Lisp入門

Common Lisp / Scheme

第0.1版

中村勝則

2020年2月29日

免責事項

本書の内容は参考的資料であり，掲載したプログラムリストは全て試作品である．本書の使用に伴って発生した不利益，損害の一切の責任を筆者は負わない．

目 次

1 序章

Lisp の歴史は長く，高水準のプログラミング言語としては FORTRAN の次に古い．Lisp の名は “list processor” に由来しており，その名が示す通り Lisp は主たるデータ構造として**リスト（連結リスト）**を取り扱う．

連結リストは，数値，文字列をはじめとする様々な型のデータを要素として保持できるだけでなく，要素として連結リストを保持することもできる．すなわち，リストはデータ型を問わずに要素を保持することができる，非常に柔軟性の高いデータ構造である．リストの利便性の高さゆえに，多くのプログラミング言語がこれを扱うための機能を提供している．

Lisp は，リストを扱うための機能を実現した最も古い言語処理系であり，取り扱うデータとプログラムそれ自体の両方をリストとして記述する．これにより，データとして作成したリストをプログラムとみなして実行することが可能となり，他の言語処理システムと比べて，Lisp は動作の柔軟性において一線を画す．このことについては本書の中で具体的に示す．

Lisp の言語仕様として具体的に定められたものとしては **Common Lisp** と **Scheme** の 2 種類が広く採用されており，言語処理系の実装もどちらかに準拠したものが多い．本書の前半では Common Lisp を想定して Lisp について解説し，後半で Scheme についても触れる．

最近では，IT 産業における Lisp を用いた開発案件はあまり多くはないものの，基本的概念のシンプルさと処理系の動作の柔軟性，プログラム実行時の性能の高さゆえに，Lisp は AI （人工知能）関連分野で一定の地位を確立している．したがって，コンピュータサイエンスを学ぶ者は Lisp について学んでおくことが望ましい．

1.1 Common Lisp

Common Lisp は Lisp の規格の 1 つであり，最初の版の仕様は CLtL （Common Lisp the Language：文献 [2]）に記されている．本書の執筆時点での最新版は CLtL2 であり，書籍やカーネギーメロン大学のインターネットサイト（文献 [3]）で公開されている．また，初期の Common Lisp の仕様策定を行った中心的な組織である X3J13 に関わった Kent Pitman によるインターネットサイト（文献 [4]）も Common Lisp の言語仕様に関する情報源として重要である．

1.2 本書が取り扱う範囲

本書は初めて Lisp を学ぶ者に向けた入門書である．従って，取り扱う内容は入門に必要な範囲とする．すなわち，言語仕様の中心的かつ初歩的な部分に絞った内容とし，本書による学びを終えた後は，言語の仕様書などを独力で読みながら，プログラム作成を行うだけでなく，熟達者が記述した Lisp のソースコードを解読するなど，学びを続けていただきたい．

1.3 本書で用いる処理系

本書では Common Lisp の処理系として，基本的には **SBCL** （Steel Bank Common Lisp）を想定する．この処理系はフリーソフトウェアであり，公式インターネットサイト http://www.sbcl.org/ から入手できる．SBCL は Linux, macOS，Windows といった各種の OS 用のものが開発されて公開されている．

Lisp の設計思想は非常にシンプルであるが，Common Lisp の言語仕様はそれに反して非常に大きい．SBCL は Common Lisp の仕様を広範囲に実装した処理系であり，動作も非常に早い．

1.3.1 Lisp 処理系の起動と終了

SBCL を起動するには OS（オペレーティングシステム）のターミナルウィンドウから sbcl コマンドを投入する．起動すると処理系の起動メッセージが表示され，プロンプト「*」（アスタリスク）が表示される．

例. Windows のコマンドプロンプトウィンドウ（cmd.exe）から SBCL を起動した例

```
C:¥Users¥katsu>sbcl   Enter          ← SBCL の起動
This is SBCL 1.4.2, an implementation of ANSI Common Lisp.          ←起動時のメッセージ群
More information about SBCL is available at <http://www.sbcl.org/>.  ←が表示される
                    ⋮
                （途中省略）
                    ⋮
WARNING: the Windows port is fragile, particularly for multithreaded
code.  Unfortunately, the development team currently lacks the time
and resources this platform demands.
*                                ← SBCL のプロンプト（これに続いて Common Lisp の式を入力する）
```

これに続いて Common Lisp の式を入力して Enter を押すとそれが実行（評価）されて結果が表示される．

処理系を終了して OS に戻るには `(quit)` と入力して Enter を押す．

2 基礎事項

Lisp（Common Lisp）で扱うリストは ‘(···)’ で括ったデータ列である．

リストの例. `(a b c)` `(1 2 3)`

要素は空白で区切られる．

2.1 プログラムとしてのリスト

Lisp では，プログラムそれ自体をリストとして記述する．この場合，リストの先頭の要素に関数名や演算子といった計算処理を意味するものを置く．

例. 3 + 4 の計算

```
(+ 3 4)   Enter        ←加算の式を投入
→  7          ←計算結果
```

この例のように，(+ 数 1 数 2 … 数 n) というリストを処理系に投入すると，先頭要素の ‘+’ が加算を意味する**オペレータ**とみなされ，数 1 + 数 2 + … + 数 n の計算結果が得られる．そしてそれが表示される．

Lisp 以外の言語処理系では，処理を記述する**文**と計算結果の値を返す**式**からプログラムが構成される．これに対して Lisp には「文」というものがなく，プログラムはリストの形式で記述された「式」として扱われる．また，Lisp の処理系が行うことは,「投入された式を**評価**して，その値を返す」ことである．厳密に表現すると，Lisp は「プログラムを実行する」ものではなく,「式を評価する」ものである．

Lisp は投入された式を評価しようとするため，評価できないものを投入するとエラーが発生する．

例. 評価できないものを与える試み（SBCL での例）

```
* (1 2 3)   Enter       ← (1 2 3) というリストを投入
; in:  1 2              ←ここからエラメッセージ
;     (1 2 3)
;
; caught ERROR:
;   illegal function call
                    ⋮
                （途中省略）
                    ⋮
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.
restarts (invokable by number or by possibly-abbreviated name):
  0:  [ABORT] Exit debugger, returning to top level.

((LAMBDA ()))
   source:  (1 2 3)          ←ここまでエラメッセージ
0]          ←ここに ‘ABORT’ と入力して Enter すると復帰する
```

この例では，式として評価できないリスト（1 2 3）を処理系に与えており，リストの先頭要素の 1 が**オペレータ**（関数名や演算子など）として解釈できないことが原因となってエラーが発生している．

重要）Lisp 処理系の評価機構に与えるリストは，その先頭要素が計算処理を意味するオペレータで
なければならない．

2.1.1 式の再帰的な記述

式は再帰的に記述する（入れ子にする）ことができる．例えば (1+2)*(3+4) という計算は次のような式として記述できる．

例. (1+2)*(3+4) の式

```
(* (+ 1 2) (+ 3 4))  Enter        ←リストの式による表現
→  21                ←計算結果
```

この例では，リスト内部の式 (+ 1 2) と (+ 3 4) を評価して 3 と 7 を得た後，式全体を (* 3 7) として評価している．このように，式を評価する際は内側の式から評価する．

2.1.2 ファイルから式を読み込む方法

テキストファイルに記述された Common Lisp の式を読み込んで評価するには関数 load を使用する．

ファイルからの式の読み込み： `(load "ファイル名")`

この様な式を評価することで,「ファイル名」のファイルの内容を読み込み，Lisp の式として評価する．この方法は，後に説明する関数，マクロ，大域変数（スペシャル変数）などの定義をファイルから読み込む場合（**プログラムの読み込み**）に用いる．ファイル内に日本語などのマルチバイト文字が含まれる場合は，load にキーワード引数 :external-format で文字コードの体系を与える．

UTF-8 で読み込む例. `(load "ファイル名" :external-format :utf-8)`

文字コードの体系に関しては p.43 の表 8 に記載する．

参考）Lisp の式を記述したテキストファイルの拡張子は '.lisp' とすることが一般的である．

2.1.2.1 コメントの記述

Common Lisp では**コメント**の記述ができる．コメントとは式とは見なされない記述であり，特にファイル中に記述した Lisp の式に対する注釈を付ける場合に記述すると良い．

セミコロン「;」で始まる記述は，その行末までがコメントとなる．また，

```
#|  （複数行のコメント）  |#
```

のように括ると複数行に渡るコメントを記述することができる．

2.2 評価と値

値として基本的なものに**数値**，**文字列**などがある．文字列は二重引用符「"」で括ったものである．

文字列の例. "abcde" "日本国"

数値や文字列などはそれらが最終的な値であり，それらを Lisp 処理系に与えるとそのまま返される．

例. 数値，文字列を Lisp 処理系に与える試み

```
123  Enter        ←数値を与える
→  123            ←そのまま返される．

"Japan"  Enter    ←文字列を与える
→  "Japan"        ←そのまま返される．
```

2.2.1 quote

リストを式とみなさずにそのまま値とするには quote を用いる．これは与えられた記述を評価せずにそのまま返すものである．

書き方： (quote 記述)

例. リスト (a b c) を評価せずに，それ自体を値として返す

```
(quote (a b c))  Enter        ←(a b c) を式とみなさずそのまま値とする
→  (A B C)          ←それ自身が値となる
```

もっと簡単に (quote 記述) は**単引用符** 1 つで「’記述」とすることもできる．

例. リスト (a b c) を評価せずに，それ自体を値として返す（その 2）

```
’(a b c)  Enter        ←(a b c) を式とみなさずそのまま値とする
→  (A B C)          ←それ自身が値となる
```

この例では，評価結果のリストの要素のアルファベットが大文字に変えられている．Common Lisp では，基本的に**シンボル**（次に説明する）は大文字である．

2.3 シンボル

シンボルは変数やオペレータとなり得る特別なオブジェクトである．シンボルは英数字などの組み合わせで記述されるが文字列とは別のものである．

シンボルの例. a, x1, 59e, z-1, t_3, 2h, _5, -g, a.co.jp …

シンボルは数字から始まってもよい．また，ハイフンやアンダースコア，ピリオドを含めることができる．

シンボルをそのまま処理系に与えると，システムはそのシンボルが持つ値を評価しようとする．その際に当該シンボルに何も値が割り当てられていなければ評価ができずエラーとなる．

例. 未設定のシンボル A を Lisp 処理系に与える試み（SBCL での例）

```
* a  Enter        ←値が未設定のシンボルを投入
debugger invoked on a UNBOUND-VARIABLE in thread        ←ここからエラメッセージ
#<THREAD "main thread" RUNNING 10027900C3>:
   The variable A is unbound.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
   0:  [CONTINUE   ] Retry using A.
   1:  [USE-VALUE  ] Use specified value.
   2:  [STORE-VALUE] Set specified value and use it.
   3:  [ABORT      ] Exit debugger, returning to top level.

(SB-INT:SIMPLE-EVAL-IN-LEXENV A #<NULL-LEXENV>)        ←ここまでエラメッセージ
0]        ←ここに ‘ABORT’ と入力して Enter すると復帰する
```

シンボルを評価せずにそのシンボル自身を値とするには (quote シンボル) とする．

例. シンボルを評価しない記述

```
(quote a)  Enter        ←シンボルを返す記述
→  A          ←シンボルが返される

’a  Enter        ←シンボルを返す記述
→  A          ←シンボルが返される
```

長い名前のシンボルも使用できる．

例. 長い名前のシンボル

```
'Japan  Enter        ←長い名前のシンボル（その 1）
→  JAPAN             ←シンボルが返される

'b29  Enter          ←長い名前のシンボル（その 2）
→  B29               ←シンボルが返される
```

重要）シンボルと文字列は全く別のものである.

2.4 S 式

シンボルや各種の値，リストなど，Lisp 処理系が扱えるものを **S 式**と呼ぶ.

2.5 変数

他の言語と同様に，Lisp でも変数を使用することができる．変数はその使用に先立って宣言しておく.

2.5.1 defvar

変数の宣言に defvar を用いる方法がある [1] ．また，変数への値の設定には setq を使用する.

変数の宣言： `(defvar シンボル)`

値の設定： `(setq シンボル 値)`　あるいは，
`(setq シンボル 1 値 1 シンボル 2 値 2 … シンボル n 値 n)`

変数への値の割り当ては，シンボルへの値の割り当てによって実現 [2] する.

例. 変数の宣言と値の設定

```
(defvar a)  Enter        ←シンボル A を変数として使用することを宣言
→  A          ← A が変数となった（シンボル A が返される）

(setq a 123)  Enter      ←変数 A に値 123 を設定
→  123        ←値 123 が設定された（123 が返される）

(* 2 a)  Enter           ←変数 A を用いた式を評価
→  246        ←評価結果
```

defvar では 2 つ目の引数を与えて，変数の初期値を設定することができる.

初期値の設定： `(defvar 変数 初期値)`

注意）defvar による変数への初期値の設定は初回のみ有効である．すなわち，2 回目以降の宣言にける初期値の設定は無効である.

例. defvar による初期値の設定

```
(defvar a 10)  Enter     ←シンボル A を宣言し，初期値を 10 とする
→  A          ← A が変数となった（シンボル A が返される）

a  Enter                 ←値の確認
→  10         ← A の値は 10（初期値）

(defvar a 100)  Enter    ←再度変数 A を宣言し，初期値を 100 とする試み
→  A          ←新たに設定されたように見えるが…

a  Enter                 ←値を確認すると
→  10         ← A の値は初回に宣言したままである
```

[1] Common Lisp では，defvar による変数宣言がなくても，setq によるシンボルへの値の割り当てができる．ただしその場合，SBCL では警告メッセージが表示される.

[2] これは**大域変数**（スペシャル変数）を実現する方法である.

defvar で変数を宣言した後での値の再設定（変更）には setq を使用すること．

シンボルを変数として宣言した後でもシンボルとしての性質は変わらず，’A を評価すると当該シンボルである A が返される．

例. 先の例の続き

```
’a  Enter        ←シンボル A を評価しない形で（quote して）投入
→  A             ←シンボルとして返される
```

2.5.2 defparameter

変数は defparameter を用いて宣言することもできる．defparameter による宣言時には必ず値を与える．

変数の宣言： `(defparameter シンボル 値)`

defparameter による変数宣言では，再度の宣言においてその値が更新できるので，defvar に比べて扱いが簡便である．

2.6 関数

評価可能なリストの先頭要素はオペレータであり，次のような形式を取る．

評価可能なリスト： `( オペレータ 引数１ 引数２ … 引数ｎ )`

オペレータは引数 1〜引数 n を使用して計算を実行する．

Lisp の処理系には多くのオペレータが予め用意されている．オペレータは**関数**，**マクロ**，**特殊オペレータ**といった種類に分類される．ここでは関数について説明する．

2.6.1 リスト処理のための基本的な関数

複数の要素を持つリストは，先頭の要素と残りの要素から成るリストを **cons セル**（単に**セル**とも呼ぶ）で結合したものである．例えば (1 2 3) というリストは図 1 のように構成されている．

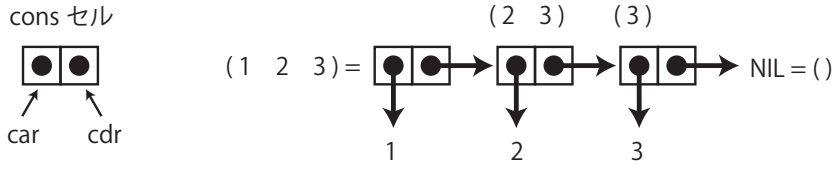


図 1: cons セルで構成されるリスト

cons セルはオブジェクトへのポインタを 2 つペアにしたもので，前の部分を car，後ろの部分を cdr と表記する．リストはセルの連結により構成され，セルの car が指し示すものがリストの要素である．また，末尾のセルの cdr は**空リスト** () を意味する NIL を指し示す．

新たなセルを作成して car と cdr にオブジェクトを割り当てるには関数 cons を使用する．

《cons セルの作成》
(cons car に割り当てるオブジェクト cdr に割り当てるオブジェクト)

例. cons によるリストの構成

```
(cons 3 ’()) Enter        ← car に 3 を，cdr に空リストを割り当てたセルを作成
→  (3)            ←結果

(cons 2 (cons 3 ’())) Enter     ← car に 2 を，cdr に上のリストを割り当てたセルを作成
→  (2 3)          ←結果

(cons 1 (cons 2 (cons 3 ’()))) Enter   ← car に 1 を，cdr に上のリストを割り当てたセルを作成
→  (1 2 3)        ←結果
```

このように，セルの連結で実現されたリストは，空リストの前方（左側）に要素を次々と追加する形で構成される．

car と cdr のオブジェクトからセルを作成するには**ドット対**を使用することもできる．

ドット対によるセルの作成： （car 部 . cdr 部）

注意）小数点のドットと区別するために，ドットの左右には空白を開けておくこと．

例. ドット対

```
'(3 . ()) Enter        ← car に 3 を，cdr に空リストを割り当てたセルを作成
→ (3)          ←結果

'(2 . (3 . ())) Enter      ← car に 2 を，cdr に上のリストを割り当てたセルを作成
→ (2 3)          ←結果

'(1 . (2 . (3 . ()))) Enter  ← car に 1 を，cdr に上のリストを割り当てたセルを作成
→ (1 2 3)          ←結果
```

式の先頭要素をオペレータと見なして評価することを防ぐために，式の入力時は quote している．

リストの car 部，cdr 部を得るには car，cdr 関数を用いる．

《car 部，cdr 部の取得》
(car リスト) ： 「リスト」の car 部を取得する．
(cdr リスト) ： 「リスト」の cdr 部を取得する．

例. car 部，cdr 部の取得

```
(car '(1 2 3)) Enter        ← (1 2 3) の car 部の取得
→ 1          ←結果

(cdr '(1 2 3)) Enter        ← (1 2 3) の cdr 部の取得
→ (2 3)          ←結果
```

注意）式の評価時は引数の部分から優先的に評価される．従って (1 2 3) を式ではなくデータとして扱うために quote している．

演習課題. '(a . b) は cons セル（p.6 の図 1 参照）で表現するとどのようになるか図示せよ．

演習課題. '(a . b) と '(a b) の違いを説明せよ．

演習課題. '(a b c . d) と等価なリストはどのようなものか答えよ．

演習課題. '(a (b (c d) e) f) を cons セルで表現するとどのようになるか図示せよ．

2.6.2 リストの作成

list 関数は引数に与えたものを要素とするリストを返す．

例. リストの作成

```
(list 1 2 3 4 5) Enter        ←引数に与えたものをリストにする
→ (1 2 3 4 5)          ←戻り値
```

2.6.3 要素の個数

リストの要素の個数を求めるには length 関数を使用する．

書き方： (length リスト)

「リスト」の要素の個数を返す．

例. リストの要素数を求める

```
(length '(a b c))  Enter  ←リスト (a b c) の要素数を求める
→  3          ←要素数は 3
```

2.6.4 要素の取得

nth を使用するとリストから要素を取得することができる.

書き方： (nth インデックス　リスト)

「リスト」の要素の内,「インデックス」で指定した位置のものを返す. **インデックス**は 0 から 要素数-1 までの整数値で指定する. 要素数以上のインデックスを指定すると戻り値は NIL となる. また負のインデックスを与えるとエラーが発生する.

例. リストの要素をインデックス指定で取得する

```
(nth 0 '(a b c))  Enter  ←インデックス位置 0 の要素を求める
→  A          ←戻り値

(nth 1 '(a b c))  Enter  ←インデックス位置 1 の要素を求める
→  B          ←戻り値

(nth 3 '(a b c))  Enter  ←長さ以上のインデックス位置を指定すると…
→  NIL          ← NIL となる
```

重要）nth は**アクセサ**である.

2.6.4.1 アクセサについて

アクセサとは，リストを始めとするデータ構造の部分を指し示すものであり，setf などを用いて値を直接的に設定することができる.（次の例参照）

例. アクセサを介して値を設定する

```
(defparameter s '(a b c d))  Enter  ←変数 s にリストを設定
→  S          ←変数 s が宣言された

s  Enter  ←値の確認
→  (A B C D)          ←変数 s の値

(setf (nth 1 s) 'x)  Enter  ←リストのインデックス 1 の位置に 'x を設定
→  X          ←設定された値

s  Enter  ←変数 s の値の確認
→  (A X C D)          ←対象部分が変更されている
```

先に説明した car，cdr もアクセサである.

例. アクセサを介して値を設定する：その 2（先の例の続き）

```
(setf (car s) 'w)  Enter  ←リストの car の位置に 'w を設定
→  W          ←設定された値

s  Enter  ←値の確認
→  (W X C D)          ←変数 s の値

(setf (cdr s) '(i n d o w s))  Enter  ←リストの cdr の位置に別のリストを設定
→  (I N D O W S)          ←設定された値

s  Enter  ←値の確認
→  (W I N D O W S)          ←変数 s の値
```

2.6.5　要素の探索

find 関数を使用すると，指定した要素がリストの中に含まれるかどうかを検査できる．

書き方：　(find 要素　リスト)

「リスト」の中に「要素」が存在すればその要素を返し，存在しなければ NIL を返す．

例. 要素の探索

```
(find 'b '(a b c))  Enter        ←要素 'b がリスト '(a b c) の中にあるか？
→  B          ←ある
```

要素を探索する関数には member もある．この関数は見つけた要素以下のリストを返す．

例. member 関数による探索

```
(member 'b '(a b c d e))  Enter  ←要素 'b をリスト '(a b c d e) の中から探索
→  (B C D E)          ←見つけた箇所以降のリスト
```

探索対象が見つからなければ NIL を返す．

find，member ともに，要素検出の判定には eql を使用する，そのため，探索対象の要素が文字列やリストの場合は equal などで検出する必要がある．検出判定の関数を明に与えるには，キーワード引数 :test を与える．(次の例参照)

例. 文字列の要素を探索

```
(find "b" '(a "b" c d e) :test #'equal)  Enter  ←文字列要素の探索
→  "b"          ←要素 "b" が存在する

(member "b" '(a "b" c d e) :test #'equal)  Enter  ←文字列要素の探索
→  ("b" C D E)          ←要素 "b" 以降のリスト
```

キーワード引数 :test で判定用の関数 #'equal を与えている．

参考）関数名の接頭辞 #' は「関数そのもの」を意味する**特殊オペレータ**である．

2.6.6　リストの連結

append を用いるとリストを連結することができる．

書き方：　(append リスト 1　リスト 2　…　リスト n)

引数に与えたリストを連結したものを返す．

例. リストの連結

```
(append '(a b c) '(d e f))  Enter    ←リストの連結
→  (A B C D E F)          ←連結結果

(append '(a b) '(c d) '(e f))  Enter    ←リストの連結
→  (A B C D E F)          ←連結結果
```

2.6.7　関数の定義

新たな関数を定義するには defun を用いる．

《関数の定義》

(defun　関数名　(仮引数 1　仮引数 2　…　仮引数 n)　定義内容)

仮引数 1〜仮引数 n に受け取った値を用いた計算処理を「定義内容」として記述する．これにより「関数名」のオペレータが新たに定義される．

例. 引数に与えた数を 2 倍する関数 dbl の定義

```
(defun dbl (n) (* 2 n))  Enter        ← dbl(n) = 2n を定義
→  DBL          ←関数 DBL が定義された（関数名のシンボルが戻り値）

(dbl 3)  Enter        ← dbl(3) を評価
→  6          ←評価結果
```

演習課題. 2 次元のベクトルの成分からなるリスト（x の値　y の値）を引数に取り，そのノルム $\sqrt{x^2+y^2}$ を返す関数 nrm を定義せよ．n の平方根を求める関数 (sqrt n) を使用して良い．

関数は引数を取って計算処理を行うことができるが，上に述べた形による関数定義では引数の個数が固定される．すなわち，評価の際には defun の定義で記述した引数と同じ個数の引数を与えなければならない．従って，先に例示した関数 dbl の評価において (dbl) や (dbl 3 4) といった式を処理系に与えるとエラーが発生する．

例. 引数が多い場合（先の例の続き）

```
(dbl 3 4)  Enter        ←多く引数を与えると…
debugger invoked on a SB-INT:SIMPLE-PROGRAM-ERROR in thread  ←エラーが発生する
#<THREAD "main thread" RUNNING {10010B0523}>:
  invalid number of arguments:  2
                  :
          (途中省略)
                  :
(DBL 3 4) [external]
    source:  (SB-INT:NAMED-LAMBDA DBL
                 (N)
               (BLOCK DBL (* 2 N)))
0]          ←ここに 'ABORT' と入力して Enter すると復帰する
```

引数の個数を固定しない柔軟な形式の関数を定義するには，defun で関数を定義する際の引数の並びに，以下に説明する各種の**キーワード**を用いる．

2.6.7.1　省略可能な引数：オプションパラメータ

省略可能な引数を取る関数を定義するには &optional キーワードを使用する．

> **《オプションパラメータの書き方》**
>
> **(defun 関数名**
> **(仮引数 1　仮引数 2　…　&optional　(仮引数 o1　暗黙値 1)　(仮引数 o2　暗黙値 2)…)**
> **定義内容)**
>
> 「仮引数 o1」,「仮引数 o2」，…　は，関数の評価時に省略可能な引数である．省略した場合は「暗黙値 1」,「暗黙値 2」，…　がそれぞれ設定される．暗黙値の記述を省略すると暗黙値は NIL となる．

例. オプションパラメータを持つ関数 mag の定義

```
(defun mag (n &optional (m 2)) (* n m))  Enter     ←省略可能な引数 m を取る関数
→  MAG          ←関数 MAG が定義された

(mag 3)  Enter     ←オプションパラメータを省略
→  6          ←暗黙知を用いた計算結果

(mag 3 4)  Enter     ←オプションパラメータを与えて評価
→  12          ←与えた値を用いた計算結果
```

この例は，与えた 2 つの数の積を求める関数 mag を定義するものであるが，2 つ目の引数はオプションパラメータであり省略可能（暗黙値は 2）である．

2.6.7.2　キーワード引数：キーパラメータ

関数に与える引数は単なる値の羅列であるが，**キーワード**を添える形にすると，引数の意味がわかりやすくなる．これを実現するには &key を使用する．

《キーワードパラメータの書き方》

(defun 関数名
　(仮引数 1　仮引数 2　…　&key　(仮引数 k1　暗黙値 1)　(仮引数 k2　暗黙値 2)…)
　　定義内容)

「仮引数 k1」,「仮引数 k2」，…　は，関数の評価時に与える**キーワード**となり，当該関数内では仮引数の名前（変数名）として扱われる．またこれらの引数は評価時に省略することが可能で，省略した場合は「暗黙値 1」，「暗黙値 2」，…　がそれぞれ設定される．暗黙値の記述を省略すると暗黙値は NIL となる．

このように定義された関数を評価する場合は

:引数 k1　値 1　:引数 k2　値 2　…

という形で引数名の前にコロン「:」を付け，その後ろに値を添える．

例. キーパラメータ sei，mei を持つ関数 fullname の定義

```
(defun fullname (&key (sei "伊藤") (mei "博文"))
    (princ "氏名：") (princ sei) (princ " ") (princ mei)     ←処理内容
    (list sei mei)     ←戻り値
)
```

実行結果.

```
(fullname :mei "太郎")  Enter     ←キーパラメータ mei のみ与える
→　氏名：伊藤 太郎          ← princ による出力
→　("伊藤" "太郎")          ←戻り値

(fullname :sei "佐藤")  Enter     ←キーパラメータ sei のみ与える
→　氏名：佐藤 博文          ← princ による出力
→　("佐藤" "博文")          ←戻り値

(fullname :mei "英機" :sei "東条")  Enter     ←キーパラメータの順序は任意
→　氏名：東条 英機          ← princ による出力
→　("東条" "英機")          ←戻り値
```

関数の評価時に与えるキーワード引数の順序は任意で良い

2.6.7.3　任意の個数の引数：レストパラメータ

任意の個数の引数を取る関数を定義するには　&rest キーワード（**レストパラメータ**）を使用する．

《レストパラメータの書き方》

(defun 関数名
　(仮引数 1　仮引数 2　…　仮引数 n　&rest　仮引数 r)
　　定義内容)

「仮引数 1」～「仮引数 n」に続く引数を「仮引数 r」にリストの形で受け取る．評価時に引数が「仮引数 n」までしか与えられない場合は「仮引数 r」は NIL となる．

例. レストパラメータを持つ関数 sumall の定義[3]

```
(defun sumall (n &rest r)
    (let ((s n))              ←局所変数 s を宣言し，n の値を設定している
        (dolist (m r s)        ← r の要素を全て足し合わせるループ
            (setq s (+ s m))       r の要素を 1 つずつ m に取り出しながら
        )                          処理を行い，s を戻り値として返す.
    )
)
```

実行結果.

(sumall 0) Enter ←引数を 1 つだけ与える
→ 0 ←戻り値
(sumall 0 1 2) Enter ←引数を 3 つ与える
→ 3 ←戻り値

let に関しては後の「2.7.1 逐次実行」(p.14) で，dotimes に関しては「2.7.3 繰り返し」(p.19) で説明する.

2.6.7.4 関数内の局所変数：補助パラメータ

関数内で使用する**局所変数**を定義するには &aux キーワード（**補助パラメータ**）を使用する.

《補助パラメータの書き方》

(defun 関数名
(仮引数 1 仮引数 2 … &aux (変数 a1 暗黙値 1) (変数 a2 暗黙値 2)…)
定義内容)

「変数 a1」,「変数 a2」, … は，関数の評価時に与える引数を受け取るためのものではなく，関数内部で使用する局所変数を宣言するものである．暗黙値の記述を省略すると暗黙値は NIL となる.

例. 補助パラメータを持つ関数 auxtest の定義

```
(defun auxtest (&aux (x 3))
    (princ "x=") (princ x)            ←局所変数の値を表示
)
```

実行結果.

(defparameter x 5) Enter ←大域変数 x に 5 を設定
→ X

(auxtest) Enter ←関数 auxtest を評価
→ x=3 ← princ による出力（局所変数）
→ 3 ←戻り値（局所変数）

x Enter ←大域変数の値を確認
→ 5 ←最初に設定した値のまま（局所変数とは区別されている）

[3] この関数は cons と eval を使用するともっと簡潔に書くことができる.

2.6.8 lambda

評価可能なリストは，その第 1 要素が関数名などの**オペレータ**となっている．これまでの解説においても，各種の関数名のシンボルをリストの第 1 要素に記述して式を評価する例を挙げてきたが，オペレータの記述の代わりに lambda を記述することもできる．

《**lambda**》

(lambda (仮引数の列)　定義内容)

「仮引数の列」に記述した仮引数を使用した「定義内容」の計算を表す．

defun では計算の定義を関数名のシンボルに与えるが，lambda は関数名を持たない「計算の定義そのもの」であると言うことができる．

先に例示した，与えられた数を 2 倍する関数 dbl について再度考える．

2 倍する関数 dbl の定義：　`(defun dbl (n) (* 2 n))`

この関数を評価するには次のような式を処理系に投入する．

3 を 2 倍する式：　`(dbl 3)`

これと同じ計算を行う式を lambda を用いて記述すると次のようになる．

lambda による記述：　`((lambda (n) (* 2 n)) 3)`

これは先の式 `(dbl 3)` の dbl の部分を `(lambda (n) (* 2 n))` に置き換えたものと見ることができる．実際にこの式を処理系に投入する例を示す．

例. lambda を用いた式の評価

```
((lambda (n) (* 2 n)) 3)  Enter      ←式の評価
→  6                      ←評価結果
```

関数 dbl の評価と同じ結果が得られる．

lambda の記述は defun による関数定義の記述と共通点が多い．特に，仮引数の記述の部分から定義内容の記述の部分までは同じである．

lambda は**無名関数**などと呼ばれることもあり，計算処理の定義を明に実体として扱うものである．lambda は A. チャーチが考案した**ラムダ計算**（文献 [7]）に由来するものであり，Lisp はこれに基づいて計算を実行する処理系であると見ることができる．

2.7 制御構造

2.7.1 逐次実行

羅列した式を順番に評価するには `let` を使用する.

《let による逐次実行》

(let ((変数 1　初期値 1)　(変数 2　初期値 2) … (変数 n　初期値 n))
式 1
式 2
⋮
式 m)

式 1〜式 m を順番に評価する．変数 1〜変数 n は let の式の内部で有効な**局所変数**であり，初期値を与えることができる．最後に評価した 式 m の値が let の式の評価結果となる.

例. let による逐次実行（評価）

```
(let ((x 0))  Enter        ← let の記述の開始．局所変数 x に初期値 0 を与えている
      (print x)  Enter     ←最初に評価する式：x の値を表示
      (setq x (+ x 1))  Enter    ← 2 番目に評価する式：x の値を 1 増やす
      (print x))  Enter    ←最後に評価する式：x の値を表示
0       ←最初の式の print による表示
1       ←最後の式の print による表示
1       ← let の評価結果の値
```

print 関数は引数に与えた値を標準出力（ターミナルウィンドウ）に出力するものである.（print 関数の戻り値は出力した内容）

この例の中で使用されている変数 x は let の内部で局所的である．従って let の外部で宣言した変数 x とは名前が同じであっても別のものとして扱われる.

例. 局所変数であることの確認

```
(defparameter x 1000)  Enter      ← let の外部で変数 x を宣言して 1000 を設定（大域変数）
X          ←変数 X が宣言された
(let ((x 0)) (print x) (setq x (+ x 1)) (print x))  Enter  ←先の例と同じ let の式
0       ←最初の式の print による表示
1       ←最後の式の print による表示
1       ← let の評価結果の値
x  Enter       ← let の外部で宣言した変数 x の値を確認
1000       ← let 内部の x の変化とは無関係である
```

参考）let 以外にも，複数の式を逐次評価するたに prog1，prog2，progn，block が存在する.

2.7.2　条件分岐

2.7.2.1　if

if を用いることで条件を判定し，評価する式を選択することができる．

> **《if による条件判定》**
> **(if　条件式　真の場合に評価する式　偽の場合に評価する式)**
> 「条件式」が成立する場合は「真の場合に評価する式」を，成立しない場合は「偽の場合に評価する式」を評価する．

例. if による評価の選択

```
(if (> 10 3) (print "10 > 3")  Enter          ← if の記述の開始
             (print "10 < 3 is false"))  Enter     ← if の記述の終了
"10 > 3"     ← 条件成立：1 つめの print 関数の出力
"10 > 3"     ← if の戻り値

(if (< 10 3) (print "10 > 3")  Enter          ← if の記述の開始
             (print "10 < 3 is false"))  Enter     ← if の記述の終了
"10 < 3 is false"     ← 条件成立せず：2 つめの print 関数の出力
"10 < 3 is false"     ← if の戻り値
```

この例で使用しているオペレータ ‘>’，‘<’ は数値の大きさを比較するものである．

評価する対象の式を更に if による式にすること（if の入れ子）で複雑な条件判定が可能となる．たくさんのの条件毎に判定する式を選択するには，後に説明する cond を使用すると良い．

2.7.2.2　条件式とその戻り値

条件判定に使用する条件式は真か偽を意味する値を返す．Common Lisp における「真」はシンボル T である．T はシステムで予約された特殊なシンボルであり，変数のためのシンボルとして使用することはできない．

例. 「真」を意味する T

```
(> 10 3)  Enter          ← 10 > 3 の検査
→  T          ←「真」を意味する T が返される．

T  Enter          ← シンボル T そのものを評価すると…
→  T          ←やはり「真」を意味する T
```

Common Lisp では空リスト NIL（あるいは’()）で「偽」を表す．

例. 「偽」を意味する NIL

```
(< 10 3)  Enter          ← 10 < 3 の検査
→  NIL          ←「偽」を意味する NIL が返される．
```

重要）Common Lisp における条件判定では，NIL 以外の値は全て「真」とみなされる．

例. NIL 以外は「真」

```
(if 3.14 "true" "false")  Enter        ←数値も「真」とみなす
→  "true"     ←条件成立
(if "文字列" "true" "false")  Enter        ←文字列も「真」とみなす
→  "true"     ←条件成立
(if '(a b c) "true" "false")  Enter        ←リストも「真」とみなす
→  "true"     ←条件成立
```

2.7.2.3　論理演算

条件式は and，or で結合でき，not で反転できる.（表 1 参照）

表 1: 条件式の論理演算

式	解説
(and 式 1　式 2　…　式 n)	式 1　式 2　…　式 n が全て T の時 T を返す.
(or 式 1　式 2　…　式 n)	式 1　式 2　…　式 n の内少なくとも 1 つが T の時 T を返す.
(not 式)	式 の真偽を反転したものを返す.

2.7.2.4　リスト／ cons ／アトムの判定

要素を持った（空リストではない）リストは cons セルの連結で実現されている．オブジェクトがセルかどうかを判定する関数に consp がある.

書き方：　(consp オブジェクト)

「オブジェクト」がセルの場合は T，それ以外の場合は NIL を返す.

例. オブジェクトがセルかどうかの判定

```
(consp '(1 2 3))  Enter        ←要素を持つリストを検査
→  T          ←セルである

(consp 'a)  Enter          ←シンボル A を検査
→  NIL          ←セルではない

(consp 1)  Enter          ←数値 1 を検査
→  NIL          ←セルではない
```

セル以外のオブジェクト（シンボル，数値，文字列など）を**アトム**という．オブジェクトがアトムかどうかを判定する関数に atom がある.

書き方：　(atom オブジェクト)

「オブジェクト」がアトムの場合は T，それ以外の場合は NIL を返す.

例. オブジェクトがアトムかどうかの判定

```
(atom '(1 2 3))  Enter        ←要素を持つリストを検査
→  NIL          ←アトムではない

(atom 'a)  Enter          ←シンボル A を検査
→  T          ←アトムである

(consp 1)  Enter          ←数値 1 を検査
→  T          ←アトムである
```

重要）空リスト NIL は「要素を持たないリスト」という意味でリストである．また同時に NIL はアトムでもある．これは，NIL がセルではない ことによる.（p.6 の図 1 参照）

オブジェクトがリストかどうかを判定する関数に listp がある.

書き方：　(listp オブジェクト)

「オブジェクト」がリストの場合は T，それ以外の場合は NIL を返す.

例. リスト／cons／アトムの判定

```
(listp '(1 2 3))  Enter        ←要素を持つリストを検査
→  T          ←リストである

(listp '())  Enter          ←空リストはリストか？
→  T          ←リストである

(atom '())  Enter           ←空リストはアトムか？
→  T          ←アトムである

(consp '())  Enter          ←空リストはセルか？
→  NIL          ←セルではない
```

2.7.2.5　2 つのオブジェクトが同一か／等値かの判定

Common Lisp には 2 つのオブジェクトを比較する関数 eql と equal がある．これら関数は，比較対象の 2 つのオブジェクトを引数に取る．

関数 eql は引数に与えられたオブジェクトが同一のものである場合は T を，それ以外の場合は NIL を返す．

例. eql による比較

```
(defparameter v1 '(a b c))  Enter      ←変数 V1 にリストを設定
→  V1          ←変数 V1 が宣言された
(defparameter v2 '(a b c))  Enter      ←変数 V2 に上と同じ内容のリストを設定
→  V2          ←変数 V2 が宣言された

(eql v1 v2)  Enter          ←上記の変数 V1，V2 を比較
→  NIL          ←変数 V1，V2 が指し示すものは 同一のものではない
```

この結果は見る者を困惑させるかも知れない．2 つの変数 V1，V2 が同じ内容のリストであるにもかかわらず，eql の評価結果が NIL となっている．この例の 2 つの変数が別々の実体を指し示している様子を図 2 に示す．

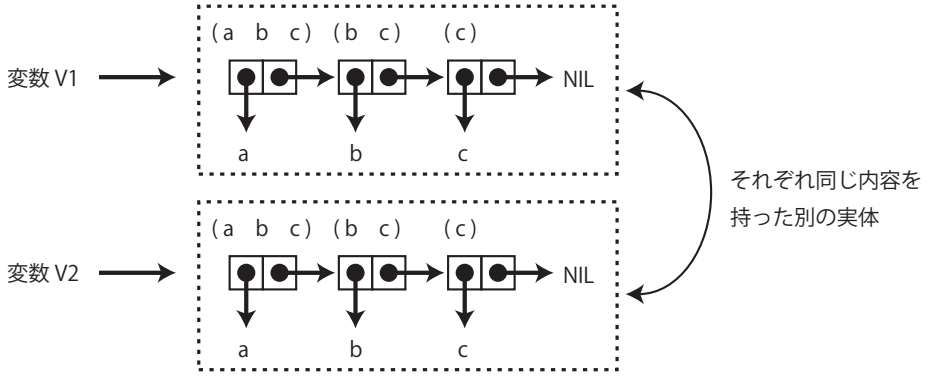


図 2: 内容が同じでも別々の実体

リストを構成する cons セルはそれ自体が記憶資源上の実体であり，個々に区別される．従って，(setq v2 v1) によって V1 と V2 が同じ 1 つの実体を指し示す形（図 3）にすると，(eql v1 v2) は T となる．

例. eql による比較：その 2（先の例の続き）

```
(setq v2 v1)  Enter         ←変数 V2 が変数 V1 と同じものを指し示す形に設定
→  (A B C)          ←変数に設定したもの

(eql v1 v2)  Enter          ←変数 V1，V2 を比較
→  T          ←変数 V1，V2 が指し示しているものは同一のものである
```

この例の setq で 2 つの変数 V1，V2 は 1 つの同じものを指し示す．(図 3)

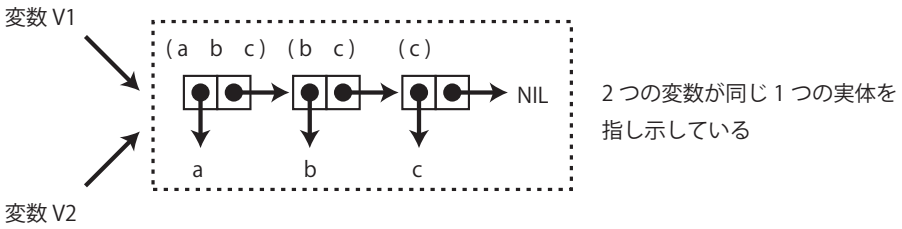


図 3: 2 つの変数 V1，V2 が同一のものを指し示している状態

関数 eql は同じシンボルの比較では T を返す．また，数値同士の比較においては，型と値が同じもの同士の場合は T を返す．同じ内容の文字列同士の比較においては NIL を返す．

参考）eql よりも更に原始的な関数に eq がある．

オブジェクトの内容が同じかどうかを判定するには equal 関数を使用する．

例. equal 関数を用いたオブジェクト内容の比較

```
(setq v1 '(a b c))  Enter     ←変数 V1 にリスト (a b c) を設定
→  (A B C)          ←設定された内容
(setq v2 '(a b c))  Enter     ←変数 V2 にリスト (a b c) を設定：上のものとは別の実体
→  (A B C)          ←設定された内容：変数 V1 のものとは別のオブジェクト

(equal v1 v2)  Enter          ←内容を比較
→  T          ←変数 V1，V2 が指し示すオブジェクトの内容は同じである
```

equal 関数は，実体として異なるオブジェクト同士であってもその内容が同じであれば T を返す.（もちろん，値が同じアトム同士の場合も T を返す）

参考）数値の比較にはイコール「=」が使用できる．

例. イコールによる数値の比較

```
(= 3 3)  Enter     ←整数同士の比較
→  T          ←等しい
(= 3.14 3.14)  Enter     ←浮動小数点数同士の比較
→  T          ←等しい
(= 4 5)  Enter     ←異なる値同士は…
→  NIL          ←等しくない
```

2.7.2.6 cond

複数の条件式毎に評価する式を選択するために cond を使用することができる．

《cond による評価対象の選択》

(cond (条件式 1　式 1)
(条件式 2　式 2)
⋮
(条件式 n　式 n))

条件式 1〜条件式 n を順に評価し，T となる条件式に対応する式を評価して戻り値とする．最初に T となる条件式に対応する式を評価する．

例. cond による多分岐

```
(defparameter x 2)  Enter      ←変数に値を設定
→  X                ← X に 2 が設定された

(cond ((= x 1) (print "x=1"))  Enter      ← cond の記述の開始
      ((= x 2) (print "x=2"))  Enter
      (t (print "many")))      Enter      ← cond の記述の終了

"x=2"               ← print 関数による出力
"x=2"               ← cond の式の戻り値
```

2.7.2.7 case

case を使用すると，値を個別に識別することで評価する式を選択することができる.

《case による評価対象の選択》

(case 値
((値並び 1)　式 1)
((値並び 2)　式 2)
⋮
((値並び n)　式 n)
(otherwise　式 x))

値並び 1〜値並び n は値の集合であり,「値」がどの並びに含まれるかを判定して，該当する式を評価して case の戻り値とする.「値」が含まれるかの検査は 値並び 1〜値並び n の順に行い，最初に該当する値並びのみが対象となる.「値」がどの値並びにも含まれない場合は「式 x」を評価して戻り値とする.

例. case による多分岐

```
(defparameter x 4)  Enter      ←変数に値を設定
→  X                ← X に 4 が設定された

(case x  Enter      ← case の記述の開始
      ((1 2) (print "x=1 or 2"))  Enter      ← x が 1 か 2 の場合の評価
      ((3 4 5) (print "x=3 or 4 or 5"))  Enter      ← x が 3 か 4 か 5 の場合の評価
      (otherwise (print "other")))  Enter      ← x がどれにも含まれない場合の評価

"x=3 or 4 or 5"          ← print 関数による出力
"x=3 or 4 or 5"          ← case の式の戻り値
```

2.7.3 繰り返し

一連の式を繰り返し実行する方法について説明する.

2.7.3.1 loop

繰り返し実行のための最も簡単な方法が loop の使用である.

《loop による繰り返し》

(loop　式 1　式 2　…　式 n)

式 1，式 2，…，式 n の連続した評価を繰り返す．loop が評価する式の中で **(return 戻り値)** を評価すると繰り返しを終了し loop は「戻り値」を返す.

注意）return による終了がなければ無限の繰り返しとなる.

例. loop による評価の繰り返し

```
(defparameter x 0)  Enter      ←変数に値を設定
→  X          ← X に 0 が設定された

(loop (if (> x 3) (return "end"))  Enter      ← loop の記述の開始
      (print x)
      (setq x (+ x 1)))  Enter      ← loop の記述の終了

0       ← print による出力（繰り返しの初回）
1
2
3       ← print による出力（繰り返しの最終回）
"end"  ← loop の戻り値
```

この例は X の値を 0 から順番に 1 ずつ増やしながら表示するものである．X が 3 より大きな値になった時点で繰り返しを終了する．

この例では return の式は loop の最初の引数の中に含まれているが，return の式は目的に応じて任意の評価位置に配置することができる．(次の例参照)

例. loop による評価の繰り返し：その 2

```
(defparameter x 0)  Enter      ←変数に値を設定
→  X          ← X に 0 が設定された

(loop (print x)  Enter      ← loop の記述の開始
      (if (> x 3) (return "end"))
      (setq x (+ x 1)))  Enter      ← loop の記述の終了

0       ← print による出力（繰り返しの初回）
1
2
3
4       ← print による出力（繰り返しの最終回）
"end"  ← loop の戻り値
```

これは先の例と似ているが，return が loop の 2 番目の引数に含まれているので実行結果が少し異なる．

2.7.3.2 dotimes

指定した回数だけ評価を繰り返すために dotimes がある．

《dotimes による繰り返し》

(dotimes (カウント変数 繰り返し回数 戻り値) 式 1 式 2 … 式 n)

式 1，式 2，…，式 n の連続した評価を「繰り返し回数」だけ繰り返す．「カウント変数」は dotimes の式の中で局所的であり，評価中の回数を保持している．「カウント変数」は 0 から「繰り返し回数」から 1 を引いた値まで変化する．繰り返し処理が終われば dotimes は「戻り値」を返す．

例. dotimes による繰り返し

```
(dotimes (x 4 "end")  Enter      ← dotimes の記述の開始
         (print x)
         (princ ":")
         (princ (* x x)))  Enter      ← dotimes の記述の終了
0 :0        ←初回の処理
1 :1
2 :4
3 :9        ←最終回の処理
"end"       ← dotimes の戻り値
```

この例ではカウント変数は x で繰り返しの度に 0 ～ 3 まで変化する．出力用の関数として print と princ が使用されているが，princ は改行処理をしない．

2.7.3.3 dolist

与えたリストから要素を 1 つずつ取り出しながら繰り返し処理をするには dolist が使える．

《dolist による繰り返し》

(dolist (変数 リスト 戻り値) 式 1 式 2 … 式 n)

「リスト」から要素を 1 つずつ取り出しながら，それに対応する形で 式 1，式 2，…，式 n の連続した評価を行う．繰り返しの各回において「リスト」から取り出された要素は「変数」に与えられる．この「変数」は dolist の式の中で局所的である．繰り返し処理が終われば dolist は「戻り値」を返す．

例. dolist による繰り返し

```
(dolist (x '(a b c) "end")  Enter      ← dolist の記述の開始
        (print '>)
        (princ x))  Enter      ← dolist の記述の終了
> A       ←初回の処理
> B
> C       ←最終回の処理
"end"       ← dolist の戻り値
```

この例では与えたリスト '(a b c) の各要素に対応する形で繰り返し処理を行っている．

2.7.3.4 do

与えた終了条件を満たすまで評価を繰り返すには do を使用する．

《do による繰り返し》

(do ((変数 1 初期値 1) (変数 2 初期値 2) … (変数 n 初期値 n))
(終了条件 戻り値)
式 1
式 2
⋮
式 m)

式 1～式 m の評価を繰り返す．変数 1～変数 n は do の式の内部で有効な**局所変数**であり，初期値を与えることができる．「終了条件」は 式 1～式 m の評価の前に毎回検査し，これを満たした時点で繰り返しを終了して「戻り値」を返す．

例. do による繰り返し

```
(do ((x 0)) ((> x 3) "end")  Enter      ← do の記述の開始
          (print x)
          (setq x (+ x 1)))  Enter      ← do の記述の終了
0       ←初回の処理
1
2
3       ←最終回の処理
"end"       ← do の戻り値
```

2.8 関数の再帰的定義

Lisp では，再帰的な関数の定義ができる．例えば非負の整数 n の階乗 $n!$ は次のように再帰的な定義ができる．

$$n! = \begin{cases} n = 0 & \rightarrow & 1 \\ n > 0 & \rightarrow & n(n-1)! \end{cases}$$

これに基づいて階乗を計算する関数 fct を次のように定義することができる．

《階乗を求める関数 fct》

```
(defun fct (n)
   (cond ((= n 0) 1)
         (t (* n (fct (- n 1))))))
```

例. fct の評価

```
(fct 5)  Enter      ← 5!
→  120          ←評価結果
(fct 40)  Enter     ← 40!
→  815915283247897734345611269596115894272000000000          ←評価結果
```

演習課題. 階乗を求める関数を，再帰的定義ではなく繰り返し dotimes を用いた形で定義せよ．

2.8.1 再帰的定義の問題点

関数の再帰的定義は記述が簡潔であるが，しばしば計算量が大きくなるので注意が必要である．例えばフィボナッチ数 F_n は次のように再帰的に定義される．

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

この定義をそのまま Common Lisp で記述すると次のようになる．

```
(defun fib (n)
   (cond ((= n 0) 0)
         ((= n 1) 1)
         (t (+ (fib (- n 1)) (fib (- n 2))))))
```

この関数 fib はフィボナッチ数を求めるものである．この関数を繰り返し評価してフィボナッチ数列を表示してみる．

例. フィボナッチ数の算出

```
(dotimes (n 40)  Enter      ← 繰り返しの記述の開始
    (princ (fib n))  Enter
    (princ " "))  Enter      ← 繰り返しの記述の終了

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
NIL          ← dotimes の戻り値
```

これは $F_0 \sim F_{39}$ を表示するものであるが，出力が得られるまでに時間がかかる．これは関数 fib の再帰的呼び出しの回数が急激に増加することによる．すなわち，再帰的定義によるフィボナッチ数 F_n の呼び出しを展開すると図 4 のようになる．

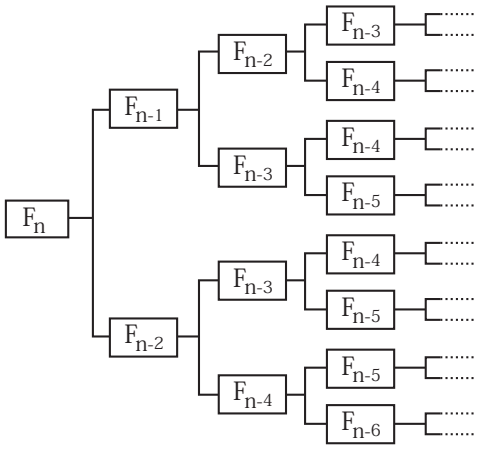


図 4: F_n の呼び出しを展開する様子

しかし，フィボナッチ数 F_n はその定義から F_{n-1} と F_{n-2} を用いて繰り返し処理によって次々と生成することができる.（図 5）

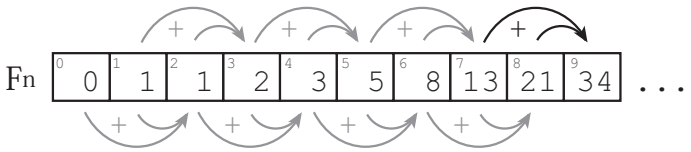


図 5: F_n を次々と生成する方法

この考えに沿って dotimes による繰り返しでプログラムを構成することを考える．図 5 のようにフィボナッチ数を次々と生成してリストに追加してゆく方法を取ると次のような関数 fib2 を定義することができる．

```
(defun fib2 (n)
   (let ((r '(0)))
      (dotimes (m n r)
         (if (= m 0) (setq r (cons 1 r))
            (setq r (cons (+ (car r) (cadr r)) r))))))
```

この関数はフィボナッチ数列（逆順）を生成する．

例. フィボナッチ数列（逆順）の生成

```
(fib2 39)  Enter      ← F0〜F39 の列（逆順）を生成
→  (63245986 39088169 24157817 14930352 9227465 5702887 3524578 2178309 1346269
    832040 514229 317811 196418 121393 75025 46368 28657 17711 10946 6765 4181
    2584 1597 987 610 377 233 144 89 55 34 21 13 8 5 3 2 1 1 0)
```

この評価は先の関数 fib よりも遥かに早い.

2.9 数値

2.9.1 整数

整数は任意の桁の値が扱える．例えば p.22 の階乗関数 fct は大きな値を返すことができる.

例. 大きな整数値

```
(fct 200)  Enter      ← 200!
→  788657867364790503552363213932185062295135977687173263294742533244359449963403342
   920304284011984623904177212138919638830257642790242637105061926624952829931113462
   857270763317237396988943922445621451664240254033291864131227428294853277524242407
   573903240321257405579568660226031904170324062351700858796178922222789623703897
   37472000000000000000000000000000000000000000000000000000
```

2.9.2 浮動小数点数

Common Lisp では浮動小数点数を扱うことができる.

浮動小数点数の例. 1.0　3.14

また,「仮数部 e 指数部」の**指数表現**も可能である.

指数表現の例. $6.02 \times 10^{23} \rightarrow$ `6.02e23`　$9.1 \times 10^{-31} \rightarrow$ `9.1e-31`

Common Lisp の浮動小数点数の精度は暗黙では 単精度 である．**倍精度**の浮動小数点数を扱う場合は ‘d’ による指数表現を用いる.

例. 精度の異なる浮動小数点数

```
(/ 1.0 3.0)  Enter          ← 1 ÷ 3 の演算
→  0.33333334               ←単精度の浮動小数点数
(/ 1.0d0 3.0d0)  Enter      ←倍精度による 1 ÷ 3 の演算
→  0.3333333333333333d0     ←倍精度の浮動小数点数
```

敢えて単精度の指数表現をする場合は ‘s’ を用いる.

例. 単精度の指数表現

```
3.0s0  Enter          ←単精度の指数表現
→  3.0                ←単精度の浮動小数点数
```

参考）処理系によっては**長精度**，**短精度**の浮動小数点数数が扱える．長精度の浮動小数点数は ‘L’ による指数表現で，短精度の浮動小数点数は ‘s’ による指数表現で記述する．SBCL では ‘L’ による指数表現は倍精度，‘s’ による指数表現は単精度と同じ扱いである.

浮動小数点数の小数点以下の**丸め**には round を使用する.

例. 小数点以下の丸め

```
(round 1.4)  Enter          ← 1.4 を丸めて整数にする
→  1                ←丸めの結果
→  0.39999998      ←元の数との差

(round 1.5)  Enter          ← 1.5 を丸めて整数にする
→  2                ←丸めの結果
→  -0.5             ←元の数との差
```

この例からわかるように，round は多値を返す．

参考）小数点を単純に切り捨てて整数に変換するには truncate を使用する．また，小数点の切り捨て方に応じて floor，ceiling などもある．

2.9.3 分数

Common Lisp では数としての分数が扱える．分数は分子と分母（共に整数）をスラッシュ '/' で繋いだ形で表現する

例. 分数の扱い

```
1/3   Enter          ← 1/3 を評価
→  1/3          ←値として評価された

100/200   Enter          ← 100/200 を評価
→  1/2          ←約分される

(/ 1 3)   Enter          ←整数同士の除算を評価
→  1/3          ←分数となる

(+ 1/3 1/2)   Enter          ←分数同士の加算
→  5/6          ←戻り値は分数
```

numerator で分数から**分子**を，denominator で**分母**を取得することができる．

例. 分母，分子の取得

```
(numerator 17/23)   Enter          ←分子の取得
→  17          ←分子

(denominator 17/23)   Enter          ←分母の取得
→  23          ←分母
```

分数を浮動小数点数に変換する，あるいはその逆を行うための関数がある．

分数から浮動小数点数への変換： `(float 分数)`
浮動小数点数から分数への変換： `(rational 浮動小数点数)`

例. 分数から浮動小数点数への変換

```
(float 2/3)   Enter          ← 2/3 の近似値を求める
→  0.6666667                ←近似値

(float 2/3 1.0d0)   Enter          ← 2/3 の近似値を倍精度で求める
→  0.6666666666666666d0          ←倍精度による近似値
```

float 関数は引数に与えられた値を浮動小数点数に変換した値を返す．また第 2 引数に値を与えると，その値と同じ型（精度）の値を返す．上の例で float の第 2 引数に 1.0d0 という値を与えているが，この値そのものにはあまり意味はなく，その型が戻り値に適用される．float 関数の第 1 引数に与えるものは分数だけでなく浮動小数点数などを与えることもできる．これは型（精度）の変換に利用できる．

浮動小数点数を分数に変換する例を次に示す.

例. 浮動小数点数から分数への変換

```
(rational 3.141592653589793d0)  Enter      ←与えた値に最も近い分数を求める
→  884279719003555/281474976710656          ←近い分数
```

このような変換処理においては若干の問題が起こる.(次の例参照)

例. 浮動小数点数の精度の問題

```
(float 1/10 1.0d0)  Enter      ← 1/10 を倍精度浮動小数点数に変換する
→  0.1d0          ←正確に変換できたように見えるが…

(rational 0.1d0)  Enter       ←逆の変換では…
→  3602879701896397/36028797018963968          ←このような分数となる
```

演習課題. 上の例の様な分数となる理由を説明せよ.

浮動小数点数の精度を配慮した形で分数に変換するには rationalize を使用する.

例. 浮動小数点数から分数への変換：倍精度浮動小数点数の精度への配慮

```
(rationalize 0.1d0)  Enter      ←分数に変換
→  1/10          ←変換結果
```

2.9.4 複素数

Common Lisp では次のような形式の複素数が扱える.

書き方： `#c(実部 虚部)`

例. 複素数

```
(* #c(0 1) #c(0 1))  Enter      ← i × i の計算
→  -1          ←計算結果

(sqrt -2)  Enter          ← √−2 の計算
→  #C(0.0 1.4142135)          ←計算結果 0.0 + 1.4142135 i
```

複素数は `(complex 実部 虚部)` という式で得ることができる.

例. 複素数を得る

```
(complex 1 2)  Enter          ←複素数を作る
→  #C(1 2)          ←複素数となる
```

複素数の**実部**をとりだすには realpart を，**虚部**を取り出すには imagpart を使用する.

例. 実部，虚部の取り出し

```
(realpart #c(2 3))  Enter      ←実部の取り出し
→  2          ←実部

(imagpart #c(2 3))  Enter      ←虚部の取り出し
→  3          ←虚部
```

2.9.5 乱数

関数 random で乱数を生成することができる.

書き方： `(random 上限値)`

「上限値」には正の数値を与える．この関数は 0 以上,「上限値」未満の乱数を生成する．

例. 乱数の生成

```
(random 100)  Enter        ← 0 以上 100 未満の整数の乱数（1 回目）
→  92          ←得られた乱数

(random 100)  Enter        ← 0 以上 100 未満の整数の乱数（2 回目）
→  44          ←得られた乱数

(random 1.0d0)  Enter      ← 0 以上 1.0 未満の倍精度浮動小数点数の乱数（1 回目）
→  0.9377354749449307d0          ←得られた乱数

random 1.0d0)  Enter       ← 0 以上 1.0 未満の倍精度浮動小数点数の乱数（2 回目）
→  0.442067749366045d0           ←得られた乱数
```

2.9.5.1　乱数生成の再現性について

random 関数が生成する乱数の系列には一般化できる規則性はないが，その系列は確定的である．すなわち，処理系を起動した直後から生成する乱数列のパターンは決まっている．これを**疑似乱数**という．

例. 確定的な系列の乱数列（SBCL での例）

```
(random 100)  Enter        ←整数の乱数（1 回目）
→  92          ←得られた乱数
(random 100)  Enter        ←整数の乱数（2 回目）
→  44          ←得られた乱数
(random 100)  Enter        ←整数の乱数（3 回目）
→  95          ←得られた乱数
(random 100)  Enter        ←整数の乱数（4 回目）
→  5         ←得られた乱数
(quit)  Enter        ←ここで Lisp を終了して OS に戻る…
        （Lisp 処理系を再度起動）
(random 100)  Enter        ←整数の乱数（1 回目）
→  92          ←得られた乱数
(random 100)  Enter        ←整数の乱数（2 回目）
→  44          ←得られた乱数
(random 100)  Enter        ←整数の乱数（3 回目）
→  95          ←得られた乱数
(random 100)  Enter        ←整数の乱数（4 回目）
→  5         ←得られた乱数
```

この例からわかるように乱数列のパターンは 92, 44, 95, 5, … と確定的である．

参考）疑似乱数に関する考察

疑似乱数の利用は乱数表（乱数列を記載したデータ）の利用に似ている．乱数表の値を順番に取り出すことにより，得られる系列が予測可能なものとなり，秘匿性が求められる情報処理を実現する場合などにおいて脆弱性の原因となり得る [4]．しかし，統計処理システムのテストにおいて再現性が求められる局面においては，確定的な乱数列が必要とされる場合もあり，疑似乱数には利用価値がある．

[4] 例えば，暗号化に関する処理に疑似乱数を使用するとクラッキングされるリスクが高まるといったことなどが挙げられる．

2.9.5.2 random state

random 関数が生成する乱数の系列は random state と呼ばれるオブジェクトによって決定される．Common Lisp では random 関数が使用する random state はスペシャル変数 *random-state* が保持している．これとは別の random state を作成して random 関数に与えることで，生成する乱数列の予測を困難なものにすることができる．random state を作成するには make-random-state 関数を使用する．

random state の作成： `((make-random-state t))`

random state オブジェクトを返す．make-random-state 関数に与える引数に関しては文献 [3],[4] を参照のこと．

作成した random state オブジェクトは random 関数の第 2 引数に与える．

例. random state の応用

```
(defparameter *r* (make-random-state t))  Enter  ← random state オブジェクトの作成
→  *R*          ←変数 *r* に random state オブジェクトが得られた
(random 100 *r*)  Enter        ←整数の乱数（1 回目）
→  97           ←得られた乱数
(random 100 *r*)  Enter        ←整数の乱数（2 回目）
→  30           ←得られた乱数
(random 100 *r*)  Enter        ←整数の乱数（3 回目）
→  79           ←得られた乱数
(random 100 *r*)  Enter        ←整数の乱数（4 回目）
→  63           ←得られた乱数
(quit)  Enter        ←ここで Lisp を終了して OS に戻る…
        ⋮
（Lisp 処理系を再度起動）
        ⋮
(defparameter *r* (make-random-state t))  Enter  ← random state オブジェクトの作成
→  *R*
(random 100 *r*)  Enter        ←整数の乱数（1 回目）
→  54           ←得られた乱数
(random 100 *r*)  Enter        ←整数の乱数（2 回目）
→  3            ←得られた乱数
(random 100 *r*)  Enter        ←整数の乱数（3 回目）
→  23           ←得られた乱数
(random 100 *r*)  Enter        ←整数の乱数（4 回目）
→  83           ←得られた乱数
```

このように，生成する乱数系列のパターンが異なるものとなる．

2.9.6 数値計算，数値の判定のための関数

数値に関する関数を表 2〜6 に示す．

表 2: 各種の判定

関数	解説	関数	解説	関数	解説
(integerp n)	n は整数	(floatp n)	n は浮動小数点数	(complexp n)	n は複素数
(numberp n)	n は数値	(plusp n)	$n > 0$	(minusp n)	$n < 0$
(zerop n)	$n = 0$	(oddp n)	n は奇数	(evenp n)	n は偶数

表 3: 基本的な演算（算術など）

関数	解説	関数	解説
(+ v1 v2 ⋯ vn)	$v_1 + v_2 + \cdots + v_n$	(- v1 v2 ⋯ vn)	$v_1 - v_2 - \cdots - v_n$
(* v1 v2 ⋯ vn)	$v_1 \times v_2 \times \cdots \times v_n$	(/ v1 v2 ⋯ vn)	$v_1/v_2/\cdots/v_n$
(1+ v)	$v + 1$	(1- v)	$v - 1$
(gcd v1 v2 ⋯ vn)	$v_1, v_2, \cdots, v_n$ の最大公約数	(lcm v1 v2 ⋯ vn)	$v_1, v_2, \cdots, v_n$ の最小公倍数
(rem v1 v2)	v_1 を v_2 で割った際の余り	(mod v1 v2)	v_1 の法 v_2 における剰余
(conjugate c)	c の**共役複素数**		

表 4: 値の比較など

関数	解説	関数	解説
(= v1 v2 ⋯ vn)	$v_1 = v_2 = \cdots = v_n$	(/= v1 v2 ⋯ vn)	$v_1, v_2, \cdots, v_n$ が全て異なる値
(< v1 v2 ⋯ vn)	$v_1 < v_2 < \cdots < v_n$	(> v1 v2 ⋯ vn)	$v_1 > v_2 > \cdots > v_n$
(<= v1 v2 ⋯ vn)	$v_1 \leq v_2 \leq \cdots \leq v_n$	(>= v1 v2 ⋯ vn)	$v_1 \geq v_2 \geq \cdots \geq v_n$

表 5: その他，数学関数

関数	解説	関数	解説
(exp x)	e^x	(expt x y)	x^y
(log x)	$\log_e x$	(log x n)	$\log_n x$
(sqrt x)	$\sqrt{x}$	(isqrt x)	整数値 x の平方根（整数値）
(abs x)	x の絶対値（複素数も可）	(signum x)	x 符号 *
(sin x)	$\sin(x)$	(cos x)	$\cos(x)$
(tan x)	$\tan(x)$	(asin x)	$\sin^{-1}(x)$
(acos x)	$\cos^{-1}(x)$	(atan x)	$\tan^{-1}(x)$
(sinh x)	$\sinh(x)$	(cosh x)	$\cosh(x)$
(tanh x)	$\tanh(x)$	(asinh x)	$\sinh^{-1}(x)$
(acosh x)	$\cosh^{-1}(x)$	(atanh x)	$\tanh^{-1}(x)$

* 複素数の場合は長さ 1 の方向ベクトル

表 6: 定数

定数	解説	定数	解説
most-positive-fixnum	fixnum の正の最大値	most-negative-fixnum	fixnum の負の最小値
pi	円周率 π		

* ここに挙げたもの以外にも，各種の数値型の制限値に関する定数が多数定義されている．most-⋯，least-⋯ で始まるシンボルに関して文献 [3], [4] を参照のこと．

3　実際のプログラミングに必要な事柄

3.1　配列

リストは柔軟なデータ構造であり，リストを入れ子にすることで多次元配列のようなデータ集合を表現することもできる．ただし，**インデックス**（要素の格納位置）を直接指定する形のアクセスにおいてはリストは処理速度の面で不利である．Common Lisp は多次元の配列を扱うための機能を提供しており，サイズの大きなデータ配列を高速に処理することができる．

3.1.1　配列の生成

配列オブジェクトは make-array で生成することができる．

配列の生成：　(make-array　(要素数 1　要素数 2　…　要素数 n))

make-array は各次元の要素数が「要素数 1」「要素数 2」…「要素数 n」の配列を作成して返す．

例. 配列の生成

```
(make-array '(2 3))  Enter          ← 2 × 3 のサイズの 2 次元配列を生成
→  #2A((0 0 0) (0 0 0))             ← 2 次元配列

(make-array '(1 2 3))  Enter        ← 1 × 2 × 3 のサイズの 3 次元配列を生成
→  #3A(((0 0 0) (0 0 0)))           ← 2 次元配列

(make-array 5)  Enter               ←要素数 5 の 1 次元配列を生成
→  #(0 0 0 0 0)                     ← 1 次元配列
```

配列オブジェクトは '#' から始まる．1 次元の配列（ベクトル）を生成する場合は make-array の引数には要素数の整数値を 1 つ与えるだけで良い．配列の要素の初期値は不定である．(SBCL では 0 が初期値）要素は括弧 '(…)' で括られるがリストではない．

多次元の配列は 1 次元の列を入れ子にする形で表現される．この際，最上位の列が第 1 の次元，それらの要素（次の階層）が第 2 の次元という形で構成される．

重要）配列はアトムである．

例. 配列がアトムであることの確認

```
(atom (make-array '(1 2 3)))  Enter   ← 3 次元配列がアトムかどうか検査
→  T                          ←アトムである
```

3.1.1.1　ベクトルの生成

特に 1 次元配列を**ベクトル**と呼び，生成には vector を用いる．この際，引数に要素の初期値を与えることができる．

書き方：　(vector　要素 1　要素 2　…　要素 n)

vector は 要素 1　要素 2　…　要素 n を初期値とする 1 次元配列を生成する．

例. vector による 1 次元配列の作成

```
(vector 1 2 3 4 5)  Enter           ←初期値を与えて 1 次元配列を生成
→  #(1 2 3 4 5)                     ← 1 次元配列
```

3.1.2　配列の要素へのアクセス

配列の要素にアクセスするには aref を用いる．

配列の要素へのアクセス：　(aref　配列　インデックス 1　インデックス 2　…　インデックス n)

「インデックス 1」は第 1 の次元のインデックス,「インデック 2」は第 2 の次元のインデックス，…「インデック n」は第 n の次元のインデックスを意味する．(aref はアクセサである）

例. ベクトルの要素へのアクセス

```
(defparameter vct (vector 1 2 3 4 5))  Enter  ←変数 vct に 1 次元配列を割り当てる
→  VCT
(aref vct 0)  Enter        ←インデックス位置が 0 の要素（先頭要素）にアクセス
→  1          ←要素の値
(aref vct 1)  Enter        ←インデックス位置が 1 の要素にアクセス
→  2          ←要素の値
(aref vct 4)  Enter        ←インデックス位置が 4 の要素にアクセス
→  5          ←要素の値
```

既存の配列の要素に値を設定するには setf を用いる[5]．

例. 要素への値の設定（先の例の続き）

```
(setf (aref vct 1) 20)  Enter        ←配列 vct のインデックス位置 1 の要素に 20 を設定
→  20          ←設定された値
vct  Enter        ←配列の内容を確認
→  #(1 20 3 4 5)          ←インデックス位置 1 の要素の値が変更されている
```

次に，多次元の配列の場合について例示する．

例. 多次元配列の要素へのアクセス

```
(defparameter ar (make-array '(2 2)))  Enter  ←変数 ar に 2 次元配列を割り当てる
→  AR
(setf (aref ar 0 1) 10)  Enter     ←配列 ar の「インデックス位置 0 の 1」の要素に 10 を設定
→  10          ←要素の値
ar  Enter        ←配列の内容を確認
→  #2A((0 10) (0 0))          ←該当位置の要素の値が 10 に設定されている
```

特に 2 次元の配列は使用する頻度が高いので「行」と「列」という形の解釈を図 6 に示す．

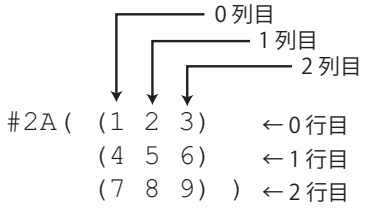


図 6: 2 次元の配列を「行」「列」で解釈する

3.1.3 配列のサイズの取得

配列のサイズを取得するには array-dimensions を使用する．

書き方： (array-dimensions 配列)

「配列」の各次元のサイズがリストとして得られる．

例. 配列のサイズの取得（先の例の続き）

```
(array-dimensions ar)  Enter        ←配列 ar のサイズを取得
→  (2 2)          ←2 × 2 のサイズの 2 次元配列である
```

[5] **setf** はデータ構造の内部を編集する形で値を設定する場合などに使用する．これを用いて変数に値を設定することもできる．

3.1.4 要素の型の指定

配列の生成時に要素の型を指定することができる.

要素の型指定： (make-array 要素サイズの指定 :element-type 型名)

make-array に**キーワード引数** :element-type と「型名」のシンボルを与える.

例. 型を指定した配列の生成

```
(make-array 5 :element-type 'double-float) Enter ←長さ 5 の倍精度浮動小数点数の配列を生成
→ #(0.0d0 0.0d0 0.0d0 0.0d0 0.0d0)          ←得られた配列

(make-array 5 :element-type 'integer) Enter      ←長さ 5 の整数の配列を生成
→ #(0 0 0 0 0)          ←得られた配列
```

「型名」のシンボルに関しては，基本的なものを「5.2 型の階層」(p.60) で紹介する.

3.2 文字と文字列

Common Lisp の文字列は**文字**（character 型）の要素から成る配列である.

例. 文字列を配列と見て要素を取り出す試み

```
(aref "abc" 1) Enter ←アスキー文字列 "abc" のインデックス位置 1 の要素を取り出す
→ #¥b          ←得られた要素：アスキー文字

(aref "記号処理" 2) Enter ← UNICODE 文字列 "記号処理" のインデックス位置 2 の要素を取り出す
→ #¥U51E6          ←得られた要素：UNICODE 文字「処」
```

この例でわかるように，文字は接頭辞 '#¥' を付けて表される.

当然のことであるが，'character 型の配列を生成して要素に文字を設定することで文字列を生成することもできる.

例. 文字列を配列として生成して要素（文字）を与える試み

```
(defparameter u1 (make-array 2 :element-type 'character)) Enter ← 2 文字から成る文字列を生成
→ U1          ←変数 U1 として生成

(setf (aref u1 0) #¥U51E6) Enter      ←最初の文字に 51E6 の UNICODE 文字を設定
→ #¥U51E6          ←設定された
(setf (aref u1 1) #¥U7406) Enter      ←次の文字に 7406 の UNICODE 文字を設定
→ #¥U7406          ←設定された

u1 Enter      ←変数 U1 の値を確認
→ "処理"          ←出来上がった文字列
```

3.2.1 特殊な文字

空白やタブ，改行といった文字を表 7 に示す.

表 7: 特殊な文字（一部）

文字	解　説	文字	解　説	文字	解　説
#¥space	空白	#¥tab	水平タブ	#¥newline	改行
#¥return	CR	#¥linefeed	LF		

参考） CR はキャラクタコード 13, LF はキャラクタコード 10.
SBCL では #¥return は#¥newline と同じ.

3.2.2　文字，文字コードの間の変換

指定した文字の**文字コード**を求めるには char-code を使用する．

書き方：　(char-code 文字)

例. 文字から文字コードを求める

```
(char-code #¥a)  Enter        ←英小文字 'a' の文字コードを求める
→  97          ←得られた文字コード

(char-code #¥あ)  Enter       ←日本語文字 'あ' の文字コードを求める
→  12354         ←得られた文字コード
```

指定した文字コードに対応する文字を得るには code-char を使用する．

書き方：　(code-char 文字コード)

例. 文字コードに対応する文字を取得する

```
(code-char 97)  Enter         ←文字コード 97 に対応する文字を求める
→  #¥a          ←得られた文字

(code-char 12354)  Enter      ←文字コード 12354 に対応する文字を求める
→  #¥HIRAGANA_LETTER_A        ←得られた文字：'あ' を意味する
```

3.2.3　文字列の連結

concatenate 関数を使用して文字列を連結することができる．concatenate は列の形式（sequence 型）のデータを連結するための関数である．

書き方：　(concatenate 型名　列 1　列 2　…　列 n)

「型名」の型の 列 1　列 2　…　列 n を連結したものを返す．引数に与えたデータは変化させない．

例. 文字列の連結

```
(defparameter q1 "I")  Enter      ← 1 つ目の文字列
→  Q1
(defparameter q2 " love")  Enter  ← 2 つ目の文字列
→  Q2
(defparameter q3 " lisp.")  Enter ← 3 つ目の文字列
→  Q3

(concatenate 'string q1 q2 q3)  Enter   ←文字列型（'string）の q1，q2，q3 を連結
→  "I love lisp."          ←連結結果

(list q1 q2 q3)  Enter        ←元の q1，q2，q3 を確認
→  ("I" " love" " lisp.")     ←変化なし
```

3.2.4　書式整形

各種の値を**書式整形**して文字列にするための format 関数がある．

書き方：　(format　出力先　書式制御　値 1　値 2　…　値 n)

値 1　値 2　…　値 n を「書式制御」の文字列に従って整形する．「出力先」として T を指定すると整形結果を標準出力に出力して NIL を返す．また「出力先」に NIL を指定すると整形結果を文字列として返し，標準出力には出力しない．(「出力先」には具体的な出力ストリームを指定することもできる)

「書式制御」は値を整形表示するための各種の**書式指定**（directive）含んだ文字列である．書式指定は '~' で始まる記述である．

例. 浮動小数点数を，小数点以下 2 桁，全長 5 桁で整形

```
(format t ">>>~5,2,F<<<" 3.1415926535d0)  Enter    ←これを評価すると…
>>> 3.14<<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

この例は倍精度浮動小数点数の 3.1415926535 の整形表示である．この例では書式指定として

```
~5,2,F
```

含んでおり，これによって「全長 5 桁，小数点以下 2 桁，浮動小数点数」を表している．このように，書式指定は

~パラメタ 1, パラメタ 2,…, パラメタ n, 書式文字

という形式で記述する.「書式文字」は表現の種類などを意味するものであり，F は浮動小数点数を意味する.（パラメタ列と書式文字の間には更にいくつかの修飾子を記述することがある）

format 関数に与える書式制御には上記の「書式指定」以外にも任意の文字列を含むことができる．上の例では '>>>' や '<<<' といったものを含めているが，それらはそのまま出力される．

3.2.4.1 各種の書式

書式指定の内，特に重要なものに関して説明する．

・整数

~桁数,D
~桁数,@D　　「@」は符号+/-を必ず付ける指定
~桁数,:D　　「:」は 3 桁区切りのコンマを表示する指定

「@」「:」は両方同時に記述することもできる．

例. 整数の書式整形

```
(format t ">>>~14,D<<<" 1234567890)  Enter    ←全長 14 桁で整形
>>>     1234567890<<<          ←整形結果の出力
NIL          ← format 関数の戻り値

(format t ">>>~14,@D<<<" 1234567890)  Enter    ←符号付き全長 14 桁で整形
>>>   +1234567890<<<          ←整形結果の出力
NIL          ← format 関数の戻り値

(format t ">>>~14,:@D<<<" 1234567890)  Enter    ←符号付き全長 14 桁，コンマ区切りで整形
>>>+1,234,567,890<<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

書式指定の「D」の代わりに「B」を用いると 2 進数,「O」（オー）を用いると 8 進数,「X」を用いると 16 進数の表示となる．

例. 2 進数，16 進数，8 進数への整形

```
(format t "~8,B" 255)  Enter    ← 2 進数に整形
11111111          ←整形結果の出力
NIL          ←戻り値

(format t "~2,X" 255)  Enter    ← 16 進数に整形
FF          ←整形結果の出力
NIL          ←戻り値

(format t "~3,O" 255)  Enter    ← 8 進数に整形
377          ←整形結果の出力
NIL          ←戻り値
```

・浮動小数点数

~全長, 小数点以下の桁数,F

~全長,E　　指数表現 (1)

~全長, 小数点以下の桁数,E　　指数表現 (2)

~全長, 小数点以下の桁数, 指数部の桁数 E　　指数表現 (3)

例. 指数表現

```
(format t ">>>~10,E<<<" 3.1415926535d0)  Enter    ←全長 10 桁で整形
>>>3.14159d+0<<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

```
(format t ">>>~10,2,E<<<" 3.1415926535d0)  Enter    ←全長 10 桁，小数点以下 2 桁で整形
>>>   3.14d+0<<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

```
(format t ">>>~10,2,2E<<<" 3.1415926535d0) Enter ←全長 10 桁, 小数点以下 2 桁, 指数部 2 桁で整形
>>>   3.14d+00<<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

・文字列など

~全長,A　　A は文字列を始めとするさまざまな形式に対応する書式指定

例. 文字列

```
(format t ">>>~A<<<" "abcdef")  Enter    ←そのままの文字列
>>>abcdef<<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

```
(format t ">>>~10,A<<<" "abcdef")  Enter    ←全長 10 桁で文字列を整形
>>>abcdef    <<<          ←整形結果の出力
NIL          ← format 関数の戻り値
```

```
(format t "~A" 'abc)  Enter        ←シンボルを書式整形
ABC          ←整形結果の出力
NIL          ← format 関数の戻り値
```

・改行

~%

~回数%　　「回数」だけ改行する

例. 改行

```
(format t "1st line~%2nd line~%3rd line")  Enter    ←改行を含んだ文字列
1st line          ←整形結果の出力（ここから）
2nd line
3rd line          ←（ここまで）
NIL          ← format 関数の戻り値
```

重要）(format nil …) とすると，各種の値を文字列に変換することができる.

例. 浮動小数点数を文字列に変換する

```
(defparameter x (format nil "~F" 3.1415))  Enter  ←浮動小数点数を変換して変数 x に設定する
→  X
```

```
x  Enter    ←変数 x の値を確認
→  "3.1415"          ←文字列である
```

3.2.5 文字列からの値の読み取り

文字列の記述内容を Common Lisp の値として読み取るには read-from-string 関数を使用する.

書き方： `(read-from-string` 文字列

「文字列」から値を読み取って返す.

例. 文字列から値を読み取る

```
(defvar x)  Enter      ←変数 x を宣言
→  X
(setq x (read-from-string "123"))  Enter      ←文字列 ”123” から値を読み取って x に設定
→  123          ←読み取った値
(* 2 x)  Enter      ← x の値が数値として使用できるか確認
→  246          ← x の値が数値であることがわかる
```

read-from-string 関数は**多値**を返す.

例. read-from-string 関数の戻り値を確認

```
(read-from-string "123")  Enter      ←戻り値を直接確認
→  123          ← 1 番目の戻り値
→  3            ← 2 番目の戻り値
```

このように，read-from-string 関数は 2 つの値を返す．この関数の戻り値を他の関数の引数に渡すと 1 番目の値が採用される．read-from-string 関数の 2 番目の戻り値は，読み取り元の文字列の次の読み取り位置のインデックスである．

多値に関しては後の「3.5 多値」（p.40）で説明する.

read-from-string 関数は，読み取り元の文字列の内容を適切に識別する.

例. 読み取り時の型の識別

```
(read-from-string "a")  Enter        ←シンボルの読み取り
→  A        ←シンボル
→  1

(read-from-string "2.718")  Enter        ←浮動小数点数の読み取り
→  2.718        ←浮動小数点数
→  5

(read-from-string "(a b c)")  Enter        ←リストの読み取り
→  (A B C)        ←リスト
→  7
```

3.3 データ構造の変換

3.3.1 coerce による変換

coerce を用いるとリストとベクトル，文字列の間で構造を変換することができる.

書き方： `(coerce` 元のデータ　型)

「元のデータ」を指定した「型」に変換したものを返す.

例. リストをベクトルに変換

```
(coerce '(a b) 'vector)  Enter        ←変換
→  #(A B)          ←ベクトル
```

例. ベクトルをリストに変換

```
(coerce #(a b) 'list)  Enter        ←変換
→  (A B)          ←リスト
```

例. 文字列をリストに変換

```
(coerce "ab" 'list)  Enter        ←変換
→  (#¥a #¥b)          ←リスト
```

例. リストを文字列に変換

```
(coerce '(#¥a #¥b) 'string)  Enter        ←変換
→  "ab"          ←文字列
```

例. ベクトルを文字列に変換

```
(coerce #(#¥a #¥b) 'string)  Enter        ←変換
→  "ab"          ←文字列
```

3.4 ハッシュ表

多量のデータの保持と読み出しを効率よく行うためのデータ構造として配列がある．配列は要素の格納位置のインデックス（整数値）に基づいてデータ要素にアクセスする．これに対して**ハッシュ表**（ハッシュテーブル）は，数値ではないデータに値を紐付ける形でたくさんのデータを管理することができる．このような**キー**に対して値を対応させる形のデータ構造は**連想配列**や**辞書**などと呼ばれることも多い．

3.4.1 基本的な使い方

ハッシュ表は make-hash-table で作成する．

書き方： `(make-hash-table)`

ハッシュテーブルを作成して返す．

例. ハッシュテーブルの作成

```
(defparameter h (make-hash-table))  Enter        ←ハッシュ表を作成して変数 h に設定
→  H          ←変数 H が宣言された
```

この例では，作成したハッシュ表を変数 H に設定している．

ハッシュ表のキーにアクセスするには gethash を使用する．

書き方： `(gethash キー　ハッシュ表)`

gethash は「ハッシュ表」の「キー」への参照を返すアクセサである．キーに対する値は次のようにして設定することができる．

```
(setf (gethash キー　ハッシュ表)　値)
```

例. ハッシュ表へのキーと値の登録（先の例の続き）

```
(setf (gethash 'orange h) "みかん")  Enter        ←キーと値のペアをハッシュ表に登録
→  "みかん"          ←値が登録された
```

キーに対して値を設定した後は gethash で値を取り出すことができる．

例. キーに対する値の取り出し（先の例の続き）

```
(gethash 'orange h)  Enter        ←キー 'orange に対する値を取得する
→  "みかん"        ←キーに対する値が得られた（1 番目の値）
→  T                ←キーが値を持っていることを示す T（2 番目の値）

(gethash 'grape h)  Enter         ←キー 'grape に対する値（未登録）を取得する試み
→  NIL         ←値が得られないと NIL（1 番目の値）
→  NIL         ←キーに対する値が登録されていないことを示す NIL（2 番目の値）
```

gethash でキーにアクセスすると，キーに対する値と，格納状況の 2 つの値を**多値**の形で返す．1 番目の値はキーに紐付けられた値（見つからなければ NIL），2 番目の値はキーに値が設定されている（T）か否（NIL）かを意味する．

3.4.2 要素の個数の調査

ハッシュ表に登録されているキーと値のペアの数を調べるには hash-table-count を使用する．

書き方： `(hash-table-count ハッシュ表)`

例. ハッシュ表に登録されているキーと値のペアの数（先の例の続き）

```
(setf (gethash 'grape h) "ぶどう")  Enter        ←キーと値のペアをハッシュ表に追加登録
→  "ぶどう"          ←値が登録された

(hash-table-count h)  Enter       ←ハッシュ表の要素数を調べる
→  2          ←データが 2 組登録されている
```

3.4.3 要素の削除

ハッシュ表に登録されているキーと値のペアを削除するには remhash を使用する．

書き方： `(remhash キー  ハッシュ表)`

「ハッシュ表」に登録されている「キー」とそれに紐付けられた値を削除する．削除の処理が正常に終了すると T，登録されていない「キー」を与えると NIL を返す．

例. ハッシュ表の要素の削除（先の例の続き）

```
(setf (gethash 'apple h) "りんご")  Enter        ←キーと値のペアをハッシュ表に追加登録
→  "りんご"          ←値が登録された
(hash-table-count h)  Enter       ←ハッシュ表の要素数を調べる
→  3          ←データが 3 組登録されている

(remhash 'orange h)  Enter        ←ハッシュ表 h から 'orange のキーを持つ要素を削除
→  T          ←削除された

(gethash 'orange h)  Enter        ←削除した要素を参照する試み
→  NIL         ←削除されており値が得られないので NIL（1 番目の値）
→  NIL         ←キーに対する値が登録されていないことを示す NIL（2 番目の値）

(hash-table-count h)  Enter       ←ハッシュ表の要素数を調べる
→  2          ←データが 2 組残っている
```

ハッシュ表に登録されている全てのデータを抹消するには clrhash を用いる．

書き方： `(clrhash ハッシュ表)`

この関数はハッシュ表（全要素の削除後）を返す．

例. ハッシュ表の全ての要素を抹消（先の例の続き）

```
(clrhash h)  Enter        ←ハッシュ表の全要素を抹消
→  #<HASH-TABLE :TEST EQL :COUNT 0 1004251E73>  ←削除された（SBCL の場合の戻り値）

(hash-table-count h)  Enter        ←ハッシュ表の要素数を調べる
→  0          ←要素数が 0 になっている
```

3.4.4　キーの照合方法の設定

ハッシュ表に登録されているキーを gethash で探す際，gethash の引数に与えたキーとハッシュ表に登録されているキーを比較照合している．このとき，暗黙では eql 関数で比較照合している．この場合，文字列をはじめとする複雑なデータをキーにすることができない．先に挙げた例では，キーとしてシンボルを与えており，eql による照合が可能であった．

例. 文字列をキーとして値を登録する試み（先の例の続き）

```
(setf (gethash "orange" h) "みかん")  Enter        ←キーに文字列を与えてデータを登録
→  "みかん"          ←値が登録されたが…

(gethash "orange" h)  Enter        ←値の取得を試みると…
→  NIL        ←値が得られないので NIL（1 番目の値）
→  NIL        ←キーの探索が失敗したことを示す NIL（2 番目の値）
```

この例では，一見すると登録されたはずのデータが取得できないように見えるかもしれないが，同値の文字列同士は eql 関数による比較では NIL となるので，ハッシュ表内でのキーの検索は失敗する．

シンボル以外のものをキーとするハッシュ表を作るには，make-hash-table を実行する（表を生成する）段階でキーワード引数 `:test` を与える．

書き方：　`(make-hash-table :test`　比較用関数`)`

暗黙では「比較用関数」として `#'eql` が設定されている．文字列など複雑な構造のデータは equal 関数で同値であることを検証するので

```
(make-hash-table :test #'equal)
```

としてハッシュ表を生成すると良い．

参考）関数名の接頭辞 `#'` は「関数そのもの」を意味する**特殊オペレータ**である．

例. equal でキーを照合するハッシュテーブル

```
(defparameter h2 (make-hash-table :test #'equal))  Enter  ← equal で照合するハッシュ表 h2 を作成
→  H2          ←変数 H2 にハッシュ表が作成された
(setf (gethash "orange" h2) "みかん")  Enter        ←キーに文字列を与えてデータを登録
→  "みかん"          ←値が登録されたが…
(setf (gethash "grape" h2) "ぶどう")  Enter        ←キーに文字列を与えてデータを登録
→  "ぶどう"          ←値が登録されたが…

(gethash "orange" h2)  Enter        ←値の取得を試みる
→  "みかん"        ←値が得られている（1 番目の値）
→  T        ←キーの探索が成功した（2 番目の値）
```

3.5 多値

values を使用すると複数の値を返すことができる．

書き方： (values 値 1 値 2 … 値 n)

値 1，値 2，…，値 n の値を個別に返す．

例えば次のような関数について考える．

```
(defun sqrt2 (n)
   (values (sqrt n)                ← 1 番目の値
           (* -1 (sqrt n))))       ← 2 番目の値
```

この関数 sqrt2 は正負の 2 つの平方根を返す．

例. 上記の sqrt2 を評価

```
(sqrt2 2d0)  Enter        ← sqrt2 を評価
→  1.4142135623730951d0          ← 1 番目の戻り値
→  -1.4142135623730951d0         ← 2 番目の戻り値
```

ただし，これら複数の戻り値を別の関数などに渡すと 1 番目の値が採用される．

例. sqrt2 の戻り値を変数に設定する試み（先の例の続き）

```
(defvar x)  Enter      ←変数 x を宣言
→  X
(setq x (sqrt2 2d0))  Enter      ← sqrt2 の戻り値を x に設定
→  1.4142135623730951d0          ←設定された値

x  Enter      ← x の値を確認
→  1.4142135623730951d0          ← sqrt2 の 1 番目の戻り値
```

多値の値を受け取る方法の 1 つが，multiple-value-setq を用いた複数の変数への値の設定である．

書き方： (multiple-value-setq 変数リスト 多値)

「変数リスト」の要素に複数の変数を記述すると，それら変数に「多値」（多値を返す式）の値を受け取ることができる．

例. sqrt2 の戻り値を 2 つの変数に設定する（先の例の続き）

```
(defvar x1)  Enter      ←変数 x1 を宣言
→  X1
(defvar x2)  Enter      ←変数 x2 を宣言
→  X2
(multiple-value-setq (x1 x2) (sqrt2 2d0))  Enter      ←変数 x1, x2 に多値を受け取る
→  1.4142135623730951d0

x1  Enter      ← x1 の値を確認
→  1.4142135623730951d0          ← sqrt2 の 1 番目の戻り値

x2  Enter      ← x2 の値を確認
→  -1.4142135623730951d0         ← sqrt2 の 2 番目の戻り値
```

多値の値を局所変数に受け取り，それらを用いて式（の列）を評価するには multiple-value-bind を使用する．

書き方： (multiple-value-bind 変数リスト 多値 式 1 式 2 … 式 n)

「変数リスト」に記述した複数の変数に「多値」（多値を返す式）の値を受け取り，式 1 式 2 … 式 n を評価する．最後の式 n が戻り値となる．

例. sqrt2 の戻り値を 2 つの局所変数 r1，r2 に受け取った後，一連の式を評価する（先の例の続き）

```
(multiple-value-bind (r1 r2) (sqrt2 2.0d0)  Enter     ←（記述ここから）
    (princ "First value:  ")  Enter     ←式 1
    (princ r1)  Enter     ←式 2
    (terpri)  Enter     ←式 3
    (princ "Second value:  ")  Enter     ←式 4
    (princ r2)  Enter     ←式 5
    (list r1 r2)  Enter     ←式 6
)  Enter     ←（記述ここまで）

First value:  1.4142135623730951d0
Second value:  -1.4142135623730951d0
(1.4142135623730951d0 -1.4142135623730951d0)        ←戻り値
```

3.5.1 多値からリストへの変換

多値をリストに変換するには multiple-value-list を使用する．

書き方： `(multiple-value-list 多値を返す式)`

例. 多値をリストに変換する（先の例の続き）

```
(multiple-value-list (sqrt2 2.0d0))  Enter     ←多値をリストに変換
→  (1.4142135623730951d0 -1.4142135623730951d0)     ← sqrt2 の戻り値がリストになった
```

3.6 例外処理（最も素朴な方法）

他の言語処理系と同様に Common Lisp も**例外処理**（エラーへの対処）の方法を持つ．Common Lisp の処理系には **Condition System** と呼ばれる機構が備わっており，処理系の状態に基づいた高度な制御が可能である．

ここでは Common Lisp の入門に際して必要となる，Condition System の機能の一部について解説する．

式の評価の際に，様々な理由で例外的な状況が発生して評価の処理が続行できない状態に陥ることがある．

例. 例外の発生：0 による除算（SBCL の場合）

```
(/ 1 0)  Enter        ← 1 ÷ 0 の値を求めようとすると…

debugger invoked on a DIVISION-BY-ZERO in thread     ←例外の発生によりエラーメッセージ
#<THREAD "main thread" RUNNING 10027900C3>:            が表示され，式の評価の処理が一時
  arithmetic error DIVISION-BY-ZERO signalled          中断する．
Operation was (/ 1 0).

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
  0:  [ABORT] Exit debugger, returning to top level.

(SB-KERNEL::INTEGER-/-INTEGER 1 0)
0]                  ←一時中断状態．ここに ‘ABORT’ と入力して Enter を押すと復帰する．
```

これは，0 による除算を試みて評価の処理が一時中断した例である．当然のことながら $\frac{1}{0}$ の値は定義されておらず，Common Lisp のみならず，他の言語処理系においても $\frac{1}{0}$ の演算を試みるとエラーが発生する．上の例のように処理が一時中断した状態では，ユーザは何らかの指示を与えなければならない．例えば上の例において ‘ABORT’ と入力して Enter を押すと処理系の状態は復帰する．

この例の様な演算における例外に対しては，例えば分母の値が 0 になるかどうかを演算に先立って判定することが可能であり，プログラミングの中で予防措置を取ることができる．しかし，入出力を始めとするシステム資源へのアクセスを含む処理では，起こりうる全ての例外を予防することが難しいだけでなく，プログラミング上の対策も煩雑なものとなる．

上の例に示すような「一時中断」の状態は，ユーザに判断を任せるためのものであるが，例外処理の機構を利用することで，例外が発生した原因に基づいて，次に行うべき処理を自動的に実行することができる．

Condition System の handler-case は例外処理のための 1 つの方法を与える．handler-case の最も単純な使用方法を次に示す．

《handler-case による例外処理》

(handler-case 式
(error (変数) 式の列 $_e$…)
(warning (変数) 式の列 $_w$…)
(:no-error (変数) 式の列 $_n$…)
)

例外が発生し得る「式」を評価する．この時エラーが発生すれば「式の列 $_e$…」を，警告が発生すれば「式の列 $_w$…」を，例外が発生しなければ「式の列 $_n$…」を評価し，式の列の最後の値を返す．

エラーや警告が発生した場合，例外に関する情報を持つ**コンディション**（condition）が「変数」に得られ，「式の列」の中で使用することができる．例外が発生しなければ「式」の評価結果が「変数」に得られ，「式の列 $_n$…」の中で使用することができる．

例. (/ 1 0) で発生するエラーのハンドリング

```
(defun extest ()
    (handler-case (/ 1 0)
        (error (c)                          ←エラー発生時のハンドリング
            (princ "Error!")                 ←"Error!"と表示
            (print c)                        ←コンディションの表示
            "ERR"                            ←戻り値
        )
        (warning (c)                        ←警告発生時のハンドリング
            (princ "Warning...")             ←"Warning..."と表示
            (print c)                        ←コンディションの表示
            "WRN"                            ←戻り値
        )
        (:no-error (c)                      ←例外がない場合のハンドリング
            (princ "No problem.")           ←"No problem."と表示
            c                                ←戻り値
        )
    )
)
```

この関数 extest を SBCL で評価すると次のような結果となる．

SBCL での評価.

```
(extest)  Enter        ←これを評価すると内部で (/ 1 0) が評価される
Error!                          ← princ による出力
#<DIVISION-BY-ZERO 10049CADD3>  ← print による出力
"ERR"  ←戻り値
```

演習課題. 上記関数 extest 内の handler-case の第 1 引数を (/ 6 2)（例外の起こらない式）に変更して extest の評価結果を確かめよ．

3.7　入出力

Common Lisp では入出力の対象を**ストリーム**（stream）という型のオブジェクトとして扱う．出力用のファイルをオープンするとそれが出力用のストリームとして，入力用のファイルをオープンするとそれが入力用のストリームとして得られる．そして，入出力の処理はストリームを介して行う．

3.7.1　ファイルのオープンとクローズ

ファイルをオープンするには open 関数を使用する．

《ファイルのオープン：open 関数》

(open　対象ファイル　:direction　入出力の種別)

「対象ファイル」にはファイル名を文字列で与える．「入出力の種別」には出力の場合に :output を，入力の場合に :input を指定する．（デフォルトは :input）この関数は開いたファイルに対応するストリームを返す．

open 関数には更にいくつかのキーワード引数を与えることができる．

■ 出力時に重要なキーワード引数

出力用にオープンするファイルが既に存在する場合は :if-exists で処理（`:supersede / :overwrite`）を選択する．

既存の内容を破棄：　`(open ファイル名 :direction :output :if-exists :supersede)`

既存の内容に上書き：　`(open ファイル名 :direction :output :if-exists :overwrite)`

この場合の上書き（`:overwrite`）は，既存の内容を削除しない．すなわち，既存の内容よりも上書きした内容の方が短い場合は，それより後の部分の既存の内容がファイルに残る．

■ 文字のエンコーディングに関するキーワード引数

入出力対象のファイルの**エンコーディング**（文字コードの体系）を指定するにはキーワード引数 `:external-format` に対象となるコード体系の種別（表 8）を指定する．

例. UTF-8 コードを含むテキストファイル ‘`test01.txt`’ を入力用にオープンする

```
(open "test01.txt" :direction :input :external-format :utf-8)
```

表 8: 入出力の文字コードの指定（SBCL の場合）

引数	文字コード体系	引数	文字コード体系
`:utf-8`	UTF-8	`:euc-jp`	EUC（Extended Unix Code）
`:shiftjis`	シフト JIS		

入出力の処理を終えた後はファイル（ストリーム）をクローズする．

《ファイルのクローズ：close 関数》

(close　ストリーム)

「ストリーム」を閉じて対応するファイルをクローズする．処理が終わると T を返す．

3.7.2　標準入出力

Common Lisp では標準入出力（ターミナルウィンドウの入出力）のためのストリーム（表 9）が予め用意されている．

表 9: 標準入出力のストリーム

ストリーム	解説
standard-input	標準入力（キーボード）
standard-output	標準出力（ターミナルウィンドウ）
error-output	標準エラー出力（SBCL の場合はターミナルウィンドウ）

3.7.3 入出力のための関数

3.7.3.1 出力

出力ストリームに対する出力処理を行うための基本的な関数に write がある．

《**write 関数**》

(write　S 式　:stream　出力用ストリーム)

「出力用ストリーム」に対して「**S 式**」の評価結果を出力する．出力した値と同じものを返す．キーワード引数 :stream と「出力用ストリーム」を省略すると，暗黙値である標準出力が出力先となる．

write 関数でファイル出力すると，出力内容の末尾に次の出力位置が移動する．すなわち，次々と write 関数を評価することで連続的に内容を出力することができる．

■ **改行出力**

出力用ストリームに対して**改行出力**するには terpri 関数を使用する．

書き方：　(terpri　出力用ストリーム)

引数の「出力用ストリーム」を省略すると標準出力が対象となる．

文字列データをテキストデータとして出力する場合は write-line 関数を使用すると良い．

《**write-line 関数**》

(write-line　文字列　出力用ストリーム)

「出力用ストリーム」に対して「文字列」を出力する．(:stream は不要）出力した文字列と同じものを返す．「出力用ストリーム」を省略すると，暗黙値である標準出力が出力先となる．

write，write-line でファイル出力すると，出力内容の末尾に次の出力位置が移動する．すなわち，これら関数を次々と評価することで連続的に内容を出力することができる．

本書内でこれまで用いて来た print，princ も出力先ストリームを指定することができる．

《**print，princ 関数**》

(print　S 式　出力用ストリーム)
(princ　S 式　出力用ストリーム)

「S 式」の評価結果を「出力用ストリーム」に対して出力する．出力ストリームを省略すると，標準出力が出力先となる．print は内容の出力に先立って改行出力をする．

print，princ 関数は，次のような write 関数による定義と等価である．(文献 [4])

(princ object output-stream)　と等価な記述

```
(write object stream output-stream :escape nil :readably nil)
```

(print object output-stream)　と等価な記述

```
(progn (terpri output-stream)
    (write object :stream output-stream :escape t)
    (write-char #¥space output-stream))
```

3.7.3.2　入力

入力ストリームにから入力処理を行うための基本的な関数に read がある.

《read 関数》

(read　入力用ストリーム)

「入力用ストリーム」から **S 式**を 1 つ読み取って 評価せずに 返す.「入力用ストリーム」を省略すると，暗黙値である標準入力からの入力となる.

例えば次のようなファイルから S 式を読み取ることを考える.

入力用ファイル：test01.lisp

```
1 (car '(a b c))
```

このファイルをオープンして S 式を読み取る処理を例示する.

例. ファイルのオープンと読み取り

```
(defparameter fi (open "test01.lisp" :direction :input))  Enter      ←ファイルのオープン
→  FI        ←入力用ストリームが変数 FI に得られた

(defparameter s1 (read fi))  Enter    ←入力用ストリームから S 式を 1 つ読み取る
→  S1        ←得られた S 式が変数 S1 に得られた

s1  Enter    ←内容確認
→  (CAR '(A B C))        ←得られた S 式（未評価）
```

入力ストリームにから行単位で内容を読み取る関数に read-line がある.

《read-line 関数》

(read-line　入力用ストリーム)

「入力用ストリーム」から 1 行文の内容を読み取って文字列として返す.「入力用ストリーム」を省略すると，暗黙値である標準入力からの入力となる.

read-line 関数は改行コードを区切りとして行を取り出す．すなわち，現在の入力位置から次の改行コードまでを読み取って返す．また，この関数は多値を返す．1 番目の値として読み取った行を返す．2 番目の値は，読み取った行の末尾が改行コードになっていたかどうかの情報である．すなわち，末尾が改行コードであった場合に NIL を，改行コードでなかった場合に T を返す.

注意）Common Lisp でテキストファイルを読み込む際は改行コードの扱いに注意すること．例えば Windows 版の SBCL（1.4）では Windows 標準の改行コード CR+LF を正しく認識せず，LF のみを改行コードとみなす．このことにより，Windows の標準的なテキストファイルを read-line で読み取った行の末尾に CR が残る.

3.7.3.3　ファイルの終端（EOF）の検出

ファイルからの入力を繰り返し，ファイルの終端に到達した状態で read や read-line を評価すると END-OF-FILE のエラー（例外）が発生する．例えば次のようなファイル data1.txt からの読み込みについて考える．

入力用ファイル：data1.txt

```
1 (1st line)
2 (2nd data)
```

このファイルは 2 行のリストから成るもので，この内容を読み込む試みを示す．

例. read による S 式の読み込み（SBCL の場合）

```
(defparameter fi (open "data1.txt" :direction :input))  Enter  ←ファイルを入力用に開く
→  FI          ←ファイルストリームが fi に得られた

(read fi)  Enter        ← fi から S 式を読み込む
→  (1ST LINE)          ← 1 行目の S 式が得られた
(read fi)  Enter        ← fi から S 式を読み込む
→  (2ND DATA)          ← 2 行目の S 式が得られた
(read fi)  Enter        ← fi から S 式を読み込むと…

debugger invoked on a END-OF-FILE in thread          ←エラー（例外）が発生する
#<THREAD "main thread" RUNNING 10027900C3>:
                    ⋮
                （途中省略）
                    ⋮
(SB-INT:FAST-READ-CHAR-REFILL #<SB-SYS:FD-STREAM for
"file C:¥¥Users¥¥katsu¥¥data1.txt" 10042687A3> T)
0]                  ←一時中断状態．ここに ‘ABORT’ と入力して Enter を押すと復帰する．
```

この例からわかるように，”data1.txt” の 2 行目の式を読み取った後はファイルの終端に到達し，その状態で read を評価するとエラー（例外）が発生している．

注意）この後 ‘ABORT’ して fi をクローズしておく必要がある．

read 関数には，次のように 2 番目，3 番目の引数を与えることができる．

（read ストリーム　eof-error-p　eof-value）

この「eof-error-p」は，ファイル終端での読み込み処理をエラーにするかどうかを設定するものであり，暗黙で T が設定（例外の発生が設定）されている．これに nil を設定すると例外を発生させずに，3 番目の引数「eof-value」を戻り値とする．これを応用して，読み込み時にファイルの終端を検出することができる．

例. ファイル終端を検出する形での読み込み

```
(defparameter fi (open "data1.txt" :direction :input))  Enter  ←ファイルを入力用に開く
→  FI          ←ファイルストリームが fi に得られた

(read fi nil "EOF")  Enter        ← fi から S 式を読み込む
→  (1ST LINE)          ← 1 行目の S 式が得られた
(read fi nil "EOF")  Enter        ← fi から S 式を読み込む
→  (2ND DATA)          ← 2 行目の S 式が得られた（この時点でファイル終端）
(read fi nil "EOF")  Enter        ← fi から S 式を読み込む
→  "EOF"          ← 3 番目の引数に与えた ”EOF” が戻り値（ファイル終端であることが判定できる）

(close fi)  Enter        ← fi を閉じる
→  T          ←ファイルが閉じられた
```

read-line 関数も，read 関数と同様に

（read-line ストリーム　**eof-error-p**　**eof-value**）

とすることができる．

参考）読み込みにおけるファイル終端の検出には，例外処理を応用する方法もある．

演習課題. 例外処理を応用してファイル終端を検出する形で，ファイルの全ての行を読み込むアルゴリズムについて考えよ．

3.8　日付，時刻，時間

3.8.1　日付・時刻の取得

現在の時刻を取得するには get-universal-time を使用する．

《現在時刻の取得》

(get-universal-time)　　引数なし

西暦 1900 年の開始時点からの秒数を表す整数値を返す．このような整数値を**ユニバーサルタイム**（universal time）と呼ぶ．

例. 現在時刻（秒）の取得

```
(get-universal-time)  Enter        ←現在時刻を秒単位で取得
→  3788751004          ←戻り値
```

日付，時刻に展開された形で値を取得するには get-decoded-time を使用する．

《現在の日付・時刻の取得》

(get-decoded-time)　　引数なし

「秒」,「分」,「時」,「日」,「月」,「西暦年」,「曜日」,「サマータイム」,「タイムゾーン」の 9 個の値を**多値**で返す.「曜日」は 0〜6 の整数値で 0 が月曜日，1 が火曜日…，6 が日曜日に対応する.「サマータイム」は適用されておれば T，適用されていなければ NIL となる.「タイムゾーン」は世界標準時との時差であり，得られた日付・時刻にこの値を加えたものが世界標準時となる．従って UTC との時差の表現とは正負が逆になっている．このような形で展開された日付・時刻の並びを decoded time と呼ぶ．get-decoded-time は閏秒（うるう秒）には対応していない．

例. 現在の日付・時刻の取得

```
(get-decoded-time)  Enter        ←現在の日付・時刻を取得
4          ←秒
50         ←分
15         ←時
23         ←日
1          ←月
2020       ←年
3          ←曜日（木）
NIL        ←サマータイム（適用されず）
-9         ←世界標準時との時差（UTC+9:00）
```

日付・時刻を構成する 9 個の多値を受け取るには multiple-value-setq や multiple-value-bind を使用すると良い．

例. get-decoded-time からの戻り値（多値）を受け取る方法

```
(multiple-value-bind (sec min hour date mon year day daylight-p zone)
  (get-decoded-time)
    式１　式２　…　式ｎ
)
```

この例では式内の局所変数 sec，min，hour，date，mon，year，day，daylight-p，zone に decoded time のそれぞれの値が設定され，それらを用いて 式 1，式 2，…，式 n が評価される．最後の「式 n」が戻り値となる．

3.8.2　日付・時刻とユニバーサルタイムの間の変換

decoded time から曜日とサマータイムの値を除いた列，すなわち

「秒」,「分」,「時」,「日」,「月」,「西暦年」,「タイムゾーン」

からユニバーサルタイム（整数値）に変換するには encode-universal-time を使用する．

書き方：　(encode-universal-time　秒　分　時　日　月　西暦年　タイムゾーン)

例. 日付・時刻からユニバーサルタイムへの変換

```
(encode-universal-time 4 50 15 23 1 2020 -9)  Enter  ←ユニバーサルタイムに変換
→  3788751004          ←ユニバーサルタイムが得られた
```

ユニバーサルタイムから日付・時刻の形式（decoded time の形式）に変換するには decode-universal-time を使用する．

書き方：　(decode-universal-time ユニバーサルタイム)

例. ユニバーサルタイムから decoded time への変換

```
(decode-universal-time 3788751004)  Enter  ← decoded time に変換
4          ←秒
50         ←分
15         ←時
23         ←日
1          ←月
2020       ←年
3          ←曜日（木）
NIL        ←サマータイム（適用されず）
-9         ←世界標準時との時差（UTC+9:00）
```

3.8.3　処理時間の計測

式の評価にかかる時間を計測するには time を使用する．

書き方：　(time 式)

引数に与えた「式」を評価するのに要した時間に関する各種情報を表示する．time の戻り値は「式」の戻り値である．

例. p.22 で示した fib 関数の評価にかかる時間を調べる

```
(time (fib 40))  Enter      ←評価時間の計測

Evaluation took:                                          ←計測結果の表示が開始される．
  2.017 seconds of real time                             ←要した全時間
  2.015625 seconds of total run time (2.015625 user, 0.000000 system)   ←評価に要した時間
  99.95% CPU
  5,235,862,004 processor cycles                         ←要した CPU のサイクル
  0 bytes consed

102334155                ← (fib 40) の戻り値
```

計測環境）CPU：Intel Core i7-6770HQ 2.6GHz，RAM：16GB，OS：Windows 10 Pro，Lisp：SBCL2.0

3.9 オブジェクトに関する情報の取得：describe

describe 関数を使用すると，オブジェクトに関する情報を出力することができる．

書き方：　(describe オブジェクト)

例. describe による情報表示（SBCL の場合）

```
(describe 'a)  Enter        ←変数でないシンボル a の情報
COMMON-LISP-USER::A
  [symbol]                  ←「シンボルである」
(defparameter a 123)  Enter       ←シンボル a を変数として宣言
A                       ←変数として宣言した
(describe 'a)  Enter        ←変数となったシンボル a の情報
COMMON-LISP-USER::A
  [symbol]                  ←「シンボルである」
A names a special variable:             ←「a は変数である」という情報
  Value:  123                   ←「123 という値を持つ」という情報
(describe 'car)  Enter        ←シンボル car の情報
COMMON-LISP:CAR
  [symbol]                  ←「シンボルである」
CAR names a compiled function:              ←「car は関数である」という情報
  Lambda-list:  (LIST)
  Declared type:  (FUNCTION (LIST) (VALUES T &OPTIONAL))
                  :
                （以下省略）
                  :
```

4　式の評価に関する事柄

4.1　eval

Lisp の処理系が式を評価する際の動作は次の 3 種類に分けることができる.

1. リスト (op arg1 arg2 … argn) が与えられた場合，各引数 arg1，arg2，…，argn を評価した値に置き換えた後で op の評価を行う．また，各引数の評価は再帰的に行う.（quote の場合は例外）
2. シンボルが与えられた場合，それを変数と見てその値を返す.
3. 値そのもの（数値や文字列など）が与えられた場合は，それをそのまま返す.

Lisp の処理系は式が与えられると上記の処理を自動的に実行する．またこの処理自体を明に実行する関数として eval がある.

書き方：　(eval 式)

「式」を評価する.

例. eval の動作

```
(defparameter s1 '(car '(a b c)))  Enter        ←評価可能な式を敢えて quote する
→  S1          ←変数 s1 に与えられた

s1  Enter        ←内容確認
→  (CAR '(A B C))          ←変数 s1 に式を表現するリストが保持されている

(eval s1)  Enter        ← s1 の値を評価する
→  A          ← (CAR '(A B C)) の評価結果が得られている
```

この例からわかるように，Lisp では評価可能な式（Lisp のプログラム）を単なるリスト形式のデータとして取り扱うことができる．また eval によってそれを改めて評価（実行）することができる.

演習課題. (defparameter s1 '(car '(a b c))) と (defparameter s1 (car '(a b c))) の違いを説明せよ.

演習課題. 上の例において (defparameter s1 '(car (a b c))) とするとエラーが発生する．その理由を説明せよ.

4.1.1　quote と eval

quote の式はその引数を 評価せずに 返す．例えば '(a b c) という式を Lisp の処理系に与えるかもしくは eval 関数で評価すると (a b c) が得られる.

例. quote の評価

```
'(a b c)  Enter        ← quote の式
→  (A B C)          ←評価結果

(quote (a b c))  Enter        ← quote の式：上と同じ意味
→  (A B C)          ←評価結果
```

これは「2.2.1 quote」（p.4）で示したことではあるが，eval による quote の式の扱いについては正確に理解しておく必要がある.

更に特徴的な例を示す.

例. quote の評価：その 2

```
''(a b c)  Enter        ← 2 重に quote された式
→  '(A B C)             ← 1 重の quote

(quote (quote (a b c)))  Enter        上と同じ式
→  '(A B C)             ← 1 重の quote
```

これらの例から次のことがわかる.

1. quote された式（引数に与えた式）は eval によって 未評価の形で 得られる.
2. n 回 quote された 式 は n 回の eval によって得られる.
2. n 回 quote された 式の値 は $n+1$ 回の eval によって得られる.

このことは，更に次の例でも確認できる.

例. eval の繰り返しによる式の値の変遷

```
(defparameter s ''(car '(a b c)))  Enter        ← 2 重に quote された式を変数 s に設定
→  S          ←変数 s に値が設定された

s  Enter        ← s の値を確認
→  '(CAR '(A B C))        ←上の defparameter の段階で eval されている（quote が 1 つ外れる）

(defparameter s (eval s))  Enter        ← s を eval して再度 s に設定
→  S          ←変数 s に値が設定された

s  Enter        ← s の値を確認
→  (CAR '(A B C))        ← eval により quote が 1 つ外れている

(defparameter s (eval s))  Enter        ← s を eval して再度 s に設定
→  S          ←変数 s に値が設定された

s  Enter        ← s の値を確認
→  A        ←上の式の評価結果が得られている
```

演習課題. `(defparameter s '(car '(a b c)))` として変数 s にリストを与え，再度 `(defparameter s s)` とすると変数 s の値はどうなるか予想せよ．また実行して結果を確かめ，何故そのようになったか理由を述べよ.

4.2 apply と funcall

関数 apply はオペレータや lambda をリストに対して適用する.

書き方： `(apply オペレータ 引数リスト)`

「オペレータ」を「引数リスト」に対して適用する.

例. apply による評価

```
(apply '+ '(1 2 3))  Enter        ← + に引数 1，2，3 を与えて評価
→  6          ←評価結果

(apply 'car '((a b c)))  Enter        ← car に引数 (a b c) を与えて評価
→  A          ←評価結果

(apply (lambda (x y) (+ x y)) '(1 2))  Enter        ← lambda に引数 1，2 を与えて評価
→  3          ←評価結果
```

演習課題. `(apply 'car '('(a b c)))` と `(apply 'car '((a b c)))` の違いを説明せよ.

apply に似た働きをする関数に funcall がある.

書き方： `(funcall オペレータ 引数列…)`

「オペレータ」を「引数列」に対して適用する.

例. funcall による評価

```
(funcall '+ 1 2 3)  Enter        ← (+ 1 2 3) と同様の評価
→ 6          ←評価結果

(funcall 'car '(a b c))  Enter        ← (car '(a b c)) と同様の評価
→ A          ←評価結果

(funcall (lambda (x y) (+ x y)) 1 2)  Enter        ← ((lambda (x y) (+ x y)) 1 2) と同様の評価
→ 3          ←評価結果
```

参考）apply，funcall の第一引数に与えるものは接頭辞「`#'`」で修飾された関数でもよい.

4.3 map 系の処理

《mapcar 関数》

(mapcar オペレータ　リスト 1　リスト 2　…　リスト n)

リスト 1，リスト 2，…，リスト n の各要素を先頭から 1 つずつ集め，それらを引数として「オペレータ」を適用し，各々の評価結果を要素とするリストを返す.

例. mapcar を用いた処理：その 1

```
(mapcar '+ '(1 2 3) '(10 20 30) '(100 200 300))  Enter  ←各リストから 1 要素ずつ集めて加算
→ (111 222 333)          ←評価結果：加算結果のリスト
```

例. mapcar を用いた処理：その 2

```
(mapcar (lambda (n) (* 2 n)) '(1 2 3 4 5))  Enter      ← lambda も使用できる
→ (2 4 6 8 10)          ←評価結果：lambda を適用した結果のリスト
```

引数として与えるデータの型や戻り値の型に柔軟に対応する map 関数もある.

《map 関数》

(map 戻り値の型　オペレータ　構造 1　構造 2　…　構造 n)

mapcar 関数と似た働きをするが，構造 1，構造 2，…，構造 n としてリストに限らないデータ列（配列など）を受け付ける．評価結果を「戻り値の型」に指定した型で返す.

例. mapcar と同様の処理

```
(map 'list '+ '(1 2 3) '(10 20 30))  Enter      ←各リストから 1 要素ずつ集めて加算
→ (11 22 33)          ←評価結果
```

例. 文字列の各要素の文字コードをリストにする

```
(map 'list (lambda (c) (char-code c)) "abcde")  Enter      ←アスキー文字列
→ (97 98 99 100 101)          ←評価結果

(map 'list (lambda (c) (char-code c)) "あいうえお")  Enter      ←日本語文字列
→ (12354 12356 12358 12360 12362)          ←評価結果
```

例. 文字コードのリストから文字列を構成する

```
(map 'string (lambda (c) (code-char c)) '(97 98 99 100 101))  Enter    ←評価
→ "abcde"          ←得られた文字列

(map 'string (lambda (c) (code-char c)) '(12354 12356 12358 12360 12362)) Enter  ←評価
→ "あいうえお"          ←得られた文字列
```

4.3.1 hashmap

ハッシュ表のエントリ（キーと値の組）に対して順次処理を適用するには hashmap を使用する.

書き方： (hashmap オペレータ ハッシュ表)

「ハッシュ表」の個々のエントリに対して順次「オペレータ」を適用する．この場合のオペレータは 2 つの引数を取る関数（もしくは lambda）で，第 1 引数にエントリのキーが，第 2 引数にエントリの値が与えられる．hashmap は NIL を返す.

例を挙げて hashmap について解説する．まず次のようにしてサンプルのハッシュ表 ht を作成しておく.

サンプルデータ. ハッシュ表 ht の作成

```
(defparameter ht (make-hash-table :test #'equal))  Enter
→ HT
(setf (gethash "orange" ht) "みかん")  Enter    エントリの登録 1
→ "みかん"
(setf (gethash "apple" ht) "りんご")  Enter    エントリの登録 2
→ "りんご"
(setf (gethash "grape" ht) "ぶどう")  Enter    エントリの登録 3
→ "ぶどう"
```

ht に 3 つのエントリが登録された．このハッシュ表の全てのエントリのキーと対応する値を出力する処理について考える．そのために次のような関数 hm1 を定義する.

関数定義. h1

```
(defun hm1 (k v)
       (princ k) (princ ":")
       (princ v)(terpri))
```

この関数は，与えられた 2 つの引数を princ により出力するものである．これを先に作成したハッシュ表に対して hashmap で適用してみる.

hashmap の適用. （先の例の続き）

```
(maphash 'hm1 ht)  Enter
→ orange:みかん          ← 1 番目のエントリに対する hm1 の適用による出力
→ apple:りんご                          :
→ grape:ぶどう           ← 3 番目のエントリに対する hm1 の適用による出力
→ NIL          ← hashmap の戻り値
```

もちろん，これと同様の処理は lambda を用いて実行することもできる.（次の例参照）

例. lambda による実行（先の例の続き）

```
(maphash (lambda (k v) (princ k) (princ ":") (princ v)(terpri)) ht)  Enter
→ orange:みかん          ← 1 番目のエントリに対する hm1 の適用による出力
→ apple:りんご                          :
→ grape:ぶどう           ← 3 番目のエントリに対する hm1 の適用による出力
→ NIL          ← hashmap の戻り値
```

hashmap を用いると全てのエントリを一度に取り出すことができる．次にその例を示す．そのために，次のような関数 hm2 を定義する．

関数定義. h2

```
(defun hm2 (k v)
           (setq keys (cons k keys)))
```

この関数は，第 1 引数の値を関数外部の変数 keys の値に cons で連結し更新するものである．これを先に作成したハッシュ表に対して hashmap で適用してみる．

hashmap の適用. （先の例の続き）

```
(defparameter keys '())  Enter     ←変数 keys に空のリストを作成しておく
→  KEYS
(maphash 'hm2 ht)  Enter     ← hm2 をハッシュ表に適用
→  NIL
keys  Enter     ←変数 keys の値を確認
→  ("grape" "apple" "orange")          ←全てのキーがリストの形で得られた
```

もちろん，これと同様の処理は lambda を用いて実行することもできる.（次の例参照）

例. lambda による実行（先の例の続き）

```
(defparameter keys '())  Enter     ←空リストの作成
→  KEYS
(maphash (lambda (k v) (setq keys (cons k keys))) ht)  Enter     ← lambda の適用
→  NIL
keys  Enter     ←変数 keys の値を確認
→  ("grape" "apple" "orange")          ←全てのキーがリストの形で得られた
```

4.4 マクロ

quote と eval の関係について先に説明したが，Lisp では S 式を明に生成し，それを改めて評価することができる．(次の例を参照のこと)

例. 生成した式を eval で評価する

```
(defparameter s (list '* 2 3))  Enter    ←(* 2 3) という式を生成して変数 s に与える
→  S          ←s に値が設定された

s  Enter        ←変数 s の値を確認
→  (* 2 3)          ←式が出来上がっている

(eval s)  Enter        ←それを改めて評価する
→  6          ←値が得られた
```

S 式の生成と評価を定義する方法に，defmacro を用いた**マクロ**の定義がある．

> **《マクロの定義》**
>
> **(defmacro マクロ名　(引数列)　式を生成する式)**
>
> 「引数列」に受け取った式を用いて「式を生成する式」を評価する．その評価結果として得られた式を評価し，その値を返す．

例. 与えた式を 2 倍する式を生成するマクロ

```
(defmacro dblm (n) (list '* 2 n))  Enter    ←マクロ dblm の定義
→  DBLM          ←マクロ dblm が定義された

(dblm 3)  Enter        ←マクロの展開と実行
→  6          ←値が得られた
```

この例において (dblm 3) を評価している．この評価過程において，まず最初に (list '* 2 3) という式が評価され，その結果として (* 2 3) という式が生成される．更にそれが評価され，最終的に値 6 が得られる．

マクロが展開される様子を確認するには macroexpand 関数を使用する．

マクロの展開：　(macroexpand (quote 式))

この関数は多値を返す．マクロの定義に沿って式を展開したものを第 1 の戻り値として返す．与えた式が展開できないもの（マクロでないもの）である場合，第 2 の戻り値として NIL を返し，マクロとして展開できた場合は T を返す．

例. macroexpand によるマクロの式の展開

```
(macroexpand '(dblm 3))  Enter    ←(dblm 3) が展開される様子を確認
→  (* 2 3)          ←第 1 の戻り値：展開された式
→  T                ←第 2 の戻り値：マクロとして展開できた
```

4.4.1 マクロを記述するための表現

式の接頭辞として**バッククォート**「`」を用いるとマクロ定義の記述が簡便なものになる．バッククオートには通常のシングルクオートに似た働きがある．

例. バッククオートの働き (1)

```
`(a b c)  Enter    ←式 (a b c) に接頭辞のバッククォートを付ける
→  (A B C)          ←与えた式を評価せずに返す．これは一見すると次のものと同等

'(a b c)  Enter    ←シングルクォートによる評価
→  (A B C)          ←評価結果
```

バッククォートは単なる評価の抑止ではなく，その記述の中に様々な編集のための表記を用いることができる．

■ 接頭辞としてのコンマ「,」

バッククォートされた式の中でコンマを記述すると，その直後の式を評価する．

例. コンマによる値の組み込み

```
`(a ,(+ 3 4) c)  Enter     ←バッククォートされた式の中にコンマ（接頭辞）を記述
→  (A 7 C)          ←コンマで修飾された部分のみ評価されている
```

この例からわかるように，バッククォートされた式の中でコンマを記述することにより，特定の部分のみを評価して式を作り上げる（未評価の状態で）ことができる．

■ 接頭辞「,@」

「,@」は，バッククォートされた式の中に 要素の列 を組み込む．

例. 複数の要素を「,@」を用いて式に組み込む

```
(defparameter s '(b c))  Enter     ← 2 つの要素を持つリストを変数 s に用意
→  S          ←変数に値が設定された
`(a ,@s d)  Enter     ←上の s を式の中に組み込む
→  (A B C D)          ←組み込みの結果できあがった式
```

注意）こで説明したコンマは，区切り文字（中置記法）ではなく接頭辞である ことに注意すること．

■ defmacro の仮引数に使用する「&body」

defun の仮引数に使用する &rest キーワードに似たものとして，&body キーワードが defmacro の仮引数に使用できる．

例. &body の使用例

```
(defmacro qexe (&body b) `(let () ,@b "end"))  Enter     ←仮引数に&body を使用
→  QEXE          ←マクロ qexe が定義された
(qexe (princ "a") (princ "b"))  Enter     ←マクロの展開と実行
→  ab          ← 2 つの princ による表示
→  "end"          ←戻り値
```

&body キーワードの後に記述された仮引数は，それ以降のマクロの引数の全てを意味する．この例では，マクロ実行時に与えた引数の全てを let 式の内部に展開して実行している．参考までに，この例で実行したマクロの式を展開して確認する．

例. マクロを展開して確認（先の例の続き）

```
(macroexpand '(qexe (princ "a") (princ "b")))  Enter     ←マクロの式を展開して確認
(LET ()              ←展開された式（ここから）
    (PRINC "a")
    (PRINC "b")
   "end")            ←展開された式（ここまで）
T          ←多値の 2 つ目の戻り値
```

defmacro 宣言時の仮引数には，&body キーワードより前に通常の仮引数の記述もできる．(次の例参照)

例. &body の使用例：その 2

```
(defmacro qexe2 (n &body b) `(let () (princ ,n) ,@b "end"))  Enter    ←マクロの定義
→  QEXE2          ←マクロ qexe2 が定義された

(qexe2 "start:" (princ "x") (princ "y"))  Enter    ←マクロの展開と実行
→  start:xy     ← princ による表示：1 番目の引数も受理されている
→  "end"         ←戻り値

(macroexpand '(qexe2 "start:" (princ "x") (princ "y")))  Enter  ←展開して確認
(LET ()              ←展開された式（ここから）
    (PRINC "start:")
    (PRINC "x")
    (PRINC "y")
   "end")            ←展開された式（ここまで）
T          ←多値の 2 つ目の戻り値
```

4.4.2 関数とマクロの違い

関数とマクロの違いについては明確に理解しておく必要がある．例えば，先の例で挙げたマクロ dblm と同じ値を返す関数 dbl は (defun dbl (n) (* 2 n)) として定義できるが，評価段階における Lisp 処理系の動作は明確に異なる．

関数としての式は，それに与えられた引数を再帰的に評価した後で全体が評価される．これに対してマクロは，評価すべき式をまず作成（編集）し，その後でそれを評価する．すなわちマクロは，Lisp のプログラムを生成し，それを評価するものであると言える．マクロのこのような性質により，プログラムとしての S 式の構文を別の構文に変換することが可能となる．これにより，プログラマが独自の構文を定義することが可能となる．実際に，Common Lisp に標準的に備わっている loop，dotimes，dolist，do といった評価の繰り返しのためのものはマクロとして定義されており，評価の段階でそれらの構文は更に単純な block や progn といった特殊オペレータに展開される．

マクロを応用して新たな構文を作成する例を示す．

■ while マクロの定義

条件を満たす間，評価を繰り返す while マクロ（文献 [1] から引用）を定義する．

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

このマクロを用いた例を示す．

例. while マクロによる繰り返し

```
(defparameter x 0)  Enter      ←変数 x に 0 を設定
X          ←変数が設定された

(while (< x 4) (print x) (setq x (+ x 1)))  Enter    ← while による繰り返し処理
0          ←繰り返しの初回
1
2
3          ←繰り返しの最終回
NIL          ←戻り値
```

演習課題. 上記の while と同等な構文を関数定義で実現する方法について考察せよ．
その上でマクロ定義の必要性について考察せよ．

4.5 シンボルオブジェクト

シンボルの実体は**シンボルオブジェクト**であり，変数として値を割り当てることができる．また関数名として計算の定義を割り当てることもできる．シンボルはいくつかの**スロット**を持ち，**名前**，**値**，**関数**の 3 つのものを割り当てることができる．(図 7)

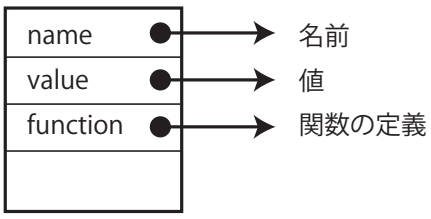


図 7: シンボルオブジェクトの重要なスロット（概略図）

これらを参照するには表 10 に示す関数や**アクセサ**を使用する．

表 10: シンボルのスロットに関する関数，アクセサ

記述	解説
`(symbol-name シンボル)`	「シンボル」の名前を返す関数.
`(symbol-value シンボル)`	「シンボル」の値を参照するアクセサ.
`(symbol-function シンボル)`	「シンボル」に割り当てられている関数を参照するアクセサ.

例. シンボルに割り当てられているものを確認する

```
(defparameter a 123)  Enter        ←変数 a に 123 を設定
→  A                  ←変数に値が設定された

(symbol-name 'a)  Enter            ←シンボル a の名前を調べる
→  "A"                ← "A" という文字列がシンボル a の名前である

(symbol-value 'a)  Enter           ←シンボル a の値を調べる
→  123                ←シンボル a の値は 123
```

シンボルの名前にはそのシンボルを表す 文字列 が割り当てられている．

symbol-value はアクセサであり，それに対して値を割り当てることができる．

例. symbol-value に対する値の設定（先の例の続き）

```
(setf (symbol-value 'a) 456)  Enter    ←値を設定
→  456                ←値が設定された

a  Enter              ←値を確認
→  456                ←値が設定されている
```

■ 関数が定義されているシンボル

関数が定義されているシンボルの symbol-function には関数定義の本体が保持されている．

例. シンボル car の関数定義を調べる

```
(symbol-function 'car)  Enter      ← symbol-function の参照
→  #<FUNCTION CAR>     ←関数定義の本体が表示された（SBCL の場合）
```

シンボルに割り当てられている関数定義の本体は**特殊オペレータ** `#'` を接頭辞にして求めることもできる．

例. シンボル car の関数定義を調べる：その 2

```
#'car  Enter        ←特殊オペレータ #' を使用する
→  #<FUNCTION CAR>          ←関数定義の本体が表示された（SBCL の場合）
```

4.5.1 各種の応用例

■ 関数が定義されているシンボルに値を設定する

```
(defun dbl (n) (* 2 n))  Enter        ←関数 dbl を定義
→  DBL          ←定義された

(setf (symbol-value 'dbl) 3)  Enter        ←シンボル dbl に値を設定
→  3          ←設定された

(dbl dbl)  Enter        ← 1 番目の dbl は関数名，2 番目の dbl は変数名
→  6          ←正しい結果が得られた
```

値を設定するスロットと関数定義を保持するスロットは別であるため，同一シンボルに値と関数定義の両方を与えることができる．

■ 関数のスロットに lambda を設定する

```
(setf (symbol-function 'sqr) (lambda (n) (* n n)))  Enter  ←関数のスロットに lambda を与える
→  #<FUNCTION (LAMBDA (N)) 1003EC235B>          ← lambda が設定された

(sqr 4)  Enter        ←評価を試みる
→  16          ←正しい結果が得られた
```

defun と同等のことができる．

5　型とデータ構造

5.1　型の判定

値やオブジェクトの型は typep 関数で検査することができる．

書き方：　(typep オブジェクト　型識別子)

「オブジェクト」の型が「型識別子」であるかどうかを判定する．

例. typep による型の判定

```
(typep 34 'number)  Enter        ←「34 は数値か？」
→  T          ←数値である

(typep "34" 'number)  Enter      ←「"34"は数値か？」
→  NIL          ←数値ではない
```

この例ではオブジェクトの型が数値かどうかを調べている．例の中にあるシンボル 'number が数値を意味する**型識別子**（type specifier）である．

数値の中でも更に「整数か浮動小数点数か」を検査する場合は，それぞれの型に対応する型識別子を用いる．

例. 整数か浮動小数点数かを判定する

```
(typep 34 'integer)  Enter       ←「34 は整数か？」
→  T          ←整数である

(typep 3.14 'integer)  Enter     ←「3.14 は整数か？」
→  NIL          ←整数ではない

(typep 3.14 'float)  Enter       ←「3.14 は浮動小数点数か？」
→  T          ←浮動小数点数である
```

この例からわかるように，型には階層がある．すなわち，下に示すように「数値」の型の中に「整数」,「浮動小数点数」が含まれる．そして，階層の各レベルにおける型の判定ができる．

《数値の型の階層》

```
number（数値）
    real（実数）
        rational（有理数）
            integer（整数）
        float（浮動小数点数）
    complex（複素数）
```

例えば，123 というデータは整数型（integer）であるだけでなく，有理数（rational）であり，実数（real）であり，数値（number）である．

演習課題. 与えられた数値が分数であるかどうかを判定する関数 fractionp を定義せよ．

5.2　型の階層

Lisp では全てのオブジェクトや値の型が階層的に整理されている．本書ではこれまで，数値，シンボル，文字，文字列，リスト，cons といった型のデータを扱ってきたが，それらは次のように階層的に整理されている．

《**Common Lisp における型の階層**》（概略）

```
t（全てのオブジェクトが属する最上位の型）
    function（関数）
    symbol（シンボル）
    character（文字）
    array（配列）
        vector（ベクトル）
            string（文字列）
        simple-array（単純な配列）
    sequence（シーケンス）
        vector（ベクトル）
        list（リスト）
            cons（cons セル）
    number（数値）
```

Common Lisp の全てのデータ型は T を頂点とする階層関係にある．ここに示した型の階層は Common Lisp の型の階層のごく 1 部である．また，上に示した vector 型のように，複数の型の配下に属するものも多い．

5.3 型名の調査

値やオブジェクトの型を正確に調べるには type-of 関数を使用する．

書き方： `(type-of オブジェクト)`

type-of は，処理系が「オブジェクト」を扱うための型の名前を返す．

例. type-of 関数による正確な型名の取得

```
(type-of 3.14)  Enter        ←浮動小数点数 3.14 の正確な型を調べる
→  SINGLE-FLOAT              ← float 配下の single-float という型である

(type-of 3.14d0)  Enter      ←浮動小数点数 3.14d0 の正確な型を調べる
→  DOUBLE-FLOAT              ← float 配下の double-float という型である
```

5.4 型の上下関係の判定

subtypep 関数を用いると，ある型が別の型の配下にあるかどうかを判定することができる．

書き方： `(subtypep 型名 1  型名 2)`

この関数は多値を返す．「型名 1」が「型名 2」の配下にある場合は 1 番目の戻り値として T を，そうでなければ NIL を返す．「型名 1」，「型名 2」ともに不正な型名でない場合は 2 番目の戻り値として T を，不正な型名があると NIL を返す．

例. 型の上下関係の判定

```
(subtypep 'integer 'number)  Enter     ←「integer は number の下にあるか？」
→  T          ←ある
→  T          ←引数に与えた型名も正常

(subtypep 'number 'integer)  Enter     ←「number は integer の下にあるか？」
→  NIL        ←ない
→  T          ←引数に与えた型名は正常
```

例. 型の上下関係の判定：不正な型の場合

```
(subtypep 'seisuu 'suuchi)  Enter        ←「seisuu は suuchi の下にあるか？」
→  NIL        ←ない…
→  NIL        ←というより，引数に与えた型名が不正
```

注意） subtypep の引数に同じ型名を 2 つ与えた場合は T を返す.

参考） 文献 [3] では，型の階層の上下関係において上位の型を supertype ， 下位の型を subtype と呼んでいる.

5.5 describe による型情報の調査

describe で型に関する情報を出力することができる.

例. describe による型情報の出力（SBCL の場合）

```
(describe 'integer)  Enter        ←整数型 integer に関する情報表示
COMMON-LISP:INTEGER
[symbol]
INTEGER names the built-in-class #<BUILT-IN-CLASS COMMON-LISP:INTEGER>:←組み込みの型である
Class precedence-list:  INTEGER, RATIONAL, REAL, NUMBER, T        ←上位の階層関係
Direct superclasses:  RATIONAL                                    ←直接の上位階層の型
Direct subclasses:  FIXNUM, BIGNUM                                ←直接の下位階層の型
Sealed.
No direct slots.
INTEGER names a primitive type-specifier:
Lambda-list:  (&OPTIONAL (SB-KERNEL::LOW '*) (SB-KERNEL::HIGH '*))
```

このように，型の階層の上下関係も出力される.

5.6 型の定義

deftype を用いて，プログラマが独自の型を定義することができる.

書き方： (deftype 型名 () 定義内容)

「定義内容」を満たす新たな型を「型名」の名前で定義する [6] .

■ 既存の型の別名として新たな型を定義する場合

書き方： (deftype 型名 () '既存の型))

例. 'integer の別名 'seisu を定義する

```
(deftype seisu () 'integer)  Enter  ← integer の別名である seisu を型として宣言する
→  SEISU        ←型として宣言された

(typep 2 'seisu)  Enter        ←型名として…
→  T        ←使用できる
(typep 2.0 'seisu)  Enter        ←型名として…
→  NIL        ←使用できる

(subtypep 'seisu 'number)  Enter        ←型の上下関係の検査
→  T        ←型の階層に
→  T        ←組み込まれている
```

[6] deftype は第 2 引数に仮引数のリストを取ることができ,「定義内容」の中で複雑な条件判定を行うことができる.
ただし，これに関することは本書では扱わない.

■ 既存の型の指定した値の範囲を新たな型とする場合

書き方： `(deftype 型名 () '(既存の型 開始値 終了値))`

「開始値」として '*'（アスタリスク）を指定すると下限なし,「終了値」として '*' を指定すると上限なしの指定となる.

例. 自然数（1 以上の整数）の型 naturalnumber を定義する

```
(deftype naturalnumber () '(integer 1 *))  Enter    ← 1 以上の整数
→  NATURALNUMBER          ←型として宣言された

(typep 12 'naturalnumber)  Enter          ←型名として…
→  T          ←使用できる
(typep 0 'naturalnumber)  Enter           ←型名として…
→  NIL          ←使用できる

(subtypep 'naturalnumber 'integer)  Enter          ←型の上下関係の検査
→  T          ←型の階層に
→  T          ←組み込まれている
```

■ 条件判定の式に基づいて新たな型を定義する場合

書き方： `(deftype 型名 () '(satisfies 関数名))`

「関数名」には 1 つの引数を取る関数の名前を与える．この関数が T となる値の型を「型名」とする.

例. 偶数の型 even を定義する

```
(evenp 2)  Enter     ←偶数かどうかを判定する関数 evenp がある
→  T          ←偶数なら T
(evenp 3)  Enter
→  NIL          ←偶数でないなら NIL

(deftype even () '(satisfies evenp))  Enter     ← evenp を満たす型 even を宣言する
→  EVEN          ←型として宣言された

(typep 2 'even)  Enter     ←型の検査
→  T          ←正しく判定できている
(typep 3 'even)  Enter     ←型の検査
→  NIL          ←正しく判定できている，しかし…

(subtypep 'even 'integer)          ←整数の配下の型であるかどうか検査
→  NIL          ←型の階層に正しく
→  NIL          ←組み込まれていない（T の配下には属している）
```

これは，Common Lisp の組み込みの関数 evenp を利用して偶数の型 even を定義しようとした例である．評価の最後の部分でわかるように，even 型は Common Lisp の型階層に正しく組み込まれていない．これを解決するには satisfies の式に加えて，明確な型の条件を加える.

例. 既存の型の指定と satisfies による型宣言

```
(deftype even () '(and integer (satisfies evenp)))  Enter  ← integer でかつ evenp を満たす
→  EVEN          ←型として宣言された

(typep 2 'even)  Enter      ←型の検査
→  T          ←正しく判定できている
(typep 3 'even)  Enter      ←型の検査
→  NIL          ←正しく判定できている，しかも…

(subtypep 'even 'integer)          ←整数の配下の型であるかどうか検査
→  T          ←型の階層に正しく
→  T          ←組み込まれている
```

この例のように，型の定義として

'(and　型指定 1　型指定 2　…　型指定 n)

のように記述すると，全ての型指定を満たすものを新たな型として宣言する．このような書き方の場合は satisfies の式を型指定に含める．もちろん

'(or　型指定 1　型指定 2　…　型指定 n)

のような記述も可能で，この場合は型指定の内少なくとも 1 つを満たすものを新たな型として宣言する．

5.7　構造体

構造体とは，複数の属性値を保持するデータオブジェクトである．Common Lisp における構造体では各属性の値は**スロット**と呼ばれるものが保持する．

《構造体の定義》

(defstruct 構造体名　スロット 1　スロット 2　…　スロット n)

スロット 1，スロット 2，…，スロット n のスロットを持つ構造体を「構造体名」の名前で定義する．各スロットは**スロット名**のみか，もしくは **(スロット名　初期値)** として初期値を与えることもできる．

例. 構造体の定義

```
(defstruct pc cpu clock ram disk os)  Enter      ←構造体 pc を定義
→  PC          ←定義された
```

この例では cpu，clock，ram，disk，os というスロットを持つ構造体 pc を定義している．

5.7.1　構造体の作成

defstruct で定義された構造体のオブジェクトを実際に作成するには **make-構造体名** を使用する．

書き方：　(make-構造体名　:スロット名 1　値 1　:スロット名 2　値 2 … :スロット名 n　値 n)

先の例で定義した構造体 pc に基づいて実際に構造体のオブジェクトを生成する例を示す．

例. 各スロットに値を与えて構造体のオブジェクトを生成する（先の例の続き）

```
(defparameter p1 (make-pc :cpu "Intel Core i7" :clock "2.4GHz" :ram "16GB"
                          :disk "1TB" :os "Windows 10 Pro")) Enter ←生成
→  P1          ←生成したオブジェクトが変数 p1 に設定された

p1  Enter        ←変数 p1 の内容を確認
→  #S(PC
      :CPU "Intel Core i7"          ←各スロットに
      :CLOCK "2.4GHz"               ←値が設定されて
      :RAM "16GB"                   ←いるのがわかる.
      :DISK "1TB"                   （SBCL の場合の表示）
      :OS "Windows 10 Pro")
```

5.7.2 スロットへのアクセス

構造体オブジェクトのスロットにアクセスするには**アクセサ「構造体名-スロット名」**を用いる.

書き方： （構造体名-スロット名　オブジェクト）

「構造体名」の構造を持つ「オブジェクト」の「スロット名」のスロットにアクセスする.

例. 構造体オブジェクトのスロットの値を参照する（先の例の続き）

```
(pc-cpu p1)  Enter  ←構造体 pc の具体的なオブジェクト p1 の，スロット cpu を参照
→  "Intel Core i7"        ←値が得られた
```

この例では構造体オブジェクトのスロットの値を参照（取得）している.
オブジェクトのスロットに値を設定するには，アクセサの式に対して setf を用いる.

例. 構造体オブジェクトのスロットに値を設定する（先の例の続き）

```
(setf (pc-cpu p1) "Intel Core i9")  Enter      ←アクセサの式に対して setf で値を設定
→  "Intel Core i9"        ←新たな値が設定された

(pc-cpu p1)  Enter      ←値の確認
→  "Intel Core i9"        ←新たな値に変更されていることがわかる
```

5.7.3 型としての構造体

defstruct で定義された構造体は，型の階層の中の 1 つの型（`structure-object` 配下の型）となる.

例. 定義された構造体の型（先の例の続き）

```
(type-of p1)  Enter       ←オブジェクト p1 の型を調べる
→  PC          ← defstruct で定義した構造体 pc が「型」となっている

(subtypep 'pc 't)  Enter       ←構造体 pc は T の配下にあるか
→  T       ←構造体 pc は T の配下の型である
→  T       ←構造体 pc は正当な型名である
```

defstruct で定義される構造体とは別に，Common Lisp では**オブジェクト指向**に基づいたデータオブジェクトの取り扱いも可能である．これに関しては次に説明する.

5.8 CLOS（オブジェクト指向）

Common Lisp は**オブジェクト指向プログラミング**を実現するための **CLOS**（**C**ommon **L**isp **O**bject **S**ystem）と呼ばれるシステムを持つ．CLOS においても他の言語処理系と同様に，**クラス**と**メソッド**を定義し，クラス定義に基づく**インスタンス**をオブジェクトとして生成する．

CLOS が他の言語処理系と異なる点としては，クラス定義とメソッドの定義が別れていることが挙げられる．例えば，Java や Python といった言語では，クラス定義の文の中に，そのクラス用のメソッドを定義する文が含まれるが，CLOS ではクラスを定義する式とメソッドを定義する式が別々のものになっている．また他の言語処理系では，クラスに含まれる要素を「メンバ」あるいは「プロパティ」と呼ぶが，CLOS では構造体の場合と同じように**スロット**と呼ぶ．

5.8.1 クラスの定義

クラスの定義には defclass を使用する．

《**クラスの定義**》

(defclass クラス名　(スーパークラス)　(スロットの定義))

「クラス名」の名前を持つ新たなクラスを定義する．この際に「スーパークラス」に指定した既存のクラスを継承する．「スーパークラス」には複数のクラス名を列挙して**多重継承**が可能である．「スロットの定義」にはスロットの名前や初期値などの定義を要素として持つリストを列挙する．

「スーパークラス」を省略すると standard-object（クラス階層の最上位）が直接のスーパークラスとなる．

スーパークラスとは，定義しようとする当該クラスの上位クラスのことで，当該クラスはスーパークラスのスロットや対応するメソッド（後述）を引き継ぐ．これをクラスの**継承**という．

他の言語処理系ではスーパークラスのことを**基底クラス**と呼ぶことがある．スーパークラスを継承するクラスを**派生クラス**，**拡張クラス**などと呼ぶ．

例えば，a，b，c の 3 つのスロットを持つクラス newclass を最も単純な形で定義するには

```
(defclass newclass () (a b c))
```

とする．

5.8.2 インスタンスの生成

クラスのインスタンスを生成するには make-instance を使用する．

《**インスタンスの生成**》

(make-instance クラス名)

「クラス名」のクラスのインスタンスを生成して返す．

クラス newclass のインスタンスを生成するには次のように記述する．

```
(make-instance 'newclass)
```

実際にはこの戻り値を変数に設定するなどして使用する．

5.8.3 スロットへのアクセス

slot-value 関数でスロットにアクセスすることができる．

書き方：　`(slot-value` インスタンス名　スロット名`)`

「インスタンス名」のオブジェクトの「スロット名」のスロットにアクセスする．

例. スロットへの値の設定と参照

```
(defclass newclass () (a b c))  Enter        ←クラス定義
→  #<STANDARD-CLASS COMMON-LISP-USER::NEWCLASS>        ←クラスが定義された
(defparameter x (make-instance 'newclass))  Enter    ←インスタンスを生成して変数 x に設定
→  X        ←変数 x が設定された
(setf (slot-value x 'a) 12)  Enter    ←オブジェクト x のスロット a に値 12 を設定
→  12        ←クラスが定義された
(slot-value x 'a)  Enter    ←オブジェクト x のスロット a を参照
→  12        ←値 12 が設定されていることがわかる
```

slot-value による方法以外にもスロットへのアクセス方法を設定できる．defclass によるクラス定義の段階で更に様々な設定が可能である．

《クラスの定義》その 2

(defclass クラス名　(スーパークラス)　(
**　(スロット 1　:accessor　アクセサ 1　:initform　式 1　:initarg　キーワード 1)**
**　(スロット 2　:accessor　アクセサ 2　:initform　式 2　:initarg　キーワード 2)**
**　　:**
**　(スロット n　:accessor　アクセサ n　:initform　式 n　:initarg　キーワード n)**
))

アクセサ 1〜アクセサ n は，それぞれのスロットにアクセスするためのものとしてプログラマが任意に命名できる．:initform に続く式は，インスタンスが生成される際に初期値を与える式として記述する．:initarg の後ろに与えるキーワードは，当該クラスのインスタンスを生成する際の初期値を与えるためのキーワードとして使用する．

例. 独自のアクセサ，初期値，初期化キーワードを与えたクラス定義

```
(defclass newclass () (
    (a :accessor slot-a :initform 1 :initarg :a)
    (b :accessor slot-b :initform 2)
    (c :accessor slot-c :initform 3))
)
```

この例では，a，b，c の各スロットに独自のアクセサを定義し，特にスロット a には初期化用のキーワード :a を定義している．

例. 上に定義した newclass のインスタンス生成 (1)

```
(defparameter x (make-instance 'newclass))  Enter  ←初期化用キーワード引数なし
X        ←変数 x にインスタンスを設定した
(slot-a x)  Enter        ← defclass で定義した独自のアクセサ slot-a でスロットを参照
1        ←初期値が設定されていることがわかる
(describe x)  Enter        ←変数 x の値に関する情報を見る
#<NEWCLASS 1003F70F03>          ←「newclass のオブジェクトである」
  [standard-object]          ←「スーパークラスは standard-object」
Slots with :INSTANCE allocation:
  A                             = 1        ←各スロットの値が表示される
  B                             = 2
  C                             = 3
```

例. newclass のインスタンス生成 (2)

```
(defparameter x (make-instance 'newclass :a 100))
```
Enter ←初期値を与えてインスタンス生成

```
X
```
←変数 x にインスタンスを設定した

```
(describe x)
```
Enter ←変数 x の値に関する情報を見る

```
#<NEWCLASS 10040CAD83>
  [standard-object]
Slots with :INSTANCE allocation:
  A                              = 100
  B                              = 2
  C                              = 3
```
←インスタンス生成時に与えた初期値が設定されている

5.8.4 メソッドの定義

defmethod で**メソッド**を定義する.

《メソッドの定義》

(defmethod メソッド名　(仮引数の列)　定義内容)

書き方は defun の場合と似ているが,「仮引数の列」の部分の書き方が異なり,**(仮引数名　型名)** を含めることができる．この場合,「仮引数名」は「型名」の型に限定した引数となる.

「仮引数の列」に **(仮引数名　型名)** を含む場合,「型名」の部分が異なる同名のメソッドの定義は別のメソッド定義として扱われる．これは，他の言語処理系における**メソッドのオーバーライド**に相当する.

メソッドの実行は

(メソッド名　引数列…)

と記述し，通常の関数の評価と同じ形式で評価を行い，戻り値を得る.

例. 異なる型のオブジェクトに対する同名のメソッド定義

```
;; number オブジェクトに対するメソッド meth1 の定義 (1)
(defmethod meth1 ((x number)) "数値")
;; string オブジェクトに対するメソッド meth1 の定義 (2)
(defmethod meth1 ((x string)) "文字列")
;; symbol オブジェクトに対するメソッド meth1 の定義 (3)
(defmethod meth1 ((x symbol)) "シンボル")
```

この定義の後，次のような評価を試みる

```
(meth1 123)
```
Enter ←数値に対してメソッドを評価

→ "数値" ←上記の定義 (1) が評価された

```
(meth1 "abc")
```
Enter ←文字列に対してメソッドを評価

→ "文字列" ←上記の定義 (2) が評価された

```
(meth1 'a)
```
Enter ←シンボルに対してメソッドを評価

→ "シンボル" ←上記の定義 (3) が評価された

メソッド meth1 が対象の型に応じて別々に定義されていることがわかる．このように，複数の型に対する別々の定義が同じメソッド名で扱えることを「メソッドの**ポリモーフィズム**」（polymorphism）と呼ぶ.

5.8.5　サンプルプログラム

■ クラスの継承とメソッドのポリモーフィズム　－「貯金箱」を例に

10 円硬貨，50 円硬貨，100 円硬貨を保持する貯金箱を考え，これを CLOS のクラス定義で実現する．

［クラス定義の概要］

貯金箱をクラス piggybank として定義する．このクラスは各種硬貨の枚数を保持するもので，10 円硬貨の枚数をスロット c10，50 円硬貨の枚数をスロット c50，100 円硬貨の枚数をスロット c100 にそれぞれ保持する．

《サンプル》piggybank のクラス定義

```
(defclass piggybank () (
    (c10 :accessor c10 :initform 0 :initarg :c10)
    (c50 :accessor c50 :initform 0 :initarg :c50)
    (c100 :accessor c100 :initform 0 :initarg :c100))
)
```

スロットと同じ名前でアクセサを定義している．また，インスタンス生成時の初期化のためのキーワードもスロット名に因んだ名前にしている．

［メソッド定義の概要］

貯金箱オブジェクト（piggybank クラスのインスタンス）に対して入金あるいは出金するためのメソッドを putin, takeout として定義する．

《サンプル》入金メソッド putin

```
(defmethod putin ((p piggybank) &key (c10 0) (c50 0) (c100 0))
    (setf (c10 p) (+ (c10 p) c10))
    (setf (c50 p) (+ (c50 p) c50))
    (setf (c100 p) (+ (c100 p) c100))
    nil)
```

《サンプル》出金メソッド takeout

```
(defmethod takeout ((p piggybank) &key (c10 0) (c50 0) (c100 0))
    (setf (c10 p) (- (c10 p) c10))
    (setf (c50 p) (- (c50 p) c50))
    (setf (c100 p) (- (c100 p) c100))
    nil)
```

入金する際はキーワード引数（:c10，:c50，:c100）に硬貨の種類毎の枚数を与える．これらメソッドの定義において，setf の式の中でアクセサとキーワード引数が同じ名前になっている点に若干注意していただきたい．

これらメソッドは nil を返す．

piggybank のインスタンスのスロット値を表示し，合計金額を返すメソッド sum を次のように定義する．

《サンプル》スロット値の表示と合計金額を算出するメソッド sum

```
(defmethod sum ((p piggybank))
    (princ " 10:  ") (princ (c10 p)) (princ #¥tab) (princ (* 10 (c10 p))) (terpri)
    (princ " 50:  ") (princ (c50 p)) (princ #¥tab) (princ (* 50 (c50 p))) (terpri)
    (princ "100:  ") (princ (c100 p)) (princ #¥tab) (princ (* 100 (c100 p))) (terpri)
    (+ (* 10 (c10 p)) (* 50 (c50 p)) (* 100 (c100 p)))
)
```

上に示したクラスとメソッドを定義して，インスタンスの生成とメソッドを実行する例を示す．

例. インスタンス生成とスロットの値の確認

```
(defparameter pg1 (make-instance 'piggybank :c10 1 :c50 1 :c100 1)) Enter ←インスタンス生成
PG1              ←変数 pg にインスタンスが設定された
(sum pg1)  Enter     ←合計金額の算出
 10:  1   10
 50:  1   50
100:  1   100
160          ←戻り値（合計金額）
```

例. 入金と金額の確認（先の例の続き）

```
(putin pg1 :c50 1 :c100 2)  Enter        ←入金
NIL          ← putin の戻り値
(sum pg1)  Enter     ←合計金額の確認
 10:  1   10
 50:  2   100
100:  3   300
410          ←戻り値（合計金額）
```

例. 出金と金額の確認（先の例の続き）

```
(takeout pg1 :c10 1)  Enter        ←出金
NIL          ← takeout の戻り値
(sum pg1)  Enter     ←合計金額の確認
 10:  0   0
 50:  2   100
100:  3   300
400          ←戻り値（合計金額）
```

［クラスの拡張とメソッドの定義］

先に示した piggybank クラスを拡張し，更に 1 円硬貨，5 円硬貨，500 円硬貨を収納するスロットを追加した piggybank-ex を定義する．

《サンプル》piggybank を継承したクラス piggybank-ex の定義

```
(defclass piggybank-ex (piggybank) (
    (c1 :accessor c1  :initform 0 :initarg :c1)
    (c5 :accessor c5  :initform 0 :initarg :c5)
    (c500 :accessor c500 :initform 0 :initarg :c500))
)
```

追加した硬貨のスロットに対応するためのメソッドを次のように定義する．

《サンプル》入金メソッド putin の定義

```
(defmethod putin ((p piggybank-ex) &key (c1 0) (c5 0) (c10 0) (c50 0) (c100 0) (c500 0))
    (setf (c1 p) (+ (c1 p) c1))
    (setf (c5 p) (+ (c5 p) c5))
    (call-next-method)
    (setf (c500 p) (+ (c500 p) c500))
    nil)
```

《サンプル》出金メソッド takeout の定義

```
(defmethod takeout ((p piggybank-ex) &key (c1 0) (c5 0) (c10 0) (c50 0) (c100 0) (c500 0))
    (setf (c1 p) (- (c1 p) c1))
    (setf (c5 p) (- (c5 p) c5))
    (call-next-method)
    (setf (c500 p) (- (c500 p) c500))
    nil)
```

● スーパークラスのメソッド呼び出し

上の定義の中に call-next-method という関数呼び出しがある[7] が，これはスーパークラスである piggybank 用に定義された同名のメソッドを呼び出すものである．

sum メソッドの定義を示す．

《サンプル》スロット値の表示と合計金額を算出するメソッド sum の定義

```
(defmethod sum ((p piggybank-ex) &aux s)
    (princ " 1:  ") (princ (c1 p)) (princ #¥tab) (princ (c1 p)) (terpri)
    (princ " 5:  ") (princ (c5 p)) (princ #¥tab) (princ (* 5 (c5 p))) (terpri)
    (setq s (call-next-method))
    (princ "500:  ") (princ (c500 p)) (princ #¥tab) (princ (* 500 (c500 p))) (terpri)
    (+ s (c1 p) (* 5 (c5 p)) (* 500 (c500 p)) )
)
```

piggybank-ex クラスのインスタンスの取り扱いを例示する．

例. piggybank-ex クラスのインスタンス生成

```
(defparameter pe1 (make-instance 'piggybank-ex
          :c10 6 :c50 5 :c100 4 :c1 3 :c5 2 :c500 1))   Enter   ←インスタンス生成
PE1            ←変数 pe にインスタンスが設定された

(sum pe1)   Enter     ←合計金額の算出
  1:  3   3
  5:  2   10
 10:  6   60
 50:  5   250
100:  4   400
500:  1   500
1223     ←合計金額
```

例. 入金と金額の確認（先の例の続き）

```
(putin pe1 :c1 2 :c50 1)   Enter        ←入金
NIL          ← putin の戻り値

(sum pe1)   Enter     ←合計金額の算出
  1:  5   5
  5:  2   10
 10:  6   60
 50:  6   300
100:  4   400
500:  1   500
1275     ←合計金額
```

[7] call-next-method の呼び出しは**メソッド結合**（method combination）の 1 つであるが，本書ではこれに関する説明は割愛する．

例. 出金と金額の確認（先の例の続き）

```
(takeout pe1 :c5 1 :c10 1)  Enter        ←出金
NIL          ← takeout の戻り値

(sum pe1)  Enter      ←合計金額の算出
  1:  5   5
  5:  1   5
 10:  5   50
 50:  6   300
100:  4   400
500:  1   500
1260      ←合計金額
```

演習課題. 上の例で用いた piggybank-ex クラスを継承し，更に紙幣（1,000 円，2,000 円，5,000 円，10,000 円）を収納できる cashbox クラスを定義せよ．また，cashbox クラスのための sum，putin，takeout メソッドを定義せよ．

6 パッケージ

Common Lisp の処理系で扱うシンボルは何らかの**パッケージ**に属する．例えば car 関数の名前のシンボルに関する情報を describe で調べると次のように表示される．

例. シンボル car に関する情報

```
(describe 'car)  Enter        ←シンボル car に関する情報の調査
COMMON-LISP:CAR          ←この部分に注目
  [symbol]
    （以下省略）
```

describe によって最初に表示された部分が COMMON-LISP:CAR となっているが，これは

「 COMMON-LISP パッケージに所属するシンボル CAR 」

を意味する．

6.1 名前空間とモジュール性

規模の大きなシステム構築においては**名前の衝突**の問題が起こる．プログラマが独自に定義する関数やマクロ，スペシャル変数（大域変数）などが多くなると，既に使用したシンボルを別の関数，マクロ，スペシャル変数の定義に割り当ててしまうことがあり，これが「名前の衝突」である．Common Lisp 以外の言語処理系においてもこの問題は深刻であり，関数や変数などに使用する記号の有効範囲を限定することで名前の衝突を回避している．このような「記号の有効範囲」のことを**名前空間**と呼び，新たに定義する関数や大域変数の名前を所属させ，他の名前空間とは同じ記号が干渉しないようにしている．

Common Lisp はパッケージに関する機能を提供しており，取り扱うシンボルは特定のパッケージに所属させることで名前の衝突を回避する．以下に例を示しながら解説する．

例. パッケージの作成（定義）

```
(defpackage :pkg1 (:use :common-lisp :common-lisp-user))  Enter   ←パッケージ pkg1 の作成
→  #<PACKAGE "PKG1">          ←パッケージ pkg1 が作成された
(defpackage :pkg2 (:use :common-lisp :common-lisp-user))  Enter   ←パッケージ pkg2 の作成
→  #<PACKAGE "PKG2">          ←パッケージ pkg2 が作成された
```

この例は別々の名前空間を持つ 2 つのパッケージ pkg1 と pkg2 を定義するものであり，パッケージを新規に定義するには defpackage を用いる．実際にシステムの名前空間を指定したパッケージに切り替えるには in-package を使用する．(次の例参照)

例. パッケージ pkg1 の選択による名前空間の切り替え（先の例の続き）

```
(in-package :pkg1)  Enter      ←パッケージ pkg1 を選択
→  #<PACKAGE "PKG1">          ←パッケージ pkg1 が選択された

(defun fun (n) (* 2 n))  Enter      ← 2 倍する関数 fun の定義
→  FUN          ←関数 fun が定義された
(fun 5)  Enter      ←関数 fun の評価
→  10          ←戻り値
```

ここまでは，通常の方法による関数の定義と評価と同じに見える．次に，別のパッケージに切り替えて，同じ名前の別の関数を定義してみる．

例. パッケージの切り替え（先の例の続き）

```
(in-package :pkg2)  Enter      ←パッケージ pkg2 を選択
→  #<PACKAGE "PKG2">           ←パッケージ pkg2 に切り替わった

(defun fun (n) (* n n))  Enter      ← 2 乗する関数 fun の定義
→  FUN          ←関数 fun が定義された
(fun 5)  Enter      ←関数 fun の評価
→  25          ←戻り値

(in-package :pkg1)  Enter      ←再度パッケージ pkg1 を選択
→  #<PACKAGE "PKG1">           ←パッケージ pkg1 に切り替わった

(fun 5)  Enter      ←関数 fun の評価
→  10          ← pkg1 で定義された関数 fun の評価結果

(in-package :common-lisp-user)  Enter      ←初期状態のパッケージ :common-lisp-user に戻す
→  #<PACKAGE "COMMON-LISP-USER">
```

この例からわかるように，同じ名前の関数 fun が異なるパッケージの間で別々に扱われている．

以上のように，パッケージの機能を利用することで，名前の衝突の問題を個々の名前空間の内側に閉じ込めることができる．パッケージ間の名前空間の独立性は，Lisp のプログラムの集まりの間の独立性（モジュール性）を実現し，システムの分割開発における安全性を高める．

演習課題. 上の例を実際に行い，:common-lisp-user に戻った状態で関数 fun を評価して「関数未定義」に由来するエラーが起こることを確認せよ．

6.2 シンボルの扱い

Common Lisp のシステム内部では，シンボルは「**パッケージ名::シンボル**」という形式で扱われている．p.73 の「**例.** シンボル car に関する情報」で示したように，関数 car の名前はシステム内部では「COMMON-LISP:CAR」として（コロン ':' の個数については後述する）扱われており，パッケージ名の接頭辞（**パッケージ修飾子**）があることで，パッケージ間でのシンボルの区別が実現されている．

例. 明にパッケージ修飾子を記述する（先の例の続き）

```
(pkg1::fun 5)  Enter      ← pkg1 の関数 fun の評価
→  10          ←戻り値

(pkg2::fun 5)  Enter      ← pkg2 の関数 fun の評価
→  25          ←戻り値
```

異なるパッケージ間で共通のシンボルも存在する．例えば，キーワード引数に用いられる「:rest」のようにコロン ':' の接頭辞を持つシンボルは，パッケージ間で共通であり，各パッケージから同一のものを指す．

参考）コロン ':' の接頭辞を持つシンボルの型は，symbol 型の配下の keyword 型である．

6.3 パッケージの扱い

6.3.1 Common Lisp の基本的なパッケージ

Common Lisp が標準的に提供する機能（関数やマクロなど）はパッケージ :common-lisp の下で提供されている．また，Common Lisp の処理系を開始した直後に式の入力を受け付ける状態になるが，これはパッケージ :common-lisp-user 配下でのインタラクションである．

6.3.2　使用中のパッケージを調べる方法

式の評価に使用している（その時点での）パッケージはスペシャル変数 *packages* に設定されている．

例. 使用中のパッケージを調べる

```
*package*  Enter      ←値を調べる
→  #<PACKAGE "COMMON-LISP-USER">          ←:common-lisp-user となっている
```

また，パッケージ名を取得するには package-name 関数に*package* の値を与えて評価する．

例. *package* からパッケージ名を取得する

```
(package-name *package*)  Enter      ←パッケージ名の取得
→  "COMMON-LISP-USER"          ←パケージ名
```

このように，パッケージ名が文字列の形で得られる．

パッケージに関する詳細は describe で調べることができる．

例. *package* の詳細情報

```
(describe *package*)  Enter          ←パッケージの詳細情報
#<PACKAGE "COMMON-LISP-USER">        ←パッケージ :common-lisp-user である
  [package]
Documentation:
  public:  the default package for user code and data
Nicknames:  CL-USER              ←このパッケージのニックネーム（別名）
Use-list:  COMMON-LISP, SB-ALIEN, SB-DEBUG, SB-EXT, SB-GRAY, SB-PROFILE
13 internal symbols.             ↑共に使用できる他のパッケージ群
```

この例からわかるように，基本的なインタラクションである :common-lisp-user は:common-lisp を始めとする様々なパッケージを読み込んでおり，それらが提供する機能を使用することができる．

6.3.3　パッケージの定義

defpackage でパッケージを定義する際，当該パッケージ内で必要となる他のパッケージを指定する．特に，Common Lisp の標準的な関数やマクロなどを定義した :common-lisp は多くの場合で必要となる．

《パッケージの定義》

(defpackage パッケージ名　(:use　既存のパッケージ　…))

「パッケージ名」の名前を持つ新たなパッケージを定義する．このパッケージで使用する関数やマクロなどのシンボルを「既存のパッケージ」から**継承**（inherit：文献 [4]）する．必要となる他のパッケージは複数指定できる．

基本的にパッケージ名は 文字列 で与えるが，シンボルの形で与えることもでき，その場合はそのシンボルの名前の文字列（p.58 「4.5 シンボルオブジェクト」参照）がパッケージ名となる．

複数のパッケージ間の継承関係を作ることで，パッケージ間の依存関係を構築することができる．

defpackage の引数には

(:nicknames 別名　…)

を与えることができ，定義するパッケージ名に別名を（複数）与えることができる．

6.3.4　パッケージの切り替え

カレントパッケージ（現在のパッケージ）はスペシャル変数 *package* が保持している．新たに定義した関数，マクロ，変数などのシンボルはカレントパッケージに属する．カレントパッケージを別のパッケージに切り替えるには in-package を使用する．

例. 新規パッケージの定義

```
(defpackage :mypackage
       (:use :common-lisp :common-lisp-user)
       (:nicknames :mypkg :mp))
```

パッケージ切り替えの例.

(in-package :mp) Enter ←カレントパッケージを :mp に切り替える

→ #<PACKAGE "MYPACKAGE">

package Enter ←カレントパッケージを確認

→ #<PACKAGE "MYPACKAGE"> ←カレントパッケージが切り替わっている

当該パッケージの使用を終えた後はカレントパッケージを :common-lisp-user に戻しておくのが良い.

例. :common-lisp-user（標準的なインタラクション）に戻す

(in-package :cl-user) Enter ← :cl-user（:common-lisp-user の別名）に戻す

→ #<PACKAGE "COMMON-LISP-USER"> ←カレントパッケージが元に戻った

:cl-user は :common-lisp-user の別名である.

6.4 プログラムのモジュール性の実現

パッケージの機能を利用すると，パッケージ毎のシンボルの局所化が実現できる．これにより，同名のシンボルをパッケージ間で別のものとして扱うことができ，名前の衝突の問題をパッケージ内に限定することができる.

実際に Lisp でプログラムを構築して利用する場合，パッケージ毎にプログラムファイルを作成して，必要に応じてそれらを Lisp 処理系に読み込んで使用するのが一般的である.

6.4.1 シンボルのエクスポート

パッケージ内で定義された関数やマクロ，スペシャル変数などのシンボルは，パッケージ外部からアクセスするには「**パッケージ名::シンボル**」と記述する．ただしこの形の記述は煩雑である.

パッケージの機能を使用する目的に名前の衝突の回避があるが，パッケージ内の限定されたシンボルのみをパッケージ修飾子を付けずに外部からアクセスできると便利である．実際にパッケージの形のプログラムを構築すると，定義した関数やマクロ，変数などは次の 2 つの種類に概ね分類できる.

1. 当該パッケージで実現しようとする主たる関数やマクロなど
2. 上のものを構成するための副次的な関数やマクロ，スペシャル変数など

特に上記 2 の種類のものは当該パッケージ内部でのみ使用するものが多い．従って，上記 1 に関するシンボルを始めとする「パッケージ外部で利用するもの」のみを**エクスポート**して外部からアクセスできるようにするのが良い.

defpackage によるパッケージ宣言の際に，キーワード引数 :export を与えて，指定したシンボルのみをパッケージ外部にエクスポートすることができる．また，use-package 関数を使用すると，カレントパッケージを切り替えずに，目的のパッケージからシンボルを継承することができる.

書き方： (use-package パッケージ名)

これにより,「パッケージ名」のパッケージがエクスポートしたシンボルのみをカレントパッケージに継承することができる.

上の説明に関する例を示す．次に上げるサンプルプログラム myvectorpackage.lisp は 2 次元ベクトルのノルム（長さ）を求める関数 norm を実装するものである．ユークリッド座標系の 2 次元ベクトル (x, y) のノルムは $\sqrt{x^2 + y^2}$ として得られるので,「2 乗を求める関数」sqr と,「それらの和の平方根」の関数 norm として実装している.

プログラム：myvectorpackage.lisp

```
;;; ベクトルの長さを求めるパッケージ
(defpackage :myvectorpackage
    (:use :common-lisp) (:nicknames :mvp)
    (:export :norm))    ;;; このパッケージのシンボル norm を外に出す
(in-package :mvp)

(defun sqr (n) (* n n)) ;;; この関数 sqr は外部に出さない

(defun norm (v)         ;;; この関数 norm は外部に出す
    (sqrt (+ (sqr (aref v 0)) (sqr (aref v 1)))))

(in-package :common-lisp-user)  ;;; CL-USERに切り替え，そこに
(use-package :mvp)              ;;; このパッケージのシンボル norm を出す
```

2 つの関数 sqr，norm の内，norm の方をパッケージ外にエクスポートする形にしている．このプログラム myvectorpackage.lisp を読み込んで使用する例を示す．

例. myvectorpackage.lisp（UTF-8）を使用する

```
(defparameter v (vector 3 4))  Enter      ←ベクトル (3, 4) を作成
→  V        ←変数 v に設定された

(load "myvectorpackage.lisp" :external-format :utf-8) Enter  ←パッケージファイルの読み込み
→  T        ←読み込み完了

(norm v)  Enter      ← v のノルムを求める
→  5.0        ←値が得られた

(sqr 3)  Enter       ← 3 の 2 乗を求めようとすると…

; in:  SQR 3                ←関数 sqr が存在しない旨のエラーメッセージ
; (SQR 3)
;
; caught STYLE-WARNING:
; undefined function:  COMMON-LISP-USER::SQR     ← COMMON-LISP-USER 配下に SQR はない
                :
            （以下省略）
                :
```

この例からわかるように，関数 norm は :common-lisp-user に向けてエクスポートされているのでパッケージ修飾子を付けずにそのまま評価できる．また，関数 sqr はエクスポートされていないのでエラーが発生している．

パッケージ内に隠蔽された（エクスポートされていない）シンボルに敢えてアクセスする場合はコロン 2 つ「**::**」を用いてパッケージ修飾子を添える．(次の例参照)

例. パッケージ内に隠蔽されたシンボルに敢えてアクセスする

```
(mvp::sqr 3)  Enter      ←「パッケージ名::」で修飾する
→  9        ←関数 sqr が使える
```

パッケージ外にエクスポートされているシンボルに明にパッケージ修飾子を付ける場合はコロン 1 つ「**:**」を記述する．

例. エクスポートされているシンボルにパッケージ修飾子を付ける

```
(mvp:norm v)  Enter      ←「パッケージ名:」で修飾する
→  5.0        ←値が得られた
```

7 Scheme

Scheme は Lisp の言語仕様の 1 つであり，最新の版は R7RS（文献 [10]）である．Scheme も Lisp の 1 種である関係上，式の書き方や，取り扱うデータや型など，基本的なことに関しては Common Lisp と共通のものが多い．本書では，Common Lisp の基礎を学んだ読者を対象に，Scheme の基本的な事柄について解説する．

7.1 本書で用いる Scheme の処理系

Scheme の仕様を満たす処理系は多数存在する．本書では **Gauche** という処理系を前提とする．Gauche は新しい言語仕様に沿っており，各種 OS（Windows，macOS，Linux）用のものが配布されている．また，メンテナンスも活発に行われており，処理系本体と関連情報を文献 [11] のインターネットサイトから入手することができる．

Gauche は R7RS に加えて独自に拡張した仕様を持つ．本書ではそれらについても若干触れる．

7.1.1 処理系の起動と終了

Gauche は OS のターミナルウィンドウから ‘gosh’ コマンドを投入することで起動する．Windows 版の Gauche では起動用のアイコン（図 8）をマウスでダブルクリックすることで起動することもできる．

図 8: Gauche 起動用のアイコン（Windows 版）

例. Windows 版 Gauche をコマンドプロンプトウィンドウで起動する

```
C:¥Users¥katsu>gosh   Enter        ← Gauche を起動するコマンド
gosh>                 ← Gauche のプロンプト．これに続いて Scheme の式を入力する
```

Gauche を終了して OS に戻るには `(exit)` と入力する．

7.2 基礎事項

Scheme では，式を括る括弧として ‘`[ … ]`’ が使用できる．ただし，この括弧は処理系内部では通常の ‘`( … )`’ として解釈される．

例. ‘`[ … ]`’ 括弧

```
[cdr '[a b c]]   Enter        ← (cdr '(a b c)) と同等の記述
→  (b c)                      ←戻り値
```

空リストは Common Lisp のような「`nil`」ではなく「`()`」と記述する．**ドット対**，car，cdr，cons に関しては Common Lisp の場合と同様である．Common Lisp における**コンスセル**に対応するものは，Scheme では単に**ペア**と呼ぶ．

Common Lisp と同様に describe でオブジェクトの情報を得ることができる．

例. describe による情報調査

```
(describe 'car)   Enter        ←シンボル car の調査
car is an instance of class <symbol>      ←各種情報が表示される
Known binding for variable car:
  In module 'scheme' (inlinable):
   #<subr (car obj)>
```

Scheme ではシンボルの大文字／小文字を区別する[8]．

例. 大文字／小文字の区別

```
'(a b C D e f)  Enter        ←大文字／小文字のシンボルの混在
→  (a b C D e f)             ←戻り値：大文字／小文字が区別されている
```

ただし，組み込みの関数，マクロなどのシンボルは基本的に小文字で記述する．

例. 組み込みの関数，マクロの名前は基本的に小文字

```
(car '(a b c))  Enter        ←小文字の関数名
→  a            ←戻り値

(CAR '(A B C))  Enter        ← car 関数を大文字で記述すると…
→  *** ERROR: unbound variable:  CAR        ← CAR が使えない旨のエラーが発生する
   Stack Trace:
                  ⋮
             （以下省略）
                  ⋮
```

car，cdr に関しては Common Lisp と同様である．また，要素の個数も Common Lisp と同様に length で求めることができる．

例. リストの長さの取得

```
(length '(a b c))  Enter     ←長さを求める length
→  3            ←戻り値
```

リストの要素への**インデックス**によるアクセスには ref が使用できる．

書き方： (ref オブジェクト　インデックス)

「オブジェクト」の「インデックス」の位置の要素にアクセスする．

例. ref による要素へのアクセス

```
(ref '(a b c d) 2)  Enter    ←要素へのアクセス
→  c            ←戻り値
```

重要） ref はリストのみならず，ベクトルや文字列といった <sequence> 型のオブジェクト（後述）に対するアクセサである．

特にリストの要素にアクセスする場合は list-ref も使用できる．

例. list-ref による要素へのアクセス

```
(list-ref '(a b c d) 2)  Enter   ←要素へのアクセス
→  c            ←戻り値
```

quote に関しては Common Lisp の場合と同様である．

例. quote の扱い

```
''(a b c)  Enter         ←二重の quote の式の評価
→  (quote (a b c))       ← 一重の quote の式
```

lambda に関しては Common Lisp の場合と同様であが，別名「^ 」が使える．

[8] 大文字／小文字を区別しないように処理系を設定することも可能である．

例. lambda とその別名

```
((lambda (n) (* 2 n)) 3)  Enter        ← lambda による式
→  6          ←評価結果

((^ (n) (* 2 n)) 3)  Enter        ← lambda の別名による式
→  6          ←評価結果（上と同じ）
```

lambda の仮引数には Common Lisp と同様に各種のパラメータが使用できるが,「&」ではなくコロン「:」の接頭辞で :optional, :key , :rest と記述する.

7.2.1 プログラムファイルの読み込み

Scheme の式を記述したファイルを読み込むには load を使用する.

書き方： (load ファイル名)

文字列で与えられた「ファイル名」のファイルから式を読み込んで評価する．ファイル名はパス名として与える．例えば，カレントディレクトリのファイル 'prg01.scm' を読み込むには (load "./prg01.scm") とする.

参考）Scheme のプログラムファイルには拡張子 '*.scm' を付けることが多いが特に制約はない.

7.2.2 変数の宣言，関数の定義

Scheme では define を用いて変数の宣言や関数の定義を行う.

《変数の宣言，関数の定義》

(define 変数名　式)　「変数名」を変数として宣言し,「式」の評価結果を割り当てる

(define 変数名)　「変数名」を変数として宣言する.

(define (関数名　仮引数列)　定義内容)

「仮引数列」の仮引数を用いて「定義内容」の計算を行う「関数名」の関数を定義する.

define で宣言された変数には set! で値を設定することができる.

書き方： (set!　変数　式)

「変数」に「式」の評価結果を設定する．宣言されていないシンボルに set! で値を設定しようとするとエラーが発生する.

例. 変数の宣言と値の設定

```
(define x 2)  Enter        ←変数 x を宣言して 2 を設定
→  x          ←変数 x が宣言された
(define y)  Enter        ←変数 y の宣言のみ
→  y          ←変数 y が宣言された
(set!  y 3)  Enter        ←変数 y への値 3 の設定
→  3          ←変数 y に値 3 が設定された
(+ x y)  Enter        ←変数を使用して式を評価
→  5          ←評価結果
```

例. 関数の定義（先の例の続き）

```
(define (plus n m) (+ n m))  Enter        ←関数 plus の定義
→  plus          ←関数 plus が定義された
(plus x y)  Enter        ←評価
→  5          ←評価結果
```

この例では plus というシンボルが関数名として宣言されているが，define で宣言されたシンボルなので set! で値を設定することもできる.（次の例参照）

例. 関数名に値を設定する試み（先の例の続き）

```
(set!  plus 13)  Enter        ←関数名であるシンボル plus に値を設定する
→  13          ←設定できた

plus  Enter          ←値を確認
→  13          ← plus の値
```

同様の方法で lambda 式を設定すると，結果として関数定義と同じことができる.（次の例参照）

例. set! で lambda 式を設定

```
(set!  plus (lambda (n m) (+ n m)))  Enter        ← plus に lambda 式を設定
→  #<closure (#f n m)>          ←クロージャオブジェクトとなる

(plus x y)  Enter        ←評価を試みる
→  5          ←値が得られた
```

lambda 式は**クロージャ**（closure）というオブジェクトになる.

Scheme では define による関数の定義も変数への値の束縛と同じ [9] であり，関数名を直接参照することができる.（次の例参照）

例. 関数名を直接参照（先の例の続き）

```
plus  Enter          ← plus の内容を確認
→  #<closure (#f n m)>          ←クロージャオブジェクトが得られる

car  Enter          ←組み込み関数 car を参照
→  #<subr (car obj)>          ← car の定義がオブジェクトとして得られる
```

`#<closure …>` や `#<subr …>` は**手続きオブジェクト**（`<procedure>` クラスのインスタンス）である.

7.2.3 逐次実行

Common Lisp と同様に let を用いることができる.

例. let による局所変数の宣言と逐次実行

```
(let ((v 0)) (print v) (set!  v (+ v 1)) (print v) v)  Enter
→  0          ← 1 つ目の print による出力
→  1          ← 2 つ目の print による出力
→  1          ← let の式の戻り値
```

7.2.4 条件分岐

Common Lisp と同様に if を用いることができる.

例. if による評価の選択

```
(if (> 10 3) "10 > 3" "other")  Enter        ←条件判定
→  "10 > 3"          ←条件式が真なのでこの戻り値
```

Common Lisp と同様に cond を用いることができる．ただし Scheme では else が使用できる.

[9] 関数の定義内容も**第一級オブジェクト**である.

例. 階乗関数 fct の定義（2 種）

```
(define (fct n)
  (cond ((= n 0) 1)
        (#t (* n (fct (- n 1))))))
```

```
(define (fct n)
  (cond ((= n 0) 1)
        (else (* n (fct (- n 1))))))
```

左の定義では「n が 0 でない場合」の条件部に「真」を意味する `#t` を記述しているが，右の定義のように「それ以外」を意味する `else` を記述することができる．

if や cond の条件部に記述する条件式は Common Lisp の場合と同じように and，or，not といった論理演算が使用できる．

Scheme では Common Lisp の case と同様の式が記述できる．ただし，Common Lisp の case における otherwise の代わりに else を記述する．

7.2.4.1 真理値

Scheme には**真理値**の型 `<boolean>` があり [10] ，真は `#t`，偽は `#f` という値である．

例. 真理値

```
(> 10 3)  Enter        ←真の式
→ #t          ←真の値
(< 10 3)  Enter        ←偽の式
→ #f          ←偽の値
```

7.2.4.2 リスト／ペアの判定

リストはペアと空リストを含む．オブジェクトがリストかどうかを判定するには list? を使用する．また，ペアかどうかを判定するには pair? を使用する．

例. リストかどうかを判定

```
(list?  '(a b))  Enter      ← (a b) はリストか？
→ #t          ←リストである
(list?  '())  Enter         ← () はリストか？
→ #t          ←リストである
(list?  'a)  Enter          ←シンボル a はリストか？
→ #f          ←リストではない
```

例. ペアかどうかを判定

```
(pair?  '(a b))  Enter      ← (a b) はペアか？
→ #t          ←ペアである
(pair?  '())  Enter         ← () はペアか？
→ #f          ←ペアではない
```

Common Lisp の atom に相当する関数は，Scheme には標準では備わっていない．

演習課題. Scheme においてアトムかどうかを判定する関数 `atom?` を定義せよ．

[10] Scheme の基礎事項として Common Lisp と大きく異なるのが真偽の値の扱いである．

7.2.4.3　同一，同値の判定

2 つのオブジェクト o1，o2 を比較する関数に eq?，eqv?，equal? がある．

eq? は o1，o2 がメモリ上の同一の実体である場合に真，そうでない場合に偽となる．

eqv? は eq? が真になる場合に加え，o1，o2 が共に同じ型の数値もしくは文字 <char> であって互いに同じ値である場合に真となる．それ以外の場合は偽となる．

equal? は o1，o2 がリストやベクトル，文字列といったデータ構造である場合の比較に使用する．o1，o2 が互いに同じ構造で全ての部分が同値である（eqv?が真になる）場合に真となる．また，eqv? が真になるような o1，o2 に対しても真となる．

表 11: eq? の評価の例

式	結果	式	結果	式	結果
(eq? 'a 'a)	#t	(eq? 2 2)	#t	(eq? #¥a #¥a)	#t
(eq? '(a b) '(a b))	#f	(eq? "abc" "abc")	#f	(eq? 2.0 2.0)	#f

アトムでないデータ構造同士であっても，同一のオブジェクトであれば eq? による比較は真となる．

例. 同一オブジェクト同士を eq? で比較する

```
(define x '(a b c))  Enter        ←変数 x にリストを設定
→  x
(define y x)  Enter        ←変数 y に変数 x が示すものを設定
→  y
(eq? x y)  Enter        ←比較すると…
→  #t        ←真
```

同じ値の浮動小数点数の比較において，eq? では偽となるが，eqv? では真となる．(次の例参照)

例. 同じ値の浮動小数点数の比較

```
(eq? 2.0 2.0)  Enter        ← eq? では…
→  #f        ←偽
(eqv? 2.0 2.0)  Enter        ← eqv? では…
→  #t        ←真
```

7.2.5　繰り返し

Common Lisp と同様に dotimes，dolist，do が使用できる．

例. dotimes

```
(dotimes (v 3 "end") (print v))  Enter
→  0        ←初回の実行による出力
→  1
→  2        ←最終回の実行による出力
→  "end"          ←戻り値
```

例. dolist

```
(dolist (x '(a b c) "end") (print x))  Enter
→  a        ←初回の実行による出力
→  b
→  c        ←最終回の実行による出力
→  "end"          ←戻り値
```

例. do

```
(do ((v 0)) ((= v 3) "end") (print v) (set!  v (+ v 1)))  Enter
→  0          ←初回の実行による出力
→  1
→  2          ←最終回の実行による出力
→  "end"           ←戻り値
```

dolist に似た繰り返し処理として for-each がある．これについていは後の「7.7.2 for-each」（p.96）で説明する．

7.2.6 多値

Scheme は Common Lisp と同様に**多値**を扱うことができる．

書き方： `(values 値1 値2 … 値n)`

値1 値2 … 値 n を多値とする．

例. 多値

```
(values 1 2 3)  Enter        ← 1，2，3 を多値とする
→  1          ← 1 番目の値
→  2          ← 2 番目の値
→  3          ← 3 番目の値
```

多値をリストに変換するには values->list を使用する．

例. 多値をリストに変換

```
(values->list (values 1 2 3))  Enter        ←多値をリストに変換
→  (1 2 3)           ←戻り値
```

多値を局所変数に受け取り，それらを用いて式（の列）を評価するには let-values を使用する．

書き方： `(let-values ((変数リスト1 多値1)(変数リスト2 多値2)…) 式の列…)`

「多値」を「変数リスト」（局所変数のリスト）に対応させる形で代入し，それらを用いて「式の列」を評価する．式の列の最後の式が戻り値となる．

例. let-values で多値を局所変数に代入する

```
(let-values (((x y) (values 3 4))) (print x) (print y) "end")  Enter
→  3          ← 1 番目の値を print
→  4          ← 2 番目の値を print
→  "end"        ←式の戻り値
```

7.2.7 数値

Scheme では，**整数** `<integer>`，**有理数** `<rational>`，**浮動小数点数**，**複素数** `<complex>` が使用できる．この内，複素数の表記が Common Lisp と異なる．

Scheme における複素数は

実部 + 虚部 i

と表記する．

複素数の例. `1-2i`，`0+i`

例. 複素数の計算

```
(* 1-2i 1+2i)  Enter        ←複素数同士の積
→  5.0           ←戻り値
```

複素数の**実部**，**虚部**を取得するには real-part，imag-part を使用する．

例. 実部，虚部の取り出し

```
(real-part 2+3i)  Enter        ←実部の取得
→  2.0            ←実部
(imag-part 2+3i)  Enter        ←虚部の取得
→  3.0            ←虚部
```

分数は Common Lisp と同様の扱いができ，分母，分子の取得にも denominator，numerator が使用できる．

浮動小数点の丸めには Common Lisp と同様に round が使用できる．

正確な値に変換するには exact を使用する．

例. exact で正確な値にする

```
(exact 1.0)  Enter        ← 1.0 を正確な値に変換
→  1              ←戻り値
(exact 0.001)  Enter      ← 0.001 を正確な値に変換
→  1/1000             ←戻り値
```

exact を使うと浮動小数点数を有理数に変換することができる．

例. exact で浮動小数点数を有理数に変換する

```
(exact 3.141592653589793)  Enter        ←正確な数に変換
→  245850922/78256779              ←戻り値
```

また同様の変換を，誤差を指定した形で行うには rationalize を使用する．

書き方： (rationalize 数値　誤差)

「誤差」に指定した許容誤差内で「数値」を有理数に変換する．この場合「数値」は正確な数に変換しておく．

例. 誤差を指定した変換

```
(rationalize (exact 3.141592653589793) 1/100)  Enter        ← 1/100 の誤差で変換
→  22/7           ←戻り値
(rationalize (exact 3.141592653589793) 1/10000)  Enter      ← 1/10000 の誤差で変換
→  333/106            ←戻り値
```

inexact を使用すると有理数を浮動小数点数に変換することができる．

例. inexact で浮動小数点数に変換

```
(inexact 1/3)  Enter        ←変換
→  0.3333333333333333              ←戻り値
```

7.2.7.1 数値に関する関数

数値に関する関数を表 12〜15 に示す.

表 12: 各種の判定

関数	解説	関数	解説	関数	解説
`(integer? n)`	n は整数	`(rational? n)`	n は有理数	`(real? n)`	n は実数
`(complex? n)`	n は複素数	`(number? n)`	n は数値		
`(positive? n)`	$n > 0$	`(negative? n)`	$n < 0$	`(zero? n)`	$n = 0$
`(odd? n)`	n は奇数	`(even? n)`	n は偶数		

表 13: 基本的な演算（算術など）

関数	解説	関数	解説
`(+ v1 v2 ⋯ vn)`	$v_1 + v_2 + \cdots + v_n$	`(- v1 v2 ⋯ vn)`	$v_1 - v_2 - \cdots - v_n$
`(* v1 v2 ⋯ vn)`	$v_1 \times v_2 \times \cdots \times v_n$	`(/ v1 v2 ⋯ vn)`	$v_1/v_2/\cdots/v_n$
`(gcd v1 v2 ⋯ vn)`	$v_1, v_2, \cdots, v_n$ の最大公約数	`(lcm v1 v2 ⋯ vn)`	$v_1, v_2, \cdots, v_n$ の最小公倍数
`(mod v1 v2)`	v_1 の法 v_2 における剰余	`(square v)`	v^2

表 14: 値の比較など

関数	解説	関数	解説
`(< v1 v2 ⋯ vn)`	$v_1 < v_2 < \cdots < v_n$	`(> v1 v2 ⋯ vn)`	$v_1 > v_2 > \cdots > v_n$
`(<= v1 v2 ⋯ vn)`	$v_1 \leq v_2 \leq \cdots \leq v_n$	`(>= v1 v2 ⋯ vn)`	$v_1 \geq v_2 \geq \cdots \geq v_n$
`(= v1 v2 ⋯ vn)`	$v_1 = v_2 = \cdots = v_n$		

表 15: その他，数学関数

関数	解説	関数	解説	関数	解説
`(exp x)`	e^x	`(expt x y)`	x^y	`(log x)`	$\log_e x$
`(log x n)`	$\log_n x$	`(sqrt x)`	$\sqrt{x}$	`(abs x)`	x の絶対値 *
`(sin x)`	$\sin(x)$	`(cos x)`	$\cos(x)$	`(tan x)`	$\tan(x)$
`(asin x)`	$\sin^{-1}(x)$	`(acos x)`	$\cos^{-1}(x)$	`(atan x)`	$\tan^{-1}(x)$
`(sinh x)`	$\sinh(x)$	`(cosh x)`	$\cosh(x)$	`(tanh x)`	$\tanh(x)$
`(asinh x)`	$\sinh^{-1}(x)$	`(acosh x)`	$\cosh^{-1}(x)$	`(atanh x)`	$\tanh^{-1}(x)$

* 複素数も可

円周率 pi はモジュール `math.const` で定義されている.

```
(use math.const)
```

としてモジュールを読み込むと pi が使用できる.

7.3 ベクトル

Scheme には一次元のベクトル[11] のためのデータ型 <vector>がある．**インデックス**（要素の位置）を指定する形での要素へのアクセスは，リストに対する場合よりも高速である．

7.3.1 ベクトルの作成

関数 vector でベクトルを作成することができる．

書き方： （vector 引数 1　引数 2　…　引数 n）

引数 1　引数 2　…　引数 n の n 個の要素から成る（n 次元）ベクトルを作成して返す．

例. ベクトルの作成

```
(vector 1 2 3)  Enter        ←ベクトルの作成
→  #(1 2 3)          ←戻り値として得られたベクトル
```

このようにベクトルの接頭辞は「#」である．これを直接記述してベクトルとすることもできる．

例. ベクトルの作成（その 2）

```
#(1 2 3)  Enter        ←ベクトルの作成
→  #(1 2 3)          ←ベクトルが得られる
```

ベクトルは make-vector 関数で作成することもできる．

書き方： （make-vector 次元）

正の整数値で「次元」を与えると値が未設定のベクトルが得られる．

例. make-vector によるベクトル作成

```
(make-vector 3)  Enter        ← 3 次元ベクトルの作成
→  #(#<undef> #<undef> #<undef>)          ←値が未設定の 3 次元ベクトル
```

#<undef> は「値を持たない」ことを意味する <undefined-object> 型の特殊なオブジェクトである．

7.3.2 要素へのアクセス

既に p.79 で ref を用いたリストの要素へのアクセスについて説明しているように，ref によってベクトルの要素にアクセスすることができるが，特にベクトルの要素にアクセスする際は vector-ref を用いると良い．

例. ベクトルの要素へのアクセス：参照

```
(define v1 (vector 1 2 3 4))  Enter        ←ベクトルの作成
→  v1          ←変数 v1 に設定された

(vector-ref v1 2)  Enter        ←ベクトルのインデックス位置 2 の要素を参照
→  3          ←要素が得られた
```

ref はアクセサであり，この式に対して値を設定することができる．（次の例参照）

例. ベクトルの要素へのアクセス：値の設定（先の例の続き）

```
(set! (vector-ref v1 2) 300)  Enter  ←インデックス位置 2 の要素に値を設定
→  #<undef>          ←設定完了

v1  Enter  ←ベクトルを確認
→  #(1 2 300 4)          ←要素に値が設定されていることがわかる
```

指定したインデックス位置に値が設定できていることがわかる．

[11] **ベクタ**と表記する文献も多い．

この例では set! で値を設定した場合，戻り値として #<undef> が得られている．set! の第 2 引数に変数のシンボルを記述した場合はそのシンボルが戻り値となるが，アクセサに対して値の設定を行った場合などに #<undef> が戻り値となる．

ベクトルの要素に値を設定する際はもっと簡単に vector-set! を使用することができる．

書き方： (vector-set! ベクトル インデックス 値)

「ベクトル」の「インデックス」の位置の要素に「値」を設定する．

例. vector-set! による要素の設定（先の例の続き）

```
(vector-set! v1 3 400) Enter ←インデックス位置 3 の要素に値を設定
→ #<undef>          ←設定完了

v1 Enter ←ベクトルを確認
→ #(1 2 300 400)          ←要素に値が設定されていることがわかる
```

7.3.3 ベクトル次元（長さ）

ベクトルの次元（要素の個数，長さ）を取得するには vector-length を使用する．

例. ベクトルの長さ（先の例の続き）

```
(vector-length v1) Enter ←ベクトルの長さ（次元）を求める
→ 4          ←戻り値
```

7.3.4 ベクトルと他のデータ構造との間の変換

ベクトルをリストに変換するには vector->list を使用する．

例. ベクトルをリストに変換（先の例の続き）

```
(vector->list v1) Enter ←ベクトルをリストに変換
→ (1 2 300 400)          ←戻り値
```

リストをベクトルに変換するには list->vector を使用する．

例. リストをベクトルに変換

```
(list->vector '(5 6 7 8)) Enter ←リストをベクトルに変換
→ #(5 6 7 8)          ←戻り値
```

文字列からベクトルに変換するには string->vector を使用する．

例. 文字列をベクトルに変換

```
(string->vector "記号処理") Enter ←文字列をベクトルに変換
→ #(#¥記 #¥号 #¥処 #¥理)          ←戻り値
```

得られるベクトルの要素は文字 <char> である．

文字 <char> を要素とするベクトルを文字列に変換するには vector->string を使用する．

例. ベクトルを文字列に変換

```
(vector->string #(#¥L #¥I #¥S #¥P)) Enter ←文字のベクトルを文字列に変換
→ "LISP"          ←戻り値
```

ベクトルの全ての要素を指定した値に設定するには vector-fill! を使用する．

書き方： (vector-fill! ベクトル 値)

「ベクトル」の全ての要素を「値」にする．

例. ベクトルを指定した値で満たす

```
(define v1 (make-vector 4))  Enter    ← 4 次元のベクトルを生成
→  v1          ←変数 v1 に設定した

(vector-fill!  v1 7)  Enter    ←ベクトルを 7 で満たす
→  #<undef>

v1          ←ベクトルを確認
→  #(7 7 7 7)          ←戻り値
```

7.4 文字と文字列

文字列 <string> は文字 <char> の列として構成される．文字は「#¥」の接頭辞を付けて記述する．

文字の例. #¥A　#¥あ　#¥U4E2D　…

Unicode の 16 進表記で文字を表すには「#¥U」の接頭辞を付ける．

エスケープ文字を文字として使用できる．また，表 16 に挙げるような特殊な文字も使用できる．

表 16: 特殊な文字（一部）

文　字	解　説	文　字	解　説	文　字	解　説	文　字	解　説
#¥space	空白	#¥tab	水平タブ	#¥newline	LF	#¥return	CR

CR は文字コード 13, LF は文字コード 10.

文字を**文字コード**の数値に変換するには char->integer を使用する．逆に文字コードの数値を文字に変換するには integer->char を使用する．

例. 文字と文字コードの間の変換

```
(char->integer #¥あ)  Enter    ←文字から文字コードを取得
→  12354          ←戻り値

(integer->char 12354)  Enter    ←文字コードから文字を取得
→  #¥あ          ←戻り値
```

文字列は二重引用符 "…" で括って記述する．

文字列の例. "abcde"　"文字列"　…

文字列の要素へのアクセスには ref が使用できる他，string-ref が使用できる．

例. 文字列の要素へのアクセス

```
(string-ref "abcde" 2)  Enter    ←インデックス位置 2 の文字にアクセス
→  #¥c          ←戻り値
```

文字列の長さは string-length で取得できる．

例. 文字列の長さを調べる

```
(string-length "abcde")  Enter    ←長さの取得
→  5          ←戻り値：長さ
```

文字列から値を読み込むには Common Lisp と同様に read-from-string を使用する．

例. 文字列から値を読み込む

```
(read-from-string "1234.5")  Enter     ←数値の読み込み
→  1234.5          ←戻り値

(read-from-string "(a b c)")  Enter     ←リストの読み込み
→  (a b c)          ←戻り値：長さ
```

7.4.1 文字列の分解と合成

文字列を分解して文字のリストにするには string->list を使用する．逆に文字のリストから文字列を合成するには list->string を使用する．

例. 文字の分解

```
(string->list "記号")  Enter     ←文字列を分解
→  (#¥記 #¥号)          ←戻り値
```

例. 文字の分解

```
(list->string '(#¥記 #¥号))  Enter     ←文字列の合成
→  "記号"          ←戻り値
```

文字列同士を連結するには string-join を使用する．

書き方： (string-join 文字列のリスト　区切り文字)

「文字列のリスト」の各要素を「区切り文字」を境にして連結する．

例. 文字列の連結

```
(string-join '("ab" "cd" "ef") ",")  Enter     ←文字列をコンマで連結
→  "ab,cd,ef"          ←戻り値

(string-join '("記号" "処理") "")  Enter     ←文字列を区切りなしで連結
→  "記号処理"          ←戻り値
```

区切り文字として空文字 "" を指定すると，区切りなしの連結となる．

string-join とは逆に，指定した区切り文字で文字列を分解するには string-split を使用する．

書き方： (string-split 文字列　区切り文字)

例. 区切り文字を境に文字列を分解

```
(string-split "ab,cd,ef" ",")  Enter     ←文字列をコンマで分解
→  ("ab" "cd" "ef")          ←戻り値
```

7.4.2 書式整形

Common Lisp のものと類似した書式整形[12] ができる．

書き方： (format 書式制御　値１　値２　…　値 n)

「書式制御」の文字列に従って「値 1」～「値 n」を整形し，文字列として返す．

[12]「3.2.4 書式整形」(p.33) で解説した内容と共通するものが多い．

例. 書式整形

```
(format ">>> 9D<<<" 65535)  Enter    ←全長 9 桁の 10 進整数に整形
→  ">>> 65535<<<"          ←戻り値

(format ">>> 9,2F<<<" 65535)  Enter  ←全長 9 桁，小数点以下 2 桁の浮動小数点数に整形
→  ">>> 65535.00<<<"       ←戻り値

(format ">>> 4,X<<<" 65535)  Enter   ←全長 4 桁の 16 進整数に整形
→  ">>>FFFF<<<"            ←戻り値
```

format はファイル出力にも使用することができる．

書き方： (format ポート 書式制御 値１ 値２ … 値 n)

「ポート」は出力先を示すものであるが，詳しくは後の「7.6 入出力」（p.93）で解説する．

7.5 ハッシュ表

7.5.1 ハッシュ表の作成

make-hash-table でハッシュ表を作成する．

書き方： (make-hash-table 比較器)

「比較器」はキーの比較に用いるもので，次の中から選ぶ．

比較器	解説	比較器	解説
eq-comparator	eq? による比較	eqv-comparator	eqv? による比較
equal-comparator	equal? による比較	string-comparator	文字列として比較

比較器を省略してハッシュ表を作成すると eq-comparator が比較器となる．

例. ハッシュ表の作成

```
(define ht (make-hash-table string-comparator))  Enter    ←ハッシュ表の作成
→  ht          ←戻り値：変数 ht にハッシュ表が設定された
```

7.5.2 エントリの登録と読み出し

ハッシュ表に**キー**と**値**のペア（**エントリ**）を登録するには hash-table-set! を使用する．

書き方： (hash-table-set! ハッシュ表 キー 値)

例. エントリの登録（先の例の続き）

```
(hash-table-set! ht "orange" "みかん")  Enter    ←ハッシュ表へのエントリの登録
→  #t          ←ハッシュ表にエントリが登録された
```

キーを指定してハッシュ表のエントリを読み出すには hash-table-get を使用する．

書き方： (hash-table-get ハッシュ表 キー)

例. エントリの読み出し（先の例の続き）

```
(hash-table-get ht "orange")  Enter    ←キーに対応するエントリの読み出し
→  "みかん"          ←読みだした値
```

登録されていないエントリをこの方法で読み出そうとするとエラーが発生する．（次の例参照）

例. 未登録のエントリへのアクセス（先の例の続き）

```
(hash-table-get ht "apple")  Enter     ←未登録のエントリの読み出しを試みると…
→  *** ERROR: #<hash-table string=?  0000000003e77…     ←エラーが発生する.
```

エントリの有無を調べるには hash-table-exists? を使用する.

書き方： `(hash-table-exists?  ハッシュ表  キー)`

「ハッシュ表」に「キー」のエントリがあれば `#t` を，なければ `#f` を返す.

例. エントリの存在検査（先の例の続き）

```
(hash-table-exists?  ht "orange")  Enter     ←"orange"のエントリは…
→  #t          ←存在する

(hash-table-exists?  ht "apple")  Enter     ←"apple"のエントリは…
→  #f          ←存在しない
```

7.5.3　エントリの削除，個数の調査

ハッシュ表のエントリを削除するには hash-table-delete! を使用する.

書き方： `(hash-table-delete!  ハッシュ表  キー)`

「ハッシュ表」から「キー」のエントリを削除する．登録されているエントリを削除すると `#t` を，元々登録されていないエントリの削除を試みると `#f` が返される.

ハッシュ表のエントリの個数を調べるには hash-table-size を使用する.

書き方： `(hash-table-size ハッシュ表)`

「ハッシュ表」に登録されているエントリの個数を返す.

例. エントリの削除と個数の調査（先の例の続き）

```
(hash-table-size ht)  Enter     ←エントリ数の調査
→  1          ← 1 個

(hash-table-delete!  ht "orange")  Enter     ←"orange"のエントリを削除
→  #t          ←削除完了

(hash-table-size ht)  Enter     ←エントリ数の調査
→  0          ← 0 個

(hash-table-delete!  ht "apple")  Enter     ←"apple"のエントリの削除を試みる
→  #f          ←元々存在しない
```

ハッシュ表に登録されている全エントリを削除するには hash-table-clear! を使用する.

書き方： `(hash-table-clear!  ハッシュ表)`

削除処理の後 `#<undef>` を返す.

例. ハッシュ表の全エントリを削除する

```
(hash-table-set!  ht "grape" "ぶどう")  Enter     ←エントリの登録
→  #t          ←登録完了

(hash-table-size ht)  Enter     ←エントリ数の調査
→  1          ← 1 個

(hash-table-clear!  ht)  Enter     ←全エントリの消去
→  #<undef>          ←戻り値

(hash-table-size ht)  Enter     ←エントリ数の調査
→  0          ← 0 個：エントリが全て消去された
```

7.6 入出力

Scheme では入出力対象のファイルを**ポート** <port> として扱う．すなわちファイルをオープンするとそれを指すポートオブジェクトが得られる．

7.6.1 ファイルのオープンとクローズ

入力用のファイルは open-input-file で，出力用のファイルは open-output-file でオープンする．

入力ファイルのオープン： (open-input-file ファイル名)
出力ファイルのオープン： (open-output-file ファイル名)

「ファイル名」は文字列で与える．正常にオープンすると，そのファイルに対応するポートオブジェクトが返される．
既存のファイルを出力用としてオープンすると既存の内容は失われるが，オープン時にキーワード引数

:if-exists 対処法

を与えて既存の内容の扱いを指定できる.「対処法」に指定するシンボルを次に示す.

:supersede	-	既存の内容を破棄する.（デフォルト）
:append	-	既存の内容の末尾に出力を追加する.
:overwrite	-	既存の内容はそのままにして，先頭から上書き出力する.

入出力ファイルの**エンコーディング**（文字コード体系）を指定するには，オープン時にキーワード引数

:encoding エンコーディング

を与える.「エンコーディング」には 'eucjp，'shift_jis，'utf-8 が指定できる.

ポートの使用が終われば **(close-port ポート)** でクローズする.

参考）入力用，出力用ポートを識別してクローズする close-input-port port ，close-output-port port もある.

7.6.2 read，write

ポートからオブジェクトを読み込むには read，ポートに対してオブジェクトを書き出すには write を使用する.

入力： (read ポート)
出力： (write オブジェクト　ポート)

read は「ポート」からオブジェクト（Lisp の式）を 1 つ読み込んで返す．write は「オブジェクト」を「ポート」に出力して #<undef> を返す．write による出力の形式は 式としての記述 になるものである．すなわち，read によって再度読み取ることができる形式である.
入力においてファイルの終端で read を評価すると #<eof> というオブジェクトが返される.
「ポート」を省略すると，read は**標準入力**から入力し，write は**標準出力**に出力する.
標準入出力のポートを表 17 に示す.

表 17: 標準入出力のポート

ポート	解説
(standard-input-port)	標準入力ポート
(standard-output-port)	標準出力ポート
(standard-error-port)	標準エラー出力ポート

行単位で文字列として入出力するには read-line，write-string を使用する．

行単位の入力： (read-line ポート)

行単位の出力： (write-string オブジェクト　ポート)

read-line は読み込んだ行を文字列として返す．write-string は文字列型の「オブジェクト」を「ポート」に出力して #<undef> を返す．「ポート」省略時の入出力は標準入出力となる．

出力ポートに対して改行出力するには **(newline ポート)** とする．

可読性の高い形で出力する display，print も使用できる．

書き方： (display オブジェクト　ポート)

ポートを省略すると標準出力に対して出力する．

書き方： (print オブジェクト 1　オブジェクト 2　…　オブジェクト n)

print は複数のオブジェクトを現在の出力ポート（標準出力など）に出力し，最後に改行処理を行う．

7.7 eval，apply，map

eval は処理系の評価機構を明に呼び出すものである．

書き方： (eval 式　環境識別子)

「環境識別子」で指定した環境の下で「式」を評価し，その値を返す．**環境**とは使用する**モジュール**（後述）や名前空間などを決めるもの（表 18）であり，Gauche 処理系の現在の版（0.9.9）においてはモジュールと同義である．

表 18: Gauche 0.9.9 における環境識別子（それを返す関数）

関数	解説
(null-environment 5)	**null モジュール.** 単なる構文的な束縛を含む.（R5RS で規定）
(scheme-report-environment 5)	**scheme モジュール.** 構文的な束縛と手続きの束縛を含む.（R5RS で規定）
(interaction-environment)	**user モジュール.** 全ての Gauche のビルトインとユーザ定義を含んむ.

例. eval による式の評価

```
(define s1 '(car '(a b c)))  Enter      ←式を quote
→  s1          ←変数 s1 に設定された
(eval s1 (interaction-environment))  Enter      ←上の式を評価
→  a          ←評価結果
```

apply は手続きオブジェクトをデータ列に対して適用して評価する．

書き方： (apply 手続きオブジェクト　データ列…)

「手続きオブジェクト」を「データ列」に対して適用する．このとき「データ列」の最後はリストでなければならない．

例. + を数値の列に適用する

```
(apply + 1 2 3 '(4))  Enter        ← + を 1, 2, 3, 4 に適用
→  10          ←評価結果
```

apply の引数の形式は一見して不自然に見えるかもしれないが

「全ての引数を右端から左に向かって順番に cons で結合した後にそれを評価する」

と考えると理解しやすい.

map はリストの全ての要素に手続きオブジェクトを適用し，それら評価結果をリストにして返す．特に Gauche 処理系では (use gauche.collection) を評価して必要なモジュールを読み込んでおくと，リスト以外にもベクトル，文字列，ハッシュ表といった <collection> 型のオブジェクトに対して map が適用できる.

書き方： (map 手続きオブジェクト　列 1　列 2　…　列 n)

列 1　列 2　…　列 n それぞれの要素を 1 つずつ順番に取り出し，それらに「手続きオブジェクト」を適用する.

例. 1 つのデータ列に対する map

```
(map square '(1 2 3 4 5))  Enter        ← + を 1, 2, 3, 4, 5 に適用
→  (1 4 9 16 25)          ←評価結果をリストにしたものが戻り値
```

例. 複数のデータ列に対する map

```
(map + '(1 2 3) '(10 20 30) '(100 200 300))  Enter
→  (111 222 333)          ←各列の要素を 1 つずつ取り出して加算し，リストにする
```

map の第 2 引数以降に与えるデータ列はリストに限らない.

例. 文字列に対する map

```
(use gauche.collection)  Enter  ←モジュールの読み込み
(map char->integer "ABC")  Enter  ←各文字の文字コードをリストにする
→  (65 66 67)          ←戻り値
```

7.7.1　ハッシュ表に対する map

ハッシュ表に対する map の処理は，個々のエントリ（キーと値の組）に対するものである．すなわち，手続きオブジェクトに渡される個々の値は **(キー . 値)** のペアである.

次のように作成したハッシュテーブル ht を例にして map の処理について説明する.

例. サンプルのハッシュ表 ht

```
(define ht (make-hash-table string-comparator))
(hash-table-set!  ht "orange" "みかん")
(hash-table-set!  ht "apple" "りんご")
(hash-table-set!  ht "grape" "ぶどう")
```

このハッシュ表に対して，値をそのまま返す lambda の式 (lambda (p) p) を map で適用する例を示す.

例. ハッシュ表に対する map

```
(map (lambda (p) p) ht)  Enter
→  (("grape" .  "ぶどう") ("apple" .  "りんご") ("orange" .  "みかん"))
```

全てのエントリが **(キー . 値)** の形で得られていることがわかる.

7.7.2 for-each

リストの個々の要素に対する処理を行うものとして for-each がある．これは map に使用方法が似ているが，値を返さない（戻り値は `#<undef>`）繰り返し処理である．特に Gauche 処理系では `(use gauche.collection)` を評価して必要なモジュールを読み込んでおくと，リスト以外にもベクトル，文字列，ハッシュ表といった `<collection>` 型のオブジェクトに対して for-each が適用できる．

書き方： `(for-each` 手続きオブジェクト　列 1　列 2　…　列 n`)`

列 1　列 2　…　列 n それぞれの要素を 1 つずつ順番に取り出し，それらに「手続きオブジェクト」を適用する．

例. 1 つのデータ列に対する for-each

```
(for-each print '(1 2 3))   Enter   ←リストの要素に順番に print を適用する
→  1          ←最初の要素を print
→  2
→  3          ←最後の要素を print
→  #<undef>          ←戻り値
```

例. 複数のデータ列に対する for-each

```
(for-each (lambda (:rest r) (print (apply + r)))   Enter   ←各リストの要素を順番に取り出し
          '(1 2 3) '(10 20 30) '(100 200 300))   Enter   ←加算して print を適用する
→  111          ←各リストの最初の要素を加算して print
→  222
→  333          ←各リストの最後の要素を加算して print
→  #<undef>          ←戻り値
```

例. 文字列に対する for-each

```
(use gauche.collection)   Enter   ←モジュールの読み込み
(for-each print "文字列")   Enter   ←文字列の各要素に対して print
→  文          ←最初の文字を print
→  字
→  列          ←最後の文字を print
→  #t          ← gauche.collection 使用時の戻り値
```

7.8 マクロ

Scheme のマクロでは，パターンマッチに基づく構文変換が実現できる．本書では Scheme のマクロの基本的な部分について例を挙げながら説明する．

マクロを定義するために define-syntax がある．また，構文変換を行うための機構として syntax-rules がある．

《マクロ定義の概略》

```
(define-syntax マクロ名
        (syntax-rules (予約後の列)
                (パターン 1    変換後の式 1)
                (パターン 2    変換後の式 2)
                        :
                (パターン n    変換後の式 n)
        )
)
```

「パターン」に一致する式を「変換後の式」の形に変換して評価する．

これを用いたマクロ定義の例を macro01.scm に示す．

プログラム：macro01.scm

```
1 (define-syntax 反復
2     (syntax-rules (条件 回数 処理)
3         ((_ 条件 p 処理 s ...)
4                 (do ((r 0)) ((not p) r) (set! r (let () s ...))))
5         ((_ 回数 n 処理 s ...)
6                 (let ((r 0)) (dotimes (v n r) (set! r (let () s ...))))))
7     )
8 )
```

これは反復処理のための構文を独自に定義したものであり，(load "./macro01.scm") などとして処理系に読み込んだ後，次のような式を評価することができる．

例. 条件指定の反復処理

```
(define n 0)  Enter        ←変数 n に値を設定
→  n
(反復 条件 (< n 4) 処理 (print n) (set!  n (+ n 1)) "end")  Enter  ←独自の構文の式
→  0          ← print による出力
→  1                  :
→  2
→  3
→  "end"          ←戻り値
```

例. 回数指定の反復処理

```
(反復 回数 3 処理 (print "Scheme!") "end")  Enter  ←独自の構文の式
→  Scheme!          ← 1 回目の出力
→  Scheme!
→  Scheme!          ← 3 回目の出力
→  "end"          ←戻り値
```

■ macro01.scm の解説

3 行目と 5 行目にパターンの定義がある．パターンの記述の中のシンボルは**パターン変数**と呼ばれるものであり，これを用いて変換後の式を記述する．例えば 3 行目には p，s という 2 つのパターン変数があり，これらを用いて 4 行目にある変換後の式を記述する．また，ドット 3 つ「...」は直前のパターン変数に続くもの全てを意味し，変換後の式の中の「...」に対応させることができる．パターンの先頭にあるアンダースコア「_」は当該マクロの名前を意味するものであり，この例においては「反復」というシンボルを意味する．

2 行目の syntax-rules 直後の (条件 回数 処理) は，パターン記述の中で使用する**予約語**であり，この例のマクロにおいてはこれら 3 つのシンボルはパターン変数としてではなく，構文を構成するための語 として扱われる．

7.8.1 マクロの展開

Common Lisp と同様に macroexpand が使用できる．

例. マクロの展開（先の例の続き）

```
(macroexpand '(反復 条件 (< n 4) 処理 (print n) (set!  n (+ n 1)) "end"))  Enter
→  (do ((r.0 0)) ((not (< n 4)) r.0)          ←↓展開された式
→    (set!  r.0 (let () (print n) (set!  n (+ n 1)) "end")))
```

7.9 データの型

注意）ここで解説する内容は Gauche 処理系独自のものが多く含まれる．

Scheme の**データ型**は階層構造を持つ．最上位の型は <top> である．型名は <…> で括った形で表記される．最下位の型である <bottom> もあり，これは自身以外のいかなる型よりも下位にある型である．

《型の階層》 基本的なもの

```
<top>
    <boolean>                  真理値型
    <symbol>                   シンボル
        <keyword>              キーワード
    <char>                     文字（文字列を構成する要素）
    <collection>
        <sequence>
            <list>             リスト
                <pair>         空でないリスト
                <null>         空リスト
            <vector>           ベクトル
            <string>           文字列
        <dictionary>           辞書
            <hash-table>       ハッシュ表
    <number>                   数値
        <complex>              複素数
            <real>             実数
                <rational>     有理数
                    <integer>  整数
                                    …  <bottom>
```

Scheme では，オブジェクトの型の検査に「**型名?**」が使用できる．

例. 型の検査

```
(symbol?  'a)  Enter        ←シンボルか？
→  #t          ←検査結果

(list?  '(a b c))  Enter    ←リストか？
→  #t          ←検査結果
```

オブジェクトの型を取得するには class-of を使用する．

例. 型の取得

```
(class-of 'a)  Enter        ← 'a の型を求める
→  #<class <symbol>>        ←シンボル型

(class-of 123)  Enter       ← 123 の型を求める
→  #<class <integer>>       ←整数型
```

上の例で得られた結果に対して class-name を使用すると型名が得られる．

例. 型名の取得

```
(class-name (class-of 'a))  Enter      ← 'a の型名を求める
→  <symbol>          ←型名が得られた
```

オブジェクトがある型かどうかを検査するには is-a? を使用する.

例. 型の検査 (2)

```
(is-a?  'a <symbol>)   Enter        ← 'a はシンボルか？
→ #t          ←検査結果

(is-a?  123 <symbol>)   Enter       ← 123 はシンボルか？
→ #f          ←検査結果
```

型の階層の上下関係は subtype? で調べる.

例. 型の上家関係の検査

```
(subtype?  <integer> <number>)   Enter        ←<integer>は<number>に属するか？
→ #t          ←「属する」
```

7.10　例外処理

注意）ここで解説する内容は Gauche 処理系独自のものである.

本書では Gauche の例外処理に関する初歩的な事柄について説明する.

《例外処理の概略》

```
(guard (変数
          (条件 1    処理 1)
          (条件 2    処理 2)
            :
          (else      処理 n)
        )
    式の列…)
```

例外の発生が想定される「式の列」を評価し，例外が発生した場合は，その**状態**を表すオブジェクトが「変数」に与えられる．それを受けて,「条件 1」,「条件 2」, … に一致するものに対応する「処理」を行う．どの条件にも一致しない場合は `else` に対応する「処理 n」を行う．例外が発生しない場合は「式の列」の戻り値を返す.

■ 条件の記述

Gauche では，例外によって発生する状態は次のような階層をなす.（文献 [11]）

《状態の階層》

```
<condition>
    +- <compound-condition>
    +- <serious-condition>
    |     +- <serious-compound-condition> ; also inherits <compound-condition>
    +- <message-condition>
          +- <error>                      ; also inherits <serious-condition>
                +- <system-error>
                +- <unhandled-signal-error>
                +- <read-error>
                +- <io-error>
                      +- <port-error>
                            +- <io-read-error>
                            +- <io-write-error>
                            +- <io-closed-error>
                            +- <io-unit-error>
```

それぞれの状態に関する詳細は文献 [11] を参照のこと.

例えば，例外の状態を保持する変数 `e` が `<read-error>`（データの読み込みに関するエラー）かどうかを判定する条件の式は次のように記述する.

```
(<read-error> e)
```

例. `(/ 1 0)` による例外をハンドリングする

```
(guard (e   Enter
       ((<error> e) (print "error") e)   Enter
       (else (print "other") e))   Enter
   (/ 1 0))   Enter
```

```
→  error          ← print による出力
→  #<error "attempt to calculate a divisio">        ←戻り値
```

参考文献

[1] “On Lisp”, Paul Graham（著）, 野田 開（著, 訳）, オーム社, 2007 年

[2] “Common Lisp the Language”,
Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb,
Digital Press, 1984, ISBN 0-932376-41-X

[3] “Common Lisp the Language, 2nd Edition”, Guy L., Steele Jr.
Digital Press, 1990; ISBN 1-55558-041-6,
(`https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/clm.html`), by Guy L., Steele Jr.

[4] “Common Lisp HyperSpec”
(`http://www.lispworks.com/documentation/HyperSpec/Front/index.htm`), Kent Pitman

[5] “SBCL 2.0.0 User Manual” (`http://www.sbcl.org/manual/`)

[6] “初めての人のための LISP” [増補改訂版], 竹内郁雄（著）, 翔泳社, 2010 年

[7] “The Calculi of Lambda Conversion”, Alonzo Church, Princeton University Press, 1941 (1985)

[8] “Scheme - Revised(5) Report on the Algorithmic Language Scheme”
(`http://people.csail.mit.edu/jaffer/r5rs_toc.html`,
`http://www.swiss.ai.mit.edu/ jaffer/r5rsj_toc.html`（日本語）),
R.Kelsey, W.Clinger, J.Rees (Editors)

[9] “The Revised6 Report on the Algorithmic Language Scheme” (`http://www.r6rs.org/`),
M.Sperber, R.K.Dybvig, M.Flatt, A.V.Straaten (Editors)

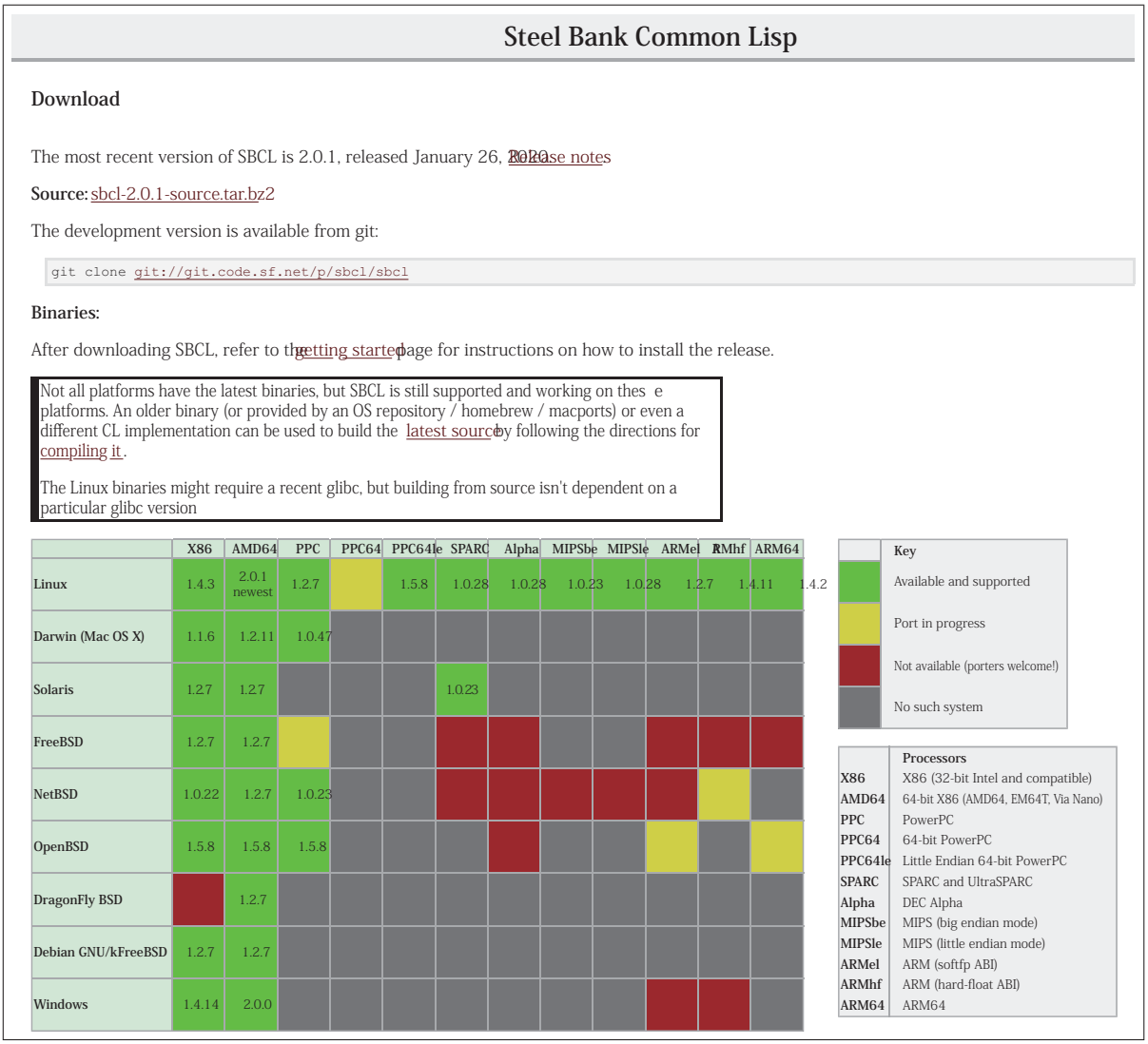
[10] “R7RS Home Page” (http://www.r7rs.org/)

[11] “Gauche - A Scheme Implementation” (`http://practical-scheme.net/gauche/index-j.html`),
川合 史朗

付録

A　SBCL（Steel Bank Common Lisp）の入手

SBCL（Steel Bank Common Lisp）はソースコード，バイナリインストーラの形で入手することができる．本書執筆時点では，下記の Web ページ `http://www.sbcl.org/platform-table.html` （図 9）からダウンロードできる．



Steel Bank Common Lisp

Download

The most recent version of SBCL is 2.0.1, released January 26, 2020. Release notes

Source: sbcl-2.0.1-source.tar.bz2

The development version is available from git:

```
git clone git://git.code.sf.net/p/sbcl/sbcl
```

Binaries:

After downloading SBCL, refer to the getting started page for instructions on how to install the release.

Not all platforms have the latest binaries, but SBCL is still supported and working on these platforms. An older binary (or provided by an OS repository / homebrew / macports) or even a different CL implementation can be used to build the latest source by following the directions for compiling it.

The Linux binaries might require a recent glibc, but building from source isn't dependent on a particular glibc version

	X86	AMD64	PPC	PPC64	PPC64le	SPARC	Alpha	MIPSbe	MIPSle	ARMel	ARMhf	ARM64
Linux	1.4.3	2.0.1 newest	1.2.7		1.5.8	1.0.28	1.0.28	1.0.23	1.0.28	1.2.7	1.4.11	1.4.2
Darwin (Mac OS X)	1.1.6	1.2.11	1.0.47									
Solaris	1.2.7	1.2.7				1.0.23						
FreeBSD	1.2.7	1.2.7										
NetBSD	1.0.22	1.2.7	1.0.23									
OpenBSD	1.5.8	1.5.8	1.5.8									
DragonFly BSD		1.2.7										
Debian GNU/kFreeBSD	1.2.7	1.2.7										
Windows	1.4.14	2.0.0										

Key	
	Available and supported
	Port in progress
	Not available (porters welcome!)
	No such system

Processors	
X86	X86 (32-bit Intel and compatible)
AMD64	64-bit X86 (AMD64, EM64T, Via Nano)
PPC	PowerPC
PPC64	64-bit PowerPC
PPC64le	Little Endian 64-bit PowerPC
SPARC	SPARC and UltraSPARC
Alpha	DEC Alpha
MIPSbe	MIPS (big endian mode)
MIPSle	MIPS (little endian mode)
ARMel	ARM (softfp ABI)
ARMhf	ARM (hard-float ABI)
ARM64	ARM64

図 9: SBCL のダウンロードページ

このページには，OS と CPU アーキテクチャのクロス表があり，バージョン番号の部分がインストーラへのリンクににになっている．

索 引

「Lisp 入門」 − Common Lisp / Scheme

著者：中村勝則

発行：2020 年 2 月 29 日

テキストの最新版と更新情報

本書の最新版と更新情報を，プログラミングに関する情報コミュニティ Qiita で配信しています.

→ https://qiita.com/KatsunoriNakamura/items/9c56bfc1fd882685c149

上記 URL の QR コード

● 連絡先

nkatsu2012@gmail.com

中村勝則