローカル・キーマップを使っているので、ローカル・キーマップはモードごとに存在すると考えられます。あるバッファのローカル・キーマップを (たとえば、 `local-set-key` を使って) 変更すると、同じキーマップを共有している別のバッファにも影響します。

Lisp モードや C モードなどいくつかの主モードで使われるローカル・キーマップは、それらのモードが一度も使われていなくても存在します。これらのローカル・マップは変数 `lisp-mode-map` や `c-mode-map` というような変数の値です。そのほかのほとんどのモード、あまり頻繁には使われないモードのローカル・キーマップはそのモードがセッション中ではじめて使われたときにだけ作られます。

ミニバッファにもローカル・キーマップがあり、その中には補完や脱出のコマンドがあります。 17.1 節「ミニバッファの紹介」p.237 を参照してください。

標準キーマップの一覧は、付録 E「標準のキーマップ」p.667 を参照してください。

global-map Variable

この変数には Emacs のキーボード入力とコマンドを対応させるデフォルトのグローバル・キーマップが入っています。通常、グローバル・キーマップはこのキーマップです。デフォルトのグローバル・キーマップは、全ての印字可能文字を `self-insert-command` にバインドする、密なキーマップです。

グローバル・マップのバインドを変更することはよく行なわれますが、この変数を起動時のキーマップ以外の値に割り当ててはいけません。

current-global-map Function

この関数は現在のグローバル・キーマップを返します。グローバル・マップの値を変更していないかぎり `global-map` の値と同じです。

```
(current-global-map)
⇒ (keymap [set-mark-command beginning-of-line ...
            delete-backward-char])
```

current-local-map Function

この関数はカレント・バッファのローカル・キーマップか、それが存在しない場合は `nil` を返します。以下に示す例は、‘`*scratch*`’ バッファ (Lisp Interaction モードを使用しています) のキーマップで、疎なキーマップになっています。 ESC、すなわち ASCII コード 27 のエントリーが別の疎なキーマップになっています。

```
(current-local-map)
⇒ (keymap
    (10 . eval-print-last-sexp)
    (9 . lisp-indent-line)
    (127 . backward-delete-char-untabify)
    (27 keymap
        (24 . eval-defun)
        (17 . indent-sexp)))
```

current-minor-mode-maps Function

この関数は現在使用中の副モードのキーマップのリストを返します。

use-global-map *keymap* Function

この関数は *keymap* を新しい現在のグローバル・キーマップにします。 `nil` を返します。

グローバル・キーマップを変更するのは非常にまれなことです。

use-local-map *keymap* Function

この関数は *keymap* をカレント・バッファの新しいローカル・キーマップにします。*keymap* が `nil` なら、バッファはローカル・キーマップを持ちません。 `use-local-map` は `nil` を返します。ほとんどの主モード・コマンドはこの関数を使用しています。

minor-mode-map-alist Variable

この変数は、ある変数の値にしたがってアクティブになったりならなかったりするキーマップを記述する alist です。この変数の要素はこのようになっています:

```
(variable . keymap)
```

キーマップ *keymap* は *variable* が非 `nil` の値を持っているときにアクティブになります。一般的に、 *variable* は副モードの使用、不使用を切替える変数です。 20.2.2 節「キーマップと副モード」p.328 を参照してください。

`minor-mode-map-alist` の要素は `minor-mode-alist` の要素と同じ構造ではないことに注意してください。マップは要素の CDR でなくてはなりません。マップのリストが 2 番目の要素になっているのでは駄目です。

さらに、キーマップ自身が CDR になっていなくてはなりません。マップが入っている変数が CDR になっていても駄目です。

複数の副モード・キーマップがアクティブになっているとき、それらの優先順位は `minor-mode-map-alist` の順番です。ですが、お互いの邪魔をしないように副モードを設計しなくてはなりません。これをちゃんと守れば、順番は問題になりません。

下記の `minor-mode-key-binding` も参照してください。副モードについての詳細は 20.2.2 節「キーマップと副モード」p.328 を参照してください。

overriding-local-map Variable

非 `nil` の場合、この変数にはバッファのローカル・キーマップと全ての副モード・キーマップの代わりに使われるキーマップが入っています。このキーマップは、もし存在すれば、現在のグローバル・マップ以外の全てのアクティブなマップを無効にします。

overriding-local-map-menu-flag Variable

この変数が非 `nil` のとき、 `overriding-local-map` の値はメニュー・バーの表示に影響します。デフォルト値は `nil` ですので、 `overriding-local-map` はメニュー・バーには何の影響も与えません。

`overriding-local-map` は、メニュー・バー表示に影響を与えないときでも、メニュー・バーを使って入力されたキー・シーケンスの実行には影響を及ぼすことに注

意してください。ですから、メニュー・バー・キー・シーケンスが入力されるときには、そのキー・シーケンスが検索されて実行される前に `overriding-local-map` をクリアしなければなりません。一般的に、 `overriding-local-map` を使うモードは何らかの方法でこれを行ないます。普通は扱わないイベントを受けとったときに "読み戻して" 抜けることによって行ないます。

19.7 キー検索

『キー検索 (key lookup)』とは与えられたキーマップからキー・シーケンスのバインドを探す処理のことをいいます。バインドの実際の実行はキー検索には含まれません。[2]

キー検索ではキー・シーケンス中のイベントそれぞれのイベント・タイプのみを使い、イベントの残りの部分は無視します。実際、キー検索に使うキー・シーケンスでは、マウス・イベント全体 (リスト) の代わりにそれらのタイプ (シンボル) だけで、マウス・イベントを指示しても構いません。 18.5 節「入力イベント」 p.266 を参照してください。このような疑似キー・シーケンスは `command-execute` には不十分ですが、キーを検索したりバインドし直すには十分です。

キー・シーケンスが複数のイベントから成り立っている場合は、キー検索はイベントを順々に処理していきます。まず、最初のイベントのバインドを探します。それはキーマップのはずです。次に 2 番目のイベントのバインドをそのキーマップの中から探します。これをキー・シーケンスの全てのイベントを使い切るまで繰り返します (このように探し出された最後のイベントのバインドは、キーマップのこともありますし、そうでないこともあります)。このようにキー検索の処理は、あるキーマップから一つのイベントを捜し出す単純な処理と定義できます。どのようにそれを行なうかは、キーマップでイベントと結びつけられているオブジェクトのタイプによります。

キーマップでイベント・タイプに結びつけられている値を『キーマップ・エントリー (keymap entry)』という用語で呼ぶことにしましょう (これにはメニュー・キー・バインドの項目文字列やほかの特別な要素は含まれません。 `lookup-key` やほかのキー検索の関数の返り値にはそれらは含まれないからです)。どんな Lisp オブジェクトがキーマップにキーマップ・エントリーとして納められていても構いませんが、全てが意味をなすわけではありません。有効なキーマップ・エントリーの種類の一覧を示します:

`nil`
: `nil` は、検索中にここまで使われたイベント群は未定義のキーになることを意味します。あるキーマップ中であるイベント・タイプを探すのに失敗し、さらにデフォルトのバインドがなかった場合は、そのイベント・タイプは `nil` にバインドされているのと等価です。

keymap
: 検索中にここまで使われたイベント群は前置キーになります。キー・シーケンスの次のイベントは *keymap* 中で探します。

command
: 検索中にここまで使われたイベント群は完結キーになり、 *command* がそのバインドです。 11.1 節「関数とは」 p.147 を参照してください。

[2] 訳注: この章では lookup に「検索」という訳をあてたため、ほかの章で「検索」と訳している search は「探す」という訳にしてあります。

array 配列 (文字列またはベクタのどちらでも) はキーボード・マクロです。検索中にここまで使われたイベント群は完結キーになり、この配列がそのバインドです。詳しくは18.13 節「キーボード・マクロ」p.291 参照。

list リストの意味はリストの要素のタイプによります。

- *list* の CAR がシンボル `keymap` ならば、そのリストはキーマップで、キーマップとして扱われます (上記参照)。
- *list* の CAR が `lambda` ならば、そのリストはラムダ式です。これはコマンドと推定され、そのように扱われます (上記参照)。
- *list* の CAR がキーマップで CDR がイベント・タイプなら、これは『間接エントリー (indirect entry)』です:

  (*othermap* . *othertype*)

  キー検索中に間接エントリーに出くわしたときは、代わりに *othermap* 中の *othertype* のバインドを探して、それを使います。

  この機能を使うとあるキーを別のキーの別名として定義することができます。たとえば、 CAR が `esc-map` というキーマップで CDR が 32(`SPC` のコード) であるようなエントリーは、“`Meta-SPC` のグローバル・バインドを使え、それが何であっても” ということになります。

symbol *symbol* の代わりに *symbol* の関数定義が使われます。それもシンボルだったら、この処理が何度でも繰り返されます。最終的には、キーマップかコマンドかキーボード・マクロかのいずれかのオブジェクトが出てくるはずです。リストは、それがキーマップかコマンドならば許されますが、間接エントリーはシンボル経由で出てきた場合は認識されません。

注意して欲しいのは、キーマップやキーボード・マクロ (文字列とベクタ) は正しい関数ではなく、同様にその関数定義がキーマップ、文字列、ベクタであるようなシンボルは関数としては正しくないことです。しかしながら、キー・バインドとしては正当です。関数定義がキーボード・マクロであるようなシンボルは、`command-execute` の引数としても正しいものです。

シンボル `undefined` は、特別に言及する価値があります。このシンボルはキーが未定義扱いになることを意味します。正確にいえば、キーは定義されていて、そのバインドはコマンド `undefined` です。しかし、このコマンドは未定義キーが自動的に行なうのと同じことを行ないます。 (`ding` を呼び出すことによって) ベルを鳴らしますが、エラーのシグナルは発生しません。

`undefined` はローカル・キーマップで、グローバル・キー・バインドを無効にして、あるキーをローカルに “未定義” にするために使われます。ローカルで `nil` にバインドしても、グローバル・バインドを無効にできないのでうまくいきません。

anything else

これら以外のタイプのオブジェクトが出てきたら、ここまでの検索に使われたイベント群は完結キーになり、そのバインドはそのオブジェクトになりますが、それはコマンドとしては実行できないバインドです。

要するに、キーマップ・エントリーは、キーマップ、コマンド、キーボード・マクロ、またはそれらのどれかを導くシンボル、もしくは間接エントリーか nil になります。次の例は、コマンドにバインドされた二つの文字と別のキーマップにバインドされた一つの文字をもつ疎なキーマップです。このマップは emacs-lisp-mode-map の通常の値です。 9 は TAB、 127 は DEL、 27 は ESC、 17 は C-q、 24 は C-x のコードであることを思い出してください。

```
(keymap (9 . lisp-indent-line)
        (127 . backward-delete-char-untabify)
        (27 keymap (17 . indent-sexp) (24 . eval-defun)))
```

19.8 キー検索のための関数

これらはキー検索関連の関数と変数です。

lookup-key *keymap key* &optional *accept-defaults* Function

この関数は *keymap* での *key* の定義を返します。文字列またはベクタである *key* が、*keymap* で指定された前置キーにしたがっても正しいキー・シーケンスにならないときは (つまり、“長過ぎ” て余分なイベントが最後にある)、返り値は *key* の先頭から完結キーをなすイベントの数になります。

accept-defaults が非 nil ならば、 lookup-key は *key* で指定されたイベントのバインドと同様にデフォルト・バインドも考慮します。そうでなければ、 lookup-key は指定されたシーケンス *key* のバインドしか報告せず、デフォルト・バインドは明示的にそれを聞かれた場合 (*key* の要素として t を与えます。 19.2 節「キーマップのフォーマット」 p.294 参照) を除いて無視します。

この章で述べるキーを検索する関数は全て lookup-key を使っています。

```
(lookup-key (current-global-map) "\C-x\C-f")
    ⇒ find-file
(lookup-key (current-global-map) "\C-x\C-f12345")
    ⇒ 2
```

key にメタ文字が含まれているときは、その文字は暗に 2 文字のシーケンス、 meta-prefix-char の値に続けて対応する非メタ文字、に置き換えられます。したがって、下の 1 番目の例は、 2 番目の例に変換されて扱われます。

```
(lookup-key (current-global-map) "\M-f")
    ⇒ forward-word
(lookup-key (current-global-map) "\ef")
    ⇒ forward-word
```

read-key-sequence とは違って、この関数は情報を捨てるようなイベントの修正は行ないません (18.6.1 節「キー・シーケンス入力」 p.279 参照)。特に、英文字を小文字に変換したり、ドラッグ・イベントをクリックに変えたりしません。

undefined Command

キーマップ中でキーを未定義にするのに使われます。 ding を呼び出しますが、エラーは起きません。

key-binding *key* &optional *accept-defaults* Function

この関数は、全てのアクティブなキーマップを調べて、現在のキーマップでの *key* のバインドを返します。 *key* がそれらのキーマップで未定義の場合は、結果は `nil` になります。

引数 *accept-defaults* は、 `lookup-key`(上記) と同様に、デフォルト・バインドを調べるかどうかを制御します。

key が文字列やベクタでない場合は、エラーになります。

```
(key-binding "\C-x\C-f")
     ⇒ find-file
```

local-key-binding *key* &optional *accept-defaults* Function

この関数は現在のローカル・キーマップでの *key* のバインドを返します。そこでは未定義なら `nil` を返します。

引数 *accept-defaults* は、 `lookup-key`(上記) と同様に、デフォルト・バインドを調べるかどうかを制御します。

global-key-binding *key* &optional *accept-defaults* Function

この関数は現在のグローバル・キーマップでの *key* のバインドを返します。そこでは未定義なら、 `nil` を返します。

引数 *accept-defaults* は、 `lookup-key`(上記) と同様に、デフォルト・バインドを調べるかどうかを制御します。

minor-mode-key-binding *key* &optional *accept-defaults* Function

この関数は、 *key* の全てのアクティブな副モード・バインドのリストを返します。もっと詳しくいうと、次のような対の alist (*modename* . *binding*) を返します。 *modename* はその副モードを使用中にする変数で、 *binding* は *key* のそのモードにおけるバインドです。

最初のバインドが前置コマンドでなければ、そのあとのほかの副モードの全てのバインドは、完全に無効になるので無視されます。同様に、前置バインドのあとの非前置バインドはリスト上では省略されます。

引数 *accept-defaults* は、 `lookup-key`(上記) と同様に、デフォルト・バインドを調べるかどうかを制御します。

meta-prefix-char Variable

この変数はメタ接頭辞の文字コードを示します。メタ文字をキーマップで検索できるように 2 文字のキー・シーケンスに変換するときに使われます。値は前置イベントでなくてはうまくいきません (19.5 節「前置キー」 p.297 参照)。デフォルト値は 27 で、これは `ESC` の ASCII コードです。

`meta-prefix-char` の値が 27 である間は、キー検索時に `M-b` は `ESC b` に変換されます。 `ESC b` は通常は `backward-word` コマンドとして定義されています。しかし、 `meta-prefix-char` を 24(`C-x` のコードです) にすると、 Emacs は `M-b` を `C-x b` に変換します。これの標準のバインドは `switch-to-buffer` コマンドです。

```
meta-prefix-char                    ; デフォルト値
     ⇒ 27
(key-binding "\M-b")
     ⇒ backward-word
?\C-x                               ; 文字の印字表現
     ⇒ 24
(setq meta-prefix-char 24)
     ⇒ 24
(key-binding "\M-b")
     ⇒ switch-to-buffer             ; 今は M-b は C-x b
                                    ;    とタイプするのと同じになる.

(setq meta-prefix-char 27)          ; 混乱を避けるため
     ⇒ 27                           ; デフォルト値に戻す!
```

19.9 キー・バインドの変更

キーをバインドし直す方法は、キーマップ中のエントリを変更することです。グローバル・キーマップのバインドを変更すれば、全てのバッファに影響がおよびます (ですが、ローカル・バインドによってグローバル・バインドが隠されているバッファには直接の影響はありません)。カレント・バッファのローカル・マップを変更すれば、通常は同じ主モードを使っている全てのバッファに影響します。 `global-set-key` と `local-set-key` 関数はこれらの操作を行なうための便利なインターフェースです (19.10 節「キー・バインド用コマンド」 p.308 参照)。もっと一般的な関数として `define-key` も使えます。このときは、変更するマップを明示的に指定しなくてはいけません。

キー・シーケンスのバインドし直しを記述するときに、制御文字やメタ文字のための特殊なエスケープ・シーケンスが便利です (2.3.8 節「文字列型」 p.28 参照)。構文 '`\C-`' はすぐあとの文字が制御文字であることを、 '`\M-`' はすぐあとの文字がメタ文字であること意味します。したがって、文字列 `"\M-x"`は単独の `M-x` として、 `"\C-f"`は単独の `C-f` として、 `"\M-\C-x"`と`"\C-\M-x"`は両方とも単独の `C-M-x` として読み込まれます。このエスケープ構文は、文字列中では許されていないほかのものと同様にベクタ中で使うこともできます。例、 '`[?\C-\H-x home]`'。 2.3.3 節「文字型」 p.21 を参照してください。

キー定義と検索の関数ではキー・シーケンスがベクタの場合、イベント・タイプの別な構文としていつくかの修飾子名と一つの基本イベント (文字またはファンクション・キー名) を含むリストを使うことができます。たとえば、 `(control ?a)` は`?\C-a` と等価で、 `(hyper control left)` は `C-H-left` と等価です。

イベント・タイプの表現としてリストを使う利点の一つに、コンパイル後のファイルに修飾ビットの数値コードがあからさまに出てこないということがあげられます。

以下の関数では、 *keymap* がキーマップでない場合や、 *key* がキー・シーケンスを表す文字列かベクタでない場合にエラーが発生します。リストであるようなイベントの代わりにそのイベントタイプ (つまりシンボル) で済ますこともできます。

define-key *keymap key binding* Function

この関数は *keymap* に *key* のバインドを設定します (*key* が複数のイベントのときは、変更は実際には *keymap* から到達する別のキーマップで行なわれます)。引数 *binding* はどんな Lisp オブジェクトでも構いませんが、ある限られたタイプのものだけが有効です (有効なタイプの一覧は 19.7 節「キー検索」p.301 参照)。 `define-key` が返す値は *binding* です。

key の全ての接頭辞は前置キー (すなわちキーマップにバインドされている) か、未定義でなくてはなりません。そうでなければエラーが発生します。

key のある接頭辞が未定義なら、 `define-key` は *key* の残りの部分が指定されたように定義できるように、それを前置キーとして定義します。

以下は、疎なキーマップを作りいくつかのバインドを設定する例です:

```
(setq map (make-sparse-keymap))
    ⇒ (keymap)
(define-key map "\C-f" 'forward-char)
    ⇒ forward-char
map
    ⇒ (keymap (6 . forward-char))
;; C-x のための疎なサブマップを作り f のバインドを設定する。
(define-key map "\C-xf" 'forward-word)
    ⇒ forward-word
map
⇒ (keymap
    (24 keymap                ; C-x
        (102 . forward-word)) ;      f
    (6 . forward-char))       ; C-f
;; C-p を ctl-x-map にバインドする。
(define-key map "\C-p" ctl-x-map)
;; ctl-x-map
⇒ [nil ... find-file ... backward-kill-sentence]
;; ctl-x-map の中で C-f を foo にバインドする。
(define-key map "\C-p\C-f" 'foo)
⇒ 'foo
map
⇒ (keymap     ; ctl-x-map の中に foo があることに注目。
    (16 keymap [nil ... foo ... backward-kill-sentence])
    (24 keymap
        (102 . forward-word))
    (6 . forward-char))
```

注意して欲しいのは、 `C-p C-f` に新しいバインドを設定することは実際には `ctl-x-map` のエントリーを変更すること、そしてこれは `C-p C-f` とデフォルトのグローバル・マップの `C-x C-f` の両方のバインドを変更することになることです。

substitute-key-definition *olddef newdef keymap* &optional *oldmap* Function

この関数は、 *keymap* の中で *olddef* にバインドされている全てのキーの *olddef* を *newdef* で置き換えます。言い替えると、 *olddef* はそれがどこにあろうとも *newdef* で置き換えられます。この関数は `nil` を返します。

たとえば、この例は標準のバインドの Emacs では `C-x C-f` を再定義します:

```
(substitute-key-definition
 'find-file 'find-file-read-only (current-global-map))
```

oldmap が非 `nil` なら、そのバインドによってどのキーをバインドし直すかが決まります。バインドし直されるのは *newmap* であって、 *oldmap* ではありません。したがって、あるマップを別のマップのバインドの制御のもとで変更することができます。たとえば、

```
(substitute-key-definition
  'delete-backward-char 'my-funny-delete
  my-map global-map)
```

グローバル・マップで標準の削除コマンドにバインドされているキーに、 `my-map` で特別な削除コマンドをバインドしています。

この例ではあるキーマップの置換前と置換後を示します:

```
(setq map '(keymap
            (?1 . olddef-1)
            (?2 . olddef-2)
            (?3 . olddef-1)))
⇒ (keymap (49 . olddef-1) (50 . olddef-2) (51 . olddef-1))

(substitute-key-definition 'olddef-1 'newdef map)
⇒ nil
map
⇒ (keymap (49 . newdef) (50 . olddef-2) (51 . newdef))
```

suppress-keymap *keymap* &optional *nodigits* Function

この関数は、密なキーマップ *keymap* を全ての印字可能文字が未定義になるように変更します。もっと正確にいうと、それらを `undefined` にバインドします。通常、これはテキストの挿入を不可能にします。 `suppress-keymap` は `nil` を返します。[3]

nodigits が `nil` なら、 `suppress-keymap` は数字を `digit-argument` を実行するように、 `-` を `negative-argument` を実行するように定義します。そうでない場合は、それらをほかの印字可能文字と同じように未定義にします。

`suppress-keymap` 関数は、 `yank` や `quoted-insert` のようなコマンドを抑制しませんので、バッファの変更を不可能にするわけではありません。バッファの変更を一切できなくするには、バッファを読出し専用にしてください (24.7 節「読みだし専用バッファ」 p.401 参照)。

この関数は *keymap* を変更するので、普通は新しく作ったキーマップに対して使用します。別の目的で使われている既存のキーマップに対して使用した場合は、たいていトラブルが起きるでしょう。たとえば、 `global-map` を抑制したなら、 Emacs はほとんど使えなくなるでしょう。

ほとんどの場合、 `suppress-keymap` は Rmail や Dired のようなテキストの挿入が必要でなく読出し専用のモードのローカル・キーマップの初期化に使われます。これは '`emacs/lisp/dired.el`' から持ってきた Dired モードのローカル・キーマップの設定を示す例です:

---

[3] 訳注: これを「キーマップの抑制」といいます。

```
...
(setq dired-mode-map (make-keymap))
(suppress-keymap dired-mode-map)
(define-key dired-mode-map "r" 'dired-rename-file)
(define-key dired-mode-map "\C-d" 'dired-flag-file-deleted)
(define-key dired-mode-map "d" 'dired-flag-file-deleted)
(define-key dired-mode-map "v" 'dired-view-file)
(define-key dired-mode-map "e" 'dired-find-file)
(define-key dired-mode-map "f" 'dired-find-file)
...
```

19.10 キー・バインド用コマンド

この節ではキー・バインドを変更するのに便利な対話的インターフェースについて述べます。それらは`define-key`を呼び出すことによって動作します。

`global-set-key`は‘`.emacs`’の中で簡単なカスタマイズのためによく使われます。たとえば、

```
(global-set-key "\C-x\C-\\" 'next-line)
```

や

```
(global-set-key [?\C-x ?\C-\\] 'next-line)
```

や

```
(global-set-key [(control ?x) (control ?\\)] 'next-line)
```

は、`C-x C-\`を一行下へ動くように再定義しています。

```
(global-set-key [M-mouse-1] 'mouse-set-point)
```

これはメタ・キーを押しながら第1(左端)マウス・ボタンを押されたときに、クリックした場所へポイントが移動するように再定義しています。

global-set-key *key definition* Command

この関数は現在のグローバル・マップの *key* のバインドに *definition* をセットします。

```
(global-set-key key definition)
≡
(define-key (current-global-map) key definition)
```

global-unset-key *key* Command

この関数は現在のグローバル・マップから *key* のバインドを取り除きます。

この関数の使い道の一つに、*key* を前置キーとして使うより長いキーを定義するための準備というのがあります。*key* が前置キーでないバインドを持っている場合は、そのままではできません。例:

```
(global-unset-key "\C-l")
    ⇒ nil
(global-set-key "\C-l\C-l" 'redraw-display)
    ⇒ nil
```

この関数は`define-key`を使って簡単に実装されています:

```
(global-unset-key key)
≡
(define-key (current-global-map) key nil)
```

local-set-key *key definition* Command

この関数は現在のローカル・キーマップの *key* のバインドに *definition* をセットします。

```
(local-set-key key definition)
≡
(define-key (current-local-map) key definition)
```

local-unset-key *key* Command

この関数は現在のローカル・マップから *key* のバインドを取り除きます。

```
(local-unset-key key)
≡
(define-key (current-local-map) key nil)
```

19.11 キーマップのスキャン

この節ではヘルプ情報を表示するために全ての現在のキーマップをスキャンするのに使われる関数について述べます。

accessible-keymaps *keymap* &optional *prefix* Function

この関数は、 *keymap* から (前置キー経由で) アクセスできる全てのキーマップのリストを返します。値は (*key* . *map*) という形式の要素の連想リストで、 *key* はその *keymap* 中の定義が *map* であるような前置キーです。

alist の要素は *key* の長さが長くなるような順番になっています。指定されたキーマップはそれ自身から空の前置キーによってアクセス可能なので、最初の要素は常に ("" . *keymap*) です。

prefix が与えられた場合、それは前置キー・シーケンスでなければなりません、accessible-keymaps はその前置キーが *prefix* ではじまるサブマップだけを返します。これらの要素は (accessible-keymaps) の値とほとんど同じで、違うのはいくつかの要素が省かれていることだけです。

下の例では、返された alist は、 '^[' として表示されているキー ESC が前置キーで、その定義が疎なキーマップ (keymap (83 . center-paragraph) (115 . foo)) であることを示しています。

```
(accessible-keymaps (current-local-map))
⇒(("" keymap
      (27 keymap   ; この ESC のためのキーマップは以下にも現れる。
          (83 . center-paragraph)
          (115 . center-line))
      (9 . tab-to-tab-stop))

   ("^[" keymap
    (83 . center-paragraph)
    (115 . foo)))
```

続く例では、 C-h は (keymap (118 . describe-variable)...) ではじまる疎なキーマップを使う前置キーです。別の前置キー C-x 4 は変数 ctl-x-4-map の値でもあるキーマップを使います。イベント mode-line は、ウィンドウの特別な部分でのマウスの動作のための前置キーとして使われる、いつくかあるダミー・イベントのうちの一つです。

```
(accessible-keymaps (current-global-map))
⇒ ((""  keymap [set-mark-command beginning-of-line ...
                  delete-backward-char])
    ("^H" keymap (118 . describe-variable) ...
     (8 . help-for-help))
    ("^X" keymap [x-flush-mouse-queue ...
     backward-kill-sentence])
    ("^[" keymap [mark-sexp backward-sexp ...
     backward-kill-word])
    ("^X4" keymap (15 . display-buffer) ...)
    ([mode-line] keymap
     (S-mouse-2 . mouse-split-window-horizontally) ...))
```

実際のケースで現れるキーマップは、これらだけではありません。

where-is-internal *command* &optional *keymap firstonly noindirect* Function

この関数は、あるキーマップの集合の中で *command* にバインドされている (いろんな長さの) キー・シーケンスのリストを返します。

引数 *command* はどんなオブジェクトでも構いません。これを、全てのキーマップ・エントリーと `eq` を使って比較します。

keymap が `nil` ならば、 `overriding-local-map` を無視して (値が `nil` であるとみなして) 現在のアクティブ・キーマップを使います。 *keymap* が非 `nil` なら、探すマップは *keymap* とグローバル・キーマップになります。

普通は、 `overriding-local-map` を *keymap* として使うのがよいでしょう。そうすれば、 `where-is-internal` はアクティブなキーマップを正確に探します。グローバル・マップだけを探したいときは、 *keymap* として `(keymap)`(空のキーマップ) を渡せばよいでしょう。

firstonly が `non-ascii` のときは、全てのキー・シーケンスのリストの代わりに、最初に見つけたキー・シーケンスを表す一つの文字列が返す値になります。 *firstonly* が `t` ならば、返す値は最初のキー・シーケンスになりますが、全てが ASCII 文字 (または ASCII のメタ文字) からなるキー・シーケンスはそれ以外のキー・シーケンスより優先されます。

nonindirect が非 `nil` なら、 `where-is-internal` は間接キーマップ・バインドをたどりません。このことにより、間接定義自身を探すことができます。

この関数は `where-is` で使われています (*The GNU Emacs Manual* の “Help” 節参照)。

```
(where-is-internal 'describe-function)
    ⇒ ("\^hf" "\^hd")
```

describe-bindings *prefix* Command

この関数は定義済みの全てのキーとそれらの定義の一覧を作成します。 ‘`*Help*`’ という名前のバッファに一覧を書き込みそれをどれかのウインドウに表示します。

prefix が非 `nil` なら、それは前置キーでなくてはなりません、一覧には *prefix* ではじまるキーだけが含まれます。

一覧では、メタ文字は `ESC` の後に対応する非メタ文字が続く形式で表されます。

いくつかの連続する ASCII コードの文字が同じ定義のときは、'*firstchar*..*lastchar*' というようにまとめて表示します。この場合、 ASCII コードを知らないと、これがどの文字群を表しているのかわかりません。たとえば、デフォルトのグローバル・マップでは、'`SPC.. ~`' の文字群が 1 行で記述されています。 `SPC` は ASCII の 32、 `~` は ASCII の 126、そしてその間の文字には全ての通常の印字可能文字 (つまり、英文字 (letter) や数字や句読点など) が含まれます。これらの全ての文字は `self-insert-command` にバインドされています。

19.12 メニュー・キーマップ

キーマップは、キーボードのキーやマウス・ボタンのバインドを定義するのと同様に、メニューを定義することもできます。メニューは通常マウスから起動されますが、キーボードで使うこともできます。

19.12.1 メニューの定義

キーマップは『見出し文字列 (overall prompt string)』、これはキーマップの要素として現れる文字列です、が含まれていればメニューのために使えます (19.2 節「キーマップのフォーマット」p.294 を参照してください)。この文字列はそのメニューの目的を表していなくてはなりません。見出し文字列つきのキーマップを作る最も簡単な方法は、 `make-keymap` や `make-sparse-keymap` を呼び出すときに文字列を引数として指定することです (19.3 節「キーマップの作成」p.295 参照)。

メニュー中の項目の順番は、キーマップ中のバインドの順番と同じです。 `define-key` は新しいバインドを先頭に入れるので、順番を気にするならメニューの項目は下のものからはじめて上のものへと定義していかなくてはなりません。すでに存在するメニューに項目を追加するときは、 `define-key-after` を使って位置を指定することができます (19.12.6 節「メニューの変更」p.315 参照)。

メニュー・キーマップの中の個々のバインドには項目文字列が含まれていなくてはいけません。これらの文字列は、メニューに表示される項目になります。項目文字列を含むバインドはこんな感じです:

(*string* . *real-binding*)

バインドの項目文字列は短く (単語一つか二つ) なくてはなりません。また、対応するコマンドの動作を表していなくてはいけません。

`define-key` にとっては、 *string* はイベントのバインドの一部分ですが、 `lookup-key` は *real-binding* のみを返し、また *real-binding* だけがキーを実行するときに使われます。

また、下のようにヘルプ文字列と呼ばれる 2 番目の文字列も与えることができます:

(*string* *help-string* . *real-binding*)

現在の Emacs は *help-string* を実際には使いません。 *real-binding* を取り出すために *help-string* を無視する方法を知っているだけです。将来は *help-strings* を、要求に応じてメニューの項目の拡張ドキュメントとして使えるようになるでしょう。

real-binding が nil なら *string* はメニューに現れますが選択できません。

real-binding がシンボルでその menu-enable 属性が非 nil なら、その属性はそのメニュー項目を使用可にするかどうかを制御する式になります。 Emacs は、メニューを表示するためにキーマップが使われると毎回その式を評価して、式の値が非 nil のときだけそのメニュー項目を使用可にします。メニュー項目が使用不可のときは、“曖昧” な形式で表示されマウスで選択することはできません。

メニュー・バーはあなたがメニューを見るたびにどの項目が使用可なのかを再算出したりはしません。これは X toolkit があらかじめメニューのツリー全体を要求するからです。強制的にメニュー・バーを再算出するには、 force-mode-line-update を呼び出してください (20.3 節「モード行の書式」p.329 参照)。

“同じ” コマンドを実行するけど違う条件で使用可になるメニューの項目を作ることが便利なときがあります。コマンドの別名を定義することによって、これを行なうことができます。次の例は、 toggle-read-only の二つの別名を作り、違う条件で使用可になるようにしています:

```
(defalias 'make-read-only 'toggle-read-only)
(put 'make-read-only 'menu-enable '(not buffer-read-only))
(defalias 'make-writable 'toggle-read-only)
(put 'make-writable 'menu-enable 'buffer-read-only)
```

メニュー項目に同じコマンドを起動するキーボードのキー・シーケンスが (もしあれば) 表示されていることに多分気がついたと思います。再算出の時間を節約するため、メニューを表示するときにこの情報をバインドのサブリストにキャッシュします。このように:

(*string* [*help-string*] (*key-binding-data*) . *real-binding*)

これらのサブリストをメニュー項目に自分で入れてはいけません。メニューを表示すると自動的にこれらを算出します。マウス・メニューの項目文字列に等価なキー・シーケンスをつけ加えないでください、冗長になります。

別名コマンド自身と等価なキー・シーケンスがないときは、本体のコマンドと等価なキー・シーケンスをメニューに表示します。上の例では、 make-read-only や make-writalbe を実行するように定義されたメニュー項目には、 toggle-read-only と等価なキー・シーケンスが表示されます。

19.12.2 メニューとマウス

メニュー・キーマップを使ってメニューを出すためには、それを前置キーの定義にします。

前置キーがマウス・イベントで終っていれば、 Emacs はユーザがマウスを使って選択できるように、メニュー・キーマップを使って可視のメニューをポップアップします。ユーザがあるメニュー項目をクリックすると、発生するイベントは文字であってもシンボルであってもそのメニュー項目がもたらすバインドを持っています (メニューが複数の階層だったりメニュー・バーから呼び出された場合は、メニュー項目はひと続きのイベントを発生するかも知れません)。

たいていの場合、ボタン押下のイベントでメニューを起動するのがいいでしょう。こうすると、ユーザはボタンを離すことによってメニューの項目を選択することができます。

一つのキーマップで複数のメニューの枠 (pane) を出すことが、明示的に指示すればできます。そのやり方は、それぞれの枠のキーマップを作り、メニューのメインのキーマップにそれらのマップそれぞれのバインドを作ります。これらのバインドそれぞれに ‘@’ ではじまる項目文字列を与えてください。項目文字列の残りの部分は枠の名前になります。‘lisp/mouse.el’ を見れば、これの例があります。‘@’ 抜きの項目文字列をもつ普通のバインドは、明示的に作られたほかのサブマップの枠と一緒に現れる別の一つの枠にまとめられます。

X toolkit のメニューは枠を複数出すことはできません。代わりにサブメニューを使うことができます。ネストされたキーマップは、項目文字列が ‘@’ ではじまっているかどうかにかかわらず全てサブメニューになります。toolkit 版の Emacs では、項目文字列の先頭にある ‘@’ に関して特別なことは、‘@’ はメニューの項目には表示されないということだけです。

別々のキーマップも複数の枠を出すことができます。前置キーの完全な定義は常にいくつかのアクティブ・キーマップ (副モード、ローカル、グローバル) から得られる定義を合成したものになります。これらのキーマップのうち複数がメニューのときは、それぞれのメニューは別の枠を出します。19.6 節「アクティブ・キーマップ」p.298 を参照してください。

toolkit 版の Emacs では、メニューは枠を複数出すことができないため、別のキーマップはサブメニューとして表現されます。それぞれのキーマップがサブメニューになります。

Lisp プログラムから明示的にメニューをポップアップしてユーザの選択を受けとることができます。このときにもキーマップが使えます。26.15 節「ポップアップ・メニュー」p.450 を参照してください。

19.12.3 メニューとキーボード

最後がキーボード・イベント (文字とかファンクション・キー) の前置キーの定義がメニュー・キーマップのときは、ユーザはキーボードでメニューの項目を選択することができます。

Emacs はメニューの選択肢 (バインドの項目文字列) をエコー領域に表示します。全部が一度に入り切らないときは、ユーザは `SPC` をタイプして次の行の選択肢を見ることができます。`SPC` を何度もタイプしてメニューの最後まできたら、最初に戻ります (変数 `menu-prompt-more-char` はこれに使う文字を指定します。`SPC` がデフォルトです)。

ユーザは望む選択肢を見つけたら、対応する文字 (バインドはその選択肢です) をタイプします。

Emacs ライクなエディターのこのようなメニューの使い方は、Hierarkey システムの影響を受けています。

menu-prompt-more-char Variable

この変数は、メニューの次の行を見るのに使う文字を指定します。初期値は 32、`SPC` のコードです。

19.12.4 メニューの例

これはマウス用メニューの設定方法の簡単な例です。

```
(defvar my-menu-map
  (make-sparse-keymap "Key Commands <==> Functions"))
(fset 'help-for-keys my-menu-map)

(define-key my-menu-map [bindings]
  '("List all keystroke commands" . describe-bindings))
(define-key my-menu-map [key]
  '("Describe key briefly" . describe-key-briefly))
(define-key my-menu-map [key-verbose]
  '("Describe key verbose" . describe-key))
(define-key my-menu-map [function]
  '("Describe Lisp function" . describe-function))
(define-key my-menu-map [where-is]
  '("Where is this command" . where-is))

(define-key global-map [C-S-down-mouse-1] 'help-for-keys)
```

メニューの中でバインドされているキー・シーケンスの中のシンボルは、架空の "ファンクション・キー" です。それらはキーボード上には現れませんが、そのことはメニューの中でそれらを使うことを妨げるわけではありません。それらの名前は、覚えやすいように選ばれています。 `where-is` や `apropos` の出力の中に対応するメニューの項目を示すために現れるからです。

しかしながら、キーボードからも使えるようなメニューにしたければ、架空のファンクション・キーと同じように実在の ASCII 文字にバインドしておかなくてはなりません。

19.12.5 メニュー・バー

ほとんどのウインドウ・システムではフレームはそれぞれ『メニュー・バー (menu bar)』をもつことができます。メニュー・バーとは、フレームの上部に横いっぱいに広がっていて、常時メニューを表示しています。メニュー・バーの項目は偽 "ファンクション・キー"`menu-bar` のサブコマンドで、全てのアクティブ・キーマップで定義されています。

メニュー・バーに項目を追加するためには、まず自分の偽 "ファンクション・キー" (*key* と呼びましょう) を考えましょう。そして、キー・シーケンス `[menu-bar` *key*`]` のバインドを作ってください。たいていの場合は、そのメニュー・バーの項目の上でボタンを押したときに別のメニューが出てくるようにするので、そのバインドはメニュー・キーマップになります。

複数のアクティブ・キーマップでメニュー・バーに同じ偽ファンクション・キーが定義されていても、その項目は一つしか現れません。そのメニュー・バーの項目をクリックすると、その項目の全てのサブコマンド (グローバル・サブコマンド、ローカル・サブコマンド、副モード・サブコマンド、全て一緒) を含む、一つに合成されたサブメニューが出てきます。

メニュー・バーの中身を決定する際には、変数 `overriding-local-map` は通常無視されます。すなわち、メニュー・バーは `overriding-local-map` が `nil` のときにアクティブになるキーマップから算出されます。 19.6 節「アクティブ・キーマップ」 p.298 を参照してください。

フレームにメニュー・バーを表示するためには、そのフレームの menu-bar-lines パラメータが 0 より大きくなくてはいけません。 Emacs は自分のメニュー・バーのために 1 行だけ使うので、複数行を指定した場合は、残りの行はフレーム中のメニュー・バーとウインドウの間に入ります。 menu-bar-lines の値は 1 か 2 をお奨めします。 26.3.3 節「X ウインドウ・フレーム・パラメータ」p.438 を参照してください。

これはメニュー・バーの項目を設定する例です:

```
(modify-frame-parameters (selected-frame)
                         '((menu-bar-lines . 2)))

;; メニュー・キーマップ (見出し文字列つき) を作り、
;; それをメニュー・バー項目の定義とする。
(define-key global-map [menu-bar words]
  (cons "Words" (make-sparse-keymap "Words")))

;; その項目のメニューのサブコマンドを定義する。
(define-key global-map
  [menu-bar words forward]
  '("Forward word" . forward-word))
(define-key global-map
  [menu-bar words backward]
  '("Backward word" . backward-word))
```

グローバル・キーマップによって作られたメニュー・バーの項目を、同じ偽ファンクション・キーをローカル・キーマップで undefined にバインドし直すことによって、消すことができます。たとえば、これは Dired が 'Edit' メニュー・バー項目を抑制するやり方です:

```
(define-key dired-mode-map [menu-bar edit] 'undefined)
```

edit はグローバル・マップで 'Edit' メニュー・バー項目に使っている偽ファンクション・キーです。グローバルのメニュー・バー項目を抑制する主な理由は、モード固有の項目のために場所を空けることです。

menu-bar-final-items Variable

通常、メニュー・バーにはグローバルの項目の次にローカル・マップで定義された項目が表示されます。

この変数には、通常の順番にかかわらずメニュー・バーの最後に表示する項目の偽ファンクション・キーのリストが入っています。デフォルト値は (help) です。したがって、普通は 'Help' メニュー項目がローカル・メニュー項目の次にメニュー・バーの最後に現れます。

19.12.6 メニューの変更

存在するメニューに新しい項目を挿入するときは、多分そのメニューの項目の中のある特定の場所に挿入したいでしょう。項目を追加するのに define-key を使うと、普通はメニューの先頭に置かれます。ほかの場所に置くためには、 define-key-after を使ってください:

define-key-after *map key binding after* Function

map 中の *key* のバインドを値 *binding* に定義します。 `define-key` とほとんど同じですが、バインドの *map* 中の位置がイベント *after* のバインドの後ろになります。引数 *key* は長さが 1、要素が一つのベクタまたは文字列、でなくてはなりません。

たとえば、

```
(define-key-after my-menu [drink]
                  '("Drink" . drink-command) 'eat)
```

偽ファンクション・キー `DRINK` のバインドを作り、それを `EAT` のバインドのすぐ後ろに置きます。

これは、 Shell モードの '`Signals`' メニューに '`Work`' という項目を項目 `break` の後ろに挿入するやり方です:

```
(define-key-after
  (lookup-key shell-mode-map [menu-bar signals])
  [work] '("Work" . work-command) 'break)
```

key は、ただ一つのイベント・タイプを含むシーケンスであるのに対し、 *after* はただのイベント・タイプである (シーケンスではない) ことに注意してください。

20 主モード、副モード

『モード (mode)』とは、 Emacs をカスタマイズするための定義のセットのことであり、また、編集中に自由に使用 / 不使用を決めることができるものです。モードは 2 種類に大別でき、それぞれ、『主モード (major mode)』と『副モード (minor mode)』と呼ばれています。主モードとは、ほかの主モードとは同時に使用できず、また、特定の種類のテキストの編集に適した機能を提供するものです。また、副モードとは、ユーザが好みに応じて自由に組み合わせて使用できるものです。

この章では、主モードと副モードの作り方、モード行での表示の仕方、ユーザによって指定されたフックの実行の仕方について説明します。キーマップ、構文テーブルといった関連事項についてはそれぞれ第 19 章「キーマップ」p.293 と第 31 章「構文テーブル」p.541 を参照してください。

20.1 主モード

主モードは、編集中のテキストの種類に合わせて Emacs を変身させます。一つのバッファで同時に複数の主モードを利用することはできません。

最も普遍的な主モードには『Fundamental モード (Fundamental mode)』という名前がつけられています。このモードにはモード固有の定義や変数の設定がないので、 Emacs のコマンドはデフォルトの動作をし、すべてのオプションはデフォルトの状態に設定されています。ほかの主モードは、さまざまなキーやオプションを再定義しています。たとえば、 Lisp Interaction モードでは `LFD` (`eval-print-last-sexp`)、 `TAB` (`lisp-indent-line`) やそのほかのキーに関して、機能の割当が変更されています。

特殊な編集作業のために新しいコマンドをいくつも作らなくてはならなくなったとき、新しい主モードを作るのは通常よい選択です。主モードを作るのは (副モードの作成がたいてい困難なのに比べれば) それほど難しくありません。

もし、新しいモードによく似たモードがすでにあっても、その古いモードを書き換えて二つの目的の両方に対応できるようにしようとするのは賢明ではありません。なぜなら、そのようなモードは使用するのも維持するのも難しくなりがちだからです。代わりに、既存のモードをコピーし名前を変更してそれを書き換えるようにするか、または、『導出モード (derived mode)』 (20.1.5 節「導出モードを定義する」p.326 参照) を定義するようにしてください。たとえば、 Rmail Edit モード (‘`emacs/lisp/rmailedit.el`’ で定義されています) は、三つの新しいコマンドが使えるほかは Text モードにそっくりな主モードです。 Rmail Edit モードの定義は Text モードの定義とは異なるものですが、全く別に作られたものではなく、 Text モードから導出されたものです。

Rmail Edit モードは、テキストを一時的に異なる主モードにして、別の方法で (つまり、 Rmail のコマンドではなく、通常の Emacs のコマンドで) テキストを編集できるようにする場合の例になっています。このような場合に一時的に使用される主モードは、そのバッファの通常のモード (この例では Rmail モード) に戻すためのコマンドを持っています。似たような機能を実現するには、コマンドなどの再定義を再帰編集の中で行ない、ユーザが再帰編集から抜けた

ところで定義を戻す、という方法もありますが、ユーザが複数のバッファで同時にこのようなことを行ったときに不便さを強要することになり、好ましくありません。再帰編集は最後に始まったものからしか順に抜けられないのです。別の主モードを使用すればこの制限はありません。18.10 節「再帰編集」p.288 を参照してください。

GNU Emacs が標準的に使用しているライブラリ・ディレクトリにはいくつかの主モードのためのコードがあり、'text-mode.el'、'texinfo.el'、'lisp-mode.el'、'c-mode.el'、'rmail.el' などに納められています。これらのファイルを見て、主モードがどのように記述されるのかを知ることができます。Fundamental モードを除くと Text モードが最も単純なモードでしょう。また、Rmail モードは複雑で特殊なモードです。

20.1.1 主モードに関する慣例

既存の主モードのコードは、ローカル・キーマップや構文テーブルの初期化、グローバルな名前、フックなどに関するさまざまな慣例にしたがって記述されています。新しい主モードを作るときには、以下の慣例にしたがってください。

- カレント・バッファを新しいモードに切替えるコマンドとして、名前が '-mode' で終る、引数なしのコマンドを定義してください。このコマンドを『モード・コマンド (mode command)』または『モード関数 (mode function)』と呼びます。このコマンドの中で、キーマップ、構文テーブル、バッファにローカルな変数のセットアップを、バッファ内のテキストを変更することなく行うようにしてください。
- そのモードの中でのみ有効なコマンドに関する説明を、モード・コマンドの説明文字列に書いてください。ユーザがモードの中で C-h m (describe-mode) をするとこの文字列が表示されます。

  また、説明の内容を自動的にユーザのキー・バインドに合わせるために説明文字列で使用できる特殊な文字列 ('\[*command*]'、'\{*keymap*}'、'\<*keymap*>') が用意されています。21.3 節「説明文字列でのキー・バインドの置換」p.342 を参照してください。
- 主モード・コマンドの先頭で、kill-all-local-variables を呼び出してください。これは、これまで有効だった主モードのローカル変数の影響を受けないようにするためです。
- 主モード・コマンドの中で、主モード・コマンド自身を指すシンボルを、変数 major-mode にセットするようにしてください。これにより、describe-mode が正しく説明文字列を表示できるようになります。
- 主モード・コマンドで、モードの "短くわかりやすい" 名前を表す文字列を、変数 mode-name にセットするようにしてください。この文字列はモード行の中に表示されます。
- 全てのグローバルな名前が単一のネーム・スペースの中にあります。モードに固有なグローバル変数、定数、関数の名前は、モード名 (長いようなら略名) で始まるようにしてください。A.1 節「明確な Lisp プログラムの記述」p.635 を参照してください。
- 各主モードでそれぞれ独自のキーマップを持ち、そのモードになっている全てのバッファで共通なローカル・キーマップとして使用できるようにしてください。主モード関数の中で use-local-map を呼ぶことによって、このローカル・マップを作るようにしてください。詳細は 19.6 節「アクティブ・キーマップ」p.298 を参照してください。

このキーマップは、モード名 -mode-map という名前のグローバル変数で保持するようにしてください。普通はそのモードを定義するライブラリがこの変数を設定します。

- モードが独自の構文テーブルを持ったり、あるいは、ほかの関連する主モードと構文テーブルを共有することができます。もし独自の構文テーブルをもつ場合は、モード名 -mode-syntax-table という名前の変数で保持するようにしてください。第 31 章「構文テーブル」p.541 を参照してください。
- モードが独自の省略形テーブルを持ったり、あるいは、ほかの関連する主モードと省略形テーブルを共有することができます。もし独自の省略形テーブルをもつ場合は、モード名 -mode-abbrev-table という名前の変数で保持するようにしてください。32.2 節「省略形テーブル」p.553 を参照してください。
- モードに関連する変数は、すでに値を持っている場合に再初期化されないように、 defvar を使用して値をセットするようにしてください。そのような再初期化は、ユーザによるカスタマイズを無効にしてしまいます。
- Emacs をカスタマイズするための変数をバッファローカルにする場合には、 make-variable-buffer-local ではなく make-local-variable を使用してください。前者を使用すると、全てのバッファにおいて次にその変数に値がセットされたときにその変数がバッファローカルになってしまい、このモードに無関係なバッファに悪影響を及ぼしてしまう可能性があります。モードはそのような大きな影響を起こすべきではありません。10.9 節「バッファローカルな変数」p.140 を参照してください。

  単一の Lisp パッケージの中でのみ使用されている変数に対して make-variable-buffer-local を使用することは構いません。
- そのモードに対する『モード・フック (mode hook)』を、モード名 -mode-hook という名前で作ってください。主モード・コマンドの最後の処理として、 run-hooks を使用してそのフックを実行するようにしてください。20.4 節「フック」p.335 を参照してください。
- 主モード・コマンドは、ほかの基本的なモードのフックを実行しても構いません。たとえば indented-text-mode は indented-text-mode-hook だけではなく text-mode-hook も実行します。このようなほかのモードのフックの実行は、モード固有のフックの直前 (つまり、ほかの処理が全て終った後) で行っても構いませんし、あるいは、もっと早い段階で行っても構いません。
- もし、ユーザがバッファをほかのモードに切替えたときに何か特殊なことを行ないたい場合は、 change-major-mode-hook にローカルな値をセットしておくことができます。
- もし、このモードが特殊なテキストのみで意味があるのなら、主モード・コマンドを指すシンボルが、 special の値をもつ mode-class という名前の属性をもつようにしてください。たとえば以下のようにしてセットします。

  ```
  (put 'funny-mode 'mode-class 'special)
  ```

  これは、 Funny モードになっているバッファから新しいバッファを作るときに、新しいバッファが Funny モードを継承しないよう、 Emacs に伝える例です。 Dired、 Rmail、Buffer List といったモードはこのように設定されています。
- ある名前を持ったファイルに対してそのモードがデフォルトとして自動的に選ばれるようにしたい場合は、 auto-mode-alist にその旨をつけ加えてください。モード・コマンドが

自動ロードされるように設定する場合には、 autoload を呼ぶそのファイルの中で auto-mode-alist の設定をする必要があります。そうでない場合は、モードの定義を行っているファイルの中で auto-mode-alist の設定も行うことにしていいでしょう。 20.1.3 節「主モードの自動選択の仕組み」p.323 を参照してください。

- auto-mode-alist に要素を追加する式や autoload 形式をユーザーが '.emacs' ファイルに追加できるように、それぞれの記述例をドキュメントに記載してください。
- モード定義のトップレベルの形式を複数回実行しても、有害な動作をしないようにしてください。定義ファイルを何回もロードしようとする人がいるかも知れません。

change-major-mode-hook Variable

kill-all-local-variables は、その最初の処理としてこの正規フックを実行します。このフックを利用して、ユーザがバッファをほかの主モードに変更したときに行う特殊な処理を定義できます。続く主モードに悪影響を及ぼさないように、この変数をバッファにローカルにして、フックの実行後にこの変数が消えるようにしてください。 20.4 節「フック」p.335 を参照してください。

20.1.2 主モードの例

Text モードは、 Fundamental モードを除くと最も単純な主モードでしょう。以下は、'text-mode.el' から抜粋した、上述のさまざまな慣例の実例です。 (訳注: 実際のコード中のコメント類は英語で書かれています)

```
;; モード特有の構文テーブルを作る。
(defvar text-mode-syntax-table nil
  "Syntax table used while in text mode.")

(if text-mode-syntax-table
    ()              ; すでにテーブルができ上がっていたら書き換えない。
  (setq text-mode-syntax-table (make-syntax-table))
  (modify-syntax-entry ?\" ".   " text-mode-syntax-table)
  (modify-syntax-entry ?\\ ".   " text-mode-syntax-table)
  (modify-syntax-entry ?' "w   " text-mode-syntax-table))

(defvar text-mode-abbrev-table nil
  "Abbrev table used while in text mode.")
(define-abbrev-table 'text-mode-abbrev-table ())

(defvar text-mode-map nil)   ; モード特有のキーマップを作る。

(if text-mode-map
    ()      ; キーマップの中身がすでにでき上がっていたら書き換えない。
  (setq text-mode-map (make-sparse-keymap))
  (define-key text-mode-map "\t" 'tab-to-tab-stop)
  (define-key text-mode-map "\es" 'center-line)
  (define-key text-mode-map "\eS" 'center-paragraph))
```

以下は、 Text モードの主モード関数の完全な定義です。

```
(defun text-mode ()
  "Major mode for editing text intended for humans to read.
 Special commands: \\{text-mode-map}
Turning on text-mode runs the hook `text-mode-hook'."
  (interactive)
  (kill-all-local-variables)
```

```
(use-local-map text-mode-map)     ; ローカル・キーマップを使う。
(setq mode-name "Text")           ; モード行に表示される名前。
(setq major-mode 'text-mode)      ; describe-mode がきちんと
                                  ; 説明文字列を見つけられるように。
(setq local-abbrev-table text-mode-abbrev-table)
(set-syntax-table text-mode-syntax-table)
(run-hooks 'text-mode-hook))      ; こうしておくと、ユーザがフック
                                  ; を使ってカスタマイズできる。
```

三つの Lisp モード (Lisp モードと Emacs Lisp モードと Lisp Interaction モード) は、 Text モードよりも多くの機能を持っていて、どれも非常に複雑です。以下の例は、'`lisp-mode.el`' からの抜粋です。これらのモードがどのように書かれているかを見てください。

```
;; モード特有のテーブルを保持する変数を作る。
(defvar lisp-mode-syntax-table nil "")
(defvar emacs-lisp-mode-syntax-table nil "")
(defvar lisp-mode-abbrev-table nil "")

(if (not emacs-lisp-mode-syntax-table) ; テーブルの中身がすでにでき
                                       ; 上がっていたらそのままに。
    (let ((i 0))
      (setq emacs-lisp-mode-syntax-table (make-syntax-table))

      ;; 数字の 0 未満の文字について、シンボル名の一部であって
      ;;   ワードではない、と定義する。
      ;;   (数字の 0 は ASCII コードで 48。)
      (while (< i ?0)
        (modify-syntax-entry i "_   " emacs-lisp-mode-syntax-table)
        (setq i (1+ i)))
      ...
      ;; ほかの文字についても構文を定義する。
      (modify-syntax-entry ?  "    " emacs-lisp-mode-syntax-table)
      (modify-syntax-entry ?\t "    " emacs-lisp-mode-syntax-table)
      ...
      (modify-syntax-entry ?\( "()  " emacs-lisp-mode-syntax-table)
      (modify-syntax-entry ?\) ")(  " emacs-lisp-mode-syntax-table)
      ...))
;; lisp-mode で使用される省略形テーブルを作る。
(define-abbrev-table 'lisp-mode-abbrev-table ())
```

沢山のコードが三つの Lisp モードの間で共有されています。以下の関数はさまざまな変数を設定する機能があり、それぞれの Lisp モードから利用されます。

```
(defun lisp-mode-variables (lisp-syntax)
  ;; Emacs Lisp では lisp-syntax 引数は nil、
  ;;   それ以外の二つの Lisp モードでは t。
  (cond (lisp-syntax
         (if (not lisp-mode-syntax-table)
             ;; Emacs Lisp モードの構文テーブルは常に存在している。
             ;;   しかし Lisp モードの構文テーブルは必要になってか
             ;;   ら作る。これは記憶領域の節約のため。
             (progn (setq lisp-mode-syntax-table
                          (copy-syntax-table emacs-lisp-mode-syntax-table))
                    ;; Lisp モードのためにいくつかのエントリを変更する
                    (modify-syntax-entry ?\| "\"   "
                                         lisp-mode-syntax-table)
                    (modify-syntax-entry ?\[ "_   "
                                         lisp-mode-syntax-table)
                    (modify-syntax-entry ?\] "_   "
                                         lisp-mode-syntax-table)))
```

```
            (set-syntax-table lisp-mode-syntax-table)))
    (setq local-abbrev-table lisp-mode-abbrev-table)
    ...)
```

forward-paragraph のような関数は paragraph-start 変数の値を使います。 Lisp コードは通常のテキストとは違うので、 paragraph-start 変数の値は Lisp に合わせて設定されなければなりません。また、 Lisp ではコメントは特殊なスタイルで字下げされるので、 Lisp モードに特化した comment-indent-function も必要です。これらの変数を設定するためのコードが、 lisp-mode-variables の残りの部分に書かれています。

```
    (make-local-variable 'paragraph-start)
    ;; こんな形で `^' を入れておかなきゃならないのは美しくないけど、
    ;; すでに page-delimiter にも混じっているし、取り除くのも面倒。
    (setq paragraph-start (concat "^$\\|" page-delimiter))
    ...
    (make-local-variable 'comment-indent-function)
    (setq comment-indent-function 'lisp-comment-indent))
```

それぞれの Lisp モードは、少しずつ異なるキーマップを持っています。たとえば、 Lisp モードでは C-c C-z は run-lisp にバインドされていますが、ほかの二つの Lisp モードでは未定義です。しかしながら、すべての Lisp モードで共通なコマンドもあります。以下の関数は、これらの共通なコマンドを引数で与えられたキーマップに加えます。

```
(defun lisp-mode-commands (map)
  (define-key map "\e\C-q" 'indent-sexp)
  (define-key map "\177" 'backward-delete-char-untabify)
  (define-key map "\t" 'lisp-indent-line))
```

以下のコードは、キーマップの初期化のために、 Emacs Lisp モードのコードの中で lisp-mode-commands を使用する例です。まず最初に、 defvar を使用してモードに特有なキーマップを保持するための変数を宣言します。この defvar は、その変数が値を持っていないときは値を nil にセットします。それから、その変数が nil ならば、新しくキーマップをセットアップします。

このコードは、もしすでにキーマップがセットアップされていたら、変更せずにそのままにします。これにより、ユーザがキーマップをカスタマイズできます。

```
(defvar emacs-lisp-mode-map () "")
(if emacs-lisp-mode-map
    ()
  (setq emacs-lisp-mode-map (make-sparse-keymap))
  (define-key emacs-lisp-mode-map "\e\C-x" 'eval-defun)
  (lisp-mode-commands emacs-lisp-mode-map))
```

で、以下が、 Emacs Lisp モードのための主モード関数の完全な定義です。

```
(defun emacs-lisp-mode ()
  "Major mode for editing Lisp code to run in Emacs.
Commands:
Delete converts tabs to spaces as it moves back.
Blank lines separate paragraphs.  Semicolons start comments.
\\{emacs-lisp-mode-map}
Entry to this mode runs the hook `emacs-lisp-mode-hook'."
  (interactive)
  (kill-all-local-variables)
  (use-local-map emacs-lisp-mode-map)    ; ローカル・キーマップを使用する。
  (set-syntax-table emacs-lisp-mode-syntax-table)
```

```
(setq major-mode 'emacs-lisp-mode)      ; describe-mode が説明
                                        ;   文字列を見つけられるように。
(setq mode-name "Emacs-Lisp")           ; モード行に表示されるモード名。
(lisp-mode-variables nil)               ; いろいろな変数を定義する。
(run-hooks 'emacs-lisp-mode-hook))      ; ユーザがカスタマイズのために定
                                        ;   定したフックをここで実行する。
```

20.1.3 主モードの自動選択の仕組み

ファイルを読み込んで新しくバッファを作る際に、 Emacs は、ファイル名やファイルの内容から主モードを自動的に選び出します。

fundamental-mode Command

Fundamental モードは、特別これといって変わったところのないモードです。ほかの主モードは、このモードを元に何がどう変わるのかという記述で定義されます。ですから、Fundamental モードをカスタマイズするべきではなく、また、 `fundamental-mode` 関数はいかなるフックも実行しません。 (Fundamental モードの振舞を変えたい場合は、Emacs の *global* state(グローバル・キーマップなど) を変更するようにしてください)。

normal-mode &optional *find-file* Command

この関数は、適切な主モードの立ち上げとローカル変数の設定を、カレント・バッファに対して行ないます。この関数は、最初に、 `set-auto-mode` を呼んでモードを立ち上げ、次に、 `hack-local-variables` を呼んでファイルに記述されているローカル変数の設定や式の評価を行ないます。

もし、 `normal-mode` 関数への *find-file* 引数が非 `nil` なら、 `normal-mode` は `find-file` より呼ばれているものと仮定します。この場合、ファイルの最後尾や '`-*-`' 行にあるローカル変数リストの処理を行うかもしれません。変数 `enable-local-variables` が、その処理を行うかどうかのコントロールを行ないます。

`normal-mode` をインタラクティブに (コマンドとして) 実行した場合、通常 *find-file* は `nil` になります。この場合、 `normal-mode` は無条件に全てのローカル変数リストの処理を行ないます。 (ローカル変数リストについては *The GNU Emacs Manual* の "Local Variables in Files" 節を参照してくださいを参照)

`normal-mode` は、主モード関数の呼出しに `condition-case` を使用します。そのため、主モード関数の評価中に発生したエラーは捕捉され、 '`File mode specification error`' としてオリジナルのエラー・メッセージと一緒に報告されます。

enable-local-variables User Option

この変数は、読み込まれたファイルのローカル変数リストをどのように扱うかを決定します。この値が `t` ならば処理し、 `nil` ならば無視、それ以外ならばファイルごとにどのように扱うかをユーザに尋ねます。デフォルト値は `t` です。

ignored-local-variables Variable

この変数は、ローカル変数リストによって設定されたくない変数のリストを表します。このリストに含まれる変数に対する値は、もしそれが存在しても、すべて無視されます。

このリストに加えてさらに、名前の `risky-local-variable` 属性が非 `nil` であるような変数に対する値も無視されます。

enable-local-eval User Option

この変数は、読み込まれたファイルのローカル変数リストに含まれる ‘`Eval:`’ をどのように扱うかを決定します。この値が `t` ならば処理し、 `nil` ならば無視、それ以外ならばファイルごとにどのように扱うかをユーザに尋ねます。デフォルト値は `maybe` です。

set-auto-mode Function

この関数は、カレント・バッファに適切な主モードを設定します。モードを選び出す判断材料となるのは、‘`-*-`’行の値、読み込まれたファイルの名前 (`auto-mode-alist` を使用)、およびローカル変数の値です。ファイル最後尾付近の ‘`mode:`’ ローカル変数の値の処理は、この関数ではなく `hack-local-variables` 関数が行ないます。 *The GNU Emacs Manual* の “How Major Modes are Chosen” 節を参照してください.

default-major-mode User Option

この変数は、新しく作られるバッファのデフォルトの主モードを保持します。標準値は `fundamental-mode` です。

`default-major-mode` の値が `nil` の場合は、以前にカレントであったバッファの主モードを新しいバッファの主モードとして使います。ただし、主モード・シンボルが `mode-class` 属性を持ちその値が `special` である場合は、この `default-major-mode` 変数の値に関わらず Fundamental モードが使われます。 Dired や Rmail のように、特殊なテキストでのみ意味のあるモードが、この属性を持っています。

set-buffer-major-mode *buffer* Function

この関数は、 *buffer* の主モードを `default-major-mode` の値にセットします。もしその変数が `nil` ならば、カレント・バッファの主モードを使用します (もしそれが適当ならば)。

バッファを作るための低レベルなプリミティブはこの関数を呼ぶことはありませんが、`switch-to-buffer` や `find-file-noselect` のような中レベルのコマンドは、バッファを作る度にこの関数を呼び出します。

initial-major-mode Variable

この変数は、 Emacs の起動時に最初に作られる ‘`*scratch*`’ バッファの主モードを決めます。この変数の値は、主モードコマンド名を示すシンボルでなければなりません。デフォルト値は `lisp-interaction-mode` です。

auto-mode-alist Variable

この変数は、ファイル名のパターンとそれに対応する主モード関数の連想リストを保持します。通常、ファイル名のパターンは ‘`.el`’ や ‘`.c`’ といった拡張子を識別するものですが、そうでなくても構いません。リストの要素は、 `(`*regexp* `.` *mode-function*`)` のような形式をしています。

例:

```
(("^/tmp/fol/" . text-mode)
 ("\\.texinfo\\'" . texinfo-mode)
 ("\\.texi\\'" . texinfo-mode)
 ("\\.el\\'" . emacs-lisp-mode)
 ("\\.c\\'" . c-mode)
 ("\\.h\\'" . c-mode)
 ...)
```

展開されたファイル名 (22.8.4 節「ファイル名を展開する関数」p.370 参照) が *regexp* に一致するファイルを読み込んだとき、 `set-auto-mode` はそれに対応した *mode-function* を呼びます。この機能により、ほとんどのファイルについて、適切な主モードの選択が自動化できます。

もし、 `auto-mode-alist` の要素が (*regexp function* `t`) のような形式だった場合、*function* を呼び出した後、ファイル名の *regexp* に一致しなかった残りの部分について `auto-mode-alist` の検索を再び行ないます。 (訳注: GNU Emacs バージョン 19.28 の DOC の記述によると (*regexp function*) の形式となっているが、それは誤り。 19.29 では修正された。)

この "もう一度一致 (match-again)" の機能は、 uncompression パッケージにおいて便利です。たとえば、 `("\\.gz\\'"` *function* `t`) の記述によって、ファイルを gunzip して、さらにその伸長されたファイルを、 '`.gz`' を除いたファイル名から求めた適切なモードにする、といった利用ができるでしょう。

複数のパターンのペアを `auto-mode-alist` に設定する方法の例を挙げます (このような設定を '`.emacs`' で行うこともできます)。

```
(setq auto-mode-alist
  (append
   ;; `.' (ピリオド) で始まるファイル名。
   '(("/\\.[^/]*\\'" . fundamental-mode)
     ;; `.' (ピリオド) を含まないファイル名。
     ("[^\\./]*\\'" . fundamental-mode)
     ;; `.C' で終わるファイル名。
     ("\\.C\\'" . c++-mode))
   auto-mode-alist))
```

interpreter-mode-alist Variable

この変数は、 '`#!`' でコマンド・インタープリタを特定しているスクリプトに使用する主モードを指定します。この変数の値は、 (*interpreter* . *mode*) のような形式の要素をもつリストで、たとえば、デフォルトでは `("perl" . perl-mode)` などが指定されています。この要素は、 *interpreter* のスクリプト・ファイルに対して *mode* を使用することを意味します。

この変数は、使用する主モードを `auto-mode-alist` で決定できなかったときにのみ適用されます。

hack-local-variables &optional *force* Function

この関数は、カレント・バッファの文字列からローカル変数や式の定義を捜し出し、実際に変数の定義や式の評価を行ないます。

`normal-mode` の説明中の `enable-local-variables` の扱いに関する記述を参照してください。引数 *force* は通常の場合、 `normal-mode` 関数の *find-file* 引数を引き継いだものです。

20.1.4 主モードの使い方をユーザに説明する

`describe-mode` 関数は、主モードに関する情報を提供するために使用されます。通常、 `C-h m` により呼び出されます。 `describe-mode` 関数は `major-mode` の値を使用します。これは、主モード関数が `major-mode` 変数をセットしなければならない理由の一つです。

describe-mode Command

この関数は現在使用中の主モードの説明を表示します。

`describe-mode` 関数は、 `major-mode` を引数に `documentation` 関数を呼ぶことで、主モード関数の説明文字列を表示します。 (21.2 節「説明文字列へのアクセス」 p.340 を参照してください)。

major-mode Variable

この変数は、カレント・バッファの主モードを示すシンボルを保持します。このシンボルは、その主モードへ切替えるためのコマンド関数の定義を持っているはずです。 `describe-mode` 関数は、主モードの説明文字列としてこの関数の説明文字列を使用します。

20.1.5 導出モードを定義する

新しいモードを定義するときに、既存の主モードを基にしたくなることがよくあります。これを簡単に行う方法の一つは `define-derived-mode` を使うことです。

define-derived-mode *variant parent name docstring body...* Macro

このマクロは、主モード・コマンドとして *variant* を定義します。このとき、モードの文字列形式の名前として *name* が使用されます。

新しくできるコマンド *variant* は、関数 *parent* を呼んだ後、以下のように親モードのいくつかの側面をオーバーライドするように定義されます。

- 新しいモードは、 *variant*`-map` という名前の独自のキーマップを持ちます。 `define-derived-mode` は、このキーマップがまだ値を持っていない場合は、 *parent*`-map` を継承するように初期化します。
- 新しいモードは、 *variant*`-syntax-table` という名前で独自の構文テーブルを持ちます。 `define-derived-mode` は、もしこのテーブルがセットされていない場合は、 *parent*`-syntax-table` をコピーして初期化します。
- 新しいモードは、 *variant*`-abbrev-table` という名前で独自の省略形テーブルを持ちます。 `define-derived-mode` は、もしこのテーブルがセットされていない場合は、 *parent*`-abbrev-table` をコピーして初期化します。

- 新しいモードは *variant*-hook という独自のモード・フックを持ち、このフックは通常どおり主モード関数の最後の処理として実行されます (新しいモードは、 *parent* を呼ぶことにより、 *parent* のモード・フックも実行します)。

さらに、 *body* を使用することで、 *parent* のほかの面をどのようにオーバーライドするかを指定できます。 *variant* コマンドは、そのオーバーライド処理の後、 *variant*-hook を実行する直前で、 *body* が保持する形式を評価します。

引数 *docstring* は、新しいモードの説明文字列を指定します。 *docstring* を省略した場合、 define-derived-mode が説明文字列を自動的に作ります。

以下は架空の例です。

```
(define-derived-mode hypertext-mode
  text-mode "Hypertext"
  "Major mode for hypertext.
\\{hypertext-mode-map}"
  (setq case-fold-search nil))

(define-key hypertext-mode-map
  [down-mouse-3] 'do-hyper-link)
```

20.2 副モード

『副モード (minor mode)』は、それを利用するかしないかを、主モードの選択とは無関係にユーザが選択できるような機能を提供します。副モードは単独で使用することもできるし、組み合わせて使用することもできます。 “副モード” という呼び名よりは “一般的に利用可能な、オプショナルな機能のモード” の方が、長過ぎるけども、より的確でしょう。

副モードは、ほとんどの場合、さまざまな主モードで使用できます。たとえば、 Auto Fill モードは、テキストの挿入を許す主モードであれば、どのような主モードの中でも利用可能です。一般論として、副モードとは、主モードの振舞に関わらず利用できるように作られるべきものです。

副モードの実装は、主モードの実装に比べて、たいてい非常に困難です。理由の一つとして、使用、不使用の選択がどのような順序で行なわれるかわからないことがあります。副モードは、主モードやほかの副モードの影響を受けずにきちんと動かなければならないのです。

副モードを実装する上で最も大きな問題となり得るのは、モードに必要なフックを Emacs につけ加える方法です。副モード・キーマップを使えば、従来使われてきた手段よりも簡単にこの問題を解決できます。

20.2.1 副モードの記述方法に関する慣例

主モードの場合と同様に、副モードの書き方にも慣例があります。主モードでの慣例のいくつか、たとえば、モードを初期化する関数の名前、グローバルなシンボルの名前、キーマップなどのテーブルの扱い、などは副モードでもそのまま当てはまります。

さらに、副モード特有の慣例もいくつかあります。

- 副モードを表し、名前が '-mode' で終る変数を作ってください。この変数の値によりそのモードの動作を有効にするか無効にするかを決定するようにしてください (nil なら無効にし、そのほかなら有効にする)。この変数を『モード変数 (mode variable)』と呼びます。

  この変数は、副モードの名前をモード行に表示するために、 minor-mode-alist とともに使用されます。また、副モード・キーマップが有効か否かを決めます。また、コマンドやフックが、この変数の値をチェックすることもあります。

  もし、副モードを各バッファごとに独立に有効にしたければ、この変数をバッファローカルにしてください。

- モード変数と同じ名前のコマンドを定義してください。そのコマンドの役割は、モード変数を設定してモードを有効 / 無効にすることです。

  そのコマンドは、一つのオプション引数を取るようにしてください。その引数が nil なら、モードをトグルするようにしてください (モードがオンだったらオフにし、オフだったらオンにする)。引数が正の整数、または、 nil でも -でもないシンボル、または、正の整数やシンボルを CAR にもつリストならばモードをオンにし、そのほかの場合はモードをオフにしてください。

  以下は transient-mark-mode の定義より抜き出した例です。モードの振る舞いを有効 / 無効にする変数としての transient-mark-mode の使い方と、前置引数の値にしたがってモードをトグル、あるいは有効 / 無効にするための正しい方法を示しています。

  ```
  (setq transient-mark-mode
        (if (null arg) (not transient-mark-mode)
          (> (prefix-numeric-value arg) 0)))
  ```

- minor-mode-alist に、それぞれの副モードに適切な値を追加してください (20.3.2 節「モード行の中で使用されている変数」p.331 参照)。この値は以下に示す形式のリストです。

  (*mode-variable string*)

  ここで、 *mode-variable* は、副モードが有効かを示す変数です。また、 *string* は、モードを示すためにモード行の中に表示される、空白文字で始まる短い文字列です。この文字列は、ほかのモードを示す文字列と一緒に表示できるよう、短いものでなくてはなりません。

  minor-mode-alist に値を追加しようとする際、すでに同じモードの値がないか assq を使用してチェックし、重複を避けるようにしてください。たとえば以下のようにします。

  ```
  (or (assq 'leif-mode minor-mode-alist)
      (setq minor-mode-alist
            (cons '(leif-mode " Leif") minor-mode-alist)))
  ```

20.2.2 キーマップと副モード

副モードは、そのモードが有効なときにのみ参照される独自のキーマップをもつことができます。副モード独自のキーマップを持ちたいときは、連想リスト minor-mode-map-alist に要素を追加します。 19.6 節「アクティブ・キーマップ」p.298 を参照してください。

副モード・キーマップの用途の一つは、いくつかの自己挿入文字の振舞を変更し、自己挿入以外の機能を持たせることです。一般に、このような目的の場合、副モード・キーマップの使用が

唯一の方法です。なぜなら、 `self-insert-command` のカスタマイズが可能なのは特殊な場合 (abbrevs モードと Auto Fill モードのみ) に限られているからです (`self-insert-command` の定義を独自のもので置き換えたりしないでください。 `self-insert-command` は、エディタのコマンド・ループの中で特殊な扱いをされています)。

minor-mode-map-alist Variable

この変数は、以下に示すような形式の要素をもつ連想リストです。

(*variable* . *keymap*)

ここで、 *variable* は副モードが有効かを示す変数で、 *keymap* はキーマップです。このキーマップ *keymap* は、 *variable* が `nil` でない値をもつときにのみ有効です。

`minor-mode-map-alist` の要素の形式は、 `minor-mode-alist` の要素の形式とは違うことに注意してください。要素は、 *variable* と *keymap* からなるリストではありません。キーマップが要素の CDR になっていなければなりません。

また、キーマップそのものが CDR に来ていなければなりません。キーマップを保持する変数を示すシンボルを CDR に置いても、望むような動作はしないでしょう。

複数の副モード・キーマップがアクティブである場合、優先順位は `minor-mode-map-alist` での並び順と一緒です。しかし、ほかの副モードの影響を受けないように副モードを設計するべきです。そのようにできれば、順序について悩むことはありません。

20.3 モード行の書式

Emacs のウィンドウはそれぞれにモード行を持ち (ミニバッファ・ウィンドウは除く)、そのウィンドウに表示されているバッファに関する情報を表示しています。モード行に表示される情報には、バッファの名前、結び付いているファイル、再帰編集の深さ、主モードと副モード、などがあります。

この節では、モード行の表示内容がどのように制御されるかについて説明します。モードの章で説明するのは、モード行に表示される情報の多くが、有効な主モードと副モードに関係するからです。

`mode-line-format` は、カレント・バッファのモード行の表示に使われるテンプレートを保持するバッファローカル変数です。同一のバッファを表示するウィンドウは同一の `mode-line-format` を使用するので、モード行の表示も (表示位置のパーセント表示と行番号は除いて) 同一になります。

モード行の表示の更新は、通常、ウィンドウに表示されているバッファが切り替わったときや、バッファの modified-status が `nil` から `t`、あるいは `t` から `nil` に変化したときに行なわれます。ですから、 `mode-line-format` から参照される変数 (20.3.2 節「モード行の中で使用されている変数」p.331 参照) を変更しても、直ちに表示に反映されるとはかぎりません。

force-mode-line-update Function

強制的にカレント・バッファのモード行の再表示を行ないます。

通常、モード行は反転表示されます (35.12 節「反転表示 (Inverse Video)」 p.617 の `mode-line-inverse-video` を参照)。

20.3.1 モード行の表示をコントロールするデータ構造

モード行の内容は、バッファローカル変数 `mode-line-format` に含まれるリスト、文字列、シンボル、数値からなるデータ構造によりコントロールされます。このデータ構造は『モード行構成要素 (mode line construct)』と呼ばれ、より簡単なモード行構成要素の再帰的な組み合わせにより構成されます。また、同じデータ構造がフレームのタイトルを作るためにも使用されています (26.4 節「フレーム・タイトル」 p.443 参照)。

mode-line-format Variable

この変数の値は、モード行全体の書式を制御するモード行構成要素です。この変数の値は、どの変数がモード行のどの場所に現れるかをコントロールします。

モード行構成要素は単なる文字列でも構いませんが、通常は、ほかの変数をどのようにモード行の構築に使用するかを指定します。これらの変数の多くは、それ自身、モード行構成要素を値としてもつように定義されています。

デフォルトの `mode-line-format` の値は、 `mode-name` や `minor-mode-alist` といった変数の値を使用するよう定義されています。ですから、 `mode-line-format` を書き換えなければならないモードはほとんどありません。たいていの場合、 `mode-line-format` から参照される変数を書き換えるだけで十分間に合います。

モード行構成要素は、リスト、シンボル、または文字列のいずれかです。もし、値がリストならば、その要素はそれぞれ、リスト、シンボル、または文字列のいずれかです。

string
: 文字列であるモード行構成要素は、『%- 指定子 (%-construct)』を除いてそのまま表示されます。 ‘%’ に続く 10 進の数字列は、あまりがスペースで埋められるフィールドの全体幅を示します (データは左詰めされるということです)。 20.3.3 節「モード行の表示に使われる %- 指定子」 p.334 を参照してください。

symbol
: シンボルであるモード行構成要素は、その値を表します。 *symbol* の値が *symbol* の代わりにモード行構成要素として扱われます。ただし、シンボルが `t` または `nil` の場合は無視されます。同様に、値が void であるシンボルも無視されます。

一つだけ例外があり、 *symbol* の値が文字列の場合は、その値がそのまま表示されます。その文字列中の %- 指定子は、特別な意味を持ちません。

(*string rest. . .*) or (*list rest. . .*)
: 最初の要素が文字列、またはリストであるリストは、全ての要素を再帰的に処理し、その結果を結合することを意味します。これは、モード行構成要素の形式として、最もよくある形です。

(*symbol then else*)
: 最初の要素がシンボルであるリストは、条件を表します。その意味は、 *symbol* の値により異なります。値が非 `nil` のときは、 2 番目の要素 *then* がモード行構成

要素として再帰的に処理されます。また、 *symbol* の値が `nil` ならば、 3 番目の要素 *else* が再帰的に処理されます。 *else* を省略することもでき、省略した場合、 *symbol* の値が `nil` ならば、そのモード行構成要素は何も表示しません。

(*width rest*...)

先頭の要素が整数であるリストは、 *rest* の結果の幅詰め、またはパディングの指定です。まず、残りの要素 *rest* がそれぞれモード行構成要素として再帰的に処理され、互いに結合されます。その後、結果が *width* 桁になるようにスペースで埋められるか (*width* が正の整数の場合)、あるいは右側が切り捨てられます (*width* が負の整数の場合、 $-width$ 桁に)。

たとえば、ウィンドウ上端のバッファ中での位置をパーセンテージ表示するための通常の指定は、 `(-3 "%p")` のようになります。

`mode-line-format` そのものを書き換えてしまう場合でも、デフォルト値 (20.3.2 節「モード行の中で使用されている変数」 p.331 参照) で使用されている変数を、その内容を複製したり別の方法で表示したりせずに、そのまま新しい値の中でも使用するべきです。ユーザや Lisp プログラム (`display-time` や主モードなど) によるモード行の表示に関するカスタマイズの多くは、それらの変数の値の変更によりなされるからです。

以下は、ホスト名と現在のディレクトリをモード行に表示する `mode-line-format` の例です。 `shell-mode` では便利かも知れません。

```
(setq mode-line-format
  (list ""
   'mode-line-modified
   "%b--"
   (getenv "HOST")        ; One element is not constant.
   ":"
   'default-directory
   "   "
   'global-mode-string
   "   %[("
   'mode-name
   'mode-line-process
   'minor-mode-alist
   "%n"
   ")%]----"
   '(line-number-mode "L%l--")
   '(-3 . "%p")
   "-%-"))
```

20.3.2 モード行の中で使用されている変数

この節では、モード行のテキスト表示のために、 `mode-line-format` 内で標準的に使用されている変数について説明します。これらの変数は先天的に特殊なものではなく、ほかのいかなる

変数も、 `mode-line-format` がそれらを使用するように変更されれば、モード行において同様の効果をもつことができます。

mode-line-modified Variable

この変数は、カレント・バッファが変更されたかを示すモード行構成要素の値を持ちます。

`mode-line-modified` のデフォルト値は `("--%1*%1+-")` です。これは、バッファが変更されているときは '`--**-`'、バッファが変更されていないときは '`-----`'、バッファが読出し専用のときは '`--%%-`'、バッファが読出し専用でなおかつ変更されているときは '`--%*-`' を、モード行に表示することを意味します。

この変数を変更しても、モード行の強制的なアップデートは起こりません。

mode-line-buffer-identification Variable

この変数は、ウィンドウに表示されているバッファを識別するためのモード行構成要素を保持します。デフォルト値は `("%F: %17b")` で、'`Emacs:`' に続けてスペース 1 文字、17 文字のバッファ名を表示することを意味します (端末フレームでは、'`Emacs`' の代わりにフレーム名 ('`Emacs-`' + フレーム番号) を表示します (訳注: GNU Emacs バージョン 19.29 から、テキスト端末でも複数のフレームをもつことができるようになった))。Rmail などのように通常の Emacs とは違う振舞をするモードでは、この変数の値を変更したくなるかも知れません。

global-mode-string Variable

この変数は、モード行上でバッファ名に続いて表示されるモード行構成要素を保持します。コマンド `display-time` は、時刻とロード情報を保持する `display-time-string` 変数を参照するように、この `global-mode-string` を設定します。

'`%M`' は `global-mode-string` の値で置き換わります。しかし、これは obsolete です。代わりに、モード行内で `global-mode-string` 変数を直接使用することができます。

mode-name Variable

このバッファローカル変数は、カレント・バッファの主モードの "短くわかりやすい" 名前を保持します。モード行にモード名が表示されるよう、各主モードはこの変数にモード名を正しく設定しなければなりません。

minor-mode-alist Variable

この変数は、副モードがアクティブであるとき、どのようにモード行に表示するかを示す要素をもつ連想リストを保持します。 `minor-mode-alist` の要素は、それぞれ、二つの要素からなる以下のようなリストです。

(*minor-mode-variable mode-line-string*)

さらに一般的に、 *mode-line-string* は、どのようなモード行構成要素であっても構いません。この値は、 *minor-mode-variable* の値が非 `nil` のとき表示され、そうでないときは表示されません。これらの文字列は、互いにくっつかないよう、スペースで始まるようにしてください。慣習的に、副モードがアクティブになったときに、そのモードに対応する *minor-mode-variable* が非 `nil` にセットされます。

minor-mode-alist のデフォルト値は、以下のようになっています。

```
minor-mode-alist
⇒ ((vc-mode vc-mode)
    (abbrev-mode " Abbrev")
    (overwrite-mode overwrite-mode)
    (auto-fill-function " Fill")
    (defining-kbd-macro " Def")
    (isearch-mode isearch-mode))
```

minor-mode-alist は、バッファローカルではありません。この連想リストの中に現れる変数は、もしそれに対応する副モードがバッファごとに有効にできるものなら、バッファローカルでなければなりません。

mode-line-process Variable

このバッファローカル変数は、サブプロセスとやりとりをするようなモードで、プロセスの状態をモード行に表示するための文字列を持ちます。この文字列は、主モード名に続けて、スペースを開けずに表示されます。たとえば、'*shell*' バッファにおいてこの変数の値が (": %s") なら、'(Shell: run)' のように表示されます。通常、この変数は nil です。

default-mode-line-format Variable

この変数は、 mode-line-format を特に設定しないモードのためのデフォルト値を保持します。これは (default-value 'mode-line-format) と同じです。

default-mode-line-format のデフォルト値は、以下のようになっています。

```
(""
 mode-line-modified
 mode-line-buffer-identification
 "   "
 global-mode-string
 "   %[("
 mode-name
 mode-line-process
 minor-mode-alist
 "%n"
 ")%]----"
 (line-number-mode "L%l--")
 (-3 . "%p")
 "-%-")
```

vc-mode Variable

それぞれのバッファにローカルな変数 vc-mode は、バッファに読み込まれたファイルがバージョン・コントロールされているか、されているのならその種類は何か、を表します。この値が nil なら、バージョン・コントロールされていないことを表します。また、それ以外なら、バージョン・コントロールされており、モード行に表示される文字列を表します。

20.3.3 モード行の表示に使われる %- 指定子

使用可能な %- 指定子とその意味を、以下に示します。 '%%' 以外の全ての %- 指定子において、表示する文字数を表す十進数を、 '%' の後ろにつけることができます。

%b
: カレント・バッファ名。 buffer-name 関数の値と同一。 24.3 節「バッファの名前」p.395 を参照してください。

%f
: バッファに読み込まれたファイルの名前。 buffer-file-name 関数の値と同一。 24.4 節「バッファ・ファイル名」p.397 を参照してください。

%F
: 選択されているフレームの名前。

%c
: ポイント (カーソル位置) の桁番号。

%l
: ポイント (カーソル位置) の行番号。

%*
: バッファが読出し専用なら '%' (buffer-read-only を参照)、
読出し専用ではなくて変更されていたら '*' (buffer-modified-p を参照)、
それ以外なら '-'。 24.5 節「バッファの変更」p.399 を参照してください。

%+
: バッファが変更されていたら '*' (buffer-modified-p を参照)、
変更されていなくて読出し専用なら '%' (buffer-read-only を参照)、
それ以外なら '-'。変更されていてかつ読出し専用のバッファに関する振る舞いのみが '%*' と異なります。 24.5 節「バッファの変更」p.399 を参照してください。

%&
: バッファが変更されていたら '*'、それ以外なら '-'。

%s
: カレント・バッファに結び付いているサブプロセスの状態。 process-status の値と同一。 33.6 節「プロセス情報」p.565 を参照してください。

%t
: 読み込まれたファイルがテキスト・ファイルなら 'T'、バイナリなら 'B'(特定のオペレーティング・システム上でのみ意味がある区別です)。

%p
: ウィンドウの上端のバッファ中での位置 (バッファ全体に対するパーセンテージ)、または 'Top'、または 'Bottom'、または 'All'。

%P
: ウィンドウの下端のバッファ中での位置 (バッファ全体に対するパーセンテージ)、または 'Top' (バッファの先頭が見えていてバッファの末尾が見えていない場合)、または 'Bottom' (バッファの末尾は見えている場合)、または 'All' (バッファ全体が見えている場合)。

%n
: ナローイングが有効なときは 'Narrow'、それ以外のときは無 (27.4 節「ナローイング」p.467 の narrow-to-region を参照してください)。

%[
: 再帰編集レベルの表示 (ミニバッファ・レベルは含まない)。レベルが深くなるごとに '[' が一つ増える。 18.10 節「再帰編集」p.288 を参照してください。

%]
: 再帰編集レベルが深くなるごとに ']' が一つ増える (ミニバッファ・レベルは含まない)。

%%
: 文字 '%'。 %- 指定子を使用可能な場所に、文字 '%' そのものを含むことができる。

`%-` モード行のあまりを埋めるのに十分な数の '-'。

以下の二つの %- 指定子は現在でも使えますが、 obsolete です。変数 `mode-name` と `global-mode-string` を使用して同じ結果を得ることができます。

`%m` `mode-name` の値。

`%M` `global-mode-string` の値。現在では、 `display-time` のみが `global-mode-string` の値を変更します。

20.4 フック

『フック (hook)』は、既存のプログラムから呼び出される関数を、ユーザそれぞれの事情により収めておくことのできる変数です。 Emacs では、カスタマイズの目的でフックの仕組みを用意しています。フックは '`.emacs`' ファイルの中で設定されることが多いのですが、それ以外の Lisp プログラムによるフックの設定も可能です。標準的なフック変数の一覧については付録 F「標準のフック」p.671 を参照してください。

Emacs におけるフックのほとんどは『正規フック (normal hook)』です。これらのフック変数は、引数なしで呼び出される関数のリストを持ちます。ほとんどのフックが正規フックである理由は、一律な扱いが可能だからです。正規フックの名前は、必ず '`-hook`' で終えることが慣例になっています。

正規フックにフック関数を追加するお奨めの方法は、 `add-hook` を呼ぶことです (後述します)。フック関数は、 `funcall` が受けつけるものであればどのような関数であっても構いません (11.1 節「関数とは」p.147 参照)。ほとんどのフック変数の初期値は void ですが、 `add-hook` はこのような変数にも正常にフック関数を追加することができます。

もう一種類のフックである『非正規フック (abnormal hook)』には、名前が '`-function`' で終るものと '`-hooks`' で終わるものとがあります。名前が '`-function`' で終わる非正規フックは、一つの関数を値として持ちます。また、名前が '`-hooks`' で終る非正規フックは、関数のリストを値として持ちます。非正規フックは、正規フックでは不可能な場合 (たとえば関数が引数を取ったり、あるいは関数の値に意味があったりする場合) に使用されるという点で普通ではありません。名前が '`-function`' や '`-hooks`' で終るフックを使おうとするときは、それが非正規フックであることを思い出し、そのフックの説明文字列で正しい使用方法を調べるようにしてください。

主モード関数は、その初期化処理の最後に、『モード・フック (mode hook)』と呼ばれるフックを実行することが慣例になっています。これによりユーザは、モードにより設定されたローカル変数を書き換えてモードの振舞をカスタマイズすることができます。また、フックはほかの状況でも使われます。たとえば、 Emacs がサスペンドする直前に実行される `suspend-hook` というフックがあります (34.2.2 節「Emacs のサスペンド」p.582 参照)。

以下は、 Lisp Interaction モードで Auto Fill モードを起動するためにモード・フックを使う例です。

```
(add-hook 'lisp-interaction-mode-hook 'turn-on-auto-fill)
```

以下は、 Emacs が C のコードを字下げする方法をカスタマイズするフックの例です (人によってコーディング・スタイルの好みはさまざまですよね)。この例では、名前のないラムダ式をフック関数として使用しています。

```
(add-hook 'c-mode-hook
  (function (lambda ()
              (setq c-indent-level 4
                    c-argdecl-indent 0
                    c-label-offset -4
                    c-continued-statement-indent 0
                    c-brace-offset 0
                    comment-column 40))))

(setq c++-mode-hook c-mode-hook)
```

フックの実行には `run-hooks` 関数が使用されます。この関数は `add-hooks` で追加されたフック関数を呼び出します。

run-hooks &rest *hookvar* Function

この関数は一つ以上のフック変数の名前を引数に取り、それぞれのフックを順に実行します。それぞれの *hookvar* 引数はフック変数を示すシンボルでなければなりません。これらの引数は、与えられた順に実行されます。

フック変数の値が非 `nil` である場合、その値は関数のリストまたは関数でなければなりません。値が関数 (関数定義を示すシンボルまたはラムダ式) ならそれが呼ばれます。値がリストなら、それぞれの要素が順に呼ばれます。いずれの場合も、フック関数の呼出しは引数なしで行なわれます。

たとえば、 `emacs-lisp-mode` のモード・フックを実行するには以下のようにします。

```
(run-hooks 'emacs-lisp-mode-hook)
```

add-hook *hook function* &optional *append local* Function

(訳注: *local* は GNU Emacs バージョン 19.29 から追加された引数)

この関数は、関数 *function* をフック変数 *hook* に追加します。引数 *function* は、 Lisp 関数として有効で正しい数の引数を取るならどのような関数でも構いません。たとえば、

```
(add-hook 'text-mode-hook 'my-text-hook-function)
```

は、 `my-text-hook-function` を `text-mode-hook` というフックに追加します。

`add-hook` は正規フックと同様に、非正規フックにも使用できます。

フック関数は、実行順序に関わらず問題なく動くように作るのが一番です。順序に依存するのはトラブルの元になります。もっとも、通常、 *function* はフック・リストの先頭に追加されるので、さらに `add-hook` が呼ばれないかぎり最初に実行されると予想できます。

オプショナル引数の *append* が非 `nil` なら、新しいフック関数はフック・リストの最後に追加され、実行も最後になされます。

もし *local* が非 `nil` なら、フック関数をカレント・バッファにローカルにします。ただし、 `make-local-hook` (`make-local-variable` ではありません) を呼ぶことにより、フック変数がバッファローカルにされていなければなりません。もしフックがバッファローカルでなければ、フック関数は *local* の値に関わらず常にグローバルになります。

remove-hook *hook function* &optional *local* Function

(訳注: *local* は GNU Emacs バージョン 19.29 から追加された引数)

この関数は、フック変数 *hook* から *function* を取り除きます。

もし *local* が非 `nil` なら、グローバル・フック・リストからではなく、ローカル・フック・リストから *function* を取り除きます。フック変数がバッファローカルでなければ、*local* の値は意味を持ちません。

make-local-hook *hook* Function

(訳注: GNU Emacs バージョン 19.29 から追加された関数)

この関数は、フック変数 `hook` をカレント・バッファにローカルにします。ローカルなフック変数は、ローカルなフック関数とグローバルなフック関数とをもつことができ、また、 `run-hooks` はその両方を実行します。

この関数は、フック変数のローカルな値として、 `t` を要素として含むリストを設定します。この `t` は、フック変数のローカルな値とともにグローバルな値にもフック関数が収められていることを示すフラグとして働きます。 `run-hooks` はこのフラグの意味を理解できるので、 `make-local-hook` はすべての正規フックに対して使用できます。非正規フックに関しては、呼出し元が `t` の意味を理解できるように対策済みの一部のものに対してのみ使用できます。

`make-local-variable` だけではフック変数をローカルにすることはできません。フック変数に対して直接 `make-local-variable` を使用しないでください。

21 ヘルプ

GNU Emacs Lisp は、便利なオンラインヘルプ機能を備えています。ヘルプ機能で使用される情報は、ほとんどの場合、関数または変数に付属している説明文字列から得られます。この章では、 Lisp プログラムに適切な説明文字列を書く方法、説明文字列にアクセスするプログラムを書く方法について説明します。

Emacs の説明文字列は、 Emacs のマニュアルと同じものではないことに注意してください。マニュアルは、それ自身のソース・ファイルを持っており、 Texinfo 言語で記述されています。説明文字列は、対応する関数または変数の定義の中で指定されます。説明文字列を集めてもマニュアルとしては不十分です。なぜなら、よいマニュアルは、説明文字列とは異なり、トピックごとに構成されているものだからです。

21.1 説明文字列の基礎

説明文字列は文字列に対する Lisp の文法に従い、テキストの両端を二重引用符で括って記述します。説明文字列も Lisp の文字列オブジェクトの一種であるからです。関数または変数の定義の際に、適切な位置に文字列を書き込むことで、説明文字列となります。関数の定義では、説明文字列は、引数リストの次に置かれます。変数の定義の場合は、変数の初期値の後に説明文字列を記述します。

説明文字列を書く際には、最初の一行で最初の一文が終わるようにします (二つの文を含むのはかまいません)。これは、 `apropos` コマンドなど、複数行にわたる説明文字列のうち最初の一行しか表示しないコマンドがあるためです。また説明文字列の 2 行目は、字下げしないようにします。こちらは、 `C-h f` (`describe-function`) や `C-h v` (`describe-variable`) の出力がおかしくならないようにするためです。

説明文字列には、表示された時点でのキーマップに対応したキー割り当てを表す数種類の特殊文字列が使用できます。これらの特殊文字列を使うことにより、ユーザがキー割り当てを変更していても、説明文字列が正しいキーを示すようにできます。 (21.2 節「説明文字列へのアクセス」 p.340 を参照してください.)

Lisp の中では、説明文字列は、関数または変数からアクセスでき、以下のように扱われます。

- 関数の説明文字列は、定義した関数そのものの中に格納されます (11.2 節「ラムダ式」 p.148 参照)。 `documentation` 関数によって、説明文字列を取り出すことができます。
- 変数の説明文字列は、変数の属性リストの中に、 `variable-documentation` という属性名で格納されます。 `documentation-property` 関数によって、説明文字列を取り出すことができます。

スペースを節約するため、あらかじめロードされる関数と変数 (プリミティブ変数や、オートロードされる変数も含みます) の説明文字列は、 ‘`emacs/etc/DOC-`*version*’ ファイルに格納されます。 Emacs セッション中にバイトコンパイルされたファイルから読み込まれた関数と変数の説明文字列もバイトコンパイルされたファイルに格納されています (14.3 節「説明文字列とコンパイル」 p.184 参照)。

Emacs 内部のデータ構造には、説明文字列の代わりに、ファイルへ格納する場合の整数オフセットまたは文字列と整数を含むリストが含まれます。 `documentation` と `documentation-property` はそのような情報を使用して、適切なファイルから説明文字列を読み込みます。これはユーザに対しては透過です。

説明文字列の使用方法については、 *The GNU Emacs Manual* の “Help” 節を参照してください。

‘`emacs/lib-src`’ ディレクトリには、 ‘`emacs/etc/DOC-`*version*’ を美しくプリントアウトするために、 ‘`sorted-doc.c`’ と ‘`digest-doc.c`’ の二つのユーティリティがあります。

21.2 説明文字列へのアクセス

documentation-property *symbol property* &optional *verbatim* Function

この関数は、 *symbol* の属性リストに *property* 属性で記録されている説明文字列を返します。必要に応じ、文字列はファイルから読み込まれることもあります。現行のキー・バインドを反映するために、 `substitute-command-keys` を実行し、特殊文字の置換を行います (*verbatim* が非 `nil` に設定されている場合、キー・バインドの置換は行われません。 *verbatim* は、 Emacs 19 の場合にのみ存在します)。

```
(documentation-property 'command-line-processed
   'variable-documentation)
     ⇒ "t once command line has been processed"
(symbol-plist 'command-line-processed)
     ⇒ (variable-documentation 188902)
```

documentation *function* &optional *verbatim* Function

この関数は、 *function* の説明文字列を返します。必要に応じ、文字列はファイルから読み込まれることもあります。 (*verbatiom* が非 `nil` に設定されている場合をのぞき) 現行のキー・バインドを反映するために、 `substitute-command-keys` を実行します。

function に関数の定義がない場合、 `documentation` は `void-function` エラーを通知します。ただし、指定された関数に説明文字列が含まれない場合はこれは正常な動作です。この場合 `documentation` は `nil` を返します。

`documentation` と `documentation-property` の 2 つの関数を使い、いくつかのシンボルの説明文字列を ‘`*Help*`’ バッファに表示する例を、以下に示します。

```
(defun describe-symbols (pattern)
  "PATTERN に一致した Emacs Lisp シンボルの説明文字列を表示します。
名前に PATTERN を含むすべてのシンボルは、 ‘*Help*’ バッファ内に、説明
文字列が表示されます。 "
  (interactive "sDescribe symbols matching: ")
  (let ((describe-func
         (function
          (lambda (s)
```

```
            ;; シンボルの説明を印字する。
            (if (fboundp s)             ; これは関数。
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (commandp s)
                       (let ((keys (where-is-internal s)))
                         (if keys
                             (concat
                              "Keys: "
                              (mapconcat 'key-description
                                         keys " "))
                           "Keys: none"))
                     "Function")
                   (or (documentation s)
                       "not documented"))))

            (if (boundp s)              ; これは変数。
                (princ
                 (format "%s\t%s\n%s\n\n" s
                   (if (user-variable-p s)
                       "Option " "Variable")
                   (or (documentation-property
                         s 'variable-documentation)
                       "not documented")))))))
        sym-list)
    ;; パターンに一致するシンボルのリストを作る。
    (mapatoms (function
               (lambda (sym)
                 (if (string-match pattern (symbol-name sym))
                     (setq sym-list (cons sym sym-list))))))
    ;; データの表示。
    (with-output-to-temp-buffer "*Help*"
      (mapcar describe-func (sort sym-list 'string<))
      (print-help-return-message))))
```

describe-symbols 関数は、 apropos 関数と同じように働きますが、より多くの情報を提供します。

```
(describe-symbols "goal")

---------- Buffer: *Help* ----------
goal-column     Option
*Semipermanent goal column for vertical motion, as set by C-x C-n, or nil.

set-goal-column Command: C-x C-n
Set the current horizontal position as a goal for C-n and C-p.
Those commands will move to this position in the line moved to
rather than trying to keep the same horizontal position.
With a non-nil argument, clears out the goal column
so that C-n and C-p resume vertical motion.
The goal column is stored in the variable ‘goal-column’.

temporary-goal-column   Variable
Current goal column for vertical motion.
It is the column where point was
at the start of current run of vertical motion commands.
When the ‘track-eol’ feature is doing its job, the value is 9999.
---------- Buffer: *Help* ----------
```

Snarf-documentation *filename* Function

この関数は、実行可能な Emacs がダンプされる直前の、 Emacs の初期化中にのみ使用されます。この関数は、 *filename* に含まれる説明文字列のファイルオフセットを探し、実際の文字列の代わりに内部の関数定義および変数属性リストに記録します。

Emacs は、 *filename* ファイルを '`emacs/etc`' ディレクトリで検索します。後にダンプされた Emacs が実行されると、同じファイルは、 `doc-directory` ディレクトリで見つかります。通常 *filename* は、 `"DOC-`*version*`"`です。

doc-directory Variable

この変数は、組み込み関数と変数、プリロードされた関数と変数の説明文字列を含む `"DOC-`*version*`"`ファイルが存在するディレクトリの名前を保持します。

ほとんどの場合、この変数の内容は `data-directory` と同一です。 Emacs を実際にインストールせずに、構築したディレクトリで実行しようとした場合は、異なることがあります。 21.5 節「ヘルプ関数」 p.344 の `data-directory` を参照してください。

古いバージョンの Emacs では、 `exec-directory` がこの用途に使われていました。

21.3 説明文字列でのキー・バインドの置換

説明文字列がキー・シーケンスを参照する場合、現在の実際に使用されているキー・バインドを使用する必要があります。これを行うには、以下のような特殊なテキスト・シーケンスを用います。通常の方法で説明文字列をアクセスすると、これらのシーケンスは、現行のキー・バインド情報で置き換えられます。これは、 `substitute-command-keys` を呼ぶことにより行われます。この関数は、明示的に呼ぶこともできます。

以下に特殊シーケンスとその意味のリストを示します。

`\[`*command*`]`
: *command* を起動するキー・シーケンスを表します。 *command* にキー・シーケンスが割り当てられていない場合は、 '`M-x` *command*' となります。

`\{`*mapvar*`}`
: *mapvar* の値のサマリを表します。 *mapvar* はキーマップでなければなりません。サマリは、 `describe-bindings` で作成されます。

`\`<*mapvar*>
: 直接テキストは表しません。副作用として、説明文字列であとに続く '`\[`*command*`]`' シーケンスのキーマップとして *mapvar* を指定します。

注意: '`\`' は Emacs Lisp 内に記述する場合、それぞれ二つ重ねて書く必要があります。

substitute-command-keys *string* Function

この関数は、 *string* をスキャンし、上記の特殊シーケンスを指定されているものに置き換え、結果を文字列として返します。この関数により、ユーザがキー・バインドを変更している場合でも、正しいバインドを参照した説明文字列を表示できます。

特殊シーケンスの使用例を示します。

```
(substitute-command-keys
   "To abort recursive edit, type: \\[abort-recursive-edit]")
⇒ "To abort recursive edit, type: C-]"
(substitute-command-keys
   "The keys that are defined for the minibuffer here are:
  \\{minibuffer-local-must-match-map}")
⇒ "The keys that are defined for the minibuffer here are:
?               minibuffer-completion-help
SPC             minibuffer-complete-word
TAB             minibuffer-complete
LFD             minibuffer-complete-and-exit
RET             minibuffer-complete-and-exit
C-g             abort-recursive-edit
"

(substitute-command-keys
   "To abort a recursive edit from the minibuffer, type\
\\<minibuffer-local-must-match-map>\\[abort-recursive-edit].")
⇒ "To abort a recursive edit from the minibuffer, type C-g."
```

21.4 ヘルプ・メッセージでの文字の表現

この関数は、イベント、キー・シーケンス、キー文字を文字列での表現に変換します。非印字文字や空白文字を通常の印字文字での表現に変換するため、任意のテキスト文字、キー・シーケンスをメッセージ内で使う場合に有用です。非印字文字や空白文字以外の通常の文字の表現は、その文字自身となります。

key-description *sequence* Function

この関数は、 *sequence* に含まれる入力イベントの Emacs 標準表記を含む文字列を返します。引数の *sequence* には、文字列、ベクタ、リストを指定できます。有効なイベントについては、 18.5 節「入力イベント」 p.266 を参照してくださいを参照してください。また以下の `single-key-description` の例も参照してください。

single-key-description *event* Function

この関数は、キーボード入力の Emacs 標準表記の中の *event* を説明する文字列を返します。通常の印字可能文字は文字自身、制御文字は ‘`C-`’ で始まる文字列、メタ文字は ‘`M-`’ で始まる文字列、スペース、改行などはそれぞれ、 ‘`SPC`’、 ‘`LFD`’ のように表されます。関数キー・シンボルは、そのまま表示されます。リストであるイベントは、リストの CAR の中のシンボルの名前として表示されます。

```
(single-key-description ?\C-x)
     ⇒ "C-x"
(key-description "\C-x \M-y \n \t \r \f123")
     ⇒ "C-x SPC M-y SPC LFD SPC TAB SPC RET SPC C-l 1 2 3"
(single-key-description 'C-mouse-1)
     ⇒ "C-mouse-1"
```

text-char-description *character* Function

この関数は、テキストに現れる文字列の Emacs 標準表記の中で *character* を示す文字列を返します。 `single-key-description` 関数と同じように働きますが、制御文字がキャ

レットに続く文字列で表される点が異なります (Emacs のバッファ内では、制御文字は通常このように表されます)。

```
(text-char-description ?\C-c)
     ⇒ "^C"
(text-char-description ?\M-m)
     ⇒ "M-m"
(text-char-description ?\C-\M-m)
     ⇒ "M-^M"
```

21.5 ヘルプ関数

Emacs は、いろいろなオンライン・ヘルプ機能を提供しており、ユーザはそのすべての機能を、 `C-h` 前置キーのサブコマンドとして使用できます。詳しくは、 *The GNU Emacs Manual* の “Help” 節を参照してください。ここでは、これらの情報へのプログラムレベルでのインタフェースについて説明します。

apropos *regexp* &optional *do-all predicate* Command

この関数は、正規表現 *regexp* に一致するシンボル名を全て検索し、そのリストを返します (30.2 節「正規表現」 p.525 参照)。また、 ‘`*Help*`’ という名前のバッファ内にそのシンボルを一行の説明とともに表示します。

do-all が非 `nil` の場合は、 `apropos` は検索された関数のキーバインドも表示します。

predicate が非 `nil` の場合、 *regexp* に一致したシンボルに対して、この変数に設定された関数が呼ばれます。このとき、 *predicate* が非 `nil` 値を返したシンボルのみがリストまたは表示されます。

以下の例のうち最初のものでは、 `apropos` は ‘`exec`’ を含む名前を持つ全てのシンボルを検索します。 2 番目の例では、コマンドであるシンボルのみを検索し返します (‘`*Help*`’ バッファに表示される結果は示していません)。

```
(apropos "exec")
     ⇒ (Buffer-menu-execute command-execute exec-directory
    exec-path execute-extended-command execute-kbd-macro
    executing-kbd-macro executing-macro)

(apropos "exec" nil 'commandp)
     ⇒ (Buffer-menu-execute execute-extended-command)
```

`C-h a`(`command-apropos`) コマンドは、 `apropos` を呼び出します。ただし *predicate* を指定して、コマンドであるシンボルのみを出力します。 `apropos` の呼び出しは、以下のようになります。

```
(apropos string t 'commandp)
```

super-apropos *regexp* &optional *do-all* Command

この関数は `apropos` と似た動作をしますが、 *regexp* に一致するシンボル名以外に説明文字列も検索する点で異なります。デフォルトでは、プリロードされた関数と変数の説明文字列のみを検索します。 *do-all* が非 `nil` に設定されている場合は、すべての関数と変数の名前と説明文字列を検索します。

help-map Variable

この変数の値は、ヘルプ・キー (C-h) に続く文字のローカル・キー・マップです。

help-command Prefix Command

このシンボルは関数ではありません。この関数の定義は、実際には help-map というキー・マップです。これは、'help.el' の中で、以下のように定義されています。

```
(define-key global-map "\C-h" 'help-command)
(fset 'help-command help-map)
```

print-help-return-message &optional *function* Function

この関数は、ヘルプコマンドが呼ばれた後でもとのウィンドウの状態に復帰させるための情報を記述した文字列を作成します。 *function* が非 nil の場合、メッセージを作成した後、メッセージに対して指定された関数を使用します。 nil の場合は、 message を呼び出してメッセージをエコー領域に表示します。

この関数は、 with-output-to-temp-buffer 特殊フォーム内で使用され、 standard-output は、この特殊フォームで指定された値をもつことを前提としています。使用方法については、 21.2 節「説明文字列へのアクセス」 p.340 の中の長い例を参照してください。

help-char Variable

この変数は、ヘルプ文字の値を示します。ヘルプ文字は、 Emacs がヘルプと認識する文字で、デフォルトでは 8、すなわち C-h となっています。 Emacs がこの文字を読み込むと、 help-form が非 nil な Lisp 式に設定されている場合、式を評価し、結果が文字列の場合はそれを表示します。

通常は、 help-form の値は、 nil です。そのためヘルプ文字は、コマンド入力レベルでは何ら特殊な意味を持たず、通常のキー・シーケンスの一部となります。 C-h は、標準のキー・バインドでは、いくつかの汎用ヘルプ機能の前置キーに割り当てられています。

ヘルプ文字は、前置キーの後でも特殊文字として扱われます。プレフィックス・キーのサブコマンドにバインドされていない場合は、 describe-prefix-bidings を実行し、サブコマンドのリストを表示します。

help-form Variable

この変数が非 nil の場合、設定された値は、 help-char 文字が読み込まれるたびに評価に使用されるフォームとして使われます。フォームの評価により文字列が得られる場合は、その文字列が表示されます。

read-event または read-char を呼び出すコマンドは、入力を行う場合、おそらく help-form を非 nil に束縛する必要があります (C-h が意味のある入力の場合は例外です)。この式を評価した結果は、入力の目的、入力の方法などを適切に説明した文字列とならなければなりません。

ミニバッファへ入ると、この変数は minibuffer-help-form の値に束縛されます (17.8 節「ミニバッファそのほか」 p.254 参照)。

prefix-help-command Variable

この変数は、前置文字のヘルプを表示する関数を保持します。ユーザが前置文字に続いてヘルプ文字を入力した場合、その前置文字のヘルプ文字がバインドされていなければ、この関数が呼ばれれます。この変数のデフォルト値は、 `describe-prefix-bindings` です。

describe-prefix-bindings Function

この関数は、 `describe-bindings` を呼び出し、最新のキー・シーケンスの前置キーの全てのサブコマンドのリストを表示します。記述される前置キーは、キー・シーケンスの最終のイベントを除くすべてのイベントとなります。(通常、最後のイベントはヘルプ文字です)。

以下の 2 つの関数は、 '`helper`' ライブラリに含まれます。これらの関数は、 "electric" モードなど、そのモード内でヘルプを提供するようなモードのためにあります。使用する場合は、`(require 'helper)` を使用してライブラリをロードする必要があります。これらの関数は、通常のヘルプ関数と異なり、関数名が '`Helper`' から始まります。

Helper-describe-bindings Command

このコマンドはウィンドウをポップアップし、ローカルおよびグローバルキーマップのすべてのキー・バインドのリストを含むヘルプ・バッファを表示します。この関数は、`describe-bindings` を呼び出して動作します。

Helper-help Command

このコマンドは、現在のモードのヘルプを提供します。ミニバッファに '`Help (Type ? for further options)`' というメッセージを表示して、ユーザに入力を求め、モードの目的、キー・バインドなどを調べる手助けをします。この関数は、 `nil` を返します。

これは、 `Helper-help-map` マップを変更することにより、カスタマイズできます。

data-directory Variable

この変数は、 Emacs に添付されている説明ファイル、テキスト・ファイルを Emacs が検索するディレクトリの名前を保持します。古いバージョンの Emacs では、 `exec-directory` がこの用途に使用されていました。

make-help-screen *fname help-line help-text help-map* Macro

このマクロは、付随するサブコマンドのリストを表示する前置キーと同様な動作をする *fname* というヘルプ・コマンドを定義します。

起動されると、 *fname* はウィンドウに *help-text* を表示し、 *help-map* にしたがってキー・シーケンスを読み込み、実行します。 *help-text* 文字列は、 *help-map* で使用できるバインドについての説明でなければなりません。

fname コマンドは、 *help-text* の表示をスクロールすることにより、それ自身でいくつかのイベントを処理するよう定義されています。 *fname* が、それらの特殊イベントの一つを読み込むと、スクロールを行ってから、次のイベントを読み込みます。 *help-map* の中にバインドがあるいくつかのイベント以外のイベントを読み込んだ場合は、そのキーにバインドされたコマンドを実行し、返ります。

help-line 引数は、 *help-map* の中の代替の一行サマリでなければなりません。現行のバージョンの Emacs では、この引数は、 `three-step-help` オプションが `t` に設定されている場合にのみ使用されます。

three-step-help User Option

この変数が非 `nil` の場合、 `make-help-screen` と共に定義されたコマンドは、 *help-line* 文字列をまずエコー領域に表示し、続けてヘルプ文字がタイプされた場合にのみ、長い *help-text* 文字列を表示します。

22 ファイル

Emacs では、ファイルとファイル・ディレクトリに対し、探索する、新規作成する、ながめる、保存する、などの作業ができます。本章では Emacs lisp のファイルに関係する関数のうちの大部分のものを説明しますが、第 24 章「バッファ」p.393 で説明しているのもあります。また、バックアップや自動セーブに関連する関数は、第 23 章「バックアップと自動セーブ」p.383 で説明しています。

ファイル関数の多くは、引数として一つないしはそれ以上の数のファイル名を取ります。ファイル名とは、実際には文字列のことです。これらの関数のほとんどは、 `expand-file-name` を使ってファイル名を展開します。これにより相対ファイル名 (‘`../`’ を含む) を取り扱うように、 ‘`~`’ を正しく取り扱います。これらの関数は ‘`$HOME`’ などの環境変数の置換は行ないません。 22.8.4 節「ファイル名を展開する関数」p.370 を参照してください。

22.1 ファイルを visit する

ファイルを visit するとは、バッファにファイルを読み込むことを意味します。 visit を一度した後は、バッファがそのファイルを『visit している (visiting)』といい、そのファイルを、バッファにより “visit されているファイル” と呼びます。

ファイルとバッファは二つの異なったものです。ファイルはコンピュータの中に (消さないかぎりは) ずっと記録されている情報です。一方バッファは、 Emacs の内部的な情報であり、Emacs の編集のセッションが終ったとき (あるいはバッファを kill したとき) にはなくなってしまうものです。通常、バッファはあなたがファイルからコピーした情報を持っています。このとき、そのバッファはファイルを visit しているといいます。バッファ中のコピーに対して、あなたは編集のコマンドを使って編集をします。そのようなバッファに対しての修正は、ファイルには反映されません。ですから、その変更点をファイルに反映させるには、バッファを『保存 (save)』しなければなりません。それは変更されたバッファの内容を、ファイルにコピーしなおすことを意味します。

ファイルとバッファには以上のような違いがあるのですが、バッファを意味しているときにファイルのことを言及してみたり、またこの逆の場合があったりということがあります。実際、“私はバッファを編集していて、すぐに同じ名前をもつファイルとして保存します” という言い方よりはむしろ、 “私はファイルを編集しています” という言い方をします。人は通常この区別を明示的にする必要はありません。しかし、コンピュータ・プログラム中でファイル、バッファを扱うときは、この区別を念頭に置くとよいでしょう。

22.1.1 ファイルを visit するための関数

本節では、ファイルを visit するのに通常使われる関数について説明します。歴史的な理由により、これらの関数は ‘`visit-`’ で始まる名前ではなく、 ‘`find-`’ で始まる名前を持ちます。バッファが visit しているファイルの名前にアクセスする、あるいは、ファイルの名前から存在するバッファをみつける関数と変数については 24.4 節「バッファ・ファイル名」p.397 を参照してください。

Lisp プログラムで、ファイルの中身は見たいが、変更はしたくない場合、最も手っ取り早い方法は、一時バッファで `insert-file-contents` を使うことです。ファイルを visit する必要はなく、 visit する方が時間もかかります。 22.3 節「ファイルからの読込み」p.355 を参照してください。

find-file *filename* Command

このコマンドは、ファイル *filename* を visit するバッファを選択します。これは、そのようなバッファがある場合には、それを使用し、もしなければ新しいバッファを作り、そのファイルを読み込みます。またこのコマンドは、そのバッファを返値として返します。

`find-file` の本体はとても単純で、以下のようになっています。

```
(switch-to-buffer (find-file-noselect filename))
```

(25.7 節「ウィンドウ内でのバッファの表示」p.416 の `switch-to-buffer` を参照。)

`find-file` が対話的に呼ばれたときは、ミニバッファにおいて *filename* の入力を促します。

find-file-noselect *filename* Function

この関数は、ファイルを visit する関数すべてのもととなるものです。これは、ファイル *filename* を visit するバッファをみつけるか、あるいは新しく作成して、そのバッファを返します。この関数は、そのバッファがすでに存在するものであれば、それを使い、そうでなければ新しいバッファを作って、そこにファイルを読み込みます。あなたが望めば、そのバッファをカレント・バッファにできるし、またウィンドウにそのバッファを表示できますが、この関数自身はそれはしません。

`find-file-noselect` が、すでに存在するバッファを使用するときは、まず最初に、そのファイルがそのバッファにおいて最後に visit、あるいは保存されてから、変更がないことを確認します。変更がある場合には、この関数は、ユーザに対し、その変更前の元のファイルを読み込み直すかどうかを尋ねます。ユーザが、この質問に ‘`yes`’ と答えた場合、それまでにそのバッファにしたすべての変更は失われます。

もし `find-file-noselect` が、バッファを新規に作成する必要があり、かつ *filename* という名前のファイルが存在しない場合、エコー領域に ‘`New file`’ というメッセージを表示し、バッファの内容を空のままにします。

`find-file-noselect` 関数は、ファイルを読み込んだ後、 `after-find-file` を呼び出します (22.1.2 節「visit のためのサブルーチン」p.351 参照)。この関数は、バッファの主モードを設定し、ローカル変数を構文解析し、今現在 visit しているファイルよりも、新しく自動セーブされたファイルが存在する場合には警告をユーザに出し、最後に、`find-file-hooks` の中の関数を起動します。

`find-file-noselect` 関数は、ファイル *filename* を visit しているバッファを返値として返します。

```
(find-file-noselect "/etc/fstab")
     ⇒ #<buffer fstab>
```

find-file-other-window *filename* Command

このコマンドはファイル *filename* を visit するバッファを選択しますが、現在選択されているウィンドウ以外のウィンドウ内で選択します。このコマンドは、ほかの存在するウィンドウを使用するかもしれませんし、一つのウィンドウを分割するかもしれません。 25.7 節「ウィンドウ内でのバッファの表示」p.416 を参照のこと。

このコマンドが対話的に呼ばれた場合、 *filename* の入力をユーザに促します。

find-file-read-only *filename* Command

このコマンドは、 `find-file` と同様、ファイル *filename* を visit するバッファを選択しますが、この関数はバッファを読出し専用とします。関連する関数および変数は、 24.7 節「読みだし専用バッファ」p.401 を参照してください。

このコマンドが対話的に呼ばれた場合、 *filename* の入力をユーザに促します。

view-file *filename* Command

このコマンドは View モードで *filename* を visit し、このコマンドの終了後には、前に存在したバッファに戻るように、再帰編集の状態でファイルを表示します。 View モードはファイルをざっと眺めるためのモードで、編集はさせません。 View モードに入ると、正規フック `view-mode-hook` を起動します。 20.4 節「フック」p.335 を参照してください。

`view-file` が対話的に呼ばれた場合、 *filename* の入力をユーザに促します。

find-file-hooks Variable

この変数の値は、ファイルが visit されたあとに呼ばれる関数のリストです。(もしあるなら) ファイルのローカル変数の仕様は、フックを走らせる前に処理されます。このフック関数が起動されるとき、ファイルを visit しているバッファが、カレント・バッファです。

この変数は、正規フックのように動作しますが、しかしこの変数名を変えてしまうのは、おすすめできません。

find-file-not-found-hooks Variable

この変数の値は、 `find-file` ないしは `find-file-noselect` が存在しないファイルに対して呼ばれたとき、呼び出される関数のリストです。 `find-file-noselect` は、ファイルが存在しないことを確認すると、すぐにこれらの関数を呼び出します。この関数は、リストの要素を、順に非 `nil` を返す関数があるまで呼ばれます。 `buffer-file-name` は、すでに設定されています。

関数の返値が使用され、その関数すべてが呼び出されないかもしれないので、これは正規フックではありません。

22.1.2 visit のためのサブルーチン

`find-file-noselect` 関数は、 `create-file-buffer` と `after-find-file` 関数をサブルーチンとして使います。それらを直接的に呼び出すことは、時には便利な場合もあります。

create-file-buffer *filename* Function

この関数は、 *filename* を visit するために、適切に名づけられたバッファを作成し、そのバッファを返値として返します。この関数は、前にバッファ名として使われたものでないなら、(ディレクトリ部分なしで)*filename* をそのままバッファの名前として使用します。そうでなければ、'<2>' のような文字列をくっつけて、前に使用されていない名前を作り出します。 24.9 節「バッファの生成」 p.403 も参照のこと。

注意: `create-file-buffer` は、ファイルとその作成された新しいバッファを関係づけるものではありません。またバッファの選択も行ないません。またこの関数は、デフォルトの主モードを使いません。

```
(create-file-buffer "foo")
     ⇒ #<buffer foo>
(create-file-buffer "foo")
     ⇒ #<buffer foo<2>>
(create-file-buffer "foo")
     ⇒ #<buffer foo<3>>
```

この関数は、 `find-file-noselect` によって使われます。この関数は、 `generate-new-buffer` を使います。 (24.9 節「バッファの生成」 p.403 参照)

after-find-file &optional *error warn* Function

この関数は、バッファの主モードを設定し、ローカル変数を構文解析します (20.1.3 節「主モードの自動選択の仕組み」 p.323 参照)。この関数は、 `find-file-noselect` と、デフォルトの復帰の関数より呼ばれます (23.3 節「復帰」 p.391 参照)。

ファイル読み込み時に、ファイルは存在しないが、ディレクトリは存在してエラーを引き起こした場合には、呼び出し側は *error* に非 `nil` 値をわたすべきです。この場合、`after-find-file` は、警告のメッセージを出します。 '`(New File)`'。より深刻なエラーの場合には、呼び出し側は、 `after-find-file` を呼び出すべきではありません。

warn が非 `nil` の場合、この関数は、自動セーブ・ファイルが存在し、それが visit されるファイルより新しく作成されたものならば、警告のメッセージを出します。

最後に、 `after-find-file` は、 `find-file-hooks` 中のすべての関数を呼び出します。

22.2 バッファを保存する

あなたが Emacs でファイルを編集しているときは、実際は、そのファイルを visit しているバッファに対して作業を行っています。すなわち、そのファイルの中味がバッファにコピーされ、そのコピーに対して編集作業をしているのです。そのバッファの内容に対する変更は、あなたがバッファを『保存 (save)』するまで実際のファイルに反映されません。この保存というのは、バッファの内容をファイルに対してコピーするということです。

save-buffer &optional *backup-option* Command

この関数は、カレント・バッファの内容が、最後に visit あるいは保存されてから修正されている場合に、そのバッファの内容を visit されているファイルに保存します。そうでない場合はなにもしません。

save-buffer は、バックアップ・ファイルの作成に関して責任を持ちます。通常、*backup-option* は nil であり、ファイルを visit してから最初に保存するときのみ、save-buffer はバックアップ・ファイルを作成します。 *backup-option* にほかの値を与えた場合、それはほかの状態でバックアップ・ファイルの作成をするよう要求したことになります。

- 一回ないしは三回 C-u を押して、それに対応して与えられる引数が 4 あるいは 64 になる場合、 save-buffer 関数は、そのバッファが次に保存されたときに、ファイルのこのバージョンのバックアップをとるように記録します。
- 二回、ないしは三回 C-u を押して、それに対応して与えられる引数が 16 ないしは 64 になる場合、この関数は、そのファイルを保存する前に、前のバージョンのバックアップを無条件でとります。

save-some-buffer &optioanl *save-silently-p exiting* Command

このコマンドはファイルを visit しているバッファで修正されたものの複数のものを対象にして保存を行ないます。通常、このコマンドはユーザに対し、それぞれのバッファに対して確認を求めます。しかし、 *save-silently-p* が非 nil である場合には、ファイルを visit しているすべてのバッファを、ユーザに確認をせず保存します。

オプションの *exiting* 引数は、非 nil の場合、ファイルを visit していないほかのバッファも保存するように、この関数に要求します。これらの対象となるバッファは、buffer-offer-save の値として、非 nil であるローカル値をもつバッファです (これらの対象バッファに対して、保存するかどうかの質問に yes と答えた場合、ファイル名を聞かれます)。 save-buffers-kill-emacs 関数は、この引数に対して非 nil の値を渡します。

buffer-offer-save Variable

Emacs は、バッファでこの変数が非 nil のとき、そのバッファがファイルを visit していない場合でも、 Emacs 終了時に保存をするような機能を提供します。この変数は、自動的にすべてのバッファでローカルな変数です。通常、 Mail モード (送信するメールを編集するのに使われます) ではこの変数を t に設定します。

write-file *filename* Command

この関数は、カレント・バッファをファイル *filename* に書き込み、バッファがそのファイルを visit しているものと設定し、そのバッファを修正がないものとして記録します。そのとき、バッファの名前を唯一なものにする必要があれば、バッファの名前を *filename* をベースとして、 ‘<2>’ などの文字列をあとに付加して名前を変えます。この関数は、この作業の大部分を set-visited-file-name、 save-buffer を呼び出して行ないます。

write-file-hooks Variable

この変数の値は、バッファの内容を visit されているファイルに書き込む前に呼び出される関数のリストです。そのリストの要素の一つが、非 nil を返した場合、そのファイルをすでに書込みがされたものとして扱い、リストの残りの要素は呼び出さず、またファイル書込みのために通常使われるコードは実行しません。

write-file-hooks 中の関数が非 nil を返した場合、その関数はバックアップ・ファイルの作成に (もしそれが適切なものならば) 責任を持ちます。これを実行するために、以下のコードを実行してください。

```
(or buffer-backed-up (backup-buffer))
```

あなたは backup-buffer が返すファイル・モードの値を保存したいかもしれません。そして、あなたが書き込むファイルに対し、モード・ビットを設定するために、それを使いたいかもしれません。これは save-buffer が通常行っていることです。

これは正規フックではないのですが、このリストを取り扱うのに add-hook と remove-hook を使うことができます。 20.4 節「フック」 p.335 を参照してください。

local-write-file-hooks Variable

これはちょうど write-file-hooks のように働きます。しかしこれは特定のバッファにローカルなことをさせるためのフックです。バッファローカルなことをするのに write-file-hooks を使うのはよい考えではありません。その代わりにこの変数を使ってください。

この変数はつねにバッファローカルなものですので、主モードを変えても、バッファローカルな値は変わりません。この変数は、特別な方法で “ファイル” の中味を読み込み、それに対応する方法でそのデータを保存するためのフックをセットするようなパッケージにとって便利です。

write-contents-hooks Variable

これはちょうど write-file-hooks のように働きます。しかしこのフックは、ファイルがどういう属性のものかということに関係するフックに対応させて、ファイルの中身に関係するものとして使われることを意識されたフックです。このようなフックは、通常主モードが設定を行ない、バッファローカルな束縛をします。新しい主モードに変わるときは、つねにこの変数をリセットします。

after-save-hook Variable

この正規フックは、バッファの内容がその visit されているファイルに保存されたあとに起動されるフックです。

file-precious-flag Variable

この変数が非 nil の場合、 save-buffer は、そのファイルがもつことになっていた名前の代わりに一時的な新しい名前で保存し、そしてエラーがないことがわかった後に、予定していた名前にそのファイルの名前替えをする、という処理の間に発生する I/O エラーからファイルを保護します。この手続きは、不適切なファイルで、ディスク空間が不足するような場合に発生する問題点を回避します。

副作用として、バックアップは必ずコピーで実行されます。 23.1.2 節「バックアップの手段は名前の変更? それともコピー?」 p.384 を参照してください。また同時に、貴重なファイルを保存することは、その保存するファイルとそのほかの別名の間のハードリンクをつねに壊すことになります。

いくつかのモードは、この変数に対して特定のバッファでローカルに非 nil を設定します。

require-final-newline User Option

この変数は、改行でファイルが終わらないようにファイルを書き込むかどうかを決めるものです。もし変数の値が `t` ならば、 `save-buffer` は保存されるバッファがそのときに改行で終わってないときは、常にそのファイルの終わりに改行を確認なしに加えます。 `save-buffer` は、もしこの変数の値が非 `nil` であるが `t` でない場合、改行を加えるような場合が発生するごとに、ユーザに確認を求めます。

もしその変数の値が `nil` ならば、 `save-buffer` は改行を全く加えません。 `nil` がこの変数に対するデフォルトの値ですが、主モードの中には、特定のバッファで `t` を設定するものもあります。

22.3 ファイルからの読込み

`insert-file-contents` 関数を使って、ディスクからファイルをコピーし、バッファのなかにそのファイルを挿入することができます。ユーザレベル・コマンド `insert-file` は、内部で印づけをするため、 Lisp プログラムの中では使用してはいけません。

insert-file-contents *filename* &optional *visit beg end replace* Function

この関数は、カレント・バッファのポイントの後にファイル *filename* の内容を挿入します。この関数は、そのファイルの絶対ファイル名と、挿入されたデータ長のリストを返します。 *filename* が読み出すことのできるファイルの名前でない場合は、エラーを発生します。

関数 `insert-file-contents` は、このファイルの内容に対して、定義されたファイル・フォーマットでチェックを行ないます。それが適切ならば、ファイルの内容を変換します。 22.12 節「ファイル・フォーマットの変換」 p.377 を参照してください。またこの関数は、リスト `after-insert-file-functions` の関数を呼び出します。 29.18.7 節「テキスト属性をファイルへセーブする」 p.517 を参照のこと。

もし *visit* が非 `nil` ならば、この関数はさらに、バッファを修正がないものとしてマークし、バッファ中のさまざまなフィールドをセットして、ファイル *filename* を visit しているように設定します。これらのフィールドとしては、バッファの visit されているファイル名、最後に保存されたときのファイルの修正時間などの情報があります。この特徴は、 `find-file-noselect` が使用し、ほとんどの場合においては、ユーザ自身からは使用すべきではありません。

もし *beg* と *end* が非 `nil` ならば、これらは、挿入されるファイルの部分を決める整数です。この場合、 *visit* は `nil` でなければなりません。たとえば、

```
(insert-file-contents filename nil 0 500)
```

はファイルの最初の 500 文字を挿入します。

もし引数 *replace* が非 `nil` ならば、バッファの内容 (実際はアクセス可能な部分のみ) をファイルの内容で置き換えることを意味します。これは、単純にバッファの内容を削除

し、ファイルの全内容を挿入する、という方法よりよい方法です。というのはこの関数は、 (1) いくつかのマーカの位置を保存する、 (2)undo リストに入るデータの量が少ない、という利点があるからです。

もしほかのプログラムがそのファイルを読むことができるように、ほかのプロセスにファイル名を渡したいならば、関数 `file-local-copy` を使用してください。 22.11 節「ファイル名を“魔力のあるもの”にする」p.375 を参照のこと。

22.4 ファイルへの書込み

`append-to-file` と `write-region` 関数を使用して、バッファの内容全部ないしは一部を、直接ディスク上のファイルとして書き込むことができます。これらの関数を、現在 visit しているファイルに書込みを行なうのに使用してはいけません。使用した場合、 visit のメカニズムに混乱を引き起こす可能性があります。

append-to-file *start end filename* Command

この関数は、カレント・バッファ中の *start* と *end* で囲まれたリージョンの内容を、 *filename* の終わりに追加書き込みします。もしそのようなファイルが存在しない場合、新規に作成されます。この関数は `nil` を返します。

もし *filename* が、書き込み可能でないファイル、ないしはファイル作成ができないディレクトリの中の存在しないファイルをさしていた場合、エラーを発生します。

write-region *start end filename* &optional *append visit* Command

この関数は、カレント・バッファの *start* と *end* で囲まれたリージョンを、 *filename* で指定されたファイルに書き込みます。

もし *start* が文字列ならば、 `write-region` はバッファからのテキストでなく、その文字列自身を書き込み、ないしは追加書き込みします。

もし *apeend* が非 `nil` ならば、その指定されたテキストは (もしあるならば) 存在するファイルの内容に追加書き込みされます。

もし *visit* が `t` ならば、 Emacs はバッファとファイルとの間に関連性を確立します。そのときこのバッファは、そのファイルを visit していることになります。この関数はまた、 *filename* の修正時刻に、カレント・バッファの最後のファイル修正の時刻をセットします。そしてそのバッファが修正がないものとして記録します。この特徴は `save-buffer` が使用しますが、ユーザはほとんどの場合この特徴を使用すべきではありません。

もし *visit* が文字列ならば、これは visit するファイル名を指定します。これにより、データをあるファイル (*filename*) に書き込み、一方でバッファは、別のファイル (*visit*) を visit しているとして記録することができます。引数 *visit* はエコー領域メッセージに使用され、またファイル・ロックのためも使われます。 *visit* は `buffer-file-name` に記録されます。この特徴は、 `file-precious-flag` を実装するのに使用されます。ユーザは本当に自分が何をしているのかが知りたいのでなければ、この特徴は使用してはいけません。

関数 `write-region` は、 `buffer-file-format` で指定されている場合に、適切なファイル・フォーマットにそのデータを変換します。 22.12 節「ファイル・フォーマットの変換」 p.377 を参照してください。またこの関数は、リスト `write-region-annotate-functions` の関数を呼び出します。 29.18.7 節「テキスト属性をファイルへセーブする」 p.517 を参照のこと。

通常、 `write-region` は、メッセージ '`Wrote file` *filename*' をエコー領域に表示します。もし、 *visit* が `t` あるいは `nil` あるいは文字列のどれでもないならば、このメッセージの表示はされません。この特徴は、内部の目的のためで、ユーザがそれについて知る必要がないようなファイルを使うプログラムを作成する場合に便利です。

22.5 ファイルのロック

二人のユーザが同時に同じファイルを編集する場合、お互いに干渉してしまう可能性があります。 Emacs は、ファイルが修正されているときに、『ファイル・ロック (file lock)』を記録して、このような状況が発生するのを回避しようとします。これにより Emacs は、ほかの Emacs ジョブによりロックされているファイルがバッファに visit されて修正されようとするのを、その最初の時点で検出することができ、ユーザにどのようにするか確認を求めます。

ファイル・ロックは、 NFS のように、複数のマシンがファイル・システムを共有している場合には正しく機能しません。おそらく将来にはよりよいファイル・ロック・システムが実装されることでしょう。ファイル・ロックが機能しないときには、二人のユーザにより、同時に一つのファイルを修正をすることは可能にはなってしまうのですが、 Emacs はこの場合も、 2 番目に保存をしようとするユーザに注意を促します。また、そのファイルがディスク上で変更されている場合に、その元ファイルを visit しているバッファが修正されているのを検出する機構は、同時編集のケースを発見することができる場合があります。 24.6 節「変更時間の比較」 p.399 を参照。

file-locked-p *filename* Function

この関数は、ファイル *filename* がこの Emacs プロセスによりロックされていない場合には `nil` を返します。この Emacs によってロックされている場合は、 `t` を返し、ほかのユーザによってロックされている場合には、そのユーザの名前を返します。

```
(file-locked-p "foo")
     ⇒ nil
```

lock-buffer &optional *filename* Function

この関数は、カレント・バッファが修正されている場合には、ファイル *filename* をロックします。この引数 *filename* は、カレント・バッファの visit しているファイルがデフォルトです。もしカレント・バッファが、ファイルを visit していない、あるいは修正がない場合には、なにもしません。

unlock-buffer Function

この関数は、カレント・バッファが修正されている場合に、そのバッファで visit しているファイルをアンロックします。バッファが修正されていない場合には、その対応する

ファイルはロックされるべきではないので、この関数はなにもしません。カレント・バッファがファイルを visit していない場合もなにもしません。

ask-user-about-lock *file other-user* Function

この関数は、ユーザが *file* を修正しようとしたが、ほかの *other-user* という名前の別のユーザによりロックされている場合に呼ばれます。この関数が返す値により、次に何が起こるかが決まります。

- `t` の場合、ファイルのロックを獲得したということを意味します。このとき、このユーザはファイルを編集するかもしれません。 *other-user* はロックを失います。
- `nil` の場合、ロックを無視し、このユーザにそのファイルを何らかの方法で編集させる、ということを意味します。
- この関数は、代わりに `file-locked` エラーを発生するかもしれません。この場合は、ユーザがそれまでにしようとした変更は実行されません。

  このエラーの場合のエラー・メッセージは以下のようになります。

  error `File is locked:` *file other-user*

  ここで、 *file* はファイルの名前であり、 *other-user* は、そのファイルをロックしているユーザ名です。

この関数のデフォルトの定義は、ユーザに何をするかについての選択を促すことです。よって望むならば、 `ask-user-about-lock` を、あなた自身が作成したほかの決定をするバージョンのもので置き換えることもできます。この関数の通常の定義のコードは、‘`userlock.el`’ に記述されています。

22.6 ファイルについての情報

この節で説明する関数はすべて、ファイル名を表す文字列に対して作用します。これらの関数の名前はみな単語 ‘`file`’ で始まります。これらの関数はすべて、実在するファイルあるいはディレクトリについての情報を返します。よって、これらの関数に対する引数は、ほかに注釈がなければ、すべて実在するファイルあるいはディレクトリでなければなりません。

22.6.1 アクセス可能性をテストする

これらの関数は特定の方法でファイル・アクセスのパーミッションを検査します。

file-exists-p *filename* Function

この関数は、 *filename* の名前のファイルが存在するように見えるとき、 `t` を返します。これは、そのファイルがあなたにとって可読であることを必ずしも意味しておらず、ファイルに付随する属性を参照することができる、というだけです (Unix では、ファイルが存在し、かつ、そのファイル自身のプロテクションに関係なく、あなたがそのファイルが含まれるディレクトリに対して実行権をもっている場合には、これは正しいです)。

そのファイルが存在しない場合、あるいはアクセス制御が厳しく、ファイルの属性を参照できないようになっている場合は、この関数は `nil` を返します。

file-readable-p *filename* Function

この関数は、 *filename* の名前をもつファイルが存在し、かつあなたが可読の場合には `t` を返します。それ以外の場合、 `nil` を返します。

```
(file-readable-p "files.texi")
     ⇒ t
(file-exists-p "/usr/spool/mqueue")
     ⇒ t
(file-readable-p "/usr/spool/mqueue")
     ⇒ nil
```

file-executable-p *filename* Function

この関数は、 *filename* の名前をもつファイルが存在し、かつ実行可能ならば、 `t` を返します。それ以外の場合は `nil` を返します。そのファイルがディレクトリの場合で、実行パーミッションがあるということは、そのディレクトリの中にあるファイルの存在と属性を確認することができ、そのファイル自身は、ファイル・モードが許していれば、オープンできるということを意味しています。

file-writable-p *filename* Function

この関数は、ファイル *filename* が書き込みができる、ないしは新規作成ができるならば、 `t` を返し、それ以外は `nil` を返します。そのファイルは存在し、かつ書き込みができるならば、ファイルは書き込み可能です。ファイルは存在しないが、指定したディレクトリは存在し、かつそのディレクトリに書き込み可能ならば、ファイルは作成可能です。

以下の 3 番目の例の場合、 '`foo`' は、親ディレクトリが存在しないために、ユーザはそのディレクトリを作ることはできるかもしれませんが、書き込み可能ではありません。

```
(file-writable-p "~/foo")
     ⇒ t
(file-writable-p "/foo")
     ⇒ nil
(file-writable-p "~/no-such-dir/foo")
     ⇒ nil
```

file-accessible-directory-p *dirname* Function

この関数は、ファイル名として *dirname* の名前をもつディレクトリの中の実在するファイルに対し、あなたがオープンできるパーミッションをもつ場合、 `t` を返します。それ以外の場合 (あるいはそのようなディレクトリが存在しない場合)、 `nil` を返します。*dirname* の値は、ディレクトリ名ないしはディレクトリのファイル名のどちらかであるかもしれません。

例: 以下の実行結果のもとで、

```
(file-accessible-directory-p "/foo")
     ⇒ nil
```

'`/foo/`' の中にあるファイルをどのように読み込もうとしてもエラーになります。

file-ownership-preserved-p *filename* Function

この関数は、ファイル *filename* を削除して次に新しく作成するときに、ファイルの所有者が変わらない場合は、 `t` を返します。

file-newer-than-file-p *filename1 filename2* Function

この関数は、ファイル *filename1* がファイル *filename2* よりも新しいとき、 `t` を返します。 *filename1* が存在しない場合、 `nil` を返します。 *filename2* が存在しない場合、 `t` を返します。

以下の例では、ファイル ‘`aug-19`’ は、 19 日に書き込まれたファイルであり、ファイル ‘`aug-20`’ は、 20 日に書き込まれたファイルであり、ファイル ‘`no-file`’ は全く存在しないファイルを表わしているとします。

```
(file-newer-than-file-p "aug-19" "aug-20")
     ⇒ nil
(file-newer-than-file-p "aug-20" "aug-19")
     ⇒ t
(file-newer-than-file-p "aug-19" "no-file")
     ⇒ t
(file-newer-than-file-p "no-file" "aug-19")
     ⇒ nil
```

`file-attributes` を使って、二つの数のリストという形式でファイルの最終修正時刻を得ることができます。 22.6.4 節「ファイルに関するそのほかの情報」 p.361 を参照してください。

22.6.2 ファイルの種類の区別

この節は、さまざまな種類のファイルをどのように区別するかについて説明します。そのファイルとは、ディレクトリ、シンボリック・リンク、通常ファイルです。

file-symlink-p *filename* Function

ファイル *filename* がシンボリック・リンクの場合、 `file-symlink-p` 関数は、リンク元のファイル名を返します。これはテキスト・ファイル名か、ディレクトリか、あるいはまたほかのシンボリック・リンクの名前かもしれませんし、あるいは存在しないファイル名かもしれません。

ファイル *filename* がシンボリック・リンクでない (あるいはそのようなファイルがない) 時には、 `file-symlink-p` は `nil` を返します。

```
(file-symlink-p "foo")
     ⇒ nil
(file-symlink-p "sym-link")
     ⇒ "foo"
(file-symlink-p "sym-link2")
     ⇒ "sym-link"
(file-symlink-p "/bin")
     ⇒ "/pub/bin"
```

file-directory-p *filename* Function

この関数は、 *filename* が実在するディレクトリの名前の場合、 `t` を返します。そうでない場合、 `nil` を返します。

```
(file-directory-p "~rms")
     ⇒ t
(file-directory-p "~rms/lewis/files.texi")
     ⇒ nil
(file-directory-p "~rms/lewis/no-such-file")
     ⇒ nil
(file-directory-p "$HOME")
     ⇒ nil
(file-directory-p
 (substitute-in-file-name "$HOME"))
     ⇒ t
```

file-regular-p *filename* Function

この関数は、ファイル *filename* が存在し、かつ正規のファイル (ディレクトリ、シンボリック・リンク、名前つきパイプ、端末、そのほかの I/O デバイス、のいずれでもないもの) である場合には、 `t` を返します。

22.6.3 実名

ファイルの『実名 (truename)』は、シンボリック・リンクをなくなるまでたどり、 '.', '..' の要素を展開した結果得られる名前です。厳密には、ファイルはただ一つの実名をもつとはかぎりません。つまり、あるファイルがもつ別個の実名の数は、そのファイルに対するハード・リンクの数と同じです。しかし実名は、名前のつけ方の多様性を生む原因となるシンボリック・リンクを取り除くので、便利なものです。

file-truename *filename* Function

関数 `file-truename` は、ファイル *filename* の実名を返します。これはシンボリック・リンクを、それがなくなるまでたどることにより得られる名前です。引数は絶対ファイル名でなければなりません。

関連する情報は 24.4 節「バッファ・ファイル名」 p.397 を参照してください。

22.6.4 ファイルに関するそのほかの情報

この節では、ファイルの内容以外の詳細な情報を得るための関数について説明します。この情報とは、アクセス・パーミッションを制御するモード・ビット、所有者およびグループ番号、名前の数、 inode 数、サイズ、アクセス時間、修正時間、などです。

file-modes *filename* Function

この関数は *filename* のモード・ビットを整数で返します。モード・ビットはファイル・パーミッションとも呼ばれ、通常の Unix の方式で、アクセス制御の方針を指定します。

もし最下位ビットが 1 ならば、そのファイルはすべてのユーザが実行可能であり、 2 番目の下位ビットが 1 ならば、そのファイルはすべてのユーザが書き込み可能である、などです。

返値として可能な最大値は 4095(8 進で 7777) で、これはすべてのユーザが読込み、書込み、実行パーミッションがあり、かつ、 SUID ビットが他人とグループの両方にセットされ、かつ、 sticky ビットがセットされていることを意味します。

```
(file-modes "~/junk/diffs")
     ⇒ 492               ; 10 進の整数
(format "%o" 492)
     ⇒ "754"             ; 8 進に変換

(set-file-modes "~/junk/diffs" 438)
     ⇒ nil

(format "%o" 438)
     ⇒ "666"             ; 8 進に変換

% ls -l diffs
  -rw-rw-rw-  1 lewis 0 3063 Oct 30 16:00 diffs
```

file-nlinks *filename* Function

この関数はファイル *filename* が持っている名前 (つまりハード・リンク) の数を返します。ファイルが存在しない場合、この関数は `nil` を返します。シンボリック・リンクは、リンク元のファイルに対する名前とは解釈されないため、この関数に対し影響をおよぼさないことに注意してください。

```
% ls -l foo*
-rw-rw-rw-  2 rms       4 Aug 19 01:27 foo
-rw-rw-rw-  2 rms       4 Aug 19 01:27 foo1

(file-nlinks "foo")
     ⇒ 2
(file-nlinks "doesnt-exist")
     ⇒ nil
```

file-attributes *filename* Function

この関数はファイル *filename* の属性をリストとして返します。指定されたファイルがオープンできない場合は、 `nil` を返します。

返値のリストの要素は、以下に示す順序になります。

0. ディレクトリに対しては `t` を、シンボリック・リンクに対しては文字列 (リンク元の名前) を、あるいはテキスト・ファイルに対しては `nil` を返します。
1. ファイルが持っている名前の数。別の名前、つまりハード・リンクは、 `add-name-to-file` 関数を使って生成できます。 (22.7 節「ファイルの名前と属性の変更」 p.364 参照)
2. ファイルの UID。
3. ファイルの GID。

4. 最終アクセス時間で、これは二つの整数のリストです。第一の整数は時間を表す数字の上位 16 ビットを持ち、第二の整数はその下位 16 ビットを持ちます。 (これは current-time の値に似た書式です。 34.5 節「1 日の時刻」 p.588 を参照)
5. (上と同様に) 二つの整数のリストで表現された最終修正時間。
6. (上と同様に) 二つの整数のリストで表現された最終状態変更時間。
7. バイト表現のファイルのサイズ。
8. 'ls -l' の出力と同様の、 10 個の英文字ないしはダッシュの列で表現されたファイルのモード。
9. ファイルが削除されて再び作成される場合に、ファイルの GID が変わるような場合は、 t を返します。それ以外は nil を返します。
10. ファイルの inode 番号。
11. そのファイルが含まれるファイル・システムの、ファイル・システム番号。これと inode 番号により、そのシステム上の任意の二つのファイルを区別できます。すなわち、これらの番号の両方が同じ値であるような二つのファイルはありません。

例として、ここに 'files.texi' のファイル属性を示します。

```
(file-attributes "files.texi")
     ⇒  (nil
          1
          2235
          75
          (8489 20284)
          (8489 20284)
          (8489 20285)
          14906
          "-rw-rw-rw-"
          nil
          129500
          -32252)
```

そして、この出力結果がどのように解釈されるかを以下に示します。

nil
: はディレクトリ、シンボリック・リンクのどちらでもないことを表しています。

1
: はただ一つの名前を (現在のデフォルト・ディレクトリに名前 'files.texi' で) もつことを表しています。

2235
: は UID 2235 のユーザが所有していることを表しています。

75
: は GID 75 のグループに属していることを表しています。

(8489 20284)
: は最後にアクセスされた時間が Aug 19 00:09 であることを表しています。不幸なことに Emacs 内部では、この番号を時間を表わす文字列に変換できません。

`(8489 20284)`
: は最終修正時間が Aug 19 00:09 であることを表しています。

`(8489 20285)`
: はこの inode を最後に変更したのは Aug 19 00:09 であることを表しています。

`14906`
: はこのファイルには 14906 文字を含んでいることを表しています。

`"-rw-rw-rw-"`
: は所有者、グループ、他人に対する、読込み、書込みアクセスのモードを表わしています。

`nil`
: はファイルが再び作られる場合、同じ GID を保持することを表わしています。

`129500`
: は 129500 という inode 番号を表わしています。

`-32252`
: は -32252 というファイル・システムの番号です。

22.7 ファイルの名前と属性の変更

この節で説明する関数は、ファイルに対し、名前の変更、コピー、削除、リンク、モードの設定を行ないます。

newname を引数としてもつ関数は、この名前をもつファイルがすでに存在する場合に、引数 *ok-if-already-exists* の値に依存して、とられる動作が決まります。

- *ok-if-already-exists* が `nil` の場合は、 `file-already-exists` エラーを発生します。
- *ok-if-already-exists* が数の場合は、確認を求めます。
- *ok-if-already-exists* がそのほかの任意の値の場合は、確認なしに、古いファイルを置き換えます。

add-name-to-file *oldname newname* &optional *ok-if-already-exists* Function

この関数は、名前 *oldname* をもつファイルに、追加の名前 *newname* を与えます。これは、 *newname* が、 *oldname* に対する新しい "ハード・リンク" になることを意味します。

以下でまず最初に、二つのファイル ‘`foo`’ と ‘`foo3`’ のリストを例として挙げます。

```
% ls -l fo*
-rw-rw-rw-  1 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

ここで、形式 `(add-name-to-file "~/lewis/foo" "~/lewis/foo2")` を評価します。再びファイルのリストを示します。これにより ‘`foo`’ と ‘`foo2`’ という二つの名前が表示されます。

```
(add-name-to-file "~/lewis/foo" "~/lewis/foo2")
     ⇒ nil
```

```
% ls -l fo*
-rw-rw-rw-  2 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  2 rms       29 Aug 18 20:32 foo2
-rw-rw-rw-  1 rms       24 Aug 18 20:31 foo3
```

最後に、次の式を評価します。

```
(add-name-to-file "~/lewis/foo" "~/lewis/foo3" t)
```

そしてファイルのリストを再度挙げます。ここでは一つのファイルに対し、三つの名前があります。すなわち、 'foo', 'foo2', 'foo3' です。古い 'foo3' の内容は失われます。

```
(add-name-to-file "~/lewis/foo" "~/lewis/foo3")
     ⇒ nil

% ls -l fo*
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo2
-rw-rw-rw-  3 rms       29 Aug 18 20:32 foo3
```

この関数は VMS では無意味です。 VMS では、一つのファイルが複数の名前をもつことは許されません。

22.6.4 節「ファイルに関するそのほかの情報」 p.361 の file-nlinks も参照のこと。

rename-file *filename newname* &optional *ok-if-already-exists* Command

このコマンドはファイル *filename* の名前を *newname* に変えます。

filename が *filename* 以外の別名をもつ場合、このファイルは、それらの名前を引き続き持ちます。事実、 add-name-to-file を使って名前 *newname* を加え、 *filename* を消去するというのは、一時的に中間的な状態をもつことを除けば、名前替えと同じことをしていることになります。

この関数が対話的に呼ばれた場合には、ミニバッファに *filename* と *newname* の入力を促します。また、 *newname* がすでに存在した場合には確認を求めます。

copy-file *oldname newname* &optional *ok-if-already-exists time* Command

このコマンドはファイル *oldname* を *newname* にコピーします。 *oldname* が存在しない場合には、エラーを発生します。

time が非 nil の場合、この関数は、新しいファイルに対し、古いファイルが持っていたものと同じ最終修正時間を与えます (これはいくつかのオペレーティングシステムでのみ機能します)。

この関数が対話的に呼び出された場合、ミニバッファに、 *filename* と *newname* の入力を促します。また、 *newname* がすでに存在する場合には、確認を求めます。

delete-file *filename* Command

このコマンドは、シェル・コマンド 'rm *filename*' と同じように、ファイル *filename* を削除します。そのファイルが複数の名前を持っている場合、それはその別名で存在し続けます。

ファイルが存在しない、ないしは削除可能でない場合には、適切な種類の `file-error` エラーを発生させます (UNIX の場合、ディレクトリが書き込み可能であれば、ファイルは削除可能です)。

22.10 節「ディレクトリの作成と削除」p.375 の `delete-directory` も参照のこと。

make-symbolic-link *filename newname* &optional ok-if-already-exists — Command

このコマンドは、 *filename* に対し、名前 *newname* のシンボリック・リンクを作成します。これは、シェル・コマンド '`ln -s` *filename newname*' と同様です。

この関数は、対話的に呼び出された場合、ミニバッファで *filename* と *newname* の入力を促します。また *newname* がすでに存在する場合には、確認を求めます。

define-logical-name *varname string* — Function

この関数は、値 *string* をもつような論理名 *name* を定義します。この関数は VMS でのみ利用可能です。

set-file-modes *filename mode* — Function

この関数は *filename* のモード・ビットを *mode* に設定します (これは整数でなければなりません)。 *mode* の下位 12 ビットのみが使用されます。

set-default-file-modes *mode* — Function

この関数は、 Emacs およびそのサブプロセスによって新規に作成されるファイルの、デフォルトのファイル・プロテクションを設定します。 Emacs で作成されるファイルは、すべて初期状態ではこのプロテクションを持ちます。 UNIX では、デフォルト・プロテクションは、 "umask" 値のビットごとの補完値です。

引数 *mode* は整数でなければなりません。 *mode* の下位 9 ビットのみが使用されます。

実在するファイルを修正して保存する場合は、ファイルを作成したこととは解釈されません。この場合はファイルのモードは変化せず、デフォルトのファイル・プロテクションは使われません。

default-file-modes — Function

この関数は現在のデフォルトのプロテクション値を返します。

MS-DOS では、 "実行可能" を表わすようなファイル・モード・ビットはありません。よって Emacs は、ファイル名が '`.com`'、 '`.bat`'、 '`.exe`' で終わる場合に、実行可能であると解釈します。これは、 `file-mode` および `file-attributes` が返す値に反映されます。

22.8 ファイル名

ファイルは、一般的には、 Emacs 内部ではどこでもその名前によって参照されます。 Emacs ではファイル名は、文字列で表現されます。ファイルに対して機能するような関数はすべて、ファイル名を引数に取ることを期待します。

ファイル自身に作用するのに加え、 Emacs Lisp プログラムは、その名前自体に作用する必要があるときがあります。つまり、もとのファイル名に関係するようなファイル名を作成するために、名前を分割し、その名前の部分を使うためにです。本節では、ファイル名をどのように操作するかについて説明します。

この節で説明する関数は、実際のファイルにはアクセスしないので、実在しないファイルないしはディレクトリを示すようなファイル名に対しても機能させることができます。

VMS では、すべてのこれらの関数は、 VMS ファイル名構文と UNIX 構文の両方を理解します。これは、すべての標準 Lisp ライブラリが、 UNIX 構文でファイル名を指定しており、そのコードの変更なしに、 VMS で適正に動作させるためです。 MS-DOS では、これらの関数は UNIX 構文と同様に、 MS-DOS ファイル名構文を理解します。

22.8.1 ファイル名の構成要素

オペレーティングシステムは、ディレクトリに含まれるファイルをグループ化します。あるファイルを指定するには、ディレクトリとそのディレクトリ内でのファイル名を指定する必要があります。よって Emacs は、ファイル名を二つの主要な部分をもつものとして解釈します。それは『ディレクトリ名 (directory name)』部分と、『非ディレクトリ (nondirectory)』部分 (すなわち『そのディレクトリ内部でのファイル名 (file name within the directory)』) です。どちらかの部分が空であるかもしれません。これら二つの部分をつなげると、オリジナルのファイル名が再度作り出せます。

Unix では、ディレクトリ部分とは、ファイル名の最初から最後のスラッシュまでの部分です。非ディレクトリ部分はその残りの部分です。 VMS の構文でのこの規則は複雑です。

いくつかの目的のために、非ディレクトリ部分は、さらに厳密な意味での名前と『バージョン番号 (version number)』に分割されます。 Unix では、バックアップ・ファイルのみが名前の中にバージョン番号を持ちます。 VMS では、すべてのファイルがバージョン番号を持ちますが、ファイル名が Emacs で実際に使われる時間の内のほとんどは、そのバージョン番号を省略します。バージョン番号は、ほとんどの場合ディレクトリ・リストの中でみられます。

file-name-directory *filename* Function

この関数は、 *filename* のディレクトリ部分を返します (あるいは *filename* がディレクトリ部分をもたない場合には、 `nil` を返します)。 Unix では、この関数の返値は最後にスラッシュが付くような文字列です。 VMS では、返値は、 ‘:’、 ‘]’、 ‘>’ のいずれかの文字が最後に付くような文字列です。

```
(file-name-directory "lewis/foo")  ; Unix の例
     ⇒ "lewis/"
(file-name-directory "foo")        ; Unix の例
     ⇒ nil
(file-name-directory "[X]FOO.TMP") ; VMS の例
     ⇒ "[X]"
```

file-name-nondirectory *filename* Function

この関数は *filename* の非ディレクトリ部分を返します。

```
(file-name-nondirectory "lewis/foo")
     ⇒ "foo"
(file-name-nondirectory "foo")
     ⇒ "foo"
;; 以下は、 VMS の場合の例です。

(file-name-nondirectory "[X]FOO.TMP")
     ⇒ "FOO.TMP"
```

file-name-sans-versions *filename* Function

この関数は、 *filename* からファイル・バージョン番号、バックアップ・バージョン番号、および最後のチルダを除いたものを返します。

```
(file-name-sans-versions "~rms/foo.~1~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo~")
     ⇒ "~rms/foo"
(file-name-sans-versions "~rms/foo")
     ⇒ "~rms/foo"
;; 以下の例は VMS の場合のみ有効です。

(file-name-sans-versions "foo;23")
     ⇒ "foo"
```

file-name-sans-extension *filename* Function

この関数は、 *filename* から (存在すれば) "拡張子" を除いたものを返します。ファイル名中の拡張子とは、名前の最後の要素であり、最後の '.' から始まる部分です。例を示します。

```
(file-name-sans-extension "foo.lose.c")
     ⇒ "foo.lose"
(file-name-sans-extension "big.hack/foo")
     ⇒ "big.hack/foo"
```

22.8.2 ディレクトリ名

『ディレクトリ名 (directory name)』は、ディレクトリの名前です。ディレクトリはファイルの一種であり、ファイル名を持ちディレクトリ名と関係がありますが、同一のものではありません (これは、通常の Unix での用語法では、全く別のものです)。同じ実体のものに対するこの二つの異なった名前は、構文的な変換によって関連づけることができます。 Unix では、この変換は単純なものです。ディレクトリ名はスラッシュで終わます。一方のディレクトリのファイル名はそのスラッシュがありません。 VMS では、この関係はより複雑です。

ディレクトリ名と、ディレクトリのファイル名の相違点は、微細な点ですが、重要です。 Emacs 変数ないしは関数の引数が、ディレクトリ名として表わされている場合には、ディレクトリのファイル名は許されません。

以下の二つの関数は、ディレクトリ名とファイル名の間で変換をします。これらの関数は、‘$HOME’ のような環境変数の置き換え、‘~’、‘..’ の展開、などの特別なことはしません。

file-name-as-directory *filename* Function

この関数は、オペレーティングシステムがディレクトリの名前として解釈できるような形式で、 *filename* を表現する文字列を返します。 Unix では、これはそのファイル名を表す文字列の最後にスラッシュをつけ加えることを意味します。 VMS では、この関数は、形式 ‘[X]Y.DIR.1’ の文字列を、形式 ‘[X.Y]’ の文字列に変換します。

```
(file-name-as-directory "~rms/lewis")
     ⇒ "~rms/lewis/"
```

directory-file-name *dirname* Function

この関数は、オペレーティングシステムがファイルの名前として解釈できるような形式で、 *dirname* を表現する文字列を返します。 Unix では、これはそのディレクトリを表す文字列から最後のスラッシュを取り除くことを意味します。 VMS では、この関数は、形式 ‘[X.Y]’ の文字列を、形式 ‘[X]Y.DIR.1’ の文字列に変換します。

```
(directory-file-name "~lewis/")
     ⇒ "~lewis"
```

ディレクトリ名の省略形は、通常シンボリック・リンクを介してアクセスされるようなディレクトリに対して有用です。ユーザはまず最初に、リンク名をディレクトリの “名前” として認識し、それからディレクトリの “本当の” 名前をみつけるのにやっかいなのを発見するのです。リンク名を本物の名前に対する省略形として定義すれば、 Emacs は代わりに、この省略形をユーザにみせます。

directory-abbrev-alist Variable

変数 `directory-abbrev-alist` は、ファイル・ディレクトリを表わすために使う省略形の alist を持ちます。おのおのの要素は、 (*from* . *to*) の形式を持ち、ディレクトリ名に *from* が現われた場合、 *to* に置き換える、という意味を持ちます。 *from* 文字列は、実際は正規表現です。これは、常に ‘^’ で始まるべきです。関数 `abbreviate-file-name` が、これらの置換を実行します。

あなたのサイトにあわせて省略形を設定する場合は、 ‘site-init.el’ でこの変数の設定ができます。

ここで例を示します。 ‘/home/fsf’ などのファイル・システムが、通常、 ‘/fsf’ などのシンボリック・リンクからアクセスされるようなシステムの場合の例です。

```
(("^/home/fsf" . "/fsf")
 ("^/home/gp" . "/gp")
 ("^/home/gd" . "/gd"))
```

ディレクトリ名を省略形に変換する場合は、次の関数を使用してください。

abbreviate-file-name *dirname* Function

この関数は、引数に `directory-abbrev-alist` の要素からの省略形を適用し、ユーザのホーム・ディレクトリを ‘~’ に置換します。

22.8.3 絶対ファイル名と相対ファイル名

ファイル・システムの中にあるすべてのディレクトリは、ルート・ディレクトリからはじまる木構造を形成しています。あるファイル名というものは、この木のルートからはじまるディレクトリ名のすべてを指定できます。この場合、これは『絶対 (absolute)』ファイル名と呼ばれます。あるいは、あるファイル名は、デフォルト・ディレクトリに対して相対的に、その木の中でのファイルの位置を、指定できます。これは『相対 (relative)』ファイル名と呼ばれます。 Unix では、絶対ファイル名はスラッシュないしはチルダ (‘~’) で始まる名前で、相対ファイル名はそうではありません。 VMS でのこの規則は複雑です。

file-name-absolute-p *filename* Function

この関数は、ファイル *filename* が絶対ファイル名である場合には `t` を返し、そうでない場合には `nil` を返します。 VMS では、この関数は Unix 構文と VMS 構文の両方を解釈します。

```
(file-name-absolute-p "~rms/foo")
     ⇒ t
(file-name-absolute-p "rms/foo")
     ⇒ nil
(file-name-absolute-p "/user/rms/foo")
     ⇒ t
```

22.8.4 ファイル名を展開する関数

ファイル名の『展開 (expansion)』とは、相対ファイル名を絶対ファイル名に変換することを意味します。これはデフォルト・ディレクトリに対して相対的に変換されるので、展開対象となるファイル名と同様、デフォルト・ディレクトリを指定しなければなりません。また展開は、‘./’、‘*name*/../’ などの冗長なものをファイル名から取り除いて簡単にします。

expand-file-name *filename* &optional *directory* Function

この関数は *filename* を絶対ファイル名に展開します。 *directory* が与えられた場合、 *filename* が相対的表現であれば、それは開始部分のディレクトリです (*directory* の値は、それ自身は絶対ディレクトリ名でなければいけません。それは ‘~’ で始まるかもしれません)。与えられない場合、カレント・バッファの `default-directory` の値が使われます。例を挙げます。

```
(expand-file-name "foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
(expand-file-name "../foo")
     ⇒ "/xcssun/users/rms/foo"
(expand-file-name "foo" "/usr/spool/")
     ⇒ "/usr/spool/foo"
(expand-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/lewis/$HOME/foo"
```

‘.’ あるいは ‘..’ を含むファイル名は、正準な形式に単純化されます。

```
(expand-file-name "bar/../foo")
     ⇒ "/xcssun/users/rms/lewis/foo"
```

'~/' はユーザのホーム・ディレクトリに展開されます。'/' に続く '/' ないしは '~' に対しては、そこから絶対ファイル名が開始されると解釈され、そこに先立つ部分を無視します。よってこの '/' ないしは '~' の前にあるものはすべて取り除かれます。例を挙げます。

```
(expand-file-name
 "/a1/gnu//usr/local/lib/emacs/etc/MACHINES")
     ⇒ "/usr/local/lib/emacs/etc/MACHINES"
(expand-file-name "/a1/gnu/~/foo")
     ⇒ "/xcssun/users/rms/foo"
```

この両方の場合で '/a1/gnu/' は、絶対ファイル名がそれに続くことにより捨てられます。

`expand-file-name` は、環境変数は展開しないことに注意してください。これをするのは `substitute-in-file-name` だけです。

file-relative-name *filename directory* Function

この関数は、展開の逆のことをします。ファイル名が *directory* に対して相対的に解釈されたとき、*filename* と等価な相対ファイル名を返そうとします (もしそのような相対ファイル名が絶対ファイル名より長ければ、この関数は代わりに絶対名を返します)。

```
(file-relative-name "/foo/bar" "/foo/")
     ⇒ "bar")
(file-relative-name "/foo/bar" "/hack/")
     ⇒ "/foo/bar")
```

default-directory Variable

このバッファローカルな変数の値は、カレント・バッファのデフォルト・ディレクトリです。これは絶対ディレクトリ名であるべきです。これは '~' で始まっているかもしれません。この変数は、すべてのバッファでローカルな変数です。

`expand-file-name` は、2 番目の引数が `nil` のとき、このデフォルト・ディレクトリを使います。

Unix システムでは、この値は、つねにスラッシュで終わる文字列です。

```
default-directory
     ⇒ "/user/lewis/manual/"
```

substitute-in-file-name *filename* Function

この関数は *filename* 中で参照される環境変数を、環境変数の値で置換します。標準の Unix のシェル構文では、'$' は環境変数の値を置き換えるための接頭辞です。

環境変数の名前は、'$' に続く、英数字 (_を含む) の列です。'$' に続く文字が '{' である場合、それに対応する '}' までのすべてが変数名です。

ここで、ユーザのホーム・ディレクトリを表わす環境変数 `HOME` が、値 '/xcssun/users/rms' をもつものとします。

```
(substitute-in-file-name "$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
```

置換後の表現で、'/' の後に '~' ないしは '/' が現われる場合は、前の '/' より前のものはすべて捨てられます。

```
(substitute-in-file-name "bar/~/foo")
     ⇒ "~/foo"
(substitute-in-file-name "/usr/local/$HOME/foo")
     ⇒ "/xcssun/users/rms/foo"
```

VMS では、'$' についての置換はされません。よって VMS では、この関数は、上に示したような不必要な最初の要素の除去以外の操作は、なにもしません。

22.8.5 唯一存在するファイル名を生成する

プログラムでは、一時的なファイルを作成する必要がある場合があります。ここでは、そのようなファイルの名前をつくるために通常使われる手法を説明します。

```
(make-temp-name (concat "/tmp/" name-of-application))
```

ここではディレクトリは '/tmp/' を使用します。というのは、これは Unix では、一時ファイルを作成するのには標準の場所だからです。 `make-temp-name` の仕事は、二人の異なるユーザ、ないしは二つの異なるジョブが、同じファイル名を使うのを回避することです。

make-temp-name *string* Function

この関数は唯一の名前として使うことができる文字列を生成します。その名前は *string* で始まり、それぞれの Emacs ジョブで異なるような数値で終わります。

```
(make-temp-name "/tmp/foo")
     ⇒ "/tmp/foo021304"
```

同一の Emacs 上で動作する別のライブラリの間で、ファイル名が衝突するのを避けるために、 `make-temp-name` を使う Emacs lisp プログラムは、すべて自身の *string* をもつべきです。名前の最後に加えられる数字は、別の Emacs のジョブで動作する同一のアプリケーションを区別します。

22.8.6 ファイル名補完

この節ではファイル名を補完するための低レベルのサブルーチンについて説明します。ほかの補完関数は、17.5 節「補完」p.242 を参照のこと。

file-name-all-completions *partial-filename directory* Function

この関数は、ディレクトリ *directory* にあり、 *partial-filename* で始まるファイル名をもつファイル名について、すべての補完候補ファイル名のリストを返します。補完リスト中の要素の順序というものは、そのディレクトリにあるファイルの順序であり、これは、前もって予期できるものでもありませんし、有益な情報を提供するわけでもありません。

引数 *partial-filename* は、ディレクトリ部分もスラッシュも含まないようなファイル名でなければいけません。もし *directory* が絶対表記でなければ、カレント・バッファのデフォルト・ディレクトリがそれに先立ちます。

以下の例では、現在のデフォルト・ディレクトリ '`~rms/lewis`' に、'`f`' で始まる名前のファイルが五つあるものとしています。'`foo`'、'`file~`'、'`file.c`'、'`file.c.~1~`'、'`file.c.~2~`' です。

```
(file-name-all-completions "f" "")
     ⇒ ("foo" "file~" "file.c.~2~"
               "file.c.~1~" "file.c")

(file-name-all-completions "fo" "")
     ⇒ ("foo")
```

file-name-completion *filename directory* Function

この関数はディレクトリ *directory* にあるファイル名 *filename* に対し、補完を行ないます。ディレクトリ *directory* 中の、 *filename* で始まるすべてのファイルに対し、共通な最も長い接頭辞を返します。

一致するものがただ一つで、 *filename* に完全に一致する場合、この関数は `t` を返します。ディレクトリ *directory* 中に *filename* で始まるファイルがない場合には `nil` を返します。

以下の例で、現在のデフォルト・ディレクトリに、'`f`' で始まる五つのファイルがあるとします。そのファイルは '`foo`'、'`file~`'、'`file.c`'、'`file.c.~1~`'、'`file.c.~2~`' です。

```
(file-name-completion "fi" "")
     ⇒ "file"

(file-name-completion "file.c.~1" "")
     ⇒ "file.c.~1~"

(file-name-completion "file.c.~1~" "")
     ⇒ t

(file-name-completion "file.c.~3" "")
     ⇒ nil
```

completion-ignored-extensions User Option

`file-name-completion` は、通常このリストにある文字列で終わるようなファイル名はすべて無視します。可能な補完候補の最後の部分が、すべてこれらの接尾辞のどれかである場合、または、すべての補完候補を示すためのバッファが表示される場合は、無視しません。

典型的な値はこのようなものかもしれません。

```
completion-ignored-extensions
     ⇒ (".o" ".elc" "~" ".dvi")
```

22.9 ディレクトリの内容

ディレクトリは一種のファイルで、さまざまな名前を持ったほかのファイルを含んでいます。ディレクトリは、ファイル・システムを特徴づけるものの一つです。

Emacs は、ディレクトリ中のファイルの名前を、 Lisp のリストとしてリストアップすることができます。あるいは、 `ls` シェル・コマンドを使用して、バッファにその名前を表示することができます。後者の場合では、 `ls` コマンドに渡すオプションに応じて、それぞれのファイルについての情報を選択的に表示することができます。

directory-files *directory* &optional *full-name match-regexp nosort* Function

この関数は、ディレクトリ *directory* にあるファイル名のリストを返します。デフォルトでは、リストはアルファベット順で並べられます。

full-name が非 `nil` の場合、この関数はファイルの絶対ファイル名を返します。そうでない場合、指定のディレクトリに対する相対名を返します。

match-regexp が非 `nil` の場合、この関数は、その正規表現に一致する部分を含むようなファイル名のみを返します。それ以外のファイル名はリストから外します。

nosort が非 `nil` の場合、 `directory-files` はリストをソートしませんので、特定の順序をつけずにファイル名を返します。可能なかぎりの最大の処理速度が必要で、ファイルの処理順序が関係ない場合に、これを使用してください。ファイル処理の順序がユーザから分かってしまう場合は、あなたがそのファイルの名前をソートしてやれば、ユーザにとってはおそらく便利でしょう。

```
(directory-files "~lewis")
     ⇒ ("#foo#" "#foo.el#" "." ".."
         "dired-mods.el" "files.texi"
         "files.texi.~1~")
```

directory が、読み出すことのできるディレクトリの名前でない場合には、エラーを発生します。

file-name-all-versions *file dirname* Function

この関数は、ディレクトリ *dirname* にある、 *file* という名前のファイルのすべてのバージョンのリストを返します。

insert-directory *file switches* &optional *wildcard full-directory-p* Function

この関数は、ディレクトリ *file* 中のファイル・リストを (カレント・バッファに) 挿入します。これは、 *switches* つきの `ls` でフォーマットしたものです。この関数により、ポイントは挿入されたテキストの後に移動します。

引数 *file* は、ディレクトリ名か、ワイルドカードを含むようなファイル仕様であるかもしれません。 *wildcard* が非 `nil` の場合、これは、 *file* をワイルドカードを含むファイル仕様として取り扱うことを意味します。

full-directory-p が非 `nil` の場合、これは *file* がディレクトリで、スイッチが ‘`-d`’ を含まないということを意味します。そしてこの場合のリストは、ディレクトリ中のすべての内

容を表示すべきです (`ls` の ‘`-d`’ オプションは、ディレクトリの内容ではなく、ディレクトリ自身の情報を表示することを意味します)。

この関数は、ディレクトリのリストを作るために、変数 `insert-directory-program` で示される名前のプログラムを起動します。また *wildcard* が非 `nil` の場合、ワイルドカードを展開するために、 `shell-file-name` で指定されるシェルもまた同様に起動します。

insert-directory-program Variable

この変数の値は、関数 `insert-directory` が、あるディレクトリ中のファイルのリストをつくるために起動するプログラムです。

22.10 ディレクトリの作成と削除

Emacs Lisp でのたいていのファイル操作関数は、それがディレクトリであるようなファイルに対して使われた場合は、エラーとなります。たとえば、 `delete-file` を使って、ディレクトリを削除することはできません。ここで説明する特別な関数は、このディレクトリを作成したり削除するためにあります。

make-directory *dirname* Function

この関数は *dirname* という名前のディレクトリを作成します。

delete-directory *dirname* Function

この関数は、 *dirname* という名前のディレクトリを削除します。関数 `delete-file` は、ディレクトリであるようなファイルには機能しません。この場合には、 `delete-directory` を使わなければいけません。そのディレクトリがなんらかのファイルを含んでいる場合には、 `delete-directory` はエラーを発生します。

22.11 ファイル名を “魔力のあるもの” にする

あるファイル名に対し、特別な扱いをさせることができます。これは、これらのファイル名を『魔力のあるもの (magic)』にする、と言います。これには、名前のクラスを定義する正規表現 (このクラスのメンバはその正規表現に一致するようなものすべてです) と、この一致するファイル名に対して機能するハンドラ、すなわちすべてのプリミティブな Emacs ファイル操作を実装するハンドラ、を与えてやらなければなりません。

変数 `file-name-handler-alist` はハンドラのリストで、それぞれのハンドラがいつ適用されるかを決定する正規表現をともに持ちます。各要素は次の形式です。

(*regexp* . *handler*)

ファイル・アクセスとファイル名変換を行なうすべての Emacs プリミティブは、引数として与えられたファイル名を、 `file-name-handler-alist` に照らし合します。ファイル名が *regexp* に一致する場合、そのプリミティブは、 *handler* を呼び出してそのファイルをハンドルします。

handler には、そのプリミティブの名前が第一の引数として与えられます。残りの引数は、そのファイル操作に与えられた引数がそのまま渡されます (典型的には、この残りの引数の一番最初のものはファイル名そのものです)。たとえば、

```
(file-exists-p filename)
```

の場合で、 *filename* がハンドラ *handler* をもつ時、 *handler* はこのように呼び出されます。

```
(funcall handler 'file-exists-p filename)
```

ここに、魔法ファイル名のハンドラがハンドルする権利を得る操作を示します。

`add-name-to-file`, `copy-file`, `delete-directory`, `delete-file`, `diff-latest-backup-file`, `directory-file-name`, `directory-files`, `dired-compress-file`, `dired-uncache`, `expand-file-name`, `file-accessible-directory-p`, `file-attributes`, `file-directory-p`, `file-executable-p`, `file-exists-p`, `file-local-copy`, `file-modes`, `file-name-all-completions`, `file-name-as-directory`, `file-name-completion`, `file-name-directory`, `file-name-nondirectory`, `file-name-sans-versions`, `file-newer-than-file-p`, `file-readable-p`, `file-regular-p`, `file-symlink-p`, `file-truename`, `file-writable-p`, `get-file-buffer`, `insert-directory`, `insert-file-contents`, `load`, `make-directory`, `make-symbolic-link`, `rename-file`, `set-file-modes`, `set-visited-file-modtime`, `unhandled-file-name-directory`, `verify-visited-file-modtime`, `write-region`.

`insert-file-contents` に対するハンドラでは、その関数の *visit* 引数が非 `nil` の場合では、典型的な処理として、(`set-buffer-modified-p nil`) で、そのバッファの修正のフラグをクリアしてやる必要があります。またこれは、バッファがロックされている場合には、それをアンロックするという効果があります。

ハンドラ関数は上の操作のすべてと、おそらく将来加わるほかの操作とをハンドルしなければいけません。これらのすべての操作を実装する必要はありません。ハンドラ関数は、ある操作に対して特別にすべきことがないときは、プリミティブを再起動してやれば、その操作を “通常のやり方” でハンドルすることができます。ハンドラ関数が関係しないような操作のプリミティブは、つねに再起動すべきです。以下にこれを行う方法の例を示します。

```
(defun my-file-handler (operation &rest args)
  ;; 最初に、特別な扱いが必要な演算についてチェックする
  (cond ((eq operation 'insert-file-contents) ...)
        ((eq operation 'write-region) ...)
        ...
        ;; 知らない操作に対するハンドル
        (t (let ((inhibit-file-name-handlers
                  (cons 'my-file-handler
                        (and (eq inhibit-file-name-operation operation)
                             inhibit-file-name-handlers)))
                 (inhibit-file-name-operation operation))
             (apply operation args)))))
```

ハンドラ関数が、その中でその通常の Emacs プリミティブを呼び出す時には、そのプリミティブが同じハンドラから再度呼ばれないようする必要があります。そうでなければ無限の再帰が生じてしまいます。上の例では、変数 `inhibit-file-name-handlers` と `inhibit-file-name-operation` を使って、これをどうやるかを示しています。これらの変数は上で示したように注意して正確に使ってください。複数のハンドラをもつ場合に適切な振る舞いをさせる場合と、二つのファイル名をもち、そのそれぞれがハンドラをもつような操作の場合に、この例のような詳細な部分が重要になります。

inhibit-file-name-handlers Variable

この変数は、ある操作に対して現在使用を禁止されているハンドラのリストを持ちます。

inhibit-file-name-operation Variable

現在、ハンドラの使用禁止の対象となっている操作。

find-file-name-handler *file operation* Function

この関数は、ファイル名 *file* に対するハンドラ関数を返します。もしハンドラがなければ `nil` を返します。引数 *operation* は、ファイルに対して機能する操作でなければいけません。 *operation* は、その操作が呼ばれたときに、その最初の引数としてハンドラに渡される値です。この操作を表わす引数は、 `inhibit-file-name-operation` との比較を行なうために必要です。

file-local-copy *filename* Function

この関数は、ファイル *filename* がすでに魔法ファイルであれば、これを通常の非魔法ファイルにコピーします。

filename が、 Emacs の外のプログラムからは直接は読み書きできないような "魔法" ファイル名を指している場合、この関数は、通常のファイルに内容をコピーし、そのファイル名を返します。

filename が通常のファイル名で魔法ファイル名でない場合、この関数はなにもせず `nil` を返します。

unhandled-file-name-directory *filename* Function

この関数は、魔力を持たないディレクトリの名前を返します。ファイル *filename* が魔力を持たない場合、この関数は *filename* のディレクトリ部分を返します。そうでなければハンドラに対して何をすべきかを尋ねます。

これは、サブプロセスを起動する場合に便利です。すべてのサブプロセスはカレント・ディレクトリとして魔力を持たないディレクトリを持たなければなりません。この関数を使うのは、この要求にあったよい方法です。

22.12 ファイル・フォーマットの変換

変数 `format-alist` は、『ファイル・フォーマット (file formats)』のリストを定義します。これは、 Emacs バッファ中のデータ (テキストやテキスト属性、場合によるとほかの情報も) がファイルとなるときに、テキスト的にどのような表現になるかを記述したものです。 Emacs は、ファイルを読んだり書いたりするときに、適切ならばフォーマット変換をします。

format-alist Variable

このリストの要素は、定義されたファイル・フォーマットごとに、一つのフォーマット定義を持ちます。

フォーマット定義は、このような形式のリストです。

(*name doc-string regexp from-fn to-fn*
modify mode-fn)

フォーマット定義の各要素の意味は次のようになります。

name このフォーマットの名前です。

doc-string このフォーマットを説明した文字列です。

regexp このフォーマットで表現されたファイルを識別するために使う正規表現です。

from-fn (ファイル・データを、通常の Emacs でのデータの表現に変換するために) データをこのフォーマットで復号化するために呼びだされる関数です。

from-fn は、二つの引数 *begin* と *end* をともなって呼び出されます。これらはバッファの変換すべき部分を指定します。この関数は、そのテキストを適切に編集することにより、変換を行うべきです。これによって、テキストの長さが変わるので、*from-fn* は修正後のバッファの最後の位置を返すべきです。

from-fn は、ファイルの頭が *regexp* に一致しないようにしてやらなければなりません。そうでなければ、またフォーマット変換が行なわれてしまいます。

to-fn (通常の Emacs でのデータの表現をこのフォーマットに変換するために) データをこのフォーマットで符号化するために呼び出される関数です。

to-fn は、二つの引数 *begin* と *end* をともなって呼び出されます。これらはバッファの変換すべき部分を指定します。この関数がフォーマット変換をする方法は、二つあります。

- バッファを適切に編集する方法。この場合、 *to-fn* は、テキストを修正した部分の最後の位置を返すべきです。
- 注釈のリストを返す方法。これは、各要素が形式 (*position* . *string*) であるようなリストです。ここで、 *position* は、書き込まれるテキストとしての中での相対的な位置を指定する整数です。また、 *string* は、その位置に書き加えられる注釈です。このリストは、 *to-fn* がそれを返すときには、位置の順序でソートされていなければなりません。

  `write-region` が実際にテキストをバッファからファイルに書き込む時、この関数は、指定された注釈を対応する位置に入れ込みます。これらをすべて、そのバッファを修正せずに行ないます。

modify フラグです。符号化関数によりバッファを修正する場合には `t` を、注釈のリストを返す場合は `nil` をいれてください。

mode visit してファイルをこのフォーマットから変換した後に呼び出すモード関数です。

関数 `insert-file-contents` は、その指定したファイルを読み込んだときに、自動的にファイル・フォーマットを識別します。この関数は、そのファイルの始めの部分のテキストをフォー

マット定義の正規表現と比較します。このときに一致部分があった場合には、そのフォーマットの復号化関数を呼び出します。そして再びすべてのフォーマットをチェックします。この関数は、ファイル・フォーマットが適用できなくなるまでこれを繰り返します。

`find-file-noselect` あるいはこれを呼び出すコマンドを使ってファイルを visit するときは、同じように変換を実行します (というのはこの関数は `insert-file-contents` を呼び出すからです)。またこの関数は、それが復号化したフォーマットに対応するモード関数を呼び出します。この関数は、バッファローカルな変数 `buffer-file-format` に、フォーマット名のリストを記録します。

buffer-file-format Variable

この変数は、 visit されているファイルのフォーマットを表わしています。つまりこれは、カレント・バッファにファイルを visit したときに、復号化を行なったファイル・フォーマット名のリストです。この関数は、すべてのバッファでつねにローカルです。

`write-region` を使ってデータをファイルに書き込むとき、この関数はまず最初に、 `buffer-file-format` のフォーマットに対する符号化関数を、そのリスト中の順で呼び出します。

format-write-file *file format* Function

このコマンドは、カレント・バッファの内容をフォーマット *format* でファイル *file* に書き込みます。そして、このバッファが後に保存される場合に使うために、そのフォーマットがデフォルトであるとします。引数 *format* は、フォーマット名のリストです。

format-find-file *file format* Function

このコマンドは、 *file* を visit しますが、そのときにフォーマット *format* にしたがってそのファイルを変換します。またこのコマンドは、そのバッファが後に保存される場合のために、 *format* をそのバッファに対するデフォルトのフォーマットにします。

引数 *format* はフォーマット名のリストです。 *format* が `nil` の場合には、変換はされません。このコマンドが対話的に呼ばれた場合には、 *format* に対しては単に `RET` を入力すれば、値 `nil` が割り当てられます。

format-insert-file *file format* &optional *beg end* Function

このコマンドは、ファイル *file* の内容を現在のバッファに挿入しますが、そのときにフォーマット *format* にしたがって変換を行ないます。 *beg* と *end* が非 `nil` の場合には、これはそのファイルの読み込まれる部分を指定することになります。これは `insert-file-contents` と同様です (22.3 節「ファイルからの読込み」 p.355 参照)。

返値は `insert-file-contents` と同様です。すなわち、その絶対ファイル名と (変換後の) 挿入されたデータ長のリストを返します。

引数 *format* はフォーマット名のリストです。 *format* が `nil` の場合には、変換はされません。このコマンドが対話的に呼ばれた場合には、 *format* に対しては単に `RET` を入力すれば、値 `nil` が割り当てられます。

auto-save-file-format Variable

この変数は、自動セーブのときに使うフォーマットを指定します。この値は、 `buffer-file-format` と同様、フォーマット名のリストですが、自動セーブ・ファイルの書き込

み時には、 `buffer-file-format` の代わりに使われるリストです。この変数は、すべてのバッファでつねにローカルです。

22.13 ファイルとMSDOS

MS-DOS 上の Emacs は、テキストとバイナリ・ファイルを区別して扱います。 MS-DOS 上での通常のテキストは、行間で二つの文字シーケンス、キャリッジ・リターンとラインフィード (CRLF) を使用するため、この区別は必要です。 Emacs は、行間に一つの改行文字 (一つのラインフィード) があることだけを期待しています。 Emacs が、 MS-DOS 上でテキスト・ファイルを読む、ないしは書くとき、行のセパレータを変換する必要があります。これは Emacs が、どのファイルがテキストで、どのファイルがバイナリかを知る必要があるということを意味しています。 Emacs は、このバイナリ・ファイルかテキスト・ファイルかという決定をファイルを visit したときに行ない、その決定結果をそのファイルを保存するときに使うために、変数 `buffer-file-type` に記録します。

サブプロセスについて関係する特徴については、 33.3 節「MS-DOS でのサブプロセス」 p.563 を参照してください。

buffer-file-type Variable

この変数は、自動的にそれぞれのバッファにローカルで、バッファが visit するファイルのファイル・タイプを記録します。値は、テキストの場合は `nil` であり、バイナリの場合は `t` です。

find-buffer-file-type *filename* Function

この関数は、ファイル *filename* がテキスト・ファイルであるかバイナリファイルであるかを決定します。テキスト・ファイルの場合は `nil` を返し、バイナリ・ファイルの場合は `t` を返します。

file-name-buffer-file-type-alist User Option

この変数は、テキスト・ファイルとバイナリ・ファイルを区別するために使う alist を持ちます。リストの要素はそれぞれ (*regexp* . *type*) という形式を持ちます。 *regexp* は、ファイル名と照合します。 *type* は、 `nil` の場合はテキストを、 `t` の場合はバイナリを意味します。あるいは *type* は、そのどちらかであるかを判断するための関数が格納されます。 *type* が関数の場合は、一つの引数 (ファイル名) を取って呼び出されて、 `t` か `nil` を返すべきです。

default-buffer-file-type User Option

この変数は、名前からは特に判断がつかないようなファイルに対してデフォルトのタイプを指定します。この値は、テキストの場合には `nil`、バイナリの場合には `t` であるべきです。

find-file-text *filename* Command

`find-file` と同様の機能を持ちますが、そのファイルを名前と無関係に、テキスト・ファイルとして取り扱います。

find-file-binary *filename* Command

`find-file` と同様の機能を持ちますが、そのファイルを名前と無関係に、バイナリ・ファイルとして取り扱います。

23 バックアップと自動セーブ

バックアップ・ファイルと自動セーブは、 Emacs がクラッシュやユーザ自身のあやまちがもたらす結果からユーザを守ろうとするのに使う二つの方法です。自動セーブは、現在の編集セッションの初期の段階のテキストを保存します。バックアップ・ファイルは現在のセッション以前のファイルの内容を保存します。

23.1 バックアップ・ファイル

『バックアップ・ファイル (backup file)』は、編集しているファイルの古い内容のコピーです。 Emacs はバッファを、読み込んだファイル (visited file) に最初にセーブするときにバックアップ・ファイルを作ります。普通は、バックアップ・ファイルには現在の編集セッションの前のファイルの内容が入っていることになります。バックアップ・ファイルの内容は、通常は一度作られたら変更されずに残ります。

バックアップは通常、読み込んだファイルを新しい名前に変更して作ります。読み込んだファイルをコピーしてバックアップ・ファイルを作るように指定することも可能です。この選択は複数の名前をもつファイルに関して違いがあります。また、編集しているファイルが元のオーナーの所有のままか、編集しているユーザの所有になるかにも影響します。

デフォルトでは、 Emacs は編集しているファイルそれぞれに一つのバックアップ・ファイルを作ります。あるいは、番号付のバックアップを指定することもできます。こうすると、新しいバックアップ・ファイルはそれぞれ新しい名前になります。古い番号つきバックアップは、もうこれ以上いらないと思ったときに消すことができますし、自動的に消すこともできます。

23.1.1 バックアップ・ファイルの作成

backup-buffer Function

この関数はカレント・バッファに読み込んでいるファイルのバックアップを、必要ならば作ります。 `save-buffer` はバッファが最初にセーブされるときにこれを呼び出します。

buffer-backed-up Variable

このバッファローカル変数は、このバッファのファイルがこのバッファのためにバックアップされているかどうかを示します。非 `nil` ならバックアップ・ファイルは書き込まれています。そうでなければファイルは次のセーブ時にバックアップする必要があります (バックアップがオンになっていれば)。これは常時ローカルで、 `kill-local-variables` は効きません。

make-backup-files User Option

この変数はバックアップ・ファイルを作るかどうかを決めます。これが非 `nil` なら、 Emacs はそれぞれのファイルのバックアップをそれが最初にセーブされるときに作ります、ただし `backup-inhibited` が `nil`(下記参照) ならばです。

次の例は、 `make-backup-files` 変数を ‘`RMAIL`’ バッファの中だけで変更して、ほかでは変更しない方法を示しています。これを `nil` にすると、 Emacs は ‘`RMAIL`’ ファイル

のバックアップを作らなくなり、ディスク・スペースの節約になります。(このコードは‘.emacs’ファイルに入れておくといいでしょう。)

```
(add-hook 'rmail-mode-hook
          (function (lambda ()
                      (make-local-variable
                       'make-backup-files)
                      (setq make-backup-files nil))))
```

backup-enable-predicate Variable

この変数の値は、ファイルをバックアップする必要があるかどうか判断するときに呼び出される関数です。この関数は引数を一つ受けとり、それをファイル名とみなします。この関数が nil を返す場合はそのファイルはバックアップする必要がありません。そうでない場合はこの節のほかの変数によってバックアップするかどうか、またその方法が決まります。

デフォルト値はこうです:

```
(lambda (name)
  (or (< (length name) 5)
      (not (string-equal "/tmp/"
                         (substring name 0 5)))))
```

backup-inhibited Variable

この変数が非 nil の場合はバックアップを行ないません。この変数には、読み込んだファイル名を backup-enable-predicate で試した結果が記録されています。この変数は、読み込んだファイルにしたがってバックアップを抑制するほかの仕組みが一緒に使っても大丈夫です。たとえば、VC はバージョン管理システムで管理されているファイルのバックアップが作られるのを防ぐためにこの変数を非 nil にセットします。

この変数は、主モードを変更しても変数の値が変わらないように常時ローカルになっています。主モードはこの変数をセットしてはいけません。代わりに make-backup-files をセットすべきです。

23.1.2 バックアップの手段は名前の変更? それともコピー?

Emacs がバックアップ・ファイルを作る方法は二つあります:

- Emacs は元のファイルをバックアップ・ファイルになるように名前を変更して、次にバッファを新しいファイルに書き出すことができます。この手続きのあとでは、元のファイルの別の名前 (つまりハード・リンク) は全てバックアップ・ファイルの方を参照することになります。新しいファイルのオーナーは編集しているユーザになり、グループはそのディレクトリでユーザが新しいファイルを作成する場合のデフォルトになります。
- Emacs は元のファイルをバックアップ・ファイルにコピーし、新しい内容を元のファイルに上書きすることができます。この手続きのあとでは、元のファイルの別の名前 (つまりハード・リンク) は全てそのファイルの現在のバージョンを参照したままになります。ファイルのオーナーとグループは変わりません。

1 番目の、名前を変更する方法がデフォルトです。

変数 backup-by-copying が非 nil なら 2 番目の方法、元のファイルをコピーしてバッファの新しい内容を上書きする方法を使います。変数 file-precious-flag が非 nil のときも同様の効果 (本来の意味の副作用としてですが) があります。 22.2 節「バッファを保存する」 p.352 を参照してください。

backup-by-copying Variable

この変数が非 nil なら、 Emacs は常にバックアップ・ファイルをコピーによって作ります。

以下の二つの変数が非 nil のときは、ある特殊な場合に 2 番目の方法が使われることになります。ファイルがその特殊な場合に当てはまらない場合はファイルの扱い方に何の影響もありません。

backup-by-copying-when-linked Variable

この変数が非 nil なら、 Emacs は複数の名前をもつファイル (ハード・リンク) のバックアップをコピーによって作ります。

backup-by-copying が非 nil なら常にコピーが使われるので、この変数は backup-by-copying が nil のときだけ意味があります。

backup-by-copying-when-mismatch Variable

この変数が非 nil なら、 Emacs は名前を変更するとオーナーやグループが変わってしまうようなファイルのバックアップをコピーによって作ります。

名前を変更してもファイルのオーナーやグループに変化がないときは、すなわちファイルがユーザーの所有で、グループがそこでユーザーがファイルを作成したときのデフォルトと一致している場合、この値は何の影響も与えません。

backup-by-copying が非 nil なら常にコピーが使われるので、この変数は backup-by-copying が nil のときだけ意味があります。

23.1.3 番号付バックアップ・ファイルの作成と削除

ファイルの名前が 'foo' なら、その番号付バックアップ・バージョンの名前は 'foo.~*v*~' になります。 *v* はさまざまな整数で、こんな感じになります: 'foo.~1~'、 'foo.~2~'、 'foo.~3~'、 . . .、 'foo.~259~' など。

version-control User Option

この変数は、単独の番号付でないバックアップ・ファイルを作るか、複数の番号付バックアップを作るかを制御します。

nil　　読み込んだファイルがすでに番号付バックアップを持っていれば番号付バックアップを作り、そうでなければ作りません。

never　　番号付バックアップを作りません。

anything else
　　番号付バックアップを作ります。

番号付バックアップを使っていると、しまいにはたくさんのバックアップ・バージョンができてしまいます。そうなったらそれらを削除しなくてはなりません。 Emacs がこれを自動で行なうこともできますし、それらを削除するかどうかユーザに問い合わせることもできます。

kept-new-versions User Option

この変数の値は、新しい番号付バックアップが作られたときに取っておく最新のバージョンの数です。新しく作られたバックアップはこの数に含まれています。デフォルト値は 2 です。

kept-old-versions User Option

この変数の値は、新しい番号付バックアップが作られたときに取っておく最も古いバージョンの数です。デフォルト値は 2 です。

1、 2、 3、 5 と 7 の番号がついたバックアップがあり、これらの変数の値が両方とも 2 なら、 1 と 2 の番号のバックアップを古いバージョンとして取っておき、 5 番と 7 番を新しいバージョンとしてとっておきます。バックアップ・バージョン 3 は過剰ということになります。関数 `find-backup-file-name`(23.1.4 節「バックアップ・ファイルの命名」p.386 参照) は、どのバックアップ・バージョンを削除するか決定する役を引き受けますが、それ自身でそれらを削除することはありません。

trim-versions-without-asking User Option

この変数が非 `nil` なら、ファイルをセーブするときに過剰なバックアップ・バージョンを黙って削除します。そうでなければ、それらを削除するかどうかユーザに問い合わせます。

dired-kept-versions User Option

この変数は Dired コマンド`.`(`dired-clean-directory`) 中で最新のバックアップ・バージョンをいくつ取っておくかを指定します。これは、新しいバックアップ・ファイルを作るときに `kept-new-versions` が指定している動作とおなじことです。デフォルト値は 2 です。

23.1.4 バックアップ・ファイルの命名

この節の関数を再定義することによってバックアップ・ファイルの名前をつけるやり方をカスタマイズできるようにすることを主な目的として説明します。一つを変更したなら、多分残りも変更する必要があるでしょう。

backup-file-name-p *filename* Function

この関数は、 *filename* がバックアップ・ファイルの名前として適当な場合、非 `nil` を返します。 *filename* という名前のファイルが存在する必要はありません。この関数は名前をチェックするだけです。

```
(backup-file-name-p "foo")
     ⇒ nil
(backup-file-name-p "foo~")
     ⇒ 3
```

この関数の標準の定義は以下のとおりです:

```
(defun backup-file-name-p (file)
  "Return non-nil if FILE is a backup file \
name (numeric or not)..."
  (string-match "~$" file))
```

したがって、この関数はファイル名が ‘~’ で終っていれば非 `nil` 値を返します。(説明文字列の最初の行を二つの行に分割するのにバックスラッシュを使っていますが、実際には 1 行になります。)

この簡単な式を独立した関数にしているのは、カスタマイズするときに再定義しやすくするためです。

make-backup-file-name *filename* Function

この関数は、ファイル *filename* の番号つきでないバックアップ・ファイルの名前の文字列を返します。 Unix では、ただ単に *filename* にチルダ (tilde) を後ろにつけたものになります。

この関数の標準の定義は以下のとおりです:

```
(defun make-backup-file-name (file)
  "Create the non-numeric backup file name for FILE.
..."
  (concat file "~"))
```

この関数を定義し直すことによって、バックアップ・ファイルの名前のつけ方を変えることができます。以下の例では、チルダを後ろにつけるのに加えて ‘.’ を前につけるように `make-backup-file-name` を定義し直しています:

```
(defun make-backup-file-name (filename)
  (concat "." filename "~"))

(make-backup-file-name "backups.texi")
     ⇒ ".backups.texi~"
```

find-backup-file-name *filename* Function

この関数は、 *filename* の新しいバックアップ・ファイルのファイル名を算出します。また、既存のバックアップ・ファイルのうち削除した方がいいものを示すこともします。 `find-backup-file-name` はリストを返します。そのリストの CAR が新しいバックアップ・ファイルの名前で、 CDR が削除した方がいいバックアップ・ファイルのリストです。

二つの変数 `kept-old-versions` と `kept-new-versions` は、取っておくべきバックアップ・バージョンを決めます。この関数はこれらのバージョンを取っておくために、返り値の CDR からこれらを除きます。 23.1.3 節「番号付バックアップ・ファイルの作成と削除」 p.385 を参照してください。

この例では、 ‘`~rms/foo.~5~`’ が新しいバックアップ・ファイルに使う名前で、 ‘`~rms/foo.~3~`’ が “過剰” バージョン、現時点で消すべきだと思われるバージョン、であることを示しています。

```
(find-backup-file-name "~rms/foo")
     ⇒ ("~rms/foo.~5~" "~rms/foo.~3~")
```

file-newest-backup *filename* Function

この関数は、 *filename* の最も新しいバックアップ・ファイルの名前を返します。そのファイルのバックアップ・ファイルがなければ `nil` を返します。

あるファイル比較コマンドは、ファイルをその最新のバックアップと自動的に比較するためにこの関数を使います。

23.2 自動セーブ

Emacs は読み込んだファイル全てを定期的にセーブします。これを『自動セーブ (auto-saving)』といいます。自動セーブはシステムがクラッシュしたときに、ある決まった量以上の作業が失われることを防ぎます。デフォルトでは自動セーブは 300 打鍵ごと、または 30 秒以上暇だと行なわれます。自動セーブのユーザ向け情報は、 *The GNU Emacs Manual* の “Auto-Saving: Protection Against Disasters” 節を参照してください。ここでは自動セーブを実装するのに使われている関数とそれらを制御する変数について説明します。

buffer-auto-save-file-name Variable

このバッファローカル変数は、カレント・バッファを自動セーブするのに使われるファイルの名前です。バッファが自動セーブされないときは `nil` です。

```
buffer-auto-save-file-name
=> "/xcssun/users/rms/lewis/#files.texi#"
```

auto-save-mode *arg* Command

対話的に引数なしで使われたときはこのコマンドはトグル・スイッチになり、カレント・バッファの自動セーブがオフならオンにし、オンならオフにします。引数 *arg* があり、その値が `t`、空ではないリスト、正の整数のいずれかのときは、このコマンドは自動セーブをオンにします。それ以外のときは自動セーブをオフにします。

auto-save-file-name-p *filename* Function

この関数は *filename* が自動セーブ・ファイルの名前になり得る文字列のとき非 `nil` 値を返します。この関数は、ハッシュ・マーク (‘`#`’) で始まって終る名前が自動セーブ・ファイル名になり得るという自動セーブ・ファイルの命名法の知識に基づいて働きます。引数 *filename* はディレクトリ部分を含んでいてはいけません。

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#files.texi#"
(auto-save-file-name-p "#files.texi#")
     ⇒ 0
(auto-save-file-name-p "files.texi")
     ⇒ nil
```

この関数の標準の定義は以下のとおりです:

```
(defun auto-save-file-name-p (filename)
  "Return non-nil if FILENAME can be yielded by..."
  (string-match "^#.*#$" filename))
```

この関数は、自動セーブ・ファイルの命名法を変更したいときにカスタマイズ可能にするために存在します。これを定義し直したときは、必ず関数 make-auto-save-file-name も定義し直してください。

make-auto-save-file-name Function

この関数はカレント・バッファの自動セーブに使うファイル名を返します。これはファイル名の前と後ろにハッシュ・マーク ('#') をつけただけのものです。この関数は変数 auto-save-visited-file-name(下記参照) を見ませんので、この関数を呼び出す前にチェックしなくてはなりません。

```
(make-auto-save-file-name)
     ⇒ "/xcssun/users/rms/lewis/#backup.texi#"
```

この関数の標準の定義は以下のとおりです:

```
(defun make-auto-save-file-name ()
  "Return file name to use for auto-saves \
of current buffer.
..."
  (if buffer-file-name
      (concat
       (file-name-directory buffer-file-name)
       "#"
       (file-name-nondirectory buffer-file-name)
       "#")
    (expand-file-name
     (concat "#%" (buffer-name) "#"))))
```

自動セーブ・ファイルの命名法をカスタマイズするためにこれを定義し直せるように、独立した関数として存在します。必ず auto-asve-file-name-p も一致するように変更してください。

auto-save-visited-file-name Variable

この変数が非 nil なら、 Emacs はバッファを読み込んだファイルに自動セーブします。すなわち、編集しているファイルと同じファイルに自動セーブが行なわれます。通常この変数は nil なので、自動セーブ・ファイルは make-auto-save-file-name によって作られた別の名前になります。

この変数の値を変更したときは、どのバッファでも次に自動セーブ・モードがもう一度オンになるまで効果がありません。自動セーブ・モードがすでにオンになっているなら、auto-save-mode が再び呼ばれるまで同じファイル名に自動セーブし続けます。

recent-auto-save-p Function

この関数は、カレント・バッファが最後に読み込みまたはセーブされたとき以降に自動セーブされていたら t を返します。

set-buffer-auto-saved Function

この関数はカレント・バッファに自動セーブされたという印をつけます。このバッファはそのテキストが次に変更されるまで自動セーブされません。この関数は nil を返します。

auto-save-interval User Option

この変数の値は、 Emacs が自動セーブと自動セーブの間にキーボードから読み込む文字の数を示しています。この文字数よりも多くの文字を読み込むたびに、自動セーブがオンになっている全てのバッファで自動セーブが行なわれます。

auto-save-timeout User Option

この変数の値は、自動セーブが起きるアイドル時間の秒数です。この長さの間ユーザが何もしないでいるたびごとに、 Emacs は必要な全てのバッファで自動セーブを行ないます。 (実際には、カレント・バッファのサイズに依存する要因のせいで、指定された時間より長くなります。)

auto-save-hook Variable

この正規フックは自動セーブが行なわれる直前に実行されます。

auto-save-default User Option

この変数が非 `nil` なら、ファイルを読み込んでいるバッファはデフォルトで自動セーブがオンになります。それ以外ならそうなりません。

do-auto-save &optional *no-message current-only* Command

この関数は自動セーブが必要な全てのバッファを自動セーブします。自動セーブがオンになっていて前回の自動セーブ以降変更されているバッファをセーブします。

通常、いくつかのバッファを自動セーブすると自動セーブの実行中に ‘`Auto-saving...`’ というメッセージがエコー領域に表示されます。ですが、 *no-message* が非 `nil` ならメッセージは抑制されます。

current-only が非 `nil` ならカレント・バッファだけを自動セーブします。

delete-auto-save-file-if-necessary Function

この関数は `delete-auto-save-files` が非 `nil` ならカレント・バッファの自動セーブ・ファイルを削除します。これはバッファをセーブするたびごとに呼び出されます。

delete-auto-save-files Variable

この変数は関数 `delete-auto-save-file-if-necessary` で使われます。これが非 `nil` なら、 Emacs は本当のセーブを (読み込んだファイルへ) 実行するときに自動セーブ・ファイルを削除します。これはディスク・スペースを節約し、あなたのディレクトリが散らからないようにします。

rename-auto-save-file Function

この関数はカレント・バッファのセーブするファイル名が変更されていれば、自動セーブ・ファイル名を調節します。また既存の自動セーブ・ファイルの名前も変えます。セーブするファイル名が変更されていなければ、この関数は何もしません。

buffer-saved-size Variable

このバッファローカル変数の値は、カレント・バッファの最後に読み込まれたかセーブされたか自動セーブされたときの長さです。これは、サイズの大量の減少を検出するのに使われ、それに応じて自動セーブをオフにします。

これが -1 なら、大量の削除のためにこのバッファの自動セーブが一時的に停止されていることを意味します。明示的にこのバッファをセーブするとこの変数に正の値が入れられるので、自動セーブは再びオンになります。自動セーブ・モードをオフしたりオンするとこの変数の値も変更されます。

auto-save-list-file-name Variable

この変数は (非 `nil` なら) 全ての自動セーブ・ファイルの名前を記録するファイルを指定します。 Emacs は自動セーブを行なうたびに、自動セーブがオンになっているバッファそれぞれにつき 2 行をこのファイルに書き込みます。 1 行目は読み込んだファイルの名前で (なければ空です)、 2 行目は自動セーブ・ファイルの名前です。

Emacs が正常に終了するときは、このファイルを削除します。 Emacs がクラッシュした場合は、下手をすると失われた作業を含むかもしれない自動セーブ・ファイル全てをそのファイルで見つけることができるでしょう。 `recover-session` コマンドはこれらのファイルを使います。

このファイルのデフォルトの名前はあなたのホーム・ディレクトリにあり ‘`.saves-`’ で始まります。 Emacs のプロセス ID とホスト名も含んでいます。

23.3 復帰

ファイルを広範囲に渡って変更してから気が変わった場合、 `revert-buffer` コマンドでファイルの直前のバージョンを読み込むことによりその変更を破棄できます。 *The GNU Emacs Manual* の “Reverting a Buffer” 節を参照してください。

revert-buffer &optional *check-auto-save noconfirm* Command

このコマンドはバッファのテキストをディスク上の読み込んだファイルのテキストで置き換えます。この動作はファイルを読み込んだまたはセーブした以降の全ての変更を取り消します。

引数 *check-auto-save* が非 `nil` で、最新の自動セーブ・ファイルが読み込んだファイルよりも新しいなら、 `revert-buffer` はユーザに代わりにそれを使うかどうか問い合わせます。そうでなければ常に読み込んだファイル (visited file) 自身のテキストが使われます。対話的に使う場合、数値前置引数があれば *check-auto-save* がセットされます。

通常 `revert-buffer` はバッファを変更する前に確認を求めますが、引数 *noconfirm* が非 `nil` なら確認を求めません。

復帰するときにはそのバッファ中のマーカの位置を、 `insert-file-contents` の置換機能を使って保存しようとします。復帰する前にバッファの内容とファイルの内容が同一なら、全てのマーカが保存されます。同一でない場合はバッファを変更して、次にバッファの先頭と末尾の変更されていないテキスト (もしあれば) の中のマーカを保存します。保存された追加のマーカはどれも信用できません。

これらの変数、ほとんどがバッファローカル変数です、を設定することによって `revert-buffer` の動作をカスタマイズできます。

revert-buffer-function Variable

この変数の値はこのバッファを復帰するのに使う関数です。非 `nil` ならこれは、復帰の作業を行なうために関数として引数なしで呼び出されます。値が `nil` なら、復帰は通常のやり方で行なわれます。

Dired モードのような編集されるテキストがファイルの内容ではないがある別のやり方で再生できるモードでは、この変数に内容を再生する関数をバッファローカルな値としてセットします。

revert-buffer-insert-file-contents-function Variable

この変数の値はもし非 `nil` なら、このバッファを復帰するときに新しい内容を挿入するのに使う関数です。この関数は二つの引数を受けとります。 1 番目は使うファイル名で、 2 番目はユーザが自動セーブ・ファイルを読み込むことを求めたなら `t` になります。

before-revert-hook Variable

この正規フックは変更された内容が実際に挿入される前に `revert-buffer` によって実行されます。ただし `revert-buffer-function` が `nil` のときだけです。

Font Lock モードはバッファがもうフォント修飾 (fontify) されないことを記録するためにこのフックを使います。

after-revert-hook Variable

この正規フックは変更された内容が実際に挿入されたあとに `revert-buffer` によって実行されます。ただし `revert-buffer-function` が `nil` のときだけです。

Font Lock モードは更新されたバッファの内容のフォントを再算出するためにこのフックを使います。

24 バッファ

『バッファ (buffer)』は、編集するテキストを格納する Lisp オブジェクトです。通常、バッファは対応づけられたファイルの内容を保持するのに用いられますが、ファイルに対応づけられていないバッファもあります。同時に複数のバッファが存在することができますが、常にただ一つのバッファが『カレント・バッファ (current buffer)』となります。ほとんどの編集命令はカレント・バッファの内容に対して行なわれます。カレント・バッファを含め、バッファはウィンドウに表示されるかもしれませんし、されないかもしれません。

24.1 バッファの基本

Emacs において編集対象としているバッファは、明確な名前を持った、編集可能なテキストを保持するオブジェクトです。バッファは Lisp プログラムの中では一つの特殊なデータ型です。バッファの内容は拡張可能な文字列と考えることができます。バッファの任意の部分ののの追加および削除が行なわれます。第 29 章「テキスト」p.479 を参照してください。

個々の Lisp バッファ・オブジェクトは多くの情報を持ちます。この情報のいくつかは変数としてプログラマから直接アクセスされ、その他の情報は特別な関数によってのみアクセスされます。たとえば、対応づけられたファイル名は変数によって直接アクセスできますが、ポイントの値はプリミティブ関数によってのみアクセスできます。

直接アクセスできるバッファ固有の情報は、その値がある特定のバッファでのみ有効である、『バッファローカル (buffer-local)』変数に格納されます。この特徴により、それぞれのバッファで変数の値を上書きすることが可能となります。ほとんどの主モードは、この手段を用いて `fill-column` や `comment-column` などのような変数を上書きしています。バッファローカル変数とその関連関数の詳細については 10.9 節「バッファローカルな変数」p.140 を参照してください。

バッファに対応づけられるファイルに関する関数や変数については、 22.1 節「ファイルを visit する」p.349 と 22.2 節「バッファを保存する」p.352 を参照してください。ウィンドウ内のバッファの表示に関する関数や変数については、 25.6 節「バッファとウィンドウ」p.415 を参照してください。

bufferp *object* Function

この関数は、 *object* がバッファであれば `t` を返し、そうでなければ `nil` を返します。

24.2 カレント・バッファ

一般に一回の Emacs セッションの間に多くのバッファが用いられます。常にその中の一つが『カレント・バッファ (current buffer)』となります。バッファ内のテキストを調べたり変更するプリミティブのほとんどが、暗黙に (implicit に) カレント・バッファに対して処理を行うため、このカレント・バッファ内でほとんどの編集が行なわれます (第 29 章「テキスト」p.479 参照)。通常は、選択されたウィンドウ内のスクリーンに表示されたバッファがカレント・バッファですが、そうでない場合もあります。 Lisp プログラムは、あるバッファの内容に対する処

理を行うために、スクリーンに表示されているものを変更することなく、そのバッファを一時的にカレント・バッファとすることができます。

Lisp プログラムでは、 set-buffer を呼ぶことによりカレント・バッファを選択します。選択されたバッファは、新たなバッファが選択されるまでカレントのままでいます。

編集命令が編集コマンド・ループ (editor command loop) に戻るとき、混乱を避けるために、コマンド・ループは選択されたウィンドウに表示されているバッファをカレント・バッファにします。つまり Emacs が命令を読み込んだときにカーソルがあったバッファがその命令の適用されるバッファとなります (第 18 章「コマンド・ループ」 p.257 を参照してください)。したがって set-buffer は、ユーザが他のバッファを編集する目的でバッファを変更するための手段ではありません。この目的のためには 25.7 節「ウィンドウ内でのバッファの表示」 p.416 に説明されている関数を使わねばなりません。

しかし、カレント・バッファを変更する Lisp 関数は、それを元に戻す作業をコマンド・ループに頼るべきではありません。 Emacs Lisp で書かれた編集命令は、コマンド・ループからのほかに、他のプログラムからも呼ばれます。呼ぶ側にとっては、サブルーチンがカレント・バッファを変更しない方が便利です (もちろん、それが目的である関数は別です)。したがって通常は、終了したときにカレント・バッファを元に戻す関数である save-excursion の中で set-buffer を使用すべきです (27.3 節「脱線」 p.466 参照)。例として、命令 append-to-buffer のコードを要約した説明文字列をつけて示します。

```
(defun append-to-buffer (buffer start end)
  "Append to specified buffer the text of the region.
..."
  (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、ローカル変数をカレント・バッファに束縛し、 save-excursion がポイント、マーク、もとのバッファをそれぞれ記録します。次に set-buffer が別のバッファをカレント・バッファにします。最後に、 insert-buffer-substring が、もとのカレント・バッファから新たなカレント・バッファへ文字列をコピーします。

文字列が追加されたバッファがウィンドウ内に表示されている場合は、次の画面の書き換えを行うときにテキストの変更が表示されるようになります。表示されていない場合は、画面からはすぐには変更を見ることができません。つまりそのバッファは、コマンドの実行中は一時的にカレントになりますが、表示されるわけではありません。

バッファローカルである変数を (let もしくは関数の引数によって) ローカル束縛するときは、ローカル束縛するスコープの最初と最後で、同じバッファがカレントであるようにしてください。そうしなければ、その変数をあるバッファで束縛した後、別のバッファ内で束縛をとく (unbind) ことになります。変数のスコープの最初と最後とで同じバッファをカレントであるようにするのに、 2 とおりの方法があります。簡単なものは、その束縛のスコープ中でカレント・バッファを変更しないことです。もう一つは、変数の束縛をとくときに、最初にカレントであったバッファを再度カレントにすることを保証する save-excursion を用いることです。

set-buffer を用いてカレント・バッファを戻すのは危険です。戻りたいバッファではないバッファがカレントである状態でとりやめ (quit) が起きたときに、 set-buffer が呼ばれないからです。次の例はやってはいけない例です。

```
(let (buffer-read-only
      (obuf (current-buffer)))
  (set-buffer ...)
  ...
  (set-buffer obuf))
```

次に示すように save-excursion を用いると、通常の動作はもとより、とりやめ (quit)、エラー、 throw も正しくハンドルされます。

```
(let (buffer-read-only)
  (save-excursion
    (set-buffer ...)
    ...))
```

current-buffer Function

この関数はカレント・バッファを返します。

```
(current-buffer)
     ⇒ #<buffer buffers.texi>
```

set-buffer *buffer-or-name* Function

この関数は *buffer-or-name* をカレント・バッファにします。現在選択されているウィンドウにも他のウィンドウにも新たなバッファを表示しません。したがって、ユーザは必ずしもカレント・バッファを見ることができません。しかし Lisp プログラムは、どのような場合でもそのバッファに対して処理をすることができます。

この関数は *buffer-or-name* で識別されるバッファを返します。 *buffer-or-name* によって既存のバッファのうちの一つを特定できない場合はエラーとなります。

24.3 バッファの名前

個々のバッファは一意な名前を文字列として持ちます。バッファに対して処理を行う多くの関数は、バッファもしくはバッファの名前を引数として受けつけます。 *buffer-or-name* と呼ばれる引数がこの種のもので、もし文字列もしくはバッファのいずれでもない場合はエラーとなります。 *buffer* と呼ばれる引数は、名前ではなくバッファ・オブジェクトでなければなりません。

短期的に用いられ、通常はユーザが興味を持たないようなバッファは空白文字で始まる名前を持ちます。こうすることによって、 list-buffers 命令と buffer-menu 命令の対象とはならなくなります。さらに、空白文字で始まる名前のバッファは、最初に undo 情報の記録が禁止されます。 29.9 節「取り消し」 p.492 を参照してください。

buffer-name &optional *buffer* Function

この関数は、 *buffer* の名前を文字列として返します。 *buffer* が与えられなかった場合は、カレント・バッファの名前を返します。

`buffer-name` が `nil` を返した場合は、 *buffer* がすでにキルされていることを意味します。 24.10 節「バッファの消去」 p.404 を参照してください。

```
(buffer-name)
     ⇒ "buffers.texi"
(setq foo (get-buffer "temp"))
     ⇒ #<buffer temp>
(kill-buffer foo)
     ⇒ nil
(buffer-name foo)
     ⇒ nil
foo
     ⇒ #<killed buffer>
```

rename-buffer *newname* &optional *unique* Command

この関数はカレント・バッファの名前を *newname* に変更します。 *newname* が文字列でなかったり、その名前のバッファがすでに存在する場合はエラーとなります。この関数は `newname` を返します。

通常、 *newname* がすでに使われている場合は、 `rename-buffer` はエラーとなります。しかし *unique* が `nil` でない場合は、関数 `generate-new-buffer` を用いて、 *newname* を使われていない名前に変更します。対話的に用いる場合は、数値前置引数によって *unique* を `nil` でない値とします。

この命令の使い方の一つとして、 ‘`*shell*`’ バッファを別の名前にするというものがあります。こうすることにより ‘`*shell*`’ という名前の第 2 のシェル・バッファを作ることが可能となります。[1]

get-buffer *buffer-or-name* Function

この関数は、 *buffer-or-name* で指定されるバッファを返します。 *buffer-or-name* が文字列であり、その名前のバッファが存在しない場合は値は `nil` となります。 *buffer-or-name* がバッファの場合は、それをそのまま返します (これはあまり役に立たないため、通常は名前が引数となります)。たとえば

```
(setq b (get-buffer "lewis"))
     ⇒ #<buffer lewis>
(get-buffer b)
     ⇒ #<buffer lewis>
(get-buffer "Frazzle-nots")
     ⇒ nil
```

24.9 節「バッファの生成」 p.403 の関数 `get-buffer-create` も参照してください。

[1] 訳注: `M-x shell` は、すでに ‘`*shell*`’ という名前のバッファがあったときは、新たに別のバッファを作ろうとしません。ここに書かれているような操作をすることにより、同時に複数の ‘`*shell*`’ バッファを作ることが可能となります。

generate-new-buffer-name *starting-name* Function

この関数は、新しいバッファのためのユニークな名前を返します。ただしバッファは作りません。新しい名前は、 *starting-name* で始まり、 '<...>' の内側に数字をいれた文字列を加えた、現在どのバッファにも使われていない名前です。

24.9 節「バッファの生成」 p.403 の関連する関数 `generate-new-buffer` を参照してください。

24.4 バッファ・ファイル名

『バッファ・ファイル名 (buffer file name)』は、バッファに対応づけられたファイルの名前です。バッファがファイルに対応づけられていない場合は、そのバッファ・ファイル名は `nil` となります。ほとんどの場合、バッファ名は、バッファ・ファイル名のディレクトリ部分を取り除いたものと同じですが、バッファ・ファイル名とバッファ名は区別され、独立に設定することができます。 22.1 節「ファイルを visit する」 p.349 を参照してください。

buffer-file-name &optional *buffer* Function

この関数は、 *buffer* が対応づけられているファイルの絶対ファイル名 (absolute file name) を返します。 *buffer* がファイルに対応づけられていない場合は、 `buffer-file-name` は `nil` を返します。 *buffer* が与えられないときは、カレント・バッファがデフォルトとなります。

```
(buffer-file-name (other-buffer))
     ⇒ "/usr/user/lewis/manual/files.texi"
```

buffer-file-name Variable

このバッファローカルな変数は、カレント・バッファに対応づけられたファイルの名前を持ちます。ファイルに対応づけられていない場合は `nil` を持ちます。常時ローカル変数であり `kill-local-variables` の影響を受けません。

```
buffer-file-name
     ⇒ "/usr/user/lewis/manual/buffers.texi"
```

ほかのいろいろな手続きを踏まずにこの変数の値を変更することは危険です。 '`files.el`' の `set-visited-file-name` の定義を参考にしてください。バッファ名の変更など、そこで行なわれていることのいくつかは厳密には必要ではありませんが、そのほかは Emacs を混乱させないために必要なことです。

buffer-file-truename Variable

このバッファローカルな変数はカレント・バッファに対応づけられたファイルの真の名前 (truename) を持ちます。ファイルに対応づけられていない場合は `nil` を持ちます。常にローカル変数であり `kill-local-variables` の影響を受けません。 22.6.3 節「実名」 p.361 を参照してください。

buffer-file-number Variable

このバッファローカルな変数は、カレント・バッファに対応づけられたファイルのファイル番号[2]とディレクトリ・デバイス番号を持ちます。あるいはファイルに対応づけられていない場合は `nil` を持ちます。常時ローカル変数であり `kill-local-variables` の影響を受けません。 22.6.3 節「実名」 p.361 を参照してください。

この変数の値は通常 (*filenum devnum*) の形式のリストです。この数字の組によって、システムでアクセス可能な全てのファイル中からそのファイルを一意に特定することができます。これらのより詳しい情報は、 22.6.4 節「ファイルに関するそのほかの情報」 p.361 の関数 `file-attributes` を参照ください。

get-file-buffer *filename* Function

この関数は、ファイル *filename* に対応づけられたバッファを返します。もしそのようなバッファがない場合は `nil` を返します。引数 *filename* (文字列でなければなりません) は展開されたあと (22.8.4 節「ファイル名を展開する関数」 p.370 参照) 全てのバッファに対応づけられたファイル名と比較されます。

```
(get-file-buffer "buffers.texi")
    ⇒ #<buffer buffers.texi>
```

特殊な状況では、同じファイル名に複数のバッファが対応づけられていることがありえます。このような場合、この関数はバッファ・リストの中の最初のものを返します。

set-visited-file-name *filename* Command

filename が空文字列でない場合は、この関数はカレント・バッファに対応づけられているファイルを *filename* に変更します (バッファがファイルに対応づけられていない場合は、バッファにファイルを対応づけます)。次にバッファが保存されるとき、それは新たに指定されたファイルに保存されます。バッファの内容が以前対応づけられていたファイルの内容と一致していたとしても、 *filename* の内容と一致しない (少なくとも Emacs にはわからない) ので、この命令はバッファに変更済みの印をつけます。

filename が `nil` もしくは空文字列である場合は、 “対応づけられたファイルがない” ことを表わします。この場合は `set-visited-file-name` はバッファに対応するファイルがないと印をつけます。

`set-visited-file-name` が対話的に呼ばれた場合はミニバッファで *filename* の入力を促します。

24.5 節「バッファの変更」 p.399 の中の `clear-visited-file-modtime` と `verify-visited-file-modtime` を参照してください。

list-buffers-directory Variable

このバッファローカル変数は、対応づけられたファイル名を持たないバッファが、バッファ一覧表にファイル名の代わりに表示する文字列を記録します。 Dired バッファはこの変数を使用します。

---

[2] 訳注: Unix ファイル・システムでは i ノード番号。

24.5 バッファの変更

Emacs は、それぞれのバッファごとにそのバッファのテキストを変更したか否かを記録するために、『変更フラグ (modified flag)』と呼ばれるフラグを管理します。このフラグはバッファの内容を変更したときに `t` となり、バッファを保存 (保存) したときに `nil` となります。したがってこのフラグは、保存されていない変更があるか否かを示します。このフラグの値は通常モード・ラインに表示され (20.3.2 節「モード行の中で使用されている変数」 p.331 参照)、ファイルの保存や (22.2 節「バッファを保存する」 p.352 参照) 自動保存 (auto-saving) の (23.2 節「自動セーブ」 p.388 参照) 制御をします。

いくつかの lisp プログラムはこのフラグを明示的に設定します。たとえば関数 `set-visited-file-name` は、フラグを `t` とします。これはテキストが以前対応づけられていたファイルから変更されてはいませんが、新たに対応づけられたファイルとは一致していないためです。

バッファの内容を変更する関数については第 29 章「テキスト」 p.479 に記述されています。

buffer-modified-p &optional *buffer* Function

この関数は、バッファ *buffer* が最後にファイルから読込まれてから、もしくは保存されてから変更された場合に `t` を返し、そうでない場合に `nil` を返します。 *buffer* が省略された場合はカレント・バッファが調べられます。

set-buffer-modified-p *flag* Function

この関数は、 *flag* が `nil` でない場合にカレント・バッファを変更されたものとし、 `nil` の場合に変更されていないものとします。

この関数をもう一つの効果は、カレント・バッファのモード・ラインを無条件に再表示することです。現に関数 `force-mode-line-update` はこの関数を呼ぶことによって実現されています。

```
(set-buffer-modified-p (buffer-modified-p))
```

not-modified &optional *mark-modified* Command

このコマンドは、カレント・バッファが変更されておらず保存する必要がないものとします。前置引数が与えられたときはバッファを変更したものとするので、 `C-x C-s` によってバッファが保存されます。この関数はエコー領域にメッセージを表示するため、プログラムの中では使わないでください。かわりに (上記の) `set-buffer-modified-p` を使用してください。

buffer-modified-tick &optional *buffer* Function

この関数は、 *buffer* の変更された回数を返します。これはバッファが変更されるたびに増加するカウンタです。 *buffer* が `nil` (もしくは省略された場合) はカレント・バッファが対象となります。

24.6 変更時間の比較

ファイルを開き、そのバッファを変更している間にディスク上のファイルが変更された状況を考えてください。この状態でバッファを保存すると、ファイルの変更が上書きされてしまいま

す。場合によってはこれが望ましい場合もありますが、通常は有益な情報が失われてしまいます。したがって Emacs は、ファイルを保存する前に、以下に記述される関数を用いてファイルの変更時間を調べます。

verify-visited-file-modtime *buffer* Function

この関数は、対応づけられたファイルの変更時間として *buffer* に記録された時間と、オペレーティング・システムによって記録されたファイル自体の変更時間とを比較します。Emacs がファイルを読み込んでから、もしくは保存してからほかのプロセスがこのファイルを書き換えないかぎり、この二つの時間は同じでなければなりません。

この関数は、最後の実際の変更時間と Emacs に記録された変更時間とが等しいときには `t` を返し、そうでないときには `nil` を返します。

clear-visited-file-modtime Function

この関数はカレント・バッファに対応づけられたファイルの最後の変更時間の記録を消去します。これによって、次にこのバッファを保存するときにファイルの変更時間の矛盾による警告を受けません。

この関数は `set-visited-file-name` の中など、変更されたファイルの上書きを防ぐための通常行なわれる検査が不要な例外的な場所から呼ばれます。

visited-file-modtime Function

この関数は、バッファに記録されているファイルの最終変更時間を、(*high* . *low*) という形式のリストとして返します (これは `file-attributes` が時間の値を返すのに用いる形式と同じものです。 22.6.4 節「ファイルに関するそのほかの情報」p.361 を参照してください)。

set-visited-file-modtime &optional *time* Function

この関数は、バッファに記録されているファイルの最終変更時間を、 *time* が `nil` でない場合は *time* で指定された値に、 `nil` の場合は対応づけられたファイルの最終変更時間に、それぞれ書き換えます。

time が `nil` でないときは、それは (*high* . *low*) もしくは (*high* *low*) の形式でなければなりません。いずれの場合も二つの整数値を持ち、それぞれが時間の 16 ビット分の値を保持します。

この関数は、バッファが普通にファイルから読まれなかった場合や、ファイル自身がある正常な理由で変更された場合などに便利です。

ask-user-about-supersession-threat *filename* Function

この関数は、その内容が対応づけられた実際のファイル *filename* に一致していない (obsolete) バッファを変更しようとしたときに、どのように処理を継続するかをユーザに問い合わせるのに用いられます。『obsolete バッファ (obsolete buffer)』とは対応づけられたディスク上のファイルがバッファの最後に保存された時間よりも新しい、変更されていないバッファのことです。これはたぶん他のプログラムによってファイルが変更されたことを意味します。

この関数は、ユーザの答えによって、正常に終了し、バッファはそのまま変更されるか、データ (*filename*) を伴って `file-supersession` エラーが通知され、バッファの変更はされないかの、いずれかとなります。

この関数は、 Emacs によって適切なときに自動的に呼ばれます。この関数を再定義することによって Emacs をカスタマイズすることができるように用意されています。標準的な定義としてはファイル '`userlock.el`' を参照してください。

22.5 節「ファイルのロック」 p.357 のファイル・ロック機構も参照してください。

24.7 読みだし専用バッファ

バッファが『読みだし専用 (read-only)』である場合は、その内容を変更することはできませんが、スクロールしたりナローイングを行うことによって、その見え方を変更することは可能です。

読みだし専用バッファは 2 種類の状況で用いられます。

- 書き換え禁止ファイルに対応づけられたバッファは、通常読みだし専用となります。

  これは、そのファイルに保存する目的でバッファを編集することが無駄である、もしくは望ましくはないことをユーザに示すことが目的です。これらにもかかわらずバッファ・テキストを変更したいユーザは、 `C-x C-q` によって読みだし専用フラグを解除することで変更を行うことができます。

- Dired や Rmail などのモードのように、通常の編集コマンドで内容を変更することがたぶん誤りであるような場合に、バッファを読みだし専用とします。

  これらのモードの特別なコマンドは、バッファのテキストを変更するときに、 `buffer-read-only` を (`let` を用いて) `nil` とするか、 `inhibit-read-only` を `t` とします。

buffer-read-only Variable

このバッファローカル変数は、バッファが読みだし専用であるか否かを指定します。この変数が `nil` でないときはそのバッファは読みだし専用です。

inhibit-read-only Variable

この変数が `nil` でないときは、読みだし専用バッファや読みだし専用文字も変更することが可能です。読みだし専用文字とは、 `nil` でない読みだし専用属性 (テキスト属性とオーバレイ属性のいずれか) をもつ文字です。テキスト特性についての詳細は 29.18.4 節「特殊な意味をもつ属性」 p.513 を参照してください。オーバレイやその属性についての詳細は 35.9 節「オーバレイ」 p.609 を参照してください。

`inhibit-read-only` が `t` の場合は、全ての読みだし専用文字属性は効果がありません。 `inhibit-read-only` がリストの場合は、読みだし専用文字属性がリストのメンバである場合に (`eq` で比較が行なわれます) その効果がなくなります。

toggle-read-only Command

このコマンドは、カレント・バッファが読みだし専用であるか否かを変更します。対話的に用いるためのものなので、プログラムの中では使用しないでください。プログラム内で

は、読みだし専用フラグを設定したいのか解除したいのかは判っているはずなので、明示的に `buffer-read-only` を `t` か `nil` の適切な値に設定することができます。

barf-if-buffer-read-only Function

この関数は、カレント・バッファが読みだし専用である場合に `buffer-read-only` エラーを通知します。カレント・バッファが読みだし専用であるときにエラーを通知するもう一つの方法として、18.3 節「対話的呼出し」p.262 を参照してください。

24.8 バッファ・リスト

『バッファ・リスト (buffer list)』は、全ての使用可能なバッファのリストです。バッファを生成するとこのリストに加えられ、バッファをキルするとこのリストから削除されます。リスト内のバッファの順序は、基本的には選択されているウィンドウに最近表示されたものほど先頭に近くなっています。バッファは、選択されるとこのリストの先頭に移動し、関数 `bury-buffer` によってリストの末尾に移動します。いくつかの関数、特に `other-buffer` は、この順序を使用します。ユーザに表示されるバッファのリストもこの順序となります。

buffer-list Function

この関数は、名前がスペースで始まるバッファも含む全てのバッファのリストを返します。各要素はバッファそのものであって、名前ではありません。

```
(buffer-list)
     ⇒ (#<buffer buffers.texi>
         #<buffer  *Minibuf-1*> #<buffer buffer.c>
         #<buffer *Help*> #<buffer TAGS>)

;; minibuffer の名前がスペースで始まることに注意！
(mapcar (function buffer-name) (buffer-list))
    ⇒ ("buffers.texi" " *Minibuf-1*"
        "buffer.c" "*Help*" "TAGS")
```

このリストは、 Emacs 内部で使われているリストのコピーです。これを変更してもバッファの順序には影響を与えません。

other-buffer &optional *buffer visible-ok* Function

この関数は、バッファ・リストの中で *buffer* 以外の最初のバッファを返します。通常は選択されているウィンドウに *buffer* を除いて最近表示されたバッファです。スペースで始まる名前のバッファは除外されます。

buffer が省略された場合は (もしくはそれがバッファではなかったら)、 `other-buffer` は、現在表示されているフレーム内のどのウィンドウにも表示されていない、バッファ・リスト内で最初のバッファを返します。

選択されたフレームが `nil` 以外の `buffer-predicate` パラメータを持っている場合は、関数 `other-buffer` は、この述語 (predicate) を用いてバッファを選択します。それぞれのバッファごとにこの述語を呼び、その値が `nil` であればそのバッファは無視されます。 26.3.3 節「X ウインドウ・フレーム・パラメータ」p.438 を参照してください。

visible-ok が `nil` であれば、ほかにバッファがあるかぎり、 `other-buffer` は現在表示されているフレーム内の全てのウィンドウに表示されているものを除外します。 *visible-ok* が `nil` でなければ、バッファが表示されているか否かは関係なくなります。

適切なバッファがない場合は、バッファ '`*scratch*`' が (必要であれば生成され) 返されます。

bury-buffer &optional *buffer-or-name* Command

この関数は、 *buffer-or-name* を他のバッファのリスト内の順序を変えることなく、バッファ・リストの最後にします。したがってこのバッファは、 `other-buffer` が返すバッファとして最も優先度の低いものになります。

buffer-or-name が `nil` もしくは省略された場合は、カレント・バッファが対象となります。さらに、カレント・バッファが選択されたウィンドウに表示されている場合は、選択されたウィンドウの (`other-buffer` によって選択される) 他のバッファが表示されます。しかし他のウィンドウに表示されている場合は、そこに表示されたままとなります。

全てのウィンドウ内に表示されているあるバッファを置き換えたい場合は、 `replace-buffer-in-windows` を使用してください。 25.6 節「バッファとウィンドウ」 p.415 を参照してください。

24.9 バッファの生成

この節ではバッファを生成する 2 種類のプリミティブについて説明します。 `get-buffer-create` は指定された名前のバッファが存在してない場合にバッファを生成します。 `generate-new-buffer` は常に新しいバッファを生成し、それにユニークな名前を与えます。

バッファを生成する関数として、ほかに `with-output-to-temp-buffer` (35.8 節「一時表示」 p.608 参照) と `create-file-buffer` (22.1 節「ファイルを visit する」 p.349 参照) とがあります。またサブプロセスの起動によってもバッファを生成することができます。

get-buffer-create *name* Function

この関数は、 *name* という名前のバッファを返します。その名前のバッファがすでに存在していれば、そのバッファを返します。そうでなければ新しいバッファを生成します。そのバッファはカレント・バッファとはなりません。つまりこの関数はカレント・バッファを変更しません。

name が文字列でない場合は、エラーが通知されます。

```
(get-buffer-create "foo")
     ⇒ #<buffer foo>
```

新たなバッファのメジャー・モードは Fundamental モードとなります。変数 `default-major-mode` はもっと上位の階層で処理されます。 20.1.3 節「主モードの自動選択の仕組み」 p.323 を参照してください。

generate-new-buffer *name* Function

この関数は、新たに生成した空のバッファを返します。しかしそのバッファはカレント・バッファとはなりません。 *name* という名前のバッファがすでに存在しない場合は、そ

れが新しいバッファの名前となります。その名前が使用されている場合は、この関数は *name* の後ろに ‘<*n*>’ (*n* は整数) という形式の接尾子を加えます。この関数は、使用できる整数を、 2 から順に調べて決定します。

name が文字列でない場合はエラーが通知されます。

```
(generate-new-buffer "bar")
     ⇒ #<buffer bar>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<2>>
(generate-new-buffer "bar")
     ⇒ #<buffer bar<3>>
```

新たなバッファの主モードは Fundamental モードとなります。変数 `default-major-mode` はもっと上位の階層で処理されます。 20.1.3 節「主モードの自動選択の仕組み」 p.323 を参照してください。

関連する関数として 24.3 節「バッファの名前」 p.395 の `generate-new-buffer-name` を参照してください。

24.10 バッファの消去

『バッファの消去 (killing a buffer)』によって、 Emacs からその名前が忘れ去られ、そのテキスト空間がほかの目的に使用できるようになります。

消去されたバッファのバッファ・オブジェクトは、それが参照される間は存在し続けますが、それをカレント・バッファにしたり表示したりはできないように特別な印がつけられます。しかし消去されたバッファもその独自性は持ち続けます。よって二つのバッファが消去されたとしても `eq` によって区別することができます。

消去されたバッファがカレント・バッファであったりウィンドウに表示されていた場合は、 Emacs は自動的に他のバッファを選択あるいは表示します。これはバッファの消去が一般にカレント・バッファを変更することを意味します。したがって、バッファを消去するときは (消去するバッファがカレント・バッファでないことが分かっているのでないかぎり)、カレント・バッファが変更される場合にしなければならない予防措置を取らなければなりません。 24.2 節「カレント・バッファ」 p.393 を参照してください。

一つもしくは複数の間接バッファの元バッファ (base buffer) を消去する場合は、その間接バッファも全て自動的に消去されます。

消去されたバッファの `buffer-name` は `nil` となります。この性質をバッファが消去されたか否かを調べるのに用いることができます。

```
(defun buffer-killed-p (buffer)
  "Return t if BUFFER is killed."
  (not (buffer-name buffer)))
```

kill-buffer *buffer-or-name* Command

この関数は、バッファ *buffer-or-name* を消去します。そしてその全てのメモリを他の目的に使用するため、もしくはオペレーティング・システムに返すために開放します。この関数は `nil` を返します。

関数 `process-buffer` がこのバッファを返す全てのプロセスに、 `SIGHUP` シグナルが送られます。これにより通常そのプロセスは終了します (`SIGHUP` は元は電話回線が遮断されたときに通知されるものでした)。 33.5 節「プロセスの削除」 p.565 を参照してください。

バッファが、あるファイルに対応づけられていて、保存されていない変更がある場合は、`kill-buffer` はバッファが消去されるまえにユーザに確認を行ないます。この確認は対話的に呼ばれなかったときにも行なわれます。確認の要求が起きないようにするためは、`kill-buffer` を呼ぶ前に、変更フラグを解除してください。 24.5 節「バッファの変更」 p.399 を参照してください。

すでに消去されているバッファを消去しても無視されるだけです。

```
(kill-buffer "foo.unchanged")
     ⇒ nil
(kill-buffer "foo.changed")

---------- Buffer: Minibuffer ----------
Buffer foo.changed modified; kill anyway? (yes or no) yes
---------- Buffer: Minibuffer ----------

     ⇒ nil
```

kill-buffer-query-functions Variable

保存されていない変更の確認のあと、 `kill-buffer` はリスト `kill-buffer-query-functions` の中の関数を、その順序で引数なしで呼びます。その呼出しの際には、消去されようとしているバッファがカレント・バッファになっています。これらの関数は、いくつかの非標準的な理由のためにユーザに確認を求めるために使用されます。もしそのいずれかが `nil` を返した場合は、 `kill-buffer` はバッファを消去しません。

kill-buffer-hook Variable

これは、全ての問い合わせのあと、バッファを実際に消去する直前に `kill-buffer` によって実行される通常のフックです。フック関数が実行されるときは、消去されるバッファがカレントとなります。 20.4 節「フック」 p.335 を参照してください。

buffer-offer-save Variable

この変数があるバッファで `nil` でない場合は、 `save-buffers-kill-emacs` と `save-some-buffers` は、通常ファイルに対応づけられたバッファを保存しようとするのと同じように、そのバッファを保存します。変数 `buffer-offer-save` は、何らかの理由で設定されたときに自動的にバッファローカルとなります。 10.9 節「バッファローカルな変数」 p.140 を参照してください。

24.11 間接バッファ

『間接バッファ (indirect buffer)』は、間接バッファの『元バッファ (base buffer)』と呼ばれる他のバッファのテキストを共有します。いくつかの点でファイルでのシンボリック・リンクに類似します。元バッファ自身がその間接バッファであってはなりません。

間接バッファのテキストは、常に元バッファのものと同一です。いずれかを編集することによる変化は、直ちにもう一方でも表示されます。文字と同様にテキスト属性も含みます。

しかし、他の全ての点で間接バッファとその元バッファとは完全に別のものです。名前も、ポイントの値も、ナローイングも、マーカやオーバレイも (ただし、いずれかのバッファでのテキストの追加や削除は、両方のバッファのマーカとオーバレイを更新しますが)、主モードも、ローカル変数も、全て異なります。

主バッファはファイルに対応づけられますが、間接バッファはファイルに対応づけられません。間接バッファを保存しようとすると、実際には主バッファが保存されます。

間接バッファを消去しても、主バッファにはなんの影響もありません。主バッファの消去は実質的にその間接バッファを消去するため、その間接バッファが 2 度とカレント・バッファとなることはありません。

make-indirect-buffer *base-buffer name* Command

これは名前が *name* で主バッファが *base-buffer* である間接バッファを生成します。引数 *base-buffer* はバッファもしくは文字列です。

base-buffer が間接バッファである場合は、その主バッファが新たなバッファの主バッファになります。

buffer-base-buffer *buffer* Function

この関数は *buffer* の主バッファを返します。 *buffer* が間接バッファでない場合は、値は `nil` となります。間接バッファである場合の値は、間接バッファではない他のバッファとなります。

25 ウィンドウ

この章では、 Emacs のウィンドウに関連する、関数と変数のほぼすべてについて述べます。ウィンドウ内でのテキスト表示のされ方については、第 35 章「Emacs の表示」 p.601 を参照してください。

25.1 Emacs ウィンドウの基本概念

Emacs における『ウィンドウ (window)』とは、バッファが表示される、画面上の物理的な領域です。この用語はまた、 Emacs Lisp において、画面領域を表現する Lisp オブジェクトを参照するためにも使用されます。どちらを意味しているのかは前後関係により明らかなはずです。

Emacs は、ウィンドウをフレームへとグループ分けします。フレームは、 Emacs が使用可能な画面の領域を表現します。各フレームは、常に少なくとも一つのウィンドウを含みますが、それらをさらに複数のオーバーラップしない Emacs ウィンドウへと垂直、あるいは水平に分割することができます。

各フレームにおいては、常に唯一一つのウィンドウが、『そのフレーム内で選択された (selected within the frame)』ものとして指示されています。そのウィンドウ内にフレームのカーソルが現われます。常に一つのフレームが選択されたフレームであり、そのフレーム内で選択されているウィンドウが『選択されたウィンドウ (selected window)』です。通常、選択されたウィンドウのバッファがカレント・バッファです (`set-buffer` の使用中を除く)。 24.2 節「カレント・バッファ」 p.393 を参照してください。

実際上、ウィンドウは、フレーム内に表示されている間のみ存在します。いったんフレームから取り除かれると、たとえまだ他の *Lisp* オブジェクトから参照されていても、そのウィンドウは事実上消去され、使われなくなります。保存されたウィンドウ構成を復元することが、もはや画面上にないウィンドウを復活させるための唯一の方法です (25.3 節「ウィンドウの消去」 p.411 を参照してください)。

各ウィンドウは、以下の属性をもっています:

- それを含んでいるフレーム
- ウィンドウの高さ
- ウィンドウの幅
- 画面、またはフレームを基準にしたウィンドウの角
- 表示しているバッファ
- ウィンドウの左上角に対応する、バッファ内での位置
- 水平スクロールしている桁数
- ポイント
- マーク
- そのウィンドウが選択されたのはどのくらい前か

ユーザは、複数のウィンドウを生成することにより、同時にいくつかのバッファを見ることができます。 Lisp ライブラリはさまざまな目的で複数のウィンドウを使用しますが、その最も多くは、互いに関係のある情報を表示するためです。たとえば Rmail では、片方のウィンドウ内のサマリ・バッファを動き回ることができ、それと同時にもう一方のウィンドウが、一度に一つのメッセージを、それが選択されるにしたがって表示します。

Emacs における “ウィンドウ” の意味は、 X のような汎用ウィンドウ・システムにおける意味と似ていますが、同一ではありません。 X ウィンドウ・システムは、画面上に X ウィンドウ群を配置します。そして Emacs は、一つ以上の X ウィンドウをフレームとして使用するのです。 Emacs をキャラクタ端末で使用するとき、 Emacs は端末画面全体を一つのフレームとして扱います。

たいていのウィンドウ・システムは、任意に配置可能なオーバーラッピング・ウインドウをサポートしています。対照的に、 Emacs のウィンドウ群は『タイル張り (tile)』されます。それらは決してオーバーラップせず、それら全部で画面またはフレーム全体を満たします。 Emacs が新しいウィンドウを生成したりリサイズするための方法により、 Emacs のフレーム上に、考えられるかぎりの数のウィンドウのタイルを配置することはできません。 25.2 節「ウィンドウの分割」 p.408 と 25.13 節「ウィンドウのサイズ」 p.427 を参照してください。

ウィンドウのバッファの内容が、どのようにそのウィンドウ内に表示されるのかについては、第 35 章「Emacs の表示」 p.601 を参照してください。

windowp *object* Function

この関数は、 *object* がウィンドウであれば `t` を返します。

25.2 ウィンドウの分割

以下で述べられている関数は、一つのウィンドウを二つに分割するために使用されるプリミティブです。このうち二つの高水準関数、 `pop-to-buffer` と `display-buffer` は、ウィンドウを分割しないこともあります (25.7 節「ウィンドウ内でのバッファの表示」 p.416 参照)。

以下で述べられている関数は、引数としてバッファを受け付けません。二つに分割された “半分ずつ” のウィンドウは、最初、分割されたもとのウィンドウが表示していたのと同じバッファを表示します。

split-window &optional *window size horizontal* Command

この関数は、 *window* を二つのウィンドウへと分割します。もとのウィンドウ *window* は選択されたウィンドウであり続けますが、以前の画面領域の一部分のみを占めます。その残りは、この関数の値として返される、新たに生成されたウィンドウによって占められます。

horizontal が非 `nil` の場合、 *window* は、二つの横並びのウィンドウへと分割されます。もとのウィンドウ *window* は、その左端 *size* 桁を維持し、残りの桁を新しいウィンドウに譲ります。 *horizontal* が `nil` の場合、 *window* は上下に分割され、その上部 *size* 行分を維持して残りの行を新しいウィンドウに譲ります。したがって、もとのウィンドウは二つのうちの左側あるいは上に、新しいウィンドウは右側あるいは下となります。

window が省略されるか nil の場合、選択されたウィンドウが分割されます。 *size* が省略されるか nil の場合、 *window* は均等に二つの部分に分割されます (もし端数の行があれば、それは新しいウィンドウに割り当てられます)。 split-window が対話的に呼び出されたとき、すべての引数は nil となります。

以下の例では、高さ 50 行、幅 80 桁の画面上唯一のウィンドウから始めて、そのウィンドウを分割しています。

```
(setq w (selected-window))
     ⇒ #<window 8 on windows.texi>
(window-edges)          ; 角 (以下の順):
     ⇒ (0 0 80 50)      ;    左 – 上 – 右 – 下
;; 生成されたウィンドウを返す
(setq w2 (split-window w 15))
     ⇒ #<window 28 on windows.texi>
(window-edges w2)
     ⇒ (0 15 80 50)     ; 下のウィンドウ;
                        ;    上端が 15 行目
(window-edges w)
     ⇒ (0 0 80 15)      ; 上のウィンドウ
```

画面は以下のようになっています:

```
         __________
        |          |   0 行目
        |    w     |
        |__________|
        |          |  15 行目
        |    w2    |
        |__________|
                      50 行目
       0 桁目      80 桁目
```

次に、上のウィンドウを水平に分割します:

```
(setq w3 (split-window w 35 t))
     ⇒ #<window 32 on windows.texi>
(window-edges w3)
     ⇒ (35 0 80 15)  ; 左の角が 35 桁目に
(window-edges w)
     ⇒ (0 0 35 15)   ; 右の角が 35 桁目に
(window-edges w2)
     ⇒ (0 15 80 50)  ; 下のウィンドウは変化なし
```

ここで、画面は以下のようになっています:

```
              35 桁目
           __________
          |   |      |    0 行目
          | w |  w3  |
          |___|______|
          |          |   15 行目
          |    w2    |
          |__________|
                         50 行目
        0 桁目       80 桁目
```

普通、 Emacs は、二つの横並びのウィンドウの間にスクロール・バー (26.3.3 節「X ウインドウ・フレーム・パラメータ」 p.438 参照) か '|' 文字を表示します。表示テーブルを使えば、代わりのボーダー文字を指定することができます。 35.14 節「表示テーブル」 p.618 参照。

split-window-vertically *size* Command

この関数は、選択されたウィンドウに *size* 行を残しつつ、そのウィンドウを上下二つのウィンドウへと分割します。

この関数は、単に `split-window` へのインタフェースとなっています。以下に、その完全な関数定義を示します:

```
(defun split-window-vertically (&optional arg)
  "Split current window into two windows, one above the other."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg))))
```

split-window-horizontally *size* Command

この関数は、選択されたウィンドウに *size* 桁を残しつつ、そのウィンドウを二つの横並びのウィンドウへと分割します。

この関数は、単に `split-window` へのインタフェースとなっています。以下に、 `split-window-horizontally` の完全な定義を示します (説明文字列の部分を除く):

```
(defun split-window-horizontally (&optional arg)
  "Split selected window into two windows, side by side..."
  (interactive "P")
  (split-window nil (and arg (prefix-numeric-value arg)) t))
```

one-window-p &optional *no-mini all-frames* Function

この関数は、ウィンドウが一つしか存在しない場合に非 `nil` を返します。引数 *no-mini* は、もし非 `nil` なら、たとえミニバッファがアクティブでもそれを数えないことを意味し、 `nil` ならアクティブなミニバッファ・ウィンドウをウィンドウ数の合計に含めます。その合計が 1 と比較されます。

引数 *all-frames* は、どのフレームについて考えるのかを指定します。以下に、取り得る値とそれらの意味を示します:

`nil` 選択されたフレーム内のウィンドウと、たとえ別のフレーム内にあっても、そのフレームが使用しているミニバッファを数えます。

`t` 既存のすべてのフレーム内のすべてのウィンドウを数えます。

`visible` すべての可視フレーム内のすべてのウィンドウを数えます。

0 すべての可視フレームとアイコン化されたフレーム内の、すべてのウィンドウを数えます。

その他 選択されたフレーム内のウィンドウをきっかり数え、その他は数えません。

25.3 ウィンドウの消去

ウィンドウは、ウィンドウを『消去 (delete)』するための関数を呼び出して消去しないかぎり、そのフレーム上で可視であり続けます。消去されたウィンドウは画面上に現われることはできませんが、 Lisp オブジェクトとしては、それへの参照がなくなるまで存在し続けます。保存したウィンドウ構成の復元以外に、ウィンドウの消去を取り消すための方法はありません (25.16 節「ウィンドウ構成」 p.432 参照)。ウィンドウ構成の復元はまた、その構成に含まれないウィンドウをみんな消去してしまいます。

ウィンドウを消去するとき、それによって取り上げられる場所は、隣接する一個の同胞ウィンドウ (sibling) に与えられます (Emacs バージョン 18 では、その場所は、隣接するすべての同胞ウィンドウに均等に配分されていました)。

window-live-p *window* Function

この関数は、もし *window* が消去されていれば `nil` を返し、そうでなければ `t` を返します。

警告!: 消去されたウィンドウをまだ生きているかのように使用した場合、結果として誤った情報、あるいは致命的エラーが生じます。

delete-window &optional *window* Command

この関数は、 *window* を表示から取り除きます。 *window* が省略された場合、選択されたウィンドウが消去されます。 `delete-window` が呼び出されたとき、ウィンドウが一つしか存在しなければエラーが通知されます。

この関数は `nil` を返します。

`delete-window` が対話的に呼び出されたとき、 *window* は選択されたウィンドウをデフォルトとします。

delete-other-windows &optional *window* Command

この関数は、フレーム内の他のウィンドウを消去することにより、 *window* をそのフレームでの唯一のウィンドウにします。 *window* が省略されるか `nil` の場合、選択されたウィンドウがデフォルトとして使用されます。

この関数は `nil` を返します。

delete-windows-on *buffer* &optional *frame* Command

この関数は、 *buffer* を表示しているすべてのウィンドウを消去します。 *buffer* を表示しているウィンドウがなければ何もしません。

`delete-windows-on` は、フレーム単位で作用します。フレーム内に異なるバッファを表示しているウィンドウがいくつかある場合、そのうち *buffer* を表示しているものが取り除かれ、その余白を満たすために他のウィンドウが広がります。フレーム内のすべてのウィンドウが *buffer* を表示している場合 (ウィンドウが一つしか存在しない場合を含む)、そのフレームは、 `other-buffer` で選択された別のバッファを表示する一個のウィンドウを保持するようになります。 24.8 節「バッファ・リスト」p.402 を参照してください。

引数 *frame* は、どのフレームに作用するのかを制御します:

- `nil` の場合、選択されたフレームに作用します。
- `t` の場合、すべてのフレームに作用します。
- `visible` の場合、すべての可視フレームに作用します。
- 0 の場合、すべての可視フレームとアイコン化されたフレームに作用します。
- フレームが指定された場合、そのフレームに作用します。

この関数は、常に `nil` を返します。

25.4 ウィンドウの選択

ウィンドウが選択されたとき、そのウィンドウ内のバッファがカレント・バッファとなり、その中にカーソルが現われます。

selected-window Function

この関数は、選択されたウィンドウを返します。これは、カーソルが現われるウィンドウであり、多くのコマンドが適用されるウィンドウでもあります。

select-window *window* Function

この関数は、 *window* を選択されたウィンドウにします。その後、 *window* 内にカーソルが現われます (再表示において)。 *window* 内に表示されているバッファは、直ちにカレント・バッファとして指示されます。

戻り値は *window* です。

```
(setq w (next-window))
(select-window w)
     ⇒ #<window 65 on windows.texi>
```

save-selected-window *forms...* Macro

このマクロは、選択されたウィンドウを記録して *forms* を次々に実行した後、はじめに選択されていたウィンドウを復元します。ウィンドウのサイズや配置、内容については保存も復元もしません。したがって、 *forms* がそれらを変更すると、その変更は永久的なものとなります。

以下の関数は、種々の選択基準をもとにして画面上のウィンドウの一つを選択します。

get-lru-window &optional *frame* Function

この関数は、最も昔に“使用された”(つまり選択された) ウィンドウを返します。選択されたウィンドウは、常に、最も最近に使用されたウィンドウとなります。

選択されたウィンドウが唯一のウィンドウである場合、それは最も昔に使用されたウィンドウにもなります。新たに生成されるウィンドウは、それが選択されるまで、最も昔に使用されたウィンドウとなります。ミニバッファ・ウィンドウは決してその候補にはなりません。

引数 *frame* は、どのウィンドウ群について考えるのかを制御します。

- `nil` の場合、選択されたフレーム上のウィンドウについて考えます。
- `t` の場合、すべてのフレーム上のウィンドウについて考えます。
- `visible` の場合、すべての可視フレーム上のウィンドウについて考えます。
- 0 の場合、すべての可視フレームとアイコン化されたフレーム上のウィンドウについて考えます。
- フレームが指定された場合、そのフレーム上のウィンドウについて考えます。

get-largest-window &optional *frame* Function

この関数は、最も広い面積 (高さ×幅) を占めるウィンドウを返します。横並びのウィンドウがない場合、最も行数の多いウィンドウがこれにあたります。ミニバッファ・ウィンドウは決してその候補にはなりません。

同じサイズのウィンドウが二つある場合、この関数は、選択されたウィンドウを起点としてウィンドウの循環順序 (次節を参照) をたどり、先に出てきたウィンドウを返します。

引数 *frame* は、どのウィンドウ集合について考えるのかを制御します。上述の `get-lru-window` 参照。

25.5 ウィンドウの循環順序

次のウィンドウを選択するために、コマンド `C-x o` (`other-window`) を使用するとき、(訳注: 選択されるウィンドウは) 画面上のすべてのウィンドウを通じて一定の循環順序で移動します。与えられたどんなウィンドウ構成に対しても、この順序は決して変わりません。これは、『ウィンドウの循環順序 (cyclic ordering of windows)』と呼ばれます。

一般に、この順序は上から下へ、また左から右へ進みます。ですがウィンドウが分割された順序によっては、先に下へ、あるいは先に右へ進むかもしれません。

もし最初の分割が垂直 (上下のウィンドウ) であり、その後それら子ウィンドウが水平に分割されていたなら、その順序はフレームの上部において左から右、次にフレームの下部において左から右、というふうになります。もし最初の分割が水平だったなら、その順序はまず左側において上から下、というふうになります。一般にウインドウ・ツリー内のどの階層においても、各同胞ウィンドウ間では、その順序は左から右、あるいは上から下となります。

next-window &optional *window minibuf all-frames* Function

この関数は、ウィンドウの循環順序の中で *window* の次にあたるウィンドウを返します。これは、 *window* が選択されているときに、 `C-x o` の打鍵によって選択されるウィンドウです。 *window* が唯一の可視ウィンドウである場合、この関数は *window* を返します。省略された場合、 *window* は選択されたウィンドウをデフォルトとします。

引数 *minibuf* の値は、ウィンドウ順序にミニバッファを含めるか否かを決定します。*minibuf* が `nil` の場合、もしミニバッファが現在アクティブならばウィンドウ順序に含めます。これは、 `C-x o` の挙動です (ミニバッファの使用中、ミニバッファ・ウィンドウがアクティブとなります。第 17 章「ミニバッファ」 p.237 を参照してください)。

minibuf が `t` の場合、たとえアクティブでなくてもミニバッファ・ウィンドウを循環順序に含めます。

minibuf が `t` でも `nil` でもない場合、たとえアクティブでもミニバッファ・ウィンドウを循環順序に含めません。

引数 *all-frames* は、どのフレームについて考えるのかを指定します。以下に、取り得る値とそれらの意味を示します:

`nil`
: *window* のフレーム内のすべてのウィンドウと、たとえ別のフレーム内にあっても、そのフレームが使用しているミニバッファについて考えます。

`t`
: 既存のすべてのフレーム内のすべてのウィンドウについて考えます。

`visible`
: すべての可視フレーム内のすべてのウィンドウについて考えます (有用な結果を得るには、 *window* が可視フレーム内にあることが保証されなくてはなりません)。

0
: すべての可視フレームとアイコン化されたフレーム内の、すべてのウィンドウについて考えます。

その他
: 厳密に *window* のフレーム内のウィンドウについてのみ考え、その他については考えません。

以下の例は、二つのウィンドウがあり、両方ともバッファ ‘`windows.texi`’ を表示しているものと仮定しています:

```
(selected-window)
     ⇒ #<window 56 on windows.texi>
(next-window (selected-window))
     ⇒ #<window 52 on windows.texi>
(next-window (next-window (selected-window)))
     ⇒ #<window 56 on windows.texi>
```

previous-window &optional *window minibuf all-frames* Function

この関数は、ウィンドウの循環順序の中で *window* の前にあたるウィンドウを返します。その他の引数は、 `next-window` と同様に、どのウィンドウを循環順序に含めるのかを指定するものです。

other-window *count* Command

この関数は、循環順序の中で *count* 番目後ろにあたるウィンドウを選択します。 *count* が負の場合、 *count* 番目前のウィンドウを選択します。 `nil` を返します。

対話的な呼出しにおいては、 *count* は、数値前置引数となります。

walk-windows *proc* &optional *minibuf all-frames* Function

この関数は、各ウィンドウに対して一度ずつ、そのウィンドウを単独の引数にして `proc` を呼び出しながら、すべてのウィンドウを一周します。

オプション引数 *minibuf* と *all-frames* は、そのスキャンの対象に含めるウィンドウ集合を指定します。詳細については、上述の `next-window` を参照。

25.6 バッファとウィンドウ

この節では、ウィンドウを調べたり、厳密な制御のもとでウィンドウにバッファを表示したりする低水準関数について述べます。使用するウィンドウを (訳注: 自動的に) 見つけ、指定したバッファの表示にそれを使用する関数については、次節を参照してください。そこで述べられている関数はこれらよりも簡単に使用できますが、ウィンドウの選択と生成において、ヒューリスティクスに頼る部分があります。完璧な制御を必要とする場合は、以下の関数を使用します。

set-window-buffer *window buffer-or-name* Function

この関数は、 *window* に対し、その内容として *buffer-or-name* を表示させせます。 `nil` を返します。

```
(set-window-buffer (selected-window) "foo")
     ⇒ nil
```

window-buffer &optional *window* Function

この関数は、 *window* が表示しているバッファを返します。 *window* が省略された場合、この関数は、選択されたウィンドウのバッファを返します。

```
(window-buffer)
     ⇒ #<buffer windows.texi>
```

get-buffer-window *buffer-or-name* &optional *all-frames* Function

この関数は、現在 *buffer-or-name* を表示しているウィンドウか、あるいはそれがなければ `nil` を返します。該当するウィンドウがいくつもある場合、この関数は、選択されたウィンドウを起点としてウィンドウの循環順序をたどり、先に出てきたウィンドウを返します。 25.5 節「ウィンドウの循環順序」 p.413 を参照してください。

引数 *all-frames* は、どのウィンドウ群について考えるのかを制御します。

- `nil` の場合、選択されたフレーム上のウィンドウについて考えます。
- `t` の場合、すべてのフレーム上のウィンドウについて考えます。
- `visible` の場合、すべての可視フレーム上のウィンドウについて考えます。
- 0 の場合、すべての可視フレームとアイコン化されたフレーム上のウィンドウについて考えます。

- フレームが指定された場合、そのフレーム上のウィンドウについて考えます。

25.7 ウィンドウ内でのバッファの表示

この節では、ウィンドウを自動的に選択して、指定したバッファの表示にそれを使用してくれる便利な関数について述べます。これらの関数は、特定の状況において、既存のウィンドウを分割することがあります。ここではまた、ウィンドウの選択に要するヒューリスティクスを特性づける変数についても述べます。より厳密な制御を可能とする低水準関数については、前節を参照してください。

あるバッファを、 Lisp プログラムがアクセスあるいは変更できるようするためだけにカレントにする場合は、この節の関数群を使用しないでください。これらは、無条件にウィンドウ内のバッファの表示を変更してユーザを驚かしてしまうため、その用途には強烈すぎるのです。代わりに、ウィンドウ内のバッファの表示に影響を与えず、プログラムによるアクセスのためにバッファをカレントに指示できる、 `set-buffer` (24.2 節「カレント・バッファ」 p.393 参照) と `save-excursion` (27.3 節「脱線」 p.466 参照) を使用してください。

switch-to-buffer *buffer-or-name* &optional *norecord* Command

この関数は、 *buffer-or-name* をカレント・バッファにし、選択されたウィンドウにそのバッファを表示します。これは、そのバッファを人間が見ることができ、また以降のキーボード・コマンドがそこへ適用されることを意味しています。 `set-buffer` と比べてみると、 `set-buffer` は *buffer-or-name* をカレント・バッファにはしますが、それを選択されたウィンドウに表示しません。 24.2 節「カレント・バッファ」 p.393 を参照してください。

buffer-or-name が既存のいずれのバッファにも一致しない場合、その名前の新しいバッファが生成されます。新しいバッファの主モードは、変数 `default-major-mode` にしたがって設定されます。 20.1.3 節「主モードの自動選択の仕組み」 p.323 を参照してください。

指定されたバッファは、普通、バッファ・リストの先頭に置かれます。これは `other-buffer` の動作に影響します。ですが *norecord* が非 `nil` の場合は、そのかぎりではありません。 24.8 節「バッファ・リスト」 p.402 を参照してください。

`switch-to-buffer` 関数は、しばしばキー・バインド `C-x b` として対話的に使用されます。また、プログラム中でも頻繁に使用されます。常に `nil` を返します。

switch-to-buffer-other-window *buffer-or-name* Command

この関数は、 *buffer-or-name* をカレント・バッファにし、選択されていないウィンドウに表示します。その後、そのウィンドウを選択します。バッファに関する操作は `switch-to-buffer` と同じです。

現在選択されているウィンドウは、絶対にこの処理に使用されることはありません。もしそれが唯一のウィンドウであれば、この処理用に別個のウィンドウを作るために、そのウィンドウが分割されます。選択されたウィンドウがすでにそのバッファを表示している

場合、それはそのままですが、それでもなお、そのバッファを表示するための別のウィンドウがさらに獲得されます。

pop-to-buffer *buffer-or-name* &optional *other-window* Function

この関数は、 *buffer-or-name* をカレント・バッファにし、できるだけその直前に選択されていたウィンドウを避けて、いずれかのウィンドウにそのバッファを表示します。その、 "pop" されるウィンドウは、それ自身のフレーム内での選択されたウィンドウとなります。

変数 `pop-up-frames` が非 `nil` の場合、 `pop-to-buffer` は、すでにそのバッファを表示している可視フレーム内のウィンドウを探します。該当するウィンドウがあればそれを返し、そのフレーム内での選択されたウィンドウとします。なければ新しいフレームを生成し、その中に *buffer-or-name* を表示します。

`pop-up-frames` が `nil` の場合、 `pop-to-buffer` は、もっぱら選択されたフレーム内で動作します (選択されたフレームがミニバッファのみを保持している場合、 `pop-to-buffer` は、最も最近に選択されていた、ミニバッファだけではないフレーム内で動作します)。

変数 `pop-up-windows` が非 `nil` の場合、もとのウィンドウとは別の新しいウィンドウを生成するために、ウィンドウが分割されることがあります。詳細については 25.8 節「表示に用いるウィンドウの選択」p.418 を参照してください。

other-window が非 `nil` の場合、 `pop-to-buffer` は、たとえ選択されたウィンドウですでに *buffer-or-name* が表示されていても、別のウィンドウを探すか生成します。そのため結果として、 *buffer-or-name* が二つのウィンドウに表示されるかもしれません。一方、選択されたウィンドウですでに *buffer-or-name* が表示されており、かつ *other-window* が `nil` の場合、 *buffer-or-name* の表示としては選択されたウィンドウで十分とみなされ、よって何もする必要がありません。

`display-buffer` に影響するすべての変数は、同様に `pop-to-buffer` にも影響します。25.8 節「表示に用いるウィンドウの選択」p.418 を参照してください。

buffer-or-name が既存のバッファ名以外を示す文字列の場合、その名前のバッファが生成されます。新しいバッファの主モードは、変数 `default-major-mode` にしたがって設定されます。 20.1.3 節「主モードの自動選択の仕組み」p.323 を参照してください。

replace-buffer-in-windows *buffer* Command

この関数は、 *buffer* を表示しているすべてのウィンドウにおいて、それを別のバッファで置き換えます。別のバッファとは、 `other-buffer` で選択されるバッファです。この関数の通常の利用においては、別のバッファとしてどれが使用されるのかを気にしないで、*buffer* がもはや表示されないことだけを期待してください。

この関数は `nil` を返します。

25.8 表示に用いるウィンドウの選択

この節では、バッファを表示するためにウィンドウを選択する、という基本的な機能を、 display-buffer に関して述べます。すべての高水準関数とコマンドはこのサブルーチンを使用します。以下に、 display-buffer の使用法とカスタマイズ方法を述べます。

display-buffer *buffer-or-name* &optional *not-this-window* Command

このコマンドは、 pop-to-buffer のように、 *buffer-or-name* をいずれかのウィンドウで見えるようにしますが、そのウィンドウを選択せず、またそのバッファをカレントにもしません。選択されたウィンドウの同一性 (identity) が、この関数によって変更されることはありません。

not-this-window が非 nil の場合、それは、たとえ指定されたバッファが選択されたウィンドウですでに表示されていても、それを別のウィンドウに表示することを意味します。このため、そのバッファが同時に二つのウィンドウで見えるようになることがあります。 *not-this-window* が nil の場合、 *buffer-or-name* がすでにいずれかのウィンドウで表示されていればそれで十分であり、よってこの関数は何もしません。

display-buffer は、 *buffer-or-name* の表示のために選択したウィンドウを返します。

display-buffer が、厳密にはどのようにウィンドウを探し生成するのかは、以下に述べる変数に依存しています。

pop-up-windows User Option

この変数は、 display-buffer が新しいウィンドウを作るか否かを制御します。これが非 nil で、かつウィンドウが一つしかない場合、そのウィンドウが分割されます。 nil の場合、 display-buffer は一つのウィンドウを分割はしませんが、その全体を使用します。

split-height-threshold User Option

この変数は、複数のウィンドウがあるとき、どんな場合に display-buffer がウィンドウを分割できるのかを決定します[1]。最大のウィンドウが少なくともこの行数分ある場合、 display-buffer は、常にそれを分割します。最大のウィンドウがこの高さに満たない場合は、それが単独のウィンドウでかつ pop-up-windows が非 nil のときにのみ、そのウィンドウを分割します。

pop-up-frames User Option

この変数は、 display-buffer が新しいフレームを作るか否かを制御します。非 nil の場合、 display-buffer は、すでにいずれかの可視フレーム上で所望のバッファを表示している既存のウィンドウを探します。もし見つかれば、そのウィンドウを返します。見つからなければ、新しいフレームを作ります。 pop-up-frames が非 nil の場合、変数 pop-up-windows と split-height-threshold は無視されます。

pop-up-frames が nil の場合、 display-buffer は、ウィンドウを分割するか再利用します。

---

[1] 訳注: 実は複数のウィンドウがある場合にも、分割は起こり得ます。

より多くの情報については、第 26 章「フレーム」p.435 を参照してください。

pop-up-frame-function Variable

この変数では、 `pop-up-frames` が非 `nil` の場合に、どうやって新しいフレームを作るのかを指定します。

その値は、引数をもたない関数です。 `display-buffer` が新しいフレームを作るとき、一個のフレームを返すこの関数を呼び出すことによって、その処理を行ないます。この変数のデフォルト値は、 `pop-up-frame-alist` 内のパラメータを用いてフレームを生成する関数です。

pop-up-frame-alist Variable

この変数は、 `display-buffer` が新しいフレームを作る際に使用する、フレーム・パラメータを指定した alist を保持します。フレーム・パラメータに関するより多くの情報については、 26.3 節「フレーム・パラメータ」p.437 を参照してください。

special-display-buffer-names Variable

特別に表示されるべきバッファの、バッファ名のリストです。もしバッファ名がこのリスト中にあれば、 `display-buffer` は、そのバッファを特別に扱います。

デフォルトでは、特別な表示とは、そのバッファに専用のフレームを与えることを意味します。

もし、その要素が文字列でなくリストなら、そのリストの CAR はバッファ名で、残りはそのフレームをどうやって生成するのかを表わします。リストの残りとしては二通りの可能性があります。それは、フレーム・パラメータを指定する alist でもあり得るし、あるいは関数とそれに与える引数[2]を含むこともできます (その関数の第一引数は表示対象のバッファです。第二引数以下に、このリストで与える引数がきます)。

special-display-regexps Variable

特別に表示されるべきバッファを指定する、正規表現のリストです。もしバッファ名がこのリスト中のいずれかの正規表現に一致すれば、 `display-buffer` は、そのバッファを特別に扱います。

デフォルトでは、特別な表示とは、そのバッファに専用のフレームを与えることを意味します。

もし、その要素が文字列でなくリストなら、そのリストの CAR は正規表現で、残りはそのフレームをどうやって生成するのかを表わします。上述の `special-display-buffer-names` の下を参照。

special-display-function Variable

この変数は、バッファを特別に表示するために呼び出される関数を保持します。その関数とは、引数としてバッファを受け取り、そのバッファの表示に使用したウィンドウを返すものです。

この変数のデフォルト値は、 `special-display-popup-frame` です。

---

[2] 訳注: (関数名 第二引数 第三引数 ...) のようなリスト。

special-display-popup-frame *buffer* Function

この関数は、 *buffer* を、それ専用のフレームの中で可視にします。 *buffer* が、すでにいずれかのフレーム内のウィンドウに表示されていれば、そのウィンドウを使用すべく、そのフレームを可視化して上昇させます。そうでない場合、 *buffer* 専用のフレームを生成します。

この関数は、 *buffer* がそれ専用のフレームで表示されているか否かにかかわらず、そのバッファを表示している既存のウィンドウを使用します。とはいえ *buffer* が生成されるより前、つまり自分の初期化ファイルで上述の変数群を設定しておけば、おそらくそのウィンドウは、この関数によって前もって作られるでしょう。

special-display-frame-alist User Option

この変数は、 `special-display-popup-frame` がフレームを生成する際に使用する、フレーム・パラメータを保持します。

same-window-buffer-names Variable

選択されたウィンドウに表示されるべきバッファの、バッファ名のリストです。もしバッファ名がこのリスト中にあれば、 `display-buffer` は、そのバッファを選択されたウィンドウに表示するよう扱います。

same-window-regexps Variable

選択されたウィンドウに表示されるべきバッファを指定する、正規表現のリストです。もしバッファ名がこのリスト中のいずれかの正規表現に一致すれば、 `display-buffer` は、そのバッファを選択されたウィンドウに表示するよう扱います。

display-buffer-function Variable

この変数は、 `display-buffer` の挙動をカスタマイズする最も柔軟な手段です。非 `nil` の場合、それは、 `display-buffer` がその処理を行なうために呼び出す関数です。その関数は、 `display-buffer` と同じ二つの引数を受け付けるものです。そして、ウィンドウを選択あるいは生成して指定されたバッファを表示し、そのウィンドウを返します。

このフックは、前述した他のオプションやフックすべてに優先します。

ウィンドウは、そのバッファ “専用” としてマークづけされ得ます。そうすると `display-buffer` は、そのウィンドウの使用を試みるのをやめます。

window-dedicated-p *window* Function

この関数は、 *window* が専用としてマークされていれば `t` を返し、そうでなければ `nil` を返します。

set-window-dedicated-p *window flag* Function

この関数は、 *flag* が非 `nil` であれば *window* を専用としてマークし、そうでなければ非専用とします。

25.9 ウィンドウとポイント

各ウィンドウは、同じバッファを表示している別のウィンドウのそれとは独立した、自分自身のポイント値をもっています。これは、一つのバッファを表示する複数のウィンドウをもつ際に役立ちます。

- ウィンドウ・ポイントは、ウィンドウが最初に生成されたときに作られます。それはそのバッファのポイントで初期化されるか、あるいはすでにそのバッファを開いているウィンドウがあれば、そのウィンドウ・ポイントで初期化されます。
- ウィンドウの選択は、そのバッファのポイント値としてそのウィンドウのポイント値を設定します。逆にウィンドウの選択をやめると、そのウィンドウのポイントとしてそのバッファのポイントが設定されます。このように、ある一つのバッファを表示しているウィンドウ間を移動するとき、選択されたウィンドウのポイント値は実際にはそのバッファ内にあり、一方、他のウィンドウのポイント値はそれらのウィンドウに格納されています。
- 選択されたウィンドウがカレント・バッファを表示しているかぎり、そのウィンドウのポイントとバッファのポイントは常に一緒に移動し、それらは同じ値のままです。
- バッファ内位置の、より詳細については、第 27 章「位置」p.457 を参照してください。

ユーザが目にするかぎりでは、ポイントはカーソルの位置であり、ユーザが他のバッファへ切り替えるとき、カーソルはそのバッファのポイントの位置へジャンプします。

window-point *window* Function

この関数は、 *window* の現在のポイント位置を返します。選択されていないウィンドウに対しては、これは、そのウィンドウが選択された場合に (そのウィンドウのバッファにおいて) ポイントが取るであろう値となります。

window が選択されたウィンドウであり、かつそのバッファがカレント・バッファである場合、返される値はそのバッファのポイントと同じです。

厳密にいえば、 `save-excursion` 形式の外における “トップレベル” のポイント値を返すのがより正しい仕様でしょう。ですが、その値を見つけるのは困難なのです。

set-window-point *window position* Function

この関数は、 *window* のポイントを、 *window* のバッファ内の位置 *position* に位置させます。

25.10 ウィンドウ開始位置

各ウィンドウは、バッファ内のどこから表示を開始すべきかを示す “バッファ内位置” の追跡に使用する、マーカをもっています。この位置は、ウィンドウの『表示開始 (display-start)』位置 (あるいは単に『開始 (start)』) と呼ばれます。この位置の直後の文字が、そのウィンドウの左上角に現われる文字です。必然ではありませんが、これは一般にテキスト行の行頭に位置します。

window-start &optional *window* Function

この関数は、ウィンドウ *window* の表示開始位置を返します。 *window* が `nil` の場合、選択されたウィンドウが使用されます。例えば、

```
(window-start)
     ⇒ 7058
```

ウィンドウを生成するとき、あるいはその中に別のバッファを表示するとき、その表示開始位置には、その同じバッファで以前に使用されていた表示開始位置か、バッファがそれをもっていなければ 1 が設定されます。

現実的な例については、 27.2.4 節「テキスト行単位での移動」 p.460 の `count-lins` の記述を参照してください。

window-end &optional *window* Function

この関数は、ウィンドウ *window* における、表示の終了位置を返します。 *window* が `nil` の場合、選択されたウィンドウが使用されます。

単に、バッファのテキストの変更やポイントの移動をしても、 `window-end` が返す値は更新されません。その値は、 Emacs が再表示を行なって実際に再表示が完了したときにのみ更新されます。

もし、 *window* の最後の再表示が横取りされ完了していなければ、 Emacs にはそのウィンドウの表示終了位置がわかりません。その場合、この関数は間違った値を返します。将来のバージョンにおいては、 `window-end` は、この場合 `nil` を返すでしょう。

set-window-start *window position* &optional *noforce* Function

この関数は、 *window* の表示開始位置として、 *window* のバッファ内の位置 *position* を設定します。 *position* を返します。

表示ルーチン群は、バッファを表示する際、ポイントの位置が可視となることを強要します。普通、それらのルーチンは、ポイントを可視とするために必要なら常に表示開始位置を変更 (つまりウィンドウをスクロール) します。しかしながら、この関数で *noforce* に `nil` を用いて開始位置を指定する場合、それは、たとえポイントの位置が画面の外に置かれることになっても、 *position* の位置からの表示開始を要求することを意味します。これにより、もしポイントが画面の外に置かれる場合、表示ルーチンはポイントをそのウィンドウの真ん中の行の左端に移動させます。

たとえばポイントが 1 で、ウィンドウの開始位置を 2 に設定する場合、ポイントはウィンドウの上端の “さらに上” となるでしょう。再表示が生じたときにポイントがまだ 1 なら、表示ルーチンは自動的にポイントを移動させます。以下に例を示します:

```
;; 以下は set-window-start 式を実行する前の
;;   ‘foo’ の様子。
```

```
---------- Buffer: foo ----------
⋆This is the contents of buffer foo.
2
3
4
5
6
---------- Buffer: foo ----------

(set-window-start
 (selected-window)
 (1+ (window-start)))
⇒ 2

;; 以下は set-window-start 式を実行した後の
;;   ‘foo’ の様子。
---------- Buffer: foo ----------
his is the contents of buffer foo.
2
3
⋆4
5
6
---------- Buffer: foo ----------
```

noforce が非 `nil` で、かつ *position* が次の再表示の際にポイントを画面の外に置いてしまう場合、 (訳注: 表示ルーチンによる) 再表示はうまくポイントと両立できる新しいウィンドウ開始位置を計算し、そのため *position* は使用されません。

pos-visible-in-window-p &optional *position window* Function

この関数は、 *position* が、 *window* の画面上で現在可視なテキストの範囲内にあれば `t` を返します。もし *position* が、可視な範囲の外に垂直にスクロールされていれば `nil` を返します。引数 *position* は現在のポイントの位置をデフォルトとし、 *window* は選択されたウィンドウをデフォルトとします。以下に例を示します:

```
(or (pos-visible-in-window-p
     (point) (selected-window))
    (recenter 0))
```

`pos-visible-in-window-p` 関数は垂直スクロールについてのみ考えます。もし *position* が、 *window* が水平にスクロールされていることのみによって可視な範囲の外にあるなら、 `pos-visible-in-window-p` は `t` を返します。 25.12 節「水平スクロール」 p.426 を参照してください。

25.11 垂直スクロール

垂直スクロールとは、ウィンドウ内でのテキストの上下移動を意味します。これは、ウィンドウの表示開始位置の値を変更することによって機能します。垂直スクロールはまた、表示開始位置を画面上に保つために `window-point` の値を変更することがあります。

コマンド `scroll-up` と `scroll-down` において、その方向 “up” と “down” は、ウィンドウを通して見ているバッファ内のテキストの動き (訳注: ウィンドウの動きではなく) を指しています。長い一巻きの紙に書かれたテキストと、その紙を上下に動かすスクロール・コマンドを想像してください。このため、もしバッファの途中のテキストを見ていて繰り返し `scroll-down` を呼び出してゆくと、やがてはバッファの先頭が見えてくるでしょう。

一部の人々は、これとは反対の約束が用いられることを主張しています。彼らは、決まった場所にとどまるテキストの上をウィンドウが移動する、と考えています。その場合、 “down” コマンドがあなたをバッファの末尾に連れてゆくことでしょう。この見方は、ウィンドウとバッファ内のテキストとの実際の関係によく調和していますが、ユーザが目にするものとはあまり似通っていません。端末上のウィンドウの位置は移動しないし、簡単なスクロール・コマンドは画面上で明らかにテキストを上下に動かします。私たちは、ユーザの視点に合った名前を選択したのです。

スクロール関数 (`scroll-other-window` を除く) は、カレント・バッファと、選択されたウィンドウで表示されるバッファとが異なる場合、予測できない結果となります。 24.2 節「カレント・バッファ」p.393 を参照してください。

scroll-up &optional *count* Command

この関数は、選択されたウィンドウ内のテキストを *count* 行だけ上向きにスクロールします。 *count* が負の場合、スクロールは実際には下向きとなります。

count が `nil` の場合 (あるいは省略された場合)、スクロールの量は、そのウィンドウが使用できる高さ (モード行は数えない) よりも `next-screen-context-lines` 行分少なくなります。

`scroll-up` は `nil` を返します。

scroll-down &optional *count* Command

この関数は、選択されたウィンドウ内のテキストを *count* 行だけ下向きにスクロールします。 *count* が負の場合、スクロールは実際には上向きとなります。

count が省略されるか `nil` の場合、スクロールの量は、そのウィンドウが使用できる高さ (モード行は数えない) よりも `next-screen-context-lines` 行分少なくなります。

`scroll-down` は `nil` を返します。

scroll-other-window &optional *count* Command

この関数は、他のウィンドウ内のテキストを、 *count* 行だけ上向きにスクロールします。 *count* が負の値か `nil` の場合、 `scroll-up` と同じように扱います。

スクロールさせるバッファは、変数 `other-window-scroll-buffer` で指定することができます。選択されたウィンドウがミニバッファである場合、普通、次のウィンドウは左

上角のウィンドウです。ミニバッファにいながら別のウィンドウをスクロールさせる場合は、変数 `minibuffer-scroll-window` で指定します。この変数は、その他のウィンドウが選択されているときには何ら影響しません。 17.8 節「ミニバッファそのほか」p.254 を参照してください。

ミニバッファがアクティブなとき、選択されたウィンドウが右下角のものならミニバッファが次のウィンドウとなります。この場合、 `scroll-other-window` は、ミニバッファをスクロールするよう試みます。もしミニバッファがただ 1 行しか含んでいなければ、スクロールすべきところがないため、エコー領域が “Beginning of buffer” というメッセージをしばらく表示した後、その行が再現します。

other-window-scroll-buffer Variable

この変数が非 `nil` の場合、それは、 `scroll-other-window` にどのバッファをスクロールすべきかを伝えます。

scroll-step User Option

この変数は、ポイントが画面の外に出てしまうとき、自動的にどれだけスクロールするかを制御します。その値が 0 なら、再表示は、ポイントがそのウィンドウの垂直方向の中央にくるようにスクロールします。その値が正の整数 n なら、再表示は、どちらかの方向に n 行スクロールすることによって、もし可能ならポイントを画面上に呼び戻します。だめならポイントを中央にもってきます。デフォルト値は 0 です。

next-screen-context-lines User Option

この変数の値は、画面いっぱいスクロールするときに残す、つなぎの行数です。たとえば、引数が `nil` での `scroll-up` は、そのウィンドウの下端にあるこの行数分の行が、代わりに上端に現われるようにスクロールします。デフォルト値は 2 です。

recenter &optional *count* Command

この関数は、ポイントがある位置のテキストを、そのウィンドウ内の指定された垂直位置に置くよう、選択されたウィンドウをスクロールします。

count が非負の数の場合、ポイントのある行をそのウィンドウの上端から *count* 行下に置きます。 *count* が負の数の場合、 -1 がそのウィンドウで使用可能な最下行を表わすようにして、そのウィンドウの下端から上向きに数えます。 *count* が非 `nil` のリストなら、それはそのウィンドウの真ん中の行を表わします。

count が `nil` の場合、 `recenter` は、ポイントのある行をウィンドウの真ん中に置き、選択されたフレーム全体ををクリアして再表示します。

`recenter` が対話的に呼ばれるとき、 *count* はそのままの前置引数[3]です。そのため、前置キーとしての `C-u` の打鍵は *count* に非 `nil` のリストを設定しますが、 `C-u 4` の打鍵は *count* に 4 を設定し、現在の行を上端から 4 行目に位置させます。

引数が 0 の場合、 `recenter` は、現在の行をそのウィンドウの上端に位置させます。この動作はとても便利なため、一部の人たちはこれを別個のキー・バインドにしています。たとえば、

[3] 訳注: 18.9 節「前置コマンド引数」 p.286 参照。

```
(defun line-to-top-of-window ()
  "Scroll current line to top of window.
Replaces three keystroke sequence C-u 0 C-l."
  (interactive)
  (recenter 0))

(global-set-key [kp-multiply] 'line-to-top-of-window)
```

25.12 水平スクロール

私たちは英語を、基本的には上から下に向かって、そのうえで左から右に向かって読むため、水平スクロールは垂直スクロールとは少し違います。垂直スクロールは表示対象のテキストの途中一部分の選択をするだけですが、水平スクロールは各行の一部を画面の外に追いやってしまいます[4]。したがって水平スクロールの量は、バッファ内の位置ではなく桁数で指定されます。水平スクロールは、 `window-start` によって返される表示開始位置とはまったく無関係です。

通常、水平スクロールは行なわれていません。したがって最も左の桁はウィンドウの左端に位置しています。この状態では、右へのスクロールによって現われるデータが画面の左側にないため、そのスクロールは意味をもたず、よって許されてもいません。左へのスクロールは可能です。これは、テキストの第一桁をウィンドウの端より外へスクロールし、以前に切り詰められていた右側の桁を表示します。ウィンドウがいったん左への非 0 の水平スクロール量を持つと、それを右へ戻すようスクロールすることができますが、正味の水平スクロールを 0 に縮めるまでだけです。左へスクロールできる量に制限はありませんが、やがてはテキスト全体が左端に隠れてしまうでしょう。

scroll-left *count* Command

この関数は、選択されたウィンドウを *count* 桁左へ (あるいは *count* が負なら右へ) スクロールします。戻り値は、その変更後に実際に左側にスクロールされている合計の量、つまり `window-hscroll` (以下参照) によって返される値と同じです。

scroll-right *count* Command

この関数は、選択されたウィンドウを *count* 桁右へ (あるいは *count* が負なら左へ) スクロールします。戻り値は、その変更後に実際に左側にスクロールされている合計の量、つまり `window-hscroll` (以下参照) によって返される値と同じです。

いったん、行けるだけ右へスクロールしてしまうと、左側への合計のスクロール量が 0 となる通常の位置へ戻り、さらに右へのスクロールの試みは何の効果ももたらしません。

window-hscroll &optional *window* Function

この関数は、 *window* の左側への合計の水平スクロール量、つまり *window* 内のテキストが左端を越えて左側にスクロールされている桁数を返します。

その値は、決して負にはなりません。 *window* 内で水平スクロールが行なわれていないときは 0 です (普段の場合)。

[4] 訳注: 水平スクロールでは、逆スクロールしながら読み進む必要があります。

window が nil の場合、選択されたウィンドウが使用されます。

```
(window-hscroll)
     ⇒ 0
(scroll-left 5)
     ⇒ 5
(window-hscroll)
     ⇒ 5
```

set-window-hscroll *window columns* Function

この関数は、 *window* がスクロールされる左端からの桁数として、 *column* の値を設定します。引数 *columns* は 0 以上です。そうでない場合は 0 として取られます。

返される値は *columns* です。

```
(set-window-hscroll (selected-window) 10)
     ⇒ 10
```

以下は、水平スクロールの結果、与えられた位置 *position* が画面の外にあるか否かを判断するための方法です:

```
(defun hscroll-on-screen (window position)
  (save-excursion
    (goto-char position)
    (and
     (>= (- (current-column) (window-hscroll window)) 0)
     (< (- (current-column) (window-hscroll window))
        (window-width window)))))
```

25.13 ウィンドウのサイズ

Emacs のウィンドウは矩形であり、そのサイズ情報は、高さ (行数) と幅 (各行に置かれる文字数) から成ります。モード行は高さに含まれます。ですが幅は、スクロール・バーや、横並びのウィンドウを分離する '|' 文字の桁を数えません。

以下の三つの関数は、ウィンドウに関するサイズ情報を返します:

window-height &optional *window* Function

この関数は、モード行を含む、 *window* 内の行数を返します。 *window* がフレーム全体を占めている場合、これはそのフレーム上での frame-height の値よりも小さいものとなります (最後の行は常にミニバッファ用に予約されているため)。

window が nil の場合、この関数は、選択されたウィンドウを使用します。

```
(window-height)
     ⇒ 23
(split-window-vertically)
     ⇒ #<window 4 on windows.texi>
(window-height)
     ⇒ 11
```

window-width &optional *window* Function

この関数は、 *window* 内の桁数を返します。 *window* がフレーム全体を占めている場合、これはそのフレーム上での `frame-width` の値と等しくなります。その幅は、そのウィンドウのスクロール・バーや、横並びのウィンドウを分離する '|' 文字の桁を含みません。

window が `nil` の場合、この関数は、選択されたウィンドウを使用します。

```
(window-width)
     ⇒ 80
```

window-edges &optional *window* Function

この関数は、 *window* の角の座標のリストを返します。 *window* が `nil` の場合、選択されたウィンドウが使用されます。

そのリストの順序は (*left top right bottom*) で、すべての要素はそのフレームの左上角を 0、 0 とする相対値です。その値の *right* の要素は、 *window* によって使用される最も右の桁よりも大きく、 *bottom* の要素も *window* によって使用される最も下の行やモード行よりも大きくなります。

横並びのウィンドウがあるとき、右側に隣接ウィンドウがある方のウィンドウ (訳注: 左側のウィンドウ) の右の角の値は、そのウィンドウと隣接ウィンドウとのセパレータの幅を含みます。このセパレータは、 '|' 文字の桁、あるいはスクロール・バーとなります。ウィンドウの幅はこのセパレータを含まないため、その幅はこの場合、左右の角の差とは等しくなりません。

以下に、ただ一つのウィンドウをもった、典型的な 24 行端末上で得られる結果を示します:

```
(window-edges (selected-window))
     ⇒ (0 0 80 23)
```

最後の行はエコー領域であるため、下の角は 23 行目にあたります。

window がそのフレームの左上角にある場合、 *bottom* は `(window-height)` の値と等しく、 *right* はほぼ `(window-width)` の値と等しく[5]、 *top* と *left* は 0 となります。たとえば、以下のウィンドウの各角は'`0 0 5 8`'です。フレームが 8 桁以上だと仮定すると、そのウィンドウの端の桁 (7 桁目) は、テキストではなくボーダーを保持します。最後の行 (4 行目) は、以下で '`xxxxxxxxx`' と示されている、モード行を保持します。

```
           0
           _______
        0 |       |
          |       |
          |       |
          |       |
          xxxxxxxxx  4

                  7
```

[5] *right* は、垂直分離線あるいはスクロール・バーを含みますが、`(window-width)` には含まれないため、これらは厳密には等価ではありません。

横並びのウィンドウがあるとき、そのフレームの右端にないウィンドウはみんな、その最後の数桁にセパレータを保持します。セパレータは、そのウィンドウの幅のうちの 1 桁あるいは 2 桁として数えます。セパレータは、左側のウィンドウに属するため、ウィンドウは決してその左端にはセパレータを含みません。

以下の例では、フレームの幅が 7 桁であると仮定しましょう。このとき左のウィンドウの角は'`0 0 4 3`'で、右のウィンドウの角は'`4 0 7 3`'です。

```
 ___ ___
|   |   |
|   |   |
xxxxxxxxx

 0  34  7
```

25.14 ウィンドウのサイズの変更

ウィンドウ・サイズ関数は、ウィンドウのサイズを変更する高水準コマンドと、ウィンドウ・サイズを入手する低水準関数の二種類に分けられます。 Emacs は、ウィンドウのオーバーラップやウィンドウ間のギャップを許していないため、あるウィンドウのリサイズは、他のウィンドウに影響を与えます。

enlarge-window *size* &optional *horizontal* Command

この関数は、選択されたウィンドウを *size* 行高くし、隣接するウィンドウから行を奪います。まず、一つのウィンドウからそのウィンドウが使い果たされるまで行を取り、その後別のウィンドウから取ります。もし、行が奪われる側のウィンドウが `window-min-height` 行未満に縮むと、そのウィンドウは消えてしまいます。

horizontal が非 `nil` の場合、この関数は、行ではなく桁を奪いながら *window* を *size* 桁だけ横に広げます。もし、桁が奪われる側のウィンドウが `window-min-width` 桁未満に縮むと、そのウィンドウは消えてしまいます。

要求されたサイズがそのウィンドウのフレームからはみ出す場合、この関数は、そのウィンドウがフレームの高さ (あるいは幅) 全体を占有するようにします。

size が負の場合、この関数は、ウィンドウを $-size$ 行あるいは $-size$ 桁だけ縮めます。これによってウィンドウが最小サイズ (`window-min-height` と `window-min-width`) よりも小さくなる場合、 `enlarge-window` はそのウィンドウを消去します。

`enlarge-widnow` は `nil` を返します。

enlarge-window-horizontally *columns* Command

この関数は、選択されたウィンドウを *columns* だけ横に広げます。これは以下のように定義されます:

```
(defun enlarge-window-horizontally (columns)
  (enlarge-window columns t))
```

shrink-window *size* &optional *horizontal* Command

この関数は、 `enlarge-window` に似ていますが、他のウィンドウに行 (あるいは桁) を譲ることによって選択されたウィンドウを小さくし、引数 *size* 分を打ち消します。そのウィンドウは、 `window-min-height` か `window-min-width` 未満に縮むと消えてしまいます。

size が負の場合、そのウィンドウは、 $-size$ 行あるいは $-size$ 桁だけ広げられます。

shrink-window-horizontally *columns* Command

この関数は、選択されたウィンドウを *columns* だけ狭くします。これは以下のように定義されます:

```
(defun shrink-window-horizontally (columns)
  (shrink-window columns t))
```

以下の二つの変数は、ウィンドウ・サイズ変更関数 (window-size-changing functions) に、最小の高さと幅を強要します。

window-min-height User Option

この変数の値は、ウィンドウが自動的に消去されるまでにその高さがどのくらい低くなるかを決定します。ウィンドウを `window-min-height` よりも小さくすると、自動的に消去され、これよりも低い高さのウィンドウを生成することはできません。絶対的な最低高は 2 です (モード行のための 1 行と、バッファ表示のための 1 行が許されます)。この値が 2 よりも小さい場合、ウィンドウサイズを変更する動作は、この変数を 2 にリセットします。デフォルト値は 4 です。

window-min-width User Option

この変数の値は、ウィンドウが自動的に消去されるまでにどのくらい狭くなるかを決定します。ウィンドウを `window-min-width` よりも小さくすると、自動的に消去され、これよりも狭いウィンドウを生成することはできません。絶対的な最小幅は 1 です。これ未満の値は無視されます。デフォルト値は 10 です。

window-size-change-functions Variable

この変数は、何らかの理由でいずれかのウィンドウのサイズが変わる際に呼ばれる、関数のリストを保持します。この関数群は再表示のたびに一度だけ呼ばれ、またフレームのサイズ変更が生じた際にもその各フレームに対して一度だけ呼ばれます。

各関数は、単独の引数としてそのフレームを受け取ります。どのウィンドウのサイズが、あるいは厳密にどのように変更されたのかを見つけ出す直接的な方法はありません。しかしながら、あなたのサイズ変更関数 (size-change function) が、あなたが関心があるウィンドウを各変更のたびに追跡すれば、古いサイズ情報を新しいもの比較することによって、何が変更されたのかを割り出すことができます。

ウィンドウの生成や消去はサイズの変更とみなし、したがってこれらの関数の呼出しを引き起こします。フレーム・サイズの変更も既存のウィンドウのサイズを変更するため、同様にみなされます。

これらの関数の内部で、 `save-window-excursion` を使用するのはよくありません。なぜなら、それは常にサイズの変更とみなされ、これらの関数が繰り返し呼び出されてしまうからです。多くの場合、ここで必要とされているのは `save-selected-window` でしょう。

25.15 座標とウィンドウ

この節では、画面の座標とウィンドウが、どのように関連しているのかを述べます。

window-at *x y* &optional *frame* Function

この関数は、フレーム *frame* において、指定されたカーソル位置を含んでいるウィンドウを返します。座標 *x* と *y* は文字数で計られ、そのフレームの左上角から数えられます。それらがもしフレームの範囲外なら、 `window-at` は `nil` を返します。

frame を省略した場合、選択されたフレームが使用されます。

coordinates-in-window-p *coordinates window* Function

この関数は、フレーム内の特定の位置が、ウィンドウ *window* の範囲内か否かを調べます。

引数 *coordinates* は、この形式のコンスセルです:

(*x* . *y*)

座標 *x* と *y* は文字数で計られ、画面あるいはフレームの左上角から数えられます。

その座標が *window* の内側なら、 `coordinates-in-window-p` の値は非 `nil` となります。その値はまた、その位置が、ウィンドウのどの部分にあたるのかを以下のように示します:

(*relx* . *rely*)
: その座標は、 *window* の内側にあります。数字 *relx* と *rely* は、指定された位置をそのウィンドウの左上角にあたる 0 から数えた、ウィンドウ相対座標 (window-relative coordinates) と等価です。

`mode-line`
: その座標は、 *window* のモード行の上にあります。

`vertical-split`
: その座標は、 *window* とその右側に隣接するウィンドウとの間の、垂直線の上にあります。この値は、そのウィンドウがスクロール・バーをもたないときにのみあり得ます。スクロール・バー上の位置は、そのウィンドウの外側と考えられるためです。

`nil`
: その座標は、 *window* のいずれの部分にもあたりません。

関数 `coordinates-in-window-p` は、常に、 *window* があるフレームを使用するため、引数としてはフレームを要求しません。

25.16 ウィンドウ構成

『ウィンドウ構成 (window configuration)』は、フレーム全体の配置 (全ウィンドウ、それらのサイズ、それらがどのバッファを含んでいるか、各バッファの表示されている部分、そしてポイントとマークの値) を記録します。以前に保存されたウィンドウ構成を復元することによって、以前の配置全体を回復させることができます。

ただ一つのフレームだけでなく、すべてのフレームについて記録したい場合は、ウィンドウ構成の代わりにフレーム構成を使用します。26.12 節「フレーム構成」p.449 を参照してください。

current-window-configuration Function

この関数は、 Emacs の現在のウィンドウ構成 (すなわちウィンドウの数、それらのサイズとカレント・バッファ、どのウィンドウが選択されたウィンドウか、各ウィンドウにおいて表示されているバッファ、表示開始位置、そしてポイントとマークの位置) を表現する、新たなオブジェクト返します。カレント・バッファ内のポイントについては例外で、その値は保存されません。

set-window-configuration *configuration* Function

この関数は、 Emacs のウィンドウとバッファの構成を、 *configuration* によって指定された状態に復元します。引数 *configuration* は、以前に `current-window-configuration` によって返された値でなくてはなりません。

この関数は、常にウィンドウ・サイズの変更とみなされ、 `window-size-change-functions` の実行を引き起こします (この関数は、新しいウィンドウ構成が、実際に古いものと異なるのか否かを見分ける手段を備えていません)。

`save-window-excursion` と同じ効果を得る、この関数の利用方法を以下に示します:

```
(let ((config (current-window-configuration)))
  (unwind-protect
      (progn (split-window-vertically nil)
             ...)
    (set-window-configuration config)))
```

save-window-excursion *forms...* Special Form

この特殊形式は、ウィンドウ構成を記録して *forms* を次々に実行した後、はじめのウィンドウ構成を復元します。そのウィンドウ構成には、ポイント値とバッファの可視の部分が含まれます。また、選択されたウィンドウも含まれます。しかしながら、カレント・バッファ内のポイント値は含まれません。これを保護したい場合は、 `save-excursion` を使用します。

`save-selected-window` で十分な場合は、この構文要素を使用しないでください。

`save-window-excursion` からの脱出は、常に `window-size-change-functions` の実行を引き起こします (この特殊形式は、復元されるウィンドウ構成が、実際に *forms* 終了時のものと異なるのか否かを見分ける手段を備えていません)。

戻り値は、 *forms* 内の最後の形式の値です。

```
(split-window)
     ⇒ #<window 25 on control.texi>
(setq w (selected-window))
     ⇒ #<window 19 on control.texi>
(save-window-excursion
  (delete-other-windows w)
  (switch-to-buffer "foo")
  'do-something)
     ⇒ do-something
     ;; ここで再び画面が分割されます。
```

window-configuration-p *object* Function

この関数は、 *object* がウィンドウ構成であれば `t` を返します。

ウィンドウ構成の内部を見るためのプリミティブは、理にかなってはいますが、実装はされていません。それらが、実装に値するほど十分便利かどうかが明らかではないのです。

26 フレーム

『フレーム (frame)』は一つ以上の Emacs のウィンドウを含む画面上の長方形状のものです。フレームには最初は一つのウィンドウ (と多分ミニバッファ) しかありませんが、縦方向にも横方向にも分割することができます。

Emacs がテキスト専用端末で起動されたときは、一つの『端末フレーム (terminal frame)』で立ち上がります。追加のフレームを作成したときは、 Emacs は同時にはただ一つのフレームだけを、当たり前ですが、端末の画面に表示します。

Emacs が X サーバーと直接やりとりするときは、端末フレームの代わりに一つの『X ウインドウ・フレーム (X Window frame)』で立ち上がります。この場合は同時に複数の X ウインドウ・フレームをそれ自身の X ウインドウに表示できます。

framep *object* Function

この述語は *object* がフレームのときに `t` を、それ以外のときは `nil` を返します。

関連する情報は第 35 章「Emacs の表示」 p.601 を参照してください。

26.1 フレームの作成

新しいフレームを作るには、 `make-frame` 関数を呼び出します。

make-frame &optional *alist* Function

この関数は新しいフレームを作ります。 X を使っていれば X ウインドウフレームを、そうでなければ端末フレームを作ります。

引数はフレームのパラメータを指定する alist です。 *alist* によって指定されないパラメータは `default-frame-alist` 変数の値が用いられます。これらで指定されていないパラメータは、標準の X defaults ファイルか X リソースがデフォルトになります。

どのパラメータが指定できるかは、原則として Emacs がどのようなウインドウ・システムにフレームを表示しているかによります。指定可能な個々のパラメータについての説明は、 26.3.3 節「X ウインドウ・フレーム・パラメータ」 p.438 を参照してください。

before-make-frame-hook Variable

`make-frame` によって実際にフレームが作られる前に起動される正規フックです。

after-make-frame-hook Variable

`make-frame` によってフレームが作られたあとに起動される正規フックです。

26.2 複数のディスプレイ

一つの Emacs は一つ以上の X ウインドウ・ディスプレイとやりとりすることができます。起動時には、 Emacs は一つのディスプレイだけを使います。そのディスプレイは、 `DISPLAY` 環

境変数や '--display' オプションで指定されます (*The GNU Emacs Manual* の "Initial Options" 節参照)。別のディスプレイに接続するにはコマンド `make-frame-on-display` を使うか、フレームを作成するときに `display` フレーム・パラメータを指定してください。

Emacs はそれぞれの X サーバーを独立した端末として扱います。それぞれに選択されたフレームとミニバッファがあります。いくつかの Lisp 変数は現在の端末 (すなわち現在の選択されたフレームに対応する端末) にローカルな値を持っています。 `default-minibuffer-frame`、 `defining-kbd-macro`、 `last-kbd-macro`、 `multiple-frames`、 `system-key-alist` がそうです。これらの変数は常に端末ローカルでバッファローカルにはなり得ません。

一つの X サーバーは一つ以上の画面を扱うことができます。ディスプレイの名前 '*host*.*server*.*screen*' は三つの部分からなります。最後の部分は与えられたサーバーの画面番号を指定しています。一つのサーバーに属する二つの画面を使うときは、 Emacs はそれらの名前が似通っていることによりそれらが一つのキーボードを共有していることを知り、それらを一つの端末として扱います。

make-frame-on-display *display* &optional *parameters* — Command

これはディスプレイ *display* 上に新しいフレームを作ります。ほかのパラメータは *parameters* から得ます。 *display* 引数を別にすれば、 `make-frame` と同様です (26.1 節「フレームの作成」p.435 参照)。

x-display-list — Function

これは Emacs が接続している X ディスプレイを示すリストを返します。リストの要素は文字列で、それぞれがディスプレイの名前です。

x-open-connection *display* &optional *xrm-string* — Function

この関数は X ディスプレイ *display* に接続します。そのディスプレイ上にフレームを作ることはしませんが、そのディスプレイとの通信が確立されているかチェックすることはできます。

オプション引数 *resource-string* が `nil` でなければ、それは '.Xresources' ファイルと同じフォーマットのリソース名と値の文字列です。指定した値は X サーバー自身に記録されている値に優先します。これらはこのディスプレイ上に作られる全ての Emacs のフレームに適用されます。これはこの文字列の例です:

```
"*BorderWidth: 3\n*InternalBorder: 2\n"
```

26.20 節「X のリソース」p.453 を参照してください。

x-close-connection *display* — Function

この関数はディスプレイ *display* との接続を閉じます。これを実行する前に、まずそのディスプレイ上に開かれている全てのフレームを削除しなくてはいけません (26.5 節「フレームの消去」p.443 参照)。

26.3 フレーム・パラメータ

フレームには、見栄えや振舞いを制御するためのたくさんのパラメータがあります。フレームがどんなパラメータを持っているかは、どんな表示機構を使っているかによって決まります。

フレーム・パラメータはウインドウシステムのためにあります。端末フレームもいくつかパラメータを持っていますがほとんどが互換性のためのもので、高さ、幅、 `buffer-predicate` パラメータだけが本当に意味があります。

26.3.1 フレーム・パラメータへのアクセス

これらの関数はフレームのパラメータの値を得たり、変更するものです。

frame-parameters *frame* Function

関数 `frame-parameters` は *frame* のすべてのパラメータとその値からなる alist を返します。

modify-frame-parameters *frame alist* Function

この関数はフレーム *frame* のパラメータを *alist* の要素に基づいて変更します。 *alist* のそれぞれの要素は (*parm* . *value*) という形式です。 *parm* はパラメータのシンボル名です。 *alist* に存在しないパラメータの値は変更されません。

26.3.2 初期フレーム・パラメータ

'`.emacs`' ファイルで `initial-frame-alist` をセットすることで、初期起動フレームのパラメータを指定することができます。

initial-frame-alist Variable

この変数の値はパラメータの alist で、その値は初期 X ウインドウ・フレームが作られるときに使われます。それぞれの要素の形式は次のようになっています:

(*parameter* . *value*)

Emacs は初期フレームをあなたの '`~/.emacs`' ファイルを読む前に作ります。そのファイルを読んだあとで Emacs は `initial-frame-alist` をチェックし、すでに作った初期フレームの値と違う値のパラメータを適用します。

これらの設定にフレームの位置や大きさの指定が含まれている場合、フレームは間違った大きさや間違った位置に現れ、次に指定した大きさや位置に変更されることになります。これが気になるなら、 X リソースで同じように位置や大きさを指定することができます。これはフレームが作られる前に適用されます。 *The GNU Emacs Manual* の "X Resources" 節を参照してください。

一般的に X リソースは全てのフレームに適用されます。ある X リソースを初期フレームだけに指定して、その後のフレームには適用したくない場合は、つぎのようにしてください。その後のフレームではその X リソースを無効にするために `default-frame-alist` でパラメータを指定し、次にそれらが初期フレームに影響しないように同じパラメータを `initial-frame-alist` で X リソースに一致する値に指定します。

これらのパラメータが独立したミニバッファ専用フレームを (`minibuffer . nil`) で指定していて、それがまだ作られていない場合には、 Emacs はそれを作ります。

minibuffer-frame-alist Variable

この変数の値はパラメータの alist で、その値は初期ミニバッファ専用フレームが作られるときに使われます。ミニバッファ専用フレームは main 初期フレームのパラメータによって作られるかどうか決まります。

default-frame-alist Variable

これは初期フレームの後の Emacs フレームのフレーム・パラメータのデフォルト値を指定する alist です。

25.8 節「表示に用いるウィンドウの選択」p.418 の `special-display-frame-alist` も見てください。

Emacs を起動するときにウインドウの見栄えを指定するオプションを使うと、 `default-frame-alist` に要素を追加することにもなります。ただし、 ‘`-geometry`’ オプションだけは、その指定する位置が `initial-frame-alist` に追加されます。 *The GNU Emacs Manual* の “Command Arguments” 節を参照してください。

26.3.3 X ウインドウ・フレーム・パラメータ

フレームがどんなパラメータを持っているかは、どんな表示機構を使っているかによって決まります。以下は X ウインドウ・フレームのパラメータの表です。この中で `name`、 `height`、 `width`、 `buffer-predicate` は非 X フレームでも有効です。

`name` フレームの名前です。ほとんどのウインドウ・マネージャはフレームの名前をそのフレームの上の縁に表示します。名前を指定しない場合やフレームが複数ある場合には、フレーム中の選択されたウインドウに表示されているバッファを元にした名前がつきます。

フレームを作るときに明示的にフレームの名前を指定した場合は、そのフレームの X リソースを探す時にもその名前が (Emacs の実行形式の名前のかわりに) 用いられます。

`display` このフレームが開かれているディスプレイです。これは `DISPLAY` 環境変数のような形式 "*host*:*dpy*.*screen*"の文字列でなくてはなりません。

`left` ピクセルで表した画面の左端から見たウインドウの左端の画面上の位置です。値は正の数 *pos* か、もしくは負の *pos* 値も指定できる形式 (`+` *pos*) のリストのどちらでも構いません。

負の数 $-pos$ や形式 (`-` *pos*) のリストは、実は画面の右端から見たウインドウの右端の位置を指定します。 *pos* の正の値は左に向かって数えます。パラメータが負の整数 $-pos$ なら *pos* は正になります！

`top` ピクセルで表した画面の上端から見たウインドウの上端の画面上の位置です。値は正の数 *pos* か、もしくは負の *pos* 値も指定できる形式 (`+` *pos*) のリストのどちらでも構いません。

負の数 $-pos$ や形式 (`-` *pos*) のリストは、実は画面の下端から見たウインドウの下端の位置を指定します。 *pos* の正の値は上に向かって数えます。パラメータが負の整数 $-pos$ なら *pos* は正になります！

`icon-left` 画面の左端からピクセルで数えたフレームのアイコンの左端の画面上の位置です。フレームがアイコン化されたときに効果をあらわします。

`icon-top` 画面の上端からピクセルで数えたフレームのアイコンの上端の画面上の位置です。フレームがアイコン化されたときに効果をあらわします。

`user-position`

フレームの画面上の位置がユーザに明示的に (たとえば ‘`geometry`’ オプションなどで) 指定されたときに非 `nil` になります。自動的に非 `nil` になることはありません。 `left` と `top` パラメータに加えてこのパラメータを指定するのは `make-frame` を呼び出す Lisp プログラムの責任です。[1]

`height` 文字数で表したフレーム内部の高さです。 (高さをピクセル単位で得るには `frame-pixel-height` を呼び出してください。 26.3.4 節「フレームのサイズと位置」p.441 参照。)

`width` 文字数で表したフレーム内部の幅です。 (幅をピクセル単位で得るには `frame-pixel-width` を呼び出してください。 26.3.4 節「フレームのサイズと位置」p.441 参照。)

`window-id` フレームの X ウインドウの ID 番号です。

`minibuffer`

このフレームがそれ自身のミニバッファをもつかどうかを示します。値が `t` のときはミニバッファを持ち、 `nil` のときは持ちません。 `only` のときはこのフレームはミニバッファそのものになります。値が (別のフレームにある) ミニバッファ・ウインドウのときには、新しいフレームはそのミニバッファを使います。

`buffer-predicate`

このフレームのバッファ述語 (buffer-predicate) 関数です。これが `nil` でなければ、関数 `other-buffer` はどのバッファを候補にするかを決めるのに (選択されたフレームの) この述語を使います。この述語はそれぞれのバッファごとにそれを引数として呼び出されます。この述語が非 `nil` を返すとそのバッファは候補になります。

---

[1] 訳注: このパラメータは `top` や `left` の値を `USPosition` か `PPosition` のどちらとしてウインドウ・マネージャに渡すかを制御します。 `user-position` を指定すれば `USPosition` となり、どんな場合でも指定した座標は反映されますが、 `PPosition` の場合はウインドウ・マネージャの設定によっては反映されません。

`font` フレームの中でテキストを表示するフォントの名前です。これは文字列です。

`auto-raise`
: フレームが選択されたときに上昇 (raise) するかどうかを示します (非 `nil` のとき上昇します)。

`auto-lower`
: フレームが選択されなくなったときに下降 (lower) するかどうかを示します (非 `nil` のとき下降します)。

`vertical-scroll-bars`
: フレームに縦方向のスクロール・バーを表示するかどうかを示します (非 `nil` のとき表示します)。

`horizontal-scroll-bars`
: フレームに横方向のスクロール・バーを表示するかどうかを示します (非 `nil` のとき表示します)。(横方向のスクロール・バーはまだ実装されていません。)

`scroll-bar-width`
: ピクセルで表した縦方向のスクロール・バーの幅です。

`icon-type` このフレームがアイコン化されたときに表示するアイコンのタイプを示します。値が文字列なら、アイコンに使用するビットマップが入っているファイルを指定します。それ以外の全ての非 `nil` 値はデフォルトのビットマップ・アイコン (ウシカモシカ (gnu) の絵) の使用を指定します。 `nil` はテキスト・アイコンを指定します。

`icon-name` このフレームがアイコン化されたときに使う名前です。これが `nil` ならフレームのタイトルを使います。

`foreground-color`
: 文字のイメージに使われる色を示します。これは X サーバーで定義されている有効な色の名前を表す文字列です。

`background-color`
: 文字の背景に使われる色を示します。

`mouse-color`
: マウス・ポインタの色を示します。

`cursor-color`
: ポイントを表すカーソルの色を示します。

`border-color`
: フレームの境界の色を示します。

`cursor-type`
: カーソルを表示する方法を示します。認識される値は `bar` と `box` と `(bar . *width*)` です。シンボル `box` はよくあるやり方で、ポイントの後の文字に黒い箱を重ね書きします。これがデフォルトです。シンボル `bar` はカーソルとして文

字の間に縦棒を置きます。(`bar . `*width*) は *width* ピクセル幅の棒を指定します。

`border-width`
ウインドウの境界の幅をピクセルで示します。

`internal-border-width`
テキストと境界の間隔をピクセルで示します。

`unsplittable`
非 `nil` の場合、このフレームのウインドウは自動的に分割されません。

`visibility`
フレームの可視性の状態を示します。三つの値をとり得ます。 `nil` は不可視、 `t` は可視、 `icon` はアイコン化されています。 26.10 節「フレームの可視性」 p.447 を参照してください。

`menu-bar-lines`
フレームの上部のメニューバーのために割り当てる行数を示します。デフォルトは 1 です。 19.12.5 節「メニュー・バー」 p.314 を参照してください。 (X toolkit を使うバージョンの Emacs では、メニュー・バーは 1 行だけです。指定された数は 0 より大きいかどうかだけが問題になります。)

`parent-id` このフレームの親になるべきウインドウの X ウインドウにおける ID 番号です。これを指定すると、ある別のアプリケーションのウインドウの中で Emacs のウインドウを開くことができます。(これが実装されているかどうかははっきりしません。実際に試してちゃんと動くかどうか確かめてください。)

26.3.4 フレームのサイズと位置

フレーム・パラメータ `left`、 `top`、 `height`、 `width` を使うとフレームのサイズや位置を変更したり読み出したりすることができます。指定されない位置やサイズのパラメータは、ウインドウ・マネージャがそれぞれの流儀で決めてくれます。

サイズや位置を扱うための特別な機能がいくつかあります:

set-frame-position *frame left top* Function

この関数は、 *frame* の左上の角の位置を *left* と *top* にセットします。これらの引数はピクセル単位で画面の左上の角から数えます。負のパラメータ値は画面の左上の角から上または右に向かって数えます。

frame-height &optional *frame* Function
frame-width &optional *frame* Function

これらの関数は、 *frame* の高さと幅を文字数単位で返します。 *frame* を省略した場合は選択されているフレームを使用します。

frame-pixel-height &optional *frame* Function
frame-pixel-width &optional *frame* Function

これらの関数は *frame* の高さと幅をピクセル単位で返します。 *frame* を省略した場合は選択されているフレームを使用します。

frame-char-height &optional *frame* Function
frame-char-width &optional *frame* Function

これらの関数は *frame* の中の文字の高さと幅をピクセル単位で返します。この値は選択したフォントに依存します。 *frame* を省略した場合は選択されているフレームを使用します。

set-frame-size *frame cols rows* Function

この関数は *frame* のサイズを、 *cols* と *rows* で指定される文字数単位の幅と高さにセットします。

ピクセル単位でサイズをセットするには、値を `frame-char-height` と `frame-char-width` を使って文字数単位に変換してください。

マルチ・フレームをサポートしていないバージョンの Emacs で画面の高さと幅を指定するのに使われた、 `set-screen-height` と `set-screen-width` は旧式の関数ですがまだ使えます。これらの関数は選択されたフレームに対して働きます。 35.2 節「画面サイズ」 p.601 を参照してください。

x-parse-geometry *geom* Function

`x-parse-geometry` 関数は標準的な X ウインドウの geometry 文字列を `make-frame` の引数の一部として使うことのできる alist に変換します。

alist は *geom* で指定されるパラメータとその値を表現します。それぞれの要素は (*parameter* . *value*) というような形です。 *parameter* として許されるものは `left`、 `top`、 `width`、 `height` です。

サイズのパラメータの値は整数でなくてはなりません。位置のパラメータ名 `left` と `top` は、ある値のときには右または下の端の位置を示すことがあるので、完全には的確な名前ではありません。以下は *value* が位置のパラメータとしてとり得る値です:

整数
: 正の整数は画面の左または上の端と、ウインドウの左端または上端の距離です。負の整数は画面の右または下の端とウインドウの右または下の端の距離です。

`(+` *position*`)`
: これは画面の左または上の端からのウインドウの左または上の端の位置を指定します。整数 *position* は正でも負でも構いません。負の値は画面の外の位置を指定することになります。

`(-` *position*`)`
: これは画面の右または下の端からのウインドウの右または下の端の位置を指定します。整数 *position* は正でも負でも構いません。負の値は画面の外の位置を指定することになります。

例:

```
(x-parse-geometry "35x70+0-0")
     ⇒ ((width . 35) (height . 70)
         (left . 0) (top - 0))
```

26.4 フレーム・タイトル

全てのフレームはタイトルを持ち、ほとんどのウインドウ・マネージャはフレームの上端にフレーム・タイトルを表示します。 name フレーム属性を使って明示的にタイトルを指定することができます。ですが、普通はこれを明示的に指定せず、 Emacs が自動的に作成するタイトルを使います。

Emacs はフレーム・タイトルを変数 frame-title-format にはいっているテンプレートを元にして作成します。

frame-title-format Variable

この変数は明示的にタイトルが指定されなかったときに、フレームのタイトルを作るやり方を指定します。

この変数の値は実際には mode-line-format のようなモード行構成要素です。 20.3.1 節「モード行の表示をコントロールするデータ構造」p.330 を参照してください。

icon-title-format Variable

この変数は明示的にフレームタイトルが指定されなかったときに、アイコン化されたフレームのタイトルを作るやり方を指定します。このタイトルはアイコン自身に表示されます。

multiple-frames Variable

この変数は Emacs が自動的にセットします。二つ以上のフレーム (ミニバッファ専用フレームや不可視フレームは数えません) があるときに、これの値は t になります。 frame-title-format のデフォルト値では、 multiple-frames を使って二つ以上のフレームがあるときにだけバッファの名前をフレーム・タイトルに入れるようにしています。

この変数は常に現在の端末にローカルでバッファローカルにはできません。 26.2 節「複数のディスプレイ」p.435 を参照してください。

26.5 フレームの消去

フレームは明示的に『消去 (delete)』するまでは可視になることができる状態でいつづけます。消去されたフレームは画面上には出てきませんが、 Lisp オブジェクトとしては全ての参照がなくなるまで存在し続けます。消去されたフレームを元に戻す方法は、保存してあったフレームの構成 (26.12 節「フレーム構成」 p.449 参照) を復元するのを除けば、存在しません。これはウインドウの場合と同じです。

delete-frame &optional *frame* Command

この関数はフレーム *frame* を消去します。 *frame* のデフォルトは選択されているフレームです。

frame-live-p *frame* Function

関数 `frame-live-p` はフレーム *frame* が消去されていなければ非 `nil` を返します。

あるウインドウ・マネージャにはウインドウを消去するコマンドがあります。これは特別なメッセージをウインドウを操作しているプログラムに送ることによって実現されています。Emacs がこれらのコマンドのうちの一つを受けとると、 `delete-frame` イベントが発生します。このイベントの通常の定義は関数 `delete-frame` を呼び出すコマンドです。 18.5.10 節「そのほかのウィンドウ・システム・イベント」 p.274 を参照してください。

26.6 全てのフレームの探索

frame-list Function

関数 `frame-list` は消去されていない全てのフレームのリストを返します。これはバッファの場合の `buffer-list` と似ています。返されるリストは新しく作られたものなので、リストを変更しても Emacs の内部には何の効果もありません。

visible-frame-list Function

この関数は現在可視のフレームのリストを返します。 26.10 節「フレームの可視性」 p.447 を参照してください。 (端末フレームは選択された一つしか実際に表示されませんが、常に "可視" とみなされます。)

next-frame &optional *frame minibuf* Function

関数 `next-frame` は、任意のフレームから始められる全てのフレームを含む便利な環を提供します。その環の中で *frame* の "次" のフレームを返します。 *frame* が省略されたときや `nil` のときのデフォルトは選択されたフレームです。

2 番目の引数 *minibuf* はどんなフレームに注目するかを指定します:

`nil` ミニバッファ専用フレームを除外する。

`visible` 全ての可視フレームに注目する。

0 全ての可視またはアイコン化されたフレームに注目する。

ウインドウ 指定したウインドウをミニバッファとして使っているフレームだけに注目する。

その他 全てのウインドウに注目する。

previous-frame &optional *frame minibuf* Function

`next-frame` と同様ですが、環のまわる方向が逆です。

25.5 節「ウィンドウの循環順序」 p.413 の `next-window` と `previous-window` も参照してください。

26.7 フレームとウインドウ

それぞれのウインドウはただ一つのフレームに属していて、 `window-frame` を使うとそのフレームを得ることができます。

window-frame *window* Function

この関数は *window* が属しているフレームを返します。

一つのフレームに含まれる、ミニバッファ以外の全てのウインドウには循環する順番がついています。順番は上左端のウインドウが先頭で、そこから下または右へ向かって下右端のウインドウ (フレームがミニバッファを含んでいる場合は常にミニバッファ・ウインドウ) まで進み、先頭に戻ります。

frame-top-window *frame* Function

フレーム *frame* の上左端のウインドウを返します。

常に全てのフレームにはただ一つの『そのフレーム内の選択された (selected within the frame)』ウインドウがあります。つまり、フレームを選択することはすなわちこのウインドウを選択することにもなります。フレームの現在選択されているウインドウは `frame-selected-window` で得ることができます。

frame-selected-window *frame* Function

この関数は *frame* 内の選択されたウインドウを返します。

逆にいうと、 Emacs では `select-window` を使ってウインドウを選択することは、そのウインドウが存在するフレーム内の選択されたウインドウをそのウインドウにします。 25.4 節「ウィンドウの選択」 p.412 を参照してください。

(普通は) フレームの中のウインドウの一つを返すもう一つの関数に `minibuffer-window` があります。 17.8 節「ミニバッファそのほか」 p.254 を参照してください。

26.8 ミニバッファとフレーム

通常はそれぞれのフレームはその最下部にそれ自身のミニバッファ・ウインドウを持っていて、あるフレームが選択されたときはそのフレームのミニバッファ・ウインドウが使われます。フレームがミニバッファを持っていれば `minibuffer-window` で得ることができます (17.8 節「ミニバッファそのほか」 p.254 参照)。

ですが、ミニバッファを持たないフレームも作ることができます。そのようなフレームはどれかほかのフレームのミニバッファ・ウインドウを使わなければなりません。フレームを作るときにどの (ある別のフレームの) ミニバッファ・ウインドウを使うか指定することもできます。指定しなかった場合は、変数 `default-minibuffer-frame` が指すフレームのミニバッファを使うことになります。この変数の値はミニバッファを持っているフレームでなければなりません。

ミニバッファ専用フレームを使っていると、ミニバッファに入力するときに自動的に上昇して欲しいかもしれません。そういうときは変数 `minibuffer-auto-raise` を `t` にセットしましょう。 26.11 節「フレームの上昇 / 下降」 p.448 を参照してください。

default-minibuffer-frame Variable

この変数はミニバッファ・ウインドウとして使うフレームのデフォルトを指定します。これは常に現在の端末にローカルでバッファローカルにはできません。26.2 節「複数のディスプレイ」p.435 を参照してください。

26.9 インプット・フォーカス

どんなときでも、一つのフレームが『選択されたフレーム (selected frame)』になっています。選択されたウインドウはいつも選択されたフレームの中にあります。

selected-frame Function

この関数は選択されたフレームを返します。

通常 X サーバは、キーボードからの入力をマウスがいる X のウインドウに送ります。いくつかのウインドウ・マネージャはこういう X サーバの通常の動作を無効にして、マウスのクリックやキーボードのイベントを使って個々の X のウインドウへの『フォーカスの移動 (shift the focus)』を行ないます。

`select-frame` を呼ぶことによって Lisp プログラムでフレームを “一時的に” 切替えることができます。これはウインドウ・マネージャを無効にするのではなく、むしろウインドウ・マネージャがどうにかしてでしゃばってくるまでの間、ウインドウ・マネージャの制御から逃れるということです。

テキスト専用端末を使っているときはウインドウ・マネージャはいません。したがって `switch-frame` がフレームを切替える唯一の方法で、次に `switch-frame` が呼ばれるまで有効ということになります。選択された端末フレームだけが実際に端末に表示されます。初期の端末フレーム以外の端末フレームはそれぞれ番号を持っていて、選択されたフレームの番号はモード行の ‘`Emacs`’ の後に表示されます (20.3.2 節「モード行の中で使用されている変数」p.331 参照)。

select-frame *frame* Function

この関数は、一時的に X サーバのフォーカスがもしあれば無視してフレーム *frame* を選択します。 *frame* を選択している状態は、ユーザが別のフレームを選択する何らかの動作を行なうまで、または次にこの関数が呼び出されるまで続きます。

Emacs は X サーバーやウインドウ・マネージャと協力して、サーバーとウインドウ・マネージャの要求にしたがってどのフレームを選択するか調整します。このとき『フォーカス (focus)』イベントと呼ばれる特殊な入力イベントが発生します。コマンド・ループは `handle-switch-frame` を呼び出してフォーカス・イベントを処理します。 18.5.9 節「フォーカス・イベント」p.273 を参照してください。

handle-switch-frame *frame* Command

この関数はフォーカス・イベントを受け取ってフレーム *frame* を選択します。

フォーカス・イベントは通常このコマンドを起動することによって自分の仕事を行ないます。ほかにどんな理由があってもこれを呼び出してはいけません。

redirect-frame-focus *frame focus-frame* Function

この関数は *frame* から *focus-frame* へフォーカスをリダイレクトします。つまり *focus-frame* はそれ以降の *frame* へ送られるはずのキーストロークを受け取ります。このようなイベントの後には、 `last-event-frame` は *focus-frame* になります。また、 *frame* を指定するフレーム切替えイベントはかわりに *focus-frame* を選択することになります。

focus-frame が `nil` のときは、 *frame* に関する全てのリダイレクトをキャンセルします。したがって、また自分に向けられたイベントを受け取ることになります。

フォーカスのリダイレクトの使い道の一つとしてミニバッファを持たないフレームがあります。このようなフレームは別のフレームのミニバッファを使います。別のフレームのミニバッファがアクティブになるとフォーカスはそのフレームにリダイレクトされます。マウスがミニバッファをアクティブにしたフレームに残っていたとしても、フォーカスはミニバッファが所属しているフレームに移ります。

あるフレームを選択することはフォーカスのリダイレクトを変えることでもあり得ます。フレーム `foo` が選択されているときにフレーム `bar` を選択すると、 `foo` を指していた全てのリダイレクトがかわりに `bar` を指すことになります。この動作は、あるフレームから別のフレームへ `select-window` を使って切替えたときに、フォーカスのリダイレクトがきちんと働けるようしています。

つまり、自分自身にフォーカスをリダイレクトしているフレームはフォーカスのリダイレクトをしていないフレームと違う扱いを受けるということです。 `select-frame` は前者には働くが後者には働かないのです。

リダイレクトは `redirect-frame-focus` が呼ばれてリダイレクトが変更されるまで、有効です。

26.10 フレームの可視性

X ウインドウ・フレームは、『可視 (visible)』、『不可視 (invisible)』、『アイコン化 (iconified)』のいずれかの状態にあります。もし可視なら、その中身を見ることができます。もしアイコン化されていたら、フレームの中身は画面には現れていませんが、アイコンが見えます。もし不可視なら、そのフレームは画面上には見えていませんし、アイコンも同様です。

端末フレームにおいては、どんな場合でも選択されたものしか実際には表示されないので可視性は無意味です。

make-frame-visible &optional *frame* Command

この関数は *frame* を可視にします。 *frame* を省略すると選択されているフレームを可視にします。

make-frame-invisible &optional *frame* Command

この関数は *frame* を不可視にします。 *frame* を省略すると選択されているフレームを不可視にします。

iconify-frame &optional *frame* Command

この関数は *frame* をアイコン化します。 *frame* を省略すると選択されているフレームをアイコン化します。

frame-visible-p *frame* Function

フレーム *frame* の可視性の状態を返します。 *frame* が可視なら `t`、不可視なら `nil`、アイコン化されていれば `icon` を返します。

可視性の状態はフレーム・パラメータにもありますので、それを読み出したり変更したりすることもできます。 26.3.3 節「X ウインドウ・フレーム・パラメータ」p.438 を参照してください。

ユーザはウインドウ・マネージャを使ってフレームをアイコン化したりアイコン化を解除することができます。これは Emacs が制御できるレベルよりも下位のレベルで行なわれますが、このような変更をたどるのにイベントを使うことができます。

26.11 フレームの上昇 / 下降

X ウインドウ・システムは机の暗喩を用いています。この暗喩の中に、ウインドウが、画面と垂直の方向の想像上の 3 番目の次元の中に積み重なっていて、“最高” から “最低” まで順序づけられているというアイデアがあります。二つのウインドウが重なっている場所は、高い方が低い方を隠してしまいます。もっとも低いウインドウでも、ほかのウインドウがその上に重なっていなければ見ることができます。

このウインドウの順番は固定ではありません。むしろ、この順番は頻繁に変更されることが多いでしょう。ウインドウを『上昇 (raising)』させるというのは、ウインドウを重なりの 1 番上に “持ちあげる” ことを意味します。ウインドウを『下降 (lowering)』させるというのは、ウインドウを重なりの 1 番下へ移動します。この移動は想像上の 3 番目の次元の中だけで、ウインドウの画面上の位置は変わりません。

以下の関数を使って Emacs の X のウインドウを上昇、下降させることができます:

raise-frame *frame* Command

この関数はフレーム *frame* を上昇させます。

lower-frame *frame* Command

この関数はフレーム *frame* を下降させます。

minibuffer-auto-raise User Option

非 `nil` の場合、ミニバッファがアクティブになったときにミニバッファ・ウインドウが属するフレームが上昇します。

フレーム・パラメータを使ってフレームごとに、自動上昇 (フレームが選択されたときに自動的に上昇する) や自動下降 (フレームが選択されなくなったときに自動的に下降する) するようにもできます。 26.3.3 節「X ウインドウ・フレーム・パラメータ」p.438 を参照してください。

26.12 フレーム構成

『フレーム構成 (frame configuration)』とは、現在のフレーム群の配置、全ての属性、それぞれのウインドウの構成を記録することです。

current-frame-configuration Function

この関数は現在のフレーム群の配置やそれらの中身を記述したフレーム構成リストを返します。

set-frame-configuration *configuration* Function

この関数はフレームの状態を *configuration* に記述されている状態に戻します。

26.13 マウスの追跡

マウスを『追跡 (track)』することは、すなわちマウスがどこにあるのかを指し示すために何かを表示したり、それをマウスの動きに応じて動かすことは、しばしば役にたちます。効率よくマウスを追跡にするためには、マウスが実際に動くまで待つ方法が必要です。

マウスを追跡する便利な方法は、マウスの移動を表すイベントを得ることです。そうすれば、イベントを待つことによってマウスの移動を待つことができます。おまけに、おそらく起きるであろうほかの種類のイベントを容易に扱うことができます。普通、永遠にマウスを追跡したりはせず、ボタンを離すとかいったあるほかのイベントがおこるまでしか追跡しないので、これは役にたちます。

track-mouse *body...* Special Form

マウス移動の入力イベントが起きている間、 *body* を実行します。 *body* の中では、`read-event` や `read-key-sequence` を使ってそれらのイベントを読みとることができます。マウス移動イベントのフォーマットについては、 18.5.8 節「移動イベント」 p.273 を参照してください。

`track-mouse` の返り値は、 *body* の最後の形式の値になります。

マウスの追跡の通常の用途は、現在位置でボタンを押したり離したりしたことの結果を画面に表示することです。

多くの場合、 `mouse-face` テキスト属性を使うことによってマウスを追跡しなくてもすませることができます (29.18.4 節「特殊な意味をもつ属性」 p.513 参照)。これは非常に低レベルで働きますので、 Lisp レベルのマウスの追跡よりもスムーズに動作します。

26.14 マウスの位置

関数 `mouse-position` と `set-mouse-position` はマウスの現在位置へアクセスする手段を提供します。

mouse-position Function

この関数はマウスの位置を返します。返り値は (*frame* x . y) という感じで、 *x* と *y* は *frame* の上左端からの文字数単位の位置を表す整数です。

set-mouse-position *frame x y* Function

この関数は *frame* の *x*、 *y* の位置へ『マウスをワープ (warps the mouse)』させます。引数の *x* と *y* は *frame* の上左端からの文字数単位の位置を表す整数です。

mouse-pixel-position Function

この関数は、返り値の座標が文字数単位ではなくピクセル単位であることを除けば `mouse-position` と同じです。

set-mouse-pixel-position *frame x y* Function

この関数は、 *x* と *y* が文字数単位ではなくピクセル単位であることを除けば `set-mouse-position` と同じようにマウスをワープさせます。この座標はフレームの内部でなくても構いません。

26.15 ポップアップ・メニュー

X ウインドウを使っているときは、 Lisp プログラムはユーザがマウスで選択できるメニューをポップアップすることができます。

x-popup-menu *position menu* Function

この関数はポップアップ・メニューを表示して、ユーザが何を選択したかを返します。

引数 *position* は画面のどこにメニューを置くかを指定します。この引数はマウス・ボタン・イベント (ボタンを押した場所にメニューを置くことになります) か次の形式のどちらでも構いません:

((*xoffset* *yoffset*) *window*)

xoffset と *yoffset* は *window* のフレームの左上端からのピクセル単位の座標です。

position が `t` の場合は現在のマウスの位置を使います。 *position* が `nil` のときは、実際にはメニューを表示せずに *menu* の中のキーマップで指定されたものの等価なキー・バインドをあらかじめ調べておきます。

引数の *menu* はメニューに何を表示するかを指示します。キーマップかキーマップのリスト (19.12 節「メニュー・キーマップ」p.311 参照)、若しくは次のような形式をとります:

(*title* *pane1* *pane2*...)

それぞれの枠 (pane) は次の形式のリストです。

(*title* (*line* . *item*)...)

それぞれの *line* は文字列で、 *item* はその *line* が選ばれたときに返される値です。

使用上の注意: あるメニューが前置キーにメニュー・キーマップとしてバインドされている場合は、そのメニューを表示するのに `x-popup-menu` を使ってはいけません。メニューをつくるのにメニュー・キーマップを使った場合は、 `C-h c` と `C-h a` でそのメニューの中のそれぞれの項目を見ることやヘルプを出すことができます。そうではなく、 `x-popup-menu` を呼び出すコマンドによってメニューを定義した場合は、ヘルプ機能はそのコマンドの中で何が行なわれているのかを知ることはできませんので、そのメニューの項目に関してヘルプを出すことはできません。そのため、メニュー・バーの全ての項目は普通はメニュー・キーマップを用いて作られています。

26.16 ダイアログ・ボックス

ダイアログ・ボックスはポップアップ・メニューの変形です。常にフレームの真中に現れ、一つの階層と一つの枠 (pane) しか持っていないところが違います (X Toolkit を使っている場合)。ダイアログ・ボックスは、 "yes" か "no" などで答えられる質問をするのに主に使われます。関数 `y-or-n-p` と `yes-or-no-p` は、マウス・クリックで起動されたコマンドから呼び出された場合、キーボードの代わりにダイアログ・ボックスを使用します。

x-popup-dialog *position contents* Function

この関数はポップアップ・ダイアログ・ボックスを表示し、ユーザが何を選択したかを示すものを返します。引数 *contents* には選択肢を以下のフォーマットで指定します:

(*title* (*string* . *value*)...)

これは `x-popup-menu` に一つの枠を指定するときのリストに似ています。

返り値は選ばれた選択肢の *value* になります。

リストの要素は (*string* . *value*) というコンス・セルの代わりにただの文字列でも構いません。その場合は選択できない箱になります。

リストの中に `nil` がある場合は、項目を右と左に分ける働きをします。 `nil` よりも前にある項目は左に、後にある項目は右に表示されます。リストの中に `nil` がない場合は、おおよそ半分ずつ左右に分けられます。

ダイアログ・ボックスは、常にフレームの中央に表示されます。引数 *position* はどのフレームに表示するかを指定します。とりうる値は `x-popup-menu` と同様ですが、細かい座標は無視され、フレームだけが認識されます。

あなたの Emacs の実行形式が X toolkit を使用しない設定で作られている場合は、本物のダイアログ・ボックスは使えません。その代わり同じ項目のポップアップ・メニューをフレームの中央に表示します。

26.17 ポインタの形状

これらの変数はいろいろな状況でのマウス・ポインタの形状を指定します:

`x-pointer-shape`
: この変数は Emacs のフレーム中の普通のポインタの形状を指定します。

`x-sensitive-text-pointer-shape`
: この変数はマウスがマウス感知テキストの上にあるときのポインタの形状を指定します。

これらの変数は新しく作られたフレームに影響を与えます。存在するフレームに影響することは普通はありませんが、フレームのマウスの色をセットするとポインタの形状もこれらの変数の現在の値に基づいて更新されます。 26.3.3 節「X ウインドウ・フレーム・パラメータ」p.438 を参照してください。

これらのポインタの形状のどちらをも指定するのに使える値は、ファイル `lisp/x-win.el` で定義されています。これらのリストを見るには `M-x apropos RET x-pointer RET` を使ってください。

26.18 X セレクション

X サーバーはアプリケーション間でデータを交換できるようにするために『セレクション (selection)』のセットを記録します。さまざまなセレクションは『セレクション・タイプ (selection type)』で識別されます。セレクション・タイプは Emacs 上ではシンボルで表されます。Emacs を含む X クライアントは、与えられたどのタイプのセレクションでも読んだりセットしたりできます。

x-set-selection *type data* Function

この関数は X サーバの "セレクション" をセットします。セレクション・タイプ *type* とそれに割り当てる値 *data* の二つの引数をとります。 *data* が `nil` の場合は、そのセレクションをクリアします。それ以外に *data* には次のいずれかを指定できます。文字列、シンボル、整数 (または二つの整数のコンスやリスト)、オーバーレイ、同じバッファを指している二つのマーカのコンス。オーバーレイやマーカのペアは、オーバーレイの中やマーカの間のテキストを表します。

data は、上記非ベクタのセレクションの値のベクタでも構いません。

それぞれの可能な *type* は、それ自身の独立に変更可能なセレクションの値を持っています。通常 *type* の値は `PRIMARY` と `SECONDARY` です。これらのシンボルは X ウインドウ・システムの慣習に合わせて大文字の名前になっています。デフォルトは `PRIMARY` です。

x-get-selection &optional *type data-type* Function

この関数は、 Emacs やほかの X クライアントがセットしたセレクションにアクセスします。この関数は二つのオプション引数 *type* と *data-type* をとります。セレクション・タイプ *type* のデフォルトは `PRIMARY` です

data-type 引数は、ほかの X クライアントから得た生のデータを Lisp データに変換するときに使用するデータ変換の形式を指定します。意味がある値は、 `TEXT`、 `STRING`、 `TARGETS`、 `LENGTH`、 `DELETE`、 `FILE_NAME`、 `CHARACTER_POSITION`、 `LINE_NUMBER`、 `COLUMN_NUMBER`、 `OWNER_OS`、 `HOST_NAME`、 `USER`、 `CLASS`、 `NAME`、 `ATOM`、 `INTEGER` です (これらのシンボルは X の慣習に合わせて大文字の名前になっています)。デフォルトは `STRING` です。

X サーバーは、テキストやほかのデータをアプリケーション間でやりとりできる、番号付の『カット・バッファ (cut buffer)』のセットも持っています。カット・バッファは obsolete とみなされていますが、 Emacs はまだこれを使っている X クライアントのためにサポートしています。

x-get-cut-buffer *n* Function

この関数は番号 *n* のカット・バッファの中身を返します。

x-set-cut-buffer *string* Function

この関数は、 *string* を先頭のカット・バッファ (カット・バッファ 0) に入れ、 Emacs で連続した kill を行なったときに、 kill リングを繰り下げるのと同様に、ほかのカット・バッファの全ての値を繰り下げます。

26.19 色の名前

x-color-defined-p *color* Function

この関数は色の名前 *color* が有効かどうかを報告します。有効なら `t`、そうでないなら `nil` を返します。

これは使っているディスプレイがその色をちゃんとサポートしているかどうかは教えてくれないことに注意してください。任意の種類のディスプレイで定義されている色のどれかについて問い合わせて、 X サーバーがどう働くかについてある結果を得ることができます。これはあなたのディスプレイが色 *color* をサポートしているかどうかテストする近似的方法です:

```
(defun x-color-supported-p (color)
  (and (x-color-defined-p color)
       (or (x-display-color-p)
           (member color '("black" "white"))
           (and (> (x-display-planes) 1)
                (equal color "gray")))))
```

x-color-values *color* Function

この関数は *color* が理想的にはどんな感じで見えるべきかを記述する値を返します。 *color* が定義されていれば、値は赤の量、緑の量、青の量を表す三つの整数のリストです。それぞれの整数の範囲は原則として 0 から 65535 ですが、実際には 65280 以上の値はないようです。 *color* が定義されていなければ値は `nil` です。

```
(x-color-values "black")
     ⇒ (0 0 0)
(x-color-values "white")
     ⇒ (65280 65280 65280)
(x-color-values "red")
     ⇒ (65280 0 0)
(x-color-values "pink")
     ⇒ (65280 49152 51968)
(x-color-values "hungry")
     ⇒ nil
```

26.20 X のリソース

x-get-resource *attribute class* &optional *component subclass* Function

この関数 `x-get-resource` は、 X ウインドウのデフォルト・データベースからリソースの値を取り出します。

リソースは、『キー (key)』と『クラス (class)』の組合せの索引がつけられています。この関数は、'*instance*.*attribute*'(*instance* は、 Emacs が起動されたときの名前です) の形式をキーに、'Emacs' をクラスとして使って検索します。

オプション引数 *component* と *subclass* は、それぞれキーとクラスにつけ加えられます。これらは両方とも指定するか、両方とも指定しないかのどちらかでなければなりません。これらを指定したときにはキーは '*instance*.*component*.*attribute*' となり、クラスは 'Emacs.*class*.*subclass*' となります。

The GNU Emacs Manual の "X Resources" 節を参照してください。

26.21 X サーバーに関するデータ

この節では、 Emacs が使っている X サーバーの能力や素性に関する情報を得るための関数について説明します。これらの関数それぞれにはあなたが注目しているディスプレイを指定するようになっています。 *display* 引数にはディスプレイの名前とフレーム (つまりそのフレームが表示されているディスプレイを使います) のどちらでも指定できます。 *display* 引数を省略、もしくは `nil` を指定した場合は、選択されたフレームのディスプレイを使うことになります。

x-display-screens &optional *display* — Function

この関数はそのディスプレイがいくつ画面を持っているかを返します。

x-server-version &optional *display* — Function

この関数はそのディスプレイで動いている X サーバーのバージョン番号のリストを返します。

x-server-vendor &optional *display* — Function

この関数はその X サーバーのソフトウェアを提供しているベンダーを返します。

x-display-pixel-height &optional *display* — Function

この関数はその画面の高さをピクセル単位で返します。

x-display-mm-height &optional *display* — Function

この関数はその画面の高さをミリメートル単位で返します。

x-display-pixel-width &optional *display* — Function

この関数はその画面の幅をピクセル単位で返します。

x-display-mm-width &optional *display* — Function

この関数はその画面の幅をミリメートル単位で返します。

x-display-backing-store &optional *display* — Function

この関数はその画面のバッキング・ストアの能力を返します。返り値は、 `always`、`when-mapped`、 `not-useful` です。

x-display-save-under &optional *display* Function

この関数はそのディスプレイが SaveUnder 機能をサポートしている場合非 `nil` を返します。

x-display-planes &optional *display* Function

この関数はそのディスプレイがサポートしているプレーンの数を返します。

x-display-visual-class &optional *display* Function

この関数はその画面のビジュアル・クラスを返します。返り値は次のシンボル、`static-gray`、`gray-scale`、`static-color`、`pseudo-color`、`true-color`、`direct-color` のいずれかです。

x-display-grayscale-p &optional *display* Function

この関数はその画面が灰色の影を表示できるときに `t` を返します。

x-display-color-p &optional *display* Function

この関数はその画面がカラー画面のときに `t` を返します。

x-display-color-cells &optional *display* Function

この関数はその画面がサポートしているカラー・セルの数を返します。

27 位置

『位置 (position)』はバッファ・テキスト中の文字を指すものです。もっと詳しくいえば、位置は文字と文字との間 (あるいは最初の文字の前、または最後の文字の後) を指すものなので、その文字はある位置の前あるいは後にある、ということができます。しかしたいてい、その文字のことをその位置「に」あるといって、その位置の後ろにあることを意味します。

位置は通常 1 から始まる整数で表現しますが、『マーカ (marker)』で表現することもできます。マーカはテキストが挿入あるいは削除されても、マーカを挟む文字がある場所にそのまま留まるように自動的に再配置される特別なオブジェクトです。第 28 章「マーカ」 p.471 を参照してください。

27.1 ポイント

『ポイント (point)』は特別なバッファ位置で、タイプされた文字の自己挿入やテキスト挿入関数を含む、多くの編集コマンドで使われます。ポイントを使うほかのコマンドは、異なる場所での編集や挿入ができるように、ポイントをテキスト内で移動させます。

ほかの位置同様に、ポイントは特定の文字というよりは、文字と文字の間 (あるいは最初の文字の前、または最後の文字の後) の場所を示します。普通、端末ではカーソルはポイント直後の文字の上に表示されます。ポイントは実際はカーソルがある文字の前にあります。

ポイントの値は 1 ~ バッファ・サイズ +1 の間の数です。ナローイング中には (27.4 節「ナローイング」 p.467 参照) ポイントはバッファのアクセス可能部分 (おそらくはその一端) に収まるように拘束されます。

各バッファはそれ自身固有のポイントの値を持っています。ほかのバッファのポイントの値とは独立です。各ウィンドウも同様にポイントの値を持っていて、同じバッファのほかのウィンドウでのポイントの値とは独立です。これが同じバッファを表示するさまざまなウィンドウにおいて、異なる値のポイントをもつことができる理由です。バッファが一つのウィンドウのみに現れている場合は、バッファのポイントとウィンドウのポイントは普通同じ値を持ちます。よってこれらの区別はあまり重要ではありません。詳細は、 25.9 節「ウィンドウとポイント」 p.421 を参照してください。

point Function

この関数は、カレント・バッファのポイントの値を整数で返します。

```
(point)
     ⇒ 175
```

point-min Function

この関数は、カレント・バッファでのポイントのアクセス可能な値の最小値を返します。これは通常 1 ですが、ナローイング中には、ナローイングしたリージョンの先頭位置となります (27.4 節「ナローイング」 p.467 を参照してください)。

point-max Function

この関数は、カレント・バッファでのポイントのアクセス可能な値の最大値を返します。これは通常 `(1+ (buffer-size))` ですが、ナローイング中には、ナローイングしたリージョンの末尾位置となります (27.4 節「ナローイング」p.467 を参照してください)。

buffer-end *flag* Function

この関数は、 *flag* が 1 よりも小さければ、 `(point-min)` を返し、それ以外では `(point-max)` を返します。引数 *flag* は、数でなければなりません。

buffer-size Function

この関数は、カレント・バッファ中の全文字数を返します。ナローイング (27.4 節「ナローイング」p.467 参照) していないときならば、 `point-max` はこの値より 1 大きい値を返します。

```
(buffer-size)
     ⇒ 35
(point-max)
     ⇒ 36
```

27.2 移動

移動関数はポイントの値を変えます。ポイントの現在の値、あるいはバッファの先頭または末尾、あるいは選択されたウィンドウの端のいずれかからの相対的な値に変えます。 27.1 節「ポイント」p.457 を参照してください。

27.2.1 文字単位での移動

これら関数は文字単位で数えたポイント移動をします。 `goto-char` が基本的なプリミティブです。ほかの関数はこれを使っているだけです。

goto-char *position* Command

この関数は、カレント・バッファのポイントを *position* の値に設定します。 *position* が 1 よりも小さいときは、バッファ先頭にポイントを移動します。 *position* がバッファ長よりも大きいときは、バッファ末尾にポイントを移動します。

ナローイング中でも、 *position* はバッファ先頭から数えますが、ポイントはアクセス可能部分の外には出られません。 *position* が範囲外ならば、 `goto-char` はポイントをアクセス可能部分の先頭あるいは末尾に移動します。

この関数が対話的に呼ばれた場合、数値前置引数があれば *position* になります。そうでなければ、ミニバッファから読み込みます。

`goto-char` は *position* を返します。

forward-char &optional *count* Command

この関数は、ポイントを *count* 文字前方に、バッファ末尾に向かって進めます (もし *count* が負ならば、後方に、バッファ先頭に向かって進めます)。関数がポイントをバッ

ファの先頭あるいは末尾を (あるいはナローイング中ならばアクセス可能部分の限界を) 超えて進めようとすると、エラー・コード `beginning-of-buffer` あるいは `end-of-buffer` をともなうエラーが通知されます。

対話的呼出しでは、 *count* は数値前置引数です。

backward-char &optional *count* Command

この関数は、ポイントを *count* 文字後方に、バッファ先頭に向かって進めます (もし *count* が負ならば、前方に、バッファ末尾に向かって進めます)。関数がポイントをバッファの先頭あるいは末尾を (あるいはナローイング中ならばアクセス可能部分の限界を) 超えて進めようとすると、エラー・コード `beginning-of-buffer` あるいは `end-of-buffer` をともなうエラーが通知されます。

対話的呼出しでは、 *count* は数値前置引数です。

27.2.2 単語単位での移動

これら語を解析する関数は、ある文字が語の一部であるか否かを決定するために構文テーブルを使います。第 31 章「構文テーブル」 p.541 を参照してください。

forward-word *count* Command

この関数は、ポイントを *count* 語前方に (あるいは `count` が負ならば後方に) 移動します。通常 `t` を返します。この移動でバッファ先頭あるいは末尾に、あるいはナローイング中のアクセス可能部分の限界に、達したならば、ポイントはそこで止まり、戻り値は `nil` となります。

対話的呼出しでは、 *count* は数値前置引数です。

backward-word *count* Command

この関数は、 `forward-word` と全く同じですが、前方ではなく、語頭に達するまで後方に移動するところが違います。

対話的呼出しでは、 *count* は数値前置引数です。

この関数をプログラム中で使うことは滅多にありません。 `forward-word` を負の引数で呼び出す方がよほど効率がいいからです。

words-include-escapes Variable

この変数は、 `forward-word` やそれを使うもの全ての振舞に影響します。非 `nil` ならば、「エスケープ」と「文字 quote」構文クラスの文字を語の一部として数えます。 `nil` ならば、そうしません。

27.2.3 バッファの端への移動

バッファ先頭へポイントを移動するには、こう書きます:

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

同様に、バッファ末尾へ移動するには、これを使います:

```
(goto-char (point-max))
```

ここにユーザがこういったことを行なうための二つのコマンドを示します。ここで説明するのは、 Lisp プログラムで使用しないように警告するためです。というのも、これらはマークを設定し、エコー領域にメッセージを表示するからです。

beginning-of-buffer &optional *n* Command

この関数は、ポイントをバッファ先頭に (あるいはナローイング中ならば、アクセス可能部分の限界に) 移動し、マークを以前の位置に設定します。 *n* が非 `nil` ならば、ポイントをバッファ先頭から *n* 割のところに置きます。

対話的呼出しでは、数値前置引数があれば *n* になります。なければ *n* は `nil` がデフォルトです。

この関数を Lisp プログラムで使ってはなりません！

end-of-buffer &optional *n* Command

この関数は、ポイントをバッファ末尾に (あるいはナローイング中ならば、アクセス可能部分の限界に) 移動し、マークを以前の位置に設定します。 *n* が非 `nil` ならば、ポイントをバッファ末尾から *n* 割のところに置きます。

対話的呼出しでは、数値前置引数があれば *n* になります。なければ *n* は `nil` がデフォルトです。

この関数を Lisp プログラムで使ってはなりません！

27.2.4 テキスト行単位での移動

テキスト行は改行文字で区切られたバッファの部分です。改行文字は行末にあるとみなします。第 1 のテキスト行はバッファ先頭から始まり、最後のテキスト行は、バッファ最後の文字が改行であろうとなかろうと、バッファ末尾で終ります。バッファのテキスト行への分割は、ウィンドウの幅にも、表示での行継続にも、タブや制御文字がどのように表示されようとも、影響されません。

goto-line *line* Command

この関数は、バッファ先頭を 1 と数えた、 *line* 番目の行頭にポイントを移動します。 *line* が 1 よりも小さければ、ポイントをバッファの先頭に移動します。 *line* がバッファの行数よりも大きいならば、バッファ末尾にポイントを移動します (即ち、バッファの最後の行の行末に移動します)。これは `goto-line` で行頭に移動しない唯一の例です。

ナローイング中でも、 *line* はバッファ先頭から数えますが、ポイントはアクセス可能部分の外には出られません。したがって、 `goto-line` はポイントをアクセス可能部分の先頭あるいは末尾に移動します。

`goto-line` の戻り値は、 *line* と (ナローイングを考えに入れない、完全なバッファで) ポイントが実際に移動することができた行の行番号との差です。よって、スキャンがバッ

ファの実際の末尾に達した場合は、正の値となります。スキャンが実際のバッファ末尾ではなく、アクセス可能部分の末尾に達したときには、 0 の値となります。[1]

対話的呼出しでは、数値前置引数があれば *line* になります。なければ *line* はミニバッファで読み込みます。

beginning-of-line &optional *count* Command

この関数は、ポイントを現在の行の行頭に移動します。引数 *count* が `nil` でも 1 でもないならば、 $count-1$ 行前方に進んで、その行頭に移動します。

この関数がバッファ (あるいはナローイング中のアクセス可能部分) 末尾に達すると、ポイントをそこに置きます。エラーは通知されません。

end-of-line &optional *count* Command

この関数は、ポイントを現在の行の行末に移動します。引数 *arg* が `nil` でも 1 でもないならば、 $count-1$ 行前方に進んで、その行末に移動します。

この関数がバッファ (あるいはナローイング中のアクセス可能部分) 末尾に達すると、ポイントをそこに置きます。エラーは通知されません。

forward-line &optional *count* Command

この関数は、ポイントを *count* 行前方に移動し、行頭に置きます。 *count* が負ならば、ポイントを $-count$ 行後方に進め、行頭に置きます。 *count* が 0 ならば、ポイントを現在の行の行頭に移動します。

`forward-line` が指定行以前にバッファ (あるいはアクセス可能部分) 先頭あるいは末尾に達すると、そこにポイントを置きます。エラーは通知されません。

`forward-line` は、 *count* と実際に移動することができた行数との差を返します。たった 3 行しかないバッファで、先頭から下に 5 行移動しようとすると、ポイントは最後の行の行末で止まって、戻り値は 2 になります。

対話的呼出しでは、 *count* は数値前置引数です。

count-lines *start end* Function

この関数は、カレント・バッファの位置 *start* と位置 *end* の間にある行数を返します。 *start* と *end* が等しければ、 0 を返します。そうでなければ、たとえ *start* と *end* が同じ行にあっても、少なくとも 1 を返します。これは、それらの間にあるテキストは、隔離して考えると、空でないかぎり少なくとも 1 行はないとおかしいからです。

`count-lines` の使用例です:

```
(defun current-line ()
  "Return the vertical position of point..."
  (+ (count-lines (window-start) (point))
     (if (= (current-column) 0) 1 0)
     -1))
```

[1] 訳注: `goto-line` は `save-restriction` して `widen` してから実行される。よってこの場合、いったん目的の行までポイントが移動した後に、アクセス可能部分末尾までポイントは戻されることになる。

29.1 節「ポイント近くのテキストの検査」p.479 の関数 `bolp` と `eolp` も見てください。これら関数はポイントを移動しませんが、それがすでに行頭あるいは行末にあるかないかを検査します。

27.2.5 画面行単位での移動

前節の行を扱う関数では、改行文字のみで区切られるテキスト行を単位としました。一方、ここで述べる関数はテキストの画面への現れ方による画面行を単位とします。テキスト行 1 行は、それが十分に短く、選択されたウィンドウの幅に収まるならば、 1 行の画面行になります。しかしそうでない場合は画面行何行かに渡ることがあります。

いくつかの場合、テキスト行は次の画面行に継続しないで、画面分で打ち切られることがあります。これらの場合、 `vertical-motion` はポイントを `forward-line` と同じように移動します。 35.3 節「切捨て表示」p.602 を参照してください。

与えられた文字列の幅はある種の文字の表示を制御するフラグに依存するので、与えられたテキストについての `vertical-motion` の振舞は、そのテキストが入っているバッファ、さらには選択されたウィンドウによっても異なります (幅、打ち切りフラグ、表示テーブルはウィンドウごとに違うかもしれないので)。 35.13 節「普通の表示方法」p.617 を参照してください。

これら関数は、画面行がどこで区切られるかを判別するためにテキストをスキャンします。そのため、スキャンした長さに比例して時間がかかります。これらを重点的に使用するのであれば、 Emacs が用意しているキャッシュを自作コードの性能向上のために使うことができます。 27.2.4 節「テキスト行単位での移動」p.460 を参照してください。

vertical-motion *count* &optional *window* Function

この関数は、 *count* 画面行下にある画面行の始めにポイントを移動します。 *count* が負ならば、代わりに上に移動します。

`vertical-motion` は移動した行数を返します。バッファ先頭か末尾に達した場合は、戻り値は絶対値で *count* よりも小さくなるかも知れません。

ウィンドウ *window* は幅、水平スクロール、表示テーブルなどのパラメータを得るために使用します。しかし *vertical-motion* は、たとえ *window* が現在何かほかのバッファを表示していたとしても、常にカレント・バッファに対して作用します。

move-to-window-line *count* Command

この関数は、選択されたウィンドウに現在表示されているテキストについてポイントを移動します。ウィンドウの上端から *count* 行目の画面行の先頭にポイントを移動します。 *count* が負ならば、下 (あるいはバッファが指定した画面位置より上で終っているならば、バッファ最後の行) から $-count$行目の位置を指します。

count が `nil` ならば、ウィンドウの真中の行頭にポイントを移動します。 *count* の絶対値がウィンドウ・サイズよりも大きいならば、ウィンドウが十分高かったならば表示されたであろう画面行の位置にポイントが移動します。これによってその位置を画面に持ってくるためのスクロールを行なう再描画が起きる可能性があります。

対話的呼出しでは、 *count* は数値前置引数です。

戻り値はポイントが移動した画面行番号です。ウィンドウ上端の行を 0 と数えます。

compute-motion *from frompos to topos width offsets window* Function

この関数は、カレント・バッファをスキャンし、画面位置を計算します。位置 *from* が画面座標 *frompos* にあると仮定し、そこから、位置 *to* あるいは座標 *topos* のいずれか先にきた方までバッファを前方にスキャンします。この関数は、スキャンが終ったバッファ位置と画面座標を返します。

座標引数 *frompos* と *topos* は、(*hpos* . *vpos*) という形式のコンス・セルです。

引数 *window* はテキスト表示に使える桁数です。これは継続行の扱いに影響します。選んだウィンドウについて `window-width` が返した値を使いましょう。通常は (`window-width` *window*) を使用します。

引数 *offset* は、 `nil` あるいは (*hscroll* . *tab-offset*) という形式のコンス・セルのいずれかです。ここで *hscroll* は左端で表示されていない桁数です。呼び手はほとんどの場合 `window-hscroll` からこの値を得ます。一方 *tab-offset* は画面の桁数とバッファの桁数とのオフセットです。継続行で、それまでの画面行の幅の合計が `tab-width` の整数倍にならないときに、この値が非 0 になり得ます。非継続行においては必ず 0 になります。

ウィンドウ *window* は、どの表示テーブルを使うかを指定するためだけに働きます。 `compute-motion` は常にカレント・バッファに作用します。 *window* にどのバッファが表示されているかは関係ありません。

戻り値は要素が五つのリストです:

(*pos vpos hpos prevhpos contin*)

ここで *pos* はスキャンが停止したバッファ位置で、 *vpos* は垂直画面位置、 `hpos` は水平画面位置です。

結果の *prevhpos* は *pos* から 1 文字戻った水平位置です。結果 *contin* はその最後の行が先の直前の文字に継続するならば、 `t` です。

たとえば、あるウィンドウの行 *line* 桁 *col* のバッファ位置を見つけるには、ウィンドウの表示開始位置を *from* として、ウィンドウの左上座標を *frompos* として渡します。バッファの (`point-max`) を *to* として渡してスキャンがバッファのアクセス可能部分の末尾を限界とするようにし、 *line* と *col* を *topos* として渡します。これを行なう関数を挙げます:

```
(defun coordinates-of-position (col line)
  (car (compute-motion (window-start)
                       '(0 . 0)
                       (point-max)
                       (cons col line)
                       (window-width)
                       (cons (window-hscroll) 0)
                       (selected-window))))
```

`compute-motion` をミニバッファに対して使用する場合は、第 1 画面行の行頭の水平位置を取るために、 `minibuffer-prompt-width` を使う必要があります。 17.8 節「ミニバッファそのほか」p.254 を参照してください。

27.2.6 釣り合った式をまたがる移動

ここに括弧が釣り合った式 (Emacs では、それらをまたがる移動に関するときは『S 式 (sexp)』とも呼びます) に関する関数をいくつか挙げます。構文テーブルはこれら関数がさまざまな文字をどう解釈するかを制御します。第 31 章「構文テーブル」p.541 参照。 S 式のあるいは S 式の部分のスキャンのための、より低水準なプリミティブについては、 31.5 節「釣り合った式の解析」p.548 を参照してください。ユーザレベルのコマンドについては、 *GNU Emacs Manual* の “Lists and Sexps” 節を参照。

forward-list *arg* — Command

この関数は、 *arg* 個の釣り合った括弧群をまたがって前方に移動します (ほかの構文要素たとえば語や対になる文字列 quote は無視します)。

backward-list *arg* — Command

この関数は、 *arg* 個の釣り合った括弧群をまたがって後方へ移動します (ほかの構文要素たとえば語や対になる文字列 quote は無視します)。

up-list *arg* — Command

この関数は、 *arg* 段の括弧の外へ向けて前方に移動します。負の引数は後方に、しかしさらに浅い場所に向けての移動を意味します。

down-list *arg* — Command

この関数は、 *arg* 段の括弧の中へ向けて前方に移動します。負の引数は後方に、しかしさらに深い括弧に向けて ($-arg$ 段) の移動を意味します。

forward-sexp *arg* — Command

この関数は、 *arg* 個の釣り合った式をまたがって前方に移動します。釣り合った式には括弧で区切られたものや、ほかの種類の語や文字列定数などの両方を含みます。たとえば、こうなります。

```
---------- Buffer: foo ----------
(concat⋆ "foo " (car x) y z)
---------- Buffer: foo ----------

(forward-sexp 3)
     ⇒ nil

---------- Buffer: foo ----------
(concat "foo " (car x) y⋆ z)
---------- Buffer: foo ----------
```

backward-sexp *arg* — Command

この関数は、 *arg* 個の釣り合った式をまたがって後方に移動します。

beginning-of-defun *arg* Command

この関数は、 *arg* 番目の関数定義の先頭に向けて後方に移動します。 *arg* が負なら、実際に前方に移動しますが、やはり関数定義の先頭に移動し、その末尾には移動しません。

end-of-defun *arg* Command

この関数は、 *arg* 番目の関数定義の末尾に向けて前方に移動します。 *arg* が負なら、実際に後方に移動しますが、やはり関数定義の末尾に移動し、その先頭には移動しません。

defun-prompt-regexp User Option

非 `nil` ならば、この変数はどんなテキストが、関数定義を開始する開き括弧の前に現れるかを指定する正規表現を保持します。つまり、関数定義はこの正規表現に一致するもので開始し、続いて開き括弧構文の文字がある、そういう行で始まります。

27.2.7 文字を飛ばす

以下の二つの関数は指定した集合の文字を超えるようにポイントを移動します。たとえば、空白を飛ばすためによく使います。関連する関数については、 31.4 節「移動と構文」 p.547 を参照してください。

skip-chars-forward *character-set* &optional *limit* Function

この関数は、カレント・バッファで、与えられた集合の文字を飛ばして、ポイントを前方に移動します。ポイントの次の文字を調べて、それが *character-set* に一致するならポイントを進めます。これを一致しない文字に達するまで繰り返します。この関数は、 `nil` を返します。

引数 *character-set* は正規表現の '`[...]`' の中のようにします。ただし、 '`]`' はもはや特殊ではなく、 '\' が '^', '-', '\' を quote するところが違います。そのため、 `"a-zA-Z"`で全ての英文字を飛ばし、最初の非英文字の前で停止し、 `"^a-zA-Z"`で非英文字を飛ばして最初の英文字の前で停止します。 30.2 節「正規表現」 p.525 を参照してください。

limit(数かマーカでなければならない) を与えると、これがバッファにおいてポイントが進める最大位置の指定になります。ポイントは *limit* で、あるいはその前で停止します。

以下の例では、ポイントは最初 '`T`' のすぐ前に位置します。形式の評価後、ポイントはその行末 ('`hat`' の '`t`' と改行の間) に位置します。この関数は全ての英文字とスペースを飛ばしますが、改行は飛ばしません。

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------
```

```
(skip-chars-forward "a-zA-Z ")
     ⇒ nil

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

skip-chars-backward *character-set* &optional *limit* Function

この関数は、 *character-set* に一致する文字を飛ばして、 *limit* までポイントを後方に移動します。 `skip-chars-forward` と全く同じですが、移動の方向が違います。

27.3 脱線

プログラムのある限定された範囲内で「一時的に」ポイントを移動したり、一時的にバッファを切替えたりすることが、便利なこともあります。これを『脱線 (excursion)』と呼び、 `save-excursion` という特殊形式で行ないます。この構文要素は、脱線が完了したときにカレント・バッファとそのポイントおよびマークを復元できるように、これらを保存します。

ウィンドウの構成の保存と復元のための形式は、別の場所で説明します (25.16 節「ウィンドウ構成」p.432 と、 26.12 節「フレーム構成」p.449 参照を参照)。

save-excursion *forms*. . . Special Form

`save-excursion` 特殊形式はカレント・バッファが何であったか[2]とその中のポイントおよびマークの値を保存し、 *forms* を評価して、最後に、バッファとその保存されたポイントおよびマークの値を復元します。三つの保存された値全てが、 `throw` やエラーによる異常終了 (9.5 節「非局所脱出」p.116 参照) の場合にも復元されます。

`save-excursion` 特殊形式は、プログラムの残りの部分への影響を避けてプログラムのある部分内でバッファを切替えたりポイントを移動するための標準手法です。 Emacs の Lisp ソース中では、 500 回以上も使われています。

`save-excursion` は、ほかのバッファのポイントおよびマークの値は保存しません。そのためほかのバッファでの変更は `save-excursion` を抜けてもそのまま有効です。

つけ加えると、 `save-excursion` は `switch-to-buffer` のような関数によって変えられたウィンドウ - バッファの対応は復元しません。これら対応と選択されたウィンドウを復元する一つの方法は、 `save-window-excursion` を `save-excursion` 内で使うことです (25.16 節「ウィンドウ構成」p.432 参照)。

`save-excursion` が返す値は、 *forms* の最終結果となるか、あるいは *forms* が与えられなければ `nil` となります。

[2] 訳注: 原語は identity。

```
(save-excursion
  forms)
≡
(let ((old-buf (current-buffer))
      (old-pnt (point-marker))
      (old-mark (copy-marker (mark-marker))))
  (unwind-protect
      (progn forms)
    (set-buffer old-buf)
    (goto-char old-pnt)
    (set-marker (mark-marker) old-mark)))
```

27.4 ナローイング

『ナローイング (narrowing)』とは、 Emacs の編集コマンドがアドレス可能なテキストを、バッファ内の限られた範囲内の文字に制限することを意味します。アドレス可能なまま残ったテキストをそのバッファの『アクセス可能部分 (accessible portion)』と呼びます。

ナローイングは二つのバッファ位置によって指定され、それらがアクセス可能部分の先頭と末尾になります。ほとんどの編集コマンドや Emacs のプリミティブについて、これらの位置でバッファの先頭と末尾の値を置き換えます。ナローイング中には、アクセス可能部分の外側のテキストは全く表示されず、ポイントもアクセス可能部分の外側には移動できません。

位置や行番号といった値は、普通バッファ先頭から数えますし、ナローイング中でも同じように数えます。しかし、これらを使う関数はアクセス不能なテキストを操作することは拒否します。

バッファを保存するコマンドは、ナローイングの影響を受けません。これらはいかなるナローイングにも関係なく、バッファ全体を保存します。

narrow-to-region *start end* Command

この関数は、カレント・バッファの *start* を先頭に、 *end* を末尾にしたアクセス可能部分を設定します。両引数は、文字位置でなければなりません。

対話的呼出しでは、 *start* と *end* には現在のリージョン境界 (ポイントとマークで、小さい方が先です) が設定されます。

narrow-to-page *move-count* Command

この関数は、カレント・バッファの現在のページだけがアクセス可能部分となるように設定します。オプションの第 1 引数 *move-count* が非 `nil` ならば、 *move-count* ページ前方あるいは後方に移動してからナローイングすることを意味します。変数 `page-delimiter` でどこがページが先頭と末尾であるのかを指定します (30.8 節「編集用の標準正規表現」p.539 参照)。

対話的呼出しでは、 *move-count* には数値前置引数が設定されます。

widen Command

この関数は、カレント・バッファのあらゆるナローイングを取り消し、その結果中身全てがアクセス可能になります。これを『ワイドニング (widening)』と呼びます。これは、次の式と等価です:

```
(narrow-to-region 1 (1+ (buffer-size)))
```

save-restriction *body...* Special Form

本特殊形式はアクセス可能部分の現在の境界を保存し、 *body* の形式を評価して、最後に、保存した境界を復元して以前有効だったナローイング (あるいはそれがない状態) と同じ状態を復元します。ナローイングの状態は、 `throw` やエラーによる異常終了 (9.5 節「非局所脱出」p.116 参照) の場合にも復元されます。そのため、本構文要素は一時的なバッファのナローイングにきれいな方法です。

`save-restriction` が返す値は、 *body* の最後の形式が返す値となるか、あるいは *body* に本体形式が与えられないならば `nil` となります。

警告: `save-restriction` 構文要素を使用する場合、間違いを犯しやすいです。試す前にここにある全説明を読んでください。

body がカレント・バッファを変更しても、 `save-restriction` はそれでも元のバッファ (制限を保存したバッファ) の制限を復元します。しかし、カレント・バッファがなんであったかまでは復元しません。

`save-restriction` はポイントおよびマークの復元は行ないません。その目的には `save-excursion` を使ってください。 `save-restriction` と `save-excursion` とを両方一緒に使うときは、 `save-excursion` が先に (外側に) 来なければなりません。さもなくば、古いポイントの値を一時的ナローイングの最中に復元してしまうからです。その場合、古いポイントの値が一時的ナローイングの限界の外にあると、正確に復元できないことになります。

`save-restriction` 特殊形式はアクセス可能部分の先頭と末尾の値を、バッファの先頭と末尾からの距離で記録します。いいかえれば、これはアクセス可能部分の前後にあるアクセス不能なテキストの量を記録するのです。

この方法によって、 *body* がさらにナローイングを行なった場合も正しい結果が得られます。けれども `save-restriction` は *body* 本体がワイドニングを行って、保存されたナローイングの外側の範囲を変更した場合には混乱するかもしれません。こういうことを行ないたい場合には、 `save-restriction` は正しい道具ではありません。代わりにこうしなければなりません:

```
(let ((beg (point-min-marker))
      (end (point-max-marker)))
  (unwind-protect
      (progn body)
    (save-excursion
      (set-buffer (marker-buffer beg))
      (narrow-to-region beg end))))
```

これが `save-restriction` の正しい使い方の単純な例です:

```
---------- Buffer: foo ----------
This is the contents of foo
This is the contents of foo
This is the contents of foo⋆
---------- Buffer: foo ----------

(save-excursion
  (save-restriction
    (goto-char 1)
    (forward-line 2)
    (narrow-to-region 1 (point))
    (goto-char (point-min))
    (replace-string "foo" "bar")))

---------- Buffer: foo ----------
This is the contents of bar
This is the contents of bar
This is the contents of foo⋆
---------- Buffer: foo ----------
```

28 マーカ

『マーカ (marker)』とは、バッファ内のテキストに結び付けられた位置を示すために使われる Lisp オブジェクトのことです。テキストの挿入や削除を行なうと、それに追従して、マーカの位置 (バッファの先頭を起点とした) が自動的に変更されます。つまり、マーカは、テキストの特定の場所に貼り付いていることになります。

28.1 マーカとは?

マーカとは、バッファと、そのバッファ内での位置を特定するものです。マーカは、位置を表すものとして、通常の整数による表現と同じように、関数の中で使用することが出来ます。詳しい説明は第 27 章「位置」p.457 を参照してください。

マーカには、二つの属性があります。一つはそのマーカの位置であり、もう一つはそのマーカの属するバッファです。位置属性は、バッファ内のマーカの (その瞬間の) 位置に等しい整数です。ただし、位置属性の値は、そのマーカが存在する間に頻繁に変更されることもあります。テキストの挿入や削除が行なわれると、マーカは再配置されます。これは、挿入や削除によるテキストの移動にマーカも追従させるということで、挿入や削除が行なわれる前後でマーカの両側の文字が常に同じであるようにマーカの位置を変更するということです。再配置によって、マーカの位置属性の値が変更されます。

マーカの周囲のテキストが削除されても、マーカは、削除されたテキストの前後の文字の間に残ります。また、マーカの存在する位置にテキストが挿入されると、通常は、挿入されたテキストの直前にマーカが残ります。ただし、 `insert-before-markers` により挿入された場合は、マーカの位置は、挿入されたテキストの直後になります。 (29.4 節「テキストの挿入」p.482 参照)。

テキストの挿入・削除にともなって、バッファ内の全てのマーカがチェックされ、必要ならば再配置が行なわれます。そのため、マーカの数が多いと、挿入や削除に時間がかかるようになります。ですから、もはや必要なくなったとわかっているマーカは、どこも指さないようにしておくのがよいでしょう。どこからも参照されていないマーカはいずれガベージ・コレクトされますが、もしそれがどこかを指していたら、ガベージ・コレクトされるまではチェックや再配置の対象となり、時間を浪費することになります。

マーカの位置に対して算術演算を施すことはよく行なわれることなので、 `+` や `-`を含むたいていの演算子の引数として、マーカそのものを渡せるようになっています。このような場合、マーカは、その時点で指し示す位置を表す整数値であるかのように振る舞います。

以下に、マーカの作り方、マーカの設定、マーカ位置へのポイントの移動の例を示します。

```
;; とりあえずどこも指していないマーカを作ります。
(setq m1 (make-marker))
     ⇒ #<marker in no buffer>

;; このバッファの 99 文字目と 100 文字目の間にマーカ m1 を置きます。
(set-marker m1 100)
     ⇒ #<marker at 100 in markers.texi>
```

```
;; このバッファの先頭に一文字挿入してみましょう。
(goto-char (point-min))
     ⇒ 1
(insert "Q")
     ⇒ nil

;; 一文字挿入に合わせて m1 の内容が変更されます。
m1
     ⇒ #<marker at 101 in markers.texi>

;; 同じ場所を指す二つのマーカは、
;;   equal ですが、 eq ではありません。
(setq m2 (copy-marker m1))
     ⇒ #<marker at 101 in markers.texi>
(eq m1 m2)
     ⇒ nil
(equal m1 m2)
     ⇒ t

;; 必要なくなったら、マーカがどこも指さないようにします。
(set-marker m1 nil)
     ⇒ #<marker in no buffer>
```

28.2 マーカに対する述語

あるオブジェクトがマーカであるか調べることができます。また、マーカと整数のどちらかであるかを調べることもできます。後者は、マーカと整数のどちらでも引数にとることのできる算術関数を使う際に、便利でしょう。

markerp *object* Function

この関数は、 *object* がマーカであるなら `t` を返し、それ以外なら `nil` を返します。引数としてマーカと整数のどちらでもとることのできる関数は多く存在しますが、整数はマーカそのものではないことに注意してください。

integer-or-marker-p *object* Function

この関数は、 *object* がマーカもしくは整数であるなら `t` を返し、それ以外なら `nil` を返します。

number-or-marker-p *object* Function

この関数は、 *object* がマーカもしくは数であるなら `t` を返し、それ以外なら `nil` を返します。

28.3 マーカを作る関数

新たにマーカを作る際、どこも指さないマーカを作ったり、ポイント位置を指すように設定したり、バッファの (その時点で見ることができる) 先頭や最後に置いたり、ほかのマーカと同じ場所を指すように設定することができます。

make-marker Function

この関数は、どこも指さないマーカを新たに作って、それを返します。

```
(make-marker)
     ⇒ #<marker in no buffer>
```

point-marker Function

この関数は、カレント・バッファのポイント位置を指すマーカを新たに作って、それを返します。27.1 節「ポイント」p.457 を参照してください。例は、後述の copy-marker を参照してください。

point-min-marker Function

この関数は、バッファの (その時点で見ることができる) 先頭を指すマーカを新たに作って、それを返します。バッファがナローイングされていなければ、そのバッファの本当の先頭を指すことになります。27.4 節「ナローイング」p.467 を参照してください。

point-max-marker Function

この関数は、バッファの (その時点で見ることができる) 最後を指すマーカを新たに作って、それを返します。バッファがナローイングされていなければ、そのバッファの本当の最後を指すことになります。27.4 節「ナローイング」p.467 を参照してください。

この関数と point-min-marker の例を、以下に示します。この章の (訳注: 英語版の)texinfo ファイルを読み込んだバッファ上で実行した結果です。

```
(point-min-marker)
     ⇒ #<marker at 1 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 15573 in markers.texi>

(narrow-to-region 100 200)
     ⇒ nil
(point-min-marker)
     ⇒ #<marker at 100 in markers.texi>
(point-max-marker)
     ⇒ #<marker at 200 in markers.texi>
```

copy-marker *marker-or-integer* Function

この関数は、引数としてマーカが渡された場合、そのマーカ *marker-or-integer* と同じバッファ、同じ位置を指すマーカを作り、それを返します。引数として整数が渡された場合、カレント・バッファ上の位置 *marker-or-integer* を指すマーカを新たに作り、それを返します。

copy-marker に 1 よりも小さな整数を渡すと、カレント・バッファの先頭を指すマーカを返します。バッファの大きさよりも大きい整数を渡すと、バッファの最後を指すマーカを返します。

marker-or-integer がマーカでも整数でもなければ、エラーとなります。

```
(setq p (point-marker))
     ⇒ #<marker at 2139 in markers.texi>
```

```
(setq q (copy-marker p))
     ⇒ #<marker at 2139 in markers.texi>
(eq p q)
     ⇒ nil
(equal p q)
     ⇒ t
(copy-marker 0)
     ⇒ #<marker at 1 in markers.texi>
(copy-marker 20000)
     ⇒ #<marker at 7572 in markers.texi>
```

28.4 マーカから情報を得る

この節では、マーカのもつ情報を取り出す関数について説明します。

marker-position *marker* Function

この関数は、 *marker* が指す位置を返します。マーカがどこも指していない場合は、 `nil` を返します。

marker-buffer *marker* Function

この関数は、 *marker* が指す位置を含むバッファを返します。マーカがどこも指していない場合は、 `nil` を返します。

```
(setq m (make-marker))
     ⇒ #<marker in no buffer>
(marker-position m)
     ⇒ nil
(marker-buffer m)
     ⇒ nil
(set-marker m 3770 (current-buffer))
     ⇒ #<marker at 3770 in markers.texi>
(marker-buffer m)
     ⇒ #<buffer markers.texi>
(marker-position m)
     ⇒ 3770
```

二つのマーカが同じバッファの同じ位置に設定されているか、もしくはどちらもどこも指していない場合、この二つのマーカは (たとえ `eq` ではなくとも) `equal` と見做されます。

28.5 マーカ位置の変更

この節では、マーカの位置の変更の仕方について述べます。あるマーカの位置を変更する場合、それが自分のプログラム以外で使われていないか、もし、ほかで使われているなら、実際に

そのマーカを動かすことによってどのような影響がでるのか確認してください。このような確認を怠ると、 Emacs のほかの処理に混乱が生じる可能性があります。

set-marker *marker position* &optional *buffer* Function

この関数は、 *marker* を *buffer* の *position* へ移します。 *buffer* が省略された場合は、カレント・バッファに移されます。

position が 1 より小さい場合、 *marker* はバッファの先頭に移されます。 *position* がバッファの大きさより大きい場合は、バッファの最後に移されます。 *position* が `nil` だったり、どこも指していないマーカであったりした場合は、 *marker* はどこも指さないように設定されます。

返される値は、 *marker* です。

```
(setq m (point-marker))
     ⇒ #<marker at 4714 in markers.texi>
(set-marker m 55)
     ⇒ #<marker at 55 in markers.texi>
(setq b (get-buffer "foo"))
     ⇒ #<buffer foo>
(set-marker m 0 b)
     ⇒ #<marker at 1 in foo>
```

move-marker *marker position* &optional *buffer* Function

これは、 `set-marker` の別名です。

28.6 マーク

『マーク (the mark)』と呼ばれる特殊なマーカが、各バッファに一つずつ存在しています。これは、 `C-w` や `C-x TAB` のようなコマンドに備えて、位置を記憶しておくために使われるものです。 Emacs Lisp のプログラムがマークをセットするのは、ユーザが使いそうな場合に限るべきであり、プログラム内部での用途にマークを使用すべきではありません。たとえば、 `replace-regexp` 命令は、実際に置換を行なう前に、ポイントの位置をマークに入れておきます。これは置換が終了した後、元の場所に簡単に戻ることができるようにするためです。

対話的に実行された場合にポイントとマークに囲まれたテキストを加工するように作られたコマンドが、たくさんあります。このようなコマンドを作る場合は、直接マークの値を調べてはいけません。かわりに、 `interactive` 関数を ‘`r`’ 引数とともに呼び出しましょう。こうすると、対話的にコマンドを実行した場合、ポイントとマークの値が引数として渡されます。また、ほかの Lisp プログラムから呼び出す場合は、明示的に引数を与えることができます。 18.2.2 節「`interactive` 用のコード文字」p.259 を参照してください。

マークは、ほかのバッファのマークとは独立した、そのバッファ独自の値を持っています。バッファが作られるときにマークも一緒に作られますが、このマークはこの時点ではどこも指していません。この状態は “バッファにマークが存在しない” ものとして扱われます。

一度バッファ内にマークが “存在する” ようになった後では、マークが消えてしまうようなことはまずありません。ただし、 Transient Mark モードでは、マークが『無効 (inactive)』にな

ることがあります。 mark-active 変数は、全てのバッファにおいて常にローカルな変数で、マークが有効であるかどうかを示しています。この変数が非 nil なら、マークが有効であることを表します。また、変数 deactivate-mark を非 nil の値にセットすれば、エディタのコマンド・ループに戻る度にマークが無効になります。ただしこれは、 Transient Mark モードでのみ有効です。

ユーザが Transient Mark モードを使いはじめる大きな動機になり得るのは、マークが有効な状態にあるとき、リージョンがわかりやすく強調表示されるという点です。第 35 章「Emacs の表示」p.601 を参照してください。

バッファにはマークに加えて、『マーク・リング (mark ring)』があります。これは、以前のマークの値を保持しているマーカのリストです。マークの値を変更するコマンドを実行すると、通常、変更前のマークの値がマーク・リングに保存されます。変数 mark-ring-max は、マーク・リングの最大容量を示します。マーク・リングが一杯になってさらに新たにマークの値を保存する際には、一番古いマーカが消去されます。

mark &optional *force* Function

この関数は、カレント・バッファのマークの位置を、整数値で返します。

マークが無効の状態になっている場合は、通常、エラーになります。ただし、 *force* が非 nil なら、マークの状態いかんに関わらずその位置、もしくは nil(バッファにマークがまだ設定されていない場合) を返します。

mark-marker Function

この関数は、カレント・バッファのマークを返します。これはマークの位置を記録しているマーカのコピーではなく、マーカそのものを返します。したがって、このマーカの位置属性の値を変更すると、マークの位置も変わってしまいます。意図的でないかぎり、このような変更はしないでください。

```
(setq m (mark-marker))
     ⇒ #<marker at 3420 in markers.texi>
(set-marker m 100)
     ⇒ #<marker at 100 in markers.texi>
(mark-marker)
     ⇒ #<marker at 100 in markers.texi>
```

ほかのマーカと同じように、このマーカをほかのバッファに移すことができます。しかし、マークをほかのバッファに移すようなことは止めておいた方がいいでしょう。仮にそうした場合、問題は起こらないにしても、かなり奇妙な現象が発生することになるでしょう。

set-mark *position* Function

この関数は、 *position* を指すようにマークを設定し、また、マークを有効にします。変更前のマークの値は、マーク・リングには保存されません。

注意: この関数を使うのは、マークを動かしたということをユーザに見せ、なおかつ、直前のマークの位置情報を破棄する場合にのみ使ってください。通常は、マークを新たに設

定するときには、古いマークを mark-ring に保存しておくべきです。したがってほとんどの場合、 set-mark は使わずに、 push-mark と pop-mark を使うべきです。

Emacs Lisp プログラミングの初心者は、マークを間違った用途に使おうとしがちです。マークはユーザの便宜を図るために、位置を記憶するものです。編集用のコマンドでは、ユーザとのやり取りのためにマークを変更する必要がないかぎり、マークを変更すべきではありません。また、変更の必要がある場合でも、マークが変更されることをドキュメントに明記しておくべきです。 Lisp プログラム内部の都合でどこかの位置を記録しておくのなら、 Lisp の変数に保存しましょう。たとえば以下のようにします。

```
(let ((beg (point)))
  (forward-line 1)
  (delete-region beg (point))).
```

push-mark &optional *position nomsg activate* Function

この関数は、カレント・バッファのマークを、 *position* を指すように設定します。変更前の値は、 mark-ring に保存されます。 *position* が nil ならば、ポイントの値が使われます。この関数は、 nil を返します。

push-mark は通常、マークを有効にしません。 *activate* として t が渡されれば有効にします。

nomsg が非 nil でない限り、 'Mark set' というメッセージが表示されます。

pop-mark Function

この関数は、 mark-ring の先頭の要素を取り出して、それをバッファのマークとして設定します。ポイントは移動しません。 mark-ring が空なら何もしません。また、マークを無効にします。

返される値には、意味がありません。

transient-mark-mode User Option

この変数が非 nil であるとき、 Transient Mark モードが有効になります。このモードでは、バッファの内容を変更するプリミティブ関数が、 deactivate-mark を非 nil に設定するようになります。その結果、バッファ内容を変更するコマンドを実行するたびに、通常、マークが無効になります。

deactivate-mark Variable

この変数が非 nil であるとき、エディタのコマンド・ループは、コマンドを実行するたびにマークを無効にするようになります。ただし、これは Transient Mark モードでのみ有効です。

deactivate-mark Function

マークを無効にします。ただし、これは Transient Mark モードでのみ有効です。

mark-active Variable

この変数を非 nil に設定すると、マークが有効になります。この変数は、常にバッファローカルです。

activate-mark-hook Variable
deactivate-mark-hook Variable

これらの正規フックは、それぞれ、マークが有効になった時と無効になった時に実行されます。また、 `activate-mark-hook` は、マークが有効でリージョンが変更された時にも(その変更を行なったコマンドの直後に) 実行されます。

mark-ring Variable

この変数は、バッファローカルな変数であり、以前にカレント・バッファで設定されたことのあるマークの値を、リストにして保存しています。最も新しいマークがリストの先頭に入っています。

```
mark-ring
⇒ (#<marker at 11050 in markers.texi>
    #<marker at 10832 in markers.texi>
    ...)
```

mark-ring-max User Option

この変数は、 `mark-ring` の容量の最大値を保持します。この値より多くの個数のマークを `push-mark` で `mark-ring` に保存しようとすると、一番古いマークが捨てられます。

28.7 リージョン

ポイントとマークの間にあるテキストを、『リージョン (the region)』と呼びます。ポイントとマークで区切られたテキストを加工する関数はいろいろとありますが、ここでは特に、リージョンそのものに関わる関数について説明します。

region-beginning Function

この関数は、リージョンの先頭位置を、整数値で返します。これは、ポイント位置かマーク位置のどちらか小さい方です。

マークがどこも指していない場合は、エラーとなります。

region-end Function

この関数は、リージョンの最後の位置を、整数値で返します。これは、ポイント位置かマーク位置のどちらか大きい方です。

マークがどこも指していない場合は、エラーとなります。

`region-beginning` や `region-end` を使う必要があるプログラムはほとんどないはずです。リージョンを使用するようなコマンドは、通常、 `interactive` 関数を ‘`r`’ 引数とともに呼び出してリージョンの始まりと終りの位置を得るように作られるべきです。このようにすれば、ほかの Lisp プログラムからこのようなコマンドを呼び出す場合にも、リージョンの境界を引数として明示的に渡すことができます (18.2.2 節「`interactive` 用のコード文字」 p.259 を参照してください)。

29 テキスト

本章では、バッファ中のテキストを扱う関数について述べます。ほとんどがカレント・バッファのテキスト、それもポイント近辺の検査、挿入、消去を行なうものです。多くは対話的です。テキストを変更するすべての関数は、その変更の取り消しが可能です (29.9 節「取り消し」p.492 参照)。

テキスト関係の関数の多くは、 *start* および *end* という引数で渡される、二つのバッファ内位置で定義されるテキストのリージョンにたいして、操作を行ないます。これらの位置はマーカ (第 28 章「マーカ」p.471 参照) か、数値による文字の位置 (第 27 章「位置」p.457 参照) です。この二つの位置が与えられている順番は重要ではありません。 *start* で終り *end* で始まってもかまいません。たとえば、 `(delete-region 1 10)` と `(delete-region 10 1)` は等価です。どちらかの位置がバッファのバッファの範囲に存在しない場合、 `args-out-of-range` エラーになります。対話的に呼び出した場合、これらの引数値はポイントとマークになります。

本章では、「テキスト」は対象となるバッファ中の文字と、 (もし意味があるときは) テキストの属性を指します。

29.1 ポイント近くのテキストの検査

ポイントのまわりの文字を調べる多くの関数があります。ここではそのうちいくつかの簡単なものについて述べます。 30.3 節「正規表現の検索」p.530 の `looking-at` も参照してください。

char-after *position* — Function

この関数は、カレント・バッファの位置 *position* の (直後の) 文字を返します。 *position* が範囲の外にある場合、つまり、バッファの先頭より前や、末尾の後の場合、値は `nil` です。

以下の例において、バッファの最初の文字は ‘`@`’ になっています。

```
(char-to-string (char-after 1))
     ⇒ "@"
```

following-char — Function

この関数は、カレント・バッファ中のポイントに続く文字を返します。これは `(char-after (point))` と似ています。ポイントがバッファの最後の位置にある場合、 `following-char` は 0 を返します。

ポイントは文字と文字の間にあるのですが、端末のカーソルは普通ポイントの直後の文字にかぶさって見えます。したがって `following-char` が返す文字は、カーソル上の文字です。

次の例において、ポイントは ‘a’ と ‘c’ の間にあります。

```
---------- Buffer: foo ----------
Gentlemen may cry ``Pea⋆ce! Peace!,''
but there is no peace.
---------- Buffer: foo ----------
```

```
(char-to-string (preceding-char))
     ⇒ "a"
(char-to-string (following-char))
     ⇒ "c"
```

preceding-char Function

この関数は、カレント・バッファのポイントの前にある文字を返します。上記の例の following-char の下を見てください。ポイントがバッファの最初の位置にある場合、preceding-char は 0 を返します。

bobp Function

この関数は、ポイントがバッファの先頭にある場合 t を返します。ナローイングが用いられている場合、ここでいう先頭はそのアクセス可能部分の先頭を意味します。 27.1 節「ポイント」p.457 の point-min も参照してください。

eobp Function

この関数は、ポイントがバッファの最後にある場合 t を返します。ナローイングが用いられている場合、ここでいう先頭はそのアクセス可能部分の最後を意味します。 27.1 節「ポイント」p.457 を参照してくださいの point-max も参照してください。

bolp Function

この関数は、ポイントが行の先頭にある場合 t を返します。 27.2.4 節「テキスト行単位での移動」p.460 を参照してください。バッファ (あるいはアクセス可能部分) の先頭は、いつでも行頭のうちにはいります。

eolp Function

この関数は、ポイントが行の終りにある場合 t を返します。バッファ (あるいはアクセス可能部分) の終りは、いつでも行末とみなします。

29.2 バッファ内容の調査

この節では、 Lisp プログラムでバッファのテキストの一部を文字列へ変換するための、二つの関数について述べます。

buffer-substring *start end* Function

この関数は、カレント・バッファ中の位置 *start* と *end* で定義されたリージョンのテキストのコピーを文字列に変換して返します。引数がバッファのアクセス可能部分を指さない場合、 buffer-substring は args-out-of-range エラーを通知します。

start を *end* より前にする必要はなく、引数はどちらの位置でもかまいません。しかし普通は、小さい方が前です。

もし転記されるテキストにテキスト属性があるとき、転記した文字のテキスト属性もいっしょに転記されます。 29.18 節「テキスト属性」p.509 を参照してください。しかしなが

らバッファのオーバレイ (35.9 節「オーバレイ」p.609 参照) とその属性は、無視され転記されません。

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------
(buffer-substring 1 10)
⇒ "This is t"
(buffer-substring (point-max) 10)
⇒ "he contents of buffer foo
"
```

buffer-substring-no-properties *start end* Function

これは buffer-substring に似ていますが、テキスト属性の転記をしません。 29.18 節「テキスト属性」p.509 を参照してください。この関数を使い alist を参照して文字列をとってくる例を示します。

```
(setq flammable
      ;; 訳注: 原文は buffer-substring だった。
      (assoc (buffer-substring-no-properties start end)
             '(("wood" . t) ("paper" . t)
               ("steel" . nil) ("asbestos" . nil))))
```

もしこれを buffer-substring で書いても、必ずしも正しく動作しません。バッファから転記した文字列にたまたまテキスト属性があったときは、比較に失敗してしまいます。[1]

buffer-string Function

この関数は、カレント・バッファのアクセス可能部分の内容を文字列として返します。これは (point-min) と (point-max) の間の部分です (27.4 節「ナローイング」p.467 参照)。

```
---------- Buffer: foo ----------
This is the contents of buffer foo

---------- Buffer: foo ----------

(buffer-string)
     ⇒ "This is the contents of buffer foo
"
```

[1] 原著でこの関数の名前は buffer-substring-without-properties となっていましたが、実際には buffer-substring-no-properties という関数しかなく、原著が訂正されるとのことです。

29.3 テキストの比較

この関数は、文字列に変換することなく、バッファ内のある部分を比較します。

compare-buffer-substrings *buffer1 start1 end1 buffer2 start2 end2* Function

この関数は、文字列に変換することなく、同じあるいは異なるバッファ内のある部分を比較します。はじめの三つの引数でバッファと二つの位置を与えて、これで一つの部分を指定します。終りの三つの引数で同様に別の部分を指定します。*buffer1* や *buffer2*、あるいは両方に `nil` を指定すると、カレント・バッファになります。

はじめの部分が小さければ値は負になり、はじめの方が大きければ正、もし同じなら 0 になります。結果の絶対値は、比較部分で最初の異なる文字の添え字に 1 を加えたものになります。

`case-fold-search` が非 `nil` のとき、当関数は文字の比較で大文字小文字の違いを無視します。テキスト属性は常に無視します。

カレント・バッファに '`foobarbar haha!rara!`' というテキストがあったとします。そしてこの例で二つの部分を '`rbar `' と '`rara!`' します。最初の部分が次の部分よりも大きいので、値は 2 です。

```
(compare-buffer-substring nil 6 11 nil 16 21)
     ⇒ 2
```

29.4 テキストの挿入

『挿入 (insertion)』とはバッファに新しいテキストを加えることです。挿入するテキストはポイントに、つまりポイントの前の文字と後の文字の中間に、入ります。

挿入箇所より後の位置を指すマーカは挿入によって再配置しますので、まわりのテキストの場所からは止まっているようにみえます (第 28 章「マーカ」p.471 参照)。マーカが挿入箇所を指していたとき、普通はマーカの移動はおきないので、挿入したテキストの先頭を指します。しかしながら `insert-before-markers` のような特殊な関数では、挿入箇所のマーカを挿入したテキストの後に移動します。

挿入関数のうちあるものは挿入するテキストの前にポイントを残し、あるものは後に移動します。前者を『ポイントの後に (after point)』挿入するといい、後者を『ポイントの前に (before point)』挿入するといいます。

カレント・バッファが読出し専用のとき、挿入関数はエラーを通知します。

この関数群は文字列やバッファからテキスト文字を転記すると同時に、属性も転記します。挿入される文字は、転記されてくる文字とちょうど同じ属性をもちます。その一方、文字列やバッファの一部としてではなく文字を別々な引数として指定する場合、その文字はテキスト属性をまわりのテキストから継承します。

insert &rest *args* Function

この関数は、カレント・バッファのポイントを先に進めながら、ポイントのある位置にいくつかの文字列や文字 *args* を挿入する、つまり、ポイントの前にテキストを挿入します。 *args* 中に文字列でも文字でないものが含まれている場合、エラーになります。値は `nil` です。

insert-before-markers &rest *args* Function

この関数は、カレント・バッファのポイントを先に進めながら、ポイントのある位置にいくつかの文字列や文字 *args* を挿入します。 *args* 中に文字列でも文字でないものが含まれている場合、エラーになります。値は `nil` です。

当関数はほかの挿入関数と異なり、挿入箇所を指していたマーカの位置を挿入したテキストの後に再配置します。

insert-char *character count* &optional *inherit* Function

この関数は、 *count* 個の *character* をカレント・バッファのポイントの前に挿入します。引数 *count* は数値でなくてはいけません。また、 *character* は文字でなくてはいけません。値は `nil` です。

もし *inherit* が非 `nil` ならば、挿入される文字は粘着的テキスト属性を、挿入箇所の前後の 2 文字から継承します。 29.18.6 節「テキスト属性の粘着性」 p.516 を参照してください。

insert-buffer-substring *from-buffer-or-name* &optional *start end* Function

この関数は、バッファ *from-buffer-or-name* (あらかじめ存在しなくてはいけません) の一部分を、カレント・バッファのポイントの前に挿入します。挿入するテキストは *start* と *end* で定義されるリージョンです (省略するとバッファのアクセス可能な先頭と末尾になります)。当関数は `nil` を返します。

次の例では、バッファ 'bar' をカレント・バッファとしてこの形式を実行します。最初、バッファ 'bar' は空とします。

```
---------- Buffer: foo ----------
We hold these truths to be self-evident, that all
---------- Buffer: foo ----------

(insert-buffer-substring "foo" 1 20)
     ⇒ nil

---------- Buffer: bar ----------
We hold these truth⋆
---------- Buffer: bar ----------
```

挿入に加えてまわりのテキストからテキスト属性を継承するほかの挿入用の関数については、 29.18.6 節「テキスト属性の粘着性」 p.516 を参照してください。字下げ関数で挿入された空白もテキスト属性を継承します。

29.5 ユーザ・レベルの挿入コマンド

この節では、主にユーザが呼ぶことを想定しているものの、Lisp プログラムから呼んでも便利な、テキストを挿入する上位のコマンドを解説します。

insert-buffer *from-buffer-or-name* Command

このコマンドは、バッファ *from-buffer-or-name* (これは存在しなくてはいけません) の全内容をカレント・バッファのポイントの後に挿入します。挿入されたテキストの後にマークを置きます。値は `nil` です。

self-insert-command *count* Command

このコマンドは、最後にタイプされた文字を挿入します。*count* 回、ポイントの前で挿入を繰り返して `nil` を返します。ほとんどの印字文字がこのコマンドにバインドされています。`self-insert-command` は普段もっともよく呼ばれる Emacs の関数ですが、これをキーマップに入れるときをのぞけば、プログラムがこれを用いることはほとんどありません。

対話的に呼び出すと、*count* は数値前置引数になります。

`auto-fill-function` が非 `nil` なら、このコマンドはスペースあるいは改行が挿入されるたびに、`auto-fill-function` を呼び出します (29.13 節「自動的行詰め」p.499 参照)。

Abbrev モードが有効で、挿入した文字が語構成可能構文をもたないとき、当コマンドは省略形の展開を行ないます。(第 32 章「省略形とその展開」p.553 そして、31.2.1 節「構文クラスのテーブル」p.542 を参照してください)

入力した文字が閉じ括弧構文をもつとき、当関数は `blink-paren-function` を呼び出します (35.11 節「対応括弧の点滅表示」p.616 参照)。

newline &optional *number-of-newlines* Command

このコマンドは、改行をカレント・バッファのポイントの前に挿入します。*number-of-newlines* が与えられると、改行をその個数だけ挿入します。

現在の桁数が `fill-column` の値よりも大きく、*number-of-newlines* の値が `nil` のとき、この関数は、`auto-fill-function` を呼び出します。典型的な `auto-fill-function` の実行することとは、改行の挿入です。この場合最終的には二つ改行を別の場所に挿入することになります。一つはポイントで、もう一つは行のどこか前の場所です。`newline` は *number-of-newlines* が非 `nil` のとき、自動行詰めをしません。

このコマンドは左マージンが 0 でなければ、そこまで字下げを行ないます。29.12 節「行詰めのマージン」p.497 を参照してください。

返却値は `nil` です。対話的に呼ばれた場合、*number-of-newlines* は数値前置引数になります。

split-line Command

このコマンドは、現在の行をわけ、ポイントの後の (監訳者注: 空白でないところから行末までの) 部分を垂直に下に動かし、前にあった場所のすぐ下の行とします。下の行の先

頭に必要に応じて空白をあけるのは、 `indent-to` 関数で行ないます。ポイントの位置を返します。

プログラムがこの関数を使うことはめったにありません。

overwrite-mode Variable

この変数は、 Overwrite モードが有効かどうかを制御します。非 `nil` の値でモードが有効になります。どのような方法で変更しても自動的にバッファローカルになります。

29.6 テキストの消去

消去とはバッファのテキストの一部を kill リングに保存せずに、なくしてしまうことをいいます (29.8 節「kill リング」p.488 参照)。消去したテキストはヤンクできません。しかし、取り消し機構で元に戻すことはできます (29.9 節「取り消し」p.492 参照)。一部の消去の関数は、ある特別な場合に kill リングにテキストを保存することがあります。

消去の関数はすべて、カレント・バッファにたいし操作を行ない `nil` を返します。

erase-buffer Function

この関数は、カレント・バッファ中の全テキストを消去し、空にします。バッファが読出し専用の場合、 `buffer-read-only` エラーを通知します。そうでない場合、何もきかずにテキストを消去します。 `nil` を返します。

通常、バッファから多量のテキストが消去されると、「バッファが縮んだ」(“because it has shrunk”) ため、このバッファの自動セーブが禁止になります。しかしながら `erase-buffer` はそうしません。往々にして消去する前のテキストと後のテキストとはお互いに関係がないため、前のテキストの大きさとの比較をするべきではないからです。

delete-region *start end* Command

この関数は、カレント・バッファ中の *start* と *end* で定義されるリージョン内のテキストを消去します。値は `nil` です。

delete-char *count* &optional *killp* Command

このコマンドは、ポイントの直後 (*count* が負の場合は、直前) の *count* 文字を消去します。 *killp* が非 `nil` の場合、消去した文字を kill リングに保存します。

対話的に呼ばれた場合、 *count* は数値前置引数になり、 *killp* は未処理の前置引数になります。したがって前置引数が与えられるとテキストは kill リングに保存されます。前置引数が与えられない場合、 1 文字が消去され kill リングには保存されません。

返却値は常に `nil` です。

delete-backward-char *count* &optional *killp* Command

このコマンドは、ポイントの直前 (*count* が負の場合は、直後) の *count* 文字を消去します。 *killp* が非 `nil` の場合、消去した文字を kill リングに保存します。

対話的に呼ばれた場合、 *count* は数値前置引数になり、 *killp* は未処理の前置引数になります。したがって前置引数が与えられるとテキストは kill リングに保存されます。前置引数が与えられない場合、 1 文字が消去され kill リングには保存されません。

返却値は常に `nil` です。

backward-delete-char-untabify *count* &optional *killp* Command

このコマンドは、タブをスペースに変え、後ろの方向に *count* 文字を消去します。タブをスペースに変える際、ポイントを同じ桁に保持するよう十分な数のスペースをバッファに挿入します。 *killp* が非 `nil` の場合、このコマンドは消去した文字を kill リングに保存します。

タブからスペースへの変換は *count* が正の場合だけおこります。負の場合、ポイントの後のちょうど $-count$ 文字をを消去します。

対話的に呼ばれた場合、 *count* は数値前置引数になり、 *killp* は未処理の前置引数になります。したがって前置引数が与えられるとテキストは kill リングに保存されます。前置引数が与えられない場合、 1 文字が消去され kill リングには保存されません。

返却値は常に `nil` です。

29.7 ユーザ・レベルの消去コマンド

この節では、主にユーザが呼ぶことを想定しているものの、 Lisp プログラムから呼んでも便利な、テキストを消去する上位のコマンドを解説します。

delete-horizontal-space Command

この関数は、ポイントのまわりのスペースやタブをすべて消去します。 `nil` を返します。

次の例において、 `delete-horizontal-space` を各行に 1 回ずつ 4 回呼んでいます。(訳注: この関数を呼ぶ前に) ポイントは各行の 2 番目と 3 番目の文字の間に存在しています。

```
---------- Buffer: foo ----------
I ⋆thought
I ⋆     thought
We⋆ thought
Yo⋆u thought
---------- Buffer: foo ----------

(delete-horizontal-space)   ; 4 回。
     ⇒ nil

---------- Buffer: foo ----------
Ithought
Ithought
Wethought
You thought
---------- Buffer: foo ----------
```

delete-indentation &optional *join-following-p* Command

この関数は、ポイントの存在する行を前の行と結合し、場合によってスペースを一つ残してその結合部分にある空白を消去します。 *join-following-p* が非 `nil` の場合、 `delete-indentation` は、この行と後に続く行をつなげます。値は `nil` です。

行詰め接頭辞があり、つながる行がその接頭辞で始まるとき、 `delete-indentation` は、行をつなぐ前に行詰め接頭辞を消去します。 29.12 節「行詰めのマージン」 p.497 を参照してください。

次の例において、ポイントは ‘`events`’ で始まる行に位置しています。前の行の行末がスペースで終わっていても違いは生じません。

```
---------- Buffer: foo ----------
When in the course of human
⋆    events, it becomes necessary
---------- Buffer: foo ----------
(delete-indentation)
     ⇒ nil

---------- Buffer: foo ----------
When in the course of human⋆ events, it becomes necessary
---------- Buffer: foo ----------
```

行をつないだ後つなぎ目にスペースを残すかどうかは、関数 `fixup-whitspace` で決めます。

fixup-whitespace Function

この関数は、文脈にしたがって、ポイントの前後にある空白をなくしたり、一つのスペースに置き換えたりします。 `nil` を返します。

行頭や行末の場合、スペースを残すことはありません。閉じ括弧構文をもつ文字の前の場合、開き括弧や表現接頭辞の構文をもつ文字の後の場合も、スペースは残しません。上記の場合をのぞいてスペースを一つ残します。 31.2.1 節「構文クラスのテーブル」 p.542 を参照してください。

以下の例において、 `fixup-whitespace` を最初に呼ぶ際、ポイントは 1 番目の行の ‘`spaces`’ という語の前に位置しています。 2 回目に呼ぶ時、ポイントは ‘`(`’ の直後に位置しています。

```
---------- Buffer: foo ----------
This has too many     ⋆spaces
This has too many spaces at the start of (⋆   this list)
---------- Buffer: foo ----------
(fixup-whitespace)
     ⇒ nil
(fixup-whitespace)
     ⇒ nil
---------- Buffer: foo ----------
This has too many spaces
This has too many spaces at the start of (this list)
---------- Buffer: foo ----------
```

just-one-space Command

このコマンドはポイントのまわりのすべてのスペースとタブを、一つのスペースに置き換えます。`nil` を返します。

delete-blank-lines Command

この関数は、ポイントを囲む空白行を消去します。ポイントが 1 行以上の空白行をその前か後ろにもつ空白行に位置する場合、1 行を残しあとはみな消去します。ポイントが 1 行のみの空白行に存在する場合、その空白行を消去します。ポイントが空白行でない行に位置する場合、その後に続く空白行をすべて消去します。

空白行とはタブとスペース以外のものを含まない行のことです。

`delete-blank-lines` は `nil` を返します。

29.8 kill リング

『kill』の関数は消去の関数のようにテキストを消去しますが、ユーザが『ヤンク (yanking)』して再び挿入できるようにテキストを記録しておきます。ほとんどの関数名には '`kill-`' がつきます。一方、'`delete-`' で始まる名前の関数は、ヤンクのための保存をしません (取り消しはできます)。これらは「消去」の関数です。

kill のコマンドのほとんどは主に対話的な目的のためで、ここでは解説しません。ここで解説するのは、そのようなコマンドを作るための関数群です。テキストを kill するコマンドを書こうと思ったらこの関数群を使ってください。Lisp 関数の中で内部的な目的のためにテキストを消去するときは、kill リングの内容を壊さないようにするため、通常は消去のほうの関数を使う必要があります。29.6 節「テキストの消去」p.485 を参照してください。

ヤンクのために kill したテキストは『kill リング (kill ring)』に保存します。これは、一番最後の kill だけでなく、最後の何回かの kill を保存したリストです。これを "ring"(リング) というのは、ヤンクがリングの要素を周期的な順序で扱うためです。リストは変数 `kill-ring` に保管してあり、リストを扱うごく普通の関数でも操作できますが、この節で述べるようにそれをリングとして使う専門の関数もあります。

kill した (殺した) ものだからといっても破壊をしないので、"kill" という語が使われているのは残念だという人がいます。死は永久的なもので、殺された ("killed") ものは二度と戻ってこないという点では、日常生活とは対照的です。そのためほかのたとえが提案されています。たとえば、"cut ring" という用語では、それ以前にハサミと糊で手書きの切り (cut up) 貼りをしたことのある人には、意味があるでしょう。でも今となってはこの用語を変更するのは困難です。

29.8.1 kill リングの概念

kill リングはリストに文字列として、kill されたテキストを最近のものを最初にして記録します。たとえば、短い kill リングは以下のようになっています。

```
("some text" "a different piece of text" "even older text")
```

リストの長さが `kill-ring-max` になったら、新しい項目を追加すると自動的に最後の項目を消去します。

kill のコマンドがほかのコマンドにはさまれると、 kill のコマンドは新しい項目をつくります。連続した複数の kill のコマンドは kill リングで単一の項目になり、ヤンクのときの一つの単位になります。 2 回目以降の連続した kill のコマンドは、最初のそれが作った項目にテキストを追加します。

kill リング中の一つの項目は、ヤンクでリングの「前」へと指示されます。あるヤンクのコマンドは異なる要素を「前」へと指示することで、リングを「まわし」ます。しかし、この仮想的な回転は実際のリストの変更はしません。最近の項目が常にリストの最初にきます。

29.8.2 kill の関数

`kill-region` はテキストを kill するときにいつも使うサブルーチンです。このコマンドを呼ぶ関数はすべて「kill のコマンド」(で多分、名前には ‘`kill`’ がつくべき) です。 `kill-region` は新しく kill したテキストを、 kill リングの先頭に置くか、一番最近の要素につけたすかします。これは `last-command` 変数で前のコマンドが kill のコマンドだったかを判別し、もしそうだった場合、 kill したテキストを最も新しい項目に追加します。

kill-region *start end* Command

この関数は、 *start* と *end* で定義されるリージョン内のテキストを kill します。このテキストはテキスト属性といっしょに消去されますが、 kill リングに保存されます。値は常に `nil` です。

対話的に呼ばれた場合、 *start* と *end* はポイントとマークになります。

バッファが読出し専用のとき、 `kill-region` は同じように kill リングを変更し、バッファを修正せずにエラーを通知します。読出し専用のバッファから kill リングへテキストを写すとき、ユーザがどんな kill のコマンドを使うこともできて便利です。

copy-region-as-kill *start end* Command

この関数は、 *start* と *end* で定義されたリージョンを (テキスト属性も含めて)kill リング上に保存します。そのテキストをバッファから消去することはしません。 `nil` を返します。対話的に呼ばれると *start* と *end* はポイントとマークになります。一瞬カーソルを動かしたりエコー領域にメッセージを表示したりして、写したテキストを示します。

このコマンドは、 `this-command` を `kill-region` にすることはありませんので、 kill のコマンドが続いても同じ kill リングの項目に追加されません。

Emacs 18 をサポートする目的がないのなら、 Lisp プログラムでは `copy-region-as-kill` を呼ばないでください。 Emacs 19 では、 `kill-new` や `kill-append` を使うほうがよいでしょう。 29.8.4 節「下位の kill リング」 p.490 を参照してください。

29.8.3 ヤンクの関数

『ヤンク (yanking)』とは、前に kill したテキストを kill リングから再び挿入することです。テキスト属性も転記されます。

yank &optional *arg* Command

この関数は、 kill リングの最初の項目にあるテキストをポイントの前に挿入します。テキストの前にマークを置き後にポイントを置きます。

(対話的に C-u を押下し数字を押下せずに呼んだときのように) *arg* がリストの場合、 yank は上記のようにヤンクしますが、挿入したテキストの前にポイントを位置づけテキストの終りにマークを位置づけます。

arg が数字の場合、 yank は kill されたテキストの *arg* 番目に新しいもの、つまり、 kill リングのリストの *arg* 番目の要素を挿入します。

yank がその内容を変えたりすることはありません。 (訳注: 引数がなければ)kill リングを回すこともありません。 nil を返します。

yank-pop *arg* Command

この関数は、 kill リングからヤンクされたばかりの項目を kill リングの別の項目で置き換えます。

このコマンドは yank か yank-pop の直後でしか許されません。このような場合、リージョンはその前の yank で挿入されたばかりのテキストをもちます。 yank-pop はそのテキストを消去し、その場所に、 kill された別のテキストを挿入します。消去したテキストはすでに kill リング上のどこかにありますから、 kill リングへの追加は行ないません。

arg が nil の場合、置き換えるテキストは kill リングの一つ手前の要素です。 *arg* が数値の場合、置き換えるのは *arg* 番目に前の kill になります。 *arg* が負の場合、置き換えるのはより最近の kill になります (訳注: たとえば、引数が 2 なら二つ手前の kill、 -2 なら二つ後の kill ということです)。

kill リング中の kill のシーケンスは、前に戻るようになっているため、最も古いものの後に最も新しいものがきて、最も新しいものの前に最も古いものがきます。

値は常に nil です。

29.8.4 下位の kill リング

以下の関数と変数は kill リングへの下位のアクセスのためのものですが、 Lisp プログラムから使っても便利です。 X Window のセレクションを使った対話のめんどうもみます。これらは Emacs バージョン 18 には存在しません。

current-kill *n* &optional *do-not-move* Function

関数 current-kill は、 kill リングの先頭を指すヤンク・ポインタをリング中で *n* 回 (最近の kill から古い kill へ) まわして、そこにきたリングのテキストを返します。

省略可能な第 2 引数 *do-not-move* が非 nil のとき、 current-kill は、ヤンク・ポインタを変えません。現在のヤンク・ポインタから数えて *n* 番目の kill を返すだけです。

n が、 (最後の kill を示す)0 になっているとき、 current-kill は、 kill リングを見る前に interprogram-paste-function (下記参照) の値を呼びます。

kill-new *string* Function

この関数は、 kill リングの先頭にテキスト *string* を、新しい項目として入れます。必要に応じて一番古い項目をすてます。また、 `interprogram-cut-function`(下記参照) の値の起動も行ないます。

kill-append *string before-p* Function

この関数は、テキスト *string* を kill リングの最初の項目として追加します。普通は *string* は項目の末尾につきますが、 *before-p* が非 `nil` のときは先頭につきます。また、 `interprogram-cut-function`(下記参照) の値の起動も行ないます。

interprogram-paste-function Variable

この変数は、ウィンドウ・システムを使っているときに、 kill したテキストをほかのプログラムに転送する方法を提供します。値は `nil` か引数のない関数でなければなりません。

値が関数のとき、 `current-kill` が「最新の kill」を得るために、これを呼び出します。関数が非 `nil` の値を返すと、その値が「最新の kill」として利用されます。 `nil` を返すと、 `kill-ring` の最初の要素が利用されます。

このフックの通常の使用例は、 X サーバーの第 1 セレクションがほかのクライアントに所属しているときでも、このセレクションを最新の kill として取り込むことです。 26.18 節「X セレクション」 p.452 を参照してください。

interprogram-cut-function Variable

この変数は、ウィンドウ・システムを使っているときに、 kill したテキストをほかのプログラムに通信する方法を提供します。値は `nil` か引数が一つの関数でなければなりません。

値が関数のとき、 `kill-new` と `kill-append` は、 kill リングの新しい 1 番目の要素を引数に、これを呼び出します。

このフックの通常の使用例は、 X サーバーの第 1 セレクションに新しく kill したテキストとして設定することです。

29.8.5 kill リングの内部

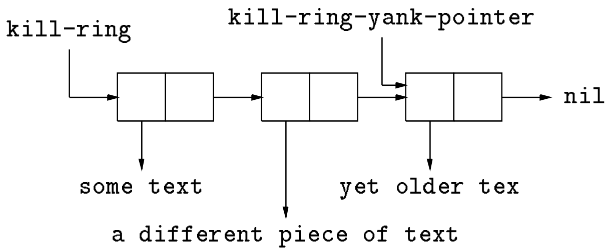
変数 `kill-ring` は、 kill リングの内容を文字列のリストとして保持します。最も最近の kill は常にリストの先頭にきます。

`kill-ring-yank-pointer` 変数は、 CAR が次にヤンクするテキストであるような、 kill リングのリストの中の連鎖を指します。これで輪の「先頭」を特定 (identify) します。 `kill-ring-yank-pointer` を違う連鎖に動かすことを、『kill リングの回転 (rotating the kill ring)』といいます。 kill リングを “ring”(リング) というのは、ヤンク・ポインタを動かす関数がリストの最後から最初へ、あるいはその逆におりかえす (wrap around) ためです。 kill リングの回転は論理的なことなので、 `kill-ring` の値を変えません。

`kill-ring` と `kill-ring-yank-pointer` は Lisp の変数で、普通はリストを値にもちます。 `kill-ring-yank-pointer` の “pointer” という語は、変数の目的が次のヤンクコマンドで使うリスト中の一つの要素を特定することを示しています。

kill-ring-yank-pointer の値は常に、 kill リングのリスト中の一つの連鎖に eq です。それが特定する要素とは、その連鎖の CAR のことです。 kill リングを変える kill のコマンドもこの変数を kill-ring の値へ変えます。これで新しく kill したテキストが前になるように、輪を回したことになります。

次の図は、変数 kill-ring-yank-pointer が kill リング ("some text" "a different piece of text" "yet older text") の第 2 項目を指している様子を示します。



この状態は C-y(yank) の直後の M-y(yank-pop) で (監訳者注: あるいは C-u 2 C-y で) おきるはずです。

kill-ring Variable

この変数は、 kill されたテキストの並びを、最近に kill したものを前にして、保持します。

kill-ring-yank-pointer Variable

この変数の値は、 kill リングのどの要素がヤンクの輪の「先頭」なのかを示します。厳密にいうと、この値は kill-ring の後部 (tail) で、 CAR は yank がヤンクすべき kill 文字列を指すものです。

kill-ring-max User Option

この変数の値は、末尾の (最も古い) 要素が捨てられ始める前に kill リングの長くなりうる最大の長さです。 kill-ring-max のデフォルト値は 30 です。

29.9 取り消し

ほとんどのバッファは取り消しができるようにするために、バッファのテキストの全変更を記録した『undo リスト (undo list)』があります (これのないバッファはたいてい、特別な目的のためのバッファで、 Emacs は取り消しが便利でないと仮定します)。バッファのテキストを修正する全プリミティブ関数は、自動的に変数 buffer-undo-list という undo リストの前に、要素を追加します。

buffer-undo-list Variable

この変数の値はカレント・バッファの undo リストです。 t という値は、取り消し情報の記録を禁止します。

以下は undo リストの要素の種類です。

整数
: この種の要素はポイントの以前の値を示します。普通のカーソル移動はどんな種類の取り消し記録もつくりませんが、消去のコマンドはこの項目を使って、コマンドの前にポイントがどこにあったかを記録します。

(*beg* . *end*)
: この種の要素は挿入されたテキストを消去する方法を示します。挿入によりテキストはバッファで *beg*–*end* の範囲を占めます。

(*text* . *position*)
: この種の要素は消去されたテキストを挿入する方法を示します。消去されたテキストは文字列 *text* です。再挿入すべき場所は (`abs` *position*) です。

(`t` *high* . *low*)
: この種の要素は無修正のバッファが修正されたことを示します。要素の *high* と *low* は二つの整数で、以前にセーブした時刻や読み込んだときの (訳注: ファイルの修正) 時刻のように、対応づけられたファイルの修正時刻を、おのおの 16 ビットずつ記録します。 `primitive-undo` はこの値を使って、バッファが再び無修正の状態になったかどうかを判別します。ファイルの修正時刻がこれらの数と一致するとき、無修正という印をつけます。

(`nil` *property value beg* . *end*)
: この種の要素はテキストの属性が変更されたことを示します。これを取り消す方法は以下のとおりです。

```
(put-text-property beg end property value)
```

position
: この要素は、前にポイントがどこにあったかを示します。この要素の取り消しでポイントを *position* にします。通常、消去で再挿入の要素といっしょにこの要素が作られます。

`nil`
: この要素は境界です。二つの境界にはさまれた要素を『変更群 (change group)』といいます。普通、各変更群は一つのキーボード・コマンドに対応します。取り消しのコマンドは普通群全体を一つの単位として取り消します。

undo-boundary Function

この関数は、境界の要素を undo リストに入れます。取り消しのコマンドはこのような境界で止まり、連続した取り消しのコマンドはどんどん前の境界へ取り消していきます。この関数は、 `nil` を返します。

エディタのコマンド・ループは、キー・シーケンスの実行前に自動的に取り消しの境界をつくります。したがって各取り消しは通常一つのコマンドの効果を取り消します。自己挿入文字の入力は例外です。コマンド・ループはそのような最初の文字で境界を作り、以降連続する 19 個の自己挿入文字の入力では境界を作りません。 20 個目で作り、自己挿入文字が続く間同様に繰り返します。

直前の取り消し可能な変更が別なバッファでなされた場合にはいつも、すべてのバッファ修正で境界ができます。こうして、複数のバッファを修正するコマンドは、変更した各バッファに境界をつくります。

この関数を陽に呼ぶことは、コマンドの効果を複数の単位にわけるとき便利です。たとえば、 `query-replace` は、各置換の後で `undo-boundary` を呼び出すので、ユーザがそれぞれ一つずつの取り消しを行なうことができます。

primitive-undo *count list* Function

これは、 undo リストの要素で取り消すための基本的な関数です。 *list* の最初から *count* 個の要素で取り消しをします。 *list* の残りを返します。 Lisp でこの関数を書くことも可能ですが、 C にしておくと便利です。

`primitive-undo` はバッファを変更したら、バッファの undo リストに要素を追加してしまいます。取り消しのコマンドは、一連の取り消し操作のはじめで undo リストの値を保存し、混乱を防いでいます。その後の取り消しの操作では、保管した値を使い、これを更新します。取り消しで追加された新しい要素はこの保管した方の値に追加されませんから、続けて取り消しを行なっても影響はありません。

29.10 undo リストの保守

この節では、与えられたバッファの取り消し情報を禁止したり、それを解除したりする方法を述べます。また、 undo リストが大きくなりすぎないように、自動的に切り詰める方法についても説明します。

新規に生成したバッファの取り消し情報の記録は普通、最初から有効です。しかし、名前がスペース文字で始まるバッファははじめ、取り消しの記録を禁止してあります。以下の二つのコマンドを使ったり、自分で `buffer-undo-list` を設定したりして、陽に取り消しの記録を禁止したり、解除したりすることができます。

buffer-enable-undo &optional *buffer-or-name* Command

このコマンドは、以降の変更が取り消せるように、バッファ *buffer-or-name* にたいする取り消し情報を保持していくようにします。引数が与えられない場合、カレント・バッファが対象です。バッファがすでに取り消し情報を保持している場合、何も変更しません。 `nil` を返します。

対話的に呼ばれた場合、 *buffer-or-name* はカレント・バッファになります。ほかのバッファは指定できません。

buffer-disable-undo &optional *buffer* Function
buffer-flush-undo &optional *buffer* Function

この関数は、バッファ *buffer* の undo リストを捨て、取り消し情報の記録を禁止します。その結果として、以前や以後の変更の取り消しが不可能になります。 *buffer* が undo リストを保持しないようにしてある場合、当関数は効果がありません。

`nil` を返します。対話的に呼ぶことはできません。

名前の `buffer-flush-undo` はまだ obsolete ではありませんが、 Emacs バージョン 19 では、新しく推奨する名前 `buffer-disable-undo` があります。

編集を続けると、 undo リストはどんどん長くなります。有効なメモリ空間を使い果たすのを防ぐため、塵集めの際に、設定のできるある大きさの上限まで切り詰めます (この目的のため、 undo リストの「大きさ」を、リストを作っているコンス・セルと消去したテキストの文字列を合せて測ります)。二つの変数 `undo-limit` と `undo-strong-limit` で、許容できる大きさの範囲を制御します。

undo-limit Variable

これは undo リストとして許容できる弱い上限です。この大きさを越える変更群は最後のものを保存します。

undo-strong-limit Variable

これは undo リストとして許容できる上限です。この大きさを越える変更群は、 (それ以前のすべての変更群といっしょに) すべて捨てられてしまいます。一つだけ例外があります。一番最後の変更群はどんなに大きくても保存されます。

29.11 行詰め

『行詰め (filling)』とは、 (行の変わるところを動かし) 行の長さを調節し、指定した最大幅 (でもそれを越えない) くらいにすることです。そして行を『揃える (justified)』、つまり、スペースを挿入し右や左のマージンがちょうど並ぶようにすることもできます。この幅は変数 `fill-column` で制御します。読みやすさのために、行は 70 桁くらいよりは長くないのがよいでしょう。

Auto Fill モード (29.13 節「自動的行詰め」 p.499 参照) を使い、既存のテキストの変更では適切な行詰めをしないまま、テキストを入力したときは自動的にテキストの行詰めをすることができます。この場合、後で陽に既存のテキストの行詰めを行なわなければなりません。

この節のほとんどのコマンドは、意味のない値を返します。行詰めを行なうすべての関数は、現在の左マージン、現在の右マージン、現在の揃え方のスタイルの影響を受けます (29.12 節「行詰めのマージン」 p.497 参照)。現在の揃え方のスタイルが `none` のとき、行詰めの関数は何もしません。

行詰め関数のうちのいくつかには *justify* 引数があります。もしこれが非 `nil` ならば、何らかの揃え方を要求していることになります。とりうるのは `left`(左揃え)、 `right`(右揃え)、 `full`(両揃え)、 `center`(中揃え) で、それぞれ指定した揃え方のスタイルになります。もしこれが `t` の場合は、テキストのその部分で現在の揃え方のスタイルを用いることを意味します (以下の `current-justification` を参照)。

行詰めの関数を対話的に呼び出すとき、前置引数を使って *justify* を値 `full` にすることができます。

fill-paragraph *justify* Command

このコマンドは、ポイントのあるまたはポイントの後の段落の行詰めを行ないます。もし *justify* が非 `nil` なら、各行の右揃えも行ないます。段落の境を見つけるのには、通常の

段落移動コマンドを使います。 *The Emacs Manual* の “Paragraphs” 節を参照してください。

fill-region *start end* &optional *justify* Command

このコマンドは *start* から *end* までのリージョンにある段落の、行詰めを行ないます。*justify* が非 `nil` なら右揃えも行ないます。

変数 `paragraph-separate` は、段落と段落をどうわけるかを制御します。 30.8 節「編集用の標準正規表現」p.539 を参照してください。

fill-individual-paragraphs *start end* &optional *justify mail-flag* Command

このコマンドは、個別の行詰め接頭辞を使って、リージョン内の段落の行詰めをします。したがって、段落の行がスペースで字下げしてあるとき、行詰めを行なった段落は同じように字下げをしたままになります。

最初の二つの引数 *start* と *end* は、行詰めを行なうリージョンの始めと終りです。 3 番目と 4 番目の引数 *justify* と *mail-flag* は、省略可能です。もし *justify* が非 `nil` なら、行詰めと同時に右揃えも行ないます。もし *mail-flag* が非 `nil` なら、この関数が扱っているのはメールのメッセージと仮定して、ヘッダーの行詰めを行ないません。

通常、 `fill-individual-paragraphs` は、字下げの変化を新しい段落の始まりとみなします。もし `fill-individual-varying-indent` が非 `nil` なら、分割行だけが段落を分割します。このとき、字下げをしてある段落で、最初の行にさらに字下げのしてあるものを処理することができます。

fill-individual-varying-indent User Option

この変数は、上で述べたように、 `fill-individual-paragraphs` の動作を変えます。

fill-region-as-paragraph *start end* &optional *justify* Command

このコマンドは、リージョンのテキストを一つの段落として行詰めを行ないます。リージョンにいくつも段落があるときは、段落の間の空行はなくなります。 *justify* が非 `nil` のとき、この関数は、行詰めと同時に右揃えを行ないます。

対話的に呼び出したとき、前置引数で右揃えを行ないます。

デフォルトで有効になっている Adaptive Fill モードでは、行詰め接頭辞のないときに、字下げをした段落での `fill-region-as-paragraph` を実行すると、段落の 2 行目の字下げを行詰め接頭辞にします。

justify-current-line *how eop nosqueeze* Command

このコマンドは現在の行の語間にスペースを挿入し、行末がちょうど `fill-column` で終わるようにします。 `nil` を返します。

もし引数 *how* が非 `nil` のとき、揃え方のスタイルを陽に指定します。とりうるのは `left`、 `right`、 `full`、 `center`、 `none` です。これがもし `t` なら指定した揃え方のスタイルにしたがいます (以下の `current-justification` を参照)。 `nil` は両揃えという意味になります。

もし *eop* が非 `nil` ならば、 `current-justification` が両揃えを指定しているとき、左揃えを行なうことを意味します。これは段落の最後の行に使用します。段落全体が両揃えであっても最後の行はそう揃えるべきではありません。

nosqueeze が非 `nil` のときは、中間の空白をいじらようにします。

default-justification User Option

この変数の値は、テキストを揃えるときのスタイルがテキスト属性で指定されていないときのスタイルを指定します。とりうる値は `left`、 `right`、 `full`、 `center`、 `none` です。デフォルト値は `left` です。

current-justification Function

この関数は、ポイントのまわりのテキストの行詰めを行なうときの、適正な揃えのスタイルを返します。

fill-paragraph-function Variable

この変数は、主モードにたいして段落の行詰めを行なう際の別法を与えます。もし値が非 `nil` のとき、 `fill-paragraph` はこの関数に実際の処理をさせます。この関数が非 `nil` の値を返したとき、 `fill-paragraph` は処理が終了したとみなし、ただちにその値を返します。

この機能の通常の利用法は、プログラミング言語のモードでコメントの行詰めを行なうことです。通常どおりに段落の行詰めをしなければならないときは、以下のようになります。

```
(let ((fill-paragraph-function nil))
  (fill-paragraph arg))
```

use-hard-newlines Variable

この変数が非 `nil` のとき、行詰め関数は `hard` テキスト属性のある改行を消去しません。「強い改行」は段落のセパレータとして作用します。

29.12 行詰めのマージン

fill-prefix User Option

この変数は、通常のテキストの始めにきて、行詰めを行なうときには無視されるテキスト文字列です。行詰め接頭辞で始まっていないすべての行は、段落の始まりとみなされます。行詰め接頭辞の後に空白のくる行もそうなります。行詰め接頭辞で始まり、次に空白のこない行は、行詰めを施される通常の行です。行詰めを行なった行は行詰め接頭辞で始めます。

左マージンがあるときは、空白の後に行詰め接頭辞がきます。

fill-column User Option

このバッファローカル変数は、行詰めをしたときの最大幅を指定します。これは、桁の数として解釈される整数でなくてはいけません。 Auto Fill モード (29.13 節「自動的行詰

め」p.499 参照) の場合を含め、すべての行詰め、右揃え、中寄せのコマンドは、この変数の影響を受けます。

他人に読んでもらうために実際にテキストを書いているとき、 `fill-column` を 70 以上に設定するべきではないでしょう。さもないと長すぎて他人に心地好く読んでもらえませんし、テキストが不格好です。

default-fill-column Variable

この変数の値は、 `fill-column` の値を個別に設定していないバッファでのデフォルト値です。これは `(default-value 'fill-column)` と同じです。

`default-fill-column` のデフォルト値は、 70 です。

set-left-margin *from to margin* Command

これは、 *from* から *to* までのテキストの `left-margin` 属性を *margin* の値に設定します。もし Auto Fill モードが有効ならば、このコマンドは、リージョンが新しいマージンにあうよう行詰めをやり直します。

set-right-margin *from to margin* Command

これは、 *from* から *to* までのテキストの `right-margin` 属性を *margin* の値に設定します。もし Auto Fill モードが有効ならば、このコマンドは、リージョンが新しいマージンにあうよう行詰めをやり直します。

current-left-margin Function

この関数は、ポイントのまわりで行詰めを行なうときに使用する適正な行詰め桁の値を返します。値は現在の行の先頭文字の `left-margin` 属性 (なければ 0) と、変数 `left-margin` の値の和です。

current-fill-column Function

この関数は、ポイントのまわりで行詰めを行なうときに使用する適正な行詰め桁の値を返します。値は `fill-column` 変数の値から、ポイントの後ろの `right-margin` 属性の値を引いたものです。

move-to-left-margin &optional *n force* Command

この関数は、現在の行の左マージンまでポイントを動かします。動かす桁は関数 `current-left-margin` を呼び出して決定します。もし引数 *n* が非 `nil` ならば、 `move-to-left-margin` は、最初 $n-1$ 行順方向に移動します。

もし *force* が非 `nil` ならば、左マージンの値と一致しない場合、行の字下げを修正することを意味します。

delete-to-left-margin *from to* Function

この関数は、 *from* から *to* までのテキストの左マージンの字下げを消去します。消去する字下げの数は `current-left-margin` を呼び出して決定します。この関数が空白でないものを消去することはありません。

indent-to-left-margin Function

これは Fundamental モード、 Text モードなどにおいて用いられるデフォルトの `indent-line-function` です。現在の行の行頭で、字下げが変数 `left-margin` で指定した値になるよう調節します。それには空白を挿入したり、消去したりします。

left-margin Variable

この変数は、基本の左マージン桁を指定します。 Fundamental モードで LFD は、この桁まで字下げを行ないます。この変数は、どのような方法で設定しても、バッファローカルになります。

29.13 自動的行詰め

Auto Fill モードは、テキストが入力されたとき自動的に行の行詰めを行なう副モードです。この節では Auto Fill が使うフックについて述べます。既存のテキストの行詰めと右揃えを陽に行なう関数の説明は、 29.11 節「行詰め」p.495 を参照してください。

Auto Fill モードは、テキストの一部の行詰めを行なうときの、マージンや右揃えのスタイルを変更する関数を、有効にするようにもします。 29.12 節「行詰めのマージン」p.497 を参照してください。

auto-fill-function Variable

この変数の値は、スペースや改行を自己挿入した後に呼び出される、(引数をもたない) 関数でなくてはいけません。これは `nil` であってもかまいません。この場合何も行ないません。

Auto Fill モードのとき、 `auto-fill-function` の値は `do-auto-fill` です。これは行を分割する通常処理の実装が目的の関数です。

> 昔のバージョンの Emacs では、この変数を `auto-fill-hook` と呼んでいましたが、フックを呼ぶ際の標準的な方法で呼ばれないため、バージョン 19 では名前が `auto-fill-function` に変更されました。

29.14 整列

この節で述べる整列の関数はみな、バッファのテキストを並べ換えるものです。この点で、リストの要素の順序を並べ換える関数 `sort` と対照的です (5.6.3 節「リストを再構成する関数」p.77 参照)。これらの関数は特に意味のない値を返します。

sort-subr *reverse nextrecfun endrecfun* &optional *startkeyfun endkeyfun* Function

この関数は、バッファをレコードに分割してそれの整列を行なう、一般的なテキストの整列ルーチンです。この節のほとんどのコマンドは、この関数を呼び出します。

`sort-subr` の働きを理解するために、バッファのアクセス可能部を、『整列レコード (sort record)』という互いに素な部分に分割して考えます。ここで、レコードは連続しているかもしれないしそうでないかもしれませんが、重なってはいません。各整列レコー

ドのある部分 (または全部) は、整列キーとして指示されます。整列では、この整列キーの順序にしたがってレコードを並べ換えます。

普通、レコードは整列キーの昇順に並べ換えられます。 `sort-subr` 関数の第 1 引数 *reverse* が非 `nil` なら、整列レコードは整列キーの逆順で並べ換えられます。

`sort-subr` への次の四つの引数は、整列レコードをとばしてポイントを動かすのに呼び出す関数です。これは `sort-subr` から何回も何回も呼び出されます。

1. *nextrecfun* は、ポイントをレコードの終りにして呼び出されます。この関数は、次のレコードの始めにポイントを動かします。最初のレコードは、 `sort-subr` が呼ばれたときのポイントの位置から始まる、と仮定します。したがって、普通 `sort-subr` を呼ぶ前には、ポイントをバッファの先頭に動かしておくべきでしょう。

   整列レコードがもうない場合、この関数は、ポイントをバッファの最後に動かします。

2. *endrecfun* は、ポイントをレコードの中に置いて呼び出されます。これはレコードの終りにポイントを動かします。

3. *startkeyfun* は、ポイントをレコードの始めから整列キーの始めに動かすときに呼び出されます。この引数は省略可能です。省いたときには、レコード全体を整列キーにします。与えた場合、関数は整列キーとして使われる非 `nil` の値を返すか、整列キーがバッファのポイントで始まることを示すために、 `nil` を返すかしなければなりません。後者のとき、整列キーの最後を見つけるため *endkeyfun* が呼び出されます。

4. *endkeyfun* は、ポイントを整列キーの最初から整列キーの最後へ動かすのに呼び出されます。この引数は省略可能です。 *startkeyfun* が `nil` を返し、この引数が略された (または `nil` だった) 場合、レコードの最後までを整列キーとします。 *startkeyfun* が非 `nil` の値を返したとき、 *endkeyfun* は不要です。

`sort-subr` の例として、 `sort-lines` の関数の完全な定義を次に示します。

```
;; 説明文字列の最初の 2 行は、
;; ユーザから見たとき本当は 1 行です。
(defun sort-lines (reverse beg end)
  "リージョン中の行を英文字順に整列する。
プログラムから呼び出したとき、引数を三つとります。
REVERSE (非 nil は逆順),
と BEG と END(整列するリージョン)。 "
  (interactive "P\nr")
  (save-restriction
    (narrow-to-region beg end)
    (goto-char (point-min))
    (sort-subr reverse
               'forward-line
               'end-of-line)))
```

ここで forward-line は、ポイントを次のレコードの始めに動かし、 end-of-line はレコードの終りに動かします。レコード全体が整列キーですから、引数 *startkeyfun* と *endkeyfun* を与えません。

sort-paragraphs 関数も同様ですが、 sort-subr の呼出しは以下のようになります。

```
(sort-subr reverse
           (function
            (lambda ()
              (skip-chars-forward "\n \t\f")))
           'forward-paragraph)
```

sort-regexp-fields *reverse record-regexp key-regexp start end* Command

この関数は、 *record-regexp* や *key-regexp* で指定されたとおりに、 *start* と *end* の間のリージョンを、英文字順に整列します。 *reverse* が負の整数の場合、整列は逆順に行ないます。

英文字順の整列とは、二つの整列キーの最初の文字を比較し、そして次の文字を比較し、以下同様に続けていくことです。最初に不一致のあった場所で小さい文字の方が小さい整列キーとなります。個々の文字は数値で比較されます。 Emacs は ASCII 文字集合を使いますから、この集合内の順序が英文字順を決定します。

record-regexp 引数は、バッファを整列レコードに分割する方法を指定します。各レコードの終りからこの正規表現で検索を行ない、それと一致したのが次のレコードになります。たとえば、正規表現が '^.+$' の場合、改行以外の文字が少なくとも一つある行と一致するので、このような行を整列レコードにすることになります。正規表現の構文と意味については 30.2 節「正規表現」 p.525 を参照してください。

key-regexp 引数の値は、各レコードの中のどの部分を整列キーにするかを指定します。 *key-regexp* はレコード全体に照合させても、一部分だけでもかまいません。後者の場合、残りの部分はレコードの整列順に関係ありませんが、レコードが新しい場所に移動するときはいっしょになります。

key-regexp 引数は、 *record-regexp* の部分式と一致するテキストも参照できますし、独自の正規表現を指定することもできます。

もし *key-regexp* が

'*digit*' の場合、 *record-regexp* の *digit* 番目の '\\(...\\)' 括弧でくくったグループに、一致するテキストが、整列キーになります。

'\\&' の場合、レコード全部が整列キーになります。

正規表現 の場合、 sort-regexp-fields はレコード中からこの正規表現の一致を検索します。見つかればそれが整列キーになります。あるレコード中で *key-regexp* に一致するものが見つからない場合、そのレコードは無視、つまり、バッファ内の位置は変わりません (まわりにあるほかのレコードは動きます)。

たとえば、リージョン中の各行を文字 ‘f’ で始まる最初の語で整列するためには、 *record-regexp* は ‘^.*$’ に、 *key-regexp* は ‘\<f\w*\>’ とします。したがって最終的な式は以下のようになります。

```
(sort-regexp-fields nil "^.*$" "\\<f\\w*\\>"
                    (region-beginning)
                    (region-end))
```

`sort-regexp-fields` を対話的に呼んだ場合、 *record-regexp* と *key-regexp* をミニバッファから求めるために、プロンプトを出します。

sort-lines *reverse start end* Command

このコマンドは、 *start* と *end* の間のリージョンにある行を、英文字順に整列します。 *reverse* が非 `nil` の場合、整列は逆順に行ないます。

sort-paragraphs *reverse start end* Command

このコマンドは、 *start* と *end* の間のリージョンにある段落を、英文字順に整列します。 *reverse* が非 `nil` の場合、整列は逆順に行ないます。

sort-pages *reverse start end* Command

このコマンドは、 *start* と *end* の間のリージョンにあるページを、英文字順に整列します。 *reverse* が非 `nil` の場合、整列は逆順に行ないます。

sort-fields *field start end* Command

このコマンドは、 *start* と *end* の間のリージョンにある行を、各行の *field* 番目をフィールドを英文字順に比較して、整列します。フィールドは空白で区切られ、 1 から数えます。 *field* が負の整数の場合、行の終りから $-$*field*番目のフィールドで整列します。このコマンドは、表の整列に便利です。

sort-numeric-fields *field start end* Command

このコマンドは、 *start* と *end* の間のリージョンにある行の *field* 番目のフィールドを数値で比較して、整列を行ないます。リージョンの各行の指定されたフィールドは数を含んでいなければなりません。フィールドは空白で区切られ、 1 から数えます。 *field* が負の場合、行の終りから $-$*field*番目のフィールドで整列します。このコマンドは、表の整列に便利です。

sort-columns *reverse* &optional *beg end* Command

このコマンドは、ある範囲の桁を英文字順で比較しながら、 *beg* と *end* の間のリージョンにある行の整列を行ないます。 *beg* と *end* の桁位置がそのまま整列範囲の桁になります。

reverse が非 `nil` の場合、整列は逆順に行ないます。

このコマンドがほかと一つ違うのは、 *beg* のある行全体と *end* のある行全体が、整列するリージョンになるということです。

`sort-columns` は `sort` ユーティリティー・プログラムを使うので、タブ文字を含むテキストではうまくいきません。整列を行なう前に `M-x untabify` を使って、タブをスペースに変換してください。

29.15 桁を数える

桁の関数は、文字の位置 (バッファの先頭から文字を数える) と、桁の位置 (行頭から画面上の文字を数える) との変換を行ないます。

文字は画面上で占める桁の数にしたがって数えます。つまり制御文字は、 `ctl-arrow` の値によって 2 あるいは 4 桁を占め、タブは `tab-width` の値とタブの始まる桁によって占める桁の数を数えます。 35.13 節「普通の表示方法」 p.617 を参照してください。

桁数の計算ではウィンドウの幅と水平スクロールは無視します。したがって、桁数はいくらでも大きくなり得ます。最初 (一番左) の桁は 0 です。

current-column Function

この関数は、ポイントの水平方向位置を返します。桁で数えて、行の先頭を 0 桁とします。桁の位置は、現在の行の先頭とポイントとの間の文字の、すべての表示された表現の幅を足し合わせて得られる値です。

`current-column` の使用例については、 27.2.4 節「テキスト行単位での移動」 p.460 の `count-lines` の解説をご覧ください。

move-to-column *column* &optional *force* Function

この関数は、現在の行の *column* にポイントを動かします。 *column* の計算は、現在の行の先頭とポイントとの間の文字の、すべての表示された表現の幅を足し合わせて得られる値です。

もし桁 *column* が行末を越えてしまう場合、ポイントを行の終りに位置づけます。もし *column* が負のときは、ポイントを行の先頭に位置づけます。

もしタブのように複数桁を占める文字の中に入ってしまい、桁 *column* へ動かすことができないとき、ポイントをその文字の終りに動かします。しかし、 *force* が非 `nil` で *column* がタブの中にあるなら、 `move-to-column` はタブをスペースに変えて、ちょうどその *column* に動かすことができます。複数桁を占めるほかの文字はわけるわけにいきませんから、 *force* を指定しても変なことになってしまいます。

引数 *force* は、行が桁 *column* より短いときにも効力があります。この場合、行末の空白の追加を行ない桁を伸ばします。

column が整数でないと、エラーを通知します。

返却値は、実際に移動した桁数です。

29.16 字下げ

字下げの関数は行頭の空白を調べたり、そこへ動いたり、変更したりするのに使います。関数によっては、行の中にある空白ならどこでも変更できます。桁と字下げは左端より 0 から数えます。

29.16.1 字下げのプリミティブ

この節では、字下げを数えたり挿入したりするのに使うプリミティブ関数について述べます。次の節の関数は、これらのプリミティブを使っています。

current-indentation Function

この関数は、現在の行の字下げを返します。これは空白でない最初の文字の、水平方向の位置です。空白文字だけの場合、行末の水平位置を返します。

indent-to *column* &optional *minimum* Command

この関数は、ポイントから *column* に達するまで、タブとスペースで字下げします。非 `nil` の *minimum* が指定されると、 *column* を越える場合も最低 *minimum* だけスペースを開けます。そうでないとき、ポイントがすでに *column* の先にあるなら、この関数は、何もしません。値は、挿入した字下げの終わる桁です。

挿入する空白文字は、まわりのテキスト (通常は先行するテキストだけ) からテキスト属性を継承します。 29.18.6 節「テキスト属性の粘着性」 p.516 を参照してください。

indent-tabs-mode User Option

非 `nil` の場合、字下げはタブを挿入してもよいことになります。そうでない場合、スペースだけ挿入します。この変数が設定されると自動的に、カレント・バッファにたいしローカルになります。

29.16.2 主モードで制御される字下げ

各主モードの重要な機能は、編集している言語にあわせて正しく字下げをするように `TAB` キーをカスタマイズすることです。この節では `TAB` キーのしくみとその制御方法について述べます。この節の関数は、予期せぬ値を返します。

indent-line-function Variable

この変数の値は、 `TAB` が (そして、いろいろな関数が) 現在の行の字下げを行なうのに使う関数です。 `indent-according-to-mode` は、この関数を呼び出すだけです。

Lisp モードでの値はシンボル `lisp-indent-line` です。 C モードでは `c-indent-line`、 Fortran モードでは `fortran-indent-line` です。 Fundamental モードや Text モード、そして標準的な字下げのないほかの多くのモードでの値は、デフォルト値の `indent-to-left-margin` です。

indent-according-to-mode Command

このコマンドは、現在の主モードにおける適切な方法で現在の行の字下げを行なう `indent-line-function` の関数を、呼び出します。

indent-for-tab-command Command

このコマンドは、 `indent-line-function` の関数で現在の行の字下げを行ないます。ただし、その関数が `indent-to-left-margin` だった場合、代わりに `insert-tab` を呼び出します (これはタブ文字を挿入する簡単なコマンドです)。

newline-and-indent Command

この関数は、改行を挿入し、主モードにしたがって (今、挿入したばかりの改行の次にある) 新しい行の字下げを行ないます。

字下げは現在の `indent-line-function` を用いて行ないます。プログラミング言語モードでは、これは `TAB` と同じになります。いくつかの Text モードでは `TAB` はタブを挿入しますが、 `newline-and-indent` は、 `left-margin` で指定した桁に字下げします。

reindent-then-newline-and-indent Command

この関数は、現在の行の再字下げを行ない、改行を挿入し、 (今、挿入したばかりの改行の次にある) 新しい行の字下げを行ないます。

このコマンドは、 `indent-line-function` の現在の値を呼び出し、現在の主モードにしたがって両方の行の字下げを行ないます。プログラミング言語モードでは、これは `TAB` と同じです。いくつかの Text モードでは `TAB` はタブを挿入しますが、 `reindent-then-newline-and-indent` は、 `left-margin` で指定された桁に字下げします。

29.16.3 リージョン全体の字下げ

この節では、リージョンの各行の字下げを行なうコマンドについて述べます。これらは予期せぬ値を返します。

indent-region *start end to-column* Command

このコマンドは、 *start*(を含む行) から *end*(は含まない) までにある、空白行以外の各行を字下げします。もし *to-column* が `nil` なら `indent-region` は、現在のモードの字下げの関数、つまり `indent-line-function` の値を呼び出して各行を字下げします。

もし *to-column* が非 `nil` なら、これは字下げを行なう桁の数を指定する整数でなければなりません。各行の空白を入れたり消したりしてちょうどこの桁になるように字下げします。

もし行詰め接頭辞があるとき、 `indent-region` は各行が行詰め接頭辞で始まるようにして、字下げを行ないます。

indent-region-function Variable

この変数の値は、 `indent-region` が処理速度を上げるために (as a short cut) 使う関数です。 (訳注: 処理にかかる時間の) 速さをのぞいて、リージョン中の各行が 1 行 1 行字下げするのと同じ結果になるよう、この関数を設計しなければなりません。

これが `nil` であるとき short cut はなく、 `indent-region` が各行にたいし実際に処理することになります。

short-cut 関数は、 C モードや Lisp モードのように、 `indent-line-function` が、関数定義の始まりをスキャンしなければならないときに、役に立ちます。各行にこれを適用すると、処理時間は自乗に比例して増大するようになります。 short cut は行の字下げを行ないながらスキャン情報を更新できるので、処理時間は線形に増大するようになります。行を個別に字下げする方が速くなるようなモードでは、 short cut の必要はありません。

indent-region に非 nil の引数 *to-column* を与えたときは、別の意味になるので、この変数は、使いません。

indent-rigidly *start end count* Command

このコマンドは、 *start*(を含む行) から *end*(は含まない) までにある、すべての行を *count* 桁だけ横に字下げします。これは、影響するリージョンを一つのきっちりした単位として動かし、「形を保存」します。

たとえば、 *count* が 3 なら、このコマンドは、指定されたリージョンの始めの行から、 3 桁分の字下げを加えます。

Mail モードで、 C-c C-y (mail-yank-original) は、 indent-rigidly を使い、返信するメッセージのテキストのコピーの字下げを行ないます。

indent-code-rigidly *start end columns* &optional *nochange-regexp* Function

これは indent-rigidly に似ていますが、文字列やコメントの行には変更をしません。

また、 *nochange-regexp* が行頭と一致する場合も、行の変更をしません (もし *nochange-regexp* が非 nil なら)。

29.16.4 前の行からの相対的な字下げ

この節では、前の行の内容を基に現在の行の字下げを行なう二つのコマンドについて述べます。

indent-relative &optional *unindented-ok* Command

前にある空白でない行における次の『字下げ点 (indent point)』まで空白を開けます。字下げ点とは空白に続く空白以外の文字です。次の字下げ点とは、現在のポイントの桁より大きい桁で最初のものです。たとえば、ポイントがテキスト行の最初の空白でない文字の左にあるとき、空白を挿入してその桁に移動します。

前にある空白でない行が、次の字下げ点をもたない (つまり十分大きい桁位置のない) 場合、 indent-relative は何もしない (*unindented-ok* が非 nil のとき) かあるいは、かわりに tab-to-tab-stop を行ないます。したがってポイントが短いテキスト行の最後の桁よりも右下にあるとき、このコマンドは、空白を挿入して次のタブ・ストップへ移動します。

indent-relative は予期せぬ値を返します。

次の例において、ポイントは 2 行目の先頭にあります。

```
            This line is indented twelve spaces.
⋆The quick brown fox jumped.
```

式 (indent-relative nil) を評価するとこうなります。

```
            This line is indented twelve spaces.
            ⋆The quick brown fox jumped.
```

この例では、ポイントが ‘jumped’ の ‘m’ と ‘p’ の間にあります。

```
                This line is indented twelve spaces.
        The quick brown fox jum⋆ped.
```

式 (indent-relative nil) を評価するとこうなります。

```
                This line is indented twelve spaces.
        The quick brown fox jum  ⋆ped.
```

indent-relative-maybe Command

このコマンドは、前にある空白でない行と同じように現在の行を字下げします。*unindented-ok* 引数を t にして、 indent-relative を呼び出します。これは予期せぬ値を返します。

もし前の空白行でない行が、現在の桁の先に字下げ点をもたないとき、このコマンドは何もしません。

29.16.5 調節可能な「タブ・ストップ」

この節ではユーザが指定する「タブ・ストップ」のしくみと、これを使ったり、設定したりするしくみについて説明します。「タブ・ストップ」という言葉は、タイプライターのタブ・ストップの機能と、この機能が似ているためにつきました。この機能は次のタブ・ストップ桁になるまで、適当な数のスペース文字やタブ文字を挿入して実現しています。バッファの中のタブ文字の表示には影響しません (35.13 節「普通の表示方法」 p.617 参照)。入力としての TAB 文字は、 Text モードのような一部の主モードでだけ、このタブ・ストップの機能を使います。

tab-to-tab-stop Command

このコマンドは、 tab-stop-list で定義された、次のタブ・ストップの桁までスペースかタブを挿入します。現在の桁数よりも大きな要素をリストから検索し、その要素がしめす桁まで字下げします。このような要素がみつからなかったときは、何もしません。

tab-stop-list User Option

この変数は、 tab-to-tab-stops で用いられるタブ・ストップの桁のリストです。要素は昇順の整数でなければなりません。タブ・ストップの桁が等間隔である必要は、ありません。

タブ・ストップの位置を対話的に編集するには M-x edit-tab-stops を用います。

29.16.6 字下げベースの移動コマンド

以下のコマンドはテキストの字下げをもとに動作するもので、主に対話的に用います。

back-to-indentation Command

このコマンドは、ポイントを現在の行 (ポイントの位置する行) の最初の空白でない文字に動かします。これは nil を返します。

backward-to-indentation *arg* Command

ポイントを *arg* 行逆方向に動かし、その行の最初の空白でない文字に動かします。これは nil を返します。

forward-to-indentation *arg* Command

arg 行先に動かし、その行の最初の空白でない文字に位置づけます。この関数は、 `nil` を返します。

29.17 大文字 / 小文字の変換

ここで述べる大文字と小文字の変換を行なう関数は、カレント・バッファのテキストに作用します。文字列や文字に作用する変換コマンドについては、4.7 節「大文字・小文字変換」p.63 を参照してください。どの文字が大文字か、小文字か、またそれらの変換をカスタマイズする方法については4.8 節「大文字小文字変換のカスタマイズ」p.65 を参照してください。

capitalize-region *start end* Command

この関数は、 *start* と *end* で定義されるリージョン内のすべての語を、大文字で始めます。大文字で始めるとは、各語の最初の文字を大文字に、残りの文字を小文字にすることを意味します。 `nil` を返します。

リージョンの一端が語の中間にあるとき、リージョンの内にある語の一部を語全体とみなします。

`capitalize-region` が対話的に呼ばれた場合、マークとポイントのうち小さい方から *start* と *end* になります。

```
---------- Buffer: foo ----------
This is the contents of the 5th foo.
---------- Buffer: foo ----------

(capitalize-region 1 44)
⇒ nil

---------- Buffer: foo ----------
This Is The Contents Of The 5th Foo.
---------- Buffer: foo ----------
```

downcase-region *start end* Command

この関数は、 *start* と *end* で定義されるリージョン中のすべての語を小文字にします。 `nil` を返します。

`downcase-region` が対話的に呼ばれた場合、 *start* と *end* は小さい方を先にしてマークとポイントになります。

upcase-region *start end* Command

この関数は、 *start* と *end* で定義されるリージョン中のすべての語を大文字にします。 `nil` を返します。

`upcase-region` が対話的に呼ばれた場合、 *start* と *end* は小さい方を先にしてマークとポイントになります。

capitalize-word *count* Command

この関数は、ポイントの後の *count* 語を、ポイントを動かしながら大文字で始めます。大文字で始めるとは、各語の最初の文字を大文字に、残りの文字を小文字にすることを意味します。*count* が負の場合、これは前の $-count$ 語を大文字で始めますが、ポイントは動かしません。値は `nil` です。

もしポイントが語の中間にあるなら、前に進む場合、ポイントの前の語の部分は無視します。残りは語全体として扱います。

`capitalize-word` が対話的に呼ばれた場合、 *count* は数値前置引数に設定されます。

downcase-word *count* Command

この関数は、ポイントの後の *count* 語をポイントを動かしながら、すべて小文字に変換します。 *count* が負の場合、これは前の $-count$ 語を変換しますが、ポイントは動かしません。値は `nil` です。

`downcase-word` が対話的に呼ばれた場合、 *count* は数値前置引数に設定されます。

upcase-word *count* Command

この関数は、ポイントの後の *count* 語をポイントを動かしながら、すべて大文字に変換します。 *count* が負の場合、これは前の $-count$ 語を変換しますが、ポイントは動かしません。値は `nil` です。

`upcase-word` が対話的に呼ばれた場合、 *count* は数値前置引数に設定されます。

29.18 テキスト属性

バッファや文字列に位置する各文字は、シンボルの属性リスト (7.4 節「属性リスト」 p.98 参照) のような、『テキスト属性リスト (text property list)』をもたせることができます。属性は特定の場所にある特定の文字に、たとえば、 (英語の原文の) この文章の先頭にある文字 ‘T’ や、‘`foo`’ の中の最初の ‘`o`’ に、属します。違う場所に同じ文字がある場合、一般には異なる属性をもっています。

各属性には名前と値があります。両方とも任意の Lisp オブジェクトでかまいませんが、通常名前はシンボルです。属性リストへの普通のアクセス方法は、名前を指定しそれに対応する値を調べることです。

もし文字が `category` という属性をもつとき、これを文字の『範疇 (category)』といいます。これはシンボルでなければなりません。シンボルの属性が文字の属性のデフォルト値となります。

文字列とバッファの間でテキストを複写しても、文字の属性は保存されます。これは `substring`、 `insert`、 `buffer-substring` のようにテキストの一部分を取り扱う関数 (diverse functions) を含みます。

29.18.1 テキスト属性の調査

テキスト属性を調べるもっとも単純な方法は、特定の文字の特定の属性の値を見ることです。これには `get-text-property` を使います。文字の全属性リストを得るには `text-properties-at` を使います。一度にいくつもの文字の属性を調べる関数については 29.18.3 節「属性検索関数」p.512 を参照してください。

これらの関数は文字列とバッファの両方を扱います。文字列の位置は 0 から始まるのにたいして、バッファは 1 から始まることに気をつけてください。

get-text-property *pos prop* &optional *object* Function

この関数は、 *object*(バッファあるいは文字列) の位置 *pos* の後の文字の *prop* 属性の値を返します。引数 *object* は省略可能で、デフォルト値はカレント・バッファです。

もし *prop* 属性がまったくなく、文字がシンボルの範疇をもつとき、 `get-text-property` はシンボルの *prop* 属性を返します。

get-char-property *pos prop* &optional *object* Function

この関数は、 `get-text-property` に似ていますが、最初にオーバレイを調べてからテキスト属性を調べます。 35.9 節「オーバレイ」p.609 を参照してください。

引数の *object* は文字列、バッファ、もしくはウィンドウです。もしウィンドウならば、ウィンドウが表示しているバッファのテキスト属性とオーバレイを使います。もし *object* がバッファなら、テキスト属性と同様にそのバッファの全オーバレイが対象になります。文字列にはオーバレイがないので、もし *object* が文字列ならテキスト属性だけが対象です。

text-properties-at *position* &optional *object* Function

この関数は、文字列やバッファの *object* の中の *position* にある文字について、すべての属性リストを返します。もし *object* が `nil` なら、カレント・バッファが対象です。

default-text-properties Variable

この変数は、テキスト属性のデフォルト値を与える属性リストを保持しています。文字がある属性の値を直接にも範疇シンボルにも指定していないときは、いつもこのリストに格納した値を代用します。

```
(setq default-text-properties '(foo 69))
;; 文字位置 1 には独自の属性のないようにする。
(set-text-properties 1 2 nil)
;; 得たものは聞いた時点でのデフォルト値。
(get-text-property 1 'foo)
     ⇒ 69
```

29.18.2 テキスト属性の変更

属性を変更するプリミティブは、指定したテキストの範囲に適用します。関数 `set-text-properties`(この節の最後を参照) は、範囲内のテキストの全属性リストを設定します。普段は、名前で指定したある特定の属性だけの追加、変更、消去が役に立ちます。

テキスト属性はバッファの内容の一部なので、画面でのバッファの見え方に影響し、テキスト属性の変更はバッファの修正とみなされます。バッファのテキスト属性の変更は取り消しできます (29.9 節「取り消し」 p.492 参照)。

put-text-property *start end prop value* &optional *object* Function

この関数は、文字列もしくはバッファ *object* のテキストの、 *start* と *end* の間にある *prop* 属性を、 *value* に設定します。もし *object* が `nil` なら、カレント・バッファが対象です。

add-text-properties *start end props* &optional *object* Function

この関数は、文字列もしくはバッファ *object* のテキストの、 *start* と *end* の間にあるテキスト属性を変更します。もし *object* が `nil` なら、カレント・バッファが対象です。

引数 *props* は、どの属性を変更するかを指定します。これは属性リスト (7.4 節「属性リスト」 p.98 参照) の形式、つまり属性の名前と対応する値が交互になっている要素をもつリストの形式を、もたなければなりません。

関数が実際にある属性の値をかえたとき、返却値は `t` です。そうでなければ (*props* が `nil` か、値がテキストのそれと同じだったら)、 `nil` です。

たとえば、テキストのある範囲の `comment` と `face` という属性を変えるには、こうします。

```
(add-text-properties start end
                     '(comment t face highlight))
```

remove-text-properties *start end props* &optional *object* Function

この関数は、文字列もしくはバッファ *object* の *start* と *end* の間のテキストから、指定したテキスト属性を消去します。もし *object* が `nil` なら、カレント・バッファが対象です。

引数 *props* は、消去する属性を指定します。これは属性リスト (7.4 節「属性リスト」 p.98 参照) の形式、つまり属性の名前と対応する値が交互になっている要素をもつリストの形式を、もたなければなりません。しかし、意味をもつのは、名前だけです。名前に付随する値は無視されます。たとえば、 `face` 属性を消去するには、こうします。

```
(remove-text-properties start end '(face nil))
```

関数が実際にある属性の値をかえたとき、返却値は `t` です。そうでなければ (*props* が `nil` か、指定したテキストにこれらの属性をもつ文字がまったくなかったら)、 `nil` です。

set-text-properties *start end props* &optional *object* Function

この関数は、文字列もしくはバッファ *object* の、 *start* と *end* のテキストの間のテキスト属性を、指定したものと置き換えます。もし *object* が `nil` なら、カレント・バッファが対象です。

引数 *props* は新しい属性リストです。これは、属性の名前と対応する値が交互になっている要素をもつリストでなければなりません。

`set-text-properties` から戻ると、指定した範囲にあるすべての文字は、同じ属性をもちます。

もし *props* が `nil` なら、指定した範囲のテキストからすべての属性を取り去ることになります。例をあげます。

```
(set-text-properties start end nil)
```

属性は転記せずにバッファのテキストを転記する関数 `buffer-substring-no-properties` については 29.2 節「バッファ内容の調査」p.480 を参照してください。

29.18.3 属性検索関数

テキスト属性の典型的な利用法では、ほとんどの間、いくつかのあるいは多くの連続した文字が、ある属性にたいし同じ値をとります。文字を一つひとつ調べるプログラムを書くよりも、同じ属性の値をもつテキストのかたまりを処理する方が速くなります。

こういうときに使える関数があります。属性値の比較には `eq` を使います。いずれの場合も、*object* のデフォルト値はカレント・バッファです。

効率を上げるため、これらの関数、特に単一の属性を検索する関数で、 *limit* 引数を使うことはとても大切です。さもないと、調べている属性が変化しないなら、バッファの最後までスキャンするので長い時間がかかるからです。

位置は常に二つの文字の間にあることに注意してください。この関数が返す値は、異なる属性をもつ二つの文字の間の位置です。

next-property-change *pos* &optional *object limit* Function

この関数は、位置 *pos* から文字列もしくはバッファの *object* のテキストを、あるテキスト属性が変化するまで前方にスキャンします。いいかえると、 *pos* 以降で文字の属性が *pos* の文字の属性と異なる最初の文字の位置を返します。

もし *limit* が非 `nil` のとき、スキャンは位置 *limit* をもって中止します。その場所までに属性の変更がないとき、 `next-property-change` は *limit* を返します。

limit が `nil` で *object* の最後まで属性が変らないときは、 `nil` を返します。もし値が非 `nil` なら、 *pos* かそれより大きな値の位置です。 *limit* と *pos* が等しいときにかぎり、値は *pos* になります。

以下に、すべての属性が一定のテキストのひとかたまりずつ、バッファをスキャンする方法の例を示します。

```
(while (not (eobp))
  (let ((plist (text-properties-at (point)))
        (next-change
         (or (next-property-change (point) (current-buffer))
             (point-max))))
    ポイントから next-change までを処理する ...
    (goto-char next-change)))
```

next-single-property-change *pos prop* &optional *object limit* Function

この関数は、位置 *pos* から文字列もしくはバッファの *object* のテキストを、 *prop* 属性が変化するまで前方にスキャンし、変化のあった位置を返します。いいかえると、 *pos* 以降で *prop* 属性が *pos* 直後の文字の属性と異なる最初の文字の位置を返します。

もし *limit* が非 `nil` ならば、スキャンは位置 *limit* で終了します。その場所までに属性の変更がないとき、 `next-single-property-change` は *limit* を返します。

属性が *object* の最後まで一定で *limit* が `nil` ならば、値は `nil` です。もし値が非 `nil` なら、 *pos* かそれより大きな値の位置です。 *limit* と *pos* が等しいときにかぎり、値は *pos* になります。

previous-property-change *pos* &optional *object limit* Function

これは `next-property-change` に似ていますが、 *pos* より順方向ではなく逆方向にスキャンを行ないます。もし値が非 `nil` なら、 *pos* かそれより小さな値の位置です。 *limit* と *pos* が等しいときにかぎり、値は *pos* になります。

previous-single-property-change *pos prop* &optional *object limit* Function

これは `next-single-property-change` に似ていますが、 *pos* より順方向ではなく逆方向にスキャンを行ないます。もし値が非 `nil` なら、 *pos* かそれより小さな値の位置です。 *limit* と *pos* が等しいときにかぎり、値は *pos* になります。

text-property-any *start end prop value* &optional *object* Function

この関数は、 *start* から *end* までの間に、属性 *prop* の値が *value* であるような文字が少なくとも一つあるとき非 `nil` の値を返します。より正確にいうと、そのような最初の文字の位置を返します。ない場合は、 `nil` を返します。

省略可能な 5 番目の引数 *object* は、スキャンされる文字列もしくはバッファを指定します。 *object* のデフォルト値は、カレント・バッファです。

位置は *object* から相対的なものになります。 *object* のデフォルト値は、カレント・バッファです。

text-property-not-all *start end prop value* &optional *object* Function

この関数は、 *start* から *end* までの間に、属性 *prop* の値が *value* と異なるような文字が少なくとも一つあるとき非 `nil` の値を返します。より正確にいうと、そのような最初の文字の位置を返します。ない場合は、 `nil` を返します。

省略可能な 5 番目の引数 *object* は、スキャンされる文字列もしくはバッファを指定します。位置は *object* から相対的なものになります。 *object* のデフォルト値は、カレント・バッファです。

29.18.4 特殊な意味をもつ属性

次の表は、組込みで特別な意味をもつテキスト属性の名前です。次の節には、行詰めの制御を行なうときに使われる特別な属性名のリストがあります。これ以外の名前に標準の意味はないので、好きなように使うことができます。

`category` 文字に category 属性があるとき、これをその文字の『範疇 (category)』といいます。これはシンボルでなければなりません。このシンボルの属性は、この文字の属性のデフォルト値となります。

`face` 属性 face を使ってテキストの字体や色を制御することができます。値は字体のフェース名かあるいはフェース名のリストです。詳しくは 35.10 節「フェース」p.613 を参照してください。この機能は一時的なものになるかもしれません。将来は、ほかの方法でテキストの表示の仕方をするよう変えるかもしれません。

`mouse-face`

属性 mouse-face は、文字の上または近くにマウスがきたとき face の代わりに使います。ここで「近い」というのは、マウスの場所とその文字の間のテキストがすべて同じ mouse-face 属性の値をもつということです。

`local-map` local-map 属性でテキストの一部分に別なキーマップを指定できます。ポイントの次の文字の属性の値が非 nil ならば、バッファのローカル・マップを置き換えます。 19.6 節「アクティブ・キーマップ」p.298 を参照してください。

`read-only` 文字が属性 read-only をもつとき、その文字の変更はできません。そのようなことをするコマンドはエラーになります。

(訳注: 属性の) 粘着性により、通常のテキストの挿入で read-only 属性が継承された場合、読出し専用文字の前後への挿入はエラーになります。したがって、粘着性を制御することにより、読出し専用テキストのまわりへの挿入の権限を制御することができます。 29.18.6 節「テキスト属性の粘着性」p.516 を参照してください。

属性の変更はバッファの修正に入るので、 inhibit-read-only を非 nil の値に束縛してから属性を消去するという方法を使わないかぎり、 read-only 属性を消去することは不可能です。 24.7 節「読みだし専用バッファ」p.401 を参照してください。

`invisible` invisible 属性が非 nil の場合、文字を画面上で不可視にすることができます。詳細は 35.5 節「不可視テキスト (invisible text)」 p.604 を参照してください。

`intangible`

もし連続した文字群に、非 nil で equal の intangible 属性がある場合、その文字群の中間にポイントを置くことができなくなります。もし順方向にポイントをその群の中へ入れようとしたとき、実際のポイントは群の後に移動します。もし逆方向にポイントをその群の中へ入れようとしたとき、実際のポイントは群の前に移動します。

変数 inhibit-point-motion-hooks が非 nil のとき、 intangible 属性は無視されます。

`modification-hooks`

文字に属性 modification-hooks があるとき、これは関数のリストでなければなりません。この関数群は、その文字の修正時に呼び出されます。各関数には二つの

引数、すなわちバッファの修正部分の最初と最後が、渡されます。ある特定の修正フック関数が複数の文字上にあるとき、単一のプリミティブによる修正では、何回呼び出されるのかを予期することはできません。

`insert-in-front-hooks`
`insert-behind-hooks`
バッファへのテキスト挿入操作では、実際にバッファを修正する前に、後続する文字の `insert-in-front-hooks` 属性のリストにある関数群と、先行する文字の `insert-behind-hooks` 属性のリストにある関数群が、呼び出されます。挿入されるテキストの最初と最後が、引数としてこの関数群に渡されます。

バッファのテキストを修正したとき呼び出されるほかのフックについては 29.22 節「変更フック」p.521 を参照してください。

`point-entered`
`point-left`
特殊な属性 `point-entered` と `point-left` には、ポイントの移動を報告するフック関数群を登録します。ポイントが移動するたびに、 Emacs は以下の属性値との比較を行ないます。

- 以前の位置の直後の文字の `point-left` 属性、そして
- 新しい位置の直後の文字の `point-entered` 属性。

もしこれら二つの値が違うとき、それらは (`nil` でなければ) いずれも、ポイントの以前の値と新しい値を引数にして呼び出されます。

同様の比較は新旧各位置の直前の文字についても行ないます。結果として、二つの `point-left` 関数 (両方が同じ場合もあります) と、二つの `point-entered` 関数 (両方が同じ場合もあります) が、実行されることになります。いずれの場合も全 `point-left` 関数が呼び出されてから、全 `point-entered` 関数が呼び出されます。

プリミティブ関数はポイントを動かさずに、いろいろな位置の文字を調べることがあります。ポイントの値の実際の変更だけが、これらのフック関数を作動させます。

inhibit-point-motion-hooks Variable

この変数が非 `nil` の場合、 `point-left` と `point-entered` フックは作動せず、`intangible` 属性の効果はなくなります。

29.18.5 整形済みテキスト属性

以下のテキスト属性は、行詰めコマンドの振舞いに影響を与えます。表示用に整形されたテキストに使います。 29.11 節「行詰め」p.495 と 29.12 節「行詰めのマージン」p.497 を参照してください。

`hard` もし改行文字にこの属性があるとき、それは「強い」意味の改行であることを示します。行詰めコマンドは強い改行を変えたり、あるいは強い改行を越えて語を動か

したりすることはありません。しかし、この効果は、 use-hard-newlines が非 nil のときだけ有効です。

right-margin
: この属性は、この部分のテキストの行詰めを行なうときの、右マージンの追加を指定します。

left-margin
: この属性は、この部分のテキストの行詰めを行なうときの、左マージンの追加を指定します。

justification
: この属性は、この部分のテキストの行詰めを行なうときの、右揃えのスタイルを指定します。

29.18.6 テキスト属性の粘着性

自己挿入文字は通常、先行する文字と同じ属性をとります。これを属性の『継承 (inheritance)』といいます。

Lisp プログラムでは、選択した挿入用プリミティブにしたがって、継承つきや継承なしの挿入を行なうことができます。 insert のような通常のテキスト挿入関数は属性の継承を行ないません。挿入する文字列と同一の属性だけを挿入し、ほかの属性ははいりません。テキストをある文脈から別の文脈へ複写するプログラムのときにも、このことは成立します。これはたとえば、 kill リングからのあるいは kill リングへの複写を行なう場合です。継承つきの挿入を行なうには、この節で説明する特殊なプリミティブを使います。自己挿入文字はこれらのプリミティブを使っているので、属性の継承を行ないます。

継承つき挿入を行なうときどの属性が継承されるかは、二つの属性に依存します。その属性とは front-sticky と rear-nonsticky です。

文字の後への挿入は『後方粘着的 (rear-sticky)』な属性を継承します。文字の前への挿入は『前方粘着的 (front-sticky)』な属性を継承します。デフォルトでテキスト属性は後方粘着的ですが、前方粘着的ではありません。したがってデフォルトでは先行する文字の全属性を継承し、後続する文字の属性はまったく継承しません。ある属性の粘着性を指定して、この振舞いを変えることができます。

もし文字の front-sticky 属性が t ならば、その文字のすべての属性は、前方粘着的です。もし front-sticky がリストならば、その文字の粘着的属性はリストに名前が出ているものになります。たとえば、もしある文字の front-sticky 属性の値が (face read-only) だったならば、前に挿入した文字は face 属性と read-only 属性を継承しますが、ほかには継承を行ないません。

rear-nonsticky は逆向きに作用します。すべての属性はデフォルトで後方粘着的ですから、 rear-nonsticky 属性はどの属性が非後方粘着的かを示すわけです。もし文字の rear-nonsticky 属性が t ならば、後方粘着的な属性はなくなります。もし rear-nonsticky 属性がリストのときは、リストに名前が出ていない属性が後方粘着的になります。

継承つきでテキストを挿入するとき、先行する文字の後方粘着的な属性と、後続する文字の前方粘着的な属性を継承します。両側で同じ属性が違う値のとき、前の文字の属性が優先します。

以下の関数が属性の継承を行なうテキストの挿入関数です。

insert-and-inherit &rest *strings* Function

関数 `insert` のように文字列 *strings* を挿入しますが、隣のテキストから粘着的な属性を継承します。

insert-before-markers-and-inherit &rest *strings* Function

関数 `insert-before-markers` のように文字列 *strings* を挿入しますが、隣のテキストから粘着的な属性を継承します。

29.18.7 テキスト属性をファイルへセーブする

テキスト属性をファイルの中に保存したり、ファイルを挿入するときテキスト属性を復帰したりするには、以下の二つのフックを使います。

write-region-annotate-functions Variable

この変数の値は関数のリストで、書き込もうとするファイルのテキストにたいする注釈のように、ある方法でテキスト属性を符号化するために、 `write-region` が作動させるものです。 22.4 節「ファイルへの書込み」p.356 を参照してください。

リスト中の各関数には二つの引数、書き込むリージョンの最初と最後を渡します。この関数は、バッファの内容を変えてはなりません。またこれは、バッファのテキストのほかにファイルに書き込む注釈を示すリストを、返さなければなりません。

各関数は、 (*position* . *string*) の形をした要素のリストを、返さなければなりません。ここで *position* は整数で、書き込むテキストからの相対位置を示し、 *string* はそこに追加する注釈です。

この関数群のうちの一つから返される各リストは、あらかじめ *position* の昇順に整列されていなければなりません。もし複数の関数があるとき、 `write-region` はリストを破壊的に併合し、一つの整列したリストにします。

`write-region` が、バッファからテキストを実際にファイルへ書き込むとき、対応する位置にある指定した注釈を混ぜ合せます。これらはバッファを修正することなく行なわれます。

after-insert-file-functions Variable

この変数は、 `insert-file-contents` がファイルの内容を挿入した後に呼び出す、関数のリストを保持します。この関数群は挿入したテキストをスキャンし対応するテキスト属性に変換します。

各関数は一つの引数、挿入したテキストの長さを受け取ります。テキストの始まりは、ポイントで示します。この関数は、注釈のテキストをスキャン、消去し、注釈が指定するテキスト属性を生成します。ある関数の返した値は、次の関数への引数になります。

この関数群が戻るときは、いつも挿入したテキストの先頭にポイントを置かなければなりません。

想定していた `after-insert-file-functions` の使い方は、テキスト化した注釈を実際のテキスト属性に変換することです。しかし、ほかの使い方をしてもかまいません。

私たちはユーザに、ファイルにテキスト属性を格納したり検索したりする Lisp プログラムを、このフックを使って書いていただきたい、と思います。そうすれば、いろいろなデータ形式を試して、いいものを見つけることができます。私たちはユーザに、 Emacs にインストールできる一般的でよい拡張をうみだしていただくことを、希望します。

属性の名前やその値に任意の Lisp オブジェクトを扱うように試みることは、提案していません。なぜなら一般的なプログラムというものは多分書くのが難しく遅いからです。その代わり、適度に柔軟なデータ型を選んで符号化が難しくならないようにしてください。

これに関連する機能については 22.12 節「ファイル・フォーマットの変換」 p.377 を参照してください。

29.18.8 テキスト属性が区間ではない理由

あるエディタは、テキストの「区間」をユーザに指定させてその区間に属性を付加して、バッファのテキストに属性を付加することをサポートしています。このようなエディタはユーザやプログラマに各区間の始めと終りを決めることを許しています。私たちは、テキストの修正にまつわるある逆説的な振舞いを避けるため、 Emacs Lisp へ異なった種類のインタフェースを慎重に与えました。

もし区間への分割に意味があるならば、ある属性のある一つの区間をもつバッファと、同じテキストをもち同じ属性が二つの区間になっているバッファとを、区別できるはずです。

一つだけの区間をもつバッファがあったとして、テキストの一部分を kill したとします。バッファに残ったテキストは一つの区間で、 kill リング (そして undo リスト) 中の複製は別の区間になります。このとき kill したテキストをヤンクすると、同じ属性をもつ二つ区間ができてしまいます。したがって編集の上では、一つの区間と二つの区間を区別することができないのです。

この問題を「修正」するのに、テキストが挿入されたときに二つの区間を合体しようとしたとしましょう。今度は、同じ属性をもつ二つの隣あった区間があるとして、片方の区間のテキストを kill しヤンクして戻したとしてください。以前の問題を解決したのと同一な区間の合体機能が、今度は問題になります。つまりヤンクすると、一つの区間になってしまうからです。繰り返しますが、編集の上で、一つの区間と二つの区間の区別を保つことができないのです。

区間と区間の境界にテキストを挿入した場合も、満足できる解答のでない問題をひき起こします。

しかしながら、「この文字の属性は何か?」という形の質問に一貫性をもって振る舞う編集法を確立するのは、簡単です。そのため、私たちはこういう質問だけに意味がある、とみなすことに決めました。どこで区間が始まりどこで区間が終わるのか、という質問に答えるような実装を私たちはとっていません。

実際、たいていは陽な区間の境界の代わりに属性の検索関数を使うことができます。可能ならばいつでも常に区間が合体すると仮定するならば、これを区間の境界を見つけたのだ、と考えることができます。 29.18.3 節「属性検索関数」 p.512 を参照してください。

表現上の機能としては Emacs には陽な区間もあります。 35.9 節「オーバレイ」 p.609 を参照してください。

29.19 文字符号の置換

以下の関数は指定したリージョンの文字を文字コードにしたがって置換します。

subst-char-in-region *start end old-char new-char* &optional *noundo* Function

この関数は、カレント・バッファ中の *start* と *end* で定義されるリージョン内の文字 *old-char* を文字 *new-char* で置き換えます。

もし *noundo* が非 `nil` の場合、 `subst-char-in-region` はこの変更を取り消し用に記録することはせず、また修正されたとしてバッファをマークすることもしません。この機能は、選択表示の制御に使用します (35.6 節「選択表示」 p.605 参照)。

`subst-char-in-region` はポイントを動かさず、 `nil` を返します。

```
---------- Buffer: foo ----------
This is the contents of the buffer before.
---------- Buffer: foo ----------

(subst-char-in-region 1 20 ?i ?X)
     ⇒ nil

---------- Buffer: foo ----------
ThXs Xs the contents of the buffer before.
---------- Buffer: foo ----------
```

translate-region *start end table* Function

この関数は、バッファの位置 *start* と *end* の間の文字に、変換テーブルを適用します。

変換テーブル *table* は文字列です。(`aref` *tableochar*) が、 *ochar* に対応する変換後の文字になります。もし *table* の長さが 256 以下なら、 *table* の長さより大きな符号の文字は変換によって変化しません。

`translate-region` の返却値は、実際に変換で変更された文字の数です。変換テーブルで自分自身に射像した文字は、含みません。

29.20 レジスタ

レジスタは、 Emacs の編集中に、マーカ、文字列、長方形、 (一つのフレームの) ウィンドウ構成、 (全フレームの) フレーム構成を保管できる変数の一種です。各レジスタには、単一文字の名前がついています。 (`C-g` を例外にして) 制御文字とメタ文字を含んだすべての文字は、レジスタ名に使用できます。したがって 255 個のレジスタが有り得ます。 Emacs Lisp ではレジスタをその名前の文字で指し示します。

この節の関数は、特に明記しないかぎり予期せぬ値を返します。

register-alist Variable

この変数は、要素が (*name* . *contents*) の形をした alist です。通常は、使用中の各 Emacs レジスタに一つの要素があります。

オブジェクト *name* はレジスタを識別する文字 (整数) です。オブジェクト *contents* は、文字列かマーカかリストで、レジスタの内容を示します。文字列はレジスタに格納したテキストを示します。マーカは位置を示します。リストは長方形を示します。その要素は長方形の 1 行あたり一つの文字列です。

get-register *reg* Function

この関数は、レジスタ *reg* の内容あるいはそれがなければ `nil` を返します。

set-register *reg value* Function

この関数は、レジスタ *reg* の内容を *value* に設定します。レジスタはどんな値でももつことができますが、ほかのレジスタ関数はあるデータ型だけを期待します。返却値は *value* です。

view-register *reg* Command

このコマンドはレジスタ *reg* に格納されているものを表示します。

insert-register *reg* &optional *beforep* Command

このコマンドは、レジスタ *reg* の内容をカレント・バッファに挿入します。

通常、このコマンドは挿入したテキストの前にポイントを、後にマークを置きます。しかしながら、もし省略可能な第 2 引数 *beforep* が非 `nil` のときは、マークを前に、ポイントを後に置きます。前置引数を与えれば、対話的にこの関数に非 `nil` を第 2 引数 *beforep* に指定することができます。

レジスタに長方形が入っていたとき、ポイントを左上にして長方形を挿入します。これは、現在の行とその下の行にテキストを挿入するということです。

レジスタに、保存したテキスト (文字列) でも長方形 (リスト) でもないものが入っていたとき、今のところ、役に立たないことがおきます。これは将来修正されるでしょう。

29.21 テキストの交換

交換コマンドは次のサブルーチンを使います。

transpose-regions *start1 end1 start2 end2* &optional *leave-markers* Function

この関数は、二つの重ならないバッファの部分を交換します。引数 *start1* と *end1* が片方の境界を指定し、引数 *start2* と *end2* がもう片方の境界を指定します。

通常、 `transpose-regions` は交換するテキストといっしょにマーカも再配置します。交換した二つの部分の内の片方に入っていたマーカが、その部分といっしょに動きます。したがって新しい部分においても文字と文字の間の位置関係は変りません。しかしなが

ら、もし *leave-markers* が非 `nil` ならば、 `transpose-regions` はマーカの移動をしません。

29.22 変更フック

以下のフック変数は、すべてのバッファ (これをバッファローカルにすれば、ある特定のバッファ) におけるすべての変更に着目するときに使います。テキストの特定の位置の変更を検出するには 29.18.4 節「特殊な意味をもつ属性」 p.513 を参照してください。

このフックを使う関数は、正規表現を使う操作を行なったとき一致データの保存と復帰をしなければなりません。そうしないと一致データを使う編集が悪影響を受けてしまいます。

before-change-functions Variable

この変数は、すべてのバッファ修正の前に呼ばれる関数のリストです。各関数は、変更されようとするリージョンの始めと終りを表わす二つの整数を、引数にうけとります。変更しようとしているバッファは常にカレント・バッファです。

after-change-functions Variable

この変数は、すべてのバッファ修正の後に呼ばれる関数のリストです。各関数は、変更されたリージョンの始めと終り、そして変更前のテキストの長さを引数に受取ります (現在の長さは、リージョンの終りから始めを引いた値です)。三つの引数はすべて整数です。変更しようとしているバッファは常にカレント・バッファです。

before-change-function Variable

この obsolete 変数は、すべてのバッファの修正の前に呼び出される関数を一つ保持します (関数がないときは `nil`)。これは `before-change-functions` の中の関数のように呼び出されます。

after-change-function Variable

この obsolete 変数は、すべてのバッファの修正の後に呼び出される関数を一つ保持します (関数がないときは `nil`)。これは `after-change-functions` の中の関数のように呼び出されます。

上の四つの変数は、いずれかの関数が作動している間は一時的に、 `nil` に束縛されます。そのためこれらの関数の一つがバッファを変更しても、この関数群を作動させることはありません。これらの関数を作動させるような変更を行なうフック関数が必要なときは、この変数群を通常の値に束縛してください。

この見込み機能による不便な結果は、 `after-change-functions` や `before-change-functions` の中で、この変数の値の変更をする関数ができないということです。しかし、これは真の制限ではありません。この関数群に作動させる関数のリストを変更させたいときは、単にある固定した一つの関数をフックに追加し、呼び出すほかの関数を別な変数から見るような関数を書きます。例をあげます。

```
(setq my-own-after-change-functions nil)
(defun indirect-after-change-function (beg end len)
```

```
  (let ((list my-own-after-change-functions))
    (while list
      (funcall (car list) beg end len)
      (setq list (cdr list)))))
(add-hooks 'after-change-functions
           'indirect-after-change-function)
```

first-change-hook Variable

この変数は、それまで修正されていない状態からバッファの変更があったときに必ず作動する、正規フックです。

30 検索と照合

GNU Emacs には、バッファ内のあるテキストを検索する方法が二つあります。それは文字列そのものによる検索と正規表現による検索です。正規表現の検索の後では、『一致データ (match data)』を調べて、どのテキストが正規表現の全体やその一部分に一致したのか判別することができます。

'skip-chars...' 関数群も一種の検索を行ないます。27.2.7 節「文字を飛ばす」p.465 を参照してください。

30.1 文字列の検索

以下の関数はバッファのテキストを検索するプリミティブ関数です。プログラムから使うことを想定していますが、対話的に呼び出すこともできます。その場合、プロンプトを出して検索する文字列の入力を促します。 *limit* と *noerror* は `nil` に設定され、 *repeat* は 1 に設定されます。

search-forward *string* &optional *limit noerror repeat* Command

この関数は、 *string* に完全一致するテキスト文字列を求め、ポイントから順方向に検索します。もし成功すれば、見つかったものの終りにポイントを置いて、新しいポイントの値を返します。もし見つからなければ、戻り値と副作用は *noerror* によります (以下参照)。

以下の例で、ポイントははじめ、行頭にあります。そして `(search-forward "fox")` によって、'`fox`' の最後の文字の後に動かされます。

```
---------- Buffer: foo ----------
⋆The quick brown fox jumped over the lazy dog.
---------- Buffer: foo ----------

(search-forward "fox")
     ⇒ 20

---------- Buffer: foo ----------
The quick brown fox⋆ jumped over the lazy dog.
---------- Buffer: foo ----------
```

引数 *limit* は検索にたいする上限を指定します (カレント・バッファ内の位置でなくてはいけません)。この位置を越えるものとは照合しません。もし *limit* が省略されたか `nil` の場合、アクセス可能なバッファの末尾になります。

検索が失敗したときの動作は、 *noerror* の値によります。もし *noerror* が `nil` ならば、`search-failed` エラーが通知されます。もし *noerror* が `t` の場合、 `search-forward` は `nil` を返し何もしません。もし *noerror* が `nil` でも `t` でもない場合、 `search-forward` はポイントを上限まで動かし、 `nil` を返します (この場合、新しいポイントの値を返した方が一貫性があるのですが、 `nil` という値に依存したプログラムがあるようです)。

もし *repeat* が与えられた場合 (正数でなければなりません)、その回数だけ (毎回前の照合の末尾から) 検索を繰り返します。もしこの連続検索が成功したら、この関数は、成功です。ポイントを動かして新しい値を返します。さもないと検索は失敗です。

search-backward *string* &optional *limit noerror repeat* Command

この関数は、ポイントから逆方向に *string* を検索します。これは `search-forward` と対応していますが、逆方向に検索し、一致した文字列の先頭にポイントを置きます。

word-search-forward *string* &optional *limit noerror repeat* Command

この関数は、 *string* に一致する「語」を求め、ポイントから順方向に検索します。もし見つかったら、一致の終りにポイントを置き、ポイントの新しい値を返します。

語の照合では、 *string* を語の列であるとみなし、語を分離する句読点文字を無視します。バッファから同じ語の列を検索します。語はそれぞれバッファ中で分離していなければなりません (語 '`ball`' の検索は語 '`balls`' には一致しません) が、句読点文字と余白の細部は無視します ('`ball boy`' の検索は '`ball. Boy!`' に一致します)。

この例において、ポイントは最初バッファの先頭にあり、検索の後 '`y`' と '`!`' の間に移動します。

```
---------- Buffer: foo ----------
⋆He said "Please!  Find
the ball boy!"
---------- Buffer: foo ----------

(word-search-forward "Please find the ball, boy.")
     ⇒ 35

---------- Buffer: foo ----------
He said "Please!  Find
the ball boy⋆!"
---------- Buffer: foo ----------
```

limit が非 `nil` (カレント・バッファ内の位置でなくてはいけません) の場合、これは検索にたいする上限になります。この位置を越えるものとは照合しません。

noerror が `nil` の場合、検索が失敗すると `word-search-forward` はエラーを通知します。 *noerror* が `t` の場合、検索が失敗するとエラーを通知する代わりに `nil` を返します。 *noerror* が `nil` でも `t` でもない場合、これはポイントを *limit*(またはバッファの末尾) まで動かし、 `nil` を返します。

もし *repeat* が非 `nil` の場合、その回数だけ検索を繰り返します。最後に一致したものの終りにポイントを位置づけます。

word-search-backward *string* &optional *limit noerror repeat* Command

この関数は、 *string* に一致する語を求め、ポイントから逆方向に検索を行ないます。この関数は、 `word-search-forward` と対応していますが、逆方向に検索し、通常は一致した文字列の先頭にポイントを置きます。

30.2 正規表現

『正規表現 (regular expression)』 (略して『regexp』) は、文字列の (ときには無限) 集合を示すパターンです。 regexp にたいする一致の検索は、非常に強力な操作です。この節では、regexp の書き方を説明し、次の節ではその検索のしかたを述べます。

30.2.1 正規表現の構文

正規表現は、その中のいくつかの文字が特殊な構文要素であり、それ以外のものは『通常の (ordinary)』文字であるという構文をもっています。通常の文字は、その文字自身にのみ一致するという単純な正規表現です。特殊文字は '.'、'*'、'+'、'?'、'['、']'、'^'、'$'、'\' です。特殊文字が新たに定義されることはありません。正規表現中のこのほかのすべての文字は、'\' がその前に置かれないかぎり、通常の文字となります。

たとえば、'f' は特殊文字ではありませんから通常の文字で、したがって 'f' は、文字列 'f' に一致し、そのほかのものには一致しない正規表現です (これは文字列 'ff' には一致しません)。同様に、'o' は 'o' のみに一致する正規表現です。

任意の二つの正規表現 a と b をつなぐことができます。 a と b を連結した結果は、 a がある文字列の頭の部分に一致し、 b がその文字列の残りの部分に一致する場合、その文字列に一致する正規表現になります。

簡単な例として、正規表現 'f' と 'o' を連結して正規表現 'fo' を得ることができます。これは文字列 'fo' だけに一致します。ここまではまだあたりまえの話です。何か強力なことを行なうためには、特殊文字を用いる必要があります。以下にそれらのリストを示します。

. (終止符) は、改行をのぞくどんな 1 文字にも一致する特殊文字です。連結を用いることで、'a' で始まり 'b' で終る任意の 3 文字の文字列に一致する 'a.b' のような正規表現を作ることができます。

* は、これだけでは構文要素になりません。これは先行する正規表現をできるかぎり繰り返すことを意味する後置演算子です。 'fo*' における '*' は 'o' に適用されるため、 'fo*' は一つの 'f' の後ろに任意個の 'o' が続くものに一致します。 0 個の 'o' も許します。 'fo*' は 'f' にも一致するのです。

'*' は常に、先行するできるかぎり小さくとった正規表現に適用します。このため 'fo*' は 'o' を繰り返しますが、 'fo' を繰り返すわけではありません。

一致処理では、連続して見つかるかぎりの繰り返しに '*' 要素が一致するように処理します。その後、残りのパターンの処理を続けます。それが失敗すると、 '*' 修飾された正規表現の一致のいくつかを捨てることで、パターンの残りに一致することが可能になる場合に、バックトラックが起こります。たとえば、文字列 'caaar' に 'ca*ar' を一致させる場合、最初 'a*' を三つの 'a' すべてに一致させようとします。しかし、残りのパターンは 'ar' であるのに、一致対象として残っているのは 'r' だけなので、この試みは失敗します。次の選択肢は 'a*' に対し二つだけ 'a' を一致させることです。この選択により、この正規表現の残りの部分の一致は成功します。

バックトラックがループする場合、ネストした繰り返し演算子は非常に遅くなります。たとえば、正規表現 '\(x+y*\)*a' を文字列 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxz' に照合すると、何時間もかかるかもしれません。この遅さは、Emacs が 35 個の 'x' の可能な組み合わせを、結局どれ一つとしてうまくいかないことがわかるまで、すべて試さなければならないためです。正規表現が速く処理できるよう、ネストした繰り返しは注意深く調べてください。

+
: 先行する正規表現が少なくとも 1 回は一致しなくてはいけない、という点をのぞき '*' に似た後置演算子です。このため、たとえば、'ca+r' は文字列 'car' や 'caaaar' には一致しますが、文字列 'cr' には一致しません。一方 'ca*r' はこの三つの文字列すべてに一致します。

?
: 先行する正規表現が 1 回一致するかあるいはまったく一致しなくてもよい、という点をのぞき '*' に似た後置演算子です。このため、たとえば、'ca?r' は 'car' にも 'cr' にも一致しますが、それ以外のものには一致しません。

[...]
: '[' は、']' で終る『文字集合 (character set)』を始めます。一番単純な場合、二つの角括弧の間にある文字群が集合を構成します。このため、'[ad]' は 1 個の 'a' か 1 個の 'd' に一致し、'[ad]*' は 'a' もしくは 'd' だけからなる文字列 (空文字列も含む) に一致します。これにより 'c[ad]*r' は 'cr'、'car'、'cdr'、'caddaar' などに一致することになります。

通常の正規表現における特殊文字は、文字集合の中では特別扱いしません。文字集合の中では、まったく別の特殊文字の集合 ']'、'-'、'^' を用います。

文字の範囲を表わすのに '-' を使います。'-' を二つの文字の間に書くことで文字の範囲を表わします。したがって、'[a-z]' はすべての小文字に一致します。範囲と個々の文字は、どのように混じっていてもかまいません。'[a-z$%.]' は、小文字か '$' か '%' か '.' に一致します。

']' を文字集合の中に含むには、それを最初の文字としなくてはいけません。たとえば、'[]a]' は ']' か 'a' に一致します。'-' を含むには、'-' を集合の最初の文字とするか範囲の直後に置きます ('-' の置き場所を作るため、独立した (つまり範囲でない)1 個の文字 *c* を範囲 '*c*-*c*' と入れ替えるといいでしょう)。'-' と ']' だけを含む集合を書くには、'[]-]' あるいは '[]---]' とします (原著では「できません」とあります)。

集合に '^' を含めるには、それを集合の最初の文字にならないようにします。

[^ ...]
: '[^' は、指定した文字以外の文字に一致する『文字補集合 (complement character set)』を始めます。このため '[^a-z0-9A-Z]' は、英文字や数字以外のどの文字にも一致します。

文字集合の '^' は、最初の文字として使われないかぎり特別扱いされません。'^' に続く文字は、あたかもそれが最初のものであるかのように扱います (したがってそこでは '-' や ']' を特別扱いしません)。

改行が一致しない文字の一つとして記述されてないかぎり、文字補集合は改行に一致します。

`^` は、空文字列に一致する特殊文字ですが、これは一致が行なわれるテキスト中の行頭でのみ一致します。さもないと、これは何にも一致しません。このため、'`^foo`' は行頭にある '`foo`' に一致します。

バッファにたいしてでなく文字列にたいして一致を行なうとき、'`^`' は文字列の先頭か改行文字 '`\n`' の後に一致します。

`$` は、'`^`' に似ていますが、行末でのみ一致します。このため、'`x+$`' は行末にある一つ以上の '`x`' からなる文字列に一致します。

バッファにたいしてでなく文字列にたいして照合を行なうとき、'`$`' は文字列の末尾か改行文字 '`\n`' の前に一致します。

`\` は、二つの働きをします。それは ('`\`' を含む) 特殊文字を quote するのと、追加された特殊構文要素を導入することです。

'`\`' は特殊文字を quote するため、'`\$`' は '`$`' にのみ一致する正規表現であり、'`\[`' は '`[`' にのみ一致する正規表現であるなどなど、ということになります。

'`\`' は Lisp 文字列 (2.3.8 節「文字列型」 p.28 参照) のリード構文で特別な意味があり、'`\`' で quote しなければならないことに注意してください。たとえば、'`\`' 文字と一致する正規表現は '`\\`' です。 2 文字の '`\\`' を含む Lisp の文字列を書くとき、 Lisp 構文では各 '`\`' を別な '`\`' で quote する必要があります。したがって、'`\`' と一致する正規表現のリード構文は `"\\\\"`です。

注意: 過去との互換性のため、特殊文字はその特別な意味が意味をもたない文脈では普通の文字として扱われます。たとえば、'`*foo`' では '`*`' の作用すべき先行正規表現がないため、'`*`' を通常の文字として扱います。この動きに依存するのはよいことではありません。それがどこに現われるにせよ、特殊文字は quote しておいたほうがよいでしょう。ほとんどの場合、任意の文字が後続する '`\`' は、その文字にのみ一致します。しかしながら、いくつか例外が存在します。それは '`\`' が先行するときに特別な構文要素になる文字の場合です。このような文字は、それだけの場合、常に通常の文字になります。以下にこの '`\`' 構成要素の表を示します。

`\|` は、選択肢を示します。'`\|`' が間にある二つの正規表現 a と b は、 a あるいは b に一致するものに一致する正規表現を構成します。

このため、'`foo\|bar`' は '`foo`' にも '`bar`' にも一致しますが、それ以外の文字列には一致しません。

'`\|`' は、その前後のできるだけ大きくとった正規表現に適用します。それを囲む '`\( ... \)`' の一団のみが、'`\|`' による分割の勢力を限ることができます。

複数の '`\|`' を処理する完全なバックトラックの機能があります。

`\( ... \)` は三つの目的をもつ分割の構文要素です。

1. ほかの操作に対し、'`\|`' 選択肢の集合を囲みます。よって '`\(foo\|bar\)x`' は、'`foox`' にも '`barx`' にも一致します。
2. '`*`' のような後置演算子が作用する正規表現を囲みます。よって '`ba\(na\)*`' は、文字列 '`na`' の (0 回以上の) 繰返しをもつ '`bananana`' などに一致します。

3. 将来参照するため一致した部分文字列を記録します。

この最後の用法は括弧による分割という考えからきたものではありません。両方の意味が実際には競合しないため、同じ'\(... \)'構文要素の第2の意味として、たまたまこれに与えた機能となっています。次にこの機能を説明します。

digit
: *digit*番目に出てきた'\(... \)'構文要素で一致したのと同じテキストと一致します。

いい換えると、'\(... \)'構文要素の後、この構成に一致したテキストの始めと終りを覚えます。'\'の後に*digit*を続けたものをこの後の正規表現で用い、*digit*番目に'\(... \)'構成に一致したものが何であれ、それと同じテキストに一致させることができます。

正規表現中に現われる'\(... \)'構文要素に一致した最初の九つの文字列には、その開き括弧が正規表現中に現われた順に1から9の番号を割振ります。対応する'\(... \)'構成に一致するテキストは、'\1'から'\9'で引用することができます。

たとえば、'\(.*\)\1'は二つの同じものからなる改行を含まないすべての文字列に一致します。'\(.*\)'は任意の頭の半分に一致し、それに続く'\1'はそれとまったく同じテキストに一致しなくてはいけません。

\w
: は、語を構成するすべての文字に一致します。エディタの構文テーブルで、どの文字が語を構成する文字かを決定します。第31章「構文テーブル」p.541を参照してください。

\W
: は、語を構成しないすべての文字に一致します。

\s*code*
: は、その構文が*code*であるすべての文字に一致します。*code*は構文符号を示す文字です。'w'は語を構成するものにたいするもの、'-'は空白にたいするもの、'('は開き括弧にたいするもの、などなどです。構文符号と対応する文字の一覧は第31章「構文テーブル」p.541を参照してください。

\S*code*
: は、その構文が*code*でないすべての文字に一致します。

以下の正規表現の構文要素は、空文字列に一致します。つまり、文字には当たりません。しかし、一致は文脈には依存します。

\\`
: は、それがバッファや、照合する文字列の先頭にある場合のみ、空文字列に一致します。

\\'
: は、それがバッファや、照合する文字列の末尾にある場合のみ、空文字列に一致します。

\=
: は、それがポイントの位置にある場合のみ、空文字列に一致します。(この構文要素は文字列にたいしての照合では定義されません。)

\b
: は、それが語頭か語尾にある場合のみ、空文字列に一致します。このため'\bfoo\b'は独立した単語として現われた'foo'に一致します。'\bballs?\b'は、独立した単語としての'ball'にも'balls'にも一致します。

`\B`　は、それが語頭にも語尾にもない場合、空文字列に一致します。

`\<`　は、それが語頭にある場合のみ、空文字列に一致します。

`\>`　は、それが語尾にある場合のみ、空文字列に一致します。

すべての文字列が、正しい正規表現というわけではありません。たとえば、不釣り合いな角括弧は不正ですし (‘[]]’ といった例外はありますが)、単一の ‘\’ で終る文字列も不正です。不正な正規表現を検索関数のいずれかに渡した場合は、 invalid-regexp エラーが通知されます。

regexp-quote *string* Function

この関数は、 *string* に完全に一致しほかのものに一致しない正規表現の文字列を返します。これで正規表現を受けつける関数の呼出しで、完全な文字列の一致を要求することができます。

```
(regexp-quote "^The cat$")
     ⇒ "\\^The cat\\$"
```

完全な文字列の一致と正規表現で記述した文脈との組み合せに、 regexp-quote を使うということもできます。次の例では、空白で囲んだ string という値の文字列を検索しています。

```
(re-search-forward
 (concat "\\s-" (regexp-quote string) "\\s-"))
```

30.2.2 複雑な正規表現の例

以下に、文末とそれに続く空白を認識するのに Emacs が用いている複雑な regexp を示します。これは変数 sentence-end の値です。

まずこの regexp を、スペースとタブ文字が区別できるようになる Lisp 構文の文字列として示すことにします。 Lisp 構文において、文字列定数は二重引用符で始まり二重引用符で終ります。 ‘\"’ は文字列の一部としての二重引用符を示し、 ‘\\’ は文字列の一部としてのバックスラッシュを、 ‘\t’ はタブを、 ‘\n’ は改行を示します。

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\|  \\)[ \t\n]*"
```

一方、変数 sentence-end を評価すると、以下のものが得られます。

```
sentence-end
⇒
"[.?!][]\"')}]*\\($\\| $\\|  \\|  \\)[
]*"
```

ここでは、タブと改行はそのまま出力されます。

この正規表現は、連続した四つの部分から構成されており、以下のように解読することができます。

`[.?!]`　第 1 の部分は終止符、疑問符、感嘆符の 3 文字いずれかに一致する文字集合です。一致はこの 3 文字の内のいずれか 1 文字で始まらなければなりません。

`[]\"')}]*` 第 2 の部分は、 0 回かそれ以上繰り返される任意の閉じ括弧や引用符に一致し、これが終止符、疑問符、感嘆符に後続することになります。 `\"`は Lisp 構文で文字列中の二重引用符です。末尾の ‘`*`’ は直前の正規表現 (この場合は文字集合) が、 0 回以上繰り返すことを示します。

`\\($\\| $\\|\t\\|  \\)`
第 3 の部分は、文末に続く空白、すなわち、行末、タブ、二つのスペースに一致します。二重のバックスラッシュは括弧と縦棒が正規表現の構文であることを示します。括弧が分割を制限し、縦棒が選択肢を分割しています。ドル記号を行末との一致に使っています。

`[ \t\n]*` 最後の部分は、文末に必要な最低限の空白に続く、余分な任意の空白です。

30.3 正規表現の検索

GNU Emacs では、次にある regexp への一致を求め、段階的もしくは普通の検索を行なうことができます。段階的検索のコマンドについては *The GNU Emacs Manual* の “Regular Expression Search” 節を参照してください。ここで述べるのは、プログラムに便利な検索関数だけです。一番重要なのが `re-search-forward` です。

re-search-forward *regexp* &optional *limit noerror repeat* Command

この関数は、カレント・バッファで正規表現 *regexp* に一致するテキスト文字列を求め、順方向に検索します。これは *regexp* に一致しないテキストをすべて飛び越し、最初に一致したテキスト文字列の終りにポイントを置きます。ポイントの値を返します。

もし *limit* が非 `nil` (これはカレント・バッファ内の位置でなくてはいけません) の場合、これは検索の上限になります。この位置を越えるものとは照合しません。

検索が失敗したときの動作は、 *noerror* の値によります。もし *noerror* が `nil` ならば、 `search-failed` エラーが通知されます。もし *noerror* が `t` の場合、 `re-search-forward` は何もせず `nil` を返します。もし *noerror* が `nil` でも `t` でもない場合、 `re-search-forward` はポイントを *limit*(またはバッファの終り) まで動かし、 `nil` を返します。

もし *repeat* が与えられた場合 (正数でなければなりません)、その回数だけ (毎回前の一致の末尾から) 検索を繰り返します。もしこの連続検索が成功したら、ポイントを動かし新しい値を返します。さもないと検索は失敗します。

次の例において、ポイントは最初 ‘`T`’ の前に置かれています。検索の呼出しを評価した後、ポイントを行末 (‘`hat`’ の `t` と改行との間) に動かします。

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------
```

```
(re-search-forward "[a-z]+" nil t 5)
     ⇒ 27

---------- Buffer: foo ----------
I read "The cat in the hat⋆
comes back" twice.
---------- Buffer: foo ----------
```

re-search-backward *regexp* &optional *limit noerror repeat* Command

この関数は、カレント・バッファで正規表現 *regexp* に一致するテキスト文字列を求め、ポイントから逆方向に検索し、見つかった最初の文字列の先頭にポイントを置きます。

この関数は、 `re-search-forward` と似ていますが、単純な鏡像ではありません。 `re-search-forward` は照合の開始が出発点になるべく近い箇所を見つけます。もし `re-search-backward` が完全な鏡像であれば、照合の終了がなるべく近い箇所を見つけるでしょう。しかしながら、これは照合の開始がなるべく近い箇所を見つけます。この理由は、与えられた点からの正規表現の照合が常に先頭から末尾へ、そして指定した先頭位置から開始するためです。

`re-search-forward` の真の鏡像には、末尾から先頭に regexp を照合させる特殊な機能が必要です。これは面倒で実装するほどの価値はありません。

string-match *regexp string* &optional *start* Function

この関数は、 *string* 中で最初の正規表現 *regexp* の一致開始場所の添字か、一致しない場合 `nil` を返します。 *start* が非 `nil` の場合、検索は *string* 中のその添字から行ないます。

たとえば、

```
(string-match
 "quick" "The quick brown fox jumped quickly.")
     ⇒ 4
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27
```

文字列の最初の文字の添字は 0 で、次の文字の添字は 1、などなどとなります。

この関数が戻った後で、一致部分を過ぎた最初の文字の添字は `(match-end 0)` で得られます。30.6 節「一致データ」p.534 を参照してください。

```
(string-match
 "quick" "The quick brown fox jumped quickly." 8)
     ⇒ 27

(match-end 0)
     ⇒ 32
```

looking-at *regexp* Function

この関数は、カレント・バッファ内のポイントの直後のテキストが正規表現 *regexp* に一致するか否かを調べます。「直後」ということは、検索が「固定」(“anchor”) され、ポ

イントに続く最初の文字から一致が成功しなくてはいけない、ということです。そうなる場合、結果は `t` で、そうならない場合 `nil` になります。

ポイントは動かしませんが、 `match-beginning` と `match-end` でアクセスできる一致データを更新します。 30.6 節「一致データ」p.534 を参照してください。

次の例において、ポイントは ‘T’ の直前にあります。別の場所に置かれていたとすると、結果は `nil` になったはずです。

```
---------- Buffer: foo ----------
I read "⋆The cat in the hat
comes back" twice.
---------- Buffer: foo ----------

(looking-at "The cat in the hat$")
     ⇒ t
```

30.4 POSIX 正規表現の検索

通常の正規表現の関数群は、 ‘\|’ と繰り返しの構文要素を処理するために、必要に応じバックトラックを行ないますが、続行は少しでも一致が見つかる間だけでしかありません。このとき関数は成功し最初の一致を返します。

この節では、正規表現照合の POSIX 標準で規定された完全なバックトラックを行なう代替えの検索関数について述べます。これはすべての可能性を試し、すべての一致を見つけるまでバックトラックを継続するので、 POSIX で要求されているように、最長の一致を報告することができます。これはとても遅いので、本当に最長の一致が必要なときだけこの関数を使うようにしてください。

Emacs バージョン 19.29 以前には、これらの関数はありません。前述の関数が完全な POSIX バックトラックを実装していました。

posix-search-forward *regexp* &optional *limit noerror repeat* — Function

これは `re-search-forward` に似ていますが、 POSIX 標準の正規表現照合で規定されている完全なバックトラックを行ないます。

posix-search-backward *regexp* &optional *limit noerror repeat* — Function

これは `re-search-backward` に似ていますが、 POSIX 標準の正規表現照合で規定されている完全なバックトラックを行ないます。

posix-looking-at *regexp* — Function

これは `looking-at` に似ていますが、 POSIX 標準の正規表現照合で規定されている完全なバックトラックを行ないます。

posix-string-match *regexp string* &optional *start* — Function

これは `string-match` に似ていますが、 POSIX 標準の正規表現照合で規定されている完全なバックトラックを行ないます。

30.5 検索と置換

perform-replace *from-string replacements query-flag regexp-flag delimited-flag* &optional *repeat-count map* — Function

この関数は、 `query-replace` や関連するコマンドの本体です。これは *from-string* の出現箇所を検索し、一部あるいは全部を置換します。もし *query-flag* が `nil` ならば、すべての出現箇所を置換します。そうでないとき、各出現箇所についてユーザにどうしたらいいか尋ねます。

もし *regexp-flag* が非 `nil` ならば、 *from-string* は正規表現とみなします。さもないと文字どおりの照合になります。もし *delimited-flag* が非 `nil` ならば、語境界で囲まれた置換だけが対象になります。

引数 *replacements* は出現したものに置き換えるものを指定します。もしこれが文字列なら、この文字列を使います。文字列のリストの場合、これを周期的に使います。

もし *repeat-count* が非 `nil` ならば、整数でなければなりません。このとき *replacements* 中の次の文字列に進む前に、各文字列を何回使うのかを指定します。

通常は、キーマップ `query-replace-map` が、問い合わせにたいするユーザのとりうる反応を定義します。引数 *map* が非 `nil` ならば、 `query-replace-map` の代わりに使うキーマップとなります。

query-replace-map — Variable

この変数は、 `y-or-n-p` や `map-y-or-n-p` 同様、 `query-replace` や関連する関数におけるユーザの有効な反応を定義する特殊なキーマップです。通常のそれと 2 点異なるところがあります。

- 「キー・バインド」はコマンドではなく、このマップを使用する関数でのみ意味をもつただのシンボルです。
- 前置キーはサポートしていません。各キー・バインドは単一イベントのキー・シーケンスでなければなりません。これは関数が入力を取るときキー・シーケンスの読込みを使っていないからです。その代わり、単一のイベントを読込み「手で」参照を行ないます。

次は `query-replace-map` で意味のある「バインド」です。いくつかは `query-replace` 一族でのみ意味をもちます。

`act` 対象の動作をとります。すなわち「はい」の返事です。

`skip` 対象の動作をとりません。すなわち「いいえ」の返事です。

`exit` 今の問いに「いいえ」で答え、以降の問いにもすべて「いいえ」で答えると仮定します。

`act-and-exit`
今の問いには「はい」で答え、以降の問いにはすべて「いいえ」で答えると仮定します。

`act-and-show`
: 今の問いには「はい」で答えますが、結果を示します。次の問いには進みません。

`automatic`
: 今の問いと以降の問いに「はい」で答え、以降の対話は行ないません。

`backup`
: 以前、質問を行なった箇所に戻ります。

`edit`
: 通常とられる動作の代わりに、再帰編集にはいって処理します。

`delete-and-edit`
: 対象のテキストを消去し、それを置換する再帰編集にはいります。

`recenter`
: ウィンドウの中央に寄せて再表示を行ない、同じ問いを繰り返させます。

`quit`
: ただちにとりやめます。 `y-or-n-p` と関連した関数だけがこの答えを使います。

`help`
: ヘルプを表示して、再び尋ねます。

30.6 一致データ

Emacs は、正規表現の検索で見つけたテキストの開始と終了の位置をおぼえています。これは、たとえば、 Rmail メッセージの日付のような複雑なパターンの検索ができ、そして、パターンの制御下にある一致の部分的取り出しができる、という意味です。

一致データは通常、一番最後の検索だけを記憶するので、後で参照する検索と一致データの利用の間で、別な検索をしないように注意深く、気を配らなければなりません。中間の検索が避けられない場合、上書きをされないように、一致データの保存と復帰を前後で行なわなければなりません。

30.6.1 単純な一致データのアクセス

この節では、直前の検索照合操作で何が一致したか見つけるための、一致データの使い方を説明します。

一致テキストの全体と、括弧でくくった特定の正規表現の部分式 (subexpression) とを求めることができます。以下の関数の *count* 引数がどちらなのかを指定します。もし *count* が 0 ならば、一致全体を求めます。もし *count* が正ならば、どの部分式が必要かを指定します。

正規表現の部分式は、エスケープした括弧 ‘`\(...\)`’ でくくることを思い出してください。*count* 番目の部分式は、正規表現全体の先頭から ‘`\(`’ の出現順に数えます。最初は 1、次は 2、などなどとなります。正規表現だけに部分式があります。単純な文字列検索で有効な情報は、一致全体です。

match-string *count* &optional *in-string* Function

この関数は、最後の検索照合操作の一致テキストを文字列として返します。もし *count* が 0 ならば、テキスト全体を返し、もし *count* が正ならば、 *count* 番目の括弧でくくった箇所に対応する部分正規表現を返します。もし *count* が範囲外の場合や部分式が一致しなかった場合、値は `nil` です。

もしそのような最後の操作が文字列にたいする `string-match` の場合、同じ文字列を引数 *in-string* として渡さなければなりません。そうでない場合、つまりバッファにたいする検索照合では、 *in-string* を省略するか `nil` を与えなければなりません。その際、`match-string` を呼ぶときのカレント・バッファは、検索照合を行なったバッファと同じくなるようにしてください。

match-beginning *count* Function

この関数は、最後の正規表現検索で一致したテキストやその部分式の開始位置を返します。

count が 0 の場合、値は *regexp* 全体に一致したテキストの開始位置です。そうでない場合、 *count* は正規表現のどの部分式の開始位置を返すかを指定し、関数の値は部分式への一致の開始位置です。

一致の起きなかった '`\|`' 選択肢の内部の部分式にたいしての値は、 `nil` になります。

match-end *count* Function

この関数は、 `match-beginning` と似ていますが、最後の正規表現検索で一致したテキストの開始位置ではなく、終了位置を返します。

一致データの使用例です。コメントはテキスト中の位置を示します。

```
(string-match "\\(qu\\)\\(ick\\)"
              "The quick fox jumped quickly.")
              ;0123456789
     ⇒ 4

(match-string 0 "The quick fox jumped quickly.")
     ⇒ "quick"
(match-string 1 "The quick fox jumped quickly.")
     ⇒ "qu"
(match-string 2 "The quick fox jumped quickly.")
     ⇒ "ick"

(match-beginning 1)       ; ‘qu’ の一致は
     ⇒ 4                  ;    添字 4 から始まる。

(match-beginning 2)       ; ‘ick’ の一致は
     ⇒ 6                  ;    添字 6 から始まる。

(match-end 1)             ; ‘qu’ との一致は
     ⇒ 6                  ;    添字 6 で終る。

(match-end 2)             ; ‘ick’ との一致は
     ⇒ 9                  ;    添字 9 で終る。
```

次の例において、ポイントは行頭に置かれています。検索後のポイントは、スペースと語 '`in`' との間になります。一致全体の開始位置は、バッファの 9 番目の文字 ('`T`') であり、最初の部分式の一致の開始位置は、 13 番目の文字 ('`c`') です。

```
(list
  (re-search-forward "The \\(cat \\)")
  (match-beginning 0)
  (match-beginning 1))
    ⇒ (9 9 13)

---------- Buffer: foo ----------
I read "The cat ⋆in the hat comes back" twice.
        ^   ^
        9   13
---------- Buffer: foo ----------
```

(この場合では、返される添字はバッファ内位置です。バッファの最初の文字は 1 になります。)

30.6.2 一致したテキストの置換

この関数は、最後の検索で一致したテキストを *replacement* で置き換えます。

replace-match *replacement* &optional *fixedcase literal string* Function

この関数は、最後の検索で一致したバッファのテキスト (もしくは *string*) を *replacement* で置き換えます。

最後の検索をバッファで行なった場合、 *string* には `nil` を指定します。このとき `replace-match` はバッファを編集し置換を行ないます。この関数は、ポイントを置換テキストの終りに置いて、 `t` を返します。

検索したのがバッファでなく文字列だった場合、同じ文字列を *string* に渡します。すると `replace-match` は新しい文字列を構成してそれを返します。

fixedcase が非 `nil` の場合、それで置き換えるテキストの大文字小文字の違いは変更しません。そうでない場合、置き換えられるテキストの capitalization にしたがい、置き換えるテキストを大文字小文字の間で変換します。元のテキストがすべて大文字の場合、置き換えるテキストを大文字に変換します。もし元のテキストの最初の語が大文字で始まっている (capitalized) なら、置き換えるテキストの最初の語を大文字で始め (capitalize) ます。もし元のテキストが 1 語しかなく、その語が 1 文字の大文字であるなら、 `replace-match` はこれをすべて大文字とは見ずに、まず最初の語が大文字で始まっている (capitalized) とみなします。

もし `case-replace` が `nil` ならば、 *fixed-case* の値にかかわらず、大文字小文字変換は行ないません。 30.7 節「検索と大文字小文字の区別」 p.538 を参照してください。

literal が非 `nil` の場合、大文字小文字の変更を生じうる点は除いて *replacement* をそのまま挿入します。もしこれが (デフォルト値の)`nil` である場合、文字 ‘\’ を特別扱いします。もし *replacement* に ‘\’ が現われる場合、以下の列のいずれかの一部分でなくてはいけません。

‘`\&`’ ‘`\&`’ は、置き換えられたテキスト全体になります。

'\$n$' n を数字とした '\$n$' は、もとの regexp における n 番目の部分式と一致したテキストになります。部分式とは '\(...\)' で区切られた正規表現のことです。

'\\' '\\' は置換テキストの中で 1 個の '\' になります。

30.6.3 一致データ全体のアクセス

関数 `match-data` と `set-match-data` は、一度に一致データを読み書きできます。

match-data Function

この関数は、最後の検索が一致したテキストに関するすべての情報を含むリストを新しく返します。要素 0 は、正規表現全体への一致の開始位置です。要素 1 は、表現全体への一致の終了位置です。次の二つの要素は、最初の部分式への一致の開始、終了位置で、以下同様です。一般に要素番号 $2n$ は (`match-beginning` n) に対応し、番号 $2n+1$ は (`match-end` n) に対応します。

バッファにたいする一致の場合、全要素はマーカか `nil` で、 `string-match` で文字列の一致をした場合、全要素は整数か `nil` です (Emacs バージョン 18 やそれ以前のバージョンでは、整数 0 の場合をのぞき、文字列にたいしてもマーカを使っていました)。

いつものように、検索の呼出しとその検索の一致データをアクセスしようとしている `match-data` の呼出しとの間に、別の検索がはいり込むようなことがあってはいけません。

```
(match-data)
     ⇒  (#<marker at 9 in foo>
          #<marker at 17 in foo>
          #<marker at 13 in foo>
          #<marker at 17 in foo>)
```

set-match-data *match-list* Function

この関数は、一致データを `match-list` の要素に設定します。 `match-list` は、以前に `match-data` の呼出しが返した値のリストでなくてはいけません。

もし *match-list* が存在しないバッファを参照していても、エラーにはなりません。一致データが、無害ではありますが、意味のないものになるだけです。

`store-match-data` は `set-match-data` の別名です。

30.6.4 一致データの保存と復帰

検索を行なうかもしれない関数を呼び出すとき、もし前の検索の一致データを後で使うために保存したいなら、その呼出しの前後で一致データの保存と復帰をする必要があるかもしれません。次の例は、一致データの保存をしないために起きる問題を示しています。

```
(re-search-forward "The \\(cat \\)")
     ⇒ 48
(foo)                       ; 多分 foo は、
                            ;    検索を行なう。
(match-end 0)
     ⇒ 61                   ; 予期せぬ結果。 48 にあらず！
```

一致データの保存と復帰は save-match-data で行なうことができます。

save-match-data *body...* Macro

この特殊形式は、前後で一致データの保存復帰を行ないつつ、 *body* を実行します。

set-match-data と match-data で特殊形式 save-match-data をまねることができます。これは Emacs 18 でも動作するコードを書くとき便利です。それには以下のようにします。

```
(let ((data (match-data)))
  (unwind-protect
      ...   ; もとの一致データを変えてしまうかもしれない。
    (set-match-data data)))
```

プロセス・フィルター関数 (33.9.2 節「プロセス・フィルタ関数」 p.571 参照) とプロセスの見張り (33.10 節「監視関数: プロセスのステータス変化の検出」 p.574 参照) を実行するとき、Emacs は自動的に一致データの保存と復帰を行ないます。

30.7 検索と大文字小文字の区別

デフォルトでは、 Emacs における検索は検索対象のテキストにおける大文字小文字の区別は無視します。 'FOO' の検索を指示した場合、 'Foo' も 'foo' も照合の対象とみなします。これは regexp そして特に文字集合 (での検索) にも適用します。したがって '[aB]' は、 'a' にも 'A' にも 'b' にも 'B' にも一致します。

この機能がいらないとき、変数 case-fold-search を nil にします。そうすると、大文字小文字の違いを含め、すべての文字が厳密に一致しなくてはいけなくなります。これはバッファローカルな変数です。この変数の変更は、カレント・バッファにたいしてのみ影響を及ぼします (10.9.1 節「バッファローカルな変数 (紹介)」 p.140 を参照してください)。そのかわりに、まだ case-fold-search を優先させていないバッファでのデフォルト値の、 default-case-fold-search の値を変えることもできます。

ユーザ・レベルでの段階的検索機能では、別な方法で大文字小文字の区別をしています。小文字がきたときは両方を探しますが、大文字がきたときは大文字だけしか探しません。しかし、これは Lisp 関数の使う検索関数とは関係のないことです。

case-replace User Option

この変数は、置き換え関数で大文字小文字の違いを保存すべきか否かを定めます。もし変数が nil ならば、置き換えテキストをそのまま使います。非 nil の値は、置き換えるテキストの大文字小文字を、置き換えられるテキストにあわせて変換することを意味します。

関数 `replace-match` で実際にこの変数が使われます。 30.6.2 節「一致したテキストの置換」p.536 を参照してください。

case-fold-search User Option

このバッファローカル変数は、検索が大文字小文字の違いを無視すべきか否かを定めます。もしこの変数が `nil` の場合、大文字小文字の違いを無視しません。そうでない場合は違いを無視します。

default-case-fold-search Variable

この変数の値は、まだそれに優先させていないバッファにおける `case-fold-search` のデフォルト値です。これは `(default-value 'case-fold-search)` と同じです。

30.8 編集用の標準正規表現

この節では、編集上のある目的で使用される正規表現を保持する変数について述べます。

page-delimiter Variable

これはページをわける行頭を示す regexp です。デフォルト値は `"^\014"` (つまり `"^^L"`あるいは `"^\C-l"`) です。改ページ文字で始まる行と一致します。

次の二つの正規表現は、一致が常に行頭から始まると仮定すべきではありません。照合位置を固定するのに ‘^’ を使うべきではありません。段落コマンドは大体、行頭でだけ一致を検査するため ‘^’ は余計です。非 0 の左マージンがあるときは、左マージン後から始まる場合の一致も受けつけるべきです。この場合 ‘^’ は誤りでしょう。しかし、左マージンがないモードでは、 ‘^’ は無害です。

paragraph-separate Variable

これは段落をわける行の先頭を示す正規表現です (これを変更する場合、 `paragraph-start` も変更する必要があります)。デフォルト値は左マージンの後にスペース、タブ、改ページだけしかない行に一致する、`"[ \t\f]*$"`です。

paragraph-start Variable

段落を始めるかもしくは段落をわける行の先頭を示す正規表現。デフォルト値は (左マージンの後に) スペース、タブ、改ページだけしかない行に一致する、`"[ \t\n\f]"`です。

sentence-end Variable

これは文末を示す正規表現です (段落の境界も、必ず文を終らせます)。デフォルト値は

```
"[.?!][]\"')}]*\\($\\| $\\|\t\\| \\)[ \t\n]*"
```

これは終止符、疑問符、感嘆符の後に省略可能な閉じ括弧、そしてその後にタブ、スペース、改行がくるものです。

この正規表現の詳細については、 30.2.2 節「複雑な正規表現の例」p.529 を参照してください。

31 構文テーブル

『構文テーブル (syntax table)』には、それぞれの文字の構文上の働きが記述されています。この情報は、解析コマンドや、複雑な移動のコマンドを制御したり、語、シンボル、構文要素の始まりと終わりを決定するために使用されます。また、現在の構文テーブルは、この章でふれる関数のほかにも、語移動関数 (27.2.2 節「単語単位での移動」p.459 参照) やリスト移動関数 (27.2.6 節「釣り合った式をまたがる移動」p.464 参照) の意味の制御も行なっています。

31.1 構文テーブルの基本的な考え方

構文テーブルは、 256 個の要素からなるベクタです。そして、 1 バイト (8 ビット) で表すことのできる 256 文字に対して、一つのエントリを持っています。おのおのの要素は整数であり、その文字の構文をコード化したものです。

構文テーブルは、 Lisp リーダのためにではなく、テキストを操作するためにだけ、利用されます。 Lisp 式を読むときには、 Emacs Lisp は、最初から組み込まれた構文規則を使います。また、これらの規則は変更できません。

バッファには、それぞれ主モードがあり、主モードごとに、さまざまな文字の構文クラスに独自の考え方を持っています。たとえば、 Lisp モードでは、 ';' という文字は、コメントの始まりとなりますが、 C モードでは、文の終わりを表します。このような変化に対応するために、Emacs ではバッファごとにローカルの構文テーブルを選択します。特に、主モードでは、おのおの独自のテーブルをもっていて、そのモードを使うバッファごとにそれらテーブルが利用可能になっています。このテーブルを変更すると、あるモードを次に利用するときの文法ばかりでなく、すべての同じモードのバッファ上での構文を変更すことになります。しばしば、似ているモードは一つのテーブルを共有しています。どのように構文テーブルをセットアップするかの例は、 20.1.2 節「主モードの例」p.320 を参照してください。

構文テーブルは、いくつかの文字については標準のテーブルから情報を継承することができます。また、残りの文字について独自に情報を指定することもできます。この構文クラスの「継承」とは、「標準の構文テーブルから、その文字の構文を継承する」ことを意味しています。ほとんどの主モードの構文テーブルは、 0 から 31 までと 128 から 255 までの文字コードを引き継いでいます。これは、 128 から 255 の間に追加のアルファベット文字をもつ ISO Latin-1 のような文字集合では便利です。こういった文字について、標準構文テーブルだけを変更すると全ての主モードに影響があります。

syntax-table-p *object* Function

この関数は、 *object* が 256 の要素をもつ長さのベクタであれば、 `t` を返します。つまり、そのベクタが構文テーブルであるかもしれないということを意味します。しかし、この検査だけでは、なかに何が含まれていても、長さが 256 のベクタであれば、構文テーブルであると判断されます。

31.2 構文記述子

この節では文字の構文を示す構文クラスやフラグについて、そしてそれらがどうやって『構文記述子 (syntax descriptor)』として表されるかについて述べます。構文記述子とは `modify-syntax-entry` に渡して望みの構文を指定するための Lisp 文字列です。

Emacs には、多くの『構文クラス (syntax classes)』が定義してあります。構文テーブルは、各文字を一つのクラスに置きます。ある構文テーブルにおけるある文字のクラスと、ほかの構文テーブルにおける同じ文字のクラスとに、関連がある必要はありません。

各クラスはニモニック文字で指示され、このニモニック文字はクラスを指定したい場合に、そのクラスの名前となります。通常この指示子はそのクラスでよく使われる文字になっています。とはいうものの、指示子としての意味は不変で、現在その文字がどんな構文を持っているかとも無関係です。

構文記述子は、構文クラスや、対応文字 (かっこの対応を調べるためだけに使われる)、フラグを記述するための Lisp 文字列です。最初の文字は、構文クラスの指示子になります。 2 番目の文字は対応文字で、使わなければ、スペースを置きます。その次には、望みのフラグ用文字がきます。対応文字やフラグが必要なければ、 1 文字だけで十分です。

たとえば、 C モードでは、文字 ‘`*`’ の記述子は ‘`. 23`’ となります (句読点、対応文字なし、コメント開始の 2 番目の文字、コメント終了の 1 番目の文字)。また、 ‘`/`’ のエントリは、 ‘`. 14`’ です (句読点、対応文字なし、コメントの開始の 1 番目の文字、コメント終了の 2 番目の文字)。

31.2.1 構文クラスのテーブル

ここでは、構文クラスのテーブルや、それらを表す文字、意味、使い方の例について述べます。

空白文字[くうはくもじ] Syntax class

『空白文字 (whitespace characters)』 (‘ ’や ‘`-`’ で指示する) は、シンボルや語をお互いから分離する働きがあります。ほとんどの場合、空白には構文上ほかに重要な意味はなく、空白が何文字あっても構文上は、 1 個の空白と同じ意味です。スペース、タブ、改行、改ページもたいていは空白とほぼ同じように分類されます。

語構成可能[ごこうせいかのう] Syntax class

『語構成可能 (word constituents)』 (‘`w`’ で指示する) は、普通の英単語の一部となるもので、たいていは、プログラム内の変数やコマンドの名前で使われます。すべての大文字や、小文字、数字が語構成可能として使われます。

シンボル構成可能[しんぼるこうせいかのう] Syntax class

『シンボル構成可能 (symbol constituents)』 (‘`_`’ で指示する) は、変数やコマンド名内で語構成可能と一緒に使われる特別な文字です。たとえば、 Lisp モードで使うシンボル構成可能クラスは、ある種の文字が、英単語の一部ではないが、シンボル名の一部になるかもしれないということを表します。これらの文字は、 ‘`$&*+-_<>`’ です。標準の C では、語構成可能以外でシンボルに使用できるのは、 (‘`_`’) だけです。

句読点文字[くとうてんもじ] Syntax class

『句読点文字 (punctuation characters)』(‘.’) は、英語における句読点と同じように使われます。また、プログラミング言語によっては、シンボルの区切りに使われることもあります。 Emacs Lisp モードを含めて、ほとんどのプログラミング言語モードは、シンボルあるいは語構成可能以外の僅かな文字には全てほかの使い道があるので、このクラスには文字がありません。

開き括弧文字[ひらきかっこもじ] Syntax class

閉じ括弧文字[とじかっこもじ] Syntax class

開きと閉じ『括弧文字 (parenthesis characters)』は、違ったものの対で、文や式を囲むために使われる文字です。こういうグループは、開き括弧文字で始まり、閉じ括弧文字で終わります。各開き括弧文字は、特定の閉じ括弧文字に対応しています。逆も同じです。普通、 Emacs は、閉じ括弧を挿入したとき、対応する開き括弧を一瞬示してくれます。 35.11 節「対応括弧の点滅表示」 p.616 を参照してください。

開き括弧のクラスは ‘(’ で、閉じ括弧のそれは、 ‘)’ で指示されます。

Text モードや C モードでは、対になる括弧には、 ‘()’、 ‘[]’、 ‘{}’ があります。 Emacs Lisp では、リストやベクタの区切り、 (‘()’ と ‘[]’) が、括弧文字として分類されています。

文字列 **quote**[もじれつ **quote**] Syntax class

『文字列 quote 文字 (String quote characters)』(‘"’ で指示する) は、 Lisp や C を含めて、多くの言語で文字列定数の区切りとして使われています。同じ文字列 quote 文字が、文字列の始まりと終わりに記述されます。そのように quote された文字列を入れ子にすることはできません。

Emacs の構文解析は、 1 文字列を 1 トークンとみなします。文字が通常もっている構文上の意味は、文字列中では抑止されます。

Lisp モードには、 2 種類の文字列 quote 文字があります。二重引用符 (‘"’) と縦棒 (‘|’) です。 ‘|’ は、 Emacs Lisp では、使われませんが、 Common Lisp では、使われます。 C 言語も、 2 種類の文字列 quote 文字があります。文字列のための二重引用符と、文字定数のための単一引用符 (‘'’) です。

英文テキストは、プログラミング言語でないので、文字列 quote 文字はありません。英文のなかで、引用符が使われることはありますが、引用符中の文字に、通常の構文上の属性をなくして欲しくはありません。

エスケープ[えすけーぷ] Syntax class

『エスケープ文字 (escape character)』(‘\’ で指示する) は、 C 言語の文字列や文字定数で使われているのと同様に、エスケープ・シーケンスを始めます。文字 ‘\’ は、 C と Lisp の両方でこのクラスに属します (C 言語では、それは、文字列の中でだけに使われますが、 C コード全体の中でこのように扱っても、障害を起こさないということがわかります)。

このクラスの文字は、 `words-include-escapes` が非 `nil` の場合、語の一部としてみなされます。 27.2.2 節「単語単位での移動」 p.459 を参照してください。

文字 quote[もじ quote] Syntax class

『文字 quote 文字 (character quote character)』(‘/’ で指示する) は、それに続いて記述される文字を quote します。 quote された文字は、通常の構文上の意味を失います。エスケープ文字との違いは、ただ直後の文字だけに効果があることです。

このクラスの文字は、 `words-include-escapes` が非 `nil` の場合、語の一部とみなされます。 27.2.2 節「単語単位での移動」 p.459 を参照してください。

TEX モードのバックスラシュために、このクラスは使われます。

対の区切り[ついのくぎり] Syntax class

『対の区切り文字 (paired delimiter characters)』(‘$’ で指示する) は、これによって囲まれた文字の構文上の属性が抑制されないことを除けば、文字列 quote 文字と同じです。この区切り文字は、現在、 TEX モードでのみで使われます (‘$’ は、数式モードの開始と終了を表します)。

式前置[しきぜんち] Syntax class

『式前置演算子 (expression prefix operator)』(‘’’ で指示する) は、式がこれに続いた場合、その式の一部としての構文上の演算子として使われます。 Lisp でのこれらの文字には、アポストロフィ、 ‘’’(quote に使われる)、カンマ、 ‘,’(マクロで使われる)、 ‘#’(あるデータ型のリード構文として使われる) が含まれます。

コメント開始[こめんとかいし] Syntax class
コメント終了[こめんとしゅうりょう] Syntax class

『コメント開始 (comment starter)』と『コメント終了 (comment ender)』文字は、さまざまな言語でコメントの範囲を定めるために使われます。これらのクラスは、それぞれ ‘<’ と ‘>’ で指示されます。

英文テキストには、コメント文字はありません。 Lisp では、セミコロン (‘;’) でコメントが始まり、改行、改ページで終わります。

継承[けいしょう] Syntax class

この構文クラスは、構文を示してはいません。この文字の構文を見つけるために、標準構文テーブルをみるようにと書かれています。この構文規則に対する指示子は、 ‘@’ です。

31.2.2 構文フラグ

構文テーブル中の文字に対するエントリには、クラスのほかにもフラグを含めることができます。六つのフラグを利用可能で、それらは、 ‘1’、 ‘2’、 ‘3’、 ‘4’、 ‘b’、 ‘p’ といった文字で表されます。

‘p’ を除くすべてのフラグは、複数文字からなるコメント区切りを表すために使われます。数字のフラグは、文字クラスに関連づいた構文上の属性に加えて、文字がコメント・シーケンスにもなりうることを示します。フラグは、 C モードの ‘*’ のような文字があるために、クラスからもほかのフラグからも独立です。 C モードの ‘*’ は、句読点文字であり、かつ、コメント開始シーケンス (‘/*’) の 2 番目の文字であり、かつ、コメント終了シーケンス (‘*/’) の最初の文字です。

文字 c のためのフラグは:

- '1' は、 c が 2 文字のコメント開始シーケンスの第 1 文字であることを意味します。
- '2' は、 c が先のシーケンスの第 2 文字であることを意味します。
- '3' は、 c が 2 文字のコメント終了シーケンスの第 1 文字であることを意味します。
- '4' は、 c が先のシーケンスの第 2 文字であることを意味します。
- 'b' は、コメント区切りとしての c が、別のコメント形式 "b" にも属していることを意味します。

  Emacs では、任意の 1 構文テーブルで同時に 2 種類のコメントの形式を利用することができます。これは、 C++ での利用を考慮したものです。コメントの形式のそれぞれは、独自のコメント開始シーケンスとコメント終了シーケンスを使います。各コメントは、どちらか一方の書き方で記述されている必要があります。もし、 "b" という形式のコメント開始シーケンスで始めた場合は、 "b" という形式のコメント終了シーケンスで終る必要があります。

  2 種類のコメント開始シーケンスは、同じ文字で始まっている必要があります。 2 番目の文字だけは、異なっていても構いません。 "b" 形式のコメント開始シーケンスの 2 番目の文字に、 'b' フラグで印をつけてください。

  コメント終了シーケンス (1 文字ないし 2 文字) は、最初の文字に 'b' フラグがセットしてあれば、 "b" 形式にあたります。そうでなければ、 "a" 形式になります。

  C++ に適したコメント構文のセッティングは、以下のようになります。

'/'	'124b'
'*'	'23'
newline	'>b'

  この表は、四つのコメント区切りシーケンスを定義しています。

  '/*' これは、 2 番目の文字 '*' に、 'b' フラグが定義されていないので、 "a" 形式のコメント開始シーケンスとなります。

  '//' これは、 2 番目の文字 '/' に、 'b' フラグが定義されているので、 "b" 形式のコメント開始シーケンスとなります。

  '*/' これは、最初の文字 '*' に、 'b' フラグが定義されていないので、 "a" 形式のコメント終了シーケンスとなります。

  newline これは、改行文字に、 'b' フラグが定義されているので、 "b" 形式のコメント終了シーケンスとなります。

- 'p' は、 Lisp 構文のために付加された前置文字を表します。この文字が、式と式の間に記述された場合は、空白と同じように扱われます。この文字が、式のなかで使われた場合は、通常の構文規則にしたがって処理されます。

  関数 `backward-prefix-chars` は、基本構文クラスが前置 ('`'`') である文字ばかりでなく、これらの文字も越えて後方に移動します。 31.4 節「移動と構文」 p.547 を参照してください。

31.3 構文テーブル関数

この節では、構文テーブルを作成したり、アクセスしたり、変更するための関数について述べます。

make-syntax-table Function

この関数は、新しい構文テーブルを作成します。文字コード 0 から 31 までと 128 から 255 までは、標準の構文テーブルを継承して作られます。ほかの文字コードについては、標準の構文テーブルを複写して作成されます。

ほとんどの主モードの構文テーブルはこのようにして作成されます。

copy-syntax-table &optional *table* Function

この関数は、 *table* のコピーを作り、それを返します。 *table* が与えられなかった場合 (もしくは、 `nil` だった場合)、現在の構文テーブルのコピーを返します。 *table* が、構文テーブルでなかった場合、エラーが通知されます。

modify-syntax-entry *char syntax-descriptor* &optional *table* Command

この関数は、 *syntax-descriptor* で *char* の構文エントリを設定します。構文は *table*(デフォルトは現在のバッファの構文テーブル) についてのみ変更され、ほかの構文テーブルは、変更されません。引数 *syntax-descriptor* で、望む構文を指定します。これは、クラス指示子で始まる文字列で、オプションとして対応文字やフラグも含んでいます。 31.2 節「構文記述子」p.542 を参照してください。

この関数は常に、 `nil` を返します。この文字のテーブル中の古い構文情報は捨てられます。

構文記述子の最初の文字が、 12 のクラス指示子の一つでなかった場合、エラーが通知されます。 *char* が、文字でない場合もまた、エラーが通知されます。

例:

```
;; スペース文字をクラス空白に入れる。
(modify-syntax-entry ?\  " ")
     ⇒ nil
;; ‘$’ を ‘^’ が対応する
;;    閉じ括弧となる、開き括弧文字とする。
(modify-syntax-entry ?$ "(^")
     ⇒ nil
;; ‘^’ を ‘$’ が対応する
;;    開き括弧となる、閉じ括弧文字とする。
(modify-syntax-entry ?^ ")$")
     ⇒ nil
```

```
;; '/' を句読点文字とする。
;;     コメント開始シーケンスの第 1 文字で、
;;     コメント終了シーケンスの第 2 文字である。
;;     C モードで使われる。
(modify-syntax-entry ?/ ". 14")
     ⇒ nil
```

char-syntax *character* Function

この関数は、ニモニック指示子で表された *character* の構文クラスを返します。この関数は、クラスだけを返します。対応括弧やフラグは返しません。

char が、文字でなければ、エラーが通知されます。

以下の例は、 C モードに当てはまります。最初の例は、スペースの構文クラスが空白 (スペースで表される) であることを示します。 2 番目の例は、 '/' の構文が句読点であることを示します。これは、コメント開始と終了のシーケンスでもあるということは、示していません。 3 番目の例は、開き括弧は、開き括弧のクラスに属していることを示しています。対応文字が ')' であることは示していません。

```
(char-to-string (char-syntax ?\ ))
     ⇒ " "

(char-to-string (char-syntax ?/))
     ⇒ "."

(char-to-string (char-syntax ?\())
     ⇒ "("
```

set-syntax-table *table* Function

この関数は、カレント・バッファの構文テーブルを *table* にします。 *table* を返します。

syntax-table Function

この関数は、現在の構文テーブル、つまりカレント・バッファで使われている構文テーブルを返します。

31.4 移動と構文

この節では、ある構文クラスの文字をまたがって移動する関数について述べます。 Emacs のバージョン 18 以前には、これらの関数は存在しません。

skip-syntax-forward *syntaxes* &optional *limit* Function

この関数は、 *syntaxes* で記述されている構文クラスをもつ文字をまたがって、ポイントを前方に移動します。バッファの終端や、 (指定されていれば)*limit* の位置、飛ばすことになっていない文字にきたとき、止まります。戻り値は、移動の距離で、非負整数です。

skip-syntax-backward *syntaxes* &optional *limit* Function

この関数は、 *syntaxes* に記述されている構文クラスをもつ文字をまたがって、ポイントを後方に移動します。バッファの終端や、 (指定されていれば)*limit* の位置、飛ばすことになっていない文字にきたとき、止まります。戻り値は、移動の距離で、非正整数です。

backward-prefix-chars Function

この関数は、式前置構文をもつ文字を越えて、ポイントを戻します。これは、式前置構文クラスの文字と ‘p’ フラグをもつ文字の両方を含みます。

31.5 釣り合った式の解析

ここに括弧が対で対応する、釣り合った式 (『S 式 (sexp)』としても知られる) を解析したりスキャンしたりする関数をいくつか挙げます。構文テーブルは、文字の解釈を制御します。つまり、これらの関数は、 Lisp モードのときは Lisp 式向けに、 C モードのときは、 C の式向けに使用されます。釣り合った式にまたがって移動する、便利で高機能の関数については、 27.2.6 節「釣り合った式をまたがる移動」p.464 を参照してください。

parse-partial-sexp *start limit* &optional *target-depth stop-before state stop-comment* Function

この関数は、 *start* から始まるカレント・バッファの S 式を解析します。 *limit* 以降はスキャンしません。位置 *limit* またはいかに示すある種の基準を満たしたとき、解析を止め、解析が止まった場所にポイントを移します。解析が止まった場所で、解析の状態を表す値を返します。

state が `nil` の場合、 *start* は、関数定義の始まりのような、括弧をつかった構造の先頭であると仮定されます。一方、構造体の中間で解析を再開したい場合もあるでしょう。このためには、解析状態の初期値を記述した *state* 引数を与えなければなりません。

3 番目の引数 *target-depth* が、非 `nil` の場合、括弧の深さが *target-depth* と等しくなると、解析を止めます。深さは、 0 もしくは、 *state* に与えられた数で始まります。

4 番目の引数 *stop-before* が、非 `nil` の場合、 S 式を始める文字が現れたとき解析が止まります。 *stop-comment* が非 `nil` の場合、コメントの開始にきたとき、解析は止まります。

5 番目の引数 *state* は、以下に述べるこの関数の値と同じ形式の八つの要素からなるリストです。 1 回の呼出しの戻り値は、 `parse-partial-sexp` に対する次の呼出し時の解析の状態を初期化するために使うことができます。

結果は、解析の最終的な状態を表す八つの要素からなるリストとなります:

0. 0 から数えた括弧の深さ。
1. 停止点を含んでいる、最も内側の括弧のグループが開始した文字位置です。なければ、 `nil`。
2. 完全な終了した直前の部分式が開始する文字位置。なければ、 `nil`。
3. 文字列内であれば、非 `nil`。より正確には、これは、文字列を終了させる文字です。
4. コメント (いずれの形式でも) の内であれば、 `t`。
5. quote 文字の直後であれば、 `t`。
6. このスキャンの間にみつけられた最小の括弧の深さ。

7. “b” 形式のコメント内であれば、 `t`。

引数 *state* 内の要素 0、 3、 4、 5、 7 は重要です。

この関数は、入れ子になった括弧を使う言語の字下げを計算するために最もよく使われます。

scan-lists *from count depth* Function

この関数は、文字位置 *from* から *count* 回釣り合った括弧のグループをスキャンします。スキャンが終わったところで、その文字の位置を返します。

depth が非ゼロならば、括弧の深さはこの値から数えます。止まるためのただ一つの候補は、括弧の深さがゼロになる場所です。 `scan-lists` は、そういう場所を *count* 回数えて、止まります。それゆえ、 *depth* の正の値は、 *depth* 段の括弧を抜け出ることを意味します。

`parse-sexp-ignore-comments` が、非 `nil` の場合、スキャンはコメントを無視します。

スキャンがバッファ (もしくは、そのアクセス可能部分) の先頭、末尾に達した場合で、深さがゼロではない場合、エラーが通知されます。深さがゼロなのに、 `count` を使い切ってない場合、 `nil` が返ります。

scan-sexps *from count* Function

この関数は、文字位置 *from* から、 *count* 回 S 式をスキャンします。スキャンが止まった文字位置を返します。

`parse-sexp-ignore-comments` が、非 `nil` ならば、コメントは無視されます。

スキャンが、括弧のグループの途中で、バッファの (アクセス可能な部分の) 先頭や末尾に達したら、エラーが通知されます。 `count` を使い切る前に、先頭や末尾に達したら、 `nil` が返されます。

parse-sexp-ignore-comments Variable

値が非 `nil` であれば、この節の関数や `forward-sexp` によってコメントは、空白として扱われます。

古い Emacs のバージョンでは、コメント終了が ‘`*/`’ のようなものであり、コメントを終らせるときにだけ現れるという場合にかぎり、使えた機能でした。改行がコメントを終わらせる言語では、全ての改行がコメントの終わりではないため、この変数を `nil` にする必要がありました。この制限は、もうなくなりました。

一つあるいは、複数のコメントを越えて前方あるいは後方に移動するために、 `forward-comment` を利用することができます。

forward-comment *count* Function

この関数は、 *count* 個のコメントをまたがって、ポイントを前方に移動します (*count* が負の場合は、後方)。コメントか空白以外のものをみつけたら止まり、止まった場所にポイントを置きます。 *count* 分進めた場合も止まります。

ポイント以後、全てのコメントと空白を越えて前方に移動するには、(`forward-comment (buffer-size)`) を使ってください。引数には (`buffer-size`) が使えます。バッファ内のコメント数がこの値を越えることは、あり得ないからです。

31.6 標準の構文テーブル

Emacs のほとんどの主モードは、おのおの独自の構文テーブルを備えています。ここでは、それらをいくつか紹介します。

standard-syntax-table Function

この関数は、標準の構文テーブルを返します。それは、 Fundamental モードで使われる構文テーブルです。

text-mode-syntax-table Variable

この変数の値は、 Text モードで使われる構文テーブルです。

c-mode-syntax-table Variable

この変数の値は、 C モード・バッファのための構文テーブルです。

emacs-lisp-mode-syntax-table Variable

この変数の値は、 Emacs Lisp モードで編集コマンドによって使われる構文テーブルです (Lisp `read` では、影響ありません)。

31.7 構文テーブルの内部

構文テーブルの各要素は、 1 文字の構文をコード化した整数です。内容は、構文クラス、対応文字候補、フラグです。 Lisp プログラムは、その要素で直接働くわけではありません。 Lisp レベルの構文テーブル関数は、普通は構文記述子 (31.2 節「構文記述子」 p.542 参照) によって機能します。

構文テーブルの各要素の下位 8 ビットは、構文クラスを示しています。

整数	クラス
0	空白
1	句読点
2	語
3	シンボル
4	開き括弧
5	閉じ括弧
6	式前置
7	文字列 quote

8	対の区切り
9	エスケープ
10	文字 quote
11	コメント開始
12	コメント終了
13	継承

次の 8 ビットは対応する逆括弧 (その文字が括弧構文をもつなら) です。違う場合、意味を持ちません。その次の 6 ビットは、フラグです。

32 省略形とその展開

省略あるいは『省略形 (abbrev)』とは、さらに長い文字列へと展開できる文字列のことです。ユーザは省略した文字列を入力し、それを省略の展開形によって自動的に置き換えることができます。このことによりタイプする量を減らせます。

現在有効な省略形は、『省略形テーブル (abbrev table)』に保持されています。各バッファはローカルな省略形テーブルを持っていますが、通常同じ主モードにあるバッファは一つの省略形テーブルを共有します。また、グローバルな省略形テーブルも存在します。通常は両方のテーブルを使用します。

省略形テーブルは、各省略形のためのシンボルを含む obarray として表現されます。このシンボルの名前が省略形であり、シンボルの値が展開形、シンボルの関数定義が省略形を展開するときのフック関数です (32.3 節「省略形の定義」 p.554 参照)。シンボルの属性リスト・セルには、省略形が展開された回数が入ります。これらのシンボルは通常の obarray には intern されないため、 Lisp 式を読み込んだ結果として現れることは決してありません。実際、普通は省略形を操作するコード以外では決して使われません。そのため、これらのシンボルを極端に通常とは違う方法で使用しても安全です。 7.3 節「シンボルの作成と intern」 p.95 を参照してください

ユーザレベルでの省略形に関するコマンドは *The GNU Emacs Manual* の “Abbrev Mode” 節を御覧ください。

32.1 Emacs で省略形が使えるように準備する

Abbrev モードは、 `abbrev-mode` 変数の値で制御される副モードです。

abbrev-mode Variable

この変数の値が非 `nil` のとき、省略形がバッファに入力されたときの自動展開が有効になります。値が `nil` のときは、省略形が定義されていたとしても、それは自動的に展開されません。

この変数はどのような方法で設定しても、自動的にバッファローカル変数になります。

default-abbrev-mode Variable

これは、 `abbrev-mode` の、その値が上書きされていないバッファにおける値です。`(default-value 'abbrev-mode)` と同じです。

32.2 省略形テーブル

この節では、省略形テーブルをどのように作成し、操作するのかを説明します。

make-abbrev-table Function

この関数は、新しい空の省略形テーブル、つまり、何もシンボルを含まない obarray を作成して返します。これはゼロで埋められたベクトルです。

clear-abbrev-table *table* Function

この関数は、 *table* のすべての省略形の定義を未定義化しますが、テーブルそのものは空にして残します。戻り値は `nil` です。

define-abbrev-table *tabname definitions* Function

この関数は、 *tabname*(シンボル) を省略形テーブルとして定義します。つまり、値に省略形テーブルをもつ変数として定義します。この関数は、 *definitions* にしたがって、テーブルのなかの省略形を定義します。 *definitions* は、(*abbrevname expansion hook usecount*) 形式の要素をもつリストです。戻り値は常に `nil` です。

abbrev-table-name-list Variable

省略形テーブルを値にもつシンボルのリストです。 `define-abbrev-table` は新しい省略形テーブルの名前をこのリストにつけ加えます。

insert-abbrev-table-description *name* &optional *human* Function

この関数は、 *name* と名づけられた省略形テーブルの記述を、ポイントの前に挿入します。引数 *name* は、省略形テーブルを値にもつシンボルです。戻り値は、常に `nil` です。

もし、 *human* が非 `nil` のとき、記述は人間が読みやすい形でおこなわれます。それ以外のばあい、記述は Lisp 式となります。この Lisp 式は、現在定義されている *name* を正確に定義する `define-abbrev-table` の呼び出し式です。

32.3 省略形の定義

以下の関数は、指定した省略形テーブルの中の省略形を定義します。 `define-abbrev` が低水準な基本関数であるのに対し、 `add-abbrev` はユーザに情報を尋ねるコマンドによって使われます。

add-abbrev *table type arg* Function

この関数は、ユーザからの情報に基づいて、省略形テーブル *table* に省略形を追加します。引数 *type* は、これから定義する省略形の種類に関して英語で記述した文字列です (典型的な例は `"global"`あるいは `"mode-specific"`)。これは、ユーザへのプロンプトに使われます。引数 *arg* は、展開形に使用する単語数です。

戻り値は、新しい省略形を内部的に示すシンボルです。または、すでに存在する省略形を上書きするかどうかの確認時にユーザが承諾しなかった場合には、 `nil` です。

define-abbrev *table name expansion hook* Function

この関数は、 *table* に、 *name* という名前の、 *expansion* へと展開され、 *hook* を呼び出す省略形を定義します。戻り値は、 Emacs 内部で省略形を表現する intern されていないシンボルです。その名前は、 *name* です。

引数 *name* は、文字列であるべきです。引数 *expansion* は、文字列か、あるいは省略形を取り消す場合は、 `nil` であるべきです。

引数 *hook* は、関数または `nil` です。もし、 *hook* が非 `nil` のときは、省略形を *expansion* で置き換えたあとに、引数なしでそれが呼ばれます。 *hook* が呼ばれるとき、ポイントは *expansion* の末尾です。

省略形の使用回数はゼロに初期化されます。

only-global-abbrevs User Option

この変数の値が非 `nil` の場合、ユーザはグローバル省略形だけを使うと決めたことになります。このことはコマンドに、モード特有の省略形を定義するかわりに、グローバルな省略形を定義するよう示します。この変数は、この節の関数の機能を変えることはなく、関数の呼びだし側でチェックされます。

32.4 省略形をファイルにセーブする

省略形の定義が保存されたファイルは、実際には Lisp コードのファイルです。省略形は、同じ内容の省略形テーブルを定義する Lisp プログラムの形で保存されます。そのため、 `load` (13.1 節「プログラムでロードする方法」 p.171 参照) でファイルを読み込むことが可能です。しかしながら、関数 `quietly-read-abbrev-file` は、より便利なインターフェースを提供します。

`save-some-buffers` のようなユーザレベルの機能は、以下に示すような変数のコントロールのもとで、自動的に省略形をファイルへ保存することができます。

abbrev-file-name User Option

省略形を読み書きするファイル名のデフォルト値です。

quietly-read-abbrev-file *filename* Function

この関数は、前に `write-abbrev-file` で書き込まれた *filename* という名前のファイルから、省略形の定義を読み込みます。もし、 *filename* が `nil` のときは、 `abbrev-file-name` で指定されたファイルが使われます。あとで変更を保存するために、 `save-abbrevs` は `t` にセットされます。

この関数は、メッセージを表示しません。 `nil` を返します。

save-abbrevs User Option

変数 `save-abbrev` が非 `nil` のとき、 Emacs はファイルをセーブする時に省略形もセーブします。 `abbrev-file-name` は、省略形をセーブするファイル名を指定します。

abbrevs-changed Variable

この変数は、省略形の定義や変更があったときに、非 `nil` にセットされます。さまざまな Emacs コマンドがあなたの省略形を保存するかどうかを判断するフラグの役目を果たします。

write-abbrev-file *filename* Command

全省略形テーブルのすべての省略形定義を、ファイル *filename* へ、ロードすると元と同じ省略形を定義する Lisp プログラムの形でセーブします。この関数は `nil` を返します。

32.5 省略形の参照と展開

省略形はたいてい、 `self-insert-command` を含む対話的に使われるコマンドによって展開されます。この節では、そのような関数を書く時に使うサブルーチンについて説明します。同じく、関数同士がコミュニケーションするために使う変数についても説明します。

abbrev-symbol *abbrev* &optional *table* Function

この関数は、 *abbrev* と名づけられた省略形を示すシンボルを返します。省略形が定義されていない場合は、 `nil` を返します。二つめのオプション引数 *table* は、省略形を探すテーブル名です。 *table* が `nil` のとき、この関数は、まずカレント・バッファのローカル省略形テーブルを探し、その次にグローバル省略形テーブルを探します。

abbrev-expansion *abbrev* &optional *table* Function

この関数は、 *abbrev* が (カレント・バッファで使われる省略形テーブルの定義にしたがって) 展開される文字列を返します。オプション引数 *table* は、 `abbrev-symbol` の場合と同じく、使用するテーブルを指定します。

expand-abbrev Command

このコマンドは、ポイントの前の省略形を展開します。もし、ポイントが省略形の後ろになければ、このコマンドはなにもしません。展開を行なった場合は `t` を返し、それ以外は `nil` を返します。

abbrev-prefix-mark &optional *arg* Command

現在のポイントを、省略形の始まりとしてマークします。次の `expand-abbrev` の呼び出しでは、通常のようにポイント直前の単語ではなく、この場所から (そのときの) ポイントまでを、展開するべき省略形として使用します。

abbrev-all-caps User Option

非 `nil` のとき、すべて大文字で入力された省略形は、すべて大文字で展開されます。それ以外の場合、すべて大文字で入力された省略形は、各単語の先頭の一文字が大文字で展開されます。

abbrev-start-location Variable

これは、 `expand-abbrev` のためのバッファ内位置で、次の省略形展開の始まりの位置に使われます。 (`nil` のときは、ポイントの前の単語を代わりに使う意味になります)。 `abbrev-start-location` は、 `expand-abbrev` が呼び出されるたびに `nil` に初期化されます。この変数は、 `abbrev-prefix-mark` によってもセットされます。

abbrev-start-location-buffer Variable

この変数の値は、 `abbrev-start-location` がセットされたバッファです。ほかのバッファで省略形展開を行なうと、 `abbrev-start-location` はクリアされます。この変数は、 `abbrev-prefix-mark` によってセットされます。

last-abbrev Variable

これは、最新の省略形の展開の `abbrev-symbol` です。この情報は、 `unexpand-abbrev` コマンドのために、 `expand-abbrev` によって設定されます。

last-abbrev-location Variable

最新の省略形展開が行なわれた場所を示します。この情報は、 `unexpand-abbrev` コマンドのために、 `expand-abbrev` によって設定されます。

last-abbrev-text Variable

これは、大文字小文字の変換も (もしあった場合は) 含めて、最新の省略形展開の正確な展開形です。もし、展開がすでに逆展開さているならば、この値は `nil` となります。この情報は、 `unexpand-abbrev` コマンドのために、 `expand-abbrev` によって設定されます。

pre-abbrev-expand-hook Variable

これは正規フックであり、省略形が展開される直前に、このフックが保持する関数が順番に実行されます。 20.4 節「フック」p.335 を参照してください。 正規フックであるため、このフック関数は引数をとりません。しかしながら、ポイントの直前を調べることで、展開される省略形を見つけることができます。

以下の例は、 `pre-abbrev-expand-hook` の簡単な使い方です。もしユーザーが省略形を句読点で終らせた場合、確認を行なうフック関数が実行されます。このため、このフックはユーザに省略形を展開するかどうか決定させ、もし確認がとれない場合はその展開を中止します。

```
(add-hook 'pre-abbrev-expand-hook 'query-if-not-space)

;; この関数は pre-abbrev-expand-hook から呼び出されます。

;; もしユーザがスペースで省略形を終らせた場合、この関数は何も
;; 行ないません (省略形の展開のために戻ります)。
;; もしユーザがそれ以外の文字を入力した場合、この関数は展開を行なうか
;; どうかを尋ねます。

;; 問い合わせに対して、ユーザが y を入力した時 nil を
;; 返します (not 関数のため)。
;; しかし戻り値は展開には影響ないので、これで問題ないでしょう。


(defun query-if-not-space ()
  (if (/= ?\  (preceding-char))
      (if (not (y-or-n-p "Do you want to expand this abbrev? "))
          (error "Not expanding this abbrev"))))
```

32.6 標準的な省略形テーブル

ここに示す変数は、 Emacs でプリロードされる主モードのために用意している省略形テーブルです。

global-abbrev-table Variable

モードに依存しない省略形テーブルです。ここで定義された省略形は、全バッファで有効になります。各バッファは、ローカルな省略形テーブルももつことができ、そこで定義された省略形は、このグローバル・テーブルのものよりも優先されます。

local-abbrev-table Variable

このバッファローカル変数の値は、カレント・バッファの (モード特有の) 省略形テーブルです。

fundamental-mode-abbrev-table Variable

Fundamental モードで使われるローカルな省略形テーブルです。全 Fundamental モード・バッファのローカル省略形テーブルとなります。

text-mode-abbrev-table Variable

Text モードで使われる省略形テーブルです。

c-mode-abbrev-table Variable

C モードで使われる省略形テーブルです。

lisp-mode-abbrev-table Variable

Lisp モードと Emacs Lisp モードで使われる省略形テーブルです。

33 プロセス

『プロセス (process)』とは、プログラムが実行できる空間を指すオペレーティング・システム用語です。 Emacs は一つのプロセス内で動作します。 Emacs Lisp プログラムは、ほかのプログラムを独立したプロセスとして起動できます。このようなプロセスは、 Emacs プロセスの『サブプロセス (subprocess)』または『子プロセス (child process』と呼ばれます。 Emacs プロセスは、これらのプロセスの『親プロセス (parent process)』になります。

Emacs のサブプロセスには、プロセスの作成方法によって、『同期 (synchronous)』と『非同期 (asynchronous)』の 2 種類があります。同期サブプロセスを作成すると、 Lisp プログラムは、そのサブプロセスが終了するのを待ってから実行を再開します。非同期サブプロセスを作成すると、そのサブプロセスは Lisp プログラムと並行して動作します。このようなサブプロセスは、 Emacs 内では、同様に “プロセス” と呼ばれる Lisp オブジェクトとして表現されます。 Lisp プログラムは、このオブジェクトを使って、サブプロセスと通信したり、サブプロセスを制御できます。たとえば、サブプロセスにシグナルを送ったり、ステータス情報を取得したり、出力を受け取ったり、入力を送ることができます。

processp *object* Function

この関数は、 *object* がプロセスならば `t`、そうでなければ `nil` を返します。

33.1 サブプロセスを作成する関数

プログラムを実行するための新しいサブプロセスを作成する関数は三つあります。そのうちの一つである `start-process` は、非同期プロセスを作成し、プロセス・オブジェクトを返します (33.4 節「非同期プロセスの作成」 p.563 参照)。後の二つの `call-process` と `call-process-region` は、同期プロセスを作成し、プロセス・オブジェクトを返しません (33.2 節「同期プロセスの作成」 p.560 参照)。

ここでは、同期プロセスと非同期プロセスについて説明します。三つの関数はどれも同様に呼び出されるため、共通な引数についてはここで説明します。

同期か非同期かにかかわらず、関数の *program* 引数には、実行するプログラムを指定します。指定されたファイルが見つからない場合や、ファイルの実行が不可能な場合は、エラーが通知されます。ファイル名が相対的ならば、変数 `exec-path` が検索するディレクトリのリストとして使われます。 Emacs は、起動時に環境変数 `PATH` の値に基づいて `exec-path` を初期化します。 ‘`~`’、 ‘`.`’、 ‘`..`’ のような標準のファイル名要素は通常どおり解釈されますが、環境変数の置換 (‘`$HOME`’ など) は認識されません。環境変数の置換を行うには、 `substitute-in-file-name` を使ってください (22.8.4 節「ファイル名を展開する関数」 p.370 参照)。

どのサブプロセス作成関数にも、起動するプログラムの標準出力の行き先を指定する *buffer-or-name* 引数があります。 *buffer-or-name* が `nil` ならば、フィルタ関数が処理しないかぎり出力を破棄することを意味します (33.9.2 節「プロセス・フィルタ関数」 p.571 を参照してください。また、第 16 章「Lisp オブジェクトの読込みと印字」 p.227)。複数のプロセスの出力を一つのバッファに送ると出力が無作為に混ざってしまうため、通常は複数のプロセスの出力を同じバッファに送らないでください。

三つのサブプロセス作成関数のすべてに、 `&rest` 引数の *args* があります。 *args* はすべて文字列でなければならず、各要素はそれぞれ独立したコマンド・ライン引数として *program* に渡されます。これらの文字列は指定されたプログラムに直接渡されるため、ワイルドカードなどのシェルの機能は使えません。

注意: *program* 引数には、プログラムの名前だけを指定します。この引数にはコマンド・ライン引数を含めないでください。コマンド・ライン引数は、 *args* を使って指定してください。

サブプロセスのカレント・ディレクトリは、 `default-directory` の値になります (22.8.4 節「ファイル名を展開する関数」 p.370 参照)。

サブプロセスは、 Emacs から環境を引き継ぎます。ただし、 `process-environment` を使って別の環境を指定することもできます。 34.3 節「OS 環境」 p.584 を参照してください。

exec-directory Variable

この変数の値は、 GNU Emacs に付属するプログラムが格納されているディレクトリの名前 (文字列) です。これらのプログラムは、 Emacs から起動するために用意されています。 `wakeup` プログラムは、このようなプログラムの例です。 `display-time` コマンドは、 `wakeup` プログラムを使って、 1 分に 1 回通知を受けます。

exec-path User Option

この変数の値は、サブプロセス内で実行されるプログラムが検索されるディレクトリのリストです。各要素は、ディレクトリの名前 (文字列) または `nil` です。 `nil` は、デフォルトのディレクトリ (`default-directory` の値) を意味します。

program 引数が絶対ファイル名でない場合、 `call-process` と `start-process` は `exec-path` の値を使います。

33.2 同期プロセスの作成

Emacs は、『同期プロセス (synchronous process)』を作成した後、そのプロセスが終了するのを待ってから実行を再開します。 Dired を起動する場合がこの例です。 Emacs は、 `ls` を同期プロセスとして実行し、その出力を少し変更します。プロセスが同期的であるため、 Emacs は、ディレクトリ一覧の全体がバッファに届いてから、バッファに対する操作を実行しようとします。

同期サブプロセスが終了するのを Emacs が待っている間、ユーザーは、 `C-g` を入力して中断できます。 1 回目の `C-g` は、サブプロセスを `SIGINT` シグナルで kill しようとしますが、サブプロセスが実際に終了するまで待ってから中断します。その間にユーザーが `C-g` をもう一度入力した場合、サブプロセスは `SIGKILL` で即座に kill され、すぐに中断されます。 18.8 節「とりやめ」 p.284 を参照してください。

バージョン 18 では、同期サブプロセス関数は `nil` を返していました。バージョン 19 では、同期サブプロセス関数はプロセスの終了状態を返します。

call-process *program* &optional *infile destination display* &rest *args* Function

この関数は、 *program* を独立したプロセスで実行し、プログラムが終了するまで待ちます。

infile が nil でなければ、プロセスの標準入力には *infile* が使われます。 *infile* が nil ならば、'/dev/null' が標準入力になります。 *destination* 引数には、プロセスの出力の行き先を指定します。 *destination* 引数の例を次に示します。

バッファ　バッファのポイントの前に出力を挿入します。プロセスの標準出力ストリームと標準エラーストリームの両方が挿入されます。

文字列　文字列を名前とするバッファに切り換えるかバッファを作成して、出力をそのバッファのポイントの前に挿入します。

t　出力を現在のバッファのポイントの前に挿入します。

nil　出力を破棄します。

0　出力を破棄し、サブプロセスが終了するのを待たずにすぐに戻ります。

この場合、プロセスは Emacs と並行に動作できるため、プロセスは本当は同期プロセスではありません。しかし、この関数が戻った時点で Emacs がサブプロセスの処理を実質的に終えるという点では、同期プロセスと考えることができます。

(*real-destination error-destination*)
標準出力ストリームと標準エラー・ストリームを個別に扱います。標準出力は *real-destination* で指定されているとおりに処理し、エラー出力は *error-destination* にしたがって処理します。 *error-destination* の値が nil ならばエラー出力を破棄し、 t ならば標準出力にエラー出力を混合します。エラー出力をリダイレクトするには、リダイレクト先のファイル名の文字列を *error-destination* に指定します。

エラー出力が挿入されるバッファを直接指定することはできません。実装するのが難しいためです。しかし、エラー出力をテンポラリ・ファイルに送り、そのファイルをバッファに挿入すれば、同様のことを実現できます。

display が nil ならば、 call-process は、出力が挿入されるたびにバッファを再表示します。 *display* が nil でなければ、 call-process は再表示を行なわず、結果が画面に表示されるのは Emacs が通常のイベントでそのバッファを再表示したときだけになります。

残余引数の *args* は、プログラムのコマンド・ライン引数を指定する文字列です。

(プロセスの終了を待つように指定した場合)call-process が返す値は、プロセスが終了した理由を示します。値が数値の場合は、サブプロセスの終了ステータスを示します。 0 は成功で、それ以外の数値は失敗を示します。プロセスがシグナルを受けて終了した場合、 call-process は、そのシグナルを示す文字列を返します。

以下の例では、バッファ 'foo' が現在のバッファであるとします。

```
(call-process "pwd" nil t)
     ⇒ nil

---------- Buffer: foo ----------
/usr/user/lewis/manual
---------- Buffer: foo ----------

(call-process "grep" nil "bar" nil "lewis" "/etc/passwd")
     ⇒ nil

---------- Buffer: bar ----------
lewis:5LTsHm66CSWKg:398:21:Bil Lewis:/user/lewis:/bin/csh

---------- Buffer: bar ----------
```

insert-directory 関数には、call-process の使い方のよい例があります。

```
(call-process insert-directory-program nil t nil switches
              (if full-directory-p
                  (concat (file-name-as-directory file) ".")
                file))
```

call-process-region *start end program* &optional *delete buffer destination display* &rest *args* Function

この関数は、*program* が実行されるプロセスの標準入力に、*start* から *end* までのテキストを送ります。*delete* が nil でなければ、この関数は送ったテキストを削除します。この機能は、*buffer* に t を指定して出力を現在のバッファに挿入する場合に役立ちます。

destination 引数と *display* 引数は、サブプロセスからの出力の処理方法と、出力が送られてきたときに表示を更新するかどうかを指定します。詳細については、上記の call-process の説明を参照してください。*destination* が整数の 0 ならば、call-process-region は、サブプロセスが終了するのを待たずに、出力を破棄して即座に nil を返します。

残余引数の *args* は、プログラムのコマンド・ライン引数を指定する文字列です。

call-process-region の戻り値は call-process の戻り値と同様です。プロセスの終了を待たずに戻るように指定した場合は、nil が返されます。それ以外の場合は、サブプロセスの終了方法を示す数値または文字列が返されます。

以下の例は、'foo' バッファの最初の 5 文字 ('input') を標準入力として、call-process-region を使って cat ユーティリティを実行します。cat は、標準入力を標準出力にコピーします。*destination* 引数には t を指定しているため、出力は現在のバッファに挿入されます。

```
---------- Buffer: foo ----------
input⋆
---------- Buffer: foo ----------

(call-process-region 1 6 "cat" nil t)
     ⇒ nil

---------- Buffer: foo ----------
inputinput⋆
---------- Buffer: foo ----------
```

shell-command-on-region コマンドは、 call-process-region を次のように使っています。

```
(call-process-region
 start end
 shell-file-name      ; プログラムの名前。
 nil                  ; リージョンを削除しません。
 buffer               ; 出力を buffer に送ります。
 nil                  ; 出力中に再表示しません。
 "-c" command)        ; シェルの引数。
```

33.3 MS-DOS でのサブプロセス

MS-DOS では、同期サブプロセスとやりとりするデータがテキストかバイナリかを指定しなければなりません。テキスト・データの場合は、 Emacs で使われる行末記法 (改行文字一つ) と、 Emacs 外で使われる行末記法 (CRLF の 2 文字) との間での変換が必要になります。

サブプロセスに送られる入力には binary-process-input 変数が適用され、サブプロセスから受け取る出力には binary-process-output 変数が適用されます。データがテキストでない場合は、これらの変数の値に nil 以外を指定します。データがテキストであり、変換が必要な場合は、これらの変数に nil を指定します。

binary-process-input Variable

この変数が nil ならば、同期サブプロセスの入力の改行を CRLF に変換します。

binary-process-output Variable

この変数が nil ならば、同期サブプロセスの入力の CRLF を改行にに変換します。

関連情報については、 22.13 節「ファイルと MSDOS」 p.380 を参照してください。

33.4 非同期プロセスの作成

『非同期プロセス (asynchronous process)』が作成された後、 Emacs と Lisp プログラムはどちらも即座に実行を再開します。プロセスはその時点から Emacs と並行に動作し、 Emacs とプロセスは以下で説明する関数を使って互いに通信できます。ここでは、 start-process による非同期プロセスの作成方法について説明します。

start-process *name buffer-or-name program* &rest *args* Function

この関数は、新しい非同期サブプロセスを作成して、その中で動作するプログラム *program* を起動します。この関数は、新しいサブプロセスを Lisp 内で表現するプロセス・オブジェクトを返します。 *name* 引数には、プロセス・オブジェクトの名前を指定します。この名前のプロセスがすでに存在していれば、 *name* は一意になるように (‘<1>’ などを追加して) 修正されます。バッファ *buffer-or-name* は、プロセスに関連付けられるバッファです。

残余引数の *args* は、プログラムのコマンド・ライン引数を指定する文字列です。

以下の例では、一つめのプロセスは起動して 100 秒間実行します (正確には、 sleep します)。その間、二つめのプロセスが起動され、一意性を保つために ‘my-process<1>’ という名前が与えられます。このプロセスは、最初のプロセスが終了する前に、ディレクトリの一覧をバッファ ‘foo’ の終端に挿入します。そして、二つめのプロセスが終了し、そのことを示すメッセージがバッファに挿入されます。しばらく経ってから、一つめのプロセスが終了し、そのことを示すメッセージがさらにバッファに挿入されます。

```
(start-process "my-process" "foo" "sleep" "100")
     ⇒ #<process my-process>
(start-process "my-process" "foo" "ls" "-l" "/user/lewis/bin")
     ⇒ #<process my-process<1>>

---------- Buffer: foo ----------
total 2
lrwxrwxrwx  1 lewis     14 Jul 22 10:12 gnuemacs --> /emacs
-rwxrwxrwx  1 lewis     19 Jul 30 21:02 lemon

Process my-process<1> finished

Process my-process finished
---------- Buffer: foo ----------
```

start-process-shell-command *name buffer-or-name command* &rest *command-args* Function

この関数は start-process に似ていますが、指定されたコマンドをシェルを使って実行する点が異なります。 *command* 引数がシェル・コマンドの名前で、 *command-args* 引数がシェル・コマンドの引数です。

process-connection-type Variable

この変数は、非同期サブプロセスとの通信に使われるデバイスの種類を制御します。この変数が nil 以外の場合、 PTY が利用可能ならば PTY が使われます。それ以外の場合は、パイプが使われます。

PTY の場合、ジョブ・コントロール (C-c、 C-z など) を使ってプロセスやその子プロセスを制御できるのに対し、パイプはプロセスの制御ができません。このため、 Shell モードのようにプロセスがユーザーから見える場合、通常は PTY が適しています。プログラムが内部的に使うプロセスの場合は、パイプの方がより効率的なため、通常はパイプを使うのが適切です。また、多くのシステムでは PTY の総数が限られているため、 PTY を浪費しないほうがよいでしょう。

start-process が呼び出されると、 process-connection-type の値が使われます。このため、 start-process の呼び出しの周辺だけでこの変数を束縛すれば、特定のサブプロセスとの通信方法を指定できます。

```
(let ((process-connection-type nil))  ; パイプを使います。
  (start-process ...))
```

特定のサブプロセスがパイプと PTY のどちらを実際に使っているかを調べるには、 process-tty-name 関数を使います (33.6 節「プロセス情報」 p.565 参照)。

33.5 プロセスの削除

『プロセスの削除 (deleting a process)』とは、 Emacs をサブプロセスから即座に切り離して、そのプロセスをアクティブなプロセスのリストから削除することです。 Emacs はサブプロセスを終了させるためにサブプロセスにシグナルを送りますが、サブプロセスが即座に終了するとはかぎりません。プロセス・オブジェクト自体は、ほかの Lisp オブジェクトがそのプロセス・オブジェクトを指しているかぎり存在し続けます。

プロセスはいつでも明示的に削除できます。プロセスは終了すると自動的に削除されますが、終了してすぐに削除されるとはかぎりません。終了したプロセスが自動的に削除される前にそのプロセスを削除しても害はありません。

delete-exited-processes Variable

この変数は、 (`exit` の呼び出しまたはシグナルの受信によって) 終了したプロセスの自動削除を制御します。この変数が `nil` ならば、ユーザーが `list-processes` を実行するまでプロセスは存在し続けます。それ以外の場合は、プロセスは終了するとすぐに削除されます。

delete-process *name* Function

この関数は、 *name* に関連付けられているプロセスを `SIGHUP` シグナルで kill し、プロセスを削除します。 *name* 引数には、プロセス、プロセスの名前、バッファ、バッファの名前のいずれかを指定できます。

```
(delete-process "*shell*")
     ⇒ nil
```

process-kill-without-query *process* Function

この関数は、 Emacs が終了するときに *process* がまだ動作していてもユーザーに問い合わせる必要がないことを宣言します。プロセスは予告なく削除されます。戻り値は `t` です。

```
(process-kill-without-query (get-process "shell"))
     ⇒ t
```

33.6 プロセス情報

一部の関数はプロセスに関する情報を返します。対話的な用途のために、 `list-processes` が用意されています。

list-processes Command

このコマンドは、アクティブなすべてのプロセスの一覧を表示します。さらに、ステータスが ‘`Exited`’ または ‘`Signaled`’ のプロセスを最終的に削除します。この関数は `nil` を返します。

process-list Function

この関数は、削除されていないプロセスすべてのリストを返します。

```
(process-list)
     ⇒ (#<process display-time> #<process shell>)
```

get-process *name* Function

この関数は、 *name* という名前のプロセスを返します。 *name* という名前のプロセスがなければ、 `nil` を返します。 *name* が文字列でなければ、エラーが発生します。

```
(get-process "shell")
     ⇒ #<process shell>
```

process-command *process* Function

この関数は、 *process* を起動するために実行されたコマンドを返します。返される値は文字列のリストで、最初の文字列は実行されたプログラムを示し、残りの文字列はそのプログラムに与えられた引数を示します。

```
(process-command (get-process "shell"))
     ⇒ ("/bin/csh" "-i")
```

process-id *process* Function

この関数は、 *process* の PID を返します。 PID は、一つのコンピュータ上で同時に動作しているほかのプロセスとプロセス *process* を区別する整数です。プロセスの PID は、プロセスが起動するときにオペレーティング・システムのカーネルによって選ばれ、プロセスが存在するかぎり変化しません。

process-name *process* Function

この関数は、 *process* の名前を返します。

process-status *process-name* Function

この関数は、 *process-name* のステータスをシンボルとして返します。 *process-name* 引数は、プロセス、バッファ、プロセス名 (文字列)、バッファ名 (文字列) のいずれかでなければなりません。

実際のサブプロセスの場合のステータス値の例を次に示します。

`run` 動作中のプロセスの場合。

`stop` 停止しているが再開可能なプロセスの場合。

`exit` 終了しているプロセスの場合。

`signal` 重大なシグナルを受け取ったプロセスの場合。

`open` オープン状態のネットワーク接続の場合。

`closed` クローズ状態のネットワーク接続の場合。いったんクローズされた接続は再オープンすることはできませんが、同じ場所への新しい接続をオープンすることはできます。

`nil` *process-name* が存在するプロセスの名前でない場合。

```
(process-status "shell")
     ⇒ run
(process-status (get-buffer "*shell*"))
     ⇒ run
x
     ⇒ #<process xx<1>>
(process-status x)
     ⇒ exit
```

ネットワーク接続の場合、 process-status は、シンボルの open または closed を返します。 closed は、接続の相手側が接続をクローズしたか、 Emacs が delete-process を実行したことを意味します。

以前のバージョン (バージョン 19 以前) の Emacs では、ネットワーク接続のステータスは、オープン状態ならば run、クローズ状態ならば exit でした。

process-exit-status *process* Function

この関数は、 *process* の終了ステータスか、そのプロセスを kill したシグナルの番号を返します (戻り値がこの二つのうちのいずれであるかを調べるには、 process-status の結果を使います)。 *process* がまだ終了していなければ、 0 が返されます。

process-tty-name *process* Function

この関数は、 *process* が Emacs との通信に使っている端末の名前を返します。プロセスが端末ではなくパイプを使っている場合は、 nil を返します (33.4 節「非同期プロセスの作成」p.563 の process-connection-type を参照してください)。

33.7 プロセスへの入力の送信

Emacs が入力を非同期サブプロセスに送ると、非同期サブプロセスはその入力を受け取ります。非同期サブプロセスに入力を送るには、ここで説明する関数を使います。関数には、入力の送信先のプロセスと、送信する入力データを指定します。データは、サブプロセスの "標準入力" に置かれます。

一部のオペレーティング・システムでは、 PTY の入力バッファの容量が限られています。そのようなシステムでは、 Emacs は送信するデータ中に EOF を周期的に挿入し、入力データを強制的に読み取らせます。 ほとんどのプログラムでは、この EOF による悪影響はありません。

process-send-string *process-name string* Function

この関数は、 *string* の内容を標準入力として *process-name* に送ります。 *process-name* 引数は、プロセスまたはプロセスの名前でなければなりません。 *process-name* 引数が nil ならば、現在のバッファのプロセスが使われます。

この関数は nil を返します。

```
(process-send-string "shell<1>" "ls\n")
     ⇒ nil

---------- Buffer: *shell* ----------
...
introduction.texi              syntax-tables.texi~
introduction.texi~             text.texi
introduction.txt               text.texi~
...
---------- Buffer: *shell* ----------
```

process-send-region *process-name start end* Command

この関数は、 *start* と *end* で定義されるリージョンを標準入力として *process-name* に送ります。 *process-name* はプロセスまたはプロセスの名前です (この引数が `nil` ならば、現在のバッファのプロセスが使われます)。

start と *end* のどちらかが現在のバッファ内での位置を示す整数またはマーカーでなければ、エラーが発生します (どちらの値が大きいかは無関係です)。

process-send-eof &optional *process-name* Function

この関数は、 *process-name* の入力に end-of-file を置きます。 EOF は、すでに送られているテキストの後に置かれます。

process-name を指定しなかった場合や、 *process-name* が `nil` の場合は、この関数は現在のバッファのプロセスに EOF を送ります。現在のバッファにプロセスがなければ、エラーが発生します。

この関数は *process-name* を返します。

```
(process-send-eof "shell")
     ⇒ "shell"
```

33.8 プロセスへのシグナルの送信

サブプロセスへの『シグナルの送信 (sending a signal)』は、サブプロセスの動作に割り込む方法の一つです。シグナルにはいくつかの種類があり、それぞれ意味があります。シグナルとその名前は、オペレーティングシステムで定義されています。たとえば、 `SIGINT` シグナルは、ユーザーが `C-c` を入力したか、それに類似した状況が発生したことを意味します。

各シグナルは、サブプロセスに対して特定の影響があります。ほとんどのシグナルはサブプロセスを kill しますが、実行を停止させたり再開させるシグナルもあります。ほとんどのシグナルは、プログラムが処理することもできます。プログラムがシグナルを処理する場合、そのシグナルの一般的な意味が通用するとはかぎりません。

シグナルを明示的に送るには、ここで説明する関数を呼び出します。また、 Emacs は、次のような場合に自動的にシグナルを送ります。バッファを kill する場合は、そのバッファに関連付けられているプロセスすべてに `SIGHUP` シグナルを送ります。 Emacs が終了する場合は、残っているプロセスすべてに `SIGHUP` シグナルを送ります (`SIGHUP` は、通常はユーザーが電話回線を切断したことを示すシグナルです)。

どのシグナル送信関数も、 *process-name* と *current-group* の二つの省略可能引数を取ります。

process-name 引数は、プロセス、プロセス名、 `nil` のいずれかでなければなりません。この引数が `nil` ならば、デフォルトとして現在のバッファに関連付けられているプロセスが使われます。 *process-name* がプロセスを示していない場合は、エラーが発生します。

current-group 引数は、ジョブ・コントロール機能を持つシェルを Emacs のサブプロセスとして実行している場合に意味を持つフラグです。 この引数が `nil` 以外ならば、 Emacs がサブプロセスとの通信に使っている端末の現在のプロセス・グループにシグナルが送られます。プロセ

スがジョブコントロール可能なシェルならば、現在のプロセス・グループはシェルの現在のサブジョブです。この引数が `nil` ならば、 Emacs の直接のサブプロセスのプロセス・グループにシグナルが送られます。サブプロセスがジョブ・コントロール可能なシェルならば、そのシェル自体にシグナルが送られます。

オペレーティング・システムはパイプの区別をサポートしていないため、パイプを使ってサブプロセスと通信している場合、 *current-group* フラグは意味を持ちません。同様の理由で、パイプを使っているときはジョブ・コントロール可能シェルはうまく動作しません。 33.4 節「非同期プロセスの作成」p.563 の `process-connection-type` を参照してください。

interrupt-process &optional *process-name current-group* Function

この関数は、 `SIGINT` シグナルを送ることによってプロセス *process-name* に割り込みます。 Emacs の外では、 "割り込み文字" (通常、 `C-c` であるシステムと `DEL` であるシステムがあります) を入力するとこのシグナルが送られます。 *current-group* 引数が `nil` 以外の場合は、 Emacs が端末で "`C-c` を入力する" ことによってサブプロセスと対話するための関数と考えることもできます。

kill-process &optional *process-name current-group* Function

この関数は、 `SIGKILL` シグナルを送ることによって、プロセス *process-name* を kill します。このシグナルはサブプロセスを即座に kill するため、サブプロセスがこのシグナルを処理することはできません。

quit-process &optional *process-name current-group* Function

このプロセスは、 `SIGQUIT` シグナルをプロセス *process-name* に送ります。このシグナルは、 Emacs の外で "中断文字" (通常は `C-b` または `C-\`) を入力したときに送られるシグナルです。

stop-process &optional *process-name current-group* Function

この関数は、 `SIGTSTP` シグナルを送ることによって、プロセス *process-name* を停止させます。プロセスの実行を再開するには、 `continue-process` を使います。

ジョブ・コントロールがあるシステムでは、 (Emacs の外で)"停止文字" (通常は `C-z`) を入力すると、このシグナルが送られます。 *current-group* が `nil` 以外の場合は、 Emacs がサブプロセスとの通信に使っている端末で "`C-z` を入力する" 関数と考えることもできます

continue-process &optional *process current-group* Function

この関数は、 `SIGCONT` シグナルを送ることによって、プロセス *process* の実行を再開します。この関数は、 *process* がすでに停止していることを仮定します。

signal-process *pid signal* Function

この関数は、プロセス *pid* にシグナルを送ります。プロセスは Emacs の子プロセスでなくてもかまいません。 *signal* 引数には、送信するシグナルを指定します。この引数は整数でなければなりません。

33.9 プロセスからの出力の受け取り

サブプロセスがそれ自体の標準出力ストリームに書き込んだ出力を受け取るには、二つの方法があります。一つは出力をバッファに挿入する方法です。このバッファは、プロセスに関連付けられたバッファと呼ばれます。もう一つの方法は、『フィルタ関数 (filter function)』と呼ばれる関数を出力に作用させる方法です。プロセスに対してバッファもフィルタ関数もない場合は、プロセスの出力は破棄されます。

33.9.1 プロセスのバッファ

プロセスには、『関連付けられたバッファ (associated buffer)』を設定できます (通常はバッファが関連付けられています)。プロセスに関連付けられたバッファは、プロセスからの出力の格納と、プロセスを kill するタイミングの決定の二つの目的に使われる、通常の Emacs バッファです。また、通常は一つのバッファには一つのプロセスしか関連付けられていないため、このバッファを使って、操作の対象となるプロセスを指定できます。また、プロセスを利用する場合のほとんどは、プロセスに送られる入力をこのバッファを使って編集しますが、この機能は Emacs Lisp には組み込まれていません。

プロセスにフィルタ関数 (33.9.2 節「プロセス・フィルタ関数」 p.571 参照) がなければ、プロセスの出力は関連付けられているバッファに挿入されます。出力が挿入される位置は、`process-mark` によって決まります。この変数は、挿入後、挿入されたテキストの終端を指すように更新されます。通常は (常にではありません)、 `process-mark` はバッファの終端です。

process-buffer *process* Function

この関数は、プロセスに関連付けられているバッファを返します。 *process.*

```
(process-buffer (get-process "shell"))
     ⇒ #<buffer *shell*>
```

process-mark *process* Function

この関数は、 *process* のプロセス・マーカーを返します。プロセス・マーカーとは、プロセスからの出力が挿入される位置を示すマーカーです。

process にバッファが関連付けられていなければ、 `process-mark` は、どこも指さないマーカーを返します。

バッファにプロセスの出力を挿入するときは、このマーカーを使って挿入位置が決定されます。また、このマーカーは、挿入されたテキストの後を指すように更新されます。連続した出力が順番に挿入されるのはこのためです。

通常のフィルタ関数では、出力がバッファに直接挿入されるときと同様にこのマーカーを使ってください。フィルタ関数のよい例としては、この項の終りにある `process-mark` を使ったフィルタ関数があります。

プロセスとの通信用のバッファにユーザーが入力を行う場合、前回の出力と新しい入力を区別するのにプロセス・マーカーが役立ちます。

set-process-buffer *process buffer* Function

この関数は、 *process* に関連付けられているバッファを *buffer* に設定します。 *buffer* が `nil` ならば、プロセスにはバッファは関連付けられません。

get-buffer-process *buffer-or-name* Function

この関数は、 *buffer-or-name* に関連付けられているプロセスを返します。バッファに複数のプロセスが関連付けられている場合は、そのうちの一つが選ばれます (現時点では、最後に作成されたバッファが選ばれます)。通常は、一つのバッファに複数のプロセスを関連付けるのはよくありません。

```
(get-buffer-process "*shell*")
     ⇒ #<process shell>
```

プロセスのバッファを kill するとそのプロセスは削除され、サブプロセスは `SIGHUP` シグナルで kill されます (33.8 節「プロセスへのシグナルの送信」 p.568 参照)。

33.9.2 プロセス・フィルタ関数

プロセス『フィルタ関数 (filter function)』とは、関連付けられているプロセスからの標準出力を受け取る関数です。プロセスにフィルタがある場合、そのプロセスからの出力のすべてがフィルタに渡されます。プロセス・バッファは、フィルタがない場合だけ、プロセスからの出力を直接受け取るために使われます。

フィルタ関数は、関連付けられているプロセスと文字列 (プロセスの出力) の二つの引数を取らなければなりません。この条件を満たせば、フィルタ関数は出力に対して任意の処理を行うことができます。

フィルタ関数は、 Emacs が待機中のとき (たとえば、端末入力待ち、時間の経過待ち、プロセスの出力待ちなど) だけ動作します。これによって、ほかの Lisp プログラムの実行中に予期せぬタイミングでフィルタを実行することによる問題を避けることができます。 Emacs を明示的に待機状態にしてフィルタ関数が実行されるようにするには、 `sit-for`、 `sleep-for` (18.7 節「時間の経過あるいは入力を待つ」 p.283 参照)、 `accept-process-output` (33.9.3 節「プロセスからの出力の受け取り」 p.573 参照) のいずれかを呼び出します。また、 Emacs は、コマンド・ループで入力を読み出しているときにも待機状態になります。

通常、フィルタ関数内では中断は禁止されています。中断が禁止されていなければ、コマンド・レベルでの `C-g` の入力やユーザー・コマンドの中断による影響を予測できなくなります。フィルタ関数内で中断を有効にするには、 `inhibit-quit` に `nil` を束縛してください。 18.8 節「とりやめ」 p.284 を参照してください。

フィルタ関数の実行中に発生したエラーは自動的に捕捉されるため、フィルタ関数が起動したときに動作していたプログラムの実行は停止しません。しかし、 `debug-on-error` が `nil` 以外ならば、エラーの捕捉は行なわれません。これによって、 Lisp デバッガを使ってフィルタ関数をデバッグできます。 15.1 節「Lisp のデバッガ」 p.193 を参照してください。

多くのフィルタ関数は、フィルタ関数がないときの Emacs の動作をまねて、ほとんどの場合テキストをプロセスのバッファに挿入します。このようなフィルタ関数において、バッファに確実に挿入できるようにするためには、 `set-buffer` を使わなければなりません。現在のバッファ

を半永久的に設定しないようにするため、このようなフィルタ関数では、 unwind-protect を使って、以前に現在のバッファとして使われていたバッファを復元できるようにしなければなりません。また、プロセス・マーカーを更新し、場合によってはポイントの値を更新しなければなりません。これらの作業の実現方法を次に示します。

```
(defun ordinary-insertion-filter (proc string)
  (let ((old-buffer (current-buffer)))
    (unwind-protect
        (let (moving)
          (set-buffer (process-buffer proc))
          (setq moving (= (point) (process-mark proc)))
          (save-excursion
            ;; テキストを挿入し、プロセス・マーカーを移動します。
            (goto-char (process-mark proc))
            (insert string)
            (set-marker (process-mark proc) (point)))
          (if moving (goto-char (process-mark proc))))
      (set-buffer old-buffer))))
```

save-excursion に現在のバッファを復元させずに、 unwind-protect を明示的に使っているのは、 goto-char によるポイントの変化を有効にするためです。

新しいテキストを受け取るたびにフィルタ関数がプロセス・バッファを強制的に表示させるには、次に示す行を unwind-protect の直前に挿入してください。

```
(display-buffer (process-buffer proc))
```

以前のポイントの位置に関係なく、ポイントを新しい出力の終端に強制的に移動するには、変数 moving を削除し、 goto-char を無条件に呼び出してください。

以前のバージョンの Emacs では、正規表現によく検索や照合を行うフィルタ関数は、必ず照合データを明示的に保存、復元しなければなりませんでした。現在では、 Emacs がこの作業を自動的に行ってくれるため、フィルタ関数側で明示的に行う必要はなくなりました。 30.6 節「一致データ」p.534 を参照してください。

プロセスのバッファに出力を書き込むフィルタ関数は、バッファがまだ有効かどうかをチェックしなければなりません。無効な (dead) バッファに出力を挿入しようとすると、エラーが発生します。バッファが無効である場合、 (buffer-name (process-buffer *process*)) は nil を返します。

フィルタ関数に渡される出力のひとまとまりのサイズは不定です。たとえば、出力を 1 行ごとに複製するプログラムが一度に 200 文字を送ってきたり、その次には 40 文字を 5 回送ってくることがあります。

set-process-filter *process filter* Function

この関数は、 *process* にフィルタ関数 *filter* を設定します。 *filter* が nil ならば、プロセスにはフィルタは設定されません。

process-filter *process* Function

この関数は、 *process* のフィルタ関数を返します。フィルタ関数がなければ、 nil を返します。

フィルタ関数の使い方の例を次に示します。

```
(defun keep-output (process output)
   (setq kept (cons output kept)))
     ⇒ keep-output
(setq kept nil)
     ⇒ nil
(set-process-filter (get-process "shell") 'keep-output)
     ⇒ keep-output
(process-send-string "shell" "ls ~/other\n")
     ⇒ nil
kept
     ⇒ ("lewis@slug[8] % "
"FINAL-W87-SHORT.MSS    backup.otl              kolstad.mss~
address.txt             backup.psf              kolstad.psf
backup.bib~             david.mss               resume-Dec-86.mss~
backup.err              david.psf               resume-Dec.psf
backup.mss              dland                   syllabus.mss
"
"#backups.mss#          backup.mss~             kolstad.mss
")
```

33.9.3 プロセスからの出力の受け取り

通常、非同期サブプロセスからの出力は、時間の経過や端末入力などのなんらかの外部イベントを Emacs が待っているときだけ到着します。しかし、出力が特定の時点で到着するように Lisp プログラムから指定したり、プロセスからの出力が到着するまで Lisp プログラムが待機できると便利なことがあります。

accept-process-output &optional *process seconds millisec* Function

この関数は、まだ完了していないプロセスからの出力を Emacs が読み出すようにします。出力は、プロセスに関連付けられているバッファに挿入されるか、フィルタ関数に渡されます。 *process* が `nil` 以外ならば、この関数は、 *process* から何らかの出力を受け取るまで戻りません。

seconds 引数と *millisec* 引数には、タイムアウトまでの時間を指定できます。 *seconds* 引数には秒単位の時間を指定し、 *millisec* 引数にはミリ秒単位の時間を指定します。二つの時間は加算され、 `accept-process-output` は、サブプロセスからの出力の有無にかかわらず、指定された時間が経過すると戻ります。

seconds 引数は整数でなくてもかまいません。 *seconds* 引数が浮動小数点数ならば、この関数は、小数点以下の秒数も含めて待機します。一部のシステムでは、秒数として整数値しかサポートしていません。そのようなシステムでは、 *seconds* は 0 に向かって丸められます。小数点以下の秒数をサポートしていないシステムで *millisec* に 0 以外の値を指定すると、エラーが発生します。

整数でない秒数での待機をサポートしていないオペレーティングシステムもあります。そのようなシステムでは、 *millisec* に 0 以外の値を指定するとエラーが発生します。

`accept-process-output` 関数は、何らかの出力を受け取った場合は `nil` 以外を返します。出力が到着する前にタイムアウト時間が経過した場合は、 `nil` を返します。

33.10 監視関数: プロセスのステータス変化の検出

『プロセスの監視関数 (process sentinel)』とは、 (Emacs から送られるか、プロセス自体の動作によって発生した) シグナルによる終了、停止、再開などの何らかの理由によって、関連付けられているプロセスのステータスが変化するたびに呼び出される関数です。また、プロセスの監視関数は、プロセスが終了するときにも呼び出されます。監視関数は、イベントが発生したプロセスと、イベントの種類を示す文字列の二つの引数を受け取ります。

イベントを示す文字列は次のいずれかのようになります。

- `"finished\n"`.
- `"exited abnormally with code` *exitcode*`\n"`.
- `"`*name-of-signal*`\n"`.
- `"`*name-of-signal* `(core dumped)\n"`.

監視関数は、 (端末入力待ちや時間経過待ち、プロセスの出力待ちなど) Emacs が待機中のときだけ動作します。ほかの Lisp プログラムの実行中に予期せぬタイミングで監視関数が実行さえることによる問題を避けることができます。 Emacs を明示的に待機状態にして監視関数が実行されるようにするには、 `sit-for`、 `sleep-for` (18.7 節「時間の経過あるいは入力を待つ」 p.283 参照)、 `accept-process-output` (33.9.3 節「プロセスからの出力の受け取り」 p.573 参照) のいずれかを呼び出します。また、 Emacs は、コマンド・ループで入力を読み出しているときにも待機状態になります。

通常、監視関数の中では中断は禁止されています。中断が禁止されていなければ、コマンド・レベルでの `C-g` の入力やユーザー・コマンドの中断による影響を予測できなくなります。監視関数の中で中断を有効にするには、 `inhibit-quit` に `nil` を束縛してください。 18.8 節「とりやめ」 p.284 を参照してください。

プロセスのバッファに出力を書き込む監視関数は、バッファがまだ有効かどうかをチェックしなければなりません。無効な (dead) バッファに出力を挿入しようとすると、エラーが発生します。バッファが無効である場合、 `(buffer-name (process-buffer` *process*`))` は `nil` を返します。

監視関数の実行中に発生したエラーは自動的に捕捉されるため、見張り番が起動したときに動作していたプログラムの実行は停止しません。しかし、 `debug-on-error` が `nil` 以外ならば、エラーの捕捉は行なわれません。これによって、 Lisp デバッガを使って監視関数をデバッグできます。 15.1 節「Lisp のデバッガ」 p.193 を参照してください。

以前のバージョンの Emacs では、正規表現によく検索や照合を行う見張り番は、必ず照合データを明示的に保存、復元しなければなりませんでした。現在では、 Emacs がこの作業を自動的に行ってくれるため、監視関数側で明示的に行う必要はなくなりました。 30.6 節「一致データ」 p.534 を参照してください。

set-process-sentinel *process sentinel* Function

この関数は監視関数 *sentinel* を *process* に関連付けます。 *sentinel* が `nil` ならば、プロセスの監視関数はなくなります。監視関数がない場合、デフォルトでは、プロセスのステータスが変化したときにプロセスのバッファにメッセージが挿入されます。

```
(defun msg-me (process event)
   (princ
     (format "Process: %s had the event `%s'" process event)))
(set-process-sentinel (get-process "shell") 'msg-me)
     ⇒ msg-me
(kill-process (get-process "shell"))
     ⊣ Process: #<process shell> had the event `killed'
     ⇒ #<process shell>
```

process-sentinel *process* Function

この関数は、 *process* の監視関数を返します。監視関数がない場合は、 `nil` を返します。

waiting-for-user-input-p Function

監視関数またはフィルタ関数が動作している間、この関数は、監視関数またはフィルタ関数が呼び出された時点で Emacs がユーザーからのキーボード入力を待っていたならば `nil` 以外を返します。そうでない場合は、 `nil` を返します。

33.11 トランザクション待ち行列

トランザクションによるサブプロセスとのより便利な通信を行うには、『トランザクション待ち行列 (transaction queue)』を使います。まず、 `tq-create` を使って、指定したプロセスとの通信を行うトランザクション待ち行列を作成します。次に、 `tq-enqueue` を呼び出して、トランザクションを送ります。

tq-create *process* Function

この関数は、 *process* と通信するためのトランザクション待ち行列を作成して返します。 *process* 引数は、バイトのストリームを送受信できるサブプロセスでなければなりません。サブプロセスの種類としては、子プロセス、 (別のマシン上の) サーバーへの TCP 接続などがあります。

tq-enqueue *queue question regexp closure fn* Function

この関数は、待ち行列 *queue* にトランザクションを送ります。待ち行列を指定すると、対象とするサブプロセスを指定する効果があります。

question は、トランザクションを開始する、こちら側からのメッセージです。 *fn* 引数は、対応する応答が帰ってきたときに呼び出される関数です。この関数は、 *closure* と受け取った応答の二つの引数で呼び出されます。

regexp 引数は、応答全体だけに一致する正規表現です。 `tq-enqueue` は、これを使って応答の終りを判断します。

`tq-enqueue` 自体の戻り値に意味はありません。

tq-close *queue* Function

トランザクション待ち行列 *queue* をシャットダウンし、完了していないトランザクションがすべて完了するのを待ち、接続または子プロセスを終了させます。

トランザクション待ち行列は、フィルタ関数を使って実装されています。 33.9.2 節「プロセス・フィルタ関数」p.571 を参照してください。

33.12 ネットワーク接続

Emacs Lisp プログラムは、同じマシンまたは別のマシンにあるほかのプロセスへの TCP ネットワーク接続をオープンできます。ネットワーク接続は、 Lisp プログラムからはサブプロセスと同様に扱われ、プロセス・オブジェクトで表現されます。しかし、通信相手のプロセスは Emacs プロセスの子プロセスではないため、通信相手のプロセスを kill したり、シグナルを送ることはできません。できるのはデータの送受信だけです。 `delete-process` は接続をクローズしますが、相手側のプロセスを kill しません。接続のクローズへの対処は、相手側のプロセスが決定しなければなりません。

ネットワーク接続を表現するプロセス・オブジェクトとサブプロセスを表現するプロセス・オブジェクトを区別するには、 `process-status` を使います。この関数は、ネットワーク接続の場合は必ず `open` または `closed` を返します。本当のサブプロセスの場合は、この二つの値のどちらも返しません。 33.6 節「プロセス情報」p.565 を参照してください。

open-network-stream *name buffer-or-name host service* Function

この関数は、サービスに関する TCP 接続をホストに対してオープンします。この関数は、接続を表現するプロセス・オブジェクトを返します。

name 引数は、プロセス・オブジェクトの名前を指定します。この名前は、名前を一意にするために必要に応じて修正されます。

buffer-or-name 引数は、接続が関連付けられるバッファです。接続からの出力を処理するフィルタ関数を指定しなければ、接続からの出力はこのバッファに挿入されます。 *buffer-or-name* が `nil` ならば、接続はどのバッファにも関連付けられません。

host 引数と *service* 引数は、接続先を指定します。 *host* はホスト名 (文字列) で、 *service* は定義済みのネットワーク・サービスの名前 (文字列) またはポート番号 (整数) です。

34 OS インタフェース

この章では、 Emacs を立ち上げたり Emacs から抜けたりすること、オペレーティング・システム環境の値を取得すること、端末と入出力とフロー制御について説明します。

関連する情報については B.1 節「Emacs を作る」p.645 を参照してください。また、端末と画面にまつわるオペレーティング・システムの状態、情報については、第 35 章「Emacs の表示」p.601 を参照してください。

34.1 Emacs の立ち上げ

この節では、 Emacs が立ち上げ時に何をするか、そして、その動作をどのようにカスタマイズするかについて述べます。

34.1.1 立ち上がりの際の一連の動作一覧

Emacs の起動時には以下の順序で実行されます (これは 'startup.el' に書かれています)。

1. ウィンドウ・システムを使っているときは、ウィンドウ・システムの初期化ライブラリをロードします。ライブラリの名前は 'term/*windowsystem*-win.el' です。
2. 初期化オプションを処理します (オプションによってはもっと前に処理するものもあります)。
3. 必要に応じて、 X ウィンドウのフレームとフェースを初期化します。
4. 正規フック before-init-hook を実行します。
5. もし '-no-site-file' を指定していなければ、ライブラリ 'site-start' をロードします。普通、ライブラリの名前は 'site-start.el' です。
6. コマンド行に '-q' が指定されていなければ、ファイル '~/.emacs' をロードします (これは '-batch' モードでは行ないません)。 '-u' オプションを指定すると、 '~' の代わりに誰のホーム・ディレクトリを使うかを指定することができます。
7. inhibit-default-init が非 nil ならば、ライブラリ 'default' をロードします ('-batch' モードやコマンド行で '-q' を指定してあるときは、ロードを行ないません)。普通、ライブラリの名前は 'default.el' です。
8. 正規フック after-init-hook を実行します。
9. initial-major-mode にしたがって主モードの設定を行ないます。ただし、この時点ではまだ '*scratch*' がカレントで、しかもまだ Fundamental モードです。
10. バッチ・モードでないときか、あるいはウィンドウ・システムを使っているとき、端末固有の Lisp ファイルがもしあれば、ロードします。
11. inhibit-startup-echo-area-message によって抑止されていなければ、最初のエコー領域メッセージを表示します。
12. コマンド行の動作引数を処理します。
13. term-setup-hook を実行します。

14. 選択されたフレームのパラメータを初期化ファイルの指定どおりに修正する、 `frame-notice-user-settings` を呼び出します。
15. `window-setup-hook` を実行します。 35.16 節「ウィンドウ・システム」 p.622 を参照してください。
16. (少し前の段階で) コマンド行引数が残ってなく、 `inhibit-startup-message` の値が `nil` ならば、 copyleft、無保証性、基本的利用のための情報を表示します。

inhibit-startup-message User Option

この変数は、 (無保証性などを示す) 最初の立ち上げメッセージを取りやめます。非 `nil` の場合、メッセージを表示しません。

立ち上げメッセージの内容を熟知したら、個人的な初期化ファイルの中でこの変数を設定すればよいようになっています。新しいユーザが見ることになっている情報の取得が妨げられるので、新しいユーザの初期化ファイルや複数のユーザに影響の出るようなやり方で、この変数を設定してはいけません。

inhibit-startup-echo-area-message User Option

この変数は、最初のエコー領域メッセージの表示を制御します。立ち上げ時のエコー領域メッセージの表示は、以下の形式のテキストを ‘`.emacs`’ ファイルに追加すれば抑止できます。

```
(setq inhibit-startup-echo-area-message
      "your-login-name")
```

単純に `inhibit-startup-echo-area-message` へログイン名を設定することは、メッセージを抑制するのに不十分です。 Emacs は陽に ‘`.emacs`’ が上記のような式を含むか調べています。ログイン名は Lisp の文字列定数として式に現われなければなりません。

こうして、あなたは簡単にメッセージを抑止できますが、誰かがやみくもにあなたの ‘`.emacs`’ ファイルを転記しても、その人へのメッセージは抑止されません。

34.1.2 初期化ファイル‘`.emacs`’

Emacs を立ち上げたとき、通常はあなたのホーム・ディレクトリ配下にあるファイル ‘`.emacs`’ をロードしようとします。このファイルがある場合は Lisp コードを含んでいなければなりません。これをあなたの『初期化ファイル (init file)』といいます。コマンド行スイッチの ‘`-q`’ と ‘`-u`’ が、初期化ファイルの利用に影響を与えます。 ‘`-q`’ は初期化ファイルをロードしないことを、そして ‘`-u`’ はあなたの初期化ファイルの代わりに、指定したユーザの初期化ファイルをロードすることを指定します。 *The GNU Emacs Manual* の “Entering Emacs” 節を参照してください。

サイトには、 ‘`default.el`’ という名前のライブラリである、『デフォルト初期化ファイル (default init file)』があることもあります。 Emacs は ‘`default.el`’ ファイルを見つけるのに、ライブラリの標準検索パスを使用します (13.1 節「プログラムでロードする方法」 p.171 参照)。 Emacs の配布物件にはこのファイルを含んでいません。サイトで固有なカスタマイズをすることができます。デフォルト初期化ファイルがあれば、バッチ・モードの場合と、 ‘`-q`’ を指定した場

合を例外として、 Emacs を立ち上げるときにいつもロードされます。個人の初期化ファイルがあれば、そちらが先にロードされます。そこで `inhibit-default-init` を非 `nil` の値に設定すると、 Emacs は ‘`default.el`’ のロードを取りやめます。

もう一つ ‘`site-start.el`’ というサイトのカスタマイズ用ファイルがあります。 Emacs は、ユーザの初期化ファイルの前に、このファイルをロードします。このファイルのロードは、オプション ‘`-no-site-file`’ で禁止できます。

site-run-file Variable

この変数は、ユーザの初期化ファイルの前にロードする、サイトのカスタマイズのファイルを指定します。通常の値は `"site-start"`です。

もし ‘`.emacs`’ のコードが大きくなりすぎたら、 ‘*something*`.el`’ のようにファイル名を変えて、それをバイトコンパイルしておき (第 14 章「バイトコンパイル」 p.181 参照)、そして ‘`.emacs`’ ファイルから `load` を使って、ほかのファイルをロード (第 13 章「ロード」 p.171 参照) した方がいいでしょう。

‘`.emacs`’ ファイルでありがちな、望ましいカスタマイズのいろいろなことについては、 *The GNU Emacs Manual* の “Init File Examples” 節を参照してください。

inhibit-default-init User Option

この変数は、 Emacs がデフォルト初期化ライブラリをロードするのを禁止します。値が非 `nil` の場合、デフォルトのライブラリをロードしません。デフォルト値は `nil` です。

before-init-hook Variable

after-init-hook Variable

この二つの正規フックはユーザの初期化ファイル、 ‘`default.el`’、 ‘`site-start.el`’ のロードの直前と直後に作動されます。

34.1.3 端末固有の初期化

各端末の型には独自の Lisp ライブラリがあって、 Emacs が作動するときにはその端末型のライブラリをロードします。 *termtype* という名前の端末型のライブラリの名前は、 ‘`term/`*termtype*’ です。ほかのファイルの場合と同様に、 Emacs は `load-path` のディレクトリを検索して、 ‘`.elc`’ や ‘`.el`’ の接尾辞のついたファイルがないか調べます。通常、端末固有のライブラリは、 ‘`emacs/lisp/term`’ という Emacs Lisp ライブラリの保管してある ‘`emacs/lisp`’ ディレクトリのサブディレクトリに置いてあります。

ライブラリの名前は、変数 `term-file-prefix` の値と端末型を結合して構築します。通常 `term-file-prefix` は、 `"term/"`という値をもっています。この値を変更することはおすすめできません。

端末固有ライブラリの通常の機能は、特殊キーの送出するシーケンスを Emacs が理解できるようにすることです。端末にたいする Termcap の項目が、端末のファンクション・キーの全部を指定してはいない場合、このライブラリで `function-key-map` の設定や追加をすることもあります。 34.8 節「端末入力」 p.592 を参照してください。

端末型の名前がハイフンを含むとき、ライブラリの名前としては最初のハイフンの前までが有効です。したがって `aaa-48' と `aaa-30-rv' という端末型は、両方とも `term/aaa' というライブラリを使います。必要であれば、ライブラリは (getenv "TERM") を評価して端末型の完全な名前を使うことができます。

`.emacs' ファイルで変数 term-file-prefix を nil に設定することで、端末固有ライブラリのロードを止めることができます。この機能は、あなたが独創的なカスタマイズを試してみるときに役に立ちます。

変数 term-setup-hook を設定することによって、端末固有ライブラリの動作の一部を無効にすることができます。これは `.emacs' ファイルと端末固有ライブラリの両方をロードした後に、つまり、 Emacs の初期化の最後に run-hooks を使って作動させる正規フックです。独自のライブラリをもたない端末の初期化の定義に、この変数を使うことができます。 20.4 節「フック」 p.335 を参照してください。

term-file-prefix Variable

もし term-file-prefix が非 nil ならば、 Emacs は、端末固有の初期化ファイルを以下のようにロードします。

```
(load (concat term-file-prefix (getenv "TERM")))
```

端末初期化ファイルのロードを望まない場合は、 `.emacs' ファイルで term-file-prefix 変数を nil に設定することができます。そうするには、 `.emacs' に (setq term-file-prefix nil) と書きます。

term-setup-hook Variable

この変数は、 `.emacs'、デフォルト初期化ファイル (もしあれば)、端末固有 Lisp ファイルのロードの後に Emacs が作動させる正規フックです。

term-setup-hook を使って、端末固有ファイルで施された定義を無効にすることができます。

関連する機能について 35.16 節「ウィンドウ・システム」 p.622 の window-setup-hook を参照してください。

34.1.4 コマンド行引数

コマンド行引数を使って、 Emacs を立ち上げるときにいろいろな動作をさせることができます。 1 日に 1 回よりも多く Emacs を立ち上げる必要はありませんし、何日も Emacs を立ち上げたままにすることもありますから、コマンド行引数を使うことはほとんどないでしょう。実際、こうした習慣こそが不必要に多く Emacs を殺したり、再起動したりすることを助長しているのであって、このようなものを使う習慣をなくすことの方が最高なのです。これらのオプションが存在するのは二つの理由があります。 (ほかのプログラムに起動されるときのための) ほかのエディタとの互換性と、 Lisp プログラムをシェル・スクリプトから起動することができるようにすることです。

この節では Emacs がコマンド行引数をどのように処理するのかと、どのようにそれをカスタマイズするのかについて述べます。

command-line Function

この関数は、 Emacs が呼ばれたときのコマンド行を解析し、それを処理し、ユーザの '.emacs' ファイルをロードし、立ち上げメッセージの表示を行ないます。

command-line-processed Variable

この変数の値は、一度コマンド行を処理すると `t` になります。

もし `dump-emacs` を呼び出して Emacs を再ダンプするときは、新しくダンプされた Emacs に新しいコマンド行引数を処理させるために、この変数を `nil` に設定したくなることでしょう。

command-switch-alist Variable

この変数の値は、ユーザ定義のコマンド行オプションとそれに対するハンドラ関数の alist です。この変数に要素を追加できるようになっています。

『コマンド行オプション (command line option)』とは以下のようなコマンド行の引数のことです。

-*option*

`command-switch-alist` の要素は以下のようになっています。

(*option* . *handler-function*)

handler-function は *option* を処理するために呼び出され、オプション名を唯一の引数として受け取ります。

場合によっては、オプションにコマンド行中の引数がともなうことがあります。そのような場合、 *handler-function* は `command-line-args-left` の中から、残りの全部の引数をとりだすことができます (コマンド行引数全体のリストは、 `command-line-args` にあります)。

コマンド行引数は、 '`startup.el`' ファイルの `command-line-1` 関数で解析します。*The GNU Emacs Manual* の "Command Line Switches and Arguments" 節を参照してください。

command-line-args Variable

この変数の値は、 Emacs に渡されたコマンド行引数のリストです。

command-line-functions Variable

この変数の値は認識されないコマンド行引数を処理する関数のリストです。処理中の次の引数が特別な意味をもっていないときは常に、このリストの関数がリストの先頭から呼び出され、どれか一つが非 `nil` の値を返すまで順に繰り返します。

この関数群は引数なしで呼び出されます。考慮中のコマンド行引数は `argi` 変数でアクセスできます。残りの引数 (処理中のものをのぞく) は、 `command-line-args-left` にあります。

関数が引数 `argi` を認識し処理を行なうとき、その引数を処理したことを示すために、非 `nil` の値を返さなければなりません。もし後続する引数も処理した場合、 `command-line-args-left` から消去して、そのことを示します。

もしすべての関数が `nil` を返したら、引数は読み込むファイルの名前として使います。

34.2 Emacs から抜ける

Emacs から抜ける方法は二つあります。 Emacs ジョブを殺せば完全に終了してしまいます。後でまた Emacs プロセスに入れるように、サスペンドすることもできます。ログ・アウトするときをのぞけば、実際それほど Emacs を殺すことはありません。サスペンドすることが、ほとんどです。

34.2.1 Emacs を殺す

Emacs を殺すということは Emacs プロセスの実行を終わらせるということです。通常は、親プロセスが制御を取り戻します。 Emacs を殺す低水準プリミティブは、 `kill-emacs` です。

kill-emacs &optional *exit-data* Function

この関数は、 Emacs のプロセスを抜けて、 kill します。

exit-data が整数の場合、それを Emacs プロセスの exit ステータスとして用います (これは主にバッチ操作のときに便利です。 34.12 節「バッチ・モード」 p.599 を参照してください)。

もし *exit-data* が文字列ならば、シェル (あるいは次に入力を読み込むプログラムなら何でも) が読み込めるよう、内容を端末の入力バッファに放り込みます。

Emacs プロセスの全情報は、セーブしたファイルをのぞいて、 Emacs が殺されたときになくなります。 Emacs を殺すといろいろな作業結果が失われてしまうので、保存の必要なバッファや走行中のサブプロセスが存在すると、 Emacs はユーザに本当に終了するべきか否かを尋ねます。これは関数 `save-buffers-kill-emacs` によります。

kill-emacs-query-functions Variable

標準的な質問の後で、 `save-buffers-kill-emacs` は、リスト `kill-buffer-query-functions` にある関数を、先頭から引数なしで呼び出します。これらの関数でユーザに付加的な確認の質問をすることができます。一つでも非 `nil` を返すと、 Emacs は殺されません。

kill-emacs-hook Variable

この変数は、正規フックです。 `save-buffers-kill-emacs` は、いったんすべてのファイルのセーブと確認を終えた後で、これはこのフックにある関数群を作動させます。

34.2.2 Emacs のサスペンド

『Emacs のサスペンド (suspending Emacs)』とは、 Emacs を一時的に停止させ、通常はシェルであろう上位のプロセスに、制御を戻すことをいいます。これは後から同じバッファ、同じ kill リング、同じ取り消し履歴などがある同じ Emacs プロセスで、編集を再開させること

ができます。 Emacs を再開させるには、親シェルで適切なコマンドを (大体 `fg` ですが) 使います。

オペレーティング・システムによっては、ジョブのサスペンドをサポートしていないこともあります。このようなシステムでの「サスペンド」は、実際、 Emacs の子プロセスとして新しいシェルを一時的に立ち上げるだけです。このときシェルを抜けると Emacs に戻ります。

X のようなウィンドウ・システムを使っている場合、再開すべき親がいないかもしれないですし、別なウィンドウに移動するだけでシェルなどのほかのジョブに入力を与えることができるので、サスペンドは役に立ちません。そのため Emacs が X のクライアントの場合、サスペンドは禁止してあります。

suspend-emacs *string* Function

この関数は、 Emacs を停止させ上位のプロセスに制御を戻します。上位プロセスが Emacs を再開させると、 `suspend-emacs` は `nil` を Lisp の呼び出し側に返します。

もし *string* が非 `nil` ならば、その文字は Emacs の上位シェルに端末入力として読み込まれます。 (訳注: 文字を読み込んだ) 結果だけが現われます。

`suspend-emacs` は、サスペンドの前に正規フック `suspend-hook` を作動させます。 `suspend-hook` は、 Emacs バージョン 18 では正規フックではありませんでした。この値は単一の関数で、もしその値が非 `nil` ならば、 `suspend-emacs` はサスペンドをせずすぐに戻っていました。

ユーザが Emacs を再開させた後、 `suspend-emacs` は、正規フック `suspend-resume-hook` を作動させます。 20.4 節「フック」 p.335 を参照してください。

`no-redraw-on-reenter` が非 `nil` でなければ、再開の直後で全画面の再表示を行ないます。 (35.1 節「画面の再表示」 p.601 参照)。

以下の例で Emacs がサスペンドした後 ‘`pwd`’ はエコーされません。しかし、シェルによって読み込まれ実行されます。

```
(suspend-emacs)
     ⇒ nil

(add-hook 'suspend-hook
          (function (lambda ()
                      (or (y-or-n-p
                            "Really suspend? ")
                          (error "Suspend cancelled")))))
     ⇒ (lambda nil
          (or (y-or-n-p "Really suspend? ")
              (error "Suspend cancelled")))
(add-hook 'suspend-resume-hook
          (function (lambda () (message "Resumed!"))))
     ⇒ (lambda nil (message "Resumed!"))
(suspend-emacs "pwd")
     ⇒ nil
---------- Buffer: Minibuffer ----------
Really suspend? y
---------- Buffer: Minibuffer ----------

---------- Parent Shell ----------
lewis@slug[23] % /user/lewis/manual
lewis@slug[24] % fg
```

```
---------- Echo Area ----------
Resumed!
```

suspend-hook Variable

この変数は、サスペンドの前の正規フックです。

suspend-resume-hook Variable

この変数は、サスペンドの後の正規フックです。

34.3 OS 環境

Emacs は、オペレーティング・システム環境にある変数へのアクセスを、さまざまな関数によって提供しています。これらの変数にはシステムの名前やユーザの UID などが含まれます。

system-type Variable

この変数の値は、 Emacs が機能しているオペレーティング・システムの型を示すシンボルです。以下にとりうる値の一覧表を示します。

`aix-v3` AIX.

`berkeley-unix`
Berkeley BSD.

`dgux` Data General DGUX operating system.

`gnu` A GNU system using the GNU HURD and Mach.

`hpux` Hewlett-Packard HPUX operating system.

`irix` Silicon Graphics Irix system.

`linux` A GNU system using the Linux kernel.

`ms-dos` Microsoft MS-DOS “operating system.”

`next-mach` NeXT Mach-based system.

`rtu` Masscomp RTU, UCB universe.

`unisoft-unix`
UniSoft UniPlus.

`usg-unix-v`
AT&T System V.

`vax-vms` VAX VMS.

`windows-nt`
Microsoft windows NT.

`xenix` SCO Xenix 386.

絶対的に必要ではないのなら、私たちはきれいな区別をするために新しいシンボルをつけ足すことは望みません。異なるオペレーティング・システムを区別するためには、`system-configuration` を利用することを推奨します。

system-configuration Variable

この変数は、システムのハードウエア / ソフトウエアの三部構成名を文字列として保持します。この文字列の一部分を比較するときは、 `string-match` を使うと便利です。

system-name Function

この関数は、走っているマシンの名前を返します。

```
(system-name)
     ⇒ "prep.ai.mit.edu"
```

シンボル `system-name` は、関数であると同時に変数でもあります。実際、関数の方は変数 `system-name` が現在保持している値が何であれ、それを返します。したがって Emacs がシステムの名前について混乱している状況なら、変数 `system-name` に値を設定することも可能です。この変数は、フレームのタイトルを構成するときにも役に立ちます (26.4 節「フレーム・タイトル」 p.443 参照)。

mail-host-address Variable

もしこの変数が非 `nil` ならば、 email アドレスを生成する目的で `system-name` の代わりに利用されます。たとえば、 `user-mail-address` のデフォルト値を構成するときに使用します。 34.4 節「ユーザの識別」 p.587 を参照してください。 (これは Emacs の立ち上がり時になされるので、実際に値が使われるのは、 Emacs をダンプしたときに保存していた値です。 B.1 節「Emacs を作る」 p.645 を参照してください。)

getenv *var* Function

この関数は、環境変数 *var* の値を文字列として返します。 Emacs 内部で環境変数の値は、 Lisp 変数の `process-environment` に保管します。

```
(getenv "USER")
     ⇒ "lewis"

lewis@slug[10] % printenv
PATH=.:/user/lewis/bin:/usr/bin:/usr/local/bin
USER=lewis
TERM=ibmapa16
SHELL=/bin/csh
HOME=/user/lewis
```

setenv *variable value* Command

このコマンドは *variable* という環境変数の値を *value* に設定します。引数は両方とも文字列です。この関数は、 `process-environment` を修正することではたらいています。 `let` でこの変数を束縛するのも実際、合理的に行なわれていることです。

process-environment Variable

この変数は、文字列のリストで、それぞれが一つの環境変数を表わします。関数 `getenv` と `setenv` はこの変数によってはたらいています。

```
process-environment
⇒ ("l=/usr/stanford/lib/gnuemacs/lisp"
    "PATH=.:/user/lewis/bin:/usr/class:/nfsusr/local/bin"
    "USER=lewis"
    "TERM=ibmapa16"
    "SHELL=/bin/csh"
    "HOME=/user/lewis")
```

path-separator Variable

この変数は、(環境変数にある) 検索パスのディレクトリを分離する文字を保管します。Unix と GNU システムでは `":"`で、MS-DOS と Windows NT では `";"`です。

invocation-name Variable

この変数は、Emacs が起動されたプログラム名を保管します。値はディレクトリ名を含まない文字列です。

invocation-directory Variable

この変数は、起動された Emacs 実行形式のディレクトリか、あるいはそれが決定できない場合は、`nil` を保管します。

installation-directory Variable

もし非 `nil` ならば、'`lib-src`' と '`etc`' のサブディレクトリを探すためのディレクトリです。これは Emacs が標準のインストール場所のディレクトリを見つけられなかったが、Emacs の実行可能形式を含むディレクトリから相対的に見つけることができたとき、非 `nil` になります。

load-average Function

この関数は、最近の 1 分間、5 分間、15 分間のロード・アベレージをリストにして返します。この値はシステムロード・アベレージに 100 を掛けたものです (ロード・アベレージは走行を試みているプロセスの数を示します)。

```
(load-average)
     ⇒ (169 48 36)

lewis@rocky[5] % uptime
 11:55am  up 1 day, 19:37,  3 users,
 load average: 1.69, 0.48, 0.36
```

emacs-pid Function

この関数は、Emacs プロセスのプロセス ID を返します。

setprv *privilege-name* &optional *setp getprv* Function

この関数は、VMS 特権の設定と解除を行ないます (Unix にはありません)。第 1 引数は文字列としての特権名です。第 2 引数 *setp* は `t` あるいは `nil` で、特権がオンか、オフかを示します。デフォルトで `nil` です。成功すると `t` を返し、そうでないと `nil` を返します。

第 3 引数 *getprv* が非 `nil` ならば、 `setprv` は特権を変更せずに、 `t` あるいは `nil` を返して、現在その特権があるかどうかを示します。

34.4 ユーザの識別

user-mail-address Variable

これは Emacs を使っているユーザの通常の email アドレスを保持します。 Emacs が立ち上がったとき、たいていは正しいデフォルト値を算出します。このデフォルト値が正しくないときはユーザが自分で値を設定します。

user-login-name &optional *uid* Function

もし *uid* を指定しなければ、この関数は、ログインしているユーザの名前を返します。もし環境変数 `LOGNAME` が設定されていたら、その値を使います。さもないと、もし環境変数 `USER` が設定されていたら、その値を使います。さもないと、値は実 UID でなく、実効 UID に基づいて取られます。

もし *uid* を指定したとき、値は *uid*(整数でなければなりません) に対応するユーザ名です。

```
(user-login-name)
     ⇒ "lewis"
```

user-real-login-name Function

この関数は、 Emacs の実 UID に対応した名前を返します。これは、実効 UID、環境変数 `LOGNAME` と `USER` を無視します。

user-full-name Function

この関数は、ユーザの氏名を返します。

```
(user-full-name)
     ⇒ "Bil Lewis"
```

シンボル `user-login-name`、 `user-real-login-name`、 `user-full-name` は、関数であると同様に変数でもあります。この関数群はこの変数群が保持する値を返します。この変数群で関数群の返す値を設定して、 Emacs を「だます」ことができます。この変数群はフレームのタイトルを構成するときにも役に立ちます (26.4 節「フレーム・タイトル」 p.443 参照)。

user-real-uid Function

この関数は、ユーザの実 UID を返します。

```
(user-real-uid)
     ⇒ 19
```

user-uid Function

この関数は、ユーザの実効 UID を返します。

34.5 1日の時刻

この節では現在の時刻と時間帯の決定方法を説明します。

current-time-string &optional *time-value* Function

この関数は、現在の時刻を人が読める文字列の形で返します。この文字列の形式は固定です。各部分 (訳注: 曜日の部分とか年の部分など) には常に同じ文字数が用いられます。このため、各部分を抽出するのに安心して `substring` を用いることができます。文字列の後ろに情報を拡張する場合を考慮すれば、文字は後ろからではなく、前から数えた方がいいでしょう。

引数 *time-value* が与えると、現在時刻の代わりに整形すべき時刻を指定します。引数は最初の二つの要素が整数であるリストでなければなりません。したがって `current-time`(以下を参照) や `file-attributes` (22.6.4 節「ファイルに関するそのほかの情報」p.361 参照) で得た値を使うことができます。

```
(current-time-string)
     ⇒ "Thu Aug 24 06:30:15 1995"
```

current-time Function

この関数は、システム時刻の値を三つの整数のリスト (*high low microsec*) として返します。整数 *high* と *low* は、 $high * 2^{16} + low$ のようにして (訳注: UTC で)1970 年 1 月 1 日 0 時 0 分 0 秒からの秒数を表わします。

第 3 要素 *microsec* は、最近の秒が始まってからのマイクロ秒数 (あるいは秒精度までしか返さないシステムでは 0) を示します。

最初の二つの要素は、関数 `file-attributes` で得られるような時刻の値と比較することができます。 22.6.4 節「ファイルに関するそのほかの情報」p.361 を参照してください。

current-time-zone &optional *time-value* Function

この関数は、ユーザの属している時間帯を返します。

値は (*offset name*) という形をしています。ここで *offset* は整数で UTC よりも進んでいる (グリニッチより東の) 秒数です。負の値はグリニッチよりも西であることを意味します。第 2 要素 *name* は文字列で時間帯の名称を表わします。双方とも夏時間の始まりと終わりに変化が生じます。夏時間の制度のないところでは常に一定です。

値を算出するのに必要な情報が、オペレーティング・システムからすべては与えられていない場合、両方の要素は `nil` です。

引数 *time-value* が与えられると、現在時刻の代わりに解析すべき時刻を指定します。引数は二つの整数を含むコンス・セルか、あるいは、先頭の二つの要素が整数のリストです。したがって `current-time` (上記参照) や `file-attributes` (22.6.4 節「ファイルに関するそのほかの情報」p.361 参照) で得られるような時刻を使うことができます。

34.6 時刻の変換

この関数群は、時刻の値 (2 個、 3 個の整数のリスト) から文字列、暦情報への変換を行ないます。また、暦情報から時刻の値へ変換する関数もあります。時刻の値は関数 `current-time` (34.5 節「1 日の時刻」 p.588 参照) と `file-attributes` (22.6.4 節「ファイルに関するそのほかの情報」 p.361 参照) から得ることができます。

format-time-string *format-string time* Function

この関数は、 *time* を *format-string* にしたがった文字列に変換します。引数 *format-string* は、時刻の一部分の置換を指示する '%' 列を含ませることができます。 '%' 列の意味の一覧は以下のとおりです。

'%a' 曜日の省略形を表わします。

'%A' 完全な曜日の名前を表わします。

'%b' 月の省略形を表わします。

'%B' 完全な月の名前を表わします。

'%c' これは '%x %X' の省略形です。

'%C' これはロケール特有の意味があります。デフォルトの C というロケールでは '%A, %B %e, %Y' と等価です。

'%d' これは 0 で埋め草した月の日付です。

'%D' これは '%m/%d/%y' の省略形です。

'%e' これは空白で埋め草した月の日付です。

'%h' これは '%mb' の省略形です。

'%H' これは時間 (00-23) を表わします。

'%I' これは時間 (00-12) を表わします。

'%j' 年間での日数 (001-366) を表わします。

'%k' これは空白で埋め草した時間 (0-23) です。

'%l' これは空白で埋め草した時間 (0-12) です。

'%m' これは月 (01-12) を表わします。

'%M' これは分 (00-59) を表わします。

'%n' これは改行を表わします。

'%p' これは必要に応じて 'AM' あるいは 'PM' を表わします。

'%r' これは '%I:%M:%S %p' の省略形です。

'%R' これは '%H:%M' の省略形です。

‘%S’ これは秒 (00-60) を表わします。

‘%t’ これはタブ文字を表わします。

‘%T’ これは ‘%H:%M:%S’ の省略形です。

‘%U’ 年間での週の数 (01-52) を表わします。週は日曜日からと仮定します。

‘%w’ これは数で表わした曜日です。日曜日は 0 です。

‘%W’ 年間での週の数 (01-52) を表わします。週は月曜からと仮定します。

‘%x’ これはロケール特有の意味があります。デフォルトの C というロケールでは ‘%D’ と等価です。

‘%X’ これはロケール特有の意味があります。デフォルトの C というロケールでは ‘%T’ と等価です。

‘%y’ 世紀での年数 (00-99) を表わします。

‘%Y’ これは世紀を含む西暦年を表わします。

‘%Z’ これは時間帯の省略形を表わします。

decode-time *time* Function

この関数は、時刻の値を暦情報に変換します。返す値は以下のような九つの要素のあるリストです。

(*seconds minutes hour day month year dow dst zone*)

要素の意味は、以下のとおりです。

sec 毎分ちょうどから経過した秒数で 0 から 59 までの整数です。

minute 毎時ちょうどから経過した分の数で 0 から 59 までの整数です。

hour 1 日が始まってから経過した時間の数で 0 から 23 までの整数です。

day 1 か月の日付で 1 から 31 までの整数です。

month 月を表わす 1 から 12 までの整数です。

year 西暦年で通常の値は 1900 より大きい整数です。

dow 週の曜日で 0 から 6 までの整数です。 0 は日曜日です。

dst 夏時間中は `t` でそうでなければ `nil` です。

zone 時間帯を示す整数でグリニッチからの秒数です。

Common Lisp での *dow* と *zone* の意味は違うので、注意してください。 (訳注: ちなみに 0 は月曜で、時間帯は時間単位。)

encode-time *seconds minutes hour day month year* &optional *zone* Function

この関数は、 `decode-time` の逆です。暦情報の 7 項目を時刻の値に変換します。引数の意味は、 `decode-time` の表を参照してください。

100 以下の年数もほかの年と同様に扱われます。 1900 以降の年の意味にしたければ、 `encode-time` を呼び出す前に自分で変えておいてください。

オプション引数 *zone* は、デフォルトで現行の時間帯と夏時間規則となります。指定したときは、 (`current-time-zone` で得られるような) リストか、あるいは (`decode-time` で得られるような) 整数が使えます。特に夏時間の修正はせずに指定した時間帯を使用します。

34.7 タイマーと遅延実行

指定した未来の時刻に関数を呼び出すことができます。

run-at-time *time repeat function* &rest *args* Function

この関数は、 *function* を引数 *args* とともに時刻 *time* に呼び出します。引数 *function* は後で呼び出す関数で、 *args* は呼び出すときに渡す引数です。時刻 *time* は文字列で指定します。

絶対時刻はいろいろな書式で指定することが可能です。全フィールドが整数の '*hour*:*min*:*sec* *timezone* *month*/*day*/*year*' という形式は受けつけられます。 `current-time-string` の返す値も有効です。

相対時刻は数量と単位で指定します。たとえば、

'`1 min`' は 1 分後です。

'`1 min 5 sec`'
: は 65 秒後です。

'`1 min 2 sec 3 hour 4 day 5 week 6 fortnight 7 month 8 year`'
: は 103 か月 123 日 と 10862 秒後です (訳注: fortnight とは 2 週間のこと)。

もし *time* が整数ならば、秒で測った相対時間です。

引数 *repeat* は呼出しを何回繰り返すか指定します。もし *repeat* が `nil` ならくり返しはせず、 *time* に一度呼び出されます。もし *repeat* が整数ならば、秒で測ったくり返しの期間を指定します。いずれにせよ *repeat* は、 *1* 回目の呼出しには影響しません。それは *time* で指定します。

`run-at-time` はスケジュールに入れた未来のこの動作を特定する、タイマーの値を返します。この値は後で `cancel-timer` を呼び出すときに使用することができます。

cancel-timer *timer* Function

以前に `run-at-time` が返した値の *timer* で要求していた動作を取消します。これは `run-at-time` を呼出しの効果を取り消します。指定時刻が来ても特別なことは何も起きません。

34.8 端末入力

この節では端末入力を記録し、操作するための関数と変数について述べます。関連した関数については第 35 章「Emacs の表示」p.601 を参照してください。

34.8.1 入力モード

set-input-mode *interrupt flow meta quit-char* Function

この関数は、キーボードから入力を読み込む時のモードを設定します。もし *interrupt* が「non-null」の場合、 Emacs は入力割込みを用います。もし `nil` の場合、 CBREAK モードを用います。 Emacs が直接 X と通信しているときは、通信の方法がわかっているので、この引数を無視して割込みを使います。

もし *flow* が非 `nil` ならば、 Emacs は端末への出力に XON/XOFF (`C-q`、 `C-s`) フロー制御を用います。これは CBREAK モード以外では何の影響も及しません。 34.11 節「フロー制御文字」p.597 を参照してください。

通常の設定はシステムに依存します。システムによっては何を指定しても常に CBREAK モードを使います。

引数 *meta* は 127 よりも大きい入力文字符号の扱いを制御します。もし *meta* が `t` ならば、 Emacs は、 8 ビット目を立てた文字をメタ文字に変換します。もし *meta* が `nil` ならば、 Emacs は 8 ビット目を無視します。これは端末がパリティービットとして使うときに必要です。もし *meta* が `t` でも `nil` でもないとき、 Emacs は 8 ビットの入力をそのまま使います。これはヨーロッパの 8 ビット文字集合を使う端末のとき便利です。

もし *quit-char* が非 `nil` なら、とりやめに使う文字を指定します。通常この文字は `C-g` です。 18.8 節「とりやめ」p.284 を参照してください。

`current-input-mode` 関数は、 Emacs が現在使っている入力モードの設定を返します。

current-input-mode Function

この関数は、現在のキーボード入力の読み込みモードを返します。これは `set-input-mode` の引数 (*interrupt flow meta quit*) に対応したリストを返します。

interrupt は、 Emacs が割込み入力を使っているとき非 `nil` です。もし `nil` なら Emacs は CBREAK モードを使用中です。

flow は Emacs が XON/XOFF (`C-q` と `C-s`) を端末出力のフロー制御に使うとき非 `nil` です。この値は *interrupt* が非 `nil` でないと、効果をもちません。

meta は、 Emacs が入力文字の 8 ビット目をメタビットとして扱うとき `t` です。 Emacs が全入力文字の 8 ビット目を落とすときは `nil` です。 Emacs が 8 ビットすべてを基本的な文字符号として使うときは、それ以外の値です。

quit は Emacs が今とりやめに使っている文字で、通常 `C-g` です。

34.8.2 入力イベントの変換

この節ではキー・シーケンスの一部になる前に、入力イベントを別なイベントに変換する機能について述べます。

extra-keyboard-modifiers Variable

この変数で Lisp プログラムでキーボードの修飾キーを「押す」ことができます。

1　`SHIFT` キー。

2　`LOCK` キー。

4　`CTL` キー。

8　`META` キー。

ユーザがキーボードのキーを押下するたびに、まるでビット・マスクで指定した修飾キーが押されているかのように、キーが変更されます。

X ウィンドウを使っているとき、プログラムで上記のように修飾キーを「押す」ことができます。それ以外の場合、 `CTL` と `META` キーだけ仮想的に押すことができます。

keyboard-translate-table Variable

この変数は、キーボード文字の変換テーブルです。コマンドのバインドを変更せずにキーボードのキーを並べ替えることができます。値は文字列か `nil` です。

もし `keyboard-translate-table` が文字列ならば、キーボードから読み込まれた各文字は、この文字列を参照し文字列中の文字列を代用します。もし文字列の長さが n ならば、n 以上の文字符号は変換しません。

以下の例では、 `keyboard-translate-table` を 128 文字の文字列に設定しています。そして文字 `C-s` と `C-\`、また、文字 `C-q` と `C-^` を交換するように設定しています。したがって、 `C-\` の押下は、通常 `C-s` を押下したときの全効果をもち、逆もまた同様です (これについてはさらに 34.11 節「フロー制御文字」 p.597 を参照してください)。

```
(defun evade-flow-control ()
  "C-s を C-\\ で C-q を C-^ で置き換える。"
  (interactive)
  (let ((the-table (make-string 128 0)))
    (let ((i 0))
      (while (< i 128)
        (aset the-table i i)
        (setq i (1+ i))))
    ;; C-s と C-\ の交換。
    (aset the-table ?\034 ?\^s)
    (aset the-table ?\^s ?\034)
    ;; C-q と C-^ の交換。
    (aset the-table ?\036 ?\^q)
    (aset the-table ?\^q ?\036)
    (setq keyboard-translate-table the-table)))
```

この変換は、端末から読み込んだ直後に行なうことに注意してください。 `recent-keys` やドリブル・ファイルのような記録保存機能は、変換の後に文字を記録します。

keyboard-translate *from to* Function

この関数は、 `keyboard-translate-table` を修正して、符号 *from* が符号 *to* に変換するようにします。必要に応じて変換テーブルの生成と拡張を行ないます。

function-key-map Variable

この変数は、普通のキャラクタ端末のファンクション・キーが送出する文字シーケンスを記述する、キーマップを保持します。このキーマップはほかのキーマップと同じデータ構造をしていますが、異なった用い方をします。これはイベントを読み込んでいる間の変換の指定をしています。

もし `function-key-map` が、あるキー・シーケンス *k* をあるベクタ *v* に「バインドする」ならば、 *k* がシーケンスのどこかに現われたとき、イベント *v* に置換されます。

たとえば、 VT100 端末は、キーパッド PF1 キーを押下したとき、 `ESC O P` を送出します。したがって、 Emacs にはこのイベント群のシーケンスを `pf1` なる単一のイベントに変換してもらいたいわけです。 VT100 を使う場合、 `function-key-map` で `ESC O P` を `[pf1]` に「バインド」します。

したがって `C-c PF1` の押下では、文字シーケンス `C-c ESC O P` を送出します。そして関数 `read-key-sequence` は、これを `C-c PF1` に変換し、結果としてベクタ `[?\C-c pf1]` を返します。

`function-key-map` 内のエントリーは、副モード、ローカル、グローバル・キーマップでできたバインドと衝突する場合、無視されます。これは本来、ファンクション・キーの送出する文字シーケンスは、コマンドとのバインドをもってないはずだ、というところにあります。

`function-key-map` の値は、たいてい端末の Terminfo や Termcap 項目に基づいて自動的に設定されますが、ときどき端末固有の Lisp ファイルで補強しなければならないこともあります。 Emacs には多くの共通な端末固有ファイルがついてきます。 Termcap と Terminfo からすぐ出てくるもの以外の項目を `function-key-map` に設定するのが、主な目的です。 34.1.3 節「端末固有の初期化」 p.579 を参照してください。

バージョン 18 以前の Emacs は、ファンクション・キーの表現する文字シーケンスの検出にまったく異なった方法を使っていました。

key-translation-map Variable

この変数は、 `function-key-map` のように入力イベントをほかのイベントに変換するのに使います。これは `function-key-map` と以下の点で異なっています。

- `key-translation-map` は、 `function-key-map` で処理した後に作用します。これは `function-key-map` の変換の結果を受け取ります。
- `key-translation-map` 実際のキー・バインドに優先します。

`key-translation-map` は、通常 `self-insert-command` にバインドしておく普通の文字も含め、ユーザのために一つの文字集合を別のものに写すことにあります。

function-key-map や key-translation-map にキー・シーケンスでなく関数を使うことによって、単なる別名づけにとどまらず、キーの変換を行なうことも可能です。

キー変換関数は、 read-key-sequence に指定されたプロンプトか、あるいはエディタのコマンド・ループで読込み中の場合は nil を、単一の引数としてとります。ほとんどたいていの場合、これは無視できます。

もし関数が自分で入力を読み取るなら、引き続くイベントの変更効果があることになります。例として C-c h で、ひき続く文字をハイパー文字に変える定義を、以下に示します。

```
(defun hyperify (prompt)
  (let ((e (read-event)))
    (vector (if (numberp e)
                (logior (lsh 1 20) e)
              (if (memq 'hyper (event-modifiers e))
                  e
                (add-event-modifier "H-" e))))))

(defun add-event-modifier (string e)
  (let ((symbol (if (symbolp e) e (car e))))
    (setq symbol (intern (concat string
                                 (symbol-name symbol))))
    (if (symbolp e)
        symbol
      (cons symbol (cdr e)))))

(define-key function-key-map "\C-ch" 'hyperify)
```

'iso-transl' ライブラリではこの機能を使って、 ASCII 以外の Latin-1 文字の入力ができるようにしています。

34.8.3 入力の記録

recent-keys Function

この関数は、キーボードとマウスからの入力イベントのうち最後の 100 個を、ベクタにして返します。入力イベントは、キー・シーケンスの一部になってもならなくても、すべてここに含めます。したがって、キーボード・マクロをのぞく最後の 100 個の入力はいつでも取得できます (キーボード・マクロからのイベントをのぞくのは、デバッグのときにそれほど有用でないからです。イベントを調べれば、マクロの起動はすぐにわかります)。

open-dribble-file *filename* Command

この関数は、 *filename* という名前の『ドリブル・ファイル (dribble file)』を開きます。ドリブル・ファイルを開くと、 (キーボード・マクロをのぞく) キーボードとマウスからの全入力イベントをこのファイルに書き込みます。文字以外のイベントは、 '<...>' で囲った印字表現で表わします。

ドリブル・ファイルを閉じるには、 nil を引数にしてこの関数を呼び出します。

この関数は、通常、バグ報告用に Emacs のバグを引き起こすのに必要な入力を記録するときに、使用します。

```
(open-dribble-file "~/dribble")
     ⇒ nil
```

open-termscript 関数も参照してください (34.9 節「端末出力」 p.596 参照)。

34.9 端末出力

端末出力関数は、端末出力を送出したり、端末に送出した出力を追跡したりします。変数 baud-rate は、端末への出力スピードをいくつにするかを Emacs に指定しています。

baud-rate Variable

この変数の値は、 Emacs の知るかぎりでの、端末の出力スピードです。この変数の設定では実際のデータ伝送スピードに変化はありませんが、パディングの計算にはこの値を使います。これはまた、画面の一部分をスクロールするか、あるいは書き換えをするかの決定の際にも使用します (ウィンドウ・システムには真の出力スピードがない、という事実にもかかわらずこう設計したのは、この決定方法の調整手段を提供するためです)。

値の単位はボー (baud) です。

もしネットワーク経由でアクセスしていて、ネットワークの一部でボー・レートが異なる場合、 Emacs の返す値は、ローカルな端末で使用中のボー・レートの値とは異なることがあります。あるネットワーク・プロトコルでは、ローカルな端末速度を遠隔装置と交信して Emacs やほかのプログラムが正しい値を取得できるようになっていますが、そうなっていないプロトコルもあります。もし Emacs の値が間違っていたら、最適でない決定を下してしまいます。この問題を解決するためには、 baud-rate を設定します。

baud-rate Function

この関数は、変数 baud-rate の値を返します。バージョン 18 やそれ以前の Emacs では、これが端末スピードを取得する唯一の方法でした。

send-string-to-terminal *string* Function

この関数は、 *string* を変更せずに端末に送ります。 *string* 中の制御文字は、端末に依存する効果を生じます。

この関数の一つの使いかたとして、ダウンロード可能なファンクション・キー定義機能をもつ端末のファンクション・キーの定義を行なうのに用います。この例では、ある端末で、ファンクション・キー 4 を (コンピュータへ文字 C-u C-f を伝送することによって) 4 文字前に進ませるものとして定義しています。

```
(send-string-to-terminal "\eF4\^U\^F")
     ⇒ nil
```

open-termscript *filename* Command

この関数は、 Emacs が端末に出力する全文字を記録する『端末記録ファイル (termscript file)』を開きます。 nil を返します。端末記録ファイルは、 Emacs が画面を壊したりす

る問題や、本当の Emacs のバグというよりも、間違った Termcap 項目や不適切な端末オプションの設定のせいでおこる問題を調べるときに役に立ちます。どの文字が実際に出力されたのかが確実にわかってしまえば、使用中の Termcap 仕様に対応する文字を確定できます。

34.8 節「端末入力」p.592 の `open-dribble-file` も参照してください。

```
(open-termscript "../junk/termscript")
     ⇒ nil
```

34.10 システム固有の X11 キーシム

システム固有の X11 キーシムを定義するには、変数 `system-key-alist` を設定します。

system-key-alist Variable

この変数の値は、システム固有の各キーシムごとに要素をもつ alist でなければなりません。各要素は (*code* . *symbol*) という形をしており、 *code* は数値のキーシム符号 (ただし「ベンダー固有ビット」の 1 << 28 は含まない) で、 *symbol* はそのファンクション・キーの名前です。

たとえば、`(168 . mute-acute)` は、 (1 << 28) + 168 という値をもつ HP の X サーバーのシステム固有キーを定義しています。

alist でほかの X サーバーのキーシムを定義しても、実際に使用中の X サーバーと衝突さえしなければ、問題ありません。

この変数は、常に現在の X 端末にローカルなので、バッファ・ローカルにすることはできません。 26.2 節「複数のディスプレイ」p.435 を参照してください。

34.11 フロー制御文字

この節では、「どうして Emacs では、フロー制御文字をコマンド文字集合に選んだのか?」という質問に答えようと思います。この件については配布物件の ‘`emacs/INSTALL`’ ファイルにあるフロー制御についてのコメントもあわせてお読みください。 Termcap 項目と DEC 端末コンセントレータに関しては、 ‘`emacs/etc/TERMS`’ をご覧ください。

昔はほとんどの端末でフロー制御が不要だったので、 `C-s` と `C-q` をフロー制御に使うことはありませんでした。したがって `C-s` と `C-q` をコマンド文字として使うのは、議論の余地のないところだったのです。キー・ストロークの経済性と可搬性から、 Emacs は可能なかぎりほとんどすべての ASCII 制御文字におぼえやすい意味を与えました。そのため、 `C-s` が search に、 `C-q` が quote になったのです。

その後、ある端末がこの文字群でフロー制御をし始めました。これは全画面での編集にはむいてなかったので、 Emacs の保守をする人は気にとめませんでした。何年か経って、 `C-s` と `C-q` でのフロー制御は端末でも一般的になりましたが、それでもたいていはオプションでした。フロー制御を使わないユーザの大部分は、フロー制御のせいでしかたなく、それほどおぼえやすくもないキー・バインドに切り替えていました。

で、どちらが「正しい」のでしょう? Emacs でしょうか? 端末やコンセントレータの製造業者でしょうか? この問いへの単純な答えはありません。

なぜ私たちが C-s や C-q で起こる問題に迎合することに気がすすまないのかという一つの理由として、これには正当性がないということがあげられます。文字の流れの透過性を保存するフロー制御の技術は、(まだそれほど共通に実用に供されていないとはいえ) ほかにもあります。この方法によるフロー制御が公式の標準でないことにも、注意してください。興味深いことに、紙テープ・パンチつきのモデル 33 テレタイプ (とても古い) は、パンチのオン・オフにコンピュータから送られた C-s と C-q を使っていました！

X サーバーやそのほかのウィンドウ・システムが、キャラクタ端末を置き換えると、この問題はだんだん解消されそうです。とりあえずの間、 Emacs は、望むならば、フロー制御を有効にする簡単な方法を提供しつづけるでしょう。これには関数 enable-flow-control を呼び出します。

enable-flow-control Function

この関数は、 C-s と C-q による出力フロー制御の使用を有効にし、 keyboard-translate-table で C-\ と C-^ という別名による入力ができるようにします (34.8.2 節「入力イベントの変換」 p.593 参照)。

'.emacs' ファイルの中で関数 enable-flow-control-on を使い、特定の端末型で自動的にフロー制御を有効にすることができます。

enable-flow-control-on &rest *termtypes* Function

この関数は、端末の型が *termtypes* 中の一つであるとき、フロー制御および C-\ と C-^ による別名を有効にします。

```
(enable-flow-control-on "vt200" "vt300" "vt101" "vt131")
```

enable-flow-control が行なうことは以下のとおり。

1. 端末入力を CBREAK モードに設定し、(set-input-mode nil t) でオペレーティング・システムにフロー制御を扱うよう指示します。
2. C-\ と C-^ が C-s と C-q になるよう、 keyboard-translate-table を設定します。最下位の部分をのぞいて、 Emacs は C-s や C-q が押下された文字ではないことは意識しません。そのため C-\ や C-^ がほかのコマンドの文字であっても押下できます。 34.8.2 節「入力イベントの変換」 p.593 を参照してください。

もし端末がフロー制御文字の発信源ならば、いったんカーネルのフロー制御を有効にすると、多分その端末の通常の場合よりもパディングが少なくてすみます。 Termcap 項目をカスタマイズしてパディングの量を減らすことができるでしょう。または baud-rate を小さい値に設定することにより、必要なパディングに (端末のボー・レートが) 遅い速度の場合の計算をさせることもできます。 34.9 節「端末出力」 p.596 を参照してください。

34.12 バッチ・モード

コマンド行オプション '`-batch`' で Emacs を非対話的にすることができます。このモードでは、 Emacs は端末からコマンドを読み込むことはありませんし、端末のモードを変更することもありませんし、消去可能な画面へ出力することを期待しません。これは作動させる Lisp プログラムを指定し、完了後 Emacs が抜けるべきであるということです。作動させるプログラムは、 '`-l` *file*' と '`-f` *function*' で指定します。ここで *file* はロードするライブラリの名前であり、 *function* という引数のない関数を呼び出します。

`message` や `prin1` などでストリームを `t` にして、通常ならエコー領域にいくような出力をする Lisp プログラムは、バッチ・モードでは Emacs の標準エラー記述子で出力を行ないます。したがって Emacs は非対話的な応用プログラムのように振る舞います (コマンドのエコーのように、通常 Emacs 自身が行なうエコー領域出力は完全に抑止されます)。

noninteractive Variable

Emacs がバッチ・モードで走っている間、この変数は、非 `nil` です。

35 Emacs の表示

この章では Emacs のユーザに対する表示機能に関連した機能について説明します。

35.1 画面の再表示

関数 `redraw-frame` は指定されたフレーム全体の内容を再表示します。第 26 章「フレーム」p.435 を参照してください。

redraw-frame *frame* Function

この関数はフレーム *frame* の内容を消去し再表示します。

さらに強力なのは `redraw-display` です。

redraw-display Command

この関数は表示されているフレームすべての内容を消去し再表示します。

ユーザからの入力の処理は再表示よりも絶対に優先されます。入力がある場合にこれらの関数を呼んでも、すぐには何もしませんが、すべての入力を処理した後で、全体の再表示がいつか必ず起こります。

通常 Emacs をサスペンドしたりリジュームしたりしたときも画面が再表示されます。端末エミュレータによっては Emacs のような画面指向のプログラムのために、普通の逐次的な表示とは別の表示内容を保持するものがありますが、このような端末を使っている時には再開時の再表示を禁止したいと思うかもしれません。

no-redraw-on-reenter Variable

この変数は Emacs がサスペンドされてそれからリジュームされた時に画面全体を再表示すべきかどうかを制御します。非 `nil` の場合は再表示を行ない、 `nil` の場合は再表示を行ないません。

35.2 画面サイズ

画面サイズ関数は端末の高さ・幅にアクセスしたり、設定したりします。複数のフレームを使っている場合には、選択されているフレームに対して働きます (第 26 章「フレーム」p.435 参照)。

screen-height Function

この関数は画面上で表示に使用可能な行数を返します。

```
(screen-height)
     ⇒ 50
```

screen-width Function

この関数は画面上で表示に使用可能な桁数を返します。

```
(screen-width)
     ⇒ 80
```

set-screen-height *lines* &optional *not-actual-size* Function

この関数は端末が *lines* 行を表示可能であることを宣言します。既存のウィンドウのサイズはこれに合うように同じ比率で変更されます。

not-actual-size が `nil` でない場合、 Emacs は *lines* 行を出力しますが、実際の画面の高さの値を変更しません。 (カーソルを正しく位置付けるためには実際のサイズが正しいことが必要なのですが)。端末が実際に持っている高さよりも小さい値を使うと、小さい画面で何が起こるかを再現するためとか、画面全体を使うと端末の動作がおかしくなる場合に便利でしょう。

lines が以前の値と違う場合には、画面全体が消去されてから新しいサイズで再表示されます。

この関数は `nil` を返します.

set-screen-width *columns* &optional *not-actual-size* Function

この関数は端末が *columns* 桁を表示可能であることを宣言します。詳細は `set-screen-height` を参照。

35.3 切捨て表示

テキストの行がウィンドウの右端を超えた場合には、その行は画面の次の行に続きが表示されるか、右端を切捨てて一行に表示されるか、の選択肢があります。長いテキスト行を表示するために使われる行のことを『継続 (continuation)』行といいます。普通、右端の桁にある '`$`' は行の切捨てを示します。また右端の '`\`' は行が次の行に "wrap"、つまり続いていることを示しています。 (表示テーブルを使えばほかの表示をすることもできます。 35.14 節「表示テーブル」p.618 を参照。)

継続は行詰め込み (filling) とは違うことに注意してください。継続は画面の上だけで起こり、バッファの内容に対しては起こらないし、また継続はちょうど画面の左端で行を分割するのであって、単語の境界ではありません。 29.11 節「行詰め」p.495 を参照してください。

truncate-lines User Option

このバッファローカル変数はウィンドウの右端にはみだす行を Emacs がどう表示するかを制御します。既定値は `nil` で、継続表示を指定します。その値が `nil` でない場合は、はみだす行は切り捨て表示されます。

変数 `truncate-partial-width-windows` が `nil` でない場合は、 `truncate-lines` の値に関わらず (ひとつのフレームの中で) 横に並んだウィンドウではいつも切捨て表示が採用されます。

default-truncate-lines User Option

この変数は `truncate-lines` に対してローカル値がないバッファに対して使われる既定値です。

truncate-partial-width-windows User Option

この変数は横に並んだウィンドウで、ウィンドウの右端を越えて伸びる行の表示の仕方を制御します (25.2 節「ウィンドウの分割」p.408 参照)。非 `nil` の場合は、そのような行は切捨て表示されます。それ以外は `truncate-lines` の指示に従います。

表示テーブルを使えば行の継続や行の切捨てを示す表示を変えることができます。 35.14 節「表示テーブル」p.618 を参照してください。

バッファに非常に長い行があるときに継続表示をさせると、それに対する計算のために Emacs の再表示が遅くなることがあります。また桁位置の計算や字下げ関数も遅くなります。この場合には `cache-long-line-scans` を `t` に設定するとよいかもしれません。

cache-long-line-scans Variable

この変数を非 `nil` にすると、字下げやカーソル移動関数、 Emacs の再表示がバッファの走査の結果をキャッシュして、バッファが変更されない限りそれを参照することによって再走査を避けるようになります。

このキャッシュを使うようにすると短い行の処理はいくぶん遅くなります。

この変数は各バッファで自動的にローカルになります。

35.4 エコー領域

『エコー領域 (echo area)』は `message` プリミティブによって作られるメッセージを表示するために、またキーストロークをエコーするために使われます。このエコー領域と画面の同じ場所にミニバッファが (アクティブな時には) 表示されてはいますが、ミニバッファと同じものではありません。 *GNU Emacs Manual* にエコー領域とミニバッファが画面スペースの使用で衝突した場合にそれをどう解決するかのルールが規定されています (*The GNU Emacs Manual* の “The Minibuffer” 節参照)。エラー・メッセージはエコー領域に表示されます。 9.5.3 節「エラー」p.119 を参照してください。

エコー領域に出力するには Lisp の表示関数 (printing functions) でストリームを `t` にすればできますが、次の方法でもできます。

message *string* &rest *arguments* Function

この関数は 1 行のメッセージをエコー領域に表示します。引数の *string* は C 言語の `printf` の制御文字列と同様です。その変換の仕様については 4.5 節「文字と文字列間の変換」p.60 の `format` を参照してください。 `message` はそれによって作られた文字列を返します。

バッチ・モードでは `message` はメッセージのテキストを標準エラー・ストリームに表示して改行します。

string を `nil` にすると `message` はエコー領域の表示を消去します。そのときミニバッファがアクティブであれば、これと同時にミニバッファの内容が画面上に見えるようになります。

```
(message "Minibuffer depth is %d."
         (minibuffer-depth))
  ⊣ Minibuffer depth is 0.
⇒ "Minibuffer depth is 0."

---------- Echo Area ----------
Minibuffer depth is 0.
---------- Echo Area ----------
```

エコー領域に表示されたメッセージのほとんどは、'*Messages*' バッファにも記録されています。

message-log-max User Option

この変数は '*Messages*' バッファに何行まで保持するかを指定します。値が t の場合は、制限無しになります。 nil の場合はメッセージの記録 (logging) はまったく行われません。次の例はメッセージを表示するだけで記録されないようにする方法です。

```
(let (message-log-max)
  (message ...))
```

echo-keystrokes Variable

この変数はコマンド文字をエコーするまでどのぐらいの時間をあけるかを決めます。値は整数で、エコーするまでに待つ秒数を指定します。ユーザがプレフィックス・キー (C-x など) をタイプしたあと、この変数の値の秒数のあいだ次のキーをタイプせずにいると、そのプレフィックス・キーがエコー領域にエコーされます。それに続く同じコマンドの中の文字もエコーされます。

値が 0 の場合には、コマンド入力はエコーされません。

cursor-in-echo-area Variable

この変数はメッセージがエコー領域に表示された時に、カーソルがどこに現れるかを制御します。これが非 nil の場合は、カーソルはメッセージの最後に現れます。それ以外は、カーソルはポイントに現れ、エコー領域にはまったく現れません。

値は通常 nil で、 Lisp プログラムが短時間だけ t に束縛します。

35.5 不可視テキスト (invisible text)

invisible 属性によって文字を『不可視 (invisible)』にして、画面上に現れないようにすることができます。これはテキストの属性でもよいし、オーバレイの属性でもかまいません。

一番簡単な場合として、 invisible 属性が非 nil なら文字が不可視になります。これはデフォルトの場合で、変数 buffer-invisibility-spec の既定値を変えなければ invisibility 属性 (訳注: invisible 属性の誤りか) はこのように働きます。この機能は選択的表示 (selective display) (35.6 節「選択表示」 p.605 参照) によく似ていますが、より一般的ですっきりしています。

より一般的には、変数 buffer-invisibility-spec を使って invisible 属性のどの値がテキストを非表示にするのかを制御できます。これによってテキストをあらかじめサブセットに分類

してそれぞれに異なった `invisible` 値を与えておき、あとで `buffer-invisibility-spec` の値を変えることでそのいろいろなサブセットの可視、不可視を切替えるということができます。

`buffer-invisibility-spec` による可視性の制御は、特にデータベースのエントリの一覧を表示するプログラムで便利でしょう。これを使うとデータベースのエントリの一部だけを見せるという方法で、便利なフィルタリングコマンドを実装することができます。この変数を設定する方法は非常に高速であり、バッファのテキストをすべて走査して変更すべき属性を探すのに比べてずっと速くなります。

buffer-invisibility-spec Variable

この変数はどの `invisible` 属性が実際に文字を不可視にするかを指定します。

`t` `invisible` 属性が非 `nil` ならば文字が不可視になります。これが既定値です。

list リストの各要素が文字を不可視にします。つまりこのリストの要素の中のあるものが文字に適用されるものならば、その文字は不可視になるということです。このリストの要素としては次の 2 種類のものが許されます。

atom `invisible` 属性値が *atom* であるか、 *atom* をメンバに持つリストである場合に、文字が不可視になります。

`(`*atom* `. t)` `invisible` 属性値が *atom* であるか、 *atom* をメンバに持つリストである場合に、文字が不可視になります。さらに、この文字が行末にあって、直後に可視の改行が続いているならば、省略記号が表示されます。

通常、テキストに働くコマンドやポイントを動かすコマンドはテキストが不可視かどうかには関知しません。しかし、ユーザレベルの行移動コマンドは不可視な改行を無いものとして扱います。

35.6 選択表示

『選択表示 (selective display)』とは画面上で行を隠す機能で、 2 種類あります。

第 1 の方法は明示的な選択表示で、 Lisp プログラムの中で使うために設計されています。プログラムがテキストを変更することでどの行を隠すかを制御します。 Outline モードは伝統的にこの方法を使っていました。この方法は一部不可視テキスト機能 (35.5 節「不可視テキスト (invisible text)」 p.604 参照) で置き換えられており、これを使った新しい Outline モードがあります。

第 2 の方法は隠す行の選択が字下げの仕方に基づいて自動的に行われるものです。この方法はユーザレベルの機能として設計されています。

明示的な選択表示を制御する方法は改行 (control-j) をキャリッジ・リターン (control-m) に置き換えるというものです。するとそれまでその改行に続く行であったものが見えなくなります。厳密には行を分けるのは改行だけなので、一時的にですがもはや行があるとは言えず、前の行の一部になっています。

選択表示は編集コマンドには直接影響を与えません。例えば C-f (forward-char) は何のためらいもなくポイントを見えないテキストの中に移動します。しかし、改行文字をキャリッジ・リターン文字で置き換えたことに影響されるコマンドがあります。例えば next-line は見えない行を飛び越えます。これはこのコマンドが改行文字だけを探すからです。選択表示を使うモードが、改行を考慮するコマンドや、テキストの一部を見えるようにしたり、見えなくしたりするコマンドを定義することもあるでしょう。

選択表示されているバッファをファイルに書き出す時には、すべての control-m は改行として出力されます。つまり次にファイルを読み込んだ時には、普通通りに見え、隠されたテキストは全くありません。選択表示は Emacs の中だけで効果があるのです。

selective-display Variable

このバッファローカル変数は選択表示を作動させます。つまり行や行の一部が見えなくされる可能性があるということです。

- selective-display の値が t ならば、 control-m の後ろの行は表示されません。
- selective-display の値が正の整数なら、その桁数よりも深く字下げされた行は表示されません。

バッファの一部が見えなくなっている場合には、垂直方向の移動コマンドはその部分が存在しないかのように動作し、 next-line コマンドは 1 回で見えない行をいくつも飛び越えることができます。しかし文字単位の移動コマンド (forward-char など) は見えない部分を飛び越えないので、見えない部分にテキストを挿入・削除することが (わかりにくくはありますが) 可能です。

次の例に、バッファ foo の表示の様子が selective-display の値によってどのように変化するかを示します。バッファの内容は変化しません。

```
(setq selective-display nil)
     ⇒ nil

---------- Buffer: foo ----------
1 on this column
 2on this column
  3n this column
  3n this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

```
(setq selective-display 2)
     ⇒ 2

---------- Buffer: foo ----------
1 on this column
 2on this column
 2on this column
1 on this column
---------- Buffer: foo ----------
```

selective-display-ellipses Variable

このバッファローカル変数が非 `nil` ならば、 Emacs は見えないテキストが続いている行の最後に '...' を表示します。次の例は前の例の続きです。

```
(setq selective-display-ellipses t)
     ⇒ t

---------- Buffer: foo ----------
1 on this column
 2on this column ...
 2on this column
1 on this column
---------- Buffer: foo ----------
```

表示テーブルを使ってこの省略記号 ('...') をほかのテキストに変えることもできます。35.14 節「表示テーブル」 p.618 を参照してください。

35.7 オーバレイ矢印

『オーバレイ矢印 (overlay arrow)』は、バッファ中のある行にユーザに編集させるというより読ませる目的でユーザの注意を向けたい時に便利です。例えば、デバッガのインタフェースとして使われるモードでは、オーバレイ矢印がこれから実行されようとしている行を示します。

overlay-arrow-string Variable

この変数はある行に注意を惹くために表示される文字列を保持します。矢印機能が使われていない場合は `nil` です。

overlay-arrow-position Variable

この変数はオーバレイ矢印をどこに表示するかを示すマーカを保持します。このマーカは行頭を指していなければなりません。その行の頭に矢印テキストが表示され、本来ならそこに表示されているはずのテキストは下に隠されます。もっとも、一般に矢印は短く、行は字下げで始まるので、普通は重要なものが隠されてしまうことはありません。

オーバレイ文字列はマーカが指しているバッファでだけ表示されます。したがってある時点にオーバレイ矢印を持ちうるバッファは一つだけです。

これと同じことは `before-string` 属性を持つオーバレイを作ることでもできます。35.9.1 節「オーバレイの属性」 p.610 を参照してください。

35.8 一時表示

一時表示 (temporary displays) は、コマンドがユーザに編集させるのではなく読ませる目的でバッファに出力するために使われます。ヘルプ・コマンドがよくこの機能を使います。

with-output-to-temp-buffer *buffer-name forms...* Special Form

この関数は *forms* をその出力がすべて *buffer-name* で名前を指定されるバッファに入れられるようにして実行します。その後そのバッファはいずれかのウィンドウに表示しますが、選択はされません。

文字列 *buffer-name* は一時バッファを指定しますが、これは既存のものである必要はありません。この引数はバッファでなく文字列でなければなりません。そのバッファはまず内容が消され (確認なし)、 `with-output-to-temp-buffer` が終了したあとには、変更なしと印がつけられます。

`with-output-to-temp-buffer` は `standard-output` を一時バッファに束縛して、*forms* を評価しています。この *forms* 中の、 Lisp 出力関数を使った出力はとくに指定しなければこのバッファに向けられます (画面表示やエコー領域のメッセージも一般的には「出力」と言えますが、この関数には影響されません)。 16.5 節「出力関数」 p.233 を参照してください。

forms の最後のフォームの値が返されます。

```
---------- Buffer: foo ----------
 This is the contents of foo.
---------- Buffer: foo ----------

(with-output-to-temp-buffer "foo"
    (print 20)
    (print standard-output))
⇒ #<buffer foo>

---------- Buffer: foo ----------
20

#<buffer foo>

---------- Buffer: foo ----------
```

temp-buffer-show-function Variable

この変数が非 `nil` であれば、 `with-output-to-temp-buffer` がそれを関数として呼び出し、ヘルプ・バッファの表示をさせます。その関数は引数を一つ受けますが、これは関数が表示を行うべきバッファです。

バージョン 18 以前の Emacs ではこの変数は `temp-buffer-show-hook` という名前でした。

momentary-string-display *string position* &optional *char message* Function

この関数は現在のバッファの *position* に *string* を一時的に表示します。 undo リストの状態やバッファの変更ステータスには影響を与えません。

一時的な表示は次の入力イベントまで表示されています。次の入力イベントが *char* ならば、 `momentary-string-display` はそれを無視して戻りますが、それ以外のイベントはバッファされたままになり、その後入力として使われます。つまり *char* をタイプすると文字列の表示が消えるだけですが、 (たとえば)`C-f` をタイプすると文字列の表示が消えたあと、 (多分) ポイントが前方に移動します。引数 *char* の既定値はスペースです。

`momentary-string-display` の返り値には特に意味はありません。

文字列 *string* が制御文字を含んでいなければ、 `before-string` 属性をもつオーバレイを作ることで、同じことをより一般的な方法ですることができます。

message が非 `nil` ならば、 *string* がバッファに表示されるのと同時に、それがエコー領域に表示されます。これが `nil` のときは、既定値の「作業を続けるには *char* を押してください」という意味のメッセージが表示されます。

次の例では、ポイントは最初二行目の行頭にあるとします。

```
---------- Buffer: foo ----------
This is the contents of foo.
⋆Second line.
---------- Buffer: foo ----------

(momentary-string-display
  "**** Important Message! ****"
  (point) ?\r
  "Type RET when done reading")
⇒ t

---------- Buffer: foo ----------
This is the contents of foo.
**** Important Message! ****Second line.
---------- Buffer: foo ----------

---------- Echo Area ----------
Type RET when done reading
---------- Echo Area ----------
```

35.9 オーバレイ

『オーバレイ (overlays)』を使って、表現形としてバッファのテキストの見え方を変えることができます。オーバレイは特定のバッファに属するオブジェクトで、決まった始めと終りの点を持っています。また、ユーザが設定したり、読みとったりできる属性もあり、オーバレイ中のテキストの表示方法に影響します。

35.9.1 オーバレイの属性

オーバレイの属性はある面ではテキスト属性に似ていますが、似ている点よりも違っている点の方が重要です。テキスト属性はテキストの一部とみなされますが、オーバレイは特にテキストの一部とはみなされません。ですから、テキストをバッファ間でコピーしてもテキスト属性は保存されますが、オーバレイを保存するようなことはしていません。バッファのテキスト属性を変更すると、バッファは変更されたとマークされますが、オーバレイを移動したりその属性値を変更したりしても変更されたとマークはされません。またテキスト属性の変更と違って、オーバレイの変更はバッファの undo リストにも記録されません。

`priority`
: この属性の値 (非負の数値でなければなりません) は、そのオーバレイの優先度を決めます。優先度が問題になるのは二つ以上のオーバレイが同じ文字の上に重なっており、しかも両方が表示するフェースを指定している場合です。この場合、`priority` の値が大きいものが優先され、そのフェース属性が優先度の低いオーバレイのフェース属性に優先して使われます。

  現在は、すべてのオーバレイはテキスト属性よりも高い優先度を持ちます。負の優先度についてはその意味をまだ決めていないので、使わないようにしてください。

`window`
: `window` 属性が非 `nil` ならば、オーバレイはそのウィンドウだけに適用されます。

`category`
: オーバレイに `category` 属性があれば、それをオーバレイの『カテゴリ (category)』といいます。これはシンボルでなければなりません。そのシンボルの属性がオーバレイの属性の既定値として働きます。

`face`
: この属性はテキストのフォントと色を制御するものです。値はフェースの名前か、フェースの名前のリストです。詳細は 35.10 節「フェース」p.613 を参照してください。この機能は一時的なものになるかもしれません。将来テキストの表示方法を指定するためのほかの方法で置き換える可能性があります。

`mouse-face`
: この属性はマウスがオーバレイの範囲内にあるとき、`face` の代わりに使われます。この機能も `face` 同様一時的なものである可能性があります。

`modification-hooks`
: この属性の値は、オーバレイ内の文字が変更されたり、その厳密な意味での内部に文字が挿入されたときに呼び出される関数のリストです。

  フック関数はこの変化の前後両方に呼び出されます。その関数が受けとった情報を保存して、呼び出しの間の情報を比較すれば、バッファのテキストにどういう変化が起こったのかを正確に知ることができます。

  変化の前に呼ばれるときには、各関数は四つの引数を受けとります。これらはオーバレイ、`nil`、変更されようとしているテキスト範囲の始点と終点、です。

  変化の後に呼ばれるときには、各関数は五つの引数を受けとります。すなわちオーバレイ、`t`、今変更されたテキスト範囲の始点と終点、その範囲で置き換えられたテキストの変更前の長さ、です。(挿入に対しては変更前の長さは 0、削除に対

してはその削除された文字の数で変更後の範囲の始点と終点が同じになっています)。

`insert-in-front-hooks`
: この属性の値は、オーバレイの始点のすぐ右側にテキストが挿入される直前と直後に呼び出されるべき関数のリストです。関数の呼び出し方は `modification-hooks` の関数と同じです。

`insert-behind-hooks`
: この属性の値はオーバレイの終点のすぐ右側にテキストが挿入される直前と直後に呼び出されるべき関数のリストです。関数の呼び出し方は `modification-hooks` の関数と同じです。

`invisible`
: `invisible` 属性はオーバレイ中のテキストを不可視にすることができます。不可視というのはつまり画面には表示されないということです。詳しくは 35.5 節「不可視テキスト (invisible text)」p.604 を参照してください。

`intangible`
: `intangible` 属性をオーバレイに付けると、ちょうど `intangible` テキスト属性と同じように働きます。詳しくは 29.18.4 節「特殊な意味をもつ属性」p.513 を参照してください。

`before-string`
: この属性の値はオーバレイの始点に加えて表示する文字列です。その文字列はどんな意味でもバッファの中には存在せず画面上にあるだけです。文字列には 1 カラムで表示される文字以外は含めない方がいいでしょう。タブや改行などの制御文字を含めるとおかしな結果になります。

`after-string`
: この属性の値は、オーバレイの終点に加えて表示する文字列です。その文字列はどんな意味でもバッファの中には存在せず画面上にあるだけです。文字列には 1 カラムで表示される文字以外は含めない方がいいでしょう。タブや改行などの制御文字を含めるとおかしな結果になります。

`evaporate`
: この属性が非 `nil` ならば、オーバレイは空になったとき (つまりまったく文字を覆わなくなったとき) に自動的に削除されます。

以下はオーバレイの属性を読み書きするための関数です。

overlay-get *overlay prop* — Function

この関数は *overlay* に記録されている属性 *prop* があれば、その値を返します。 *overlay* がその属性には値を記録していない場合でも、 `category` 属性があってそれがシンボルならば、そのシンボルの *prop* 属性が使われます。それもなければ値は `nil` です。

overlay-put *overlay prop value* — Function

この関数は *overlay* に記録されている属性 *prop* の値を *value* に設定します。 *value* を返します。

`get-char-property` というある文字に対してオーバレイ属性とテキスト属性の両方を調べる関数もありますので参照してください。 29.18.1 節「テキスト属性の調査」 p.510 を参照してください。

35.9.2 オーバレイの管理

この節ではオーバレイの生成、削除、移動、内容検査のための関数を説明します。

make-overlay *start end* &optional *buffer* Function

この関数は *buffer* に属し、 *start* から *end* の範囲のオーバレイを生成して返します。 *start* も *end* もバッファ中の位置を指定するもの、つまり整数か、マーカーでなければなりません。 *buffer* を省略すると、オーバレイは現在のバッファに作られます。

overlay-start *overlay* Function

この関数は *overlay* が始まる位置を返します。

overlay-end *overlay* Function

この関数は *overlay* が終る位置を返します。

overlay-buffer *overlay* Function

この関数は *overlay* が属するバッファを返します。

delete-overlay *overlay* Function

この関数は *overlay* を削除します。オーバレイは Lisp オブジェクトとしては存在し続けますが、それが属していたバッファの一部ではなくなり、表示にはなんの影響も与えなくなります。

move-overlay *overlay start end* &optional *buffer* Function

この関数は *overlay* を *buffer* の範囲 *start* から *end* に移動します。 *start* も *end* もバッファ中の位置を指定するもの、つまり整数か、マーカーでなければなりません。 *buffer* を省略すると、オーバレイのバッファは変わりません。

返り値は *overlay* です.

オーバレイの終点を変更する正当な手段はこの関数だけです。オーバレイの中のマーカを直接変更してしまうと、ほかの重要なデータ構造の更新が行われず失われるオーバレイが出ることになりかねないので、やめてください。

overlays-at *pos* Function

この関数は位置 *pos* を含むオーバレイすべてのリストを返します。リスト中の順番は特に規定されていません。 *pos* を含むオーバレイとは、 *pos* かそれよりも前に始まり、 *pos* よりも後ろで終るオーバレイです。

next-overlay-change *pos* Function

この関数はオーバレイの始点・終点の中で *pos* よりも後ろにある次のものを返します。

previous-overlay-change *pos* Function

この関数はオーバレイの始点・終点の中で *pos* よりも前にある次のものを返します。

35.10 フェース

『フェース (face)』とはグラフィカル属性の集まりに名前をつけたものです。グラフィカル属性にはフォント、文字色、背景色、下線を引くかどうか、があります。フェースは画面上のテキストの表示方法を制御します。

フェースにはそれぞれの『フェース id 番号 (face id number)』があり、 Emacs の内部ではそれによってフェースを互いに区別しています。しかしほとんどの目的には Lisp プログラム中でフェースを名前で参照することができます。

facep *object* Function

この関数は *object* がフェースの名前シンボルであるとき (または内部でフェースの情報を記憶するのに使われているベクタの一種であるとき) に `t` を返します。そうでない場合は `nil` を返します。

フェース名はすべてのフレームで意味を持ち、特に変更しなければすべてのフレームで同じ意味を持ちます。望むならあるフェース名が特定のフレームで別個の意味をもつようにすることも可能です。

35.10.1 標準フェース

次の表は標準のフェースすべてとその使用法のリストです。

`default` このフェースは普通のテキストに使われます。

`modeline` このフェースはモード行とメニュー・バーに使われます。

`region` このフェースは Transient Mark モードでリージョンを強調表示するときに使われます。

`secondary-selection`
このフェースはユーザの選択した第 2 セレクションを示すために使われます。

`highlight` このフェースはいろいろな強調表示に使うことを意図したものです。

`underline` これはテキストに下線を引くフェースです。

`bold` このフェースは使用可能ならボールド (太字体) フォントを使います。フレームのフォントのボールド版があればそれを使います。使いたいのであればユーザが自分でボールド版のあるフォントを既定フォントにしなければなりません。

`italic` このフェースはフレームのフォントのイタリック (斜体) 版があればそれを使います。

`bold-italic`
このフェースはフレームのフォントのボールド・イタリック (太斜体) 版があればそれを使います。

35.10.2 表示の際のフェースの合併

ここで表示にどのフェースを使うかを指定する方法をすべて示します。

- 既定値による指定。各フレームには id 番号が 0 である『既定フェース (default face)』があり、とくにほかのフェースを指定しないテキストに使われます。
- テキスト属性による指定。文字には `face` 属性をつけることができ、その文字はそのフェースで表示されます。 29.18.4 節「特殊な意味をもつ属性」 p.513 を参照してください。

  文字に `mouse-face` 属性がある場合は、マウスがその文字の「十分近くに」あるときに `face` 属性の代わりに使われます。
- オーバレイによる指定。オーバレイにも `face` 属性と `mouse-face` 属性が付けられて、オーバレイが覆っているテキストすべてに働きます。
- アクティブなリージョンによる指定。 Transient Mark モードでは、リージョンは特定のフェースを使って強調表示されます。 (後述する `region-face` を参照)。
- 特別なグリフによる指定。各グリフは特定のフェース id 番号を指定することができます。 35.14.3 節「グリフ」 p.620 を参照してください。

もしこれらの諸指定から文字に対して複数のフェースが指定される場合は、 Emacs はそれら指定されたフェースの属性を合併 (merge) します。まず特別なグリフのフェースの属性が 1 番で、次にリージョンの強調表示 (それが適用可能な場合)、その次にオーバレイからのフェースの属性、さらにテキスト属性からのもの、そして最後が既定値フェースです。

複数のオーバレイがある一つの文字を覆っている場合には、オーバレイの中で優先度がもっとも高いものがそれより低いものに優先して使われます。 35.9 節「オーバレイ」 p.609 を参照してください。

フォントや色などの属性が上記のどれによっても指定されていない場合は、フレーム自身のフォントと色が使われます。

35.10.3 フェースに働く関数

フェースが指定できる属性にはフォント、文字色、背景色、下線有無があります。これらに `nil` を設定することにより特に指定しないままにしておくこともできます。

以下フェースを生成・変更するためのプリミティブを示します。

make-face *name* Function

この関数は *name* という名前の新しいフェースを定義します。初期値としてすべての属性は `nil` になっています。 *name* という名前のフェースがすでにある場合にはなにもしません。

face-list Function

この関数は定義されているフェース名全部のリストを返します。

copy-face *old-face new-name* &optional *frame new-frame* Function

この関数は *new-name* というフェースを、 *old-face* という既存のフェイスのコピーとして定義します。 *new-name* というフェースがまだなければ生成します。

オプション引数 *frame* が与えられると、この関数はそのフレームだけに働きます。与えない場合は、各フレームに対して働き、各フレームの *old-face* から同じフレームの *new-face* に属性をコピーします。

オプション引数 *new-frame* が与えられると、`copy-face` は *frame* の *old-face* から *new-frame* の *new-name* に属性をコピーします。

以下に示す関数によって既存のフェースの属性を変更することができます。 *frame* を指定すると、そのフレームだけに働きます。指定しなければすべてのフレームに働くとともに、新規のフレームに適用される既定値も変更されます。

set-face-foreground *face color* &optional *frame* Function
set-face-background *face color* &optional *frame* Function

これらの関数はフェース *face* の文字色 (背景色) を *color* に設定します。引数 *color* は色の名前を表す文字列でなければなりません。

白黒画面では網かけパターン (stipple pattern) によって何段階かの濃淡表示が実装されています。

set-face-stipple *face pattern* &optional *frame* Function

この関数はフェース *face* の背景網かけパターンを *pattern* に設定します。引数 *pattern* は X サーバによって定義されている網かけパターンの名前でなければなりません。あるいは `nil` でもよく、網かけを使わないという意味になります。

濃淡表示をするときに自動的に網かけパターンが使われるので、普通網かけパターンを気にする必要はありません。

set-face-font *face font* &optional *frame* Function

この関数はフェース *face* のフォントを設定します引数 *font* は文字列でなければなりません。

set-face-underline-p *face underline-p* &optional *frame* Function

この関数はフェース *face* の下線属性を設定します。非 `nil` は下線付きという意味で `nil` は下線なしです。

invert-face *face* &optional *frame* Function

フェース *face* の文字色と背景色を交換します。フェースが文字色・背景色両方を指定していない場合は、その文字色・背景色にはそれぞれ既定値の背景色・文字色が設定されます。

以下の関数はフェースの属性を調べるものです。 *frame* を指定しない場合は、新規のフレームのための既定値を参照します。

face-foreground *face* &optional *frame* Function
face-background *face* &optional *frame* Function

これらの関数はフェース *face* の文字色 (背景色) を文字列で返します。

face-stipple *face* &optional *frame* Function

この関数はフェース *face* の背景網かけパターンの名前を返します。背景網かけパターンがなければ `nil` を返します。

face-font *face* &optional *frame* Function

この関数はフェース *face* のフォント名を返します。

face-underline-p *face* &optional *frame* Function

この関数はフェース *face* の下線属性を返します。

face-id *face* Function

この関数はフェース *face* のフェース id 番号を返します。

face-equal *face1 face2* &optional *frame* Function

これはフェース *face1* と *face2* が同じ表示属性を持っていれば `t` を返します。

face-differs-from-default-p *face* &optional *frame* Function

これは *face* が既定値のフェースと異なった表示をするフェースならば `t` を返します。あるフェースが既定値のフェースと同じと判断されるのは、属性値それぞれが既定値のフェースのものとと同じ値であるか、`nil`(つまり既定値を継承している) 場合です。

region-face Variable

この変数の値はリージョンがアクティブになった場合にその中の文字を表示するために使うフェース id を指定します (Transient Mark モードのみ)。これで指定されたフェースはそのリージョンのテキストに対するテキスト属性やオーバレイに由来する一切のフェースに優先されます。 Transient Mark モードの詳細については 28.6 節「マーク」 p.475 を参照してください。

通常これの値は `region` という名前のフェースの id 番号になっています。

35.11 対応括弧の点滅表示

この節ではユーザが閉じ括弧を挿入したときに Emacs が対応する開き括弧を示すメカニズムについて説明します。

blink-paren-function Variable

この変数の値は (無引数の) 関数です。この関数は閉じ括弧構文をもつ文字が挿入されたときに呼び出されます。 `blink-paren-function` の値は `nil` でもよく、この場合なにも行われません。

> 注意: この変数は Emacs の古いバージョンでは `blink-paren-hook` という名前でしたが、フックの標準的な呼び出し方にしたがって呼び出されるものではないので、バージョン 19 では `blink-paren-function` という名前に変更されました。

blink-matching-paren Variable

この変数が `nil` のときは `blink-matching-open` は何もしません。

blink-matching-paren-distance Variable

この変数は対応する括弧をスキャンする際に、これを越えたらあきらめるという距離の最大値を指定します。

blink-matching-paren-delay Variable

この変数は対応する括弧にカーソルが置かれる時間を秒単位で指定します。 1 秒未満がいいようですが、既定値はすべてのシステムで使える 1 秒になっています。

blink-matching-open Function

この関数は `blink-paren-function` の既定値です。閉じ括弧構文をもつ文字のあとにポイントがあることを前提にしています。その文字がまだ画面に表示されていなければ、その文字のコンテクストをエコー領域に表示します。処理に長い時間かかってしまうのを防ぐために、この関数は `blink-matching-paren-distance` 文字を越えては検索しません。

次の例はこの関数を明示的に呼び出すものです。

```
(defun interactive-blink-matching-open ()
  "Indicate momentarily the start of sexp before point."
  (interactive)
  (let ((blink-matching-paren-distance
         (buffer-size))
        (blink-matching-paren t))
    (blink-matching-open)))
```

35.12 反転表示 (Inverse Video)

inverse-video User Option

この変数は Emacs が画面のテキストすべてを反転表示するかどうかを制御します。非 `nil` の場合は、反転表示をし、 `nil` の場合は反転表示をしません。既定値は `nil` です。

mode-line-inverse-video User Option

この変数はモード行に反転表示を使うかどうかを制御します。これが非 `nil` ならモード行は反転表示されます。それ以外はモード行はテキスト同様に普通に表示されます。既定値は `t` です。

X ウィンドウのフレームではモード行の表示に `modeline` というフェースが使われます。これは変更しなければ、通常普通のフェースの反転になっています。

35.13 普通の表示方法

普通の表示方法はそれぞれの文字をどう表示するかを定義しています。表示テーブルを設定してこの表示方法を変えることができます。 (35.14 節「表示テーブル」p.618 参照)。以下に普通の表示方法を示します。

- 文字コード 32 から 126 まではグリフ・コード 32 から 126 に対応します。通常これはそれ自身が表示されることを意味しています。

- 文字コード 9 は水平タブです。 `tab-width` で決まる位置までの空白として表示されます。
- 文字コード 10 は改行です。
- これら以外のコードつまり 0 から 31 までとコード 127 は、 `ctl-arrow` の値によって 2 種類に表示されます。これが非 `nil` ならば、これらのコードは二つのグリフの列に対応し、 1 番目が ‘^’ の ASCII コードとなります。 (表示テーブルによって ‘^’ 以外のものを使うことが指定されているかもしれません。) それ以外はこれらのコードは 128 から 255 のコードと同じように対応づけられます。
- 文字コード 128 から 255 は四つのグリフの列に対応します。 1 番目のグリフは ‘\’ の ASCII コードで、残りは文字コードを 8 進表示したものです。 (表示テーブルに ‘\’ 以外のものを使うことが指定されているかもしれません)。

表示テーブルがある時でも、そのアクティブな表示テーブル中のエントリが `nil` であるような文字に対しては普通の表示方法が適用されますから、表示テーブルを設定する時には、普通でない動作をさせたい文字だけを指定すればいいのです。

これらの変数はある文字が画面に表示される方法に影響するものですが、文字が占める桁数も変えるので、字下げ関数にも影響します。

ctl-arrow User Option

このバッファローカル変数は制御文字の表示方法を制御します。非 `nil` ならば、キャレットのあとに文字をつけて ‘`^A`’ のように表示します。 `nil` ならばバックスラッシュのあとに 8 進 3 桁で ‘`\001`’ のように表示します。

default-ctl-arrow Variable

この変数の値は `ctl-arrow` の値がバッファで再定義されていない場合の既定値です。 10.9.3 節「バッファローカルな変数のデフォルト値」 p.144 を参照してください。

tab-width User Option

この変数の値はタブ・ストップの間隔で、 Emacs バッファでタブ文字が表示される時に使われます。既定値は 8 です。この機能は `tab-to-tab-stop` コマンドで使われるユーザが設定できるタブとは何の関係もないことに注意してください。 29.16.5 節「調節可能な「タブ・ストップ」」 p.507 を参照してください。

35.14 表示テーブル

『表示テーブル (display tale)』機能を使うと 256 種類の文字すべてについて、それが画面でどう表示されるかを制御できます。この機能は ASCII 文字セットにない文字を使うヨーロッパの言語を表示するために便利です。

表示テーブルは各文字コードを『グリフ (glyphs)』の列に対応づけます。グリフとは画面上で 1 文字分を占める像 (image) です。さらに各グリフが端末でどう表示されるかも『グリフ・テーブル (glyph table)』を使って制御できます。

35.14.1 表示テーブルの形式

表示テーブルは実は 262 要素の配列です。

make-display-table Function

表示テーブルを作って返します。テーブルは初期値としてすべての要素が `nil` になっています。

最初の 256 要素は文字コードに対応します。つまり n 番目の要素は文字コード n をどう表示するかを指定しています。その値は `nil` か、グリフ値のベクタ (35.14.3 節「グリフ」p.620 参照) でなければなりません。要素が `nil` のときは、普通の表示法にしたがって表示しろ、という意味です (35.13 節「普通の表示方法」p.617 参照)。

表示テーブルを使って改行文字の表示方法を変えると、バッファ全体が一つの長い "行" として表示されてしまいます。

表示テーブルの残りの 6 要素は特別な目的に使われており、これが `nil` の場合は、次に示す既定値を使うという意味になります。

256 切捨て表示された行の末尾に表示されるグリフ (これの既定値は '`$`')。 35.14.3 節「グリフ」p.620 を参照してください。

257 継続表示された行の末尾用のグリフ (既定値は '`\`')。

258 文字が 8 進コード表示されているということを示すグリフ (既定値は '`\`')。

259 制御文字を表すグリフ (既定値は '`^`')。

260 見えない行があることを示すグリフのベクタ (既定値は '`...`')。 35.6 節「選択表示」p.605 を参照してください。

261 横に並んだウィンドウの間に描かれる境界線に使われるグリフ (既定値は '`|`')。 25.2 節「ウィンドウの分割」p.408 を参照してください。

次の例は `ctl-arrow` を非 `nil` に設定した場合の効果を真似る表示テーブルを作る方法を示しています:

```
(setq disptab (make-display-table))
(let ((i 0))
  (while (< i 32)
    (or (= i ?\t) (= i ?\n)
        (aset disptab i (vector ?^ (+ i 64))))
    (setq i (1+ i)))
  (aset disptab 127 (vector ?^ ??)))
```

35.14.2 アクティブな表示テーブル

ウィンドウごと、バッファごとに表示テーブルを指定することができます。バッファ b がウィンドウ w に表示されていたとすると、表示部はウィンドウ w の表示テーブルがあればそれを使います。それがなければ、今度はバッファ b の表示テーブルがあればそれを使います。これも

なければ標準の表示テーブルがあればそれを使います。ここで選択された表示テーブルは『アクティブな (active)』表示テーブルと呼ばれます。

window-display-table *window* Function

この関数は *window* の表示テーブルを返します。 *window* に割り当てられた表示テーブルがなければ `nil` を返します。

set-window-display-table *window table* Function

この関数は *window* の表示テーブルを *table* に設定します。引数の *table* は表示テーブルか、 `nil` でなければなりません。

buffer-display-table Variable

この変数はすべてのバッファで自動的にローカルになります。それぞれのバッファでの値はそのバッファのための表示テーブルです。そのバッファに割り当てられた表示テーブルが特にない場合は値は `nil` です。

standard-display-table Variable

この変数の値は既定値の表示テーブルで、ウィンドウにもバッファにも表示テーブルがない場合に使われます。この変数の既定値は `nil` です。

あるウィンドウに使うべき表示テーブルがない場合 — つまりウィンドウにもなく、バッファにもなく、 `standard-display-table` にもない場合 — は、 Emacs はそのウィンドウの中のすべての文字コードに対して普通の表示方法 (usual display conventions) を使います。 35.13 節「普通の表示方法」p.617 を参照してください。

35.14.3 グリフ

『グリフ (glyph)』とは文字を一般化したもので、画面上の一つの文字位置を占める像 (image) を表します。グリフは文字同様 Lisp では整数によって表現されます。

それぞれの整数のグリフとしての意味は変数 `glyph-table` の値であるグリフ・テーブルで定義されます。

glyph-table Variable

この変数の値は現在のグリフ・テーブルです。これはベクタでなければなりません。これの g 番めの要素がグリフ・コード g を定義します。値がベクタでなく `nil` である場合は、すべてのグリフは simple であるとされます (下記参照)。

以下にグリフ・テーブルの要素として可能なものを示します:

string このグリフを表示する場合は *string* 中の文字を端末に送ります。この方法は文字端末だけで使用可能で、 X では使えません。

integer このグリフをコード *integer* の別名と定義します。別名を使ってグリフのフェース・コードを指定できます。下記参照。

`nil` グリフは simple です。一般的な端末ではグリフ・コードを 256 で割った余りが出力する文字になります。 X ではグリフ・コードを 256 で割った余りが出力する文字で、グリフ・コードを 256 で割ったものがその文字を出力するのに使われる『フェース id 番号 (face id number)』を指定します。 35.10 節「フェース」p.613 を参照してください。

グリフ・コードがグリフ・テーブルの長さと同じか、それより大きい場合は、コードは自動的に simple になります。

35.14.4 ISO Latin 1

ISO Latin 1 をすべて扱える端末があるなら、次のようにすればその文字集合を使うことができます。

```
(require 'disp-table)
;; 文字コード 160 ~ 255 がそれ自身として表示されるように設定する。
;; (コード 128 ~ 159 は制御文字。)
(standard-display-8bit 160 255)
```

もし ISO Latin 1 文字集合で書かれたバッファを編集する時に端末が ASCII 文字集合以外を扱えないならば、'`iso-ascii`' をロードすれば、ほかの ISO 文字を ASCII 文字の列でそれらしく表示するように表示テーブルを設定することができます。たとえば「ウムラウト付きの o」ならば '`{"o}`') として表示します。

西欧諸国の端末には ISO Latin 1 はサポートせずにその国の言語独自の文字をサポートしているものがあります。そのような端末を使ってその一言語で作業するための表示テーブルを定義することもできます。たとえば '`lisp/iso-swed.el`' はスウェーデンの端末用です。

端末の型に対応してその端末用の Lisp ファイルを書いておけば、あなたの端末のための適切な表示テーブルを自動的にロードすることもできます。

35.15 ベルを鳴らす

Emacs にベル音を鳴らさせて (または画面を点滅させて)、ユーザの注意を引くことができます。ただしやりすぎは禁物です。あまり頻繁にベルがなるといらいらしかねません。またエラーを通知すべきときにベルを鳴らすだけ、ということのないようによく考えてください。 (9.5.3 節「エラー」p.119 を参照してください。)

ding &optional *dont-terminate* Function

この関数はベル音を鳴らすか、画面を点滅させるかします (下の `visible-bell` を参照)。同時に *dont-terminate* が非 `nil` であればそのとき実行中のキーボード・マクロを終了させます。

beep &optional *dont-terminate* Function

これは `ding` と同じものです。

visible-bell Variable

この変数はベルが鳴ったことを表すために Emacs が画面を点滅させるかどうかを決めます。 `nil` でない場合には点滅させ、 `nil` の場合には点滅させません。これが有効なのは、 X ウィンドウ下か、現在使っている端末の Termcap エントリに、ビジブル・ベル機能 (‘`vb`’) が定義されている場合だけです。

35.16 ウィンドウ・システム

Emacs はいくつかのウィンドウ・システムのもとで動作します。中でも有名なのは X ウィンドウ・システムでしょう。 Emacs も X も「ウィンドウ」という用語を使いますが、意味が違います。 Emacs のフレームが X で言えば一つのウィンドウです。 Emacs のウィンドウ一つ一つに対しては X はまったく関知しません。

window-system Variable

この変数は Lisp プログラムから、 Emacs がどのウィンドウ・システムの下で動作しているかを知るための変数です。その値は `x`(Emacs が X の下で動作している場合) のようなシンボルか、 `nil`(Emacs が普通の端末で動作している場合) です。

window-setup-hook Variable

この変数は正規フックであり、 Emacs があなたの ‘`.emacs`’ と (もしあれば) デフォルト初期化ファイルをロードして、さらに端末それぞれの Lisp コードをロードし、フック `term-setup-hook` を実行したそのあとに実行されます。

このフックは内部処理で使用するもので、ウィンドウ・システムとの通信を確立し、最初のウィンドウを作成します。ユーザはそれに干渉すべきではありません。

36 カレンダとダイアリのカスタマイズ

カレンダとダイアリを自分の好みに合わせるカスタマイズにはいろいろあります。

36.1 カレンダのカスタマイズ

変数 `view-diary-entries-initially` を `t` に設定すると、カレンダが呼び出されたとき、自動的に今日の日付のダイアリ・エントリも表示します。ダイアリの日付は、今日の日付が見える場合のみ表示します。以下の 2 行を '`.emacs`' ファイルに加えると、

```
(setq view-diary-entries-initially t)
(calendar)
```

カレンダとダイアリが Emacs の起動時に表示されます。

同様に、変数 `view-calendar-holidays-initially` を `t` にすると、カレンダに入ったとき、自動的に当該 3ヵ月の休日表を表示します。休日の一覧は、別のウインドウに表示します。

変数 `mark-diary-entries-in-calendar` を `t` にすると、ダイアリに登録してある全ての日に印をつけることができます。これは、カレンダ・ウインドウの内容を再計算するときに作用します。これらの日に印をつける方法は二つあります。ディスプレイのサポートがある場合にはフェース (35.10 節「フェース」 p.613 参照) を変える方法、さもなくば、日付の横にプラス・サイン ('+') を置く方法があります。

同様に、変数 `mark-holidays-in-calendar` を `t` に設定すると休日にフェースを変えるか、または、アスタリスク ('*') をつけます。

変数 `calendar-holiday-marker` は、休日となる日付の印のつけ方を指定します。この値は、休日となる日付の隣に挿入する文字か、その日付を表示するために使うフェース名になります。同様に、変数 `diary-entry-marker` もダイアリに登録されている日付に対する印のつけ方を指定します。カレンダは、この目的のために `holiday-face` と `diary-face` を作成します。Emacs が端末で複数のフェースをサポートしている場合、これらのシンボルは、これらの変数のデフォルト値になっています。

変数 `calendar-load-hook` は、カレンダ・パッケージが最初にロードされたときに (実際にカレンダの表示をはじめようとする前に) 実行する正規フックです。

カレンダが起動するとき、正規フック `initial-calendar-window-hook` を実行します。カレンダ表示の再計算時には、このフックは実行しません。しかし、`q` コマンドでカレンダを抜けてから再入した場合、このフックを再度実行します。

変数 `today-visible-calendar-hook` は、今日の日付がウィンドウに見えるときに、表示するカレンダをカレンダ・バッファに準備した後に実行する正規フックです。このフックの一つの使い方は、今日の日付をアスタリスクに置き換えることです。そうするには、フック関数 `calendar-star-date` を使います。

```
(add-hook 'today-visible-calendar-hook 'calendar-star-date)
```

ほかの標準フック関数は今日の日付に、そのフェースを変えるか、アスタリスクを追加するかして、印をつけます。以下に例を示します:

```
(add-hook 'today-visible-calendar-hook 'calendar-mark-today)
```

変数 `calendar-today-marker` は、今日を示す印をつける方法を指定します。この値は、今日の日付の隣に挿入する文字か、日付を表示するために使用するフェースの名前になります。`calendar-today-face` という名前のフェースがこの目的のために用意されています。使用端末上で Emacs が複数フェースをサポートしている場合、このシンボルが、この変数のデフォルト値になります。

似たような正規フック `today-invisible-calendar-hook` は今日の日付がウインドウで見えない時に実行します。

36.2 休日のカスタマイズ

Emacs では、いくつかのリストを用いてあらかじめ休日が定義されています。必要に応じてこれらの休日リストに休日を追加、削除してカスタマイズできます。 Emacs が使う休日リストは、一般的な休日 (`general-holidays`)、ローカルな休日 (`local-holidays`)、キリスト教の休日 (`christian-holidays`)、ヘブライ (ユダヤ) 暦の休日 (`hebrew-holidays`)、イスラム教 (モスレム) 暦の休日 (`islamic-holidays`) と、そのほかの休日 (`other-holidays`) です。

デフォルトでは、一般的な休日にはアメリカ合州国共通の休日が定義されています。これらの休日の設定を取り除くには、 `general-holidays` を `nil` にします。

デフォルトではローカルな休日は設定されていません (しかし、サイトで設定しているかも知れません)。以下に述べるように、変数 `local-holidays` にいろいろな休日リストを設定することができます。

デフォルトでは、 Emacs は知っている全ての宗教上の休日を含まずに、世間一般のカレンダの中にある休日のみを含んでいます。これ以上の宗教上の休日のコレクションについては、変数 `all-christian-calendar-holidays`, `all-hebrew-calendar-holidays`, `all-islamic-calendar-holidays` のどれか (または全て) を `t` に設定してください。宗教上の休日を取り除きたいならば、対応する変数 `christian-holidays`, `hebrew-holidays`, `islamic-holidays` のいずれかまたは全てを `nil` に設定してください。

変数 `other-holidays` には、どのような休日のリストも設定することができます。このリストは、通常空で、個人使用を意図しています。

リスト (`general-holidays`, `local-holidays`, `christian-holidays`, `hebrew-holidays`, `islamic-holidays`, `other-holidays`) のそれぞれは、『休日形式 (holiday forms)』のリストです。各休日形式は、休日 (時には休日リスト) を表します。

以下が、休日形式としてとることができる種類の一覧です。日付と月の数は 1 から数えますが、「曜日名」を表す数は日曜日を 0 と数えます。要素 *string* はいつでも文字列での休日名です。

`(holiday-fixed` *month day string*`)`
: グレゴリオ暦で固定の日付。

(`holiday-float` *month dayname k string*)
: グレゴリオ暦 *month* 月の *k* 番目の *dayname* 曜日 (日曜を *dayname*0 とし、以下同様)。 *k* が負なら、月の終りから逆に数えることを意味します。

(`holiday-hebrew` *month day string*)
: ヘブライ暦で固定の日付。

(`holiday-islamic` *month day string*)
: イスラム教暦で固定の日付。

(`holiday-julian` *month day string*)
: ユリウス暦で固定の日付。

(`holiday-sexp` *sexp string*)
: Lisp 式 *sexp* で計算する日付。この式は計算のために変数 `year` を使用すべきです。そして、休日の日付を返すか、今年は休日にならないならば、 `nil` を返さなければなりません。 *sexp* の値は、休日の日付を形式 (*month day year*) というリストとして表現しなければなりません。

(`if` *condition holiday-form*)
: *condition* が真のときだけ、なる休日です。

(*function* [*args*])
: 引数 *args* で呼び出した関数 *function* が計算した日付のリストです。

たとえば、 7 月 14 日のフランス革命記念日を追加したいとします。以下のようにすればできます:

```
(setq other-holidays '((holiday-fixed 7 14 "Bastille Day")))
```

休日形式 (`holiday-fixed 7 14 "Bastille Day"`) が、 7 番目の月 (7 月) の 14 番目の日を指定します。

多くの休日は、特定の月の特定の週の曜日になります。ここに、 Virgin Islands で 8 月の第 4 月曜日に祝うとされる Hurricane Supplication Day を表す休日形式を挙げます:

```
(holiday-float 8 1 4 "Hurricane Supplication Day")
```

ここで 8 月を 8 で指定し、月曜を 1 で指定し (日曜は 0 火曜は 2 以下同様)、その月の 4 週目を 4 で指定しています (1 は最初の週を、 2 は 2 週目を、 -1 は最後の週を、 -2 は最後から 2 番目の週を、以下同様)。

ヘブライ暦やイスラム教暦、ユリウス暦などの固定の日付になる休日も指定することができます。たとえば、

```
(setq other-holidays
      '((holiday-hebrew 10 2 "Last day of Hanukkah")
        (holiday-islamic 3 12 "Mohammed's Birthday")
        (holiday-julian 4 2 "Jefferson's Birthday")))
```

は、ハヌカー最終日 (ヘブライ暦の月は Nisan を 1 として数えるため)、イスラム教暦のモハメッド誕生際 (イスラム教暦の月は Muharram を 1 として数えるため)、ユリウス暦で 1743 年 4 月 2 日である Thomas Jefferson の誕生日を追加します。

条件つき休日を含めるには、 Emacs Lisp の if か holiday-sexp 形式を使います。たとえば、アメリカ大統領選挙は 4 で割り切れる年の 11 月の最初の月曜日の次の火曜日に行なわれます:

```
(holiday-sexp (if (= 0 (% year 4))
                  (calendar-gregorian-from-absolute
                   (1+ (calendar-dayname-on-or-before
                         1 (+ 6 (calendar-absolute-from-gregorian
                                 (list 11 1 year))))))
              "US Presidential Election"))
```

かもしくは、

```
(if (= 0 (% displayed-year 4))
    (fixed 11
           (extract-calendar-day
             (calendar-gregorian-from-absolute
               (1+ (calendar-dayname-on-or-before
                     1 (+ 6 (calendar-absolute-from-gregorian
                              (list 11 1 displayed-year)))))))
           "US Presidential Election"))
```

いくつかの休日は、これらの形式のいずれにも収まりません。なぜならそれらの判別には特別な計算が絡んでいるからです。このような場合、計算するための Lisp 関数を書かねばなりません。たとえば、食を含めるには、 other-holidays に (eclipses) を追加して、 Emacs Lisp 関数 eclipses を書かなければなりません。その関数は、次のようなカレンダ・ウィンドウで見える範囲内の適切なグレゴリオ暦の日付とその説明文字列のリスト (空かもしれない) を返さなければなりません:

```
(((6 27 1991) "Lunar Eclipse") ((7 11 1991) "Solar Eclipse") ... )
```

36.3 日付表示フォーマット

calendar-date-display-form を設定することで、モード行、ダイアリやメッセージでの日付の表示の仕方をカスタマイズすることができます。この変数は、変数 month, day, year(これら全て文字列形式の数)、変数 monthname, dayname(どちらもアルファベット文字列) を含むことができる式のリストを保持しています。アメリカ・スタイルでの、このリストのデフォルト値は以下のとおりです:

```
((if dayname (concat dayname ", ")) monthname " " day ", " year)
```

一方、ヨーロッパ・スタイルでのこの値のデフォルトは以下のとおりです:

```
((if dayname (concat dayname ", ")) day " " monthname " " year)
```

ISO 標準日付 (ISO standard date) 表現は以下のとおりです:

```
(year "-" month "-" day)
```

以下にアメリカ・フォーマットを示します。

```
(month "/" day "/" (substring year -2))
```

36.4 時刻表示フォーマット

デフォルトではカレンダとダイアリは、1 ~ 12 までの時、分、'am' か 'pm' のいずれか、というお決まりのアメリカ・スタイルで時刻を表示します。もし、ヨーロッパ・スタイル、またはアメリカ陸軍スタイルとしても知られている、時を 00 ~ 23 で示す方を好む場合、変数 `calendar-time-display-form` を変えることができます。この変数は、変数 `12-hours`、`24-hours`、`minutes` (全て文字列形式の数) と、`am-pm`、`time-zone`(どちらもアルファベット文字列) を含むことができる式のリストです。`calendar-time-display-form` のデフォルト値は以下のようになります:

```
(12-hours ":" minutes am-pm
          (if time-zone " (") time-zone (if time-zone ")"))
```

ヨーロッパ・スタイルの時刻を与える値です:

```
(24-hours ":" minutes
          (if time-zone " (") time-zone (if time-zone ")"))
```

36.5 夏時間

Emacs は標準時と夏時間 (日の出日の入り、夏至冬至、春分秋分、月の満ち欠けを勘定に入れることで与えられる) の違いを理解します。夏時間の規則は、地域ごとに、および、年ごとに異なっています。正確に求めるために、Emacs はどのルールを使うのかを知る必要があります。

いくつかのオペレーティング・システムは、自分のいる地域に応じた規則に追随しています。これらシステムでは、Emacs は必要な情報をシステムから自動的に取得します。もし、いくつかあるいは全てのこれらの情報が見つからない場合、Emacs は GNU 世界の中心であるケンブリッジやマサツーセッツで使われている規則を用いて不足を埋めます。

デフォルト規則の選択が自分の地域では不適当な場合、変数 `calendar-daylight-savings-starts` と `calendar-daylight-savings-ends` を設定することによって、Emacs に使うべき規則を教えることができます。これらの値は、変数 `year` を参照して、評価結果がグレゴリオ暦の夏時間の始まる日と終る日をリスト (*month day year*) の形式で与える、Lisp 式でなければなりません。自分の地域で夏時間が使われていないときは、この値を `nil` にしなければなりません。

Emacs は、休日と同様に、夏時間の始まる日、終る日を判別するために、それから時刻を太陽暦、太陰暦に合わせるために、これらの式を使います。

ケンブリッジ、マサチューセッツでの値は、次のとおりです:

```
(calendar-nth-named-day 1 0 4 year)
(calendar-nth-named-day -1 0 10 year)
```

即ち、`year` で指定する年の第 4 月 (4 月) の第 1 第 0 曜日 (日曜日)、そして、第 10 月 (10 月) の最後の日曜日です。夏時間が 10 月 1 日から始まるように変わったとすると、`calendar-daylight-savings-starts` を以下のようにします:

```
(list 10 1 year)
```

さらに複雑な例として、ヘブライ暦 Nisan の 1 日から夏時間が始まると仮定します。`calendar-daylight-savings-starts` を以下の値に設定しなければなりません。

```
(calendar-gregorian-from-absolute
  (calendar-absolute-from-hebrew
    (list 1 1 (+ year 3760))))
```

これは、 Nisan がヘブライ暦で最初の月であり、 Nisan において、ヘブライ年はグレゴリオ年と 3760 の差があるためです。

自分の地域に夏時間がない場合、もしくは、いつも標準時が欲しいときは、 calendar-daylight-savings-starts と、 calendar-daylight-savings-ends を nil にします。

変数 calendar-daylight-time-offset は、分単位で夏時間と標準時の差を指定します。ケンブリッジでの値は 60 です。

変数 calendar-daylight-savings-starts-time と変数 calendar-daylight-savings-ends-time は、夏時間へおよび夏時間から移行するとき、現地時間で深夜以後経過すべき分数を指定します。ケンブリッジでは、この値は両方とも 120 です。

36.6 ダイアリのカスタマイズ

たいてい、ダイアリ・バッファ・ウィンドウのモード行は、ダイアリ・エントリの日付にあたるなんらかの休日を示しています。休日をチェックする処理は数秒を要するので、休日に関する情報を含める場合、ダイアリ・バッファの表示は著しく遅くなります。もし休日の情報がなくてもダイアリ・バッファの表示が速い方がいい場合、変数 holidays-in-diary-buffer を nil に設定します。

変数 number-of-diary-entries は、一度に表示するダイアリ・エントリの日数を制御します。これは view-diary-entries-initially が t のときに M-x diary コマンド同様に最初の表示に影響します。たとえば、デフォルト値が 1 のとき、今日の日付のダイアリ・エントリのみ表示することを示しています。この値が 2 ならば、今日と明日のエントリの両方が表示されます。値は、七つの要素からなるベクタもとることができます。たとえば、値が [0 2 2 2 2 4 1] のとき日曜日にはダイアリ・エントリは表示されず、月曜日から木曜日までは今日と明日のダイアリ・エントリが表示され、金曜日には金曜日から月曜日のエントリが表示され、そして土曜日には今日のエントリのみが表示されます。

変数 print-diary-entries-hook は、現在ダイアリ・バッファで見えるダイアリ・エントリだけを含む、一時バッファの準備の後に、実行する正規フックです (そのほかの、関係ないダイアリ・エントリは一時バッファには全然存在しません。ダイアリ・バッファでは、それらはほとんど隠されています)。このフックのデフォルト値は、コマンド lpr-buffer による印刷です。印刷するために別のコマンドを使いたい場合、このフックの値を変えるだけです。ほかの使用法としてたとえば、日時の順に、行を再配置するなどがあります。

ダイアリ・ファイルでの日付の形式は、標準アメリカ・スタイルでもヨーロッパ・スタイルでもないものが必要なとき、変数 diary-date-forms を設定することでカスタマイズできます。この変数は、日付を識別するためのパターンのリストです。それぞれの日付パターンは、全要素が正規表現 (30.2 節「正規表現」 p.525 参照) かシンボル month, day, year, monthname, dayname であるリストです。これらの要素全てはダイアリ・ファイル中のある種のテキストと照

合するために使うパターンです。日付パターンが一致するためには、全体としてその要素全てが連続して一致しなければなりません。

日付パターンの正規表現照合の普通の手法では、'*' が語構成可能であるように変更した標準構文テーブルを使います。

シンボル `month`, `day`, `year`, `monthname`, `dayname` は、判断中の日付の月数、日数、年数、月名、曜日名に一致します。数に一致するシンボルでは、先頭に 0 がついても大丈夫です。名前に一致するものは、3 字の省略形にしても、頭文字を大文字にしても大丈夫です。全てのシンボルは '*' と一致します。なぜなら、ダイアリ・エントリでの '*' は「任意の日」、「任意の月」、などを意味するので、判断中の日付に無関係に一致しなければならないからです。

アメリカ・スタイルでの `diary-date-forms` のデフォルト値は以下のようになります:

```
((month "/" day "[^/0-9]")
 (month "/" day "/" year "[^0-9]")
 (monthname " *" day "[^,0-9]")
 (monthname " *" day ", *" year "[^0-9]")
 (dayname "\\W"))
```

リスト中の日付パターンは相互排他的かつ、ダイアリ・エントリ自身には一致してはならず、日付と空白文字一つだけに一致しなければなりません。相互排他的であるために、パターンが (日付の終りの空白を超える) ダイアリ・エントリ・テキストの一部分に一致しなければならないのならば、日付パターンの第 1 要素は `backup` でなければなりません。これによって、日付の認識において一致を完了した後、そのダイアリ・エントリの現在の語頭に戻ることができます。`backup` を使ったとしても、その日付パターンは絶対にそのダイアリ・エントリの第 1 語の一部分以上に一致してはなりません。ヨーロッパ・スタイルでの `diary-date-forms` のデフォルト値は、このリストになります:

```
((day "/" month "[^/0-9]")
 (day "/" month "/" year "[^0-9]")
 (backup day " *" monthname "\\W+\\<[^*0-9]")
 (day " *" monthname " *" year "[^0-9]")
 (dayname "\\W"))
```

3 番目のパターン中で `backup` を使っていることに注意してください。ここでは、4 番目のパターンとの区別のために、日付自身を超えて語の一部と一致しなければならなかったのです。

36.7 ヘブライ暦やイスラム教暦での日付に基づくダイアリ・エントリ

自分のダイアリ・ファイルで、世界標準のグレゴリオ暦に基づくエントリ同様に、ヘブライ暦やイスラム教暦の日付に基づくエントリを置くことができます。しかし、そういったエントリの認識は時間を消費しますし、ほとんどの人は使わないので、明示的にそれを使用可能にしなければなりません。ダイアリにヘブライ暦の日付のダイアリ・エントリを認識させたいのなら、たとえば、このようにしなければなりません:

```
(add-hook 'nongregorian-diary-listing-hook 'list-hebrew-diary-entries)
(add-hook 'nongregorian-diary-marking-hook 'mark-hebrew-diary-entries)
```

イスラム教暦の日付のエントリを使いたいなら、こうします:

```
(add-hook 'nongregorian-diary-listing-hook 'list-islamic-diary-entries)
(add-hook 'nongregorian-diary-marking-hook 'mark-islamic-diary-entries)
```

ヘブライ暦やイスラム教暦の日付のダイアリ・エントリはグレゴリオ暦の日付のダイアリ・エントリと同じフォーマットを持ちますが、ヘブライ暦では 'H' が、イスラム教暦では 'I' が前置される点だけが違います。さらに、ヘブライ暦やイスラム教暦の月名は最初の 3 字では一意に指定できないため、省略してはなりません。たとえばヘブライ暦の日付 Heshvan 25 のダイアリ・エントリは、こんな感じになります:

```
HHeshvan 25 Happy Hebrew birthday!
```

するとこのエントリはヘブライ暦での Heshvan 25 に対応する全ての日付のダイアリに現れます。そして、これが Dhu al-Qada 25 に一致するイスラム教暦の日付のダイアリ・エントリです:

```
IDhu al-Qada 25 Happy Islamic birthday!
```

グレゴリオ暦の日付のダイアリ・エントリについては、ヘブライ暦やイスラム教暦の日付のエントリにアンパサンド ('&') が前置されていたら、印をつけません。

これがカレンダで使用する、ヘブライ暦とイスラム教暦で類似する、選択された日付やほかの日付に一致するダイアリ・エントリを作成するためのコマンドの表です:

`i h d` 選択された日付に対応するヘブライ暦の日付で、ダイアリ・エントリを追加します (`insert-hebrew-diary-entry`)。

`i h m` 選択された日付に対応するヘブライ暦の月の同じ日で、ダイアリ・エントリを追加します (`insert-monthly-hebrew-diary-entry`)。このダイアリ・エントリは選択された日付とヘブライ暦で同じ日になる任意の日付に一致します。

`i h y` 選択された日付に対応するヘブライ暦の年の同じ月日で、ダイアリ・エントリを追加します (`insert-yearly-hebrew-diary-entry`)。このダイアリ・エントリは選択された日付とヘブライ暦で同じ月日になる任意の日付に一致します。

`i i d` 選択された日付に対応するイスラム教暦の日付で、ダイアリ・エントリを追加します (`insert-islamic-diary-entry`)。

`i i m` 選択された日付に対応するイスラム教暦の月の同じ日で、ダイアリ・エントリを追加します (`insert-monthly-islamic-diary-entry`)。

`i i y` 選択された日付に対応するイスラム教暦の年の同じ月日で、ダイアリ・エントリを追加します (`insert-yearly-hebrew-diary-entry`)。

これらコマンドは普通のダイアリ・エントリの対応するコマンドと同じように動作します。実行すると、カレンダ・ウィンドウでポイントがある日付に対して、自分のダイアリ・ファイルの末尾にダイアリ・エントリの一部分としての日付部分だけを挿入します。それに続くダイアリ・エントリの残りの部分は、自分で挿入しなければなりません。

36.8 ダイアリ表示の飾りつけ

ダイアリは、ダイアリ・バッファを準備してからフック `diary-display-hook` を実行して、表示されます。このフック (`simple-diary-display`) のデフォルト値は、関係ないダイアリ・

エントリを隠してダイアリ・バッファを表示します。けれども、自分で次のようにフックを指定すると、

```
(add-hook 'diary-display-hook 'fancy-diary-display)
```

これによって、飾りつきダイアリ表示ができます。これは、表示のためだけに存在する特別なバッファにダイアリ・エントリと休日をコピーして、表示します。別のバッファにコピーすることで、表示されるテキストをきれいにするための変更を加えることができます (たとえば、エントリをそれが適用される日付でソートするなど)。

単純なダイアリ表示では、 print-diary-entries でバッファのハード・コピーを印刷できます。 1 週間分の日ごとのダイアリのハード・コピーを印刷するには、ポイントをその週の日曜に置いて、 7 d とタイプし、それから M-x print-diary-entries を実行します。いつものことですが、休日を含めると表示が若干遅くなります。変数 holidays-in-diary-buffer を nil に設定すれば、速度を稼ぐことができます。

普通は飾りつきダイアリ・バッファはダイアリ・エントリのない日は、それが休日であっても、表示しません。飾りつきダイアリ・バッファでそういった日を表示したいならば、変数 diary-list-include-blanks を t に設定してください。

飾りつきダイアリ表示をするのならば、正規フック list-diary-entries-hook を使って、各日のダイアリ・エントリをその時刻でソートできます。このようにします

```
(add-hook 'list-diary-entries-hook 'sort-diary-entries t)
```

各日について、これで認識可能な日時で始まるダイアリ・エントリが、その時刻でソートされます。日時のないダイアリ・エントリは、各日の中で最初に来ます。

飾りつきダイアリ表示はインクルードされたダイアリ・ファイルを処理することもできます。これによって、複数の人が関係するイベントについてのダイアリ・ファイルを彼らの間で共有することができます。ダイアリ・ファイル中のこの形式の行は:

#include "*filename*"

飾りつきダイアリ・バッファに、ファイル *filename* からダイアリ・エントリをインクルードします。インクルード機構は再帰的なので、インクルード・ファイルがさらにほかのファイルをインクルードし、というようにできます。もちろん、インクルードが循環しないように気をつけなければなりません。インクルード機能を可能にするにはこうします:

```
(add-hook 'list-diary-entries-hook 'include-other-diary-files)
(add-hook 'mark-diary-entries-hook 'mark-included-diary-files)
```

インクルード機構は、飾りつきダイアリ表示に限って働きます。理由は、普通のダイアリ表示はダイアリ・ファイルのエントリを直接表示するからです。

36.9 S 式のエントリと飾りつきダイアリ表示

S 式ダイアリ・エントリは、単にそのダイアリ・エントリに複雑な条件をつけるだけではありません。飾りつきダイアリ表示を使う場合は、 S 式エントリでそのエントリの日付自体に依存するテキストを生成できます。たとえば、記念日のダイアリ・エントリは記念日からの年数をダイアリ・エントリのテキストに挿入できます。そのため、このダイアリ・エントリに '%d' があります:

```
%%(diary-anniversary 10 31 1948) Arthur's birthday (%d years old)
```

これが年齢に置き換えられるので、飾りつきダイアリ・バッファに現れる 1990 年 10 月 31 日のエントリは、こうなります:

```
Arthur's birthday (42 years old)
```

もし、ダイアリ・ファイルが代わりにこのエントリを含んでいたなら:

```
%%(diary-anniversary 10 31 1948) Arthur's %d%s birthday
```

1990 年 10 月 31 日の飾りつきダイアリ・バッファのエントリは、こうなります:

```
Arthur's 42nd birthday
```

同様に、循環ダイアリ・エントリは繰り返しの回数を挿入できます:

```
%%(diary-cyclic 50 1 1 1990) Renew medication (%d%s time)
```

は、こんな感じになります:

```
Renew medication (5th time)
```

1990 年 9 月 8 日の飾りつきダイアリ表示になります。

S 式ダイアリ・エントリの一般性により、アルゴリズム的に記述可能な任意のダイアリ・エントリの指定ができます。 S 式ダイアリ・エントリは、そのエントリが任意の与えられた日付に適用されるか否かを計算する式を含みます。その値が非 `nil` ならば、エントリはその日付に適用されます。そうでなければ、適用されません。式は変数 `date` を判断中の日付を知るために使うことができます。この値は、グレゴリオ暦のリスト (*month day year*) です。

たとえば、毎月 21 日が平日ならばその日に給料を貰い、 21 日が週末ならその前の金曜に貰えると仮定します。こういった日付に一致する S 式ダイアリ・エントリは、このように書きます:

```
&%%(let ((dayname (calendar-day-of-week date))
         (day (car (cdr date))))
      (or (and (= day 21) (memq dayname '(1 2 3 4 5)))
          (and (memq day '(19 20)) (= dayname 5)))
         ) Pay check deposited
```

以下の S 式ダイアリ・エントリで (飾りつきダイアリ表示において)、テキストが日付によって異なるダイアリ・エントリを調合することができます。

`%%(diary-sunrise-sunset)`
: 今日の日の出と日の入りの、現地時間のダイアリ・エントリを作ります。

`%%(diary-phases-of-moon)`
: 月の相 (1/4) のダイアリ・エントリを作ります。

`%%(diary-day-of-year)`
: 今日が 1 年の何日目かとことと、今年の残り日数つきのダイアリ・エントリを作ります。

`%%(diary-iso-date)`
: 今日の ISO 商業日付 (ISO commercial date) つきのダイアリ・エントリを作ります。

`%%(diary-julian-date)`
: 今日のユリウス暦日付つきのダイアリ・エントリを作ります。

`%%(diary-astro-day-number)`
: 今日の天文 (ユリウス) 日付のダイアリ・エントリを作ります。

`%%(diary-hebrew-date)`
: 今日のヘブライ暦日付つきのダイアリ・エントリを作ります。

`%%(diary-islamic-date)`
: 今日のイスラム教暦日付付のダイアリ・エントリを作ります。

`%%(diary-french-date)`
: 今日のフランス革命暦日付付のダイアリ・エントリを作ります。

`%%(diary-mayan-date)`
: 今日のマヤ暦日付付のダイアリ・エントリを作ります。

よって、ダイアリ・エントリ

```
&%%(diary-hebrew-date)
```

を含めると、飾りつきダイアリ表示を使ったときに、毎日のダイアリ表示がヘブライ暦の日付も含むようになります (単純なダイアリ表示では、行 '`&%%(diary-hebrew-date)`' が任意の日付のダイアリに現れますが、特に便利なことはしません)。

これら関数は、ヘブライ暦に基づく S 式ダイアリ・エントリを構築するためにある程度標準的に使うことができます。

`%%(diary-rosh-hodesh)`
: ヘブライ暦の新しい月が始まるとき、その通知と儀式の案内をするダイアリ・エントリを作ります。

`%%(diary-parasha)`
: 毎週のユダヤの集会で読む教典を教える土曜日のダイアリ・エントリを作ります。

`%%(diary-sabbath-candles)`
: 安息日のキャンドルを灯す現地時間を教える金曜日のダイアリ・エントリを作ります。

`%%(diary-omer)`
: 適切なときに、 omer の回数を与えるダイアリ・エントリを作ります。

`%%(diary-yahrzeit` *month day year*`)` *name*
: 命日を記念日として印づけるダイアリ・エントリを作ります。日付はグレゴリオ暦 (法律上の) 命日です。適切なヘブライ暦の記念日とその前日に、このダイアリ・エントリは現れます (ヨーロッパ・スタイルでは、パラメータの順序が *day*, *month*, *year* に変わります)。

36.10 約束日時合図のカスタマイズ

Emacs が自分の約束日時の合図をどのように行なうか、そして、どれだけの間、合図を行なうかを、これら変数の設定で完全に指定できます:

`appt-message-warning-time`
: 約束日時のどれだけ前に合図を開始するかの分数。デフォルトは 10 分。

`appt-audible`
: これが非 `nil` ならば、 Emacs は端末ベルを約束日時の合図として鳴らします。デフォルトは `t`。

`appt-visible`
: これが非 `nil` ならば、 Emacs は約束についてのメッセージをエコー領域に表示します。デフォルトは `t`。

`appt-display-mode-line`
: これが非 `nil` ならば、 Emacs は約束日時までの分数をモード行に表示します。デフォルトは `t`。

`appt-msg-window`
: これが非 `nil` ならば、 Emacs は約束についてのメッセージを別ウィンドウに表示します。デフォルトは `t`。

`appt-disp-window-function`
: この変数は、約束についてのメッセージの別ウィンドウを作るために使う関数を保持します。

`appt-delete-window-function`
: この変数は、時間がきたときに、約束についてのメッセージ・ウィンドウを取り除くために使う関数を保持します。

`appt-display-duration`
: 約束についてのメッセージを表示する秒数です。デフォルトは 5 秒です。

付録 A ヒントと標準的な作法

この章には Emacs Lisp の新たな機能の説明ではなく、前章までに説明された機能を効果的に使用するためのアドバイスが記述されています。

A.1 明確な Lisp プログラムの記述

以下は、広く使用されることを目的とする Lisp コードを書くときによく犯される誤りを避けるためのヒントや作法です。

- 全ての大域変数は同じ名前空間を共有し、全ての関数も大域変数とは別の名前空間を共有します。そこで、他の Lisp プログラムのものと区別する短い単語を選び、全ての大域変数、定数、関数の名前がこの単語で始まるようにしてください。これにより名前の衝突が起きることを防ぐことができます。

  この規則は、 Emacs Lisp ではプリミティブではないが、伝統的な Lisp では primitive であるものの名前にも (たとえば `cadr` であっても) 適用されます。たとえば、いくつもの `cadr` を定義するうまい方法があります。安全のために、 `cadr` とせずに、 `foo-cadr` もしくは `mylib-cadr` などとしてください。

  ある名前 (たとえば `twiddle-files` という名前で) で Emacs の配布に加えられるべきだと思われる関数を書いたときも、あなたのプログラムの中でその名前を使って呼ばないでください。とりあえず、あなたのプログラムの中では `mylib-twiddle-files` と呼んでおいて、その関数を Emacs に加えるよう ‘`bug-gnu-emacs@prep.ai.mit.edu`’ 宛にメイルを送ってください。

  一つの接頭語では不十分である場合は、意味があるのであれば複数の共通な接頭語を使っても構いません。

  接頭語と残りのシンボル名はハイフン ‘-’ で区切ってください。これは Emacs 自身やほとんどの Emacs Lisp プログラムと一貫します。

- 少なくともライブラリ・プログラムが複数のエントリ・ポイントをもつのであれば、そのライブラリ・プログラムの中で `provide` 関数を用いると便利です。

- あるファイルが他のライブラリ・プログラムをあらかじめロードしておくことが必要であれば、ファイルの先頭のコメントに明記してください。またそれらのライブラリ・プログラムを確実にロードするように、 `require` を使用してください。

- ファイル *foo* が、別のファイル *bar* で定義されたマクロを使用する場合は、 *foo* 内でそのマクロを最初に使用する以前に、 `require` が必要です。

  ```
  (eval-when-compile (require 'bar))
  ```

  なお、 `require` がうまく働くためには、 *bar* が `(provide '` *bar*`)` を含んでいなければなりません。

  このようにすることにより、 *foo* がバイト・コンパイルされるときに *bar* がロードされます。こうしないと、必要なマクロをロードせずに *foo* をコンパイルしてしまうため、コンパイルされたコードは正しく動作しなくなります。 12.3 節「マクロとバイトコンパイル」 p.164 を参照してください。

`eval-when-compile` を用いることにより、コンパイルされた *foo* が使用されるときに、*bar* がロードされるのを防ぐことができます。

- 新たな主モードを定義するときは、通常の主モードが行っているように、`run-hooks` を用いてフック変数を実行するようにしてください。 20.4 節「フック」p.335 を参照してください。
- ある関数の目的が、ある条件が真であるか偽であるかを調べるものである場合は、その関数名を ‘p’ で終るようにしてください。名前が一語であるときは、単に ‘p’ をつけてください。名前が複数の単語からなる場合は、 ‘-p’ をつけてください。 (例 `framep`、 `frame-live-p`)
- ユーザ・オプション変数が真偽の条件を記録するものであるときは、 ‘`-flag`’ で終る名前にしてください。
- 主モードで `C-c` *letter* をキーとして定義しないでください。これらのシーケンスはユーザのために残してあるものです。ユーザのために予約されているのは、これらのシーケンスだけですから、これらを使用されると困ります。

  かわりに `C-c` *non-letter* というシーケンスを用いてください。これは主モードのために予約されています。

  この規約に従うように、 Emacs バージョン 18 の全ての主モードを変更するのは大変な作業でした。この規約を守らないことは、その作業を無駄にし、さらにはユーザにとっても不便です。
- `C-c` に `{`、`}`、 `<`、`>`、`:` もしくは`;` が続くシーケンスも主モードのために予約されています。
- `C-c` に他の句読点文字が続くシーケンスは、副モードに割り当てられています。それらを主モードで使用することが完全に禁止されているわけではありませんが、もし主モードで使用すると、主モードでのキー・バインドが副モードにより変更されるということが起きます。
- 他の (`C-c` も含めた) 前置文字に `C-h` が続くシーケンスをバインドしないようにしてください。 `C-h` をバインドしなければ、 `C-h` は自動的に、その前置文字のサブコマンドのリストを表示するヘルプ文字となります。
- 連続した二つの `ESC` で終る場合を除いて、 `ESC` で終るキー・シーケンスをバインドしてはなりません (つまり `ESC ESC` で終るシーケンスを用いることは構いません)。

  ある状況 (context) で前置文字以外で `ESC` をバインドすると、その状況でのファンクション・キーによるエスケープ・シーケンスと区別がつかなくなるため、この規則があります。
- アプリケーションは、シフト・キーを押した状態のマウス・ボタン 1 によるイベントをバインドしてはいけません。つまり `S-mouse-1`、 `M-S-mouse-1`、 `C-S-mouse-1` などが禁止されます。これらはユーザのために予約されています。
- ユーザが通常バッファのテキストを変更しないようなモードであれば、そのバッファのテキストからの参照を行うようなコマンドに `mouse-2` を用いてください。 Dired、 Info、 Compilation、 Occur などのモードがこの例です。
- あるパッケージが通常の Emacs の振舞いを変更するようなものである場合は、その効果を有効および無効にするコマンドを提供するのがよいでしょう。その効果を有効もしくは無

効にするために *whatever*-mode という名前のコマンドを定義し、それを自動ロード (autoload) してください (13.2 節「自動ロード」 p.173 参照)。パッケージを単にロードしただけでは目に見える効果がないよう、つまりその効果を有効にしないように設計してください。そうしてユーザがそのコマンドを起動することによって効果を有効にします。

- Emacs のプリミティブに別名 (aliase) を定義しないでください。標準の名前を用いてください。
- Emacs のプリミティブを再定義するのは特に避けてください。それは特定のプログラムに対してはうまく動作するかもしれませんが、ほかのプログラムに対しても動作する保証はどこにもありません。
- あるファイルが標準的な Emacs の関数やライブラリ・プログラムを置き換えるのであれば、そのファイルの先頭にどの関数が置き換えられ、もとの関数とどのように振舞いが変わるかを目立つように記述してください。
- Emacs Lisp ソース・ファイルの名前は 13 文字以下にしてください。こうすることによって、そのファイルをコンパイルしても、そのコンパイルされたファイルの名前は 14 文字以下になります。これは全ての Unix システムに十分な長さです。
- プログラムの中で `next-line` と `previous-line` を使用しないでください。ほとんど全ての場合 `forward-line` の方が便利であり、安全です。 27.2.4 節「テキスト行単位での移動」 p.460 を参照してください。
- マークを設定することがあなたのプログラムの意図した目的でないかぎり、マークを設定する関数を呼んではなりません。マークはユーザが用いるものであるため、ユーザのためになるのではないかぎりマークを変えてはなりません。 28.6 節「マーク」 p.475 を参照してください。

  特に以下の関数は用いてはいけません。

  - `beginning-of-buffer`, `end-of-buffer`
  - `replace-string`, `replace-regexp`

  これらの関数のマークを変更するという機能をユーザに対して提供するつもりではなく、ただ単にポイントを移動したい、もしくは文字列を置き変えたいというのであれば、これらの関数を 1、 2 行の簡単な Lisp コードで置き換えることができます。
- ベクタを使う特別な理由がないかぎり、ベクタのかわりにリストを使用してください。Lisp はベクタよりもリストを扱う機能を多く持っているため、通常リストを用いる方が便利です。

  ベクタは、要素の挿入や削除が必要でなく (リストだけがこのようなことができます)、 (最初から最後まで検索するのではなく) 不規則にアクセスするような大きなテーブルには有効です。
- エコー領域にメッセージを表示するためには、 `princ` よりも `message` 関数の方が適切です。 35.4 節「エコー領域」 p.603 を参照してください。
- エラー状態を検出した場合は、関数 `error` (もしくは `signal`) を呼んでください。関数 `error` はもとの関数に返りません。 9.5.3.1 節「エラーを発行する方法」 p.119 を参照してください。

エラーの通知に message、 throw、 sleep-for もしくは beep を使用しないでください。

- エラー・メッセージは大文字で開始してください。またピリオドでは終らないでください。
- 再帰編集 (recursive edit) を使うことは避けてください。そのかわりに、 Rmail の e コマンドが行っているように、もとのローカル・キーマップに戻るようなコマンドを含む、新たなキーマップを用いてください。もしくは edit-options コマンドが行っているように、別のバッファに切り替わった後、ユーザの操作により元に戻るようにしてください。 18.10 節「再帰編集」 p.288 を参照してください。
- 他のいくつかのシステムでは、変数の名前の最初と最後を '*' とする規約があります。 Emacs Lisp ではこの規約を採用しません。あなたのプログラムでも採用しないでください (Emacs ではこのような名前をプログラムによって生成されるバッファの名前にのみ用います)。全てのライブラリが同じ規約を用いていれば、ユーザに Emacs がより一貫性があるように見えます。
- それぞれの関数を、デフォルトの字下げパラメータを用いた C-M-q (indent-sexp) によって字下げしてください。
- 閉じ括弧を一行に一つずつ記述するような習慣をつけないでください。 Lisp プログラマの間では、これはよくない書き方であるとされています。ただしまれに、多くの閉じ括弧が連続するようなときに、それらを一つか二つに分けて記述するのが有効であるような場合もあります。
- 複製したものを誰かに渡すときは、著作権表示をつけてください。 Emacs の配布中の Lisp ファイルの先頭にあるものと同じものを用いてください。あなたが FSF に著作権を譲渡する書式に署名をしたことがないのであれば、著作権表示の FSF の名前があるところにあなたの名前を入れてください。

A.2 コンパイルされたコードを速くする

バイト・コンパイルされた Lisp プログラムの実行速度を改善する方法について説明します。

- プログラムのプロファイルを取るために、 'profile' ライブラリを使用してください。詳細は 'profile.el' を参照してください。
- 可能なかぎり、再帰呼出しを使わず、繰り返しを使ってください。 Emacs Lisp では、コンパイルされた関数が別のコンパイルされた関数から呼ばれる場合でも、関数呼出しは遅いものです。
- プリミティブのリスト検索関数の memq、 member、 assq もしくは assoc を使う方が、明示的な繰り返しを使うよりずっと高速です。これらのプリミティブ検索関数が使えるように、データ構造自体を再検討するのもよいでしょう。
- いくつかの組込み関数は、バイト・コンパイルされたコードでは通常の関数呼出しが行なわれないように特別に扱われます。これらの関数を使うことを推奨します。ある関数をコンパイラがこのように特別扱いするかどうかを調べるには、その関数の byte-compile 属性を調べてください。もし byte-compile 属性が nil でなければ、その関数は特別扱いされます。

たとえば次のようにすれば、 aref は特別扱いされるが (6.3 節「配列に関する関数」p.88 参照)、 elt は特別扱いされないされないことがわかります (6.1 節「シーケンス」p.85 参照)。

```
(get 'aref 'byte-compile)
     ⇒ byte-compile-two-args
```

```
(get 'elt 'byte-compile)
     ⇒ nil
```

- あなたのプログラムの実行時間の多くを小さな関数の実行が占めるのであれば、それをインライン関数にしてください。これにより関数呼出しを省くことができます。ただし関数をインラインにするとプログラムの変更時に制約が生ずるため、ユーザにとって速度が気になるぐらい遅いところを高速化する目的以外には使用しないでください。 11.9 節「インライン関数」p.160 を参照してください。

A.3 読みやすい説明文字列を書く

説明文字列 (documentation string) の書き方の作法について説明します。

- ユーザが知る必要のある全てのコマンド、関数、変数に説明文字列をつけてください。
- Lisp プログラムの内部的な関数やサブルーチンでも、説明文字列があるとよい場合があります。初期の Emacs では説明文字列のかわりにコメントを使う方がメモリ効率の点で有利でしたが、現在のバージョンではそのようなことはありません。
- 説明文字列の最初の一行は、要約を表わす一つか二つの完全な文にしてください。 M-x apropos はこの一行だけを表示するので、要約が示されなければ使いものになりません。さらに行を大文字で開始し、ピリオドで終ってください。

  説明文字列にその関数や変数の使用法の詳細を加えることもできます。これらの行も完全な文としてください。ただし、これらはそれが見やすいのであれば、行詰め (fill) を行ってください。
- 一貫性のために、説明文字列の最初の行の動詞を “to” を除いた不定詞として表わしてください。たとえば、 “Returns the cons of A and B.” とするよりも “Return the cons of A and B.” としてください。通常は最初のパラグラフの残りの部分もこうするとよいでしょう。逆に残りのパラグラフは主語がある方がよいでしょう。
- 説明文字列は、受動態ではなく能動態で、未来形ではなく現在形で書いてください。たとえば “A list containing A and B will be returned.” とせずに “Return a list containing A and B.” としてください。
- 不必要に “cause” という単語 (もしくはその同義語) を使わないでください。たとえば “Cause Emacs to display text in boldface,” ではなく “Display text in boldface.” としてください。
- 説明文字列を空白文字で開始したり、終ったりしないでください。
- 80 桁の画面の Emacs ウィンドウに合うように、説明文字列を構成してください。ほとんどの行を 60 文字を越さないようにするとよいでしょう。ただし最初の行だけは、そこに入れるべき情報を記述するために、さらに長くしても構いません。

単に説明文字列全体を行詰めをするのではなく、注意して改行を入れるとずっと読みやすくなります。説明文字列が長い場合は、話題ごとに空白行で区切ってください。

- ソース・コード中で最初の行に揃うように説明文字列の 2 行目以降を字下げしてはいけません。ソース・コードでの見ためはよくなりますが、ユーザには読みにくくなります。最初の二重引用符 (double-quote) の前の字下げは、文字列の一部ではないことに気をつけてください。
- ユーザが対話的に設定するような変数の場合は、変数の説明文字列の最初の文字を '*' としてください。その値が長いリストや関数であったり、初期化ファイルの中でのみ設定するようなものの場合は、説明文字列を '*' で開始しないでください。 10.5 節「グローバル変数の定義」p.131 を参照してください。
- 真か偽かを示す変数の説明文字列の最初は、 "`nil` でない場合は . . ." などというようにしてください。こうすることによって、全ての `nil` でない値が等価であることを明確になり、 `nil` および `nil` でない値がそれぞれ何を意味するのかも明確に示されます。
- 関数の説明文字列がその引数の値について説明する場合は、引数の名前を大文字にしてそれがその値の名前であるように表記してください。たとえば関数 / の説明文字列では、 2 番目の実引数の名前が `divisor` であるため、それを '`DIVISOR`' としています。

  たとえばリストやベクタを、変化する文法的要素 (subunits) に分解して代表的に示す場合などは、メタ構文上の変数 (meta-syntactic variable) は全て大文字としてください。[1]
- 説明文字列が Lisp シンボルを参照するときは、引用符 (single-quoate) で囲い、それが表示されるように (通常は小文字で) 記述してください。 (例: '`'lambda'`') ただし例外が二つあり、 `t` と `nil` は引用符をつけないで記述してください。
- 説明文字列に直接キー・シーケンスを書かないでください。そのかわりに '`\\[...]`' 形式を用いてください。たとえば、 '`C-f`' と書く代わりに '`\\[forward-char]`' と書いてください。 Emacs が、説明文字列を表示するときに、 `forward-char` が対応づけられているキーに置き換えます (この例では、通常は '`C-f`' となりますが、ユーザがキー・バインドを変更している場合は別の文字となります)。 21.3 節「説明文字列でのキー・バインドの置換」p.342 を参照してください。
- 主モードの説明文字列では、グローバル・マップではなく、そのモードのローカル・マップのキー・バインドを参照するのが望ましいでしょう。したがって、どのキー・マップが用いられるか示すために、説明文字列中で一度 '`\\<...>`' 形式を使用してください。さらにこれは最初に使われる '`\\[...]`' よりも以前に行ってください。 '`\\<...>`' の中は、その主モードのローカル・キーマップを格納する変数の名前としてください。

  説明文字列の表示が遅くなるため、 '`\\[...]`' を何度も何度も使うのは実用的ではありません。主モードの最も重要なコマンドの説明にのみ使用して、そのモードのキーマップのその他のものの表示には '`\\{...}`' を使用してください。

---

[1] 訳注: たとえば、関数 `keymapp` の説明文字列の `(CHAR . DEFN)` とか `(SYMBOL . DEFN)` などのこと。

A.4 コメントの書き方

ここに書かれているようにコメントを記述し、字下げを行うことが推奨されます。

‘;’ 一つのセミコロン (‘;’) で始まるコメントは、全てソース・コードの右側の同じ桁に揃えてください。通常これらのコメントは、同じ行のコードが何をするかを説明するのに用いられます。 Lisp モードや関連するモードでは、 `M-;` (`indent-for-comment`) コマンドが、自動的に正しい場所にこの ‘;’ を挿入したり、すでにコメントがある場合はその位置を揃えたりします。

この例およびこれ以降の例は Emacs のソースのものです。

```
(setq base-version-list                 ; there was a base
      (assoc (substring fn 0 start-vn)  ; version to which
             file-version-assoc-list))  ; this looks like
                                        ; a subversion
```

‘;;’ 二つのセミコロン (‘;;’) で始まるコメントは、コードの字下げと同じ位置に揃えてください。通常これらのコメントは、それ以降の行の目的や、その位置でのプログラムの状態を記述します。たとえば、

```
(prog1 (setq auto-fill-function
             ...
             ...
  ;; update mode line
  (force-mode-line-update)))
```

(そのパッケージ内でのみ使用されるため) 説明文字列を持たない関数には、かわりにその関数が何を行ないどのように呼ばれるのが正しいかを説明する二つのセミコロンから始まるコメントを書くべきです。それぞれの引数の意味とその値を関数がどのように用いるかを正確に説明してください。

‘;;;’ 三つのセミコロン (‘;;;’) で始まるコメントは、左マージンの位置から開始してください。このようなコメントは、関数定義の外でのそのプログラムの設計方針を説明する一般的な記述に用いられます。

```
;;; This Lisp code is run in Emacs
;;; when it is to operate as a server
;;; for other processes.
```

三つのセミコロンで始まるコメントのもう一つの使い方は、関数の中の行をコメント・アウトするものです。コメントが左マージンに残るようにこの目的に使用します。

```
(defun foo (a)
;;; This is no longer necessary.
;;;  (force-mode-line-update)
  (message "Finished with %s" a))
```

‘;;;;’ 四つのセミコロン (‘;;;;’) で始まるコメントは、左マージンの位置から開始し、プログラムの大きな区切りごとの先頭に用いられます。たとえば

```
;;;; The kill ring
```

`M-;` (`indent-for-comment`) や `TAB` (`lisp-indent-line`) などの Emacs の Lisp モードの字下げ命令は、自動的にこの規約に従って字下げを行ないます。 *The GNU Emacs Manual* の “Manipulating Comments” 節を参照してください。

A.5 Emacs ライブラリのヘッダの規約

Emacs バージョン 19 では、 Lisp ライブラリを段落に分けたり、その作者などの情報を付加するために特別なコメントの規約を定めています。この節ではこれらの規約について説明します。まず最初に例を示します。

```
;;; lisp-mnt.el --- minor mode for Emacs Lisp maintainers

;; Copyright (C) 1992 Free Software Foundation, Inc.

;; Author: Eric S. Raymond <esr@snark.thyrsus.com>
;; Maintainer: Eric S. Raymond <esr@snark.thyrsus.com>
;; Created: 14 Jul 1992
;; Version: 1.2
;; Keywords: docs

;; This file is part of GNU Emacs.
copying permissions...
```

1 番先頭の行は次の書式です。

```
;;; file名 --- 説明
```

説明 (description) は一行に納めてください。

著作権表示に続いて、数行の『ヘッダ・コメント (header comment)』行が置かれます。それぞれ '`;; header名:`' で始まります。以下が *header* 名で使われる規約の表です。

'`Author`'
: この行には、ライブラリの少なくとも最初の作者の名前とネットワーク・アドレスを記述します。

  複数の作者がいる場合は、次のように`;;` とタブ文字で始まる継続行を用いて記述します。

```
;; Author: Ashwin Ram <Ram-Ashwin@cs.yale.edu>
;;      Dave Sill <de5@ornl.gov>
;;      Dave Brennan <brennan@hal.com>
;;      Eric Raymond <esr@snark.thyrsus.com>
```

'`Maintainer`'
: この行には、 Author 行のように、一人分の名前とアドレスもしくは一人分のアドレスのみ、もしくは '`FSF`' と書きます。この行が省略された場合は、 Author 行の作者が管理を行っているものとみなされます。上の例は、 maintainer 行が冗長であるため、あまりよい例ではありません。

  手で名前を埋め込まなくとも "管理者へ mail を送る" Lisp 関数を作ることを可能とすることが、 '`Author`' 行と '`Maintainer`' 行のもう一つの目的です。

  ネットワーク・アドレスとフル・ネームを両方記述する場合は、必ずネットワーク・アドレスを '<...>' で囲ってください。

'`Created`'
: この行には、最初にファイルを作成した日付を記述します。省略可能です。歴史的な興味のためだけのためにあります。

'`Version`'
: 個々の Lisp プログラムにバージョン番号を記録したいのであれば、この行に記述してください。

'Adapted-By'
: この行には、(たとえばこの規約に従うように変更をした場合のように) ライブラリのあまり内部にかかわらないような[2]変更を行った人の名前を記述してください。

'Keywords'
: この行には、 finder-by-keyword ヘルプ命令のためのキーワードを記述します。ユーザは、やりたいことから探すときに、あなたのプログラムをこのキーワードから探すため、この行は重要です。キーワードを分けるときは、空白文字、コンマ、もしくはその両方を用います。

少なくとも 'Author' と 'Keywords' の行は、全ての Lisp ライブラリに必要です。適切であれば他の行も使用してください。別のヘッダ名の行を加えても構いません。標準的な意味はないため何の害もありません。

我々は、ライブラリ・ファイルの内容をいくつかに分割するために標準的なコメントを決めています。以下がその表です。

';;; Commentary:'
: そのライブラリがどのように働くかを説明する紹介説明の始まりを示します。著作権許可のすぐ後にあるべきです。

';;; Change log:'
: (変更履歴を記述する場合の) ライブラリ・ファイルの変更履歴の始まりを示します。 Emacs とともに配布されるほとんど Lisp ファイルでは、変更履歴は 'ChangeLog' に書かれており、ソース・ファイル中には全く記述されていません。したがってこれらのファイルには ';;; Change log:' 行はありません。

';;; Code:'
: 実際のプログラム・コードの始まりを示します。

';;; *filename* ends here'
: これは『フッタ行 (footer line)』で、ファイルの 1 番最後の行です。このフッタ行がないことから、ファイルの後部が切り取られたバージョンであることを知ることができます。

---

[2] 訳注: 原文では "adapt the library for installation"。

付録 B GNU Emacs の内部構造

この章では、稼動中の Emacs 実行可能形式をどのように Emacs Lisp ライブラリをプリロードしたままダンプするのか、どのように記憶領域を割りつけるのか、そして、 C のプログラマが興味をもつような GNU Emacs のいくつかの内部構造について述べます。

B.1 Emacs を作る

この節では、 Emacs の実行可能形式を作るときに起きている諸段階について説明します。この種のことはメークファイルが自動的に行なってくれるので、 Emacs を作ってインストールするときに知っておく必要はありません。この情報は Emacs の保守に関係あります。

'src' ディレクトリで C のソース・ファイルをコンパイルすると、 'temacs' という実行可能ファイルができますが、これを『bare impure Emacs』といいます。これには Emacs Lisp インタプリタと I/O ルーチンがありますが、編集コマンドはついていません。

コマンド'temacs -l loadup'では、本当に実行ができる Emacs 実行可能ファイルをつくるのに、 'temacs' を使います。この引数は 'temacs' に、ファイル 'loadup.el' 中で指定した Lisp ファイル群を評価させる、ということを指示しています。このファイル群で通常の Emacs 編集環境を準備すると、 Emacs は bare ではなくなりますが、まだ impure ではあります。

標準 Lisp ファイル群の読込みには相当時間がかかります。幸運にも、 Emacs を起動するたびにこれを繰り返すことはありません。 'temacs' は、このファイル群をプリロードした 'emacs' という実行可能プログラムをダンプすることができます。 'emacs' はもうこのファイルのロードがいらないので、とても速く立ち上がります。普通にインストールされるのはこちらの Emacs です。

'emacs' を作るには、コマンド 'temacs -batch -l loadup dump' を使います。この '-batch' で 'temacs' が端末のデータを初期化しないようにしています。これでダンプ後の Emacs の端末情報が空のままになります。引数 'dump' で 'loadup.el' にしたがって 'emacs' という新しい実行可能ファイルをダンプしています。

オペレーティング・システムによっては、ダンプのできないことがあります。こういうシステムでは、 Emacs を使うたびに、 'temacs -l loadup' というコマンドで始める必要があります。これにはけっこう時間がかかりますが、 Emacs を立ち上げるのは高々日に 1 回か、ログ・アウトしないなら週に 1 回でしょうから、この時間もそれほど深刻な問題ではありません。

'site-load.el' ライブラリを書いてプリロードするファイルを追加することができます。追加ファイル用の場所をあけるために、 'src/puresize.h' 中の PURESIZE の値を増やす必要があることもあります (十分な大きさになるまで 20000 ずつ増やしてみてください)。しかしながら、追加ファイルのプリロードの利点はマシンが速くなるにつれて減ってきています。最近のマシンではだいたい、そんなに得策ではありません。

ダンプ前に実行するほかの Lisp 式を 'site-init.el' というライブラリに書いて指定します。しかしながら、ユーザが通常の無修正の Emacs に期待する振舞いを変更する場合は、ユーザが望むときに優先できるように 'default.el' に置く方がいいでしょう。 34.1.1 節「立ち上がりの際の一連の動作一覧」p.577 を参照してください。

'loadup.el' が新しい実行可能ファイルをダンプする前に、 Snarf-documentation を呼び出して (21.2 節「説明文字列へのアクセス」p.340 参照)、プリミティブ関数とプリロードした関数 (と変数) を定義したファイルから説明文字列を探し出します。 'emacs' 実行可能ファイルを小さくするために、説明文字列は実行可能ファイルから取り除かれます。 21.1 節「説明文字列の基礎」p.339 を参照してください。

dump-emacs *to-file from-file* Function

この関数は、 Emacs の現在の状態を実行可能ファイル *to-file* にダンプします。これは (通常はファイル 'temacs' の) *from-file* からシンボルを取ります。

すでにダンプしてしまった Emacs でこの関数を使う場合、まず command-line-processed を nil にしなければうまくいきません。 34.1.4 節「コマンド行引数」p.580 を参照してください。

emacs-version Command

この関数は、走行中の Emacs のバージョン番号を示す文字列を返します。バグ報告にこの文字列を含めると役に立ちます。

```
(emacs-version)
  ⇒ "GNU Emacs 19.29.1 (i386-debian-linux) \
 of Tue Jun  6 1995 on balloon"
```

対話的に呼び出されると、この関数は、エコー領域に同じ情報を印字します。

emacs-build-time Variable

この変数の値は、ローカルなサイトで Emacs が作られた時刻です。

```
emacs-build-time
     ⇒ "Tue Jun  6 14:55:57 1995"
```

emacs-version Variable

この変数の値は、走行中の Emacs のバージョン番号です。これは "19.29.1"といった文字列になります。

以下の二つの変数は Emacs のバージョン 19.23 まではなかったため、今のところそれほど役には立ちませんが、私たちは将来これが役に立つであろうことを期待しています。

emacs-major-version Variable

整数で表わした Emacs の主バージョン番号です。 Emacs バージョン 19.29 では 19 という値になります。

emacs-minor-version Variable

整数で表わした Emacs の副バージョン番号です。 Emacs バージョン 19.29 では 29 という値になります。

B.2 純粋記憶領域

Emacs Lisp には、ユーザの作る Lisp オブジェクト用に『通常記憶領域 (normal storage)』と『純粋記憶領域 (pure storage)』の 2 種類の記憶領域があります。通常記憶領域は Emacs の

各セッションで作られた新しいデータのすべてを保持する領域です。通常記憶領域については次節を参照してください。純粋記憶領域はプリロードされる標準 Lisp ファイル中のデータで、実際の Emacs の使用中に変化することのないものにたいして使われます。

純粋記憶領域は、 'temacs' が標準の Lisp ライブラリをプリロードするときに割りつけられます。ファイル 'emacs' 中で、これは (オペレーティング・システムが許すかぎり) 読出し専用なので、マシンで同時に走る全 Emacs ジョブでメモリ空間を共有することができます。純粋記憶領域は、 Emacs のコンパイル時に固定量が割りつけられ、拡張することはできません。プリロード・ライブラリで足りなくなると 'temacs' は落ちてしまいます。こうなると、ファイル 'src/puresize.h' の PURESIZE というコンパイル時パラメータを増やすしかありません。追加のライブラリをプリロードするか、標準のライブラリに機能を追加するのでないかぎり、こうなることはまずありません。

purecopy *object* Function

この関数は、純粋記憶領域中の *object* の複製を作り、それを返します。文字列は、同じ文字群をもつものを純粋記憶領域中にただ新しく作ることで複製を行ないます。ベクタとコンス・セルの内容は、再帰的に複製します。マーカを複製するよう要求すると、エラーを通知します。

Emacs が作られダンプされた後、この関数は、空操作になります。通常は、ファイル 'emacs/lisp/loaddefs.el' だけで呼び出されますが、プリロードすると決めた場合にはこれを呼び出しているパッケージも少しあります。

pure-bytes-used Variable

この変数の値は、これまで割りつけられた純粋記憶領域のバイト数です。普通ダンプ後の Emacs で、この数は存在する純粋記憶領域の総量に非常に近くなります。もしそうでないときには、割りつける量を少なくできます。

purify-flag Variable

この変数は、 `defun` が純粋記憶領域に関数定義の複製を作るべきか否かを定めます。非 `nil` の場合、関数定義は純粋記憶領域に複製されます。

(これらの関数を共有可能で回収不能であるようにするため) 最初 Emacs を作るときに基本的な全関数をロードする間、このフラグは `t` になっています。実行可能なプログラムとして Emacs のダンプを行なうと、ダンプ前や後の値にかかわらず常に `nil` に設定されます。

Emacs の走行中にこのフラグを変更してはいけません。

B.3 ガベージ・コレクション

プログラムがリストを作ったり、(ライブラリをロードして) ユーザが新しい関数を定義したり場合、それらは通常記憶領域に置かれます。通常記憶領域が足りなくなった場合、 Emacs はオペレーティング・システムにそれ以上のメモリを 1k バイトのブロックで割りつけるよう要求します。割りつけられたブロックは、それぞれある 1 種類の型用に用いられます。つまり、シンボル、コンス・セル、マーカなどはメモリ中の異なるブロックにわけられることになります (ベク

タ、長い文字列、バッファおよびそれ以外の一部の非常に大きな編集用の型は、各オブジェクトごとに割りつけられ、短い文字列は 8k バイトのブロックに詰められます)。

記憶領域をある期間だけ用いて、その後 (たとえば) バッファを kill したり、オブジェクトに対する最後のポインタを消去したりすることにより、開放することは、非常に一般的なことです。Emacs では、この放棄された記憶領域を再生する『ガベージ・コレクタ (garbage collector)』が提供されています (この名前は伝統的なもので、この機能の直感的なたとえとしては「ゴミ再利用」“garbage recycler” の方がいいかもしれません)。

ガベージ・コレクタは、 Lisp プログラムから用いることのできるすべての Lisp オブジェクトをスキャンし、印をつけることで処理を行ないます。はじめにすべてのシンボル、その値と対応する関数定義、現在スタック上に存在するデータは用いられているものであると仮定します。これらを通して間接的に到達可能なオブジェクトもすべて用いられているという刻印を押します。

印づけが終わったとき、刻印のないものはすべてガベージです。 Lisp プログラムやユーザが何をしようとも、もはやここへ到達する方法がないため、ここを使うことは不可能です。この空間は再利用してもほかと影響しません。ガベージ・コレクタの第 2 の「掃き出し」(“sweep”) 段階では、この場所の再利用を行ないます。

この「掃き出し」段階において、用いられてないコンス・セル、シンボル、マーカは、将来の割りつけ用に『フリーリスト (free list)』につなげられます。使用中の文字列はなるべく少ない 8k ブロックに収まるようまず圧縮され、その後、空いた 8k ブロックは解放します。ベクタ、バッファ、ウィンドウ、そのほか大きなオブジェクトは、個々に `malloc` と `free` を用いて割りつけと解放を行ないます。

> **Common Lisp 注意書き:** ほかの Lisp と異なり、 GNU Emacs Lisp ではフリーリストが空になった場合にガベージ・コレクタは呼ばれません。そのかわり、 GNU Emacs Lisp は、ただ単にオペレーティング・システムへもっと多くの記憶領域を割りつけするよう要求し、 `gc-cons-threshold` バイト使われるまで処理を続けます。
>
> これは、陽にガベージ・コレクタを呼び出すことによって、 Lisp プログラムのある部分を実行中にガベージ・コレクタが動かないことを (プログラムの一部が次のガベージ・コレクションを引き起こすほど大きな空間を消費しないことを前提にして) 保証できます。

garbage-collect Command

このコマンドはガベージ・コレクションを行ない、用いられている記憶領域の量に関する情報を返します (ガベージ・コレクションは、前のガベージ・コレクションから `gc-cons-threshold` バイト以上 Lisp データが用られた場合、自動的に行なわれます)。

`garbage-collect` は以下の情報を含むリストを返します。

((*used-conses* . *free-conses*)
 (*used-syms* . *free-syms*)
 (*used-markers* . *free-markers*)
 used-string-chars
 used-vector-slots
 (*used-floats* . *free-floats*))

```
(garbage-collect)
     ⇒ ((3435 . 2332) (1688 . 0)
          (57 . 417) 24510 3839 (4 . 1))
```

次の表は各要素の説明です。

used-conses 使用中のコンス・セルの個数。

free-conses オペレーティング・システムから空間を得てはいるが、今のところ使われていないコンス・セルの個数。

used-syms 使用中のシンボルの個数。

free-syms オペレーティング・システムから空間を得てはいるが、今のところ使われていないシンボルの個数。

used-markers
使用中のマーカの個数。

free-markers
オペレーティング・システムから空間を得てはいるが、今のところ使われていないマーカの個数。

used-string-chars
文字数で測った全文字列の長さの合計。

used-vector-slots
既存のベクタの要素数の合計。

used-floats 使用中の浮動小数点数の個数。

free-floats オペレーティング・システムから空間を得てはいるが、今のところ使われていない浮動小数点数の個数。

gc-cons-threshold User Option

この変数の値は、ガベージ・コレクションが行なわれた後、次のガベージ・コレクションが自動的に呼ばれるまでに用いることのできる、記憶領域のバイト数です。コンス・セルは 8 バイト、文字列は 1 文字当り 1 バイト足す数バイトのオーバーヘッド、などのように数えます。バッファの内容は勘定に入れません。それ以降は、次に Lisp 評価子を呼び出すときをのぞき、この閾値をこえるまでガベージ・コレクションがおこりません。

この閾値の初期値は 300,000 です。もし大きな値を設定すると、ガベージ・コレクションは起こりにくくなります。ガベージ・コレクションに使われる時間は減りますが、メモリの総使用量は増加します。多量の Lisp データを作るようなプログラムを使う場合には、このようにしたくなるかもしれません。

ガベージ・コレクションの回数を増やすため、 10,000 以上の小さな値に設定することもできます。これを 10,000 以下の値に設定しても、次のガベージ・コレクションまでしか効果はありません。そのときに `garbage-collect` はこれを 10,000 に戻します。

memory-limit Function

この関数は、 Emacs が割りつけた最後のバイトのアドレスを 1024 で割ったものです。1024 で割っているのは値が Lisp 整数の範囲に収まるようにするためです。

メモリ使用量により影響を受ける動作の決定にこれを使うことができます。

B.4 Emacs のプリミティブを書く

Lisp プリミティブとは、 C で実装した Lisp 関数のことです。 Lisp で C の関数が呼び出せるようにするインタフェースの詳細を、 C のマクロで扱っています。 C のコードの書きかたを本当に理解する唯一の方法は、ソースを読むことですが、ここではある程度の情報を示しておきます。

特殊形式の例として `or` の定義を ‘`eval.c`’ から取りました (通常の関数も同様に一般的な形をしています)。

```
DEFUN ("or", For, Sor, 0, UNEVALLED, 0,
  "Eval args until one of them yields non-nil, then return that value.\n\
The remaining args are not evalled at all.\n\
If all args return nil, return nil.")
  (args)
     Lisp_Object args;
{
  register Lisp_Object val;
  Lisp_Object args_left;
  struct gcpro gcpro1;

  if (NULL (args))
    return Qnil;

  args_left = args;
  GCPRO1 (args_left);

  do
    {
      val = Feval (Fcar (args_left));
      if (!NULL (val))
        break;
      args_left = Fcdr (args_left);
    }
  while (!NULL (args_left));

  UNGCPRO;
  return val;
}
```

`DEFUN` マクロに対する引数の詳細な説明から始めましょう。見本を以下に示します。

`DEFUN` (*lname*, *fname*, *sname*, *min*, *max*, *interactive*, *doc*)

lname これは Lisp シンボルの名前で、定義する関数の名前となります。前の例の場合これは `or` です。

fname これは、この関数に対する C の関数名です。これは C コードからこの関数を呼ぶ際、用いられる名前です。慣例として名前は Lisp 名の先頭に ‘`F`’ をつけ、 Lisp 名におけるダッシュ (`-`) をすべてアンダースコアに変えたものです。したがっ

て、 C コードからこの関数を呼ぶ場合、それは `For` を呼ぶことになります。引数は `Lisp_Object` でなくてはいけないことを覚えておいてください。型 `Lisp_Object` を作るさまざまなマクロと関数は、ファイル '`lisp.h`' で宣言されています。

sname これは、 Lisp での関数を表現する subr オブジェクトのデータを保持する構造体に用いる C の変数名です。この構造体は、シンボルを生成し subr オブジェクトを定義に格納する初期化ルーチンに、 Lisp シンボル名を渡します。この名前は慣例として *fname* の '`F`' を '`S`' に置き換えたものとなります。

min これは、与えられなくてはいけない最小の引数の数です。関数 `or` の場合、引数のない場合を許しています。

max これは、固定の最大値があれば、関数が受けつける引数の最大個数です。または、引数を評価しないことを示す `UNEVALLED` か、 (`&rest` と等価な) 任意個の評価される引数をもつことを示す `MANY` をもつことになります。 `UNEVALLED` と `MANY` は両方ともマクロです。 *max* が数の場合、 *min* より小さかったり、 7 より大きかったりしてはいけません。

interactive これは、 Lisp 関数の `interactive` への引数に用いる文字列と同じ interactive 指示です。 `or` の場合、これは 0(ヌル・ポインタ) で、 `or` を対話的に呼ぶことはできないことを示します。値 `""`は、対話的に呼んだ場合に引数をとらない関数を示します。

doc これは説明文字列です。各行末で '`\n\`' を書かなければならない点を除いて、 Lisp で定義する関数の説明文字列と同じように書きます。特に、 1 行目は一つの文にしてください。

`DEFUN` マクロ呼びだしの後で、すべての C の関数にあるような引数名リストと、その後に普通の C の引数宣言を書かなくてはなりません。引数の最大個数が固定の関数は、各 Lisp の引数に対して C の全引数を `Lisp_Object` 型として宣言しなくてはいけません。 Lisp 関数に引数の数の上限がないとき、 C の実装ではちょうど二つの引数を取ります。 1 番目が Lisp 引数の個数で、 2 番目が引数の値を格納したブロックのアドレスです。これらの型はそれぞれ `int` と`Lisp_Object *`です。

関数 `For` 中でマクロ `GCPRO1` と `UNGCPRO` の使い方に注目してください。 `GCPRO1` は変数をガベージ・コレクタから保護するのに使います。これは、ガベージ・コレクタがこの変数の中を見て、この内容がアクセス可能なオブジェクトであるとみなすようにします。これは `Feval` を呼び出す場合や、 `Feval` を直接間接に呼び出すもののうちのいずれかを呼び出す場合には常に必要となります。このような場合、 (そのような関数の呼出しの後で) 再び参照しようとしている Lisp オブジェクトも全部保護しなければなりません。 `UNGCPRO` は現在の関数で保護している変数の保護を取消します。これは陽にそうする必要があります。

ほとんどのデータ型については、オブジェクトへのポインタを少なくとも一つ保護すれば十分です。オブジェクトが再利用されるまでは、そこへのポインタはずっと有効なままです。ガベージ・コレクタが移動させてしまうので、文字列について、これはあてはまりません。ガベージ・コレクタが文字列を移動するとき、わかる範囲のすべてのポインタのさしかえを行ないますが、

それ以外のポインタは無効になります。したがって、ガベージ・コレクションが起こりうる場所以降のすべての文字列へのポインタを保護しなければなりません。

マクロ `GCPRO1` は局所変数を 1 個保護します。 2 個保護したいときは、 `GCPRO2` を使います。 `GCPRO1` を繰り返してもうまくいきません。なおマクロ `GCPRO3` と `GCPRO4` もあります。

このマクロ群は暗に `gcpro1` といった局所変数を使います。これは `struct gcpro` 型で陽に宣言しなければなりません。したがって、 `GCPRO2` を使うときは `gcpro1` と `gcpro2` を宣言します。残念ながら、ここでトリッキーな詳細について説明することはできません。

いったんダンプしてしまった Emacs では、 (いったん Emacs をダンプしたからにはもう書き変えないという場合以外) 静的変数や外部変数で C の初期値を使ってはいけません。ダンプ後の Emacs で、初期値のある変数は、 (あるオペレーティング・システムで) 読出し専用のメモリ領域に割りつけられるためです。

関数の中では、静的な変数を使ってはいけません。静的変数はファイルのトップ・レベルに置いてください。これは、あるオペレーティング・システムにおいてキーワードの `static` を空マクロとして定義するためにそうする必要があります (この定義は、初期化の有無にかかわらず、そのようなシステムにおいて静的に宣言した全変数がダンプ後に読出し専用になってしまうために、使われています)。

C の関数を定義するだけでは Lisp プリミティブは有効になりません。プリミティブの Lisp シンボルを生成し、関数セルに適切な subr オブジェクトを格納する必要があります。これは以下のようなコードになります。

```
defsubr (&subr-structure-name);
```

ここで *subr-structure-name* は `DEFUN` の第 3 引数にした (訳注: subr オブジェクトの) 変数名です。

すでに Lisp のプリミティブを定義しているファイルにプリミティブを追加する場合、 (そのファイルの終わり近くにある) `syms_of_`*something* という名前の関数を探して、 `defsubr` への呼出しを追加してください。ファイルにその関数がない場合や新しくファイルを作る場合、ファイルに `syms_of_`*filename* を (たとえば、 `syms_of_myfile` といったふうに) 追加してください。そして ‘`emacs.c`’ でこのような関数を呼び出している場所を探し、 `syms_of_`*filename* の呼出しを追加します。

関数 `syms_of_`*filename* は、 Lisp 変数としても見える C の変数を定義する場所でもあります。 `DEFVAR_LISP` は、 Lisp から可視な `Lisp_Object` 型の C 変数を作ります。 `DEFVAR_INT` は、 Lisp での値が常に整数になるように見える `int` 型の C 変数を作ります。 `DEFVAR_BOOL` は、 Lisp で `t` か `nil` のどちらかを値に取るように見える C の `int` 型変数を作ります。

次に、より複雑な引数をもつ別の関数を示します。これは X ウィンドウ・システム用のコードからのもので、 Lisp オブジェクトを操作するマクロと関数の使用法を示します。

```
DEFUN ("coordinates-in-window-p", Fcoordinates_in_window_p,
  Scoordinates_in_window_p, 2, 2,
  "xSpecify coordinate pair: \nXExpression which evals to window: ",
  "Return non-nil if POSITIONS is in WINDOW.\n\
  \(POSITIONS is a list, (SCREEN-X SCREEN-Y)\)\n\
```

```
  Returned value is list of positions expressed\n\
  relative to window upper left corner.")
  (coordinate, window)
     register Lisp_Object coordinate, window;
{
  register Lisp_Object xcoord, ycoord;

  if (!CONSP (coordinate)) wrong_type_argument (Qlistp, coordinate);
  CHECK_WINDOW (window, 2);
  xcoord = Fcar (coordinate);
  ycoord = Fcar (Fcdr (coordinate));
  CHECK_NUMBER (xcoord, 0);
  CHECK_NUMBER (ycoord, 1);
  if ((XINT (xcoord) < XINT (XWINDOW (window)->left))
      || (XINT (xcoord) >= (XINT (XWINDOW (window)->left)
                            + XINT (XWINDOW (window)->width))))
    return Qnil;
  XFASTINT (xcoord) -= XFASTINT (XWINDOW (window)->left);
  if (XINT (ycoord) == (screen_height - 1))
    return Qnil;
  if ((XINT (ycoord) < XINT (XWINDOW (window)->top))
      || (XINT (ycoord) >= (XINT (XWINDOW (window)->top)
                            + XINT (XWINDOW (window)->height)) - 1))
    return Qnil;
  XFASTINT (ycoord) -= XFASTINT (XWINDOW (window)->top);
  return (Fcons (xcoord, Fcons (ycoord, Qnil)));
}
```

C で定義してある関数以外は、 C のコードから関数名での呼出しができないことに注意してください。 Lisp で書いた関数を呼び出すには、 Lisp 関数 `funcall` の本体である `Ffuncall` を使います。 Lisp 関数の `funcall` は引数を無制限に受けつけるので、 C では Lisp レベルでの引数の個数と、その値を格納した 1 次元配列の二つの引数を受けつけます。 Lisp レベルでの最初の引数は呼び出そうとする Lisp の関数で、残りはそれへの引数です。 `Ffuncall` は評価子を呼び出すことがあるので、 `Ffuncall` のまわりではポインタを保護する必要があります。

C の関数 `call0`、 `call1`、 `call2` などによって、一定個数の引数で Lisp 関数を手軽に呼び出すことができます。これは `Ffuncall` の呼出しによるものです。

例になるものを探すには、 ‘`eval.c`’ は大変適したファイルです。 ‘`lisp.h`’ には、いくつか重要なマクロの定義や関数 (訳注: の宣言) があります。

B.5 オブジェクトの内部構造

GNU Emacs Lisp は、多くの異なる型を扱います。実際のデータはヒープに格納され、プログラムは唯一ポインタを通し、そのデータに対しアクセスします。ポインタはほとんどの実装において 32 ビットの幅をもちます。オペレーティング・システムと、 Emacs をコンパイルするマシンの型によって、 3 ビットにオブジェクトの型を示すタグを用い、残りの 28 ビットくらいでオブジェクトを指します (原著では「オブジェクトを指すのに 24 ビットから 26 ビット分ぐらいを用い、残りの 6 ビットから 8 ビットをオブジェクトの型を示すタグとして用います。」としている)。

Lisp オブジェクトに対するアクセスは、すべてタグのつけられたポインタで表わすため、どんなオブジェクトであってもオブジェクトの Lisp データ型を決めることは常に可能です。 C のデータ型 `Lisp_Object` は、どんな型の Lisp オブジェクトでも保持することができます。普通の変数は、 `Lisp_Object` 型なので、どんな Lisp 値でもとることができます。実際のデータ型は実行時に決定できます。関数引数も同様です。ある型の引数のみを受け取る関数が必要な場合、適切な述語を用い、明示的に型の検査を行なう必要があります (2.5 節「型述語」 p.36 参照)。

B.5.1 バッファの内部構造

おのおののバッファは、 Lisp プログラマが直接アクセスすることのできないフィールドがあります。ここでは C のコードで使う名前でこれらを解説します。 Lisp プログラムからは Lisp プリミティブでアクセス可能なフィールドが、数多くあります。

`name`
: バッファ名は、バッファを名づける文字列です。これは一意であることが保証されています。 24.3 節「バッファの名前」 p.395 を参照してください。

`save_modified`
: このフィールドには、バッファを最後にセーブした時刻を、整数で格納します。 24.5 節「バッファの変更」 p.399 を参照してください。

`modtime`
: このフィールドは、読み込んでいるファイルの修正時刻を格納します。これはそのファイルが書かれるか読まれるかしたときに設定されます。バッファがファイルに書かれるたびに、このフィールドはファイルの修正時刻と比較されます。 24.5 節「バッファの変更」 p.399 を参照してください。

`auto_save_modified`
: このフィールドは、バッファが最後に自動セーブされた時刻を格納します。

`last_window_start`
: このフィールドは、バッファの `window-start` 位置、すなわち、最後にバッファがウィンドウに表示された際、その表示を開始したバッファ中の位置を格納します。

`undo_list`
: このフィールドは、バッファの undo リストを指します。 29.9 節「取り消し」 p.492 を参照してください。

`syntax_table`
: (原著では `syntax_table_v` としています) このフィールドは、バッファに対する構文テーブルを格納します。第 31 章「構文テーブル」 p.541 を参照してください。

`downcase_table`
: このフィールドは、テキストを小文字に変換するときの変換テーブルを格納します。 4.8 節「大文字小文字変換のカスタマイズ」 p.65 を参照してください。

`upcase_table`
: このフィールドは、テキストを大文字に変換するときの変換テーブルを格納します。 4.8 節「大文字小文字変換のカスタマイズ」 p.65 を参照してください。

`case_canon_table`
: このフィールドは、 case-fold 検索でテキストを正準化するときの変換テーブルを格納します。 4.8 節「大文字小文字変換のカスタマイズ」p.65 を参照してください。

`case_eqv_table`
: このフィールドは、 case-fold 検索の等価テーブルを格納します。 4.8 節「大文字小文字変換のカスタマイズ」p.65 を参照してください。

`display_table`
: このフィールドは、バッファの表示テーブルか、それが存在しない場合は、 `nil` を格納します。 35.14 節「表示テーブル」p.618 を参照してください。

`markers`
: このフィールドは、現在、バッファの中を指すすべてのマーカの連鎖を格納します。バッファのテキストを消去したりバッファのギャップを移動したりした場合にはこれらの各マーカを検査し、多分更新をしなくてはいけません。

`backed_up`
: このフィールドは、このバッファに読み込んでいるファイルのバックアップをとったか否かを示すフラグです。

`mark`
: このフィールドは、バッファに対するマークを格納します。マークはマーカですから、マークはリスト `markers` にも含まれます。 28.6 節「マーク」p.475 を参照してください。

`mark_active`
: このフィールドは、バッファのマークがアクティブなら非 `nil` です。

`local_var_alist`
: このフィールドは、バッファ・オブジェクトの特別な場所 (ここでは説明を省きます) に存在するローカル変数以外の、バッファのローカル変数と、それに対応する値とを含む連想リストを格納します。 10.9 節「バッファローカルな変数」p.140 を参照してください。

`base_buffer`
: このフィールドは、バッファの基本バッファ (もし間接バッファなら) か `nil` を保持します。

`keymap`
: このフィールドは、バッファのローカル・キーマップを保持します。第 19 章「キーマップ」p.293 を参照してください。

`overlay_center`
: このフィールドは、現在のオーバレイの中央位置を保持します。 35.9 節「オーバレイ」p.609 を参照してください。

`overlays_before`
: このフィールドは、このバッファの現在のオーバレイの中央位置かその前で終わるオーバレイのリストを保持します。終了位置の逆順で整列してあります。

`overlays_after`
: このフィールドは、このバッファの現在のオーバレイの中央位置よりも、後ろで終わるオーバレイのリストを保持します。先頭位置の昇順に整列してあります。

B.5.2 ウィンドウの内部構造

ウィンドウは、アクセス可能な以下のフィールドがあります。

`frame`
: ウィンドウのあるフレーム。

`mini_p`
: もしこのウィンドウがミニバッファのウィンドウならば非 `nil`。

`buffer`
: ウィンドウが表示しているバッファ。これはウィンドウの存在期間中に変更されることがしばしばです。

`dedicated`
: もしこのウィンドウがそのバッファ専用ならば非 `nil`。

`pointm`
: これは、ウィンドウが選択されている場合、カレント・バッファのポイントです。選択されていない場合、前の値を保持します。

`start`
: ウィンドウに表示される最初の文字のバッファ中の位置。

`force_start`
: もしこのフラグが非 `nil` ならば、 Lisp プログラムが陽にウィンドウをスクロールしたことを示します。ポイントが画面からはずれた場合、これは次の再表示で何がおきるかに影響します。ポイントのまわりのテキストを見せるためにウィンドウをスクロールする代わりに、画面上のある場所にポイントを動かします。

`last_modified`
: 最後にウィンドウの再表示を行なったときの、ウィンドウのバッファの `modified` フィールド。

`last_point`
: 最後にウィンドウの再表示を行なったときの、ウィンドウのバッファのポイントの値。

`left`
: これは、ウィンドウの左端の桁位置です (画面の左端は桁 0 です)。

`top`
: これは、ウィンドウの上端の行位置です (画面の上端は行 0 です)。

`height`
: ウィンドウの高さ (行数)。

`width`
: ウィンドウの幅 (桁数)。

`next`
: これは、同胞 (訳注: slibing) の連鎖の中で次に位置するウィンドウです。同胞のグループで最も右あるいは下のウィンドウでは、 `nil` です。

`prev`
: これは、同胞の連鎖の中で前に位置するウィンドウです。同胞のグループで最も左あるいは上のウィンドウでは、 `nil` です。

`parent` Emacs は内部的にウィンドウの木をもっています。同胞の各グループには、同胞全部を含んだ親ウィンドウがあります。このフィールドはその親ウィンドウを指します。

親ウィンドウはバッファの表示をしません。そして子ウィンドウの形状のほかは、表示に際した役割をそれほどもおってはいません。 Emacs Lisp プログラムでは親ウィンドウにアクセスすることは通常できません。プログラム上は、実際にバッファを表示する、木の葉にあたるウィンドウだけを操作します。

`hscroll` これは、ウィンドウ中の表示が水平方向に左にスクロールされるときの桁数です。通常は 0 です。

`use_time` これは、ウィンドウが最後に選択された時刻です。このフィールドは `get-lru-window` で用いられます。

`display_table`
ウィンドウの表示テーブルか、もしそれがなければ `nil`。

`update_mode_line`
非 `nil` はウィンドウのモード行の更新が必要なことを意味します。

`base_line_number`
バッファ上のある位置の行番号または `nil`。これはモード行にポイントの行番号を表示するときに使います。

`base_line_pos`
行番号のわかっているバッファ上の位置、あるいは不明な場合は `nil`。

`region_showing`
もしこのウィンドウでリージョン (かその一部分) が強調表示しているなら、このフィールドはリージョンの一端となるマークの位置。さもなくばこのフィールドは `nil`。

B.5.3 プロセスの内部構造

プロセスのフィールドは、以下のとおり。

`name` プロセスの名前の文字列。

`command` プロセスを始動したときのコマンド引数を格納するリスト。

`filter` バッファのかわりに、プロセスからの出力を受け取るのに用いられる関数。

`sentinel` プロセスがシグナルを受け取ったときに呼ばれる関数か、 `nil`。

`buffer` プロセスに対応したバッファ。

`pid` プロセスの Unix プロセス ID の整数。

`childp` フラグで、本当の子プロセスなら非 `nil`。ネットワーク接続なら `nil`。

`mark` このプロセスからバッファに挿入された最後の出力の終りを設定するマーカ。バッファの最後であることがしばしばですが、常にそうではありません。

`kill_without_query`
: もし非 `nil` ならば、このプロセスの実行中に Emacs が死ぬ際、このプロセスを殺す前の確認をとらないことを意味します。

`raw_status_low`
`raw_status_high`
: この二つのフィールドは、 `wait` システムコールが返したプロセス状態を、 16 ビットずつ記録しています。

`status` `process-status` が返すべきプロセス状態。

`tick`
`update_tick`
: もしこれら二つのフィールドが等しくないなら、見張りを実行するかプロセスバッファにメッセージを挿入するかしてプロセス状態の変更が必要であることを示します。

`pty_flag` もしサブプロセスとの通信に PTY を使うなら非 `nil`、もしパイプを使うのなら `nil`。

`infd` プロセスからの入力ファイル記述子。

`outfd` プロセスへの出力ファイル記述子。

`subtty` 子プロセスが使う端末のファイル記述子 (システムによっては記録する必要がないので、値は `nil` です)。

`tty_name` サブプロセスの使っている端末の名前か、または、パイプを使っている場合は `nil`。

付録 C 標準のエラー

ここにあげるのは標準の Emacs のすべてのエラー・シンボルの一覧であり、概念によってグループ化されています。その一覧は個々のシンボルのメッセージ (シンボルの `error-message` 属性) と、そのエラーがどのようにして起きるかを説明している箇所への相互参照を含んでいます。

個々のエラー・シンボルはシンボルのリストである `error-conditions` 属性を持っています。通常、このリストはエラー・シンボル自身とシンボル `error` を含んでいます。これ以外に中間的なクラスで `error` よりも狭いが単なるエラー・シンボルよりも広い属性をもつシンボルを含んでいることもあります。例えば、ファイルのアクセスに関するエラーはすべて `file-error` という属性を持っています。

特別な例外として、エラー・シンボル `quit` はとりやめがエラーとみなされないので、状態 `error` を持っていません。

どのようにエラーが発生し、どのようにエラーをハンドルするかについては、 9.5.3 節「エラー」p.119 を参照してください。

symbol　　*string*; *reference*。

`error`
: `"error"`
9.5.3 節「エラー」p.119 を参照してください。

`quit`
: `"Quit"`
18.8 節「とりやめ」p.284 を参照してください。

`args-out-of-range`
: `"Args out of range"`
第 6 章「シーケンス、配列、ベクタ」p.85 を参照してください。

`arith-error`
: `"Arithmetic error"`
第 3 章「数値」p.41 の `/` と `%` を参照。

`beginning-of-buffer`
: `"Beginning of buffer"`
27.2 節「移動」p.458 を参照してください。

`buffer-read-only`
: `"Buffer is read-only"`
24.7 節「読みだし専用バッファ」p.401 を参照してください。

`cyclic-function-indirection`
: `"Symbol's chain of function indirections contains a loop"`
8.2.4 節「シンボル関数間接参照」p.105 を参照してください。

`end-of-buffer`
: `"End of buffer"`
27.2 節「移動」p.458 を参照してください。

`end-of-file`
: "End of file during parsing"
これは`file-error`ではありません。
16.3 節「入力関数」p.230 を参照してください。

`file-error`
: このエラーとその仲間 (subcategories) は、エラー文字列を持ちません。そのようになっているのは、エラー状態`file-error`がある場合に、エラーメッセージがデータ項目から単独で構築されるからです。
第 22 章「ファイル」p.349 を参照してください。

`file-locked`
: これは`file-error`です。
22.5 節「ファイルのロック」p.357 を参照してください。

`file-already-exists`
: これは`file-error`です。
22.4 節「ファイルへの書込み」p.356 を参照してください。

`file-supersession`
: これは`file-error`です。
24.6 節「変更時間の比較」p.399 を参照してください。

`invalid-function`
: "Invalid function"
8.2.3 節「リスト形式の分類」p.105 を参照してください。

`invalid-read-syntax`
: "Invalid read syntax"
16.3 節「入力関数」p.230 を参照してください。

`invalid-regexp`
: "Invalid regexp"
30.2 節「正規表現」p.525 を参照してください。

`no-catch`
: "No catch for tag"
9.5.1 節「明示的な非局所脱出: `catch` と `throw`」p.116 を参照してください。

`search-failed`
: "Search failed"
第 30 章「検索と照合」p.523 を参照してください。

`setting-constant`
: "Attempt to set a constant symbol"
シンボル`nil`と`t`の値は変更できません。
10.2 節「Variables that Never Change」p.128 を参照してください。

`undefined-color`
: "Undefined color"
26.19 節「色の名前」p.453 を参照してください。

`void-function`
: "Symbol's function definition is void"
11.8 節「関数セルの内容へのアクセス」p.158 を参照してください。

`void-variable`
: "Symbol's value as variable is void"
10.6 節「変数の値へのアクセス」p.134 を参照してください。

`wrong-number-of-arguments`
: "Wrong number of arguments"
8.2.3 節「リスト形式の分類」p.105 を参照してください。

`wrong-type-argument`
: "Wrong type argument"
2.5 節「型述語」p.36 を参照してください。

以下のエラータイプ (すべて `arith-error` の特殊なケースに分類される) は、特定のシステムにおいて数学関数を正しく使用しなかった場合に発生することがあります。

`domain-error`
: "Arithmetic domain error"
3.9 節「標準的な数学関数」p.52 を参照してください。

`overflow-error`
: "Arithmetic overflow error"
3.9 節「標準的な数学関数」p.52 を参照してください。

`range-error`
: "Arithmetic range error"
3.9 節「標準的な数学関数」p.52 を参照してください。

`singularity-error`
: "Arithmetic singularity error"
3.9 節「標準的な数学関数」p.52 を参照してください。

`underflow-error`
: "Arithmetic underflow error"
3.9 節「標準的な数学関数」p.52 を参照してください。

付録 D バッファ・ローカル変数

ここにあげるのはバッファごとに自動的にローカル変数として定義される汎用の Emacs 変数の一覧です。多くの Lisp パッケージでは内部で使用するためにローカル変数を定義しますが、ここではそれらの変数については列挙しません。

`abbrev-mode`
: 第 32 章「省略形とその展開」p.553 参照

`auto-fill-function`
: 29.13 節「自動的行詰め」p.499 参照

`buffer-auto-save-file-name`
: 23.2 節「自動セーブ」p.388 参照

`buffer-backed-up`
: 23.1 節「バックアップ・ファイル」p.383 参照

`buffer-display-table`
: 35.14 節「表示テーブル」p.618 参照

`buffer-file-format`
: 22.12 節「ファイル・フォーマットの変換」p.377 参照

`buffer-file-name`
: 24.4 節「バッファ・ファイル名」p.397 参照

`buffer-file-number`
: 24.4 節「バッファ・ファイル名」p.397 参照

`buffer-file-truename`
: 24.4 節「バッファ・ファイル名」p.397 参照

`buffer-file-type`
: 22.13 節「ファイルと MSDOS」p.380 参照

`buffer-invisibility-spec`
: 35.5 節「不可視テキスト (invisible text)」p.604 参照

`buffer-offer-save`
: 22.2 節「バッファを保存する」p.352 参照

`buffer-read-only`
: 24.7 節「読みだし専用バッファ」p.401 参照

`buffer-saved-size`
: 27.1 節「ポイント」p.457 参照

`buffer-undo-list`
: 29.9 節「取り消し」p.492 参照

`cache-long-line-scans`
: 27.2.4 節「テキスト行単位での移動」p.460 参照

`case-fold-search`
: 30.7 節「検索と大文字小文字の区別」p.538 参照

`ctl-arrow`
: 35.13 節「普通の表示方法」p.617 参照

`comment-column`
: *The GNU Emacs Manual* の “Comments” 節参照

`default-directory`
: 34.3 節「OS 環境」p.584 参照

`defun-prompt-regexp`
: 27.2.6 節「釣り合った式をまたがる移動」p.464 参照

`fill-column`
: 29.13 節「自動的行詰め」p.499 参照

`goal-column`
: *The GNU Emacs Manual* の “Moving Point” 節参照

`left-margin`
: 29.16 節「字下げ」p.503 参照

`local-abbrev-table`
: 第 32 章「省略形とその展開」p.553 参照

`local-write-file-hooks`
: 22.2 節「バッファを保存する」p.352 参照

`major-mode`
: 20.1.4 節「主モードの使い方をユーザに説明する」p.326 参照

`mark-active`
: 28.6 節「マーク」p.475 参照

`mark-ring`
: 28.6 節「マーク」p.475 参照

`minor-modes`
: 20.2 節「副モード」p.327 参照

`mode-line-buffer-identification`
: 20.3.2 節「モード行の中で使用されている変数」p.331 参照

`mode-line-format`
: 20.3.1 節「モード行の表示をコントロールするデータ構造」p.330 参照

`mode-line-modified`
: 20.3.2 節「モード行の中で使用されている変数」p.331 参照

`mode-line-process`
　　20.3.2 節「モード行の中で使用されている変数」p.331 参照

`mode-name`　20.3.2 節「モード行の中で使用されている変数」p.331 参照

`overwrite-mode`
　　29.4 節「テキストの挿入」p.482 参照

`paragraph-separate`
　　30.8 節「編集用の標準正規表現」p.539 参照

`paragraph-start`
　　30.8 節「編集用の標準正規表現」p.539 参照

`point-before-scroll`
　　マウス・コマンドとスクロール・バー・コマンドとの通信に使用されます。

`require-final-newline`
　　29.4 節「テキストの挿入」p.482 参照

`selective-display`
　　35.6 節「選択表示」p.605 参照

`selective-display-ellipses`
　　35.6 節「選択表示」p.605 参照

`tab-width`　35.13 節「普通の表示方法」p.617 参照

`truncate-lines`
　　35.3 節「切捨て表示」p.602 参照

`vc-mode`　20.3.2 節「モード行の中で使用されている変数」p.331 参照

付録 E 標準のキーマップ

以下のシンボルは、さまざまなキーマップの名前として使用されます。これらのシンボルのうちいくつかは Emacs が起動されたときから存在していますが、それ以外のものは、それぞれのモードが使用されたときにはじめてロードされます。この一覧はすべてのキーマップを網羅しているわけではありません。

これらのマップのほとんど全てはローカル・マップとして使用されます。実際、現在のところ存在しているモードの中では Vip モートと Terminal モードだけが常にグローバル・キーマップを変更します。

`Buffer-menu-mode-map`
: Buffer Menu モードで使用される密なキーマップ。

`c-mode-map`
: C モードで使用される疎なキーマップ。

`command-history-map`
: Command History モードで使用される密なキーマップ。

`ctl-x-4-map`
: 前置キー `C-x 4` のサブコマンドのための疎なキーマップ。

`ctl-x-5-map`
: 前置キー `C-x 5` のサブコマンドのための疎なキーマップ。

`ctl-x-map`
: `C-x` コマンドのための密なキーマップ。

`debugger-mode-map`
: Debugger モードで使用される密なキーマップ。

`dired-mode-map`
: `dired-mode` バッファのための密なキーマップ。

`edit-abbrevs-map`
: `edit-abbrevs` で使用される疎なキーマップ。

`edit-tab-stops-map`
: `edit-tab-stops` で使用される疎なキーマップ。

`electric-buffer-menu-mode-map`
: Electric Buffer Menu モードで使用される密なキーマップ。

`electric-history-map`
: Electric Command History モードで使用される密なキーマップ。

`emacs-lisp-mode-map`
: Emacs Lisp モードで使用される疎なキーマップ。

`facemenu-menu`
: Text Properties メニューを表示するキーマップ。

`facemenu-background-menu`
: Text Properties メニューの Background Color サブメニューを表示するキーマップ。

`facemenu-face-menu`
: Text Properties メニューの Face サブメニューを表示するキーマップ。

`facemenu-foreground-menu`
: Text Properties メニューの Foreground Color サブメニューを表示するキーマップ。

`facemenu-indentation-menu`
: Text Properties メニューの Indentation サブメニューを表示するキーマップ。

`facemenu-justification-menu`
: Text Properties メニューの Justification サブメニューを表示するキーマップ。

`facemenu-special-menu`
: Text Properties メニューの Special Props サブメニューを表示するキーマップ。

`function-key-map`
: キーパッドとファンクションキーを変換するためのキーマップ。
何も定義されていなければ、それは空の疎なキーマップを含みます。

`fundamental-mode-map`
: Fundamental モードのためのローカル・キーマップ。
それは空であり、変更されるべきではありません。

`Helper-help-map`
: ヘルプ・ユーティリティ・パッケージによって使用される密なキーマップ。
それは、その値セルとその関数セルのものと同じキーマップを持ちます。

`Info-edit-map`
: Info の `e` コマンドによって使用される疎なキーマップ。

`Info-mode-map`
: Info コマンドを含む疎なキーマップ。

`isearch-mode-map`
: 段階的検索でタイプすることのできる文字を定義するキーマップ。

`key-translation-map`
: キーの変換のためのキーマップ。これは `function-key-map` とは違い、普通のキー・バインドよりも優先して扱われます。

`lisp-interaction-mode-map`
: Lisp モードで使用される疎なキーマップ。

`lisp-mode-map`
: Lisp モードによって使用される疎なキーマップ。

`menu-bar-edit-menu`
: メニュー・バーの Edit メニューを表示するキーマップ。

`menu-bar-files-menu`
: メニュー・バーの File メニューを表示するキーマップ。

`menu-bar-help-menu`
: メニュー・バーの Help メニューを表示するキーマップ。

`menu-bar-search-menu`
: メニュー・バーの Search メニューを表示するキーマップ。

`menu-bar-tools-menu`
: メニュー・バーの Tools メニューを表示するキーマップ。

`mode-specific-map`
: `C-c` に続く文字のためのキーマップ。これはグローバル・キーマップ中にあることに注意してください。このマップは、実際にはモード固有ではありません。その名前は `C-c` 前置キーの主な使い道を説明する `C-h b` (`display-bindings`) でユーザにとって有益なものが選択されます。

`occur-mode-map`
: Occur モードで使用されるローカル・キーマップ。

`query-replace-map`
: `query-replace` と関連するコマンドでの応答で使用されるローカル・キーマップ。 `y-or-n-p` と `map-y-or-n-p` でも使用されます。このマップを使用する関数は前置キーをサポートしません。同時に一つのイベントしか見ないためです。

`text-mode-map`
: Text モードで使用される疎なキーマップ。

`view-mode-map`
: View モードで使用される密なキーマップ。

付録 F 標準のフック

ここにあげるのは、適切な時点であなたが用意した関数を Emacs から呼び出されるようにするためのフック変数の一覧です。

これらの変数のほとんどは、'-hook' で終る名前を持ちます。そのような名前をもつ変数は『正規フック (normal hook)』であり、 run-hooks を使用して実行します。そのようなフックの値は関数のリストです。フックに新しい関数を追加するには add-hook を使うことが推奨されます。フックの使用法についての詳細は、 20.4 節「フック」p.335 を参照してください。

'-function' で終る名前をもつ変数は、その値としてただ一つの関数を持ちます。一般に変数が正規フックでない場合には、関数に引数を渡す必要があるなどの特別の理由があります。(古いバージョンの Emacs では、これらの変数のいくつかは正規フックでないにもかかわらず、'-hook' で終る名前を持っていました。)

'-hooks' または '-functions' で終る名前をもつ変数は、その値として関数のリストを持ちますが、これらの関数は特殊な方法で呼び出されます (それらは引数を渡されるか、さもなければ、その値が使用されます)。

activate-mark-hook

after-change-function

after-change-functions

after-init-hook

after-insert-file-functions

after-make-frame-hook

auto-fill-function

auto-save-hook

before-change-function

before-change-functions

before-init-hook

before-make-frame-hook

blink-paren-function

c-mode-hook

calendar-load-hook

command-history-hook

comment-indent-function

deactivate-mark-hook

diary-display-hook

diary-hook

dired-mode-hook

disabled-command-hook

edit-picture-hook

electric-buffer-menu-mode-hook

electric-command-history-hook

electric-help-mode-hook

emacs-lisp-mode-hook

find-file-hooks

find-file-not-found-hooks

first-change-hook

fortran-comment-hook

fortran-mode-hook

ftp-setup-write-file-hooks

ftp-write-file-hook

indent-mim-hook

initial-calendar-window-hook

kill-buffer-query-functions

kill-emacs-query-functions

LaTeX-mode-hook

ledit-mode-hook

lisp-indent-function

lisp-interaction-mode-hook

lisp-mode-hook

list-diary-entries-hook

m2-mode-hook

mail-mode-hook

mail-setup-hook

mark-diary-entries-hook

medit-mode-hook

mh-compose-letter-hook

mh-folder-mode-hook

mh-letter-mode-hook

mim-mode-hook

minibuffer-setup-hook

minibuffer-exit-hook

news-mode-hook

news-reply-mode-hook

```
news-setup-hook
nongregorian-diary-listing-hook
nongregorian-diary-marking-hook
nroff-mode-hook
outline-mode-hook
plain-TeX-mode-hook
post-command-hook
pre-abbrev-expand-hook
pre-command-hook
print-diary-entries-hook
prolog-mode-hook
protect-innocence-hook
rmail-edit-mode-hook
rmail-mode-hook
rmail-summary-mode-hook
scheme-indent-hook
scheme-mode-hook
scribe-mode-hook
shell-mode-hook
shell-set-directory-error-hook
suspend-hook
suspend-resume-hook
temp-buffer-show-function
term-setup-hook
terminal-mode-hook
terminal-mode-break-hook
TeX-mode-hook
text-mode-hook
today-visible-calendar-hook
today-invisible-calendar-hook
vi-mode-hook
view-hook
window-setup-hook
write-contents-hooks
write-file-hooks
write-region-annotate-functions
```

付録 G Emacs 18 ニュース (の反対)

過去の世界に住むユーザたちへ、 Emacs バージョン 18 へのダウングレード情報を送ります。 Emacs 19 特有の機能が沢山なくなって、素晴らしくも単純になった世界をお楽しみください。

G.1 Lisp に残っていた古い機能

Emacs バージョン 18 でなくなったり変更された関数は以下のとおりです。

- 関数 `delete`、 `member`、 `indirect-function`、 `map-y-or-n-p`、 `invocation-name` は削除されました。
- 関数 `read` は、バッファから読み込むときにシンボルの終端文字をスキップするようになりました。たとえば、ポイント位置に続いて ‘`foo(bar)`’ が存在するバッファで `read` を使うと、 `foo` が返され、ポイントは開かっこの後に移動します。つまり、 ‘`(bar)`’ をそのままの形で正確に読むことはできません。

  この単純化のおかげで、入力ストリーム関数はオプション引数を取らずに済むようになりました。 Emacs 18 では、入力ストリームは引数を取らず必ず次の文字を返します。
- 関数 `documentation` は引数を一つだけ、関数 `documentation-property` は二つだけ取るようになりました。
- `random` はオプション引数 *n* を取らなくなりました。
- Lisp ライブラリがロードされたときにフックを実行する仕組みはなくなりました。変数 `after-load-alist` と関数 `eval-after-load` は削除されました。
- 関数 `autoload` はキーマップの自動ロードをサポートしなくなりました。
- ‘`;;;###autoload`’ の形式のコメントはまさしく単なるコメントとして扱われるようになりました。これらのコメントは見ためが美しいだけで、まったく何もしません。デフォルトで関数を自動ロードさせるように設定したいときは、ファイル ‘`loaddefs.h`’ を直接書き換えてください。エディタってのはそのためにあるものでしょ?
- `format` 関数で ‘`%S`’ は使えなくなりました。 `prin1-to-string` からはオプション引数 *noescap* が削除されました。 `print-level` も削除されました。

G.2 コンパイル機能の変更

- Emacs 18 ではインライン関数はありません。関数呼出しを実現するコードを必要以上に大きくしてしまうことが判明したからです。 (サイズが小さいのも Emacs 18 の新しい特徴の一つです)
- 特殊形式 `eval-when-compile` と `eval-and-compile` はなくなりました。 `compile-defun` コマンドもなくなりました。
- 同一ディレクトリに ‘`.elc`’ ファイルとそれより新しい ‘`.el`’ ファイルがあっても、 Lisp ファイルやライブラリのロード時にそれについての警告はしないようになりました。うるさいもんね。

- バイトコード関数を表す特別なデータ型はなくなりました。コンパイルされた関数は関数 `bytecode` を起動することによって動くようになりました。この関数はバイトコード・インタプリタを起動します。

G.3 浮動小数点数

Emacs 18 では浮動小数点数を扱う組込みの機能はなくなりました。代わりに浮動小数点数をエミュレートする使いやすい Lisp プログラムがあります。でも、これを使うにはプログラムを書かなきゃなりませんが。

なんにせよ、マクロや関数、述語などは浮動小数点数の存在を考える必要がなくなりました。

- 関数 `string-to-number`、述語 `floatp`、変数 `float-output-format` はなくなりました。
- 関数 `float`、 `truncate`、 `floor`、 `ceil`、 `round`、 `logb` はなくなりました。同様に `abs`、 `cos`、 `sin`、 `tan`、 `acos`、 `asin`、 `atan`、 `exp`、 `expt`、 `log10`、 `log`、 `sqrt` もなくなりました。
- `format` 関数は、浮動小数点数の印字を指示する '%e'、 '%f'、 '%g' を扱わなくなりました。`message` についても同様です。

G.4 基本編集関数の変更

- kill リングにテキストを格納するためのプリミティブである `kill-new` と `kill-append` はなくなりました。
- 変数 `interprogram-paste-function` と `interprogram-cut-function` は Emacs 18 では削除されました。

  それから、 Transient Mark モードがなくなったので `mark-active` と `deactivate-mark` は必要なくなりました。フック `activate-mark-hook` と `deactivate-mark-hook` も削除されました。
- `kill-region` 関数は読出し専用のバッファでは使用できなくなりました。 `compare-buffer-substrings` と `current-kill` 関数は削除されました。
- 変数 `overwrite-mode-binary` は削除されました。
- 関数 `move-to-column` は *column* 引数だけしか取らないようになりました。
- 検索関数は、検索成功時には `t` だけを返すようになりました。この変更は、 `search-forward`、 `search-backward`、 `word-search-forward`、 `word-search-backward`、 `re-search-forward`、 `re-search-backward` についてなされました。
- 正規表現での検索や照合で、 `match-beginning` と `match-end` を使用できる '`\(...\)`' ペアの数は 10 個までとなりました。さらに、 `save-match-data` がなくなったので、一致データの保存には明示的に `unwind-protect` を使用する必要があります。
- `translate-region` はなくなりました。

- 変数 before-change-function、 after-change-function、 first-change-hook はなくなりました。
- insert-abbrev-table-description の 2 番目の引数は必須になりました。

G.5 テキスト属性

テキスト属性はなくなりました。

G.6 ファイル関連の機能

ファイル関連の関数の多くが削除されたり単純化されました。変更内容の概略を以下に示します。

- 関数 file-accessible-directory-p、 file-truename、 make-directory、 delete-directory、 set-visited-file-modtime、 write-region、 write-contents-hooks、after-save-hook、 directory-abbrev-alist、 abbreviate-file-name、 set-default-file-modes、 default-file-modes、 unix-sync はなくなりました。
- read-file-name の "initial file name" 引数はなくなりました。
- それから、 file-attributes から返されるリストの 12 番目の要素は削除されました。
- directory-files は常にファイル名のリストをソートするようになりました。滅茶苦茶な順番に並んだファイル名を見せるのはユーザ・フレンドリーじゃないですから。
- 変数 write-contents-hooks、 local-write-file-hooks はなくなりました。

G.7 魔力をもつファイル名

魔法のファイル名はなくなりました。すまん。マジックポイントを全部使い切ってしまったのだ。

G.8 フレーム

Emacs 18 ではフレームは一つしかありません。フレーム関連の関数は全て削除されました。

G.9 X Window System 関連の機能

大艦巨砲主義を排し、 Emacs と X との関係を単純化することに成功しました。

- 関数 mouse-position と set-mouse-position、特殊形式 track-mouse はなくなりました。
- 同様に、関数 x-set-selection、 x-set-cut-buffer、 x-open-connection、 x-close-current-connection は Emacs Lisp 18 から全て削除されました。

- X サーバとスクリーンに関する情報を扱う関数群がばっさり削除され、すべてのサーバと平等につき合えるようになりました。削除された関数は、x-display-screens、x-server-version、x-server-vendor、x-display-pixel-height、x-display-mm-height、x-display-pixel-width、x-display-mm-width、x-display-backing-store、x-display-save-under、x-display-planes、x-display-visual-class、x-display-color-p、x-display-color-cells です。
- それから、変数 x-no-window-manager と関数 x-synchronize、x-get-resource が削除されました。
- x-display-color-p は、名前が x-color-display-p に変更されました。x-color-defined-p は廃止されました。
- x-popup-menu の 1 番目の引数としてキーマップを渡すことはできなくなりました。
- 関数 x-rebind-key と x-rebind-keys は削除されました。
- x-parse-geometry は廃止されました。

G.10 便利になったウィンドウ操作

Emacs 18 がある今となっては、Emacs 19 のウィンドウはいろいろな面で時代遅れとなりました。我々はウィンドウ操作について全面的な見直しを行ない、以下のような変更を施しました。

- 関数 window-at、window-minibuffer-p、set-window-dedicated-p、window-dedicated-p、coordinates-in-window-p、walk-windows、window-end を削除しました。
- 変数 pop-up-frames、pop-up-frame-function、other-window-scroll-buffer、display-buffer-function を削除しました。
- フレームは Emacs version 18 ではオブジェクトではなくなったので、関数 minibuffer-window の引数としてフレームを渡すことはできなくなりました。この関数はミニバッファが使用しているウィンドウを返します。
- フレームは一つしかないので、関数 next-window と previous-window は *all-frames* 引数を受け取らなくなりました。
- 関数 get-lru-window、get-largest-window、get-buffer-window、get-buffer-window も、調べるべきフレームは一つしかないので、オプション引数 *all-frames* を取らなくなりました。

G.11 表示機能

- オーバーレイやフェースはなくなりました。
- 後のバージョンでは現在の行番号の表示に使われているモード行構成要素 '%l' は削除されました。変数 line-number-mode と line-number-display-limit は削除されました。
- baud-rate は変数ではなく関数になりました。

- nil 一つだけを引数にして message を呼ぶことはできなくなりました。つまり、ミニバッファの内容が確実に表示されているようにすることはできなくなりました。
- 変数 temp-buffer-show-function の名前は temp-buffer-show-hook に変更されました。
- 関数 force-mode-line-update は削除されました。代わりに以下のような式を使用してください。

```
(set-buffer-modified-p (buffer-modified-p))
```

- 表示テーブルはなくなりました。 ASCII は本来あるべき姿で表示されるようになりました。

G.12 入力イベントの扱い

入力イベントについてもっとも大きな変更は、ファンクション・キーとマウスに関するイベントが削除されたことでしょう。入力イベントは文字だけになったのです。それに、文字は 0 から 127 の範囲のコードとメタ・ビットとで表現されるようになりました。これらの変更により、単純化が非常に大きく進みました。

- define-key や global-set-key、 read-key-sequence、 local-set-key のような関数は Emacs 19 では文字列またはベクタを受けつけるようになっていましたが、文字列のみを受けつけるようになりました。
- 説明文字列に関する関数 (single-key-description、 key-description など) もベクタは受けつけずに文字列だけを受けつけるようになりました。
- 関数 read-event、 event-start、 posn-window、 posn-point、 posn-col-row、 posn-timestamp、 scroll-bar-scale、 event-end は文字イベントには意味がないので削除されました。
- unread-command-events と last-event-frame 変数は削除されました。
- 関数 this-command-keys と recent-keys は常に文字列を返すようになりました。同様に、キーボード・マクロの定義にもベクタは使用できず、文字列のみしか使用できなくなりました。
- 文字イベントでは意味がないので、対話引数の指定子として ‘e’ は使用できなくなりました。
- Emacs 18 では、メタ文字を文字オブジェクトとして、文字列と同じく、 128 と ASCII コードとの和で表現するようになりました。

G.13 メニュー

- メニューをキーマップとして定義することはできなくなりました。それぞれの仕組みにはそれぞれの特性に合わせてプログラム・インタフェースを用意するのがよいシステム設計であり、それでこそ両方式の利点を活かせるのです。キーマップとメニューの統合による単純化には無理があると我々は考えています。

- Emacs 18 では、メニューはマウスでのみ使用できるようになりました。キーボードでのメニュー操作は非常に多くのユーザを大混乱におとしいれることになってしまいます。
- Emacs 18 ではメニュー・バーはなくなりました。メニュー・バー関連の全ての関数と変数は削除されました。

G.14 ミニバッファ関連の変更

- ミニバッファでのヒストリ機能はなくなりました。また、ミニバッファでの入力関数 `read-from-minibuffer` と `completing-read` のオプション引数 *hist* を削除しました。
- `read-from-minibuffer` などのミニバッファでの入力関数の *initial* 引数として、(*string* . *position*) の形式のコンス・セルを渡すことはできなくなりました。
- 関数 `read-no-blanks-input` の *initial* 引数は必須になりました

G.15 コマンド定義に関する新機能

- 対話引数の指定子として '`@`' は使用できなくなりました。
- Emacs 18 では、対話引数の指定のプロンプト文字列として、書式指定と同様な '`%`'- 指定子は使用できなくなりました。
- 属性 `enable-recursive-minibuffers` は意味を持たなくなりました。

G.16 入力からの読込みに関して削除された機能

- 関数 `set-input-mode` の 3 番目の引数 (*meta*) は削除されました。代わりに、変数 `meta-flag` が追加されました。この変数を `t` にセットすると Meta キーが有効になり、 `nil` なら無効になります。変数 `meta-flag` で指定できるのはこの 2 種類だけです。
- また、変数 `extra-keyboard-modifiers` も削除されました。
- 関数 `keyboard-translate`、変数 `num-input-keys` と `function-key-map` は削除されました。

G.17 構文テーブルに関して削除された機能

- 関数 `skip-syntax-forward`、 `skip-syntax-backward`、 `forward-comment` は削除されました。
- "接頭辞構文 (prefix syntax)" を表す構文フラグはなくなりました。また、もう一つのコメント形式を表すフラグもなくなりました。 Emacs 18 では、構文テーブルにはそれぞれ 1 種類ずつしかコメント形式を持てません。
- 変数 `words-include-escapes` はなくなりました。

G.18 ケース・テーブル

- ケース・テーブルは Emacs 18 ではなくなりました。そのため、関連する関数 set-standard-case-table、standard-case-table、current-case-table、set-case-table、set-case-syntax-pair は削除されました。

G.19 バッファを扱う機能

- バッファ関連の関数 buffer-modified-tick と generate-new-buffer-name は削除されました。
- buffer-disable-undo の名前は buffer-flush-undo に変更されました。わかりやすい名前になった、でしょ?
- Emacs 18 では関数 other-buffer は引数を一つしか取らないようになりました。
- 関数 rename-buffer の引数として、すでに存在しているバッファ名を渡すことはできなくなりました。
- ローカル変数 list-buffers-directory は削除されました。
- フック kill-buffer-hook はなくなりました。

G.20 ローカル変数について

- 関数 kill-all-local-variables は、常にカレント・バッファにローカルな変数全てを消し去るようになりました。
- void である変数をバッファローカルにした際に、その変数の値が nil になるようになりました。
- 変数のグローバルな値が void であることはなくなったので、関数 default-boundp は削除されました。
- 特殊形式 defconst と defvar は、変数がカレント・バッファにローカルなとき、変数のグローバルな値ではなくローカルな値をセットするようになりました。

G.21 サブプロセス関連の機能

- call-process と call-process-region はサブプロセスの終了ステータスを返さないようになりました。きっとうまく実行されますよ。信じてください。
- 非同期プロセス関連の標準機能は VMS の上では使用できなくなり、代わりに、VMS 向けに特別に設計された非同期プロセス関数群が追加されました。でも、VMS でしか使えない機能のドキュメントをわざわざ書くのも面倒なので、すいません。ルーク、ソースを使え。
- 関数 signal-process は削除されました。
- トランザクション待ち行列の機能は廃止されました。関連する関数 tq-create、tq-enqueue、tq-close は削除されました。

G.22 時刻、時間、遅延の扱い

- 関数 current-time、 current-time-zone、 run-at-time、 cancel-timer は削除されました。
- 関数 current-time-string はオプション引数を取らなくなりました。
- 関数 sit-for と sleep-for はミリ秒単位での時間指定を行なうためのオプション引数を取らないようになりました。また、 sit-for の三つ目のオプション引数 *nodisp* はなくなりました。
- accept-process-output 関数の二つ目と三つ目のオプション引数はなくなりました。引数はプロセス一つだけになりました。

G.23 Lisp デバッガからなくなった機能

- Emacs 18 では、起きたエラーの種類によってデバッガの起動を抑制することはできなくなりました。変数 debug-on-error の値が非 nil ならば、エラーの種類によらず必ずデバッガが立ち上がるようになりました。
- 変数 command-debug-status と関数 backtrace-frame は削除されました。

G.24 メモリ割り当てに関する変更

- 関数 memory-limit は削除されました。
- Emacs 18 では浮動小数点数は扱えないので、 garbage-collect の返すリストは浮動小数点数に関する情報を含まないようになりました。

G.25 フックに関する変更

- フック auto-save-hook、 pre-abbrev-expand-hook、 pre-command-hook、 post-command-hook は削除されました。
- 変数 revert-buffer-insert-file-contents-function は削除されました。
- また、関数 add-hook も削除され、フックの追加は手で行なわなくてはならなくなりました。確実な動作を望むのであれば、フック変数の設定には旧来の方法を取ってください。つまり、関数のリストではなく、関数一つだけを入れるようにしてください。まあ、強制するわけではありませんが。
- 変数 lisp-indent-hook の名前は lisp-indent-function に変更されました。
- 変数 auto-fill-function の名前は auto-fill-hook に変更されました。
- blink-paren-function の名前は blink-paren-hook に変更されました。
- 変数 temp-buffer-show-function の名前は temp-buffer-show-hook に変更されました。

索引

訳注: 原書の索引のエントリーもそのまま残してあります。

<

1

2

8

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

U

V

W

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

と

な

に

ぬ

ね

は

ひ

ふ

へ

ほ

ま

み

む

め

も

や

ゆ

よ

ら

り

る

れ

ろ

わ

簡略目次

目次