X3J13 Document 88-002R
June 1988

Common Lisp Object System Specification

1. Programmer Interface Concepts

Authors: Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel,
Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: June 15, 1988

For information about obtaining the sources for this document, send an Internet message to common-lisp-object-system-specification-request@sail.stanford.edu.

The authors wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions to this document.

At the X3J13 meeting on June 15, 1988, the following motion was adopted:

"The X3J13 Committee hereby accepts chapters 1 and 2 of the Common Lisp Object System, as defined in document 88-002R, for inclusion in the Common Lisp language being specified by this committee. Subsequent changes will be handled through the usual editorial and cleanup processes."

X3J13 Document 88-002R
June 1988

Common Lisp Object System Specification

第1章　プログラマインタフェースの概念

著者：Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel,
Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: June 15, 1988

この文書のソースを入手するための情報は common-lisp-object-system-specification-request@sail.stanford.edu に電子メイルを送ることによってえられる．

The authors wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions to this document.

At the X3J13 meeting on June 15, 1988, the following motion was adopted:

「X3J13 委員会は 88-002R で定義された Common Lisp Object System の第1章および第2章を，本委員会で扱う Common Lisp 言語に含めることをアクセプトする．以後の変更は通常の編集・クリーンアップの過程を経て行なわれる．」

目　次

1. 序

Common Lisp Object System（略して CLOS）は，*Common Lisp: The Language* (Guy L. Steele Jr.）に定義されている Common Lisp に対するオブジェクト指向機能拡張であり，総称関数，多重継承，宣言的メソッド結合，メタオブジェクトプロトコルを基本としている．

本仕様書の最初の２章では CLOS の標準プログラマインタフェースについて解説する．第１章では CLOS の概念を，第２章ではプログラマインタフェースを構成する関数やマクロの説明を行なう．「CLOS メタオブジェクトプロトコル」の章（本訳書には含まれていない）では CLOS 上に既存のオブジェクト指向パラダイムや新たなパラダイムを定義するための方法について述べる．

CLOS の基本的なオブジェクトはクラス (class)，インスタンス (instance)，総称関数 (generic function)，メソッド (method) である．

クラスオブジェクトは，**インスタンス**と呼ぶオブジェクトの構造や振舞いを決定する．すべての Common Lisp オブジェクトはあるクラスのインスタンスである．オブジェクトのクラスはそのオブジェクトに対して実行できる操作の集合を決定する．

総称関数は，与えられた引数のクラスまたは引数の一致に依存した振舞いをする関数である．総称関数オブジェクトはメソッドの集合，ラムダリスト，メソッド結合方式，その他の情報から構成される．**メソッド**は総称関数にクラス特有の振舞いと操作を定義する．したがって，メソッドは総称関数を**特定化**（***specialize***）するという．総称関数が呼び出されると，与えられた引数のクラスに基づいて，その総称関数に属するメソッドの部分集合を実行する．

総称関数は Common Lisp における通常の関数と同様な方法で使用できる．特に funcall や apply への引数として使え，また大域的あるいは局所的な名前を与えられる．

メソッドは，メソッド関数，与えられたメソッドが適用されるケースを規定する**引数特定子**（***parameter specializer***）の並び，**メソッド結合**機能がメソッドを区別するのに用いる**修飾子**（***qualifier***）の並びから構成されるオブジェクトである．各メソッドの必須引数は，対応する引数特定子をもっていて，メソッドはその引数特定子を満足する引数が与えられたときだけ実行される．

メソッド結合機能はメソッドの選択や実行順序，総称関数が返す値を制御する．CLOS はデフォルトのメソッド結合方式を提供しており，また，新しいメソッド結合方式を宣言するための機能も提供している．

2. エラー用語

エラー状態を表わすために本仕様書で用いている用語は，*Common Lisp: The Language* (Guy L. Steele Jr.）で用いられている用語とは異なっている．本仕様書では**状態**（***situation***：ある特殊な文脈における式の評価のこと）という用語を用いる．たとえば，ある指定された制約を満たさない引数で

関数が呼び出されたというのは1つの状態である.

CLOSの仕様では，すべての状態におけるプログラムの振舞いが記述されており，処理系作成者が選択できるオプションを定義している．各処理系は，オブジェクトシステムの仕様で明確に拡張を許すと書いていない限り CLOS の文法や意味を拡張することは許されない．特に，CLOS の文法との間で曖昧性が生じる可能性がある拡張は許されない．

「状態Sが生じたとき，エラーを発する.」
この用語は以下のような意味をもつ.

- この状態が生じたならば，インタプリタ上，およびコンパイラのすべての safety 最適化レベルのもとでコンパイルされたコードにおいて，エラーが発せられる.
- インタプリタ上，およびコンパイラのすべての safety 最適化レベルのもとでコンパイルされたコードにおいて，エラーが発せられることを前提としてプログラムを作成してよい.
- すべての処理系は，インタプリタ上，およびコンパイラのすべての safety 最適化レベルのもとでコンパイルされたコードにおいて，そのようなエラーを検出しなければならない.

「状態Sが生じたとき，エラーを発するべきである.」
この用語は以下のような意味をもつ.

- この状態が生じたならば，少なくともインタプリタ，およびコンパイラの最も安全な safety 最適化レベルのもとでコンパイルされたコードにおいてエラーが発せられる.
- エラーが発せられることを前提としてプログラムを作成してはならない.
- すべての処理系は，少なくともインタプリタ，およびコンパイラの最も安全な safety 最適化レベルのもとでコンパイルされたコードにおいて，エラーを検出しなければならない.
- エラーが発せられない場合，結果は未定義である．（以下を見よ）

「状態Sが生じたとき，結果は未定義である.」
この用語は以下の意味をもつ.

- この状態が生じたとき，結果は予測できない．無害なことも致命的なこともある.
- 処理系はこの状態を見いだしエラーを発してもかまわないが，この状態を検出することを強要されることはない.
- プログラムはこの状態によって引き起こされる結果に依存してはならない．すべてのプログラムはこの状態からどんな結果が生ずるかは予測不能として対処しなければならない.

「状態Sが生じたときの結果を規定しない.」
この用語は以下の意味をもつ.

- この状態の結果を CLOS では規定しないが，結果は無害である.
- 処理系はこの状態の結果を規定してもよい.
- ポータブルなプログラムはこの状態の結果に依存してはならない．すべてのポータブルなプログラムはこの状態からどんな結果が生ずるかは予測不能であるが，無害なものとして扱うことが要求される.

「CLOS を状態Sに対処するように拡張してもよい.」

この用語は，処理系は状態Sを次の3つのいずれかの方法で取り扱ってよいということである．

- 状態Sが生じたとき，少なくともインタプリタ上，およびコンパイルの最も安全な safety 最適化レベルのもとでコンパイルされたコードにおいて，エラーを発する．
- 状態Sが生じたときの結果は未定義とする．
- 状態Sが生じたときの結果を定義しておく．

さらに，この用語には以下の意味がある．

- ポータブルなプログラムはこの状態の影響に依存してはならない．そして，すべてのポータブルなプログラムはその状態を未定義なものとして取り扱わなければならない．

「処理系は構文Sを自由に拡張してよい．」

この用語は以下の意味をもつ．

- 処理系は構文Sに対して曖昧性のない拡張を自由に定義してよい．
- ポータブルなプログラムはこの拡張に依存してはならない．そして，すべてのポータブルなプログラムはこの構文を無意味なものとして取り扱う必要がある．

CLOS には，拡張してよい部分といけない部分がある．

3. クラス

クラスは，**インスタンス**と呼ぶオブジェクトの構造と振舞いを規定するオブジェクトである．

クラスは他のクラスから構造と振舞いを継承する．クラス定義において，継承を目的として参照しているクラスを，参照されているクラスの**サブクラス**といい，参照されているクラスを，参照しているクラスの**スーパークラス**という．

クラスは名前をもつことができる．関数 class-name はクラスオブジェクトを引数にとり，その名前を返す．名前の無いクラスに対しては nil を返す．シンボルはクラスの名前になりうる．関数 find-class はシンボルを引数にとり，それを名前にもつクラスを返す．クラスの名前がシンボルで，その名前によってクラスに名前がつけられているとき，そのクラスは**固有名**（***proper name***）をもつという．すなわち，クラスCが**固有名**Sをもつとは，$S=$ (class-name C) でありかつ，$C=$ (find-class S) ということである．(find-class S_1)＝(find-class S_2) で $S_1 \neq S_2$ ということも起こりうることに注意せよ．もし $C=$(find-class S) なら，CはSという名前のクラスであるという．

クラス C_2 がそのクラス定義においてクラス C_1 をスーパークラスとしていれば，C_1 を C_2 の**ダイレクトスーパークラス**と呼び，C_2 を C_1 の**ダイレクトサブクラス**と呼ぶ．各$1 \leq i < n$に対してクラス C_{i+1} がクラス C_i のダイレクトスーパークラスとなっているようなクラス $C_2, \ldots, C_{n-1}$ が存在するとき，クラス C_n をクラス C_1 の**スーパークラス**と呼び，クラス C_1 をクラス C_n の**サブクラス**と呼ぶ．クラスは自分自身のスーパークラスでもサブクラスでもない．すなわち，C_1 が C_2 のスーパークラスであるとき，$C_1 \neq C_2$ が成立する．クラスCとそのスーパークラスの集合を参照したいときには，「Cとそのスーパークラス」という言い方をする．

各クラスは，そのクラスとそのスーパークラスの集合上の全順序である**クラス優先順位リスト**（***class precedence list***）をもつ．全順序は**最も特定的**（***most specific***）なクラスから，**最も特定的でない**

(***least specific***) クラスへと並べたリストで表現される．クラス優先順位リストはさまざまな所で使われるが，一般的には，より特定的なクラスは，もしそれがなければ継承されていたであろうより特定的でないクラスの性質を**シャドウ**，つまり覆い隠してしまう．クラス優先順位リストは，メソッド選択や結合の過程でメソッドをより特定的なものからそうでないものの順に並べるために用いられる．

クラスを定義するとき，その定義のフォーム中で記述されているそのクラスのダイレクトスーパークラスの順序は重要である．各クラスは，**局所優先順位** (***local precedence order***) をもち，それはそのクラスの後ろにクラス定義の中で記述されているダイレクトスーパークラスをその順序で並べたリストである．

クラス優先順位リストはそのリスト中のクラスの局所優先順位に矛盾しない．局所優先順位中のクラスの順序は，クラス優先順位リストに現われる順序と同じである．局所優先順位が互いに矛盾を含むなら，クラス優先順位リストは構成されずエラーが発せられる．クラス優先順位リストとその計算の方法については「6. クラス優先順位リストの決定」で述べる．

クラス全体の集合は**閉路のない有向グラフ** (***directed acyclic graph***) をなしている．また，t と standard-object という名前の特別なクラスがあり，クラス t はスーパークラスをもたず，自分自身を除くすべてのクラスのスーパークラスになっている．クラス standard-object はクラス standard-class のインスタンスであり，自分自身を除く standard-class のインスタンスであるクラスすべてのスーパークラスである．

Common Lisp のクラス空間から CLOS の型空間の中への写像がある．また，*CLtL* に述べられている Common Lisp 標準の型の多くは対応する同名のクラスがあるが，対応するクラスがないものもある．Lisp の型とクラスの統合については「5. 型とクラスの統合」で述べる．

クラス自身もオブジェクトであり，あるクラスのインスタンスである．オブジェクトのクラスのクラスは，そのオブジェクトの**メタクラス** (***metaclass***) と呼ばれる．誤解がない場合には，そのクラスのインスタンスがクラスであるようなクラスをメタクラスと呼ぶ．メタクラスはそのインスタンスであるクラスの継承の仕方とそれらのクラスのインスタンスの表現を決定する．CLOS では多くのプログラムに便利なデフォルトのメタクラス standard-class を用意している．メタオブジェクトプロトコルにより，新しいメタクラスを定義し，使うことができる．

本章で述べるすべてのクラスは断わりがなければすべてクラス standard-class のインスタンスであり，すべての総称関数はクラス standard-generic-function のインスタンスであり，すべてのメソッドは standard-method のインスタンスである．

3.1 クラスの定義

名前をもつクラスはマクロ defclass を用いて定義される．defclass の構文を第2章図 1 (278ページ) に示す．

クラス定義には以下のものが含まれる．

- クラスの名前．新しく定義されたクラスに対してはこの名前は固有名となる．

- その新しいクラスのダイレクトスーパークラスのリスト.
- 幾つかの**スロット指定子** (***slot specifier***). 各スロット指定子はスロット名と0個以上の**スロットオプション**からなり，そのスロットオプションはそのスロットだけに影響する．もし，クラス定義に同じスロット名をもつ2つのスロット指定子があればエラーが発せられる.
- 幾つかの**クラスオプション**. 各クラスオプションはそのクラス全体に影響する.

スロットオプションとクラスオプションには以下の機能がある.

- スロットのデフォルト初期値フォームを与える.
- スロットを読み書きする総称関数のメソッドを自動的に生成するよう要求する.
- スロットをそのクラスのインスタンスで共有するか，個々のインスタンスで個別にもつかを制御する.
- インスタンス生成時に用いられる初期化引数とそのデフォルトを与える.
- インスタンスがデフォルトでないメタクラスをもつように指示する.
- スロットに設定される値の期待される型を指定する.
- スロットに文書文字列を対応付ける.

3.2 インスタンスの生成

総称関数 make-instance は，クラスの新しいインスタンスを生成し，それを返す．CLOS は新しいインスタンスをどのように初期化するかを指示する幾つかの機構を用意している．たとえば，新しく作られたインスタンスのスロットの初期値は make-instance の引数として与えることにより，あるいは，デフォルト初期値により与えることができる．また，初期化プロトコルの一部である総称関数に対してメソッドを定義することにより一層の制御が可能である．初期化プロトコルの詳細は「10. オブジェクトの生成と初期化」で述べる.

3.3 スロット

standard-class をメタクラスにもつオブジェクトは0個以上の名前付きスロットをもつ．オブジェクトのスロットは，そのオブジェクトのクラスによって決定される．各スロットは1個の値をもつことができる．スロットの名前は Common Lisp の変数名として構文的に有効なシンボルである.

スロットが値をもたないとき，そのスロットは**未束縛** (***unbound***) であるという．未束縛のスロットが読まれたときには，総称関数 slot-unbound が呼ばれる． slot-unbound に対してシステムが提供する基本メソッドはエラーを発する.

スロットのデフォルト初期値フォームはスロットオプション:initform で定義される． 初期値を与えるために:initform が使われたならば， そのフォームは defclass フォームが評価されたレキシカルな環境で評価される．defclass フォームが評価されたレキシカルな環境をもつ:initform を，**捕捉された** (***captured***) :initform という．詳細は「10. オブジェクトの生成と初期化」を参照のこと.

局所スロット (***local slot***) はそのスロットが割り付けられた個々のインスタンスからしか見えないスロットとして定義される． **共有スロット** (***shared slot***) は与えられたクラスとそのサブクラスのイ

ンスタンスから見えるスロットとして定義される．

あるクラスの defclass フォームがある名前のスロット指定子を含むとき，そのクラスはその名前をもつスロットを定義しているという．局所スロットを定義してもすぐにはそのスロットは作られず，そのクラスのインスタンスが作られるたびにそのスロットが作られる．共有スロットを定義するとすぐにそのスロットが作られる．

defclass のスロットオプション：allocation でどちらの種類のスロットを定義するかを制御できる．スロットオプション：allocation の値が：instance なら局所スロットが作られ，：class なら共有スロットが作られる．

スロットがそのインスタンスのクラスで定義されているか，あるいはそのクラスのスーパークラスから継承しているとき，そのスロットはそのインスタンス中で**アクセス可能**（***accessible***）であるという．あるインスタンス中では，与えられた名前のスロットは高々1つだけがアクセス可能である．あるクラスに定義された共有スロットは，そのクラスのすべてのインスタンスからアクセス可能である．スロットの継承の詳細な説明は「4.2 スロットとスロットオプションの継承」で述べる．

3.4 スロットのアクセス

スロットは関数 slot-value によって，あるいは defclass フォームで定義された総称関数によってアクセスできる．

関数 slot-value によって，defclass フォームに指定されたスロット名で，そのクラスのインスタンスのスロットをアクセスできる．

マクロ defclass はスロットを読み書きするメソッドを生成する構文を提供する．**リーダ**を作るように要求すると，スロットの値を読むためのメソッドが自動的に作られるが，そのスロットに値を入れるためのメソッドは作られない．また，**ライタ**を作るように要求すると，スロットに値を入れるためのメソッドが自動的に作られるが，そのスロットの値を読むためのメソッドは作られない．**アクセサ**を作るように要求すると，スロットの値を読むためのメソッドと，そのスロットに値を入れるためのメソッドが作られる．リーダとライタのメソッドは slot-value を用いて実現されている．

リーダとライタが個々のスロットに対して指定されたときには，それによって作られるメソッドがどの総称関数に属するかが直接指定される．もし，ライタオプションで指定された名前がシンボルならば，そのスロットに値を入れるための総称関数はそのシンボルを名前としてもち，新しい値とインスタンスの2つをその順序で引数にとる．もし，アクセサオプションで指定された名前がシンボルならば，そのスロットの値を読むための総称関数はそのシンボルを名前としてもち，そのスロットに値を入れるための総称関数はリスト（setf〈シンボル〉）を名前としてもつ．

リーダ，ライタあるいはアクセサのスロットオプションによって作られたか修正された総称関数は通常の総称関数として取り扱われる．

あるスロットに対するリーダやライタのメソッドがなくても，そのスロットの値を読み書きするのに slot-value を使えることに注意せよ．slot-value はリーダやライタのメソッドを呼び出さない．

マクロ with-slots によって，あるスロットがあたかも変数のようにレキシカルにアクセス可能となるレキシカルな環境を作ることができる．マクロ with-slots は，スロットをアクセスするときに，関数 slot-value を呼ぶ．

マクロ with-accessors によって，あるスロットがあたかも変数のようにレキシカルにアクセス可能となるレキシカルな環境を作ることができる．マクロ with-accessors は，スロットをアクセスするときに，適切なアクセサを呼ぶ．with-accessors によって指定されたアクセサは，それが使われる前に定義されなければならない．

4. 継　　承

クラスはメソッド，スロット，defclass のオプションの幾つかをスーパークラスから継承することができる．この節ではメソッドの継承，スロットオプションの継承，クラスオプションの継承について説明する．

4.1 メソッドの継承

あるクラスのインスタンスに適用可能なメソッドは，そのクラスのサブクラスのインスタンスにも適用可能であるという意味でサブクラスはメソッドを継承する．

メソッドの継承は，メソッドがメソッド定義フォームを用いて作られたものであろうと，メソッドを自動的に生成する defclass のオプションを用いて作られたものであろうと同じように行なわれる．

メソッドの継承は「8. メソッド選択と結合」で詳しく述べる．

4.2 スロットとスロットオプションの継承

クラスCのインスタンス中でアクセス可能なスロット名の集合は，Cとそのスーパークラスで定義されたスロットの名前の集合の和集合である．インスタンスの構造は，そのインスタンスのもっている局所スロットの集合で決まる．

最も単純なのは，Cとそのスーパークラスの中のただ1つのクラスだけが，与えられたスロット名に対するスロットを定義している場合である．もしスロットがCのスーパークラスで定義されているならば，そのスロットは継承されているという．スロットの特性は，そのスロットを定義しているクラスに記述されたスロット指定子により決定される．スロットSを定義しているクラスを考えよう．もしそのスロットオプション：allocation が：instance ならば，Sは局所スロットであり，そのクラスとそのすべてのサブクラスのインスタンスに，そのスロットのための記憶領域が割り付けられる．また，もしそのスロットオプション：allocation が：class ならば，Sは共有スロットであり，Sを定義したクラスにそのスロットのための記憶領域が割り付けられる．そして，すべてのCのインスタンスはそのスロットだけをアクセスすることができる．もし，スロットオプション：allocation が省略されたならば，：instance が用いられる．

一般には，あるスロット名のスロットは，Cとそのスーパークラス中の複数のクラスで定義することができる．その場合，Cのインスタンス内では，ある名前のスロットは高々1つだけがアクセス可能である．そして，そのスロットの性質は以下に計算されるように複数のスロット記述を組み合わせたものとなる．

- 与えられたスロット名をもつすべてのスロット指定子はそれらを定義しているクラスCのクラス優先順位リストに従って，最も特定的なものから最も特定的でないものへの順に並べられる．
- スロットの割付けは最も特定的なスロット指定子によって決定される．もし最も特定的なスロット指定子がスロットオプション：allocation を含まないならば，：instance が用いられる．それより特定的でないスロット指定子はそのスロットの割付けに影響を及ぼさない．
- スロットのデフォルト初期値フォームはスロットオプション：initform を含む最も特定的なスロット指定子の：initform の値が使われる．もし：initform を含むスロット指定子がなければ，そのスロットはデフォルト初期値フォームをもたない．
- スロットの内容は常に（and $T_1 \ldots T_n$）という型をもつ．ここで，$T_1, \ldots, T_n$ はすべてのスロット指定子に含まれるスロットオプション：type の値である．もし：type を含むスロット指定子がなければ，そのスロットの内容は常に型 t をもつ．そのスロットの型を満たさない値をそのスロットに入れようとしたときの結果は未定義である．
- スロットを初期化する初期化引数の集合はすべてのスロット指定子のスロットオプション：initarg で宣言された初期化引数の和集合である．
- スロットの文書文字列は，スロットオプション：documentation をもつ最も特定的なクラスに記述された値である．もしどのスロット指定子も：documentation を含んでいなければ，そのスロットは文書文字列をもたない．

割付け規則の結果として，共有スロットはシャドウされることがある．たとえば，クラス C_1 が S という名前のスロットを定義していてそのスロットオプション：allocation の値が：class であれば，そのスロットは C_1 とそのサブクラスのインスタンス中でアクセス可能である．しかし，C_2 が C_1 のサブクラスであり，C_2 もまた S という名前のスロットを定義しているなら，C_1 のスロットは C_2 とそのサブクラスのインスタンスでは共有されない．クラス C_1 が共有スロットを定義しているとき，C_1 のすべてのサブクラス C_2 は，C_2 の defclass フォームが同じ名前のスロットを指定しているか，あるいは C_2 のクラス優先順位リスト中に同じ名前のスロットを定義している C_1 よりも特定的な C_2 のスーパークラスがあるのでなければ，そのスロットを C_1 と共有する．

型規則の結果として，スロットの値はそのスロットに関係するすべてのスロット指定子の型の制限を満たしている．しかし，そのスロットに対する型の制限を満たさない値をそのスロットに入れようとしたときの結果は未定義なので，スロットの値はそのスロットの型の制限を満たさないようになるかもしれない．

スロットオプション：reader, ：writer あるいは ：accessor はスロットの性質を定義するというよりはむしろメソッドを生成する．リーダとアクセサのメソッドは「4.1 メソッドの継承」で述べられているような意味で継承される．

スロットをアクセスするメソッドはスロットの名前とスロットの値の型だけを用いる．ある名前の共有スロットをアクセスするメソッドをスーパークラスが与えて，サブクラスが同じ名前の局所スロットを定義しているとしよう．この場合，もしスーパークラスで与えられているメソッドがサブクラスのインスタンスに用いられたならば，そのメソッドは局所スロットをアクセスする．

4.3 クラスオプションの継承

クラスオプション :default-initargs は継承される．クラスのデフォルト初期化引数の集合とは，そのクラスとそのスーパークラスのクラスオプション :default-initargs に指定された初期化引数の集合の和集合である．もしある初期化引数に対して複数のデフォルト初期値フォームが与えられたならば，デフォルト初期値フォームとして使われるのはクラス優先順位リスト中で最も特定的なクラスによって与えられたものである．

もし1つの :default-initargs で同じ名前の初期化引数を2回以上指定したならば，エラーが発せられる．

4.4 例

```
(defclass C1 ()
   ((S1 :initform 5.4 :type number)
    (S2 :allocation :class)))

(defclass C2 (C1)
   ((S1 :initform 5 :type integer)
    (S2 :allocation :instance)
    (S3 :accessor C2-S3)))
```

クラス C1 のインスタンスは S1 という名前の局所スロットをもち，そのデフォルト初期値は 5.4 でその値は常に number 型である．また C1 は S2 という名前の共有スロットをもつ．

C2 のインスタンスには S1 という名前の局所スロットがある．S1 のデフォルト初期値は 5 で，S1 の値は (and integer number) 型である．また C2 のインスタンスには S2 と S3 という名前の局所スロットがある．クラス C2 にはスロット S3 の値を読むための C2-S3 に対するメソッドがある．また，S3 の値を書くための (setf C2-S3) に対するメソッドもある．

5. 型とクラスの統合

CLOS は，クラス空間から Common Lisp の型空間の中へ写像をもっている．固有名をもつすべてのクラスには同じ名前の対応する型がある．

おのおののクラスの固有名は型指定子として有効である．また，すべてのクラスオブジェクトも型指定子として有効である．したがって，式 (typep *object class*) は，もし *object* のクラスが *class* で

あるかあるいは *class* のサブクラスである場合，真と評価される．式（subtypep *class*1 *class*2）の評価は *class*1 が *class*2 のサブクラスであるかまたは，それらが同じクラスであれば値として t t を返す．それ以外は値として nil t を返す．名前が S というクラス C のインスタンスを I とし，C が standard-class のインスタンスであるならば，式（type-of I）の評価は，S が C の固有名であるならば S を返す．一方 S が C の固有名でなければ式（type-of I）は C を返す．

クラスの名前とクラスオブジェクトは型指定子なので，それらは特殊形式 the の中や型宣言の中で用いることができる．

あらかじめ定義されている Common Lisp の型指定子の多くは（しかし，すべてではない）対応する同じ固有名のクラスをもっている．これらの型指定子を図1に示す．たとえば，array 型は対応するクラス array をもっている．リストになっている型指定子，たとえば（vector double-float 100）は対応するクラスをもたない．式 deftype はいかなるクラスも作り出さない．

あらかじめ定義された Common Lisp の型指定子に対応するおのおのの クラスは，処理系に応じて3つの方法のうちの1つで実現される．これらは **標準クラス**（***standard class*** : defclass で定義される），**構造体クラス**（***structure class*** : defstruct で定義される），**組込みクラス**（***built-in class*** : 特殊な，拡張できない方法で実現される）である．

組込みクラスはそのインスタンスが制限された機能あるいは特殊な表現をもっているものである．組込みクラスのサブクラスを defclass を用いて定義しようとするとエラーを発する．組込みクラスのインスタンスを作り出すために make-instance を呼び出すと エラーを発する．組込みクラスのインスタンスに対して slot-value を呼び出すとエラーを発する．組込みクラスを再定義しようとしたり，あるいはあるインスタンスのクラスを組込みクラスに（から）変更するために change-class を呼び出そうとするとエラーを発する．しかし，組込みクラスはメソッドの引数特定子として使える．

メタクラスを調べることでクラスが組込みクラスであるかどうか決定できる．標準クラスは standard-class のインスタンスであり，組込みクラスは built-in-class のインスタンスであり，構造体クラスは structure-class のインスタンスである．

:type オプションを使わずに defstruct によって生成されるおのおのの構造体型は，対応するクラスをもっている．このクラスは structure-class のインスタンス である．defstruct の :include オプションは対応するクラスのダイレクトサブクラスを生成する．

Common Lisp の標準の型指定子の多くが対応するクラスをもっているとすることの目的は，プログラマがこれらの型により区別されるメソッドを書けるようにするためである．メソッド選択は，おのおののクラスについてクラス優先順位リストが決定できることを必要とする．

Common Lisp の型指定子の階層的関係は それらの型に対応するクラスの関係に そのまま反映される．あらかじめ定義されている Common Lisp の型の間に存在する 階層関係は，対応する おのおののクラスのクラス優先順位リストを決定するために用いられる．幾つかの場合，*Common Lisp: The Language* は，与えられた型指定子の2つのスーパー型に対する局所優先順位を規定していない．たとえば，null はシンボルおよびリストの副型であるが，*Common Lisp: The Language* はシンボルとリ

ストのどちらがより特定的であるかについて規定していない．CLOS の仕様はこのような場合における関係をすべてのクラスに対して定義している．

図1はオブジェクトシステムが必要とするクラスの集合をまとめたものである．各クラスのスーパークラスは最も特定的なものから最も一般的なものへと並べられているので，そのクラスのクラス優先順位リストを定義していることになる．おのおのの型指定子の局所優先順位はこの図から導き出すことができる．

既定義の Common Lisp 型	対応するクラスのクラス優先順位リスト
array	(array t)
bit-vector	(bit-vector vector array sequence t)
character	(character t)
complex	(complex number t)
cons	(cons list sequence t)
float	(float number t)
integer	(integer rational number t)
list	(list sequence t)
null	(null symbol list sequence t)
number	(number t)
ratio	(ratio rational number t)
rational	(rational number t)
sequence	(sequence t)
string	(string vector array sequence t)
symbol	(symbol t)
t	(t)
vector	(vector array sequence t)

図 1

おのおのの処理系は他の型指定子が対応するクラスをもつように定義を拡張してもよい．また，*Common Lisp: The Language* に述べられている型の関係や disjoint 性の要求に抵触しない限り，この表におけるクラス優先順位リストに他の要素を加えたり他のサブクラスの関係を加えたりする拡張は許される．ダイレクトスーパークラスをもたないように定義された標準クラスは，t というクラスを除くすべてのクラスと disjoint であることが保証されている．以下の Common Lisp の型は，もし Common Lisp がこれらの型を cons, symbol, array, number, character と disjoint として定義するように変更されるならば，対応するクラスをもてるようになる．

- function
- hash-table
- package
- pathname
- random-state
- readtable
- stream

6. クラス優先順位リストの決定

defclass フォームにより，クラスとそのクラスのダイレクトスーパークラスの全順序が与えられる．この順序は**局所優先順位**(***local precedence order***)と呼ばれる．これはクラスとそのクラスのダイレクトスーパークラスの順序付きリストである．クラスCに対する**クラス優先順位リスト**(***class precedence list***)は，クラスCとそのスーパークラスをそれぞれ局所優先順位に矛盾を生じないように並べた全順序である．

クラスはそのダイレクトスーパークラスに優先し，おのおののダイレクトスーパークラスは defclass フォームのスーパークラスリストの中でそれより右に位置する他のダイレクトスーパークラスよりも優先する．すべてのクラスCに対し

$$R_C=\{(C, C_1),\ (C_1, C_2), \ldots, (C_{n-1}, C_n)\}$$

を定義する．ここで，$C_1, \ldots, C_n$ は defclass フォームの中で記述されている順序に従ったCのダイレクトスーパークラスである．これらの順序対は，クラスCとそのクラスのダイレクトスーパークラスの全順序を生成する．

Cとそのスーパークラスの集合を S_C とし，またRを次のように定義する．

$$R=\bigcup_{c\in S_C} R_c$$

Rが半順序を形成するかどうかは，R_c, $c\in S_C$, に矛盾がないかどうかに依存するが，ここでは，矛盾がなく，Rは半順序を形成するものと仮定する．R_c が無矛盾でないとき Rに矛盾があるということにする．

Cのクラス優先順位リストを計算するために，Rによって形成される半順序を用いて S_C の要素をトポロジカルにソートする．その際，Rの中に他のクラスに優先されていないクラスが2個以上ある場合，これらから1つを選び出すには，以下に示す方法であいまいのないように選択する．

もしRに矛盾があれば，エラーが発せられる．

6.1 トポロジカルソーティング

トポロジカルソーティングは，次のようにして行なう．まず S_C の中から，Rの要素の中で他のクラスが優先しないようなクラスCを見つけ出し，結果の先頭にCを置く．S_C からCを除きRの中から(C, D), $D\in S_C$, であるすべての組を削除する．S_C の中で優先クラスをもたないクラスを結果の最後に追加する処理を繰り返す．優先クラスをもたない要素が見つからなくなった時点で処理を終える．

もし S_C が空でないのに処理が終了した場合は，Rは矛盾している．もしクラスの有限集合中のすべてのクラスが他のどれかのクラスに優先されているならば，Rはループを含んでいる．すなわち，C_iが$C_{i+1}, 1\leqq i<n$，に優先し，C_n が C_1 に優先するようなクラスの連鎖 $C_1, \ldots, C_n$ が存在する．

S_C の中で優先クラスをもたないクラスが幾つか存在する場合がある．このとき，次に選ばれるクラスはそのクラスのダイレクトサブクラスが，そこまでに作られたクラス優先順位リスト上で最も右にあるものである．ダイレクトスーパークラスは右側のダイレクトスーパークラスより優先するので，このような候補はただ1つしかない．もしそのような候補がなければ，Rは半順序を形成しない．つまり，Rc, $c \in S_C$, は矛盾を含んでいる．

以下に詳しく述べる．$\{N_1, ..., N_m\}$, $m \geqq 2$, を優先クラスをもたない S_C の要素であるとする．$(C_1 ... C_n)$, $n \geqq 1$, をこれまでに構成されたクラス優先順位リストとしよう． C_1 は最も特定的なクラスであり，C_n は最も特定的でないクラスである． j, $1 \leqq j \leqq n$, を，ある i, $1 \leqq i \leqq m$, に対して N_i が C_j のダイレクトスーパークラスになるものの中で最も大きい数とする．そのとき，N_i が次にクラス優先順位リストに加えられるクラスである．

優先クラスをもたないクラスの集合から次にクラス優先順位リストに加えられるクラスを選び出すためのこのルールは，単純なスーパークラスの連鎖をなすクラスをクラス優先順位リストにおいて隣接して並べ，相互に分離したサブグラフをなすクラスはクラス優先順位リストにおいても隣接して並べる効果がある．たとえば，クラス J を唯一共通の要素にもつ T_1 と T_2 というサブグラフを考える．T_1, T_2 のいずれにもクラス J のスーパークラスは存在しないものとする． C_1 を T_1 の底とし C_2 を T_2 の底とする．C がそのダイレクトスーパークラスに C_1 と C_2 をこの順でもつクラスとすれば，C に対するクラス優先順位リストはCから始まりその後にクラス J を除く T_1 の中のすべてのクラスが続き，次に T_2 のすべてのクラスが続く．クラス J とそのスーパークラスは最後に置かれる．

6.2 例

クラス pie のクラス優先順位リストを決定する次の例を考える．クラスを次のように定義する．

```
(defclass pie (apple cinnamon) ())

(defclass apple (fruit) ())

(defclass cinnamon (spice) ())

(defclass fruit (food) ())

(defclass spice (food) ())

(defclass food () ())
```

集合S, Rは次のようになる．

S = {pie, apple, cinnamon, fruit, spice, food, standard-object, t}

R = {(pie, apple), (apple, cinnamon), (apple, fruit), (cinnamon, spice),
(fruit, food), (spice, food), (food, standard-object),
(standard-object, t)}

クラス pie は他のいずれのクラスにも優先されていないので，まずこれが最初にくる．ここまでの

結果は (pie) である．Sから pie を，Rから pie を含む組を削除する．

集合S，Rは次のようになる．

S={apple, cinnamon, fruit, spice, food, standard-object, t}

R={(apple, cinnamon),(apple, fruit),(cinnamon, spice),(fruit, food),
(spice, food),(food, standard-object),(standard-object, t)}

クラス apple は他のいずれのクラスにも優先されていないので，apple を選ぶ．結果は (pie apple) となる．Sから apple を，Rから apple を含む組を削除する．

集合S，Rは次のようになる．

S={cinnamon, fruit, spice, food, standard-object, t}

R={(cinnamon, spice), (fruit, food), (spice, food), (food, standard-object), (standard-object, t)}

クラス cinnamon と fruit は他のいずれのクラスにも優先されていない．そこでこの2つのうち，これまで計算してきたクラス優先順位リストの中で，より右側にダイレクトサブクラスをもっているほうを次に並べる．クラス apple は fruit のダイレクトサブクラスであり，クラス pie は cinnamon のダイレクトサブクラスである．クラス優先順位リストにおいて apple が pie より右側にあるので，クラス fruit が次に選ばれ，結果は (pie apple fruit) となる．

集合S，Rは次のようになる．

S={cinnamon, spice, food, standard-object, t}

R={(cinnamon, spice),(spice, food),(food, standard-object),(standard-object, t)}

クラス cinnamon が次にくる．結果は (pie apple fruit cinnamon) となり，この時点で，集合S，Rは次のようになる．

S={spice, food, standard-object, t}

R={(spice, food),(food, standard-object),(standard-object, t)}

クラス spice, food, standard-object, t がこの順で結果に加えられ，結局，クラス優先順位リストは，

```
(pie apple fruit cinnamon spice food standard-object t)
```

となる．

順序付けできないクラス定義を書くことも可能である．たとえば，

```
(defclass new-class (fruit apple) ())
```

```
(defclass apple (fruit) ())
```

new-class の定義内でスーパークラスリストでの局所順位は保たれねばならないので，クラス fruit は，クラス apple より優先する．一方 apple の定義により，クラス apple は，fruit に優先する．この状態が生じたときは，システムがクラス優先順位リストを計算しようとしたときにエラーを発する．

一見すれば矛盾を起こしていそうな次の例を考える．

```
(defclass pie (apple cinnamon) ())

(defclass pastry (cinnamon apple) ())

(defclass apple () ())

(defclass cinnamon () ())
```

pie に対するクラス優先順位リストは (pie apple cinnamon standard-object t) となる．

pastry に対するクラス優先順位リストは (pastry cinnamon apple standard-object t) となる．

pie のスーパークラスの順序では apple は cinnamon に優先しており，pastry のそれでは逆であるがこれは問題ではない．しかし，pie と pastry の両方をスーパークラスにもつような新しいクラスの定義はできない．

7. 総称関数とメソッド

総称関数は，与えられた引数のクラスまたは引数の一致に依存した振舞いをする関数である．**メソッド**は総称関数に引数のクラスに特有の振舞いと操作を定義する．この節では総称関数とメソッドについて説明する．

7.1 総称関数の概要

総称関数オブジェクトは，メソッドの集合，ラムダリスト，**メソッド結合方式** (***method combination type***) その他の情報から構成される．

総称関数は，通常の Lisp 関数と同様に，引数を渡され，一連のオペレーションを行ない，そして一般に何らかの値を返す．通常の関数はボディを１つもっていて，呼び出されると常にそれを実行する．これに対して総称関数はボディの集合をもっていて，呼び出されるとそのうちの何個かを選んで実行する．実行されるボディの選び方とそれらの結合の仕方は，総称関数に渡される引数のクラスや引数の一致そしてメソッド結合方式によって決まる．通常の関数も総称関数も同じ構文を使って呼び出される．

総称関数は，引数として渡したり，funcall および apply の第一引数として用いることができるの

で，真に関数であるといえる．

Common Lisp では次の２つの方法で関数に名前をつけることができる． defun で大域的な名前を，そして特殊形式 flet または labels で局所的な名前をつけられる．総称関数には defmethod または defgeneric で大域的な名前を，そして特殊形式 generic-flet, generic-labels, または with-added-methods で局所的な名前をつけられる．通常の関数と同様に，総称関数の名前の形式は，シンボルまたは最初の要素が setf で２番目の要素がシンボルからなるリストである．このことは局所的な名前であるか大域的な名前であるかによらない．

マクロ generic-flet は局所的な総称関数を生成する．この総称関数に属するメソッドは generic-flet 内のメソッド定義で作られる．generic-flet 内のフォームからの総称関数の見え方は flet の場合と同様である．

特殊形式 generic-labels は互いに再帰的に呼べる局所的な総称関数を生成する． この総称関数に属するメソッドは generic-labels 内のメソッド定義で作られる．generic-labels 内のフォームからの総称関数の見え方は labels の場合と同様である．

特殊形式 with-added-methods は，１つの名前と複数のメソッド定義を与え，それらを，同名のレキシカルに見えていた総称関数のメソッドのコピーに加え，新しい総称関数を局所的に生成する．もし同名の通常の関数が見えていた場合は，その関数は新しく定義される局所的な総称関数のデフォルトメソッドのメソッド関数になる．

特殊形式 generic-function は，その中に定義しているメソッドをもった名前無しの総称関数を生成する．

defgeneric が評価されたときは，次の３つのうちのいずれかの動作が行なわれる．

- もし同名の総称関数がすでにあれば，前からあった総称関数オブジェクトは修正される．以前の defgeneric フォームによって定義されていたメソッドは削除され，新しい defgeneric フォーム内に定義されているメソッドが加えられる．defgeneric で追加されるメソッドによって，以前に defmethod または defclass で定義されていたメソッドが置き換えられる場合もある． その総称関数に属する他のメソッドは，これによって影響を受けたり，置き換えられたりすることはない．
- もし定義しようとしている総称関数の名前が，通常の関数，マクロ，または特殊形式であれば，エラーを発する．
- 上の２つ以外の場合は，defgeneric 内に定義されたメソッドをもつ総称関数が生成される．

メソッド結合方式や引数優先順位などの総称関数のオプションを指定できるフォームを「総称関数のオプションを指定するフォーム」と呼ぶ．このようなフォームとしては，defgeneric, generic-function, generic-flet, generic-labels, with-added-methods がある．

総称関数にメソッドを定義できるフォームを「メソッド定義フォーム」と呼ぶ．このようなフォームとしては，defgeneric, defmethod, generic-function, generic-flet, generic-labels, with-added-methods, defclass がある．これらの中で defclass と defmethod 以外のフォームは，総称関数にオプションを指定することもできるので，「総称関数のオプションを指定するフォーム」でもある点に注意せ

よ.

7.2 メソッドの概要

メソッドオブジェクトは，メソッド関数，どのような場合にメソッドが適用可能であるかを指示する**引数特定子**（***parameter specializer***）の並び，ラムダリスト，そしてメソッド結合機能がメソッドを区別するために用いる**修飾子**（***qualifier***）の並びをもっている.

メソッドオブジェクトは関数ではなく，関数として呼び出すことはできない．CLOS の中ではメソッドオブジェクトからメソッド関数を求めて，それを起動している．総称関数が起動されるときもこの方法が用いられる．このようなとき，メソッドが起動されたとか，メソッドが呼び出されたという.

メソッド定義フォームは，総称関数への引数により選ばれたメソッドが起動されたときに実行するコードをもっている．メソッド定義フォームが評価されると，メソッドオブジェクトが生成され，次の4つのうちのいずれかの動作をする.

- もし同名の総称関数がすでにあって，定義しようとしているメソッドと引数特定子と修飾子において一致するメソッドオブジェクトがあれば，古いメソッドオブジェクトは新しいメソッドオブジェクトと置き換えられる．あるメソッドが別のメソッドと引数特定子と修飾子において一致するということの意味は，「7.3 引数特定子と修飾子に関する一致」を参照せよ.
- もし同名の総称関数がすでにあって，定義しようとしているメソッドと引数特定子と修飾子において一致するメソッドオブジェクトが無い場合は，新しいメソッドオブジェクトを含むように総称関数オブジェクトは修正される.
- もし同名の通常の関数，マクロまたは特殊形式があれば，エラーを発する.
- 上の3つ以外の場合は，メソッド定義フォームにより定義されたメソッドをもった総称関数が生成される.

もし，新しいメソッドのラムダリストが総称関数のラムダリストと**適合**（***congruent***）しなければ，エラーを発する．もし，総称関数のオプションを指定できないメソッド定義フォームが総称関数を新しく作った場合は，メソッドのラムダリストと適合するようなラムダリストが総称関数のラムダリストになる．適合性についての議論は，「7.4 1つの総称関数に属するすべてのメソッドの適合ラムダリスト」を参照せよ.

それぞれのメソッドは**特定化されたラムダリスト**（***specialized lambda-list***）をもっており，これによりどんなときにそのメソッドが適用されるかが決定される．特定化されたラムダリストは通常のラムダリストと似ているが，必須引数の名前の代わりに**特定化された引数**（***specialized parameter***）を置いてもよい．特定化された引数は（*variable-name* 引数特定子名）のようなリストであり，引数特定子名は次のいずれかである.

- クラスにつけられた名前
- (eql *form*)

引数特定子名は次のように引数特定子を表わす.

- クラス名はクラスオブジェクトを表わす.
- リスト（eql *form*）は（eql *object*）を表わす．ここで *object* は *form* を評価した結果である．*form* はメソッド定義フォームが評価されるレキシカルな環境で評価される．このとき *form* はメソッドが定義されたときに一度だけ評価されるのであり，総称関数が呼び出されるたびに評価されるのではない点に注意せよ.

引数特定子の名前は，defmethod のようにユーザが使うインタフェースとなることを意図したマクロで用いられ，一方，引数特定子は関数間のインタフェースとして用いられる.

必須引数だけを特定化することができる．必須引数それぞれに引数特定子がついていなければならない．記述を簡単化するために，メソッド定義フォームの特定化されたラムダリストの中の必須引数が単に変数名であるものは，引数特定子としてtというクラス名が省略されていると見なす.

総称関数と引数が与えられたとき，**適用可能メソッド**（***applicable method***）とは，総称関数に属するメソッドの中で，それぞれの引数が対応する引数特定子を満足するものである．次の定義は，メソッドが適用可能であるとはどういう意味か，引数が引数特定子を満足するとはどういうことかを示す.

総称関数の必須引数が $\langle A_1, \ldots, A_n \rangle$ であり，メソッド M の各引数に対応する引数特定子が $\langle P_1, \ldots, P_n \rangle$ であるとする．メソッド M はそれぞれの A_i が P_i を満足するときに適用可能であるという．もし P_i がクラスであり A_i がクラス C のインスタンスであれば，$C = P_i$ であるかまたは C が P_i のサブクラスであるときに，A_i は P_i を満足するという．もし P_i が（eql *object*）であれば，A_i が *object* と eql であるときに，A_i は P_i を満足するという.

引数特定子は型指定子なので，メソッド選択において引数が引数特定子を満足するかどうかの判定に，関数 typep を用いることができる．ただし，一般に，(vector single-float) のような型指定子リストは引数特定子として使用できない．リスト形式の引数特定子は（eql *object*）だけである．この提案は，Common Lisp に次のような定義の eql を型指定子として含めるよう修正することを要求している.

```
(deftype eql (object) `(member ,object))
```

すべての引数特定子が t という名前のクラスであるメソッドは，**デフォルトメソッド**（***default method***）と呼ばれ，常に適用可能であるが，通常は，より特定的なメソッドによってシャドウされる.

メソッドは**修飾子**をもつことができ，メソッド結合の手続きはこれを用いてメソッドを区別する．1個またはそれ以上の修飾子をもつメソッドを，**修飾されたメソッド**（***qualified method***）と呼ぶ．これに対して，修飾子をもたないメソッドを，**修飾されていないメソッド**（***unqualified method***）と呼ぶ．修飾子は，リストでない任意の Lisp オブジェクト，すなわち nil でないアトムである．標準メソッド結合方式と組込みメソッド結合方式で定義されている修飾子はシンボルである.

本仕様書では，メソッド結合における使用目的に応じてメソッドを分けるために，**基本メソッド**（***primary method***）と**補助メソッド**（***auxiliary method***）という用語を用いた．標準メソッド結合では，基本メソッドは修飾されていないメソッドを表わし，補助メソッドは :around, :before, または

:after のうちのいずれか1つを修飾子にもつメソッドを表わす．短形式を用いて定義されたメソッド結合方式では，基本メソッドとはメソッド結合方式の名前と同じ名前の修飾子をもつメソッドのことであり，補助メソッドは修飾子 :around をもっている．このように，基本メソッドや補助メソッドという用語は，単に与えられたメソッド結合方式に依存するような意味しかもたない．

7.3 引数特定子と修飾子に関する一致

次の条件を満たすとき，2つのメソッドは引数特定子と修飾子に関して一致しているという．

1．2つのメソッドは同じ個数の必須引数をもっていなければならない．それぞれのメソッドの引数特定子が $P_{1,1}...P_{1,n}$ と $P_{2,1}...P_{2,n}$ であるとしよう．
2．各 $1 \leqq i \leqq n$ について $P_{1,i}$ と $P_{2,i}$ は一致しなければならない．引数特定子 $P_{1,i}$ と $P_{2,i}$ が一致するとは，$P_{1,i}$ と $P_{2,i}$ が同じクラスであるが，あるいは $P_{1,i}$=(eql *object1*)，$P_{2,i}$=(eql *object2*) であって (eql *object1* *object2*) が成り立つ場合である．それ以外の場合は一致しない．
3．2つの修飾子のリストは nil でない同じアトムを同じ順序でもっていなければならない．すなわち，2つのリストは equal でなければならない．

7.4 1つの総称関数に属するすべてのメソッドの適合ラムダリスト

以下の規則は，総称関数およびその総称関数に属するメソッドのラムダリストの適合性を定義する．

1．それぞれのラムダリストは同じ個数の必須引数をもっていなければならない．
2．それぞれのラムダリストは同じ個数の付加引数をもっていなければならない．ただしそれらのデフォルト値はメソッドごとに異なっていてもよい．
3．もしどれかのラムダリストが &rest または &key をもっていれば，それぞれのラムダリストはそれらのうちの一方あるいは両方を使っていなければならない．
4．もし総称関数のラムダリストが &key をもっていれば，それぞれのメソッドはそこにあるすべてのキーワード名を陽に受け付けるようにするか，&allow-other-keys を使うか，あるいは &rest を指定しなければならない．それぞれのメソッドは独自のキーワード引数を受け付けるようにしてもよい．キーワード名の有効性のチェックは総称関数において行なわれ，メソッドにおいては行なわれない．メソッドはキーワード :allow-other-keys とその値 t をもっているかのようにして起動される．しかしこのキーワードは実際の引数としては渡されない．
5．&allow-other-keys の使われ方に関しては，ラムダリストの間で制限はない．もし適用可能ないずれかのメソッドまたは総称関数のラムダリストで &allow-other-keys を指定していた場合は，その総称関数は任意のキーワード引数で呼び出せる．
6．&aux の使い方に関しては，ラムダリストの間で制限はない．

もし，総称関数のオプションを指定できないメソッド定義フォームが総称関数を生成した場合は，メソッドがキーワード引数を使っていれば，生成された総称関数のラムダリストも &key を使ったものになる．ただしこの場合，総称関数のラムダリストにはキーワード引数はつかない．

7.5　総称関数とメソッドのキーワード引数

総称関数あるいはその総称関数に属するメソッドがラムダリストに &key をもっている場合，総称関数が受け付けるキーワード引数は適用可能なメソッドの選ばれ方によって変わる．ある呼出しにおいて，総称関数が受け付けるキーワード引数は，すべての適用可能メソッドのキーワード引数と総称関数のラムダリストのキーワード引数の和集合である．&rest はもっているが &key をもっていないメソッドは，受付け可能なキーワード引数がどんなものであるかには影響を与えない．もし適用可能メソッドのいずれかあるいは総称関数の定義に &allow-other-keys が含まれていれば，その総称関数の呼出しには任意のキーワード引数を与えることができる．

総称関数に属するすべてのメソッドは，総称関数に定義された &key 以降のキーワードをすべて受け付けなければならない．これは，それぞれのキーワードを陽に指定するか，&allow-other-keys を指定するか，あるいは &key を用いずに &rest を指定することで達成できる．また各メソッドは総称関数のラムダリストにあるキーワード以外にも独自のキーワードを受け付けることができる．

総称関数に渡されたキーワード引数を受け付ける適用可能メソッドが1つもないとエラーを発する．

たとえば，次のような width という2つのメソッドがあったとする．

```
(defmethod width ((c character-class) &key font)...)

(defmethod width ((p picture-class) &key pixel-size)...)
```

これら以外には width というメソッドも総称関数の定義もないとしよう．このとき，次のフォームを評価するとエラーを発する．

```
(width (make-instance 'character-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

次のフォームを評価してもエラーを発する．

```
(width (make-instance 'picture-class :glyph (glyph #\Q))
       :font 'baskerville :pixel-size 10)
```

しかし，character-picture-class というクラスがクラス picture-class とクラス character-class の両方のサブクラスであれば，次のフォームを評価してもエラーにならない．

```
(width (make-instance 'character-picture-class :char #\Q)
       :font 'baskerville :pixel-size 10)
```

8.　メソッド選択と結合

総称関数はある特定の引数で呼ばれたとき，実行するコードを決定しなければならない．このコードはその引数に対する**実効メソッド**（***effective method***）と呼ばれる．実効メソッドは，総称関数の中の

適用可能なメソッドの組合せである．メソッドの組合せは，メソッドの幾つかまたはすべての呼出しを含む Lisp 式である．総称関数が呼ばれたが，メソッドが1つも適用されなかったとき，総称関数 no-applicable-method が呼ばれる．

実効メソッドが決定されると，総称関数が受け取った引数と同じ引数で実効メソッドが呼ばれる．実効メソッドの返す値が総称関数の返す値となる．

8.1 実効メソッドの決定

実効メソッドは以下の3段階の手続きを経て決定される．

1．適用可能なメソッドを選ぶ．
2．適用可能なメソッドを，優先順位に従い，最も特定的なメソッドを先頭にするようにソートする．
3．適用可能なメソッドをソートしたリストに対してメソッド結合を適用して，実効メソッドを作り出す．

8.1.1 適用可能メソッドの選択

この手続きについては，「7.2 メソッドの概要」で説明する．

8.1.2 優先順位による適用可能メソッドのソート

2つのメソッドの優先順位を比較するには，両者の引数特定子を順に検査する．デフォルトの検査順序は左から右であるが，defgeneric または総称関数のオプションを指定する他のフォームに：argument-precedence-order オプションを与えることによって異なる検査順序を指定できる．

各メソッドの対応する引数特定子が比較される．対となる引数特定子が等しいときは，次の一対に移り，その等価性を比較する．対応する引数特定子がすべて等しいならば，2つのメソッドは異なる修飾子をもたねばならない．修飾子だけが異なる場合は，どちらのメソッドを優先して選択してもよい．

もし，対応する引数特定子に等しくないものがあれば，等しくない引数特定子の最初の一対が順位を決定する．両方の引数特定子がクラスならば，2つのメソッドのうち，より特定的なメソッドは，引数特定子がクラス優先順位リストにおいて先に出現したメソッドである．適用可能メソッドを決定する方法から，両方のメソッドの引数特定子は引数のクラスのクラス優先順位リストに存在することが保証される．

片方の引数特定子が（eql *object*）であるなら，その引数特定子をもつメソッドのほうが，他のメソッドより優先する．両方の引数特定子が eql フォームなら，それらは等しくなければならない．（そうでなければ，2つのメソッドがこの引数に対し共に適用可能とはならないであろう．）

結果として得られる適用可能メソッドのリストは，最も特定的なメソッドを先頭に，また最も特定的でないメソッドを最後にもつ．

8.1.3　適用可能メソッドをソートしたリストに対するメソッド結合の適用

標準メソッド結合を用い，かつ適用可能メソッドがすべて基本メソッドであるような単純な場合は，実効メソッドは最も特定的なメソッドだけである．そのメソッドは関数 call-next-method を使用して，次に特定的なメソッドを呼び出すことができる．この呼び出されるメソッドは**次メソッド**(***next method***)と呼ばれる．述語 next-method-p は，次メソッドが存在するか否かを調べる．call-next-method が呼ばれたが，次に特定的なメソッドが無かった場合は，総称関数 no-next-method が起動される．

一般には，実効メソッドは，適用可能なメソッドを組み合わせたものであり，Lisp フォームとして定義される．この Lisp フォームは，幾つかあるいはすべての適用可能メソッドを呼出し，総称関数の値となる値を返し，また call-next-method を用いて幾つかのメソッドをアクセスすることもできる．この Lisp フォームは実効メソッドのボディであり，これを関数にするために，適当なラムダリストが付け加えられる．

実効メソッド内での各メソッドの役割はメソッド修飾子とメソッドの特定度によって決まる．メソッド修飾子は，メソッドにつけられた印として働く．メソッド修飾子の意味は，この印がメソッド結合手続きでどのように使われるかによって決まる．適用可能メソッドに承認できない修飾子があると，このステップはエラーを発し，そのメソッドを実効メソッドに含めない．

標準メソッド結合が，修飾されたメソッドと共に用いられたときは，「8.2 標準メソッド結合」で述べるような方法で実効メソッドを生成する．

defgeneric または，総称関数オプションを指定する他のフォームの :method-combination オプションを用いて，別方式のメソッド結合を選択できる．これによって，このステップを独自化できる．

新しい方式のメソッド結合は，マクロ define-method-combination を用いて定義できる．

メタオブジェクトのレベルでも，新しいメソッド結合方式を定義する機構が用意されている．総称関数 compute-effective-method は，総称関数，メソッド結合オブジェクト，適用可能メソッドのソートされたリストを引数として受け取り，実効メソッドを定義する Lisp フォームを返す．compute-effective-method のメソッドは，defmethod により直接的に，または define-method-combination により間接的に定義することができる．メソッド結合オブジェクトは，メソッド結合方式，そして総称関数のオプションを指定するフォームに対する :method-combination オプションの値をカプセル化したオブジェクトである．

処理系に対する注：

最も簡単な処理系では，総称関数が呼ばれるたびに実効メソッドを計算するであろう．実際問題として，これでは効率が悪すぎる処理系があるだろう．その代わり，それらの処理系では，さまざまな最適化を上に述べた3段階の各手続きにおいて行なうことになろう．最適化の例を以下に示す．

- 引数のクラスをキーとしたハッシュテーブルに，実効メソッドを格納する．

- 実効メソッドをコンパイルし，コンパイル済みの関数をテーブルに保存する．
- その Lisp フォームを制御構造のパターンのインスタンスと見なし，その制御構造を実現するクロージャに置き換える．
- 総称関数のすべてのメソッドの引数特定子を検査し，すべての可能な実効メソッドを列挙する．それらの実効メソッドと，これらの中からの選択処理を行なうコードとを結合して1つの関数とし，それをコンパイルする．そして総称関数が呼ばれたときにはその関数を呼び出すようにする．

8.2 標準メソッド結合

標準メソッド結合は，クラス standard-generic-function によってサポートされる．他の方式のメソッド結合が指定されないか，または，組込みメソッド結合方式 standard が指定されていたなら，標準メソッド結合が用いられる．

基本メソッドは，実効メソッドの主要な働きを決定する．一方，**補助メソッド**は，3通りの方法でこの働きを修正する．基本メソッドはメソッド修飾子をもたない．

補助メソッドは，メソッド修飾子が :before, :after, :around であるようなメソッドである．標準メソッド結合では，メソッドごとに1つだけ修飾子を許す．もしメソッド定義で1つのメソッドに2つ以上の修飾子を指定すると，エラーが発せられる．

- :before メソッドは，キーワード :before を唯一の修飾子としてもつ． :before メソッドは，基本メソッドより先に実行されるコードを指定する．
- :after メソッドは，キーワード :after を唯一の修飾子としてもつ． :after メソッドは，基本メソッドより後に実行されるコードを指定する．
- :around メソッドは，キーワード :around を唯一の修飾子としてもつ． :around メソッドは，他の適用可能メソッドの代わりに実行されるコードを指定する．しかし，この中から他のメソッドの幾つかを実行させることもできる．

標準メソッド結合の意味を以下に示す．

- もし :around メソッドがあれば，最も特定的な :around メソッドが呼ばれ，それが総称関数の値または多値を返す．
- :around メソッドのボディの中では，次メソッドを呼ぶために call-next-method を使用できる．次メソッドから復帰したとき，:around メソッドは，返された値に基づきさらに実行を続けてもよい．call-next-method が使われたのに，呼ぶべき適用可能メソッドがないときは，総称関数 no-next-method が起動される．次メソッドが存在するか否かの判定には，関数 next-method-p を使えばよい．
- :around メソッドが call-next-method を実行したときは，適用可能な次に特定的な :around メソッドがあれば，それが呼ばれる．:around メソッドが1つもないか，最も特定的でない :around メソッドによって call-next-method が実行された場合は，他のメソッドが以下に示すように呼ばれる．

— 最も特定的なものを先頭にした順ですべての :before メソッドが呼ばれる．返ってきた値は無視される．:before メソッド内で call-next-method が使われると，エラーが発せられる．
— 最も特定的な基本メソッドが呼ばれる．基本メソッドのボディ内から次に特定的な基本メソッドを呼ぶには，call-next-method を用いる．そのメソッドから戻ったとき，最初の基本メソッドは，返された値に基づきさらに実行を続けてもよい．もし，call-next-method が実行されたのに，適用可能な基本メソッドがもうないならば，総称関数 no-next-method が起動される．次メソッドが存在するか否かの判定には，関数 next-method-p が使える．call-next-method が使用されなければ，最も特定的な基本メソッドだけが呼ばれる．
— 最も特定的でないものを先頭にする順で，すべての :after メソッドが呼ばれる．返ってきた値は無視される．:after メソッド内で call-next-method が使われると，エラーが発せられる．

• :around メソッドが1つも起動されないならば，最も特定的な基本メソッドの値または多値が総称関数の値となる．最も特定的でない :around メソッドで call-next-method を実行したときに返される値または多値は，最も特定的な基本メソッドが返した値である．

標準メソッド結合においては，適用可能メソッドはあるが，適用可能な基本メソッドが1つもない場合は，エラーが発せられる．

:before メソッドは特定的なものを先に実行するのに対して，:after メソッドは特定的でないものを先に実行する．この違いの設計理念について，例題で説明する．クラス C_1 が，そのスーパークラス C_2 の振舞いを :before および :after メソッドを定義することで変えるとしよう．その場合，C_2 の振舞いが C_2 について定義されたものであるか継承したものであるかは，クラス C_1 のインスタンスに対して起動されるメソッドの相対的実行順序に影響を与えない．クラス C_1 の :before メソッドは，クラス C_2 のすべてのメソッドの前に実行される．クラス C_1 の :after メソッドは，クラス C_2 のすべてのメソッドの後で実行される．

これに対して，:around メソッドは，他のどんなメソッドよりも前に実行される．したがって，より特定的でない :around メソッドは，より特定的な基本メソッドより前に実行される．

もし基本メソッドだけが使用され，call-next-method が用いられないならば，最も特定的なメソッドだけが実行される．これは，より特定的なメソッドがより一般的なメソッドをシャドウするということである．

8.3 宣言的メソッド結合

マクロ define-method-combination は新しい方式のメソッド結合を定義する．これは，実効メソッドの生成を独自化する機構を提供する．実効メソッドを生成するためのデフォルトの手続きについては「8.1 実効メソッドの決定」で述べている．define-method-combination には，2つの形式がある．短形式は簡単な機能だが，長形式はより強力でより複雑である．長形式は，そのボディが Lisp フォームを計算する式であるという点で defmacro と似ている．長形式は，メソッド結合において任意の制御構造を実現したり，メソッド修飾子を自由に処理するための機構を提供する．

define-method-combination の両形式の構文および使用法は，第2章で説明する．

8.4 組込みメソッド結合方式

CLOS は幾つかの組込みメソッド結合方式を提供する．ある総称関数がこれらのメソッド結合方式の1つを用いると指定するには，defgeneric または総称関数のオプションを指定する他のフォームの :method-combination オプションへの引数にそのメソッド結合方式の名前を指定すればよい．

組込みメソッド結合方式の名前は，+，and，append，list，max，min，nconc，or，progn，standard である．

標準組込みメソッド結合方式の意味は，「8.2 標準メソッド結合」で述べている．他の組込みメソッド結合方式は**単純組込みメソッド結合方式**と呼ばれる．

単純組込みメソッド結合方式は，define-method-combination の短形式でそれが定義されたかのように振舞う．これはメソッドの2つの役割を識別する．

- :around メソッドは唯一の修飾子としてキーワードシンボル :around をもつ．:around メソッドの意味は標準メソッド結合での意味と同じである．:around メソッドでは call-next-method と next-method-p を使用できる．
- 基本メソッドは唯一の修飾子として，メソッド結合方式の名前をもつ．たとえば，組込みメソッド結合方式 and は修飾子 and をもつメソッドを基本メソッドであると認識する．基本メソッドでは call-next-method と next-method-p を使用できない．

単純組込みメソッド結合方式の意味は以下のとおりである．

- もし :around メソッドがあれば，最も特定的な :around メソッドが呼ばれる．その返す値が総称関数の値となる．
- :around メソッドのボディ内では，次メソッドを呼ぶために call-next-method を使うことができる．call-next-method が使われたのに，呼び出すべき適用可能メソッドが1つも無いと，総称関数 no-next-method が起動される．次メソッドの存在を確かめるには next-method-p を使用できる．次メソッドから戻ったときに，:around メソッドは返された値に基づきさらに実行を続けることができる．
- :around メソッドが call-next-method を実行すると，適用可能な，次に最も特定的な :around メソッドがあれば，それが呼び出される．:around メソッドが1つも無いか，または最も特定的でない :around メソッドから call-next-method が呼ばれたなら，組込みメソッド結合方式名と適用可能な基本メソッドのリストから導き出した Lisp フォームが，総称関数の値を生成するために評価される．メソッド結合方式名が *operator* であり，総称関数の呼び出しフォームが以下のとおりだとする．

　　(*generic-function* $a_1 \ldots a_n$)

$M_1, \ldots, M_k$ が，この順序で適用可能な基本メソッドとする．そのとき，導かれる Lisp フォームは，次のようになる．

(*operator* $\langle M_1\ a_1...a_n\rangle$ ... $\langle M_k\ a_1...a_n\rangle$)

もし，式 $\langle M_i\ a_1...a_n\rangle$ が評価されると，メソッド M_i が引数 $a_1...a_n$ に適用される．もし *operator* が or ならば，式 $\langle M_i\ a_1...a_n\rangle$ は，$\langle M_j\ a_1...a_n\rangle$ が $1 \leqq j < i$ で nil を返すときだけ，評価される．

基本メソッドのデフォルトの順序は，最も特定的なものを先頭にした順序（:most-specific-first）である．しかし，:method-combination オプションへの第二引数で :most-specific-last を与えることにより順序を反転させることができる．

単純組込みメソッド結合方式は，メソッドごとに1個だけ修飾子を必要とする．修飾子をもたない適用可能メソッドがあったり，そのメソッド結合方式でサポートしていない修飾子があると，エラーが発せられる．適用可能な :around メソッドがあるのに適用可能な基本メソッドが1つも無いと，エラーが発せられる．

9. メタオブジェクト

オブジェクトシステムの処理系は，クラスとメソッドと総称関数を取り扱う．メタオブジェクトのプロトコルは，クラスを取り扱うメソッドにより定義された総称関数を定める．これらの総称関数の振舞いが，オブジェクトシステムの振舞いを定義する．これらのメソッドが定義されているクラスのインスタンスは**メタオブジェクト**と呼ばれる．メタオブジェクトプロトコルのレベルでのプログラミングは，メタオブジェクトの新しいクラスを，そのクラスに特定化されたメソッドと共に定義することにより行なわれる．

9.1 メタクラス

オブジェクトの**メタクラス**は，そのオブジェクトのクラスのクラスである．メタクラスは，そのメタクラスのインスタンスのインスタンスの表現および，スロット定義やメソッド継承のために，そのインスタンスが用いる継承の方式を定める．特定の最適化方式を提供したり，特定用途向けに CLOS を仕立てるために，メタクラス機構が使用できる．メタクラスを定義するためのプロトコルは，「CLOS メタオブジェクトプロトコル」で説明する．

9.2 標準メタクラス

CLOS は，幾つかのあらかじめ定義されたメタクラスを提供する．これには standard-class, built-in-class, structure-class がある．

- クラス standard-class は，defclass で定義されるクラスのデフォルトのクラスである．
- クラス built-in-class は，特別に実現された限られた能力しかもたないクラスをインスタンスとするメタクラスである．*CLtL* に定められた Common Lisp の型に対応するクラスは built-in-class のインスタンスとして実現されるかも知れない．対応するクラスをもつ必要のある，既定義の

Common Lisp 型指定子を図1 (233ページ) に示す．これらのクラスを組込みクラスとして実現するかどうかは，処理系依存である．

- defstruct を用いて定義されるすべてのクラスは，structure-class のインスタンスである．

9.3 標準メタオブジェクト

オブジェクトシステムは，**標準メタオブジェクト**と呼ばれる一連のメタオブジェクトを提供する．これには，クラス standard-object が含まれ，また，standard-method, standard-generic-function, そして method-combination のクラスのインスタンスが含まれる．

- クラス standard-method は，defmethod や defgeneric, generic-function, generic-flet, generic-labels, with-added-methods で定義されるメソッドのデフォルトのクラスである．
- クラス standard-generic-function は，defmethod, defgeneric, generic-function, generic-flet, generic-labels, with-added-methods, defclass で定義される総称関数のデフォルトのクラスである．
- standard-object という名前のクラスは，standard-class のインスタンスであり，また自分自身と structure-class を除く standard-class のインスタンスであるすべてのクラスのスーパークラスである．
- すべてのメソッド結合オブジェクトは，クラス method-combination のサブクラスのインスタンスである．

10. オブジェクトの生成と初期化

総称関数 make-instance はクラスから新しいインスタンスを生成してそれを返す．最初の引数はクラスまたはクラスの名前であり，残りの引数は**初期化引数リスト**（***initialization argument list***）である．

インスタンスの初期化は次のような処理からなる．陽に与えられた初期化引数と，与えられなかった引数のデフォルト値を組み合わせる．この引数が妥当であるかチェックする．インスタンスのためのメモリ領域を割り付ける．スロットに値を埋める．ユーザが初期化のために与えたメソッドを実行する．make-instance はこれらの各処理を総称関数を用いて実現しているので，ユーザは各ステップで独自な処理をすることができる．さらに make-instance 自体も総称関数なので，これにメソッドを定義して独自化することができる．

CLOS は各ステップにシステムが提供する基本メソッドを定義しているので，初期化の全過程に十分に定義された標準の振舞いを与えている．標準の振舞いは，初期化を制御できるように次の4つの簡単な機構を提供している．

- スロットの初期化引数となるシンボルを宣言する．これは defclass のスロットオプション :initarg で宣言される．これにより make-instance でスロットに初期値を与えることができる．
- 初期化引数のデフォルト初期値を計算するフォームを指定する．これは defclass のクラスオプション :default-initargs で指定する．もし初期化引数が make-instance の引数として陽に与えら

れなかったなら，デフォルト値フォームを defclass フォームが定義されたレキシカルな環境で評価し，その値を初期化引数の値として用いる．

- スロットにデフォルトの初期値を計算するフォームを指定する．これは defclass のスロットオプション :initform で定義する．このフォームは，make-instance の引数としても，またクラスオプション :default-initargs でも初期値を与えられなかったスロットに対して，このフォームを定義した defclass のレキシカルな環境で評価されスロットに設定される．局所スロットの :initform フォームは，インスタンス生成時，クラスの再定義に伴うインスタンスの更新時，そしてインスタンスのクラスの変更に伴うインスタンスの更新時に用いられる．共有スロットの :initform フォームはクラス定義時，そしてクラスの再定義時に用いられる．
- initialize-instance あるいは shared-initialize にメソッドを定義する．上に述べたスロットを埋める動作は，システム提供の基本メソッド initialize-instance によって実現されている．initialize-instance は shared-initialize を起動する．総称関数 shared-initialize は次の4つの状況で共通に用いられ，初期化処理の一部を担う．これらは，インスタンスの生成時，インスタンスの再初期化時，クラスの再定義に伴うインスタンスの更新時，インスタンスのクラスの変更に伴うインスタンスの更新時に用いられる．システム提供の基本メソッド shared-initialize が上に述べたスロットを埋める動作をする．initialize-instance は単に shared-initialize を起動するだけである．

10.1　初期化引数

初期化引数はオブジェクトの生成と初期化を制御する．初期化引数の名前には慣例的にキーワードシンボルを用いるが，実際は nil を含むどんなシンボルでもかまわない．初期化引数はスロットに値を埋めるため，あるいは初期化メソッドに引数を与えるために用いる．1つの初期化引数を両方の目的で使うこともできる．

初期化引数リストは初期化引数の名前と値が交互に現われるリストである．この構造は属性リストと同じであり，&key パラメータで処理される引数のリストの部分とも同じである．これらのリストと同じように，もし初期化引数リストに同じ名前のものが2つ以上ある場合は，最も左にあるものが値を与え，後にあるものは無視される．make-instance の引数の2番目以降にあるものは初期化引数リストである．もし初期化引数リストにキーワード引数 :allow-other-keys があってその値が nil 以外のものであれば初期化引数リストの有効性のチェックは行なわれない．

スロットに関連する初期化引数がある．初期化引数リストにスロットの初期化引数があれば，その値が新しく生成されたオブジェクトのスロットに代入される．初期化引数による指定はスロットに :initform で定義された初期値フォームより優先される．1つの初期化引数を用いて1個以上のスロットを初期化することができる．共有スロットを初期化する初期化引数は，その共有スロットの値を新しい値で置き換える．

メソッドに関連する初期化引数もある．オブジェクトが生成され初期化引数が与えられたとき，総称関数 initialize-instance, shared-initialize, そして allocate-instance は，その引数の名前と値をキーワード引数の組として渡され呼び出される．もし初期化の値が初期化引数リストで与えられなかった場合は，メソッドのラムダリストがデフォルト値を与える．

初期化引数は，インスタンス生成時，インスタンスの再初期化時，クラスの再定義時に伴うインスタンスの更新時，インスタンスのクラスの変更に伴うインスタンスの更新時の４つの場合に用いられる．

初期化引数はある特定のクラスのインスタンスの生成と初期化を制御するために使われるので，初期化引数は「そのクラスのための初期化引数」という言い方をする．

10.2 初期化引数の妥当性の宣言

初期化引数の妥当性のチェックは上記の４つの場合に行なわれる．１つの初期化引数がある場合には妥当であるが，別の場合には妥当でないということがある．たとえばクラス standard-class 上に定義されている，システム提供の make-instance の基本メソッドは，初期化引数の妥当性をチェックし，妥当であると宣言されていない初期化引数がある場合にエラーを発する．

初期化引数が妥当であると宣言する方法は２つある．

- スロットに値を埋めるための初期化引数は defclass のスロットオプション :initarg によって宣言される．スロットオプション :initarg はスーパークラスから継承される．したがって，あるクラスのスロットを埋めるために妥当な初期化引数は，そのクラスおよびクラスのスーパークラスで妥当と宣言されたスロットを埋めるための初期化引数の集合の和である．スロットを埋めるための初期化引数は４つの場合すべてで妥当である．
- メソッドの引数となる初期化引数の妥当性はメソッドを定義することで宣言される．メソッドのラムダリストに書かれたキーワード引数のキーワード名は，そのメソッドが適用可能なすべてのクラスの初期化引数になる．したがって，メソッドの引数として妥当な初期化引数の集合は，メソッドの継承によってコントロールされる．メソッドを定義することにより初期化引数の妥当性が宣言される総称関数には次のものがある．
  - — インスタンス生成に関するもの．allocate-instance, initialize-instance, そして shared-initialize. これらのメソッドで妥当と宣言された初期化引数は，クラスからインスタンスを生成するときに使える．
  - — インスタンスの再初期化に関するもの．reinitialize-instance と shared-initialize. これらのメソッドで妥当と宣言された初期化引数は，インスタンスを再初期化するときに使える．
  - — クラスの再定義に伴うインスタンスの更新に関するもの．update-instance-for-redefined-class と shared-initialize. これらのメソッドで妥当と宣言された初期化引数は，インスタンスを再定義されたクラスの定義に合うように更新するときに使える．
  - — インスタンスのクラスの変更に伴うインスタンスの更新に関するもの．update-instance-for-different-class と shared-initialize. これらのメソッドで妥当と宣言された初期化引数は，インスタンスのクラスを別のクラスに変えることに伴いインスタンスを新しいクラスの定義に合うように更新するときに使える．

あるクラスのインスタンスを初期化するのに妥当な初期化引数は，スロットの値を埋めるのに妥当な初期化引数，メソッドの引数として妥当な初期化引数，そしてあらかじめ初期化引数として妥当性を与えられている :allow-other-keys である．:allow-other-keys のデフォルト値は nil である．:allow-

other-keys の意味は通常の関数に渡されるときの意味と同じである．

10.3　初期化引数のデフォルト

初期化引数には**デフォルト初期値フォーム**(***default value form***)をクラスオプション :default-initargs で与えることができる．あるクラスで妥当と宣言された初期化引数に別のクラスでデフォルトフォームを与える場合がある．これは継承された初期化引数に :default-initargs でデフォルト値を与えているのである．

オプション :default-initargs は初期化引数にデフォルト値を与えるためだけに使え，新しく初期化引数名を宣言することはできない．また :default-initargs はインスタンスを生成するときに初期化引数にデフォルト値を与えるためだけに使える．

:default-initargs の引数は初期化引数名とフォームが交互に現われるリストである．それぞれのフォームは対応する初期化引数のデフォルト値を与える．このフォームが使われ評価されるのは，その初期化引数が make-instance の引数として渡されず，より特定的なクラスの :default-initargs でその初期化引数のデフォルト値が指定されていないときである．このフォームは，このフォームを定義した defclass の環境で評価され，その結果は初期化引数の値として使われる．

make-instance に与えられた初期化引数は，デフォルト初期化引数と組み合わされ，**デフォルト値付き初期化引数リスト**(***defaulted initialization argument list***)が作られる．デフォルト値付き初期化引数リストは初期化引数名と値が交互に現われるリストである．このリストの中では，指定されなかった初期化引数はデフォルト値が与えられ，陽に指定された引数はデフォルト初期化引数より先に現われる．デフォルト初期化引数は，それらのデフォルト値を与えたクラスのクラス優先順位リストの順序に従って並べられる．

:default-initargs と :initform は，スロットを初期化する目的が異なる．クラスオプション :default-initargs はユーザに初期化引数のデフォルト初期値フォームを指定させるのが目的で，この値がスロットを初期化するために使われるのか，メソッドに渡される引数として使われるのか関知しない．make-instance を呼び出すときにその初期化引数が指定されなかったときは，この初期値フォームが評価され，あたかも make-instance を呼び出す引数として指定されたかのように用いられる．これに対し，スロットオプション :initform はユーザにスロットのデフォルト初期値フォームを指定させるために使われる．:initform は，そのスロットの初期化引数が make-instance の引数としても :default-initargs でも与えられなかったときにスロットを初期化するために用いられる．

デフォルト初期値フォームと :initform フォームがどの順序で評価されるかは定義しない．もしこれらのフォームの評価の順序が影響を与えるような場合は，initialize-instance または shared-initialize にメソッドを定義して制御しなければならない．

10.4　初期化引数の規則

1つのスロットに2つ以上のスロットオプション :initarg を指定してもよい．

初期化引数の定義に重複があってもよいのは次の場合である.

- 1つの初期化引数を2つ以上のスロットを初期化するのに用いてもよい. これは別のスロットが同じ初期化引数名をもっている場合である.
- 1つの初期化引数名が2つ以上の初期化メソッドのラムダリストに現われてもよい.
- 1つの初期化引数名がスロットオプション :initarg と初期化メソッドのラムダリストの両方に現われてもよい.

もし make-instance の引数としてあるスロットを初期化する初期化引数が2個以上与えられたときは, 一番左にある初期化引数がスロットの値を与える. たとえ初期化引数名が異なっていても, スロットを初期化するのには最も左側にあるものが使われる.

もし, あるスロットの初期化引数が make-instance の引数として与えられずに, そのスロットを初期化するクラスオプション :default-initargs が幾つかあるときは, 最も特定化されたクラスに書かれている :default-initargs の値が用いられる. また, もし make-instance の引数に, あるスロットを初期化する引数が陽に与えられず, 1つの :default-initargs に同じスロットを初期化する異なった名前の引数があった場合は, 最も左に現われたものが使われ, その後に出て来る同じスロットを初期化する引数は無視される.

make-instance に陽に与えられた初期化引数は, :default-initargs によるデフォルト初期化引数よりも左に置かれる. クラス C_1 と C_2 が異なったスロットに対するデフォルト初期化引数をもっていて, クラス C_1 はクラス C_2 より特定的であるとしよう. このとき C_1 のもっているデフォルト初期化引数は C_2 のもっているデフォルト初期化引数より左に置かれたものがデフォルト値付き初期化引数リストになる. もし1つの :default-initargs が2つの別々のスロットに対する初期化引数の値をもっている場合は, :default-initargs の引数のうちでより左にあるものがデフォルト値付き初期化引数リスト上でもより左に出てくる. もしあるスロットが :initform と :initarg の両方のスロットオプションをもっていて, :default-initargs によりデフォルト初期化引数が与えられるか, make-instance に陽に引数として与えられた場合は, 捕捉された (captured) :initform のフォームは使われないし評価もされない.

次にこれらの規則の適用例を挙げる.

```
(defclass q () ((x :initarg a)))
(defclass r (q) ((x :initarg b))
  (:default-initargs a 1 b 2))
```

フォーム	デフォルト値付き初期化引数リスト	スロットXの値
(make-instance 'r)	(a 1 b 2)	1
(make-instance 'r 'a 3)	(a 3 b 2)	3
(make-instance 'r 'b 4)	(b 4 a 1)	4
(make-instance 'r 'a 1 'a 2)	(a 1 a 2 b 2)	1

10.5 shared-initialize

shared-initialize は，インスタンス生成時，インスタンスの再初期化時，クラスの再定義に伴うインスタンスの更新時，そしてインスタンスのクラスの変更に伴うインスタンスの更新時に初期化引数と:initform フォームを使ってインスタンスのスロットを埋める．shared-initialize は標準メソッド結合を用いる．shared-initialize の引数は，初期化されるインスタンス，そのインスタンスのアクセス可能なスロットの名前の集合の指定，そして任意個の初期化引数である．3番目以降の引数は初期化引数リストになっていなければならない．

shared-initialize の第二引数は次のいずれかである．

- スロット名のリスト．これはスロット名の集合を表わす．
- nil．これはスロット名の集合が空であることを表わす．
- t．これはインスタンスのもつすべてのスロットを表わす．

shared-initialize にはシステム提供の基本メソッドがあり，その第一引数の引数特定子はクラス standard-object である．このメソッドは，各スロットに対して，そのスロットが共有スロットであろうが，局所スロットであろうが，次のような処理をする．

- もし初期化引数リストにそのスロットの初期化引数が指定されていたら，スロットにその値を設定する．もしこのメソッドが走る前にスロットが値をもっていても値の設定は行なわれる．この動作は shared-initialize の第二引数にどんなスロットが指定されているかに影響されない．
- 第二引数に指定されていて，この時点で未束縛なスロットは，そのスロットの :initform フォームに従って初期化される．:initform フォームは defclass フォームが定義されたレキシカルな環境で評価され，その結果をスロットに設定する．:before メソッドなどによってすでに値の設定されているスロットについては，:initform フォームは使われない．もし第二引数にそのインスタンスではアクセスできないスロット名を指定していた場合，結果は未定義である．
- 「10.4 初期化引数の規則」に従う．

shared-initialize は，reinitialize-instance, update-instance-for-different-class, update-instance-for-redefined-class, initialize-instance に定義されたシステム提供の基本メソッドから呼ばれる．したがって，これらのどの総称関数が呼ばれたときにも実行したい処理があれば，shared-initialize にメソッドを定義すればよい．

10.6 initialize-instance

総称関数 initialize-instance は make-instance から呼ばれ，新しく生成されたインスタンスを初期化する．initialize-instance は標準メソッド結合を用いる．initialize-instance にメソッドを定義することにより，スロットに値を埋めるメカニズムだけではできないような任意の初期化処理をすることができる．

初期化の過程において，initialize-instance は次の処理が行なわれた後で起動される．

- 与えられた初期化引数と，クラスのもっているデフォルト初期化引数を組み合わせて，デフォルト

値付き初期化引数リストを計算する.

- デフォルト値付き初期化引数リストの妥当性をチェックする. もし妥当性を宣言していない引数があればエラーを発する.
- 値が未束縛 (unbound) のスロットをもつ新しいインスタンスが作られる.

次に, 新しいインスタンスとデフォルト値付き初期化引数リストを引数として総称関数 initialize-instance が呼び出される. initialize-instance には, クラス standard-object を引数特定子とするような, システムが提供する基本メソッドがある. このメソッドは総称関数 shared-initialize を呼ぶ. shared-initialize は初期化引数と :initform に従って各スロットに値を設定する. initialize-instance は, インスタンス, t, デフォルト値付き初期化引数として shared-initialize を呼び出す.

initialize-instance は, 引数としてデフォルト値付き初期化引数リストを shared-initialize に渡す点に注意せよ. つまり, システム提供の基本メソッド shared-initialize の第1ステップで行なう, 初期化引数に基づく初期化は, make-instance に指定した初期化引数だけでなく :default-initargs に記述されている初期化引数も使う.

initialize-instance にメソッドを定義することにより, インスタンスを初期化するときの動作を指定できる. initialize-instance に :after メソッドだけを定義する場合, このメソッドはシステムが提供した基本メソッドの後に起動され, initialize-instance のデフォルトの振舞いとは干渉しない.

CLOS は initialize-instance のメソッドのボディの中で使うと便利な2つの関数を提供している. 関数 slot-boundp はスロットが値をもっているかどうかを示す論理値を返す. これは, まだ初期化されていないスロットを初期化する initialize-instance の :after メソッドを書くのに便利な機能である. 関数 slot-makunbound はスロットを未束縛にする.

10.7 make-instance と initialize-instance の定義

総称関数 make-instance は次のように定義されているかのように振舞う. ただしある種の最適化は許されている.

```
(defmethod make-instance ((class standard-class) &rest initargs)
  (setq initargs (default-initargs class initargs))
  ...
  (let ((instance (apply #'allocate-instance class initargs)))
    (apply #'initialize-instance instance initargs)
    instance))

(defmethod make-instance ((class-name symbol) &rest initargs)
  (apply #'make-instance (find-class class-name) initargs))
```

上の make-instance の定義で省略されている箇所では, 与えられた初期化引数に, スロットを埋めるためあるいは適用可能メソッドの引数となるために妥当と宣言された初期化引数ではないものがないかチェックする. このチェックは総称関数 class-prototype, compute-applicable-methods, function-

keywords そして class-slot-initargs を用いて実現することができる．この初期化引数のチェックについては第3章の記述を参照のこと．

総称関数 initialize-instance は次のように定義されているかのように振舞う．ただしある種の最適化は許されている．

```
(defmethod initialize-instance ((instance standard-object)
                                &rest initargs)
  (apply #'shared-initialize instance t initargs))
```

これらの手続きは，プログラマインタフェースのレベルとメタオブジェクトのレベルの一方あるいは両方を使って独自化できる．

プログラマインタフェースのレベルにおける独自化は，defclass に指定するオプション :initform, :initarg, :default-initargs を用いること，そして make-instance や initialize-instance にメソッドを定義することで行なわれる．また総称関数 reinitialize-instance, update-instance-for-redefined-class, update-instance-for-different-class, そして initialize-instance から呼ばれる shared-initialize にメソッドを定義することでも可能である．メタオブジェクトのレベルでは，もっと自由な独自化が可能で，これは make-instance, default-initargs, または allocate-instance にメソッドを定義することでなされる．これらの総称関数とシステム提供の基本メソッドは第2章，第3章に記述している．

処理系は initialize-instance と shared-initialize にある種の最適化を行なってもよい．第2章「shared-initialize」の説明箇所で可能な最適化を記述している．

最適化を行なっている場合は，妥当な初期化引数のチェックは総称関数 class-prototype, compute-applicable-methods, function-keywords や class-slot-initargs を用いないで実現されるかもしれない．さらに総称関数 default-initargs 上のメソッドや，allocate-instance, initialize-instance, shared-initialize 上に定義されたシステム提供の基本メソッドは，make-instance を呼ぶたびに呼ばれない場合もある．また期待したとおりの引数を渡されない場合もある．

11.　クラスの再定義

standard-class のインスタンスとなっているクラスは，新しいクラスがやはり standard-class のインスタンスになるのであれば再定義可能である．クラスの再定義は新しいクラスの定義を反映するようにすでにあるクラスオブジェクトを更新するのであって，新しいクラスオブジェクトを生成するわけではない．古い defclass フォームの :reader, :writer, :accessor オプションで作られていたメソッドオブジェクトは，対応する総称関数から削除される．新しい defclass フォームで指定されたメソッドは追加される．

クラス C が再定義されると，その変更はそのクラスのインスタンスとそのクラスのすべてのサブクラスのインスタンスに伝達される．そのようなインスタンスの更新の時期は処理系に依存するが，次にそのインスタンスのスロットが読まれるか書き込まれるまでには更新されている．インスタンスの更新において，更新前後のオブジェクトは，関数 eq において等しい．更新手続きはあるインスタンスのスロ

ットを変更することはあっても，新しくインスタンスを作り出すことはない．インスタンスの更新が常にメモリを消費するかどうかは処理系に依存する．

クラスの再定義は，スロットの追加や削除を引き起こす場合もあることに注意せよ．クラスの再定義によって，クラスに属するインスタンスのアクセス可能な局所スロットの集合が変化したときは，インスタンスは更新される．クラスの再定義によって，インスタンスのアクセス可能な局所スロットの集合が変化しないときに，インスタンスが更新されるかどうかは処理系に依存する．

新しいクラスと古いクラスの両方で共有スロットと指定されたスロットの値は，そのまま保持される．もし，古いクラスにおいてそのような共有スロットが未束縛であれば新しいクラスにおいても未束縛のままである．古いクラスでは局所スロットであったが，新しいクラスでは共有スロットになったスロットは初期化される．新しく追加された共有スロットは初期化される．

新しく追加されたそれぞれの共有スロットは，新しいクラスの defclass フォームの中で，それらのスロットに対して指定された :initform フォームの評価値に設定される．もし，:initform フォームがないならば，スロットは未束縛である．

クラスの再定義によって，クラスに属するインスタンスのアクセス可能な局所スロットの集合が変化する場合，それらクラスに属するインスタンスの更新には2段階の手続きが実行される．この手続きは，総称関数 make-instances-obsolete を起動することで陽に開始を指定できる．この2段階の手続きは，処理系によっては他の場合にも実行されることがあってもかまわない．たとえばある処理系においては，この手続きはメモリ内のスロットの順序が変化した場合にも実行されるだろう．

2段階の手続きのうちの第1ステップでは，新しく局所スロットを加えたり，新しいクラスの中ではもはや定義されなくなった局所スロットを破棄するようにインスタンスの構造を修正する．第2ステップでは新たに加えられた局所スロットを初期化し，プログラマが定義したアクションがあればそれを実行する．これら2ステップの処理は次の項で詳しく述べる．

11.1 インスタンス構造の修正

第1ステップでは再定義されたクラスのインスタンスの構造をそのクラスの新しい定義に合うように修正する．古いクラスでは局所スロットとしても共有スロットとしても指定されていないが，新しいクラス定義では局所スロットと指定されたスロットは付け加え，古いクラスでは局所スロットと指定されているが，新しいクラス定義では局所スロットとしても共有スロットとしても指定されていないスロットは破棄される．これら追加または破棄されたスロットの名前は，update-instance-for-redefined-class の引数として渡される．この点については次の項で述べる．

古いクラスおよび新しいクラスの両方で指定されている局所スロットの値はそのまま保持される．もし，そのようなスロットが未束縛であれば，そのまま未束縛である．

古いクラスで共有スロットと指定されていたスロットが，新しいクラスでは局所スロットと指定された場合，そのスロットの値は古いクラスの該当する共有スロットから新しいクラスの該当する局所スロットに移される．もし，そのような共有スロットが未束縛であれば，局所スロットも未束縛のままであ

る．

11.2　新たに追加された局所スロットの初期化

第2ステップでは新たに付け加えられた局所スロットを初期化し，プログラマが定義したアクションがあればそれを実行する．この手続きは総称関数 update-instance-for-redefined-class によって実現されており，この総称関数はインスタンスの構造を変更する第1ステップの処理が終わった後に呼び出される．

総称関数 update-instance-for-redefined-class は4つの必須引数をとる．それらは，第1ステップの手続きを経て変更されたインスタンス，新たに追加された局所スロットの名前のリスト，破棄された局所スロットの名前のリスト，破棄された局所スロットで値をもっていたスロットの名前と値をもつ属性リスト，である．古いクラスでは局所スロットであり，新しいクラスで共有スロットとなったスロットも破棄されたスロットの中に含まれる．

システムによって提供される update-instance-for-redefined-class の基本メソッドがあり，そのメソッドの引数特定子は，クラス standard-object である．このメソッドは，まず最初に初期化引数が妥当であるかチェックする．もし妥当性の宣言されていない初期化引数が渡された場合はエラーを発する．（詳細は「10.2 初期化引数の妥当性の宣言」を参照のこと．）次に総称関数 shared-initialize を呼ぶ．このとき引数として，受け取った初期化引数を渡す．

11.3　クラスの再定義の独自化

update-instance-for-redefined-class にメソッドを定義することで，インスタンスが更新されるときに実行されるアクションを指定できる．もし，update-instance-for-redefined-class に対して :after メソッドだけを定義するならば，それらのメソッドはシステムが提供する初期化のための基本メソッドの後に実行される．したがって，これらのメソッドは update-instance-for-redefined-class のデフォルトの振舞いと干渉することはない．update-instance-for-redefined-class がシステムから呼ばれるときは，初期化引数は渡されない．したがって，update-instance-for-redefined-class の :before メソッドで値が設定されたスロットは，shared-initialize によってそのスロットの :initform フォームが評価されることはない．

shared-initialize にメソッドを定義することで，クラスの再定義を独自化できる．詳細は第2章「shared-initialize」を参照のこと．

11.4　拡　張

クラスの再定義に関して許される2つの拡張がある．

- オブジェクトシステムを，新しいクラスが古いクラスのメタクラスとは違ったメタクラスのインスタンスになれるように拡張してもよい．
- オブジェクトシステムを，古いクラスあるいは新しいクラスのいずれかが，standard-class 以外のクラスのインスタンス，つまり組込みでないメタクラスのインスタンスである場合にも更新処理を

サポートするように拡張してもよい.

12. インスタンスのクラスの変更

関数 change-class を用いてインスタンスのクラスを現在のクラス C_{from} から異なるクラス C_{to} に変えることができる. この関数はインスタンスの構造をクラス C_{to} の定義に合うように変える.

インスタンスのクラスの変更がスロットの追加や削除を引き起こすことがあることに注意せよ.

あるインスタンスに対して change-class が呼び出されたとき, インスタンスのクラスの更新に2段階の処理が行なわれる. 第1ステップでは, 新しい局所スロットを追加したり, 新しいクラスの定義にはもはや指定されていない局所スロットを破棄し, インスタンスの構造を修正する. 第2ステップでは, 新たに加えられた局所スロットを初期化し, プログラマが定義したアクションがあればそれを実行する. これら2ステップの処理は次の項で詳しく述べる.

12.1 インスタンスの構造の修正

インスタンスをクラス C_{to} の定義に合わせるために, クラス C_{from} では定義されていないがクラス C_{to} で定義されている局所スロットは追加され, クラス C_{from} では定義されているがクラス C_{to} で定義されていない局所スロットは破棄される.

クラス C_{to} とクラス C_{from} の両方で定義されている局所スロットの値はそのまま残る. もしそのような局所スロットが未束縛であれば未束縛のままである.

クラス C_{from} では共有であると宣言されているがクラス C_{to} では局所であると宣言されているスロットの値は, そのまま残される.

更新の第1ステップでは共有スロットの値になんら影響を与えない.

12.2 新しく追加された局所スロットの初期化

更新の第2ステップでは, 新たに付け加えられた局所スロットを初期化し, プログラマが定義したアクションがあればそれを実行する. この手続きは総称関数 update-instance-for-different-class によって実現されており, この総称関数はインスタンス構造を変更する第1ステップが終わった後に関数 change-class から呼び出される.

総称関数 update-instance-for-different-class は change-class によって計算された2つの引数で呼び出される. 第一引数は更新されるインスタンスのコピーであり, クラス C_{from} のインスタンスである. このコピーは総称関数 change-class の中では動的エクステントをもつ. 第二引数は change-class で更新中のインスタンスであり, クラス C_{to} のインスタンスである.

総称関数 update-instance-for-different-class はさらに任意個の初期化引数を受け取る. change-class から呼ばれるときは, 初期化引数は渡されない.

update-instance-for-different-class にはシステム提供の基本メソッドがある．このメソッドはクラス standard-object を引数特定子とする2つの引数をもっている．このメソッドは，まず最初に初期化引数が妥当であるかチェックする．もし妥当性の宣言されていない初期化引数が渡された場合はエラーを発する．（詳細は「10.2 初期化引数の妥当性の宣言」を参照のこと.）次に総称関数 shared-initialize を呼ぶ．このとき引数として，インスタンス，追加されたスロットの名前のリスト，そして引数として受け取った初期化引数を渡す．

12.3 インスタンスのクラスの変更の独自化

update-instance-for-different-class にメソッドを定義することで，インスタンスが更新されたときの処理を指定できる．もし update-instance-for-different-class に対して :after メソッドだけを定義すれば，それらのメソッドはシステムが提供する初期化のための基本メソッドの後に実行される．したがってこれらのメソッドは update-instance-for-different-class のデフォルトの振舞いと干渉することはない．update-instance-for-different-class が change-class から呼ばれるときは，初期化引数は渡されない．したがって update-instance-for-different-class の :before メソッドで値が設定されたスロットは，shared-initialize によってそのスロットの :initform フォームが評価されることはない．

shared-initialize にメソッドを定義することで，クラスの再定義を独自化できる．詳細は第2章「shared-initialize」を参照のこと．

13. インスタンスの再初期化

総称関数 reinitialize-instance を用いて，初期化引数に従ってスロットの値を変更できる．

この再初期化のプロセスにより，幾つかのスロット値を変えたり，あるいは，ユーザ定義の任意のアクションを実行できる．スロットを加えたり，破棄するために，インスタンスの構造を修正することはない．また，スロットを初期化するために :initform フォームを用いることはない．

総称関数 reinitialize-instance は直接呼び出してもよい．これは必須引数を1つとり，それはインスタンスである．また，reinitialize-instance あるいは shared-initialize に対するメソッドが用いる初期化引数を任意個取ってよい．必須のインスタンスの後の引数は，1つの初期化引数リストを形づくらなければならない．

reinitialize-instance には，引数特定子が standard-object であるシステム提供の基本メソッドがある．このメソッドは，まず初期化引数の妥当性をチェックし，妥当であると宣言されない初期化引数が与えられたらエラーを発する．（詳細は，「10.2 初期化引数の妥当性の宣言」を参照のこと.）次に，インスタンス，nil，受け取った初期化引数を引数として総称関数 shared-initialize を呼ぶ．

13.1 再初期化の独自化

reinitialize-instance にメソッドを定義することで，インスタンスが更新されるときに行なわれるべき動作を指定できる．reinitialize-instance に :after メソッドだけを定義するなら，システムが提供す

る初期化のための基本メソッドの後に実行される．したがって，それらは，reinitialize-instance のデフォルトの振舞いに干渉しない．

shared-initialize にメソッドを定義することで，クラスの再定義を独自化できる．詳細は第2章「shared-initialize」を参照のこと．

X3J13 Document 88-002R
June 1988

Common Lisp Object System Specification

2. Functions in the Programmer Interface

Authors: Daniel G. Bobrow, Linda G DeMichiel, Richard P. Gabriel,
Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: June 15, 1988

For information about obtaining the sources for this document, send an Internet message to common-lisp-object-system-specification-request@sail.stanford.edu.

The authors wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions to this document.

At the X3J13 meeting on June 15, 1988, the following motion was adopted:

"The X3J13 Committee hereby accepts chapters 1 and 2 of the Common Lisp Object System, as defined in document 88-002R, for inclusion in the Common Lisp language being specified by this committee. Subsequent changes will be handled through the usual editorial and cleanup processes."

X3J13 Document 88-002R
June 1988

Common Lisp Object System Specification

第2章　プログラマインタフェースにおける機能

著者：Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel,
Sonya E. Keene, Gregor Kiczales, and David A. Moon.

Draft Dated: June 15, 1988

この文書のソースを入手するための情報は common-lisp-object-system-specification-request@sail.stanford.edu に電子メイルを送ることによってえられる．

The authors wish to thank Patrick Dussud, Kenneth Kahn, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L. Weinreb, and Jon L White for their contributions to this document.

At the X3J13 meeting on June 15, 1988, the following motion was adopted:

「X3J13 委員会は 88-002R で定義された Common Lisp Object System の第1章および第2章を，本委員会で扱う Common Lisp 言語に含めることをアクセプトする．以後の変更は通常の編集・クリーンアップの過程を経て行なわれる．」

目　次

序

本章では，CLOSプログラマインタフェースで定義している関数，マクロ，特殊形式そして総称関数について記述する．このプログラマインタフェースは，ほとんどのオブジェクト指向プログラムを記述するのに十分な関数とマクロからなる．

本章は，CLOSの基本概念を理解していることを前提としている．なお，使いやすいように関数はアルファベット順に並べている．

各関数，マクロ，特殊形式そして総称関数の説明では，その目的，構文，引数の意味，および値について述べ，また例や関連した関数も示す．

関数，マクロ，特殊形式の構文記述ではその引数についても述べる．以下はその文法記述の一例である．

● **構　文：**

F *x y* &optional *z* &key *k*　　［総称関数］

この説明は，総称関数 F は必須引数*x*，*y*をもち，さらに付加引数*z*，キーワード引数*k*をもつことを示している．

本章で記述する総称関数はすべて標準総称関数であり，すべて標準メソッド結合を使用できる．

総称関数の説明では，CLOSがその総称関数の上に定義しているメソッドについても説明している．これらのメソッドの引数と引数特定子を説明するのに「既定義メソッド」を用いる．以下は既定義メソッドの一例である．

● **既定義メソッド：**

F (*x class*) (*y* t) &optional *z* &key *k*　　［基本メソッド］

この記法は，総称関数 F に属するこのメソッドは2つの必須引数*x*（これはクラス *class* のインスタンスでなければならない）および*y*（任意のオブジェクト）をもち，さらに付加引数*z*，キーワード引数*k*をもつことを示している．また，この記述は総称関数 F に属するこのメソッドは基本メソッドであり，修飾子をもたないことも示している．

総称関数の「構文」では総称関数自身のラムダリストについて，一方「既定義メソッド」ではそのメソッド自身のラムダリストについて述べる．

処理系は本章で述べている総称関数に他のメソッドも追加して提供してもよい．

以下は，関数とマクロを役割により分類したものである．

- 簡単なオブジェクト指向プログラミングのためのもの．

これらは，新たなクラス，メソッド，総称関数の定義，そしてインスタンスの生成に用いる．またメソッドのボディ内で用いるものも列挙してある．幾つかマクロには，より低レベルで同様な働きをする対応する関数がある．

call-next-method
change-class
defclass
defgeneric
defmethod
generic-flet
generic-function
generic-labels
initialize-instance
make-instance
next-method-p
slot-boundp
slot-value
with-accessors
with-added-methods
with-slots

- 一般に利用されるマクロから呼んでいる関数．

add-method
class-name
compute-applicable-methods
ensure-generic-function
find-class
find-method
function-keywords
make-instances-obsolete
no-applicable-method
no-next-method
reinitialize-instance
remove-method
shared-initialize
slot-exists-p
slot-makunbound
slot-missing
slot-unbound

update-instance-for-different-class
update-instance-for-redefined-class

- 宣言的メソッド結合のためのもの.

call-method
define-method-combination
invalid-method-error
method-combination-error
method-qualifiers

- 通常の Common Lisp でサポートされるもの.

class-of
describe
documentation
print-object
symbol-macrolet

記　法

本仕様書では拡張バッカス記法 (BNF) をオブジェクトシステムの構文の説明に用いる. 本節では BNF 表現の構文について述べる. 基本的な拡張を以下に示す.

[*O*]

この形式の表現は, ある要素のリストがより大きな構造に挿入されるときに用いられる. なおこのとき, その要素は任意の順序をとりうる. 記号*O*は挿入される複数の構文要素の記述を表わしている. この記述は以下のような形式をとる.

$O_1|...|O_N$

ここで, 各 O_i は形式S, または形式 $S*$ である. 表現 [*O*] は以下の形式のリスト

$(O_{i1}...O_{ij})$　$1 \leqq j$

が括弧で囲まれた表現内に挿入されることを意味している. そのため, もし$n \neq m$かつ$1 \leqq n$, $m \leqq j$ ならば, $O_{in} \neq O_{im}$ となるか, あるいは $1 \leqq k \leqq N$ においてある O_k が形式 Q_k* であるようなときに $O_{in}=O_{im}=Q_k$ となるかのいずれかである.

たとえば次の表現がある.

(x [A | B* | C] y)

これは, 高々1つの A, 0個以上の B, 高々1つの C がどのような順序で並んでもよいことを意味している. これは以下のものを記述していることになる.

(x y)
(x B A C y)
(x A B B B B B C y)
(x C B A B B B y)

しかし，次のものを記述していることにはならない．

(x B B A A C C y)
(x C B C y)

なぜなら，第1のものはAとCが多すぎ，また第2のものはCが多すぎるからである．

読みやすくするために，簡単な間接指定を取り入れるように拡張する．

↓*O*

もし*O*が非終端記号であるならば，その定義の右辺が表現↓*O*と置換される．たとえば以下のBNFは先の例のBNFと等しい．

(x [↓*O*] y)
O ::=A | B* | C

---

add-method　　標準総称関数

● **目　的：**

総称関数 add-method は，メソッドを総称関数に付け加える．これは総称関数を破壊的に修正し，修正された総称関数を返す．

● **構　文：**

add-method *generic-function method*　　[総称関数]

● **既定義メソッド：**

add-method (*generic-function* standard-generic-function)　　[基本メソッド]
　　(*method* method)

● **引　数：**

引数 *generic-function* は総称関数オブジェクトである．

引数 *method* はメソッドオブジェクトである．メソッド関数のラムダリストは，総称関数のラムダリストと適合しなくてはならない．そうでない場合はエラーが発せられる．

● 値：

修正された総称関数が返される．add-method の結果は引数 *generic-function* と eq である．

● 注 意：

もし与えられたメソッドが，すでに存在する総称関数のメソッドの１つと，引数特定子と修飾子において一致する場合，存在するメソッドは置き換えられる．ここでの一致の定義については「7.3 引数特定子と修飾子に関する一致」を参照のこと．

● 参 照：

「7.3 引数特定子と修飾子に関する一致」
defmethod
defgeneric
find-method
remove-method

call-method

マクロ

● 目 的：

マクロ call-method はメソッド結合で用いる．これは，どのようにメソッドが呼ばれるかという処理系に依存する詳細を隠す．マクロ call-method は静的な有効範囲をもち，有効メソッドのフォーム内だけで使用可能である．

マクロ call-method は指定されたメソッドを呼び出す．メソッドには，引数，そして call-next-method と next-method-p の定義を与える．この引数とは，call-method を起動する実効メソッドに与える引数である．call-next-method と next-method-p の定義は，call-method の第二引数に与えられたメソッドオブジェクトのリストに依存する．

関数 call-next-method が，最初のサブフォームのメソッドの中で利用される場合，２番目のサブフォームのリスト中の最初のメソッドを呼び出す．そのメソッドの中でさらに call-next-method が使われると，２番目のサブフォームのリストの第２要素のメソッドを呼び出す．以下，同様に，次メソッドのリストが無くなるまで呼び出される．

● 構 文：

call-method *method next-method-list* ［マクロ］

● 引 数：

引数 *method* はメソッドオブジェクト，*next-method-list* はメソッドオブジェクトのリストである．

call-method の最初のサブフォームであるメソッドオブジェクトや2番目のサブフォームの要素のメソッドオブジェクトの代わりに，最初の要素が make-method であり2番目の要素が Lisp フォームであるリストを用いることができる．このようなリストは，その Lisp フォームをメソッド関数のボディとするメソッドオブジェクトを表わす．

● 値：

call-method は，呼び出されたメソッドが返す値または多値を返す．

● 参 照：

call-next-method
define-method-combination
next-method-p

call-next-method　関 数

● 目 的：

関数 call-next-method はメソッド定義フォームによって定義されるメソッドのボディの中で用いることができ，次メソッドを呼び出す．

関数 call-next-method はそれが呼び出したメソッドの返す値または多値を返す．次メソッドが無ければ，総称関数 no-next-method が呼ばれる．

メソッド結合方式によってどのメソッド中で call-next-method を実行し得るか決まる．標準メソッド結合方式では基本メソッドと :around メソッドの中からだけ call-next-method が使用できる．標準メソッド結合方式では次メソッドを以下のように定義している．

- call-next-method が :around メソッドで用いられたら，次メソッドは，もし適用可能なものがあれば，次に特定的な :around メソッドである．
- もし，:around メソッドが無いか，あるいは call-next-method が最も特定的でない :around メソッドで用いられたら，以下のように他のメソッドが呼び出される．
  - — すべての :before メソッドが特定的なものから順次呼び出される．関数 call-next-method は :before メソッドの中では使用できない．
  - — 最も特定的な基本メソッドが呼び出される．基本メソッドのボディでは call-next-method によって次に特定的な基本メソッドに制御を移すことができる．基本メソッドの中で call-next-method が実行されたにもかかわらず，適用可能な基本メソッドが存在しなければ，総称関数 no-next-method が呼ばれる．
  - — すべての :after メソッドが特定的でないものから順次呼び出される．関数 call-next-method は :after メソッドの中では使用できない．

call-next-method に関するさらに詳しい議論は「8.2 標準メソッド結合」と「8.4 組込みメソッド結合方式」を参照のこと.

● 構 文:

call-next-method &rest *args* [関 数]

● 引 数:

関数 call-next-method を引数無しで呼ぶと，現在実行中のメソッドに渡された引数をそのまま次メソッドに渡す. 引数のデフォルト値，引数に対する setq の使用. 引数名と同じ名前への再束縛は call-next-method へ渡される引数に影響を与えない.

関数 call-next-method に引数をつけて呼び出すと，次メソッドはそれらの引数で呼び出される. call-next-method に引数をつけて呼ぶときは，次の条件を満足しなければならない. もしそうでなかったら，エラーが発せられる. call-next-method に指定された引数の適用可能なメソッドの順序集合は，総称関数に元の引数を与えたときに適用可能なメソッドの順序集合と一致しなければならない. エラーチェックを最適化してもよいが，call-next-method の意味は保たなければならない.

もし call-next-method が引数付きで呼ばれたが，付加引数を省略していた場合は，次メソッドはデフォルト値を使う.

● 値:

関数 call-next-method は，それが呼び出したメソッドの返す値または多値を返す.

● 注 意:

call-next-method から戻った後，さらに処理を続けることが可能である.

関数 call-next-method は静的スコープをもち，無限エクステントをもつ.

define-method-combination の短形式を用いて定義されたメソッド結合方式をもつ総称関数では，call-next-method は :around メソッドでだけ使用できる.

関数 next-method-p を用いて次メソッドがあるか否かを調べることができる.

call-next-method がそれをサポートしていないメソッドの中で使われたら，エラーが発せられる.

● 参 照:

「8. メソッド選択と結合」
「8.2 標準メソッド結合」
「8.4 組込みメソッド結合方式」
define-method-combination

next-method-p
no-next-method

change-class　　標準総称関数

● 目　的：

総称関数 change-class は，インスタンスのクラスを新しいクラスに変更する．change-class はインスタンスを破壊的に変更し，その変更されたインスタンスを返す．

古いクラスと新しいクラスで同じ名前の局所スロットはそのまま保持される．これは，そのようなスロットの slot-value によって得られる値が change-class の適用前のものと適用後のものが eql の関係であることを意味している．同様に，未束縛なスロットは束縛されていないままである．他のスロットは「12. インスタンスのクラスの変更」の節で述べているように初期化される．

● 構　文：

change-class *instance new-class*　　[総称関数]

● 既定義メソッド：

change-class (*instance* standard-object) (*new-class* standard-class)　　[基本メソッド]

change-class (*instance* t) (*new-class* symbol)　　[基本メソッド]

● 引　数：

引数 *instance* は Lisp オブジェクトである．

引数 *new-class* はクラスオブジェクト，またはクラス名のシンボルである．

2番目のメソッドでは，*instance* と (find-class *new-class*) を引数として change-class を呼び出す．

● 値：

変更されたインスタンスが返される．change-class の返すインスタンスは，引数 *instance* と eq の関係である．

● 例：

```
(defclass position () ())

(defclass x-y-position (position)
   ((x :initform 0 :initarg :x)
```

```
    (y :initform 0 :initarg :y)))

(defclass rho-theta-position (position)
   ((rho :initform 0)
    (theta :initform 0)))

(defmethod update-instance-for-different-class :before
   ((old x-y-position)
    (new rho-theta-position) &key)
   ;;x-y 座標による古い表現から rho-theta 座標による表現に変換する
   (let ((x (slot-value old 'x))
         (y (slot-value old 'y)))
     (setf (slot-value new 'rho) (sqrt (+ (* x x) (* y y)))
           (slot-value new 'theta) (atan y x))))

;;;この時点で change-class を使えばクラス x-y-position のインスタンスは
;;;クラス rho-theta-position のインスタンスに変換できる

(setq p1 (make-instance 'x-y-position :x 2 :y 0))

(change-class p1 'rho-theta-position)

;;;p1 に束縛されているインスタンスはクラス rho-theta-position の
;;;インスタンスになった. update-instance-for-different-class に定義
;;;されたメソッドが x-y 座標表現から rho-theta 表現に変更した
```

● 注 意：

change-class は最後に総称関数 update-instance-for-different-class を呼び出す. update-instance-for-different-class は変換されたインスタンスのスロットへ値を代入するために使うことができる.

総称関数 change-class には幾つかの意味的困難さがある. 第1に，あるメソッドを選択するために用いたインスタンスのクラスを変更するという破壊的操作が，そのメソッドの中でできるということである. メソッド結合によって複数のメソッドが起動される場合は，現在実行中のメソッドばかりでなく，これから実行されるメソッドももはや適用可能なメソッドではなくなっている. 第2に，スロットアクセスに関してコンパイラで最適化を行なっている処理系もあるだろうが，そのようなシステムではインスタンスのクラスが変更されたら，コンパイラの用いた前提条件が壊れてしまうかもしれない. このことは，プログラマは，もしある総称関数に属するいずれかのメソッドがいずれかのスロットをアクセスしているならば，その総称関数に属するどのメソッドの中からも change-class を呼ぶべきではなく，もし呼ぶと結果は未定義であるということを意味している.

● 参 照：

「12. インスタンスのクラスの変更」

update-instance-for-different-class

class-name, (setf class-name)　　標準総称関数

● 目　的：

総称関数 class-name はクラスオブジェクトを引数としてとり，その名前を返す．

総称関数 (setf class-name) はクラスオブジェクトを引数としてとり，その名前を設定する．

● 構　文：

class-name *class*　　[総称関数]

(setf class-name) *new-value class*　　[総称関数]

● 既定義メソッド：

class-name (*class* class)　　[基本メソッド]

(setf class-name) *new-value* (*class* class)　　[基本メソッド]

● 引　数：

引数 *class* はクラスオブジェクトである．

引数 *new-value* は任意のオブジェクトである．

● 値：

与えられたクラスの名前を返す．

● 注　意：

名無しのクラスの名前は nil である．

もしSがシンボルであり，S=(class-name C) かつ C=(find-class S) であれば，SはCの固有名である．より詳細な議論は「3. クラス」を参照のこと．

● 参　照：

「3. クラス」
find-class

class-of

関 数

● 目 的：

関数 class-of は与えられたオブジェクトをインスタンスとするクラスを返す．

● 構 文：

class-of *object* ［関 数］

● 引 数：

class-of の引数は，任意の Common Lisp オブジェクトである．

● 値：

関数 class-of は与えられた引数をインスタンスとするクラスを返す．

compute-applicable-methods

関 数

● 目 的：

総称関数と一連の引数を与えられたとき，関数 compute-applicable-methods はこれらの引数について適用可能なメソッドの集まりを返す．メソッドは優先順位に従ってソートされている．「8. メソッド選択と結合」を参照のこと．

● 構 文：

compute-applicable-methods *generic-function function-arguments* ［関 数］

● 引 数：

引数 *generic-function* は総称関数オブジェクトである．引数 *function-arguments* はその総称関数に与えられた引数のリストである．

● 値：

結果は優先順位に従って並べられた適用可能メソッドのリストである．

● 参 照：

「8. メソッド選択と結合」

defclass　　マクロ

● 目　的：

マクロ defclass は新しいクラスを定義する．defclass は結果として新しいクラスオブジェクトを返す．

defclass の構文は，スロットの初期化引数を指定するオプション，スロットのデフォルト初期値を指定するオプション，スロット値の書込みや参照を行なう総称関数にメソッドを自動的に生成するよう要求するオプションをもつ．省略時には，これらのライタ関数やリーダ関数は定義されない．これらの関数の生成は陽な指定によって行なわれる．

新しいクラスを定義すると，同じ名前の Lisp データ型も定義される．述語 (typep *object class-name*) を評価すると，*object* のクラスが *class-name* であるかまたは *class-name* のサブクラスであれば真となる．クラスオブジェクトは型指定子として使用できる．つまり，(typep *object class*) を評価すると，*object* のクラスが *class* かまたは *class* のサブクラスであれば真となる．

● 構　文：

defclass *class-name* ({*superclass-name*}*) ({*slot-specifier*}*) [↓*class-option*] [マクロ]

class-name ::= *symbol*

superclass-name ::= *symbol*

slot-specifier ::= *slot-name* | (*slot-name* [↓ *slot-option*])

slot-name ::= *symbol*

slot-option ::= {:reader *reader-function-name*}* |
{:writer *writer-function-name*}* |
{:accessor *reader-function-name*}* |
{:allocation *allocation-type*} |
{:initarg *initarg-name*}* |
{:initform *form*} |
{:type *type-specifier*} |
{:documentation *string*}

reader-function-name ::= *symbol*

writer-function-name ::= *function-specifier*

function-specifier ::= {*symbol* | (setf *symbol*)}

initarg-name ::= *symbol*

allocation-type ::= :instance | :class

class-option ::= (:default-initargs ***initarg-list***) |
(:documentation *string*) |
(:metaclass *class-name*)

initarg-list ::= {*initarg-name default-initial-value-form*}*

図 1 defclass の構文

● 引 数:

引数 *class-name* は nil でないシンボルである. *class-name* は新しいクラスの固有名となる. もし *class-name* に該当するクラスがすでにあり, そのクラスが standard-class のインスタンスであって, 新しいクラスのクラスも standard-class であれば, 再定義される.

引数 *superclass-name* は nil でないシンボルであり, 新しいクラスのダイレクトスーパークラスを指定する. 新しいクラスはおのおののダイレクトスーパークラス, そしてさらにそれらのダイレクトスーパークラスというようにすべてのスーパークラスから, スロットとメソッドを継承する. スロットやメソッドの継承がどのように行なわれるかについては,「4. 継承」を参照せよ.

引数 *slot-specifier* は, それぞれスロット名またはスロット名と 0 個以上のオプションからなるリストである. 引数 *slot-name* は Common Lisp の変数名として構文的に有効なシンボルである. もしスロット名に重複があれば, エラーが発せられる.

以下のスロットオプションの指定が可能である.

- :reader オプションはスロット値を読み込むためのメソッドとして, 修飾されていないメソッドを *reader-function-name* という名前の総称関数に定義するよう要求する. 引数 *reader-function-name* は nil でないシンボルである. :reader オプションは 1 つのスロットに対して複数回指定してもよい.
- :writer オプションはスロット値を書き込むためのメソッドとして, 修飾されていないメソッドを *writer-function-name* という名前の総称関数に定義するよう要求する. 引数 *writer-function-name* は関数指定子である. :writer オプションは 1 つのスロットに対して複数回指定してもよい.
- :accessor オプションはスロット値を読み込むためのメソッドとして, 修飾されていないメソッドを与えられた総称関数 *reader-function-name* に定義することを要求する. さらに, 総称関数 (setf *reader-function-name*) に修飾されていないメソッドを定義し, setf を用いたスロット値の変更を可能にする. 引数 *reader-function-name* は nil でないシンボルである. :accessor オプションは 1 つのスロットに対して複数回指定してもよい.
- :allocation オプションはそのスロットをどの記憶領域に割り付けるかを指定する. スロットの記憶領域はそれぞれのインスタンスあるいはクラスオブジェクト自身に割り付けることが可能であ

る．引数 *allocation-type* の値はキーワード :instance または :class のいずれかである．:allocation オプションは1つのスロットに対して1回だけ指定できる．:allocation オプションが指定されなかった場合は，:allocation :instance と指定されたのと同じように動く．

— *allocation-type* が :instance の場合は，与えられた名前の局所スロットがそのクラスの各インスタンスに割り付けられる．

— *allocation-type* が :class の場合は，与えられた名前の共有スロットが defclass によって生成されるクラスオブジェクトに割り付けられる．そのスロットの値はそのクラスのすべてのインスタンスによって共有される．もしクラス C_1 でそのような共有スロットを定義したら，C_1 のどのサブクラス C_2 でもそのスロットを共有できる．ただし，C_2 には同名のスロット定義は無く，C_2 のクラス優先順位リスト上で C_1 より先行するクラスに同名のスロット定義が無い場合である．

- :initform オプションはそのスロットの初期化に用いるデフォルト初期値フォームを与える．:initform オプションは1つのスロットに対してせいぜい1回指定できる．このフォームは用いられるたびに評価される．このフォームが評価されるときのレキシカルな環境は defclass が評価されたときのレキシカルな環境と同じである．ここでレキシカルな環境とは変数と関数の両方を指している．局所スロットに関しての動的環境は make-instance が呼ばれたときの動的環境である．共有スロットに関しての動的環境は defclass が評価されたときの動的環境である．「10. オブジェクトの生成と初期化」を参照のこと．

処理系は defclass の構文を拡張して（*slot-name* `:initform` *form*）の省略形として（*slot-name form*）を許すようにしてはならない．

- :initarg オプションは，*initarg-name* という名前の初期化引数を宣言し，この初期化引数を用いて与えられたスロットを初期化できるようする．initialize-instance の引数として初期化引数とその値を渡すと，その値がスロットに設定される．このときは，:initform オプションがあってもそれは評価されない．もしどの初期化引数も与えられたスロットを初期化する値をもっていなかったら，スロットは指定された :initform オプションに従って初期化される．:initarg オプションは1つのスロットに対して複数回指定してもよい．引数 *initarg-name* はどんなシンボルでもよい．
- :type オプションは，そのスロットの内容が常に指定されたデータ型であるということを指定する．この指定は，そのクラスのオブジェクトに対してリーダ総称関数が適用されたときに返す値の型を実質的に宣言している．スロットに代入しようとしている値が型の制限を満たさない場合の結果については未定義である．:type オプションは1つのスロットに対してせいぜい1回指定できる．:type オプションに関しては「4.2 スロットとスロットオプションの継承」でさらに議論する．

クラスオプションはクラス全体，またはクラスのすべてのスロットに関するものである．次のようなクラスオプションがある．

- :default-initargs オプションの後には，初期化引数名とデフォルト初期値フォームが交互に現われるリストが続く．もしこれらの初期化引数のいずれかが make-instance に与えられた初期化引数リストに無かったら，対応するデフォルト初期値フォームが評価され，初期化引数名とフォーム

の値が，インスタンスが生成されるより前に，初期化引数のリストの終りに追加される（「10. オブジェクトの生成と初期化」を参照）．デフォルト初期値フォームは使われるたびに評価される．このフォームが評価されるレキシカルな環境は defclass が評価されたときのレキシカル環境である．動的環境は make-instance が呼ばれたときのものである．もし１つの :default-initargs オプションに重複する初期化引数名が出現したら，エラーが発せられる．:default-initargs オプションは１つのクラスに対して１回だけ指定できる．

- :documentation オプションはクラス名につける文書文字列を与える．この文字列の文書型は type である．(documentation *class-name* 'type) を評価すると文書文字列を得ることができる．:documentation オプションは１つのクラスに対してせいぜい１回指定できる．
- :metaclass オプションは，生成されるインスタンスのメタクラスをシステムの標準メタクラス（クラス standard-class）以外のものにしたいときに指定する．引数 *class-name* はメタクラス名である．:metaclass オプションの指定は１つのクラスに対して１回だけ指定できる．

● **値：**

defclass は結果として新しいクラスオブジェクトを返す．

● **注　意：**

もし同じ固有名のクラスがすでに存在し，そのクラスがクラス standard-class のインスタンスであり，定義しようとする新しいクラスのクラスも standard-class なら，そのクラスは再定義され，そのクラス（とサブクラス）のインスタンスは次にアクセスされるときに新しい定義に合うように更新される．詳細は「11. クラスの再定義」を参照のこと．

標準クラスの defclass は以下の規則に従う．

- 定義しようとしているクラスのスーパークラスは，defclass の評価時にはまだ定義されていなくてもよい．
- そのクラスのインスタンスが生成されるより前に，すべてのスーパークラスが定義されていなければならない．
- あるクラスが defmethod の引数特定子として使用されるより前に，そのクラスは定義されていなければならない．

オブジェクトシステムの振舞いは次に述べる規則に違反しない場合に拡張できる．

スロットオプションにはスーパークラスより継承されるものや，局所スロット記述によってシャドウされたり変更されるものもある．:default-initargs 以外のクラスオプションは継承されない．スロットとスロットオプションがいかに継承されるかについての詳細は「4.2 スロットとスロットオプションの継承」を参照のこと．

defclass に他のオプションを追加することは処理系の自由である．そのためすべての処理系において，それぞれの処理系に用意されていないスロットオプションまたはクラスオプションが与えられたときにはエラーを発することが必要である．

1つのスロットに2つ以上のリーダ，ライタ，アクセサ，そして初期化引数を指定することが可能である．他のスロットオプションは1つのスロットについて2回以上指定することはできない．

あるスロットにリーダ，ライタ，アクセサのいずれも指定されなかったら，そのスロットは関数 slot-value によってだけアクセス可能である．

● 参　照：

「3. クラス」
「4. 継承」
「11. クラスの再定義」
「6. クラス優先順位リストの決定」
「10. オブジェクトの生成と初期化」
slot-value
make-instance
initialize-instance

defgeneric　　マクロ

● 目　的：

マクロ defgeneric は総称関数を定義するため，あるいは総称関数全体に関するオプションと宣言を指定するために使用する．

もし（fboundp *function-specifier*）が nil なら新しい総称関数が生成される．もし（symbol-function *function-specifier*）が既存の総称関数なら，その総称関数は修正される．もし *function-specifier* が普通の関数，マクロまたは特殊形式の名前であれば，defgeneric はエラーを発する．

method-description はその総称関数上にメソッドを定義する．メソッドのラムダリストは *lambda-list* と適合しなければならない．この条件が成り立たなければエラーが発せられる．この文脈の中の適合性の定義に関しては「7.4 1つの総称関数に属するすべてのメソッドの適合ラムダリスト」を参照のこと．

マクロ defgeneric は結果として総称関数オブジェクトを返す．

● 構　文：

defgeneric *function-specifier lambda-list* [↓ *option* | *method-description**]　　[マクロ]

function-specifier ::= {*symbol* | (setf *symbol*)}

lamda-list ::= ({*var*}*
[&optional {*var* | (*var*)}*]
[&rest *var*]

```
                [&key {var | ({var | (keyword var)})}*
                      [&allow-other-keys]])

option ::= (:argument-precedence-order {parameter-name}+) |
           (declare {declaration}+) |
           (:documentation string) |
           (:method-combination symbol {arg}*) |
           (:generic-function-class class-name) |
           (:method-class class-name)

method-description ::= (:method {method-qualifier}* specialized-lambda-list
                                {declaration | documentation}* {form}*)

method-qualifier ::= non-nil-atom

specialized-lambda-list ::= ({var | (var parameter-specializer-name)}*
                             {&optional {var | (var [initform [supplied-p-parameter]])}*]
                             [&rest var]
                             [&key {var | ({var | (keyword var)} [initform [supplied-p-
                              parameter]])}*
                                [&allow-other-keys]]
                             [&aux {var | (var [initform])}*])

parameter-specializer-name ::= symbol | (eql eql-specializer-form)
```

● 引 数：

引数 *function-specifier* は nil でないシンボルまたはリスト (setf *symbol*) である．

引数 *lambda-list* は通常の関数のラムダリストと同様であるが，以下の点で異なっている．

- &aux は許されない．
- 付加引数とキーワード引数はデフォルト初期値フォームをもてないし，supplied-p パラメータを用いることもできない．総称関数は引数に与えられた値をすべてそのまま引き渡し，それ以外は渡さない．すなわち，デフォルト値は扱わない．しかしメソッドの定義では，付加引数とキーワード引数はデフォルト初期値フォームをもつことができ，supplied-p パラメータも用いることができる点に注意せよ．

次のオプションが指定可能である．これらのオプションは1回だけ指定できる．もし2回以上指定するとエラーが発せられる．

- :argument-precedence-order オプションは，総称関数が呼ばれ，メソッドが選択されるときに，特定性を調べる順序を指定する．すべての必須引数は正確に一度だけ *parameter-name* として出現する必要がある．こうすることにより十分で不明確さの無い優先順序が決まる．もしこの条件が満

足されていなければエラーが発せられる.

- declare オプションは総称関数に関する宣言を指定するために用いる. 次の標準 Common Lisp 宣言が許されている.
  - — optimize 宣言によって, メソッド選択の処理において, 速度とスペースのどちらを最適化すべきかを指定する. これはメソッドには影響を与えない. メソッド自身をいかに最適化するかを制御するためには declare オプションを直接 defmethod あるいはメソッド記述に書かねばならない. 標準仕様で最適化の性質として速度とスペースだけを定めるが, 各処理系では他の性質も最適化ができるようにしてよい. メソッド選択の手法を1つしかもたず, optimize オプションが無視されるような, 単純な処理系もあってよい.

special, ftype, function, inline, notinline, そして declaration を宣言することは許されない. 処理系は, 他の宣言をサポートするよう declare オプションを拡張してもよい. もし処理系でサポートしていなくて, それが非標準のオプションであるとの宣言もされていなければ, 処理系は警告を発するべきである.

- :documentation オプションは文書文字列を総称関数に関連付ける. この文字列の文書型は function である. (`documentation` *function-specifier* '`function`) によって関連付けた文字列が得られる.
- :generic-function-class オプションはシステムが提供しているデフォルトのクラス (クラス standard-generic-function) ではないクラスの総称関数を作ることを指定をする. 引数 *class-name* は総称関数のクラスとなりうるクラスの名前である. もし *function-specifier* が既存の総称関数の名前であり, 新しい総称関数の :generic-function-class の値が既存の総称関数の :generic-function-class の値と異なっている場合は, 新しい総称関数のクラスが古いものと互換性があるなら, change-class が呼ばれて総称関数のクラスを変更する. 互換性が無ければエラーが発せられる.
- :method-class オプションはシステムが提供しているデフォルトのクラス (クラス standard-method) ではないクラスをその総称関数のメソッドのクラスにすることを指定する. 引数 *class-name* はメソッドのクラスとなりうるクラスの名前である.
- :method-combination オプションの後にはメソッド結合方式の名前を表わすシンボルが続く. シンボルに続く引数 (もしあれば) はメソッド結合方式に依存する. 標準メソッド結合方式は, 引数をサポートしていない. しかし define-method-combination の短形式で定義されるすべてのメソッド結合方式は *order* という付加引数をとる. *order* のデフォルト値は :most-specific-first である. これに :most-specific-last を与えると基本メソッドを並べる順序が逆になるが, 補助メソッドの順序には影響しない.

引数 *method-description* は総称関数にメソッドを定義する. *method-qualifier* と *specialized-lambda-list* 引数は defmethod のものと同じである.

引数 *form* はメソッドのボディを決める. メソッドのボディは暗黙的なブロックに囲まれる. もし *function-specifier* がシンボルならこのブロックは総称関数と同じ名前をもつ. もし *function-specifier* がリスト (`setf` *symbol*) ならブロックの名前は *symbol* となる.

● 値：

マクロ defgeneric は結果として総称関数オブジェクトを返す．

● 注　意：

マクロ defgeneric は次のような3つのステップを実行すると考えてよい．第1に，以前の defgeneric で定義されたメソッドを削除する．第2に，ensure-generic-function を呼ぶ．最後に，現在の defgeneric で定義されたメソッドを総称関数に追加する．

メソッド記述が無く，同じ名前の総称関数が存在しなかったら，メソッドをもたない総称関数が生成される．

defgeneric の引数 *lambda-list* は総称関数に属するメソッドのラムダリストの形を規定する．この総称関数のメソッドのラムダリストは，ここに指定した形と適合していなければならない．defgeneric が評価されたとき，対応する総称関数に属するメソッドの中に，指定されたラムダリストと適合しないラムダリストをもつものがあれば，エラーが発せられる．メソッドの適合性に関する詳細は「7.4 1つの総称関数に属するすべてのメソッドの適合ラムダリスト」を参照のこと．

処理系は，defgeneric を拡張し，他のオプションが指定できるようにしてよい．しかし，その処理系では実現していないオプションがあればエラーを発することが必要である．

● 参　照：

「7.4 1つの総称関数に属するすべてのメソッドの適合ラムダリスト」
defmethod
ensure-generic-function
generic-function

define-method-combination　　マクロ

● 目　的：

マクロ define-method-combination は新しい方式のメソッド結合を定義するために用いる．

define-method-combination には，短形式と長形式の2つの形式がある．前者は多くの一般的必要性を満たす簡単な機能であり，後者は強力であるがその反面複雑である．長形式は，ボディが通常バッククォートを用いて Lisp 式を計算するような式なので defmacro に似ており，任意の制御構造を実現することが可能である．また，長形式はメソッド修飾子の任意の処理を許している．

● 構　文：

define-method-combination *name* [↓ *short-form-option*]　　[マクロ]

short-form-option ::= :documentation *string* |
:identity-with-one-argument *boolean* |
:operator *operator*

define-method-combination *name lambda-list* [マクロ]
({*method-group-specifier*}*)
[(:arguments . *lambda-list*)]
[(:generic-function *generic-function-symbol*)]
{*declaration* | *doc-string*}*
{*form*}*

method-group-specifier ::= (*variable* {{*qualifier-pattern*}+ | *predicate*}
[↓ *long-form-option*])

long-form-option ::= :description *format-string* |
:order *order* |
:required *boolean*

● **引　数：**

短形式と長形式のいずれの場合も *name* はシンボルである．通常キーワードでなく nil でないシンボルが用いられる．

● **短形式の引数：**

第二引数が nil でないシンボルであるか存在しない場合，短形式であると判断される．短形式の場合，(*operator method-call method-call...*) という Lisp 式を生成するメソッド結合の方式として *name* が定義される．*operator* は，関数，マクロ，あるいは特殊形式の名前を表わすシンボルである．*operator* は，キーワードオプションで指定する．デフォルトは *name* である．

短形式に対するキーワードオプションは，以下のとおりである．

- :documentation オプションは，method-combination 型の文書文字列の登録に用いる．
- :identity-with-one-argument オプションは，*boolean* が真の場合に（デフォルトは偽）最適化を許す．もし適用可能メソッドが1つしかなく，しかもそれが基本メソッドである場合，そのメソッドが実効メソッドになり，*operator* は呼ばれない．このオプションは新しい実効メソッドを生成したり関数を呼び出したりする手間を省く．このオプションは，progn, and, +, max などのオプションと共に用いるよう設計されている．
- :operator オプションは，オペレータの名前を指定する．引数 *operator* は，関数，マクロ，特殊形式の名前を表わすシンボルである．通常，*name* と *operator* は同じシンボルを用いるが，そうしなければならないわけではない．

サブフォームはいずれも評価されない．

これらのメソッド結合方式は，それぞれのメソッドが修飾子を１つもつことを要求する．もし修飾子をもたない適用可能メソッドがあったり，そのメソッド結合方式ではサポートしていない修飾子をもつメソッドがあると，エラーが発せられる．

この方法で定義されたメソッド結合手続きは，メソッドの２つの役割を認識する．メソッド結合方式の名前のシンボルを唯一の修飾子にもつようなメソッドは基本メソッドとして定義される．適用可能な基本メソッドが少なくとも１つは存在する必要があり，１つも無いとエラーを発する．修飾子として，:around だけをもつメソッドは補助メソッドであり，標準メソッド結合の :around メソッドと同じ振舞いをする．関数 call-next-method は，:around メソッドの中だけで使用可能で，マクロ define-method-combination の短形式で定義された基本メソッドの中では使用できない．

この方法で定義されたメソッド結合手続きは，付加引数 *order* を受け付ける．デフォルトは，:most-specific-first である．値が :most-specific-last であれば，基本メソッドの順序を逆にする．このとき，補助メソッドの順序は変化しない．

短形式は，自動的にエラーチェックを行ない，:around メソッドをサポートする．

組込みメソッド結合方式については，「8.4 組込みメソッド結合方式」を参照のこと．

● **長形式の引数：**

第二引数がリストの場合，長形式であると判断される．

引数 *lambda-list* は普通のラムダリストである．これは，defgeneric の :method-combination オプションのメソッド結合方式の名前の後に与えられた任意の引数を受け取る．

その次にメソッドグループ指定子 (method-group-specifier) のリストが続く．おのおのの指定子は，適用可能メソッドの中で，特定の役割を果たすサブセットを選ぶ．選択は，メソッドの修飾子を何らかのパターンとマッチングを取るか，修飾子を述語でテストするかによって行なわれる．これらのメソッドグループ指定子は，このメソッド結合方式で使用可能なすべてのメソッド修飾子を定義する．もし，適用可能メソッドがどのメソッドグループにも入らない場合，このメソッドが使用中のメソッド結合の種類として適当でないことを示すエラーを発する．

おのおののメソッドグループ指定子は，その指定子と同じ名前の変数を作る．define-method-combination のボディの実行中，この変数はそのメソッドグループのメソッドのリストに束縛される．このリスト中のメソッドは，より特定的なものが先になるような順序に並ぶ．

修飾子パターン (qualifier-pattern) はシンボル＊またはリストである．メソッドは，そのメソッドの修飾子のリストが修飾子パターンと equal のとき（ただし，修飾子パターンの中の＊は何とでもマッチする），マッチする．修飾子パターンとして以下のものがある．空リスト（ ）は修飾子のないメソッドにマッチする．シンボル＊はすべてのメソッドにマッチする．nil でないリストは，リストの長さと同じ個数の修飾子をもち，おのおのの修飾子が対応するリストの要素とマッチするときマッチする．シンボル＊で終わるドットリストの場合は，＊はその後の任意個の修飾子とマッチする．

それぞれの適用可能メソッドは，一連の修飾子パターンと述語に対して左から右の順序でテストされ

る．修飾子パターンがマッチするか述語が真を返せば，その時点で，このメソッドは対応するメソッドグループのメンバになり，それ以上のテストは行なわれない．したがって，もしあるメソッドが複数のメソッドグループのメンバに成り得る場合でも，そのメソッドはメンバに成り得る最初のグループにだけ属する．もし，メソッドグループが2つ以上の修飾子パターンをもっている場合は，その中の1つの修飾子パターンにマッチすれば，メソッドはそのグループのメンバになる．

メソッドグループ指定子の修飾子パターンの代わりに述語関数の名前を記述することができ，まだ先行するメソッドグループに割り当てられていないメソッドに述語が適用される．引数は1つで，そのメソッドの修飾子のリストである．述語はメソッドがそのメソッドグループのメンバであれば真を返す．述語は nil および＊でないシンボルなので，修飾子パターンとは区別できる．

メソッドグループ指定子には，修飾子パターンや述語の後にキーワードオプションを記述することができる．キーワードオプションは，リストでもシンボル＊でもないので，次に続く修飾子パターンと区別できる．キーワードオプションは以下のとおりである．

- :description オプションは，メソッドグループに属するメソッドの役割を記述するために用いる．プログラミング環境のツールが（apply #'format stream *format-string* (method-qualifiers *method*)）を使えば，この記述が出力される．これは，短いものであることが期待される．このオプションは，メソッド修飾子を定義したモジュール内にメソッド修飾子の意味を記述することを可能にする．通常この *format-string* では書式指定は使わないだろうが，一般性のために利用可能にしている．もし :description が指定されなければ，変数名，修飾子パターン，そしてこのメソッドグループが修飾されていないメソッドを含むかどうかに基づきデフォルトの記述が生成される．引数 *format-string* は評価されない．
- :order オプションはメソッドの順序を指定する．引数 *order* は，評価した結果が，:most-specific-first か :most-specific-last になるようなフォームである．もしそれ以外の値になればエラーを発する．このオプションは便宜上のものであり，実質的な表現力を高めるものではない．もし :order が指定されなければ，デフォルトとして :most-specific-first になる．
- :required オプションは，このメソッドグループに属するメソッドが少なくとも1つは必要であるかどうかを指定する．もし引数 *boolean* が nil でなく，メソッドグループが空であれば（修飾子パターンまたは述語を満たすような適用可能メソッドがない場合）エラーを発する．このオプションは便宜上のものであり，実質的な表現力を高めるものではない．もし :required が指定されなければ，デフォルトは nil である．引数 *boolean* は評価されない．

メソッドグループ指定子は，メソッドを選んだり，メソッドをある役割で分類したり，必要なエラーチェックを行なう簡単な構文を与える．通常のリスト処理操作や関数 method-qualifiers, invalid-method-error を用いてボディの中でメソッドをさらにフィルタすることも可能である．また，メソッドグループ指定子で名前付けられた変数に setq することや付加的な変数を束縛することも許されている．さらに，メソッドグループ指定子の機構をバイパスして，すべてをボディフォームの中で行なうことも可能である．これは＊を修飾子パターンにもつメソッドグループだけを定義することにより行なわれる．この場合，変数にはすべての適用可能なメソッドをより特定的な順序に並べたリストが束縛される．

ボディ *forms* は，メソッドをいかに結び付けるかを記述した Lisp 式，すなわち実効メソッドを計算して返す．実効メソッドは，マクロ call-method を用いる．このマクロは，静的な有効範囲をもち，実効メソッドのフォーム内だけで使用可能である．メソッドグループ指定子により生成されたリストの要素の1つであるメソッドオブジェクトと次メソッドのリストが与えられたとき，call-method は call-next-method を使えば次メソッドを呼べるようにしてメソッドを起動する．

実効メソッドが1つのメソッドを呼び出す効果しかもたない場合，処理系によっては，新しい実効メソッドの生成を避けるために，そのメソッドをそのまま実効メソッドとして利用するような最適化を行なうであろう．この最適化は，実効メソッドの式が，マクロ call-method を呼び出すだけであって，call-method の第一引数がメソッドオブジェクトであり第二引数が nil の場合に有効である．それぞれの define-method-combination のボディ中では，progn や multiple-value-progl などの冗長な呼出しを，もしこの最適化を望むならば，削除するべきである．

リスト（:arguments . *lambda-list*）は，宣言と文書文字列の前に置くことができる．メソッド結合方式が結合されたメソッドの動作の一部として特別な振舞いを必要とし，総称関数の引数をアクセスする必要がある場合に，このフォームが役に立つ．*lambda-list* に指定した引数変数は，実効メソッドの中に組み込むことができる式に束縛される．実効メソッドの実行の間にこの式が評価された場合，その値は総称関数に渡された対応する引数の値になる．もし，*lambda-list* が総称関数のラムダリストに適合しなければ，付加的な無視されるような引数が，適合するまで自動的に挿入される．したがって，総称関数より少ない引数を受け取るような *lambda-list* を用いてよい．

ボディの中で検出されたエラーは，method-combination-error または invalid-method-error で出力するべきである．これらの関数は，何らかの必要な文脈的な情報を付加し，適切なエラーを発する．

ボディ *forms* は，ラムダリストとメソッドグループ指定子によって作られた束縛の中で評価される．ボディの先頭の宣言は，ラムダリストによって作られた束縛のすぐ内側で，メソッドグループ変数の束縛の外側に置かれる．したがって，メソッドグループ変数についての宣言はできない．

ボディ *forms* の中で，*generic-function-symbol* が総称関数オブジェクトに束縛される．

もし引数 *doc-string* があれば，それはメソッド結合方式の文書情報になる．

関数 method-combination-error と invalid-method-error は，ボディ *forms* または，ボディ *forms* が呼ぶ関数から呼べる．これら2つの関数の動作は，総称関数 compute-effective-method が呼ばれる前に処理系が自動的に束縛した動的変数の値に依存してもよい．

同じ引数指定子をもつが修飾子が異なる2つのメソッドは，「8. メソッド選択と結合」で述べたメソッド選択と結合のプロセスのステップ2のアルゴリズムでは順序が決まらないことに注意せよ．通常これらは異なる修飾子をもつため，異なる役割を果たし，ステップ2の結果どのように順序付けされても実効メソッドは変わらない．もし，2つのメソッドが同じ役割を果たし，これらの順序が影響する場合はエラーを発する．これは，define-method-combination の修飾子のパターンのマッチングの部分で起こる．

● **値：**

マクロ define-method-combination は，新しいメソッド結合方式の名前を返す．

● **例：**

define-method-combination の長形式の例を見れば，宣言的なメソッド結合機能の一部として準備されている関数の利用法も理解できる．

```
;;;define-method-combination の短形式の例

(define-method-combination and :identity-with-one-argument t)

(defmethod func and ((x class1) y) ...)

;;;前の例と等価な長形式

(define-method-combination and
         (&optional (order ':most-specific-first))
         ((around (:around))
          (primary (and) :order order :required t))
     (let ((form (if (rest primary)
                     `(and ,@(mapcar #'(lambda (method)
                                         `(call-method ,method ()))
                                     primary))
                     `(call-method ,(first primary) ()))))
       (if around
           `(call-method ,(first around)
                         (,@(rest around)
                          (make-method ,form)))
           form)))

;;;define-method-combination の長形式の例

; デフォルトメソッド結合のテクニック
(define-method-combination standard ()
        ((around (:around))
         (before (:before))
         (primary () :required t)
         (after (:after)))
  (flet ((call-methods (methods)
           (mapcar #'(lambda (method)
                       `(call-method ,method ()))
                   methods)))
```

```
      (let ((form (if (or before after (rest primary))
                      `(multiple-value-prog1
                        (progn ,@(call-methods before)
                               (call-method ,(first primary)
                                            ,(rest primary)))
                        ,@(call-methods (reverse after)))
                      `(call-method ,(first primary) ()))))
        (if around
            `(call-method ,(first around)
                          (,@(rest around)
                           (make-method ,form)))
            form))))

; 数個のメソッド中の1つが nil でない値を返すまで繰り返す簡単な方法
(define-method-combination or ()
        ((methods (or)))
  `(or ,@(mapcar #'(lambda (method)
                     `(call-method ,method ()))
                 methods)))

; 前の例の，より完全な例
(define-method-combination or
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (or)))
   ;; order 引数の処理
   (case order
     (:most-specific-first)
     (:most-specific-last (setq primary (reverse primary)))
     (otherwise (method-combination-error "~S is an invalid order.~@
:most-specific-first and :most-specific-last are the possible values."
                                          order)))
   ;; 基本メソッドがなければならない
   (unless primary
     (method-combination-error "A primary method is required."))
   ;; 基本メソッドを呼び出す式を作る
   (let ((form (if (rest primary)
                   `(or ,@(mapcar #'(lambda (method)
                                      `(call-method ,method ()))
                                  primary))
```

```
                    `(call-method ,(first primary) ()))))
      ;; around メソッドでこの式を囲む
      (if around
          `(call-method ,(first around)
                        (,@(rest around)
                         (make-method ,form)))
          form)))

; 同じことを, :order と :required キーワードオプションを使って記述した例
(define-method-combination or
        (&optional (order ':most-specific-first))
        ((around (:around))
         (primary (or) :order order :required t))
   (let ((form (if (rest primary)
                   `(or ,@(mapcar #'(lambda (method)
                                      `(call-method ,method ()))
                                  primary))
                   `(call-method ,(first primary) ()))))
     (if around
         `(call-method ,(first around)
                       (,@(rest around)
                        (make-method ,form)))
         form)))

; 次の短形式は, 前述の例と等価な振舞いをする.
(define-method-combination or :identity-with-one-argument t)

; 正の整数の修飾子を実行順序と見なすメソッド結合
;; ここでは簡単にするために, :around メソッドは許さないものとする
(define-method-combination example-method-combination ()
        ((methods positive-integer-qualifier-p))
    `(progn ,@(mapcar #'(lambda (method)
                          `(call-method ,method ()))
                      (stable-sort methods #'<
                        :key #'(lambda (method)
                                 (first (method-qualifiers method)))))))

(defun positive-integer-qualifier-p (method-qualifiers)
    (and (= (length method-qualifiers) 1)
         (typep (first method-qualifiers) '(integer 0*))))

;;; :arguments 利用の例
(define-method-combination progn-with-lock ()
```

```
        ((methods ()))
    (:arguments object)
    '(unwind-protect
        (progn (lock (object-lock ,object))
               ,@(mapcar #'(lambda (method)
                             '(call-method ,method ()))
                         methods))
        (unlock (object-lock ,object))))
```

● 注 意：

defgeneric の :method-combination オプションは，総称関数が特別なメソッド結合を用いることを指定する．:method-combination オプションの引数はメソッド結合方式の名前である．

● 参 照：

「8. メソッド選択と結合」
「8.4 組込みメソッド結合方式」
call-method
method-qualifiers
method-combination-error
invalid-method-error
defgeneric

defmethod

マクロ

● 目 的：

マクロ defmethod は総称関数上にメソッドを定義する．

(fboundp *function-specifier*) が nil なら，次のような項目についてデフォルト値をもった新しい総称関数が生成される．引数優先順位（それぞれの引数は引数リスト上で右側にあるものより優先する），総称関数のクラス（クラス standard-generic-function），メソッドのクラス（クラス standard-method），そしてメソッド結合方式（標準メソッド結合方式）．この総称関数のラムダリストは，定義しようとしているメソッドのラムダリストに適合するように作られる．もし defmethod フォームにキーワード引数が指定されていたら，総称関数のラムダリストも &key（ただし，キーワード引数は無し）をもったものになる．もし *function-specifier* が既存の通常の関数，マクロ，あるいは特殊形式の名前と同じならば，エラーが発せられる．

もしシンボルまたはリスト（setf *symbol*）で与えられる *function-specifier* を名前にもつ総称関数があれば，メソッドのラムダリストはこの総称関数のラムダリストと適合しなければならない．もしこの条件に反していればエラーが発せられる．ラムダリストの適合性に関する定義については，「7.4 1

つの総称関数に属するすべてのメソッドの適合ラムダリスト」を参照のこと.

● 構　文:

defmethod *function-specifier* {*method-qualifier*}* *specialized-lambda-list*　　[マクロ]
　　{*declaration* | *documentation*}* {*form*}*

function-specifier ::= {*symbol* | (setf *symbol*)}

method-qualifier ::= *non-nil-atom*

specialized-lambda-list
::=({*var* | (*var parameter-specializer-name*)}*
　[&optional {*var* | (*var* [*initform* {*supplied-p-parameter*}])}*]
　[&rest *var*]
　[&key {*var* | ({*var* | (*keyword var*)} [*initform* [*supplied-p-parameter*]])}*
　　[&allow-other-keys]]
　[&aux {*var* | (*var* [*initform*])}*])

parameter-specializer-name ::= *symbol* | (eql *eql-specializer-form*)

● 引　数:

引数 *function-specifier* は nil でないシンボルまたはリスト (setf *symbol*) である. これはメソッドが定義される総称関数の名前となる.

引数 *method-qualifier* は, メソッド結合を行なうときにメソッドを識別するためのオブジェクトである. メソッド修飾子は nil でないアトムである. メソッド結合方式に応じて, 何をメソッド修飾子として許すかさらに制限を加えてもよい. 標準メソッド結合方式は, 修飾されていないメソッド, そしてキーワード :before, :after, :around のいずれか1つを修飾子にもつメソッドを許す.

引数 *specialized-lambda-list* は通常の関数のラムダリストと似ているが, 必須引数の名前の代わりに特定化された引数 (specialized parameter) が書ける. これは (*valiable-name parameter-specializer-name*) のようなリストである. 必須引数だけが特定化できる. parameter-specializer-name として許されるのは, クラス名のシンボルまたは (eql *eql-specializer-form*) である. (eql *eql-specializer-form*) という引数特定子をもったメソッドが適用可能であるためには, 対応する引数が *eql-specializer-form* の値と *eql* の意味で等しくなければならない. 必須引数に *parameter-specializer-name* が指定されていないときは, 引数特定子として t という名前のクラスが省略されているとみなす. 詳しい議論は「7.2 メソッドの概要」を参照のこと.

引数 *form* にはメソッドのボディを記述する. メソッドのボディは暗黙的なブロックに囲まれる. もし *function-specifier* がシンボルならこのブロックは総称関数と同じ名前をもつ. もし *function-specifier* がリスト (setf *symbol*) ならブロックの名前は *symbol* となる.

● 値：

defmethod は結果としてメソッドオブジェクトを返す．

● 注　意：

生成されるメソッドオブジェクトのクラスは，メソッドが定義される総称関数に与えられたメソッドのクラスを指定するオプションによって決定される．

引数特定子と修飾子に関して一致するメソッドが総称関数に定義されていたら，defmethod は既存のメソッドを新しいもので置き換える．この文中の一致という言葉の定義に関しては「7.3 引数特定子と修飾子に関する一致」を参照のこと．

引数特定子は引数特定子名から求めることができるが，それについては「7.2 メソッドの概要」を参照のこと．

マクロ defmethod はマクロ展開時に特定化された引数を「参照」する（ignore に関する記述 *Common Lisp: The Language*, p. 160 を参照）．これには陽に引数特定子名として t を指定した引数も含む．このことは，メソッドのボディが特定化された引数を参照しなくてもコンパイラは警告を発しないということである．この点においては，引数特定子名として t を指定した引数と，引数特定子をもたない引数の間に相違があるので注意せよ．

● 参　照：

「7.2 メソッドの概要」
「7.4 1つの総称関数に属するすべてのメソッドの適合ラムダリスト」
「7.3 引数特定子と修飾子に関する一致」

---

describe　　標準総称関数

● 目　的：

Common Lisp の関数 describe は総称関数として置き換える．総称関数 describe は与えられたオブジェクトに関する情報を標準出力に印字する．

各処理系はクラス standard-object に関するメソッド，そして常に適用可能メソッドが存在することを保証するに十分なメソッドを用意する必要がある．その他のクラスに関するメソッドの追加は各処理系にまかされる．ユーザは，もし処理系によって用意されたメソッドを継承することを望まなければ，自分の定義したクラスについて describe にメソッドを定義すればよい．これらのメソッドは *CLtL* で規定された describe の仕様に従わなければならない．

● 構　文：

describe *object*　　［総称関数］

● 既定義メソッド：

describe (*object* standard-object)　[基本メソッド]

● 引　数：

引数 *object* は，任意の Common Lisp オブジェクトである．

● 値：

総称関数 describe は値を返さない．

documentation, (setf documentation)　標準総称関数

● 目　的：

Common Lisp の関数 documentation は総称関数として置き換える．総称関数 documentation は与えられたオブジェクトに関する文書文字列を返す．もしそれがなければ，nil を返す．

総称関数 (setf documentation) は文書情報を更新するために用いる．

● 構　文：

documentation *x* &optional *doc-type*　[総称関数]

(setf documentation) *new-value x* &optional *doc-type*　[総称関数]

● 既定義メソッド：

documentation (*method* standard-method) &optional *doc-type*　[基本メソッド]

(setf documentation) *new-value* (*method* standard-method) &optional *doc-type*　[基本メソッド]

documentation (*generic-function* standard-generic-function) &optional *doc-type*　[基本メソッド]

(setf documentation) *new-value* (*generic-function* standard-generic-function) &optional *doc-type*　[基本メソッド]

documentation (*class* standard-class) &optional *doc-type*　[基本メソッド]

(setf documentation) *new-value* (*class* standard-class) &optional *doc-type*　[基本メソッド]

documentation (*method-combination* method-combination)　[基本メソッド]

&optional *doc-type*

(setf documentation) *new-value* [基本メソッド]
(*method-combination* method-combination)
&optional *doc-type*

documentation (*slot-description* standard-slot-description) [基本メソッド]
&optional *doc-type*

(setf documentation) *new-value* [基本メソッド]
(*slot-description* standard-slot-description)
&optional *doc-type*

documentation (*symbol* symbol) &optional *doc-type* [基本メソッド]

(setf documentation) *new-value* (*symbol* symbol) [基本メソッド]
&optional *doc-type*

documentation (*list* list) &optional *doc-type* [基本メソッド]

(setf documentation) *new-value* (*list* list) [基本メソッド]
&optional *doc-type*

● 引 数：

documentation の第一引数は，シンボル，関数指定子のリスト（setf *symbol*），メソッドオブジェクト，クラスオブジェクト，総称関数オブジェクト，メソッド結合オブジェクトまたはスロット記述オブジェクトのいずれかである．

- もし第一引数がメソッドオブジェクト，クラスオブジェクト，総称関数オブジェクト，メソッド結合オブジェクト，スロット記述オブジェクトのいずれかであれば，第二引数を与えてはならない．もし第二引数を与えるとエラーを発する．
- もし第一引数がシンボルまたはリスト（setf *symbol*）であれば，第二引数が必要である．
  - 式（documentation *symbol* 'function）および（documentation '(setf *symbol*) 'function）はそのシンボルまたはリストにより名前付けられた関数，総称関数，特殊形式，またはマクロの文書文字列を返す．
  - 式（documentation *symbol* 'variable）はそのシンボルにより名前付けられた動的変数あるいは定数の文書文字列を返す．
  - 式(documentation *symbol* 'structure)はそのシンボルにより名前付けられた defstruct 構造体の文書文字列を返す．
  - 式（documentation *symbol* 'type）はそのシンボルにより名前付けられたクラスオブジェクトが存在すれば，その文書文字列を返す．もしそのようなクラスが存在しなければ，そのシンボルにより名前付けられた型指定子の文書文字列を返す．
  - 式（documentation *symbol* 'setf）はそのシンボルに関連付けられた defsetf または

define-setf-method の定義の文書文字列を返す.

— 式 (documentation *symbol* 'method-combination) はそのシンボルにより名前付けられたメソッド結合方式の文書文字列を返す.

各処理系は第二引数として使用できるシンボルを拡張してもよい. もし処理系でそのシンボルが使用できない場合は, エラーを発する必要がある.

● 値:

与えられたオブジェクトに関連付けられた文書文字列が存在すればそれを返し, なければ nil を返す.

---

ensure-generic-function　　関　数

● 目　的:

関数 ensure-generic-function はメソッドをもたない大域的な総称関数を定義するため, または大域的な総称関数全体に関するオプションや宣言を指定または変更するために用いる.

もし (fboundp *function-specifier*) が nil なら, 新しい総称関数が生成される. もし (symbol-function *function-specifier*) が通常の関数, マクロまたは特殊形式ならばエラーを発する.

もし *function-specifier* が既存の総称関数であり, :argument-precedence-order, :declare, :documentation, :method-combination のいずれかの引数が異なっていれば, その総称関数は新しい値をもつように修正される.

もし *function-specifier* が既存の総称関数であり, 引数 :lambda-list の値が異なっている場合は, 両者の値が適合しているかあるいはメソッドをもたなければ, ラムダリストは引数に与えた値に変更される. そうでなければエラーが発せられる.

もし *function-specifier* が既存の総称関数であり, 引数 :generic-function-class の値が異なっている場合は, 新しい総称関数のクラスが古い総称関数のクラスと互換性があれば, 総称関数のクラスを変えるために change-class を呼ぶ. 互換性がなければエラーを発する.

もし *function-specifier* が既存の総称関数であり, 引数 :method-class の値が異なっている場合は, 総称関数のもっている値は変えられるが, 既存のメソッドのクラスは変更されない.

● 構　文:

ensure-generic-function *function-specifier*　　[関　数]

```
                    &key :lambda-list
                         :argument-precedence-order
                         :declare
                         :documentation
```

```
:generic-function-class
:method-combination
:method-class
:environment
```

function-specifier ::= {*symbol* | (setf *symbol*)}

● 引 数:

引数 *function-specifier* はシンボルもしくはリスト (setf *symbol*) である.

キーワード引数は, defgeneric の *option* 引数と対応する. しかしながら, 引数 :method-class および引数 :generic-function-class はクラス名でもクラスオブジェクトでもよい.

引数 :environment は, マクロ展開関数における &environment 引数と同じである. これは主に, コンパイル時と実行時の環境を識別するのに用いられる.

引数 :method-combination はメソッド結合オブジェクトである.

● 値:

総称関数オブジェクトが返される.

● 参 照:

defgeneric

---

find-class 関 数

● 目 的:

関数 find-class は, 与えられた環境で与えられたシンボルが名前となっているクラスオブジェクトを返す.

● 構 文:

find-class *symbol* &optional *errorp environment* [関 数]

● 引 数:

第一引数 *symbol* はシンボルである.

クラスが見つからなかった場合は, 付加引数 *errorp* が与えられなかったか, 与えられていても値が nil 以外であれば, エラーを発する. *errorp* が nil であれば, クラスが見つからなくてもエラーにはならないで, nil を返す. *errorp* のデフォルト値は t である.

付加引数 *environment* は，マクロ展開関数における &environment 引数と同じである．これは主に，コンパイル時と実行時の環境を識別するのに用いられる．

● **値：**

find-class の結果は，与えられたシンボルによって名付けられたクラスオブジェクトである．

● **注　意：**

シンボルに関連付けられたクラスは，setf を find-class と共に用いることにより変更できる．ユーザが，*Common Lisp : The Language* で型指定子として定義されたシンボルに関連付けられたクラスを変更しようとした場合の結果は未定義である．「5. 型とクラスの統合」を参照のこと．

find-method　　標準総称関数

● **目　的：**

総称関数 find-method は，引数として与えられた総称関数のメソッドの中から，与えられたメソッド修飾子と引数特定子において一致するメソッドオブジェクトを求める．ここでいう一致の意味については，「7.3 引数特定子と修飾子に関する一致」を参照のこと．

● **構　文：**

find-method *generic-function method-qualifiers*　　[総称関数]
　　specializers &optional *errorp*

● **既定義メソッド：**

find-method (*generic-function* standard-generic-function)　　[基本メソッド]
　　method-qualifiers specializers &optional *errorp*

● **引　数：**

引数 *generic-function* は総称関数である．

引数 *method-qualifiers* は，取り出そうとするメソッドのメソッド修飾子のリストであり，その順序は意味をもつ．

引数 *specializers* は，メソッドの引数特定子のリストであり，総称関数の必須引数の個数と同じ長さのリストでなければならない．そうでない場合はエラーが発せられる．与えられた総称関数のデフォルトメソッドを得るためには，その総称関数の必須引数の個数分の t という名前のクラスを要素とするリストが与えられる必要がある．

該当するメソッドが見つからなかった場合は，付加引数 *errorp* が与えられなかったか，与えられて

いても値が nil 以外であれば，エラーを発する．*errorp* が nil であれば，クラスが見つからなくてもエラーにはならないで，nil を返す．*errorp* のデフォルト値は t である．

● 値：

find-method の結果は，与えられたメソッド修飾子と引数特定子をもつメソッドオブジェクトである．

● 参照：

「7.3 引数特定子と修飾子に関する一致」

function-keywords　標準総称関数

● 目的：

function-keywords はメソッドのキーワード引数を求める．

● 構文：

function-keywords *method*　［総称関数］

● 既定義メソッド：

function-keywords (*method* standard-method)　［基本メソッド］

● 引数：

引数 *method* はメソッドオブジェクトである．

● 値：

function-keywords は陽に記述されたキーワードのリストと &allow-other-keys が指定されたかどうかを示す論理値を結果として返す．

generic-flet　特殊形式

● 目的：

特殊形式 generic-flet は Common Lisp の flet と似ている．generic-flet は新しい総称関数を生成し，新しい関数定義のレキシカルな束縛を確立する．これらの総称関数にはメソッド記述で指定されたメソッドが定義される．

特殊形式 generic-flet は局所的に意味のある名前をもった総称関数を定義し，これらの関数束縛のもとで一連のフォームを実行するために用いる．局所的な総称関数は任意個数定義できる．

generic-flet によって定義された総称関数の名前はレキシカルなスコープをもつ．すなわち，その名前は generic-flet のボディの中だけの局所的な定義である．generic-flet のボディの中から関数を参照したとき，その名前の関数が generic-flet によって局所的な束縛をもっているならば，たとえ同じ名前の関数が大域的な関数として定義されていても，局所的な関数のほうを参照する．しかし，このような総称関数名の束縛によって作られるスコープは generic-flet のボディの中でだけ有効で，局所的な総称関数の定義自体は含まない．すなわち局所的な総称関数に属するメソッドの中から関数を参照したとき，たとえ同じ名前の局所的な総称関数があっても大域的な関数を参照する．したがって，generic-flet を用いて再帰的な関数を定義することはできない．

● **構　文：**

generic-flet ({(*function-specifier lambda-list*　　[特殊形式]
　　[↓*option* | *method-description**])}*)
　　{*form*}*

function-specifier ::= {*symbol* | (setf *symbol*)}

lambda-list ::= ({*var*}*
　　[&optional {*var* | (*var*)}*]
　　[&rest *var*]
　　[&key {*var* | ({*var* | (*keyword var*)})}*
　　[&allow-other-keys]])

option ::= (:argument-precedence-order {*parameter-name*}+) |
　　(declare {*declaration*}+) |
　　(:documentation *string*) |
　　(:method-combination *symbol* {*arg*}*) |
　　(:generic-function-class *class-name*) |
　　(:method-class *class-name*)

method-description ::= (:method {*method-qualifier*}* *specialized-lambda-list*
　　{*declaration* | *documentation*}* {*form*}*)

● **引　数：**

引数 *function-specifier*, *lambda-list*, *option*, *method-qualifier*, そして *specialized-lambda-list* は defgeneric のものと同様である．

generic-flet の局所的なメソッドの定義の形式は defmethod のメソッド定義の形式と同じである．

おのおののメソッドのボディは暗黙的なブロックに囲まれる．もし *function-specifier* がシンボルな

らこのブロックは総称関数と同じ名前をもつ．もし *function-specifier* がリスト (setf *symbol*) ならブロックの名前は *symbol* となる．

● 値：

特殊形式 generic-flet は結果として最後に実行されたフォームの返す値または多値を返す．フォームをもっていなかったら nil を返す．

● 参 照：

generic-labels
defmethod
defgeneric
generic-function

generic-function

マクロ

● 目 的：

マクロ generic-function は名無しの総称関数を生成する．総称関数にはメソッド記述に指定されたメソッドが定義される．

● 構 文：

generic-function *lambda-list* [マクロ]
[↓ *option* | *method-description**]

lambda-list ::= ({*var*}*
[&optional {*var* | (*var*)}*]
[&rest *var*]
[&key {*var* | ({*var* | (*keyword var*)})}*
[&allow-other-keys]])

option ::= (:argument-precedence-order {*parameter-name*}+) |
(declare {*declaration*}+) |
(:documentation *string*) |
(:method-combination *symbol* {*arg*}*) |
(:generic-function-class *class-name*) |
(:method-class *class-name*)

method-description ::= (:method {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*)

● 引　数：

引数 *option*, *method-qualifier*, そして *specialized-lambda-list* は defgeneric のものと同様である.

● 値：

マクロ generic-function は結果として総称関数オブジェクトを返す.

● 注　意：

メソッド記述が無ければ, メソッドをもたない総称関数が生成される.

● 参　照：

defgeneric
generic-flet
generic-labels
defmethod

generic-labels　　特殊形式

● 目　的：

特殊形式 generic-labels は Common Lisp の特殊形式 labels と似ている. generic-labels は新しい総称関数を生成し, 新しい関数定義のレキシカルな束縛を確立する. これらの総称関数にはメソッド記述に指定されたメソッドが生成される.

特殊形式 generic-labels は局所的に意味のある名前をもった総称関数を定義し, これらの関数束縛のもとで一連のフォームを実行するために用いる. 局所的な総称関数は任意個数定義できる.

generic-labels によって定義された総称関数の名前はレキシカルなスコープをもつ. すなわち, その名前は generic-labels のボディの中だけの局所的な定義である. generic-labels のボディの中から関数を参照したとき, その名前の関数が generic-labels によって局所的な束縛をもっているならば, たとえ同じ名前の関数が大域的な関数として定義されていても, 局所的な関数のほうを参照する. このような総称関数名の束縛によって作られるスコープは generic-labels のボディの中だけでなく, 局所的な総称関数に属するメソッドのボディの中自体でも有効である.

● 構　文：

generic-labels　({(*function-specifier lambda-list*　　[特殊形式]
　　[↓*option* | *method-description**])}*)
　　{*form*}*
function-specifier ::= {*symbol* | (setf *symbol*)}

lambda-list ::= ({*var*}*
[&optional {*var* | (*var*)}*]
[&rest *var*]
[&key {*var* | ({*var* | (*keyword var*)})}*
[&allow-other-keys]])

option ::= (:argument-precedence-order {*parameter-name*}+) |
(declare {*declaration*}+) |
(:documentation *string*) |
(:method-combination *symbol* {*arg*}*) |
(:generic-function-class *class-name*) |
(:method-class *class-name*)

method-description ::= (:method {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*)

● 引 数：

引数 *function-specifier*, *lambda-list*, *option*, *method-qualifier*, そして *specialized-lambda-list* は defgeneric のそれと同様である.

generic-labels の局所的なメソッドの定義の形式は defmethod のメソッド定義の形式と同じである.

おのおののメソッドのボディは暗黙的なブロックに囲まれる. もし *function-specifier* がシンボルならこのブロックは総称関数と同じ名前をもつ. もし *function-specifier* がリスト (setf *symbol*) ならブロックの名前は *symbol* となる.

● 値：

特殊形式 generic-labels は結果として最後に実行されたフォームの返す値または多値を返す. フォームをもっていなかったら nil を返す.

● 参 照：

generic-flet
defmethod
defgeneric
generic-function

initialize-instance　　標準総称関数

● 目　的：

総称関数 initialize-instance は make-instance から呼ばれ，新しく生成されたインスタンスを初期化する．総称関数 initialize-instance は新しいインスタンスとデフォルト初期化引数を引数として呼び出される．

システムが提供する initialize-instance の基本メソッドは，初期化引数リストの値と :initform フォームに従ってスロットを初期化する．実際の初期化は，インスタンス，t（これは初期化引数を与えられなかったすべてのスロットは :initform フォームの値に従って初期化することを示す），そしてデフォルト初期化引数を引数として shared-initialize を呼び出すことで行なう．

● 構　文：

initialize-instance *instance* &rest *initargs*　　［総称関数］

● 既定義メソッド：

initialize-instance (*instance* standard-object) &rest *initargs*　　［基本メソッド］

● 引　数：

引数 *instance* は初期化されるオブジェクトである．

引数 *initargs* は初期化引数の名前と値が交互に現われるリストである．

● 値：

結果として変更されたインスタンスが返される．

● 注　意：

プログラマは initialize-instance にメソッドを定義することで，インスタンスが初期化されるときに行ないたい動作を指定できる．もし initialize-instance に :after メソッドだけを定義すれば，これらはシステム提供の基本メソッドの後に走るので，initialize-instance のデフォルトの振舞いには影響を与えない．

● 参　照：

「10. オブジェクトの生成と初期化」
「10.4 初期化引数の規則」
「10.2 初期化引数の妥当性の宣言」

shared-initialize
make-instance
slot-boundp
slot-makunbound

invalid-method-error

関 数

● 目 的：

関数 invalid-method-error は，メソッド結合方式にとって有効でない修飾子をもつ適用可能メソッドがあったときに，エラーを発するために用いる．書式制御文字列とそれに対する任意の引数によって，エラー文が作られる．処理系は，付加的文脈情報をエラー文に追加するであろうから，invalid-method-error は，メソッド結合関数の動的エクステントの中だけで呼ばれるべきである．

define-method-combination フォームにおいて，メソッドがどの修飾子パターンの述語も満足しなければ，invalid-method-error が自動的に呼ばれる．メソッドにもっと制限を加えるようなメソッド結合関数は，有効でないメソッドを見つけたときは自分で invalid-method-error を呼ばなければならない．

● 構 文：

invalid-method-error *method format-string* &rest *args* [関 数]

● 引 数：

引数 *method* は，不適当とわかったメソッドオブジェクトである．

引数 *format-string* は，format に渡すことのできる書式制御文字列で，*args* はその書式に必要な任意の引数である．

● 注 意：

invalid-method-error が呼び出し元に戻るか，throw により広域脱出するかは処理系依存である．

● 参 照：

define-method-combination

make-instance

標準総称関数

● 目 的：

総称関数 make-instance は，クラス *class* から新しいインスタンスを生成して返す．

総称関数 make-instance は，「10. オブジェクトの生成と初期化」に述べているように使える．

● 構　文：

make-instance *class* &rest *initargs*　　[総称関数]

● 既定義メソッド：

make-instance (*class* standard-class) &rest *initargs*　　[基本メソッド]

make-instance (*class* symbol) &rest *initargs*　　[基本メソッド]

● 引　数：

引数 *class* はクラスオブジェクトまたはクラスの名前のシンボルである．残りの引数は初期化引数の名前と値が交互に現われるリストである．

上記2番目のメソッドが選択されると，そのメソッドは (find-class *class*) と *initargs* を引数にして1番目の make-instance を呼ぶ．

初期化引数は make-instance の中でチェックされる．「10. オブジェクトの生成と初期化」を参照．

● 値：

新しいインスタンスが返される．

● 注　意：

メタオブジェクトプロトコルを用いて，make-instance に新しいメソッドを定義し，オブジェクト生成のプロトコルを置き換えることができる．

● 参　照：

「10. オブジェクトの生成と初期化」
defclass
initialize-instance
class-of

make-instances-obsolete　　標準総称関数

● 目　的：

総称関数 make-instances-obsolete は，defclass がすでに存在する標準クラスを再定義し，インスタンスのアクセス可能な局所スロットの集合が変更された場合，またはスロットの格納の順序が変更され

た場合に，システムにより自動的に起動される．ユーザが陽に起動することも可能である．

総称関数 make-instances-obsolete は，クラスのインスタンスを更新するプロセスを開始する働きがある．更新の途中で総称関数 update-instance-for-redefined-class が起動される．

● 構　文：

make-instances-obsolete *class* ［総称関数］

● 既定義メソッド：

make-instances-obsolete (*class* standard-class) ［基本メソッド］

make-instances-obsolete (*class* symbol) ［基本メソッド］

● 引　数：

引数 *class* は，インスタンスを廃棄しようとしているクラスオブジェクト，もしくはクラスの名前であるようなシンボルである．

上記2番目のメソッドが選択された場合，そのメソッドは (find-class *class*) を引数にして make-instances-obsolete を起動する．

● 値：

修正されたクラスが返される． make-instances-obsolete の結果は，上記の最初のメソッドに与えられる引数 *class* と eq である．

● 参　照：

「11. クラスの再定義」

update-instance-for-redefined-class

method-combination-error　関　数

● 目　的：

関数 method-combination-error は，メソッド結合時に見つかった問題を知らせる．エラー文は書式制御文字列とそれに対する任意の引数によって作成される．処理系は，付加的文脈構造をエラー文に付加する必要があるだろうから， method-combination-error は，メソッド結合関数の動的エクステントの中でだけ呼ばれるべきである．

● 構　文：

method-combination-error *format-string* &rest *args* ［関　数］

● 引　数：

引数 *format-string* は，関数 format に渡すことのできる書式制御文字列であり，引数 *args* はその書式に必要な任意の引数である．

● 注　意：

関数 method-combination-error が呼び出し元に戻るか， throw を使って広域脱出するかは，処理系依存である．

● 参　照：

define-method-combination

---

method-qualifiers　　標準総称関数

● 目　的：

総称関数 method-qualifiers は与えられたメソッドの修飾子のリストを返す．

● 構　文：

method-qualifiers *method*　　[総称関数]

● 既定義メソッド：

method-qualifiers (*method* standard-method)　　[基本メソッド]

● 引　数：

引数 *method* はメソッドオブジェクトである．

● 値：

与えられたメソッドの修飾子のリストを返す．

● 例：

```
(setq methods (remove-duplicates methods
                                 :from-end t
                                 :key #'method-qualifiers
                                 :test #'equal))
```

● 参　照：

define-method-combination

next-method-p 関 数

● 目 的：

局所的に定義される関数 next-method-p は，メソッド定義フォームによって定義されたメソッドのボディの中で，次メソッドが存在するかどうかを決定するために用いる．

● 構 文：

next-method-p [関 数]

● 引 数：

next-method-p に引数は無い．

● 値：

結果として next-method-p は真か偽を返す．

● 注 意：

next-method-p は call-next-method と同様にレキシカルなスコープと無限エクステントをもつ．

● 参 照：

call-next-method

no-applicable-method 標準総称関数

● 目 的：

総称関数 no-applicable-method は，standard-generic-function をクラスとする総称関数が起動されたが，総称関数には適用できるメソッドが存在しない場合に呼ばれる．デフォルトメソッドはエラーを発する．

総称関数 no-applicable-method はプログラマによって呼ばれることは意図していない．プログラマはこの総称関数にメソッドを書くことが期待されている．

● 構 文：

no-applicable-method *generic-function* `&rest` *function-arguments* [総称関数]

● **既定義メソッド：**

no-applicable-method (*generic-function t*) &rest *function-arguments*　[基本メソッド]

● **引　数：**

引数 *generic-function* は，適用可能なメソッドが1つも見つからなかった総称関数オブジェクトである．

引数 *function-arguments* はその総称関数への引数のリストである．

---

no-next-method　標準総称関数

● **目　的：**

総称関数 no-next-method は，call-next-method が呼ばれたが，次メソッドが無いときに呼ばれる．システム提供の no-next-method メソッドはエラーを発する．

総称関数 no-next-method はプログラマによって呼ばれることは意図していない．プログラマはこの総称関数にメソッドを書くことが期待されている．

● **構　文：**

no-next-method *generic-function method* &rest *args*　[総称関数]

● **既定義メソッド**

no-next-method (*generic-function* standard-generic-function)　[基本メソッド]
(*method* standard-method)
&rest *args*

● **引　数：**

引数 *generic-function* は第二引数のメソッドが属する総称関数オブジェクトである．

引数 *method* は call-next-method を呼んだメソッドである．

引数 *args* は call-next-method に与えられた引数のリストである．

● **参　照：**

call-next-method

print-object 標準総称関数

● 目 的：

総称関数 print-object はオブジェクトの印字表現をストリームへ出力する．関数 print-object はプリントシステムから呼び出されるものであって，ユーザが呼び出すべきものではない．

各処理系はクラス standard-object に関するメソッドと常に適用可能メソッドが存在することを保証するのに十分なメソッドを用意する必要がある．その他のクラスに関するメソッドの追加は各処理系にまかされる．ユーザは，もし処理系によって用意されたメソッドを継承することを望まなければ，自分の定義したクラスについて print-object にメソッドを定義すればよい．

● 構 文：

print-object *object stream* [総称関数]

● 既定義メソッド：

print-object (*object* standard-object) *stream* [基本メソッド]

● 引 数：

第一引数は任意の Lisp オブジェクトである．

第二引数はストリームである．しかし t または nil を用いることはできない．

● 値：

関数 print-object は第一引数のオブジェクトを返す．

● 注 意：

print-object 上に定義されたメソッドは *CLtL* で定義している印字制御用の動的変数に従わなければならない．詳細は以下のとおりである．

- 各メソッドは *print-escape* を実現すること．
- リスト以外を扱うメソッドはたいてい *print-pretty* を無視することができる．
- *print-circle* はプリンタによって処理されるので，メソッドでは無視することができる．
- もし各メソッドが正確に１レベルの構造を処理し，多重構造になっている場合には write（または同等の関数）を再帰的に呼び出すならば，プリンタは *print-level* を自動的に処理する．オブジェクトが構成要素をもっているか（それゆえ印字する深さが *print-level* 以上のときに印字されるべきでない）どうかのプリンタの判断は処理系に依存する．ある処理系では print-object メソッドを呼び出しておらず，また他の処理系では呼び出している．オブジェクトが構成要素をもってい

るかどうかの判断はストリームに対して何を印字しようとしているかによる．

- 不定の長さの出力を行なうメソッドは *print-length* を処理しなければならないが，リスト以外のものを扱うほとんどのメソッドはそれを無視することができる．
- 変数 *print-base*, *print-radix*, *print-case*, *print-gensym*, *print-array* は特定の型に対して適用されるものであり，それらのオブジェクトのためのメソッドにより処理される．

もしこれらの規則に従わない場合の結果は未定義である．

一般的には，プリンタとメソッド print-object は，構造に対して再帰的に処理を行なう場合，印字制御用の変数の値を再束縛すべきではないが，このことは処理系に依存する．

ある処理系では，メソッド print-object に渡す引数 *stream* は元々のストリームではなく，プリンタを実現するための中間的なストリームになっている．したがってメソッドはこのストリームが何であるかに依存するべきでない．

すべての既存の印字関数（write, prin1, print, princ, pprint, write-to-string, prin1-to-string, princ-to-string, ~S, ~A 書式制御，数値以外のものに対する ~B, ~D, ~E, ~F, ~G, ~$, ~O, ~R, ~X 書式制御）は総称関数 print-object を使用するように変更する必要がある．各処理系は，以前に実現していた印字処理を，1つないしはそれ以上の print-object メソッドに置き換える必要がある．どのクラスが print-object メソッドをもつかということは正確には規定しない．デフォルトメソッドを1つ用意して，システム定義のクラスはすべてのこのメソッドを継承するような実現方法でもよい．

---

reinitialize-instance　　標準総称関数

● **目　的：**

総称関数 reinitialize-instance は，初期化引数に従って局所スロットの値を変更するために用いる．この総称関数はメタオブジェクトプロトコルによって呼ばれる．ユーザが呼ぶこともできる．

システム提供の reinitialize-instance の基本メソッドは，初期化引数の有効性をチェックし，有効性の宣言されていない初期化引数があればエラーを発する．次にこのメソッドは，インスタンス，nil（:initform フォームによる初期化は行なわない），受け取った初期化引数の3つを引数にして総称関数 shared-initialize を呼ぶ．

● **構　文：**

reinitialize-instance *instance* &rest *initargs*　　[総称関数]

● **既定義メソッド：**

reinitialize-instance (*instance* standard-object) &rest *initargs*　　[基本メソッド]

● 引 数：

引数 *instance* は初期化するオブジェクトである．

引数 *initargs* は初期化引数の名前と値が交互に現われるリストである．

● 値：

修正されたインスタンスを結果として返す．

● 注 意：

初期化引数の妥当性は defclass の :initarg オプション，または reinitialize-instance あるいは shared-initialize にメソッドを定義することで宣言される．reinitialize-instance と shared-initialize 上に定義されたすべてのメソッドのキーワード引数の名前は，そのメソッドが適用可能なすべてのクラスに対して有効な初期化引数として宣言される．

● 参 照：

「13. インスタンスの再初期化」
「10. 4 初期化引数の規則」
「10. 2 初期化引数の妥当性の宣言」
initialize-instance
shared-initialize
update-instance-for-redefined-class
update-instance-for-different-class
slot-boundp
slot-makunbound

remove-method 標準総称関数

● 目 的：

総称関数 remove-method は総称関数からメソッドを削除する．これは破壊的に総称関数を修正し，修正された総称関数を返す．

● 構 文：

remove-method *generic-function method* ［総称関数］

● 既定義メソッド：

remove-method (*generic-function* standard-generic-function) ［基本メソッド］

method

● 引　数：

引数 *generic-function* は総称関数オブジェクトである．

引数 *method* はメソッドオブジェクトである．総称関数 remove-method は引数 *method* に与えられたメソッドが引数 *generic-function* に無くてもエラーを発しない．

● 値：

修正された総称関数が返される．remove-method の結果は引数 *generic-function* と eq である．

● 参　照：

find-method

shared-initialize　　標準総称関数

● 目　的：

総称関数 shared-initialize は，初期化引数と :initform フォームを用いてインスタンスのスロットを埋めるために使う．この総称関数は，インスタンスの生成時，インスタンスの再初期化時，クラスの再定義に伴うインスタンスの更新時，クラスの変更に伴うインスタンスの更新時に呼ばれる．この総称関数は，initialize-instance, reinitialize-instance, update-instance-for-redefined-class, そして update-instance-for-different-class 上に定義されたシステム提供の基本メソッドから呼ばれる．

総称関数 initialize-instance は，初期化されるインスタンス，そのインスタンスのアクセス可能なスロットの名前の集合の指定，そして任意個の初期化引数を引数に取る．3番目以降の引数は初期化引数リストになっていなければならない．システム提供の shared-initialize の基本メソッドは，初期化引数と :initform フォームに従ってスロットを初期化する．第二引数は，初期化引数の与えられなかったスロットの中で，:initform フォームに従って初期化すべきスロットを指定する．

システム提供の基本メソッドは，局所スロットであるか共有スロットであるかによらず，次のように振る舞う．

- 初期化引数リスト中にそのスロットの初期化引数があれば，そのスロットには，たとえこのメソッドが走る前に値があっても，初期化引数の値が設定される．
- 第二引数に指定されたスロットの中で，この時点でまだ未束縛なスロットがあれば，:initform フォームに従って初期化される．:initform フォームは defclass フォームが定義されたときのレキシカルな環境で評価され，その値がスロットに設定される．たとえば :before メソッドがスロットに値を設定していれば，:initform フォームはそのスロットの値を埋めるのには使われない．
- 「10.4 初期化引数の規則」で述べた規則に従う．

● 構　文：

shared-initialize *instance slot-names* &rest *initargs*　　[総称関数]

● 既定義メソッド：

shared-initialize (*instance* standard-object) *slot-names* &rest *initargs*　　[基本メソッド]

● 引　数：

引数 *instance* は初期化されるインスタンスである．

引数 *slot-names* は，初期化引数が与えられなかったときに，:initform フォームに従って初期化すべきスロットを指定する．これには次の3つのいずれかを指定する．

- スロット名のリスト．リストに示されるスロット名の集合を表わす．
- nil．スロット名の空集合を表わす．
- t．インスタンスのもっているすべてのスロットを表わす．

引数 *initargs* は初期化引数の名前と値が交互に現われるリストである．

● 値：

修正されたインスタンスを結果として返す．

● 注　意：

初期化引数の妥当性は，defclass の :initarg オプションあるいは shared-initialize にメソッドを定義することで宣言される．shared-initialize 上に定義されたすべてのメソッドのキーワード引数の名前は，そのメソッドが適用可能なすべてのクラスに対して初期化引数として宣言される．

処理系は，:initform フォームが副作用を起こさず副作用に依存しない場合は，スロットの初期化を initialize-instance の基本メソッドにまかせないで，これらのフォームをどの initialize-instance が走るよりも前に評価してあらかじめスロットに設定しておく最適化をしてもよい．(この最適化は allocate-instance メソッドがインスタンスのプロトタイプをコピーするようにすればできるだろう)．

処理系は，スロットの初期化引数に対応するデフォルト初期値フォームについて最適化してもよい．これは，完全な初期化引数リストを受け取るメソッドが standard-object に定義されたものしかない場合は，完全な初期化引数リストを実際には生成しないことで行なう．この場合，デフォルト初期値フォームは :initform フォームのように取り扱うことが可能である．この最適化はパフォーマンスを改善するだけで，見かけ上の結果には影響がない．

● 参　照：

「10. オブジェクトの生成と初期化」

「10.4 初期化引数の規則」
「10.2 初期化引数の妥当性の宣言」
initialize-instance
reinitialize-instance
update-instance-for-redefined-class
update-instance-for-different-class
slot-boundp
slot-makunbound

slot-boundp　　関　数

● **目　的：**

関数 slot-boundp は与えられたインスタンスのスロットが束縛されているかどうかテストする.

● **構　文：**

slot-boundp *instance slot-name*　　[関　数]

● **引　数：**

slot-boundp の引数はインスタンスとスロットの名前である.

● **値：**

slot-boundp は真か偽を返す.

● **注　意：**

関数 slot-boundp は, initialize-instance の :after メソッドで, 未束縛なスロットだけを初期化することを可能にする.

与えられた名前のスロットがインスタンスに存在しなかったら, slot-missing が次のように呼ばれる. (slot-missing (class-of *instance*) *instance slot-name* 'slot-boundp)

slot-boundp は slot-boundp-using-class を用いて実現されている.

● **参　照：**

slot-missing

slot-exists-p

関 数

● 目 的：

関数 slot-exists-p は，オブジェクトが与えられた名前のスロットをもっているかどうかテストする．

● 構 文：

slot-exists-p *object slot-name* ［関 数］

● 引 数：

引数 *object* は任意のオブジェクトである．引数 *slot-name* はシンボルである．

● 値：

関数 slot-exists-p は真か偽を返す．

● 注 意：

関数 slot-exists-p は slot-exists-p-using-class を用いて実現されている．

slot-makunbound

関 数

● 目 的：

関数 slot-makunbound はインスタンスのスロットを未束縛にする．

● 構 文：

slot-makunbound *instance slot-name* ［関 数］

● 引 数：

slot-makunbound の引数はインスタンスとスロット名である．

● 値：

インスタンスが結果として返される．

● 注　意：

もし与えられた名前のスロットがインスタンスに存在しない場合，slot-missing が次のように呼ばれる．(slot-missing (class-of *instance*) *instance slot-name* 'slot-makunbound)

関数 slot-makunbound は slot-makunbound-using-class を用いて実現されている．

● 参　照：

slot-missing

---

slot-missing　　標準総称関数

● 目　的：

総称関数 slot-missing は，standard-class をメタクラスとするオブジェクトのスロットをアクセスしたとき，与えられた名前のスロットがそのクラスにない場合に起動される．slot-missing のデフォルトメソッドはエラーを発する．

総称関数 slot-missing はプログラマによって呼ばれることは意図していない．プログラマはこの総称関数にメソッドを書くことが期待されている．

● 構　文：

slot-missing *class object slot-name operation* &optional *new-value*　　[総称関数]

● 既定義メソッド：

slot-missing (*class t*) *object slot-name operation* &optional *new-value*　[基本メソッド]

● 引　数：

slot-missing の必須引数は，アクセスされたオブジェクトのクラス，オブジェクト，スロット名，そして slot-missing を引き起こした操作を示すシンボルである．slot-missing の付加引数は，その操作がそのスロットに値を設定しようとしていた場合に用いられる．

● 値：

slot-missing に定義されたメソッドが値を返すならば，その値は slot-missing を引き起こした元の関数の返す値となる．

● 注　意：

総称関数 slot-missing は，slot-value，(setf slot-value)，slot-boundp，slot-makunbound の実行

中に呼ばれる可能性がある．これらの操作に対して，引数 *operation* に与えられるシンボルはそれぞれ，slot-value，setf，slot-boundp，slot-makunbound である．

引数の集合（インスタンスのクラスを含む）は，slot-missing に対してメタクラスごとのメソッドを定義することを可能にする．

slot-unbound　　標準総称関数

● 目　的：

総称関数 slot-unbound は，standard-class をメタクラスとするインスタンスの未束縛なスロットを参照した場合に起動される．slot-unbound のデフォルトメソッドはエラーを発する．

総称関数 slot-unbound はプログラマによって呼ばれることは意図していない．プログラマはこの総称関数にメソッドを書くことが期待されている．総称関数 slot-unbound は関数 slot-value-using-class だけから呼ばれる．したがって，間接的に slot-value から呼ばれる．

● 構　文：

slot-unbound *class instance slot-name*　　[総称関数]

● 既定義メソッド：

slot-unbound (*class t*) *instance slot-name*　　[基本メソッド]

● 引　数：

slot-unbound の引数は，スロットがアクセスされたインスタンスのクラス，インスタンス自身，そしてスロット名である．

● 値：

slot-unbound に定義されたメソッドが値を返すならば，その値は slot-unbound を引き起こした元の関数の返す値となる．

● 注　意：

スロットが未束縛の状態になるのは，スロット記述子に :initform の指定が無くかつスロットに値が設定されていない場合，またはスロットに対して slot-makunbound が呼ばれた場合である．

● 参　照：

slot-makunbound

slot-value　　関　数

● 目　的：

関数 slot-value は与えられたオブジェクトのスロット *slot-name* の内容を返す．もしその名前のスロットが無かったら slot-missing が呼ばれる．もしスロットが未束縛なら slot-unbound が呼ばれる．

マクロ setf を slot-value と用いることでスロットの値を変更することができる．

● 構　文：

slot-value *object slot-name*　　[関　数]

● 引　数：

引数はオブジェクトとスロット名である．

● 値：

与えられたスロット slot-name の内容を値として返す．

● 注　意：

スロットの値を読もうとして，与えられたスロット名がインスタンスに存在しなかったらslot-missingが次のように呼ばれる．(slot-missing (class-of *instance*) *instance slot-name* 'slot-value)

スロットに値を書こうとして，与えられたスロット名がインスタンスに存在しなかったら slot-missing が次のように呼ばれる．(slot-missing (class-of *instance*) *instance slot-name* 'setf *new-value*)

関数 slot-value は slot-value-using-class を用いて実現されている．

インライン展開によって slot-value の実現を最適化できる．

● 参　照：

slot-missing
slot-unbound

symbol-macrolet　　マクロ

● 目　的：

マクロ symbol-macrolet は静的な有効範囲において，フォームの代わりに変数名を使えるようにす

る.

● 構 文:

symbol-macrolet ((*symbol expansion*)*) &body *body* [マクロ]

● 引 数:

引数 *symbol* は，引数 *expansion* により指定されるフォームが関連付けされるシンボルを指定する.

● 値:

引数 *body* に指定されたフォームを実行した結果を返す.

● 例:

```
(symbol-macrolet ((x 'foo))
  (list x (let ((x 'bar)) x)))

;;;結果は (foo bar) であり，(foo foo) ではない.
;;; (list 'foo (let ((x 'bar)) x)) と展開され
;;; (list 'foo (let (('foo 'bar)) 'foo)) とはならない

(symbol-macrolet ((x (1+ x)))
  (print x))

;;; (print (1+ x)) と展開され
;;; (print (1+ (1+ (1+.....とはならない.
```

● 注 意:

symbol-macrolet の静的な範囲は *body* であり，*expansion* は含まない.

symbol-macrolet の静的な範囲で変数として参照される *symbol* は *expansion* (*expansion* の評価された結果ではない) で置換される.

symbol-macrolet の効果は let でシャドウされる. つまり，body で囲まれた静的な束縛の範囲における *symbol* についてのみ置換を行なう.

マクロ symbol-macrolet は with-slots を実現するために用いる基本的な機構である.

symbol-macrolet のボディを展開するとき，指定された変数に値を代入するために使用する setq は setf に置き換えられる.

● 参 照:

with-slots

update-instance-for-different-class　　標準総称関数

● 目　的：

総称関数 update-instance-for-different-class はプログラマによって呼ばれることは意図していない．プログラマはこの総称関数にメソッドを書くことが期待されている．総称関数 update-instance-for-different-class は change-class からだけ呼ばれる．

システム定義の update-instance-for-different-class の基本メソッドは，初期化引数の妥当性をチェックし，もし妥当性の宣言されていない初期化引数があればエラーを発する．次に初期化引数に従ってスロットを初期化し，新しく追加されたスロットは :initform フォームに従って初期化する．update-instance-for-different-class は，インスタンス，新しく追加されたスロットの名前のリスト，このメソッドに与えられた初期化引数を引数として shared-initialize を呼ぶ．新しく追加されたスロットとは，古いクラスの定義にその名前がない局所スロットである．

update-instance-for-different-class には，インスタンスが更新されるときに行なう動作を定義する．もし update-instance-for-different-class に :after メソッドだけを定義するなら，これらは初期化のためのシステム提供の基本メソッドの後に起動されるので，update-instance-for-different-class のデフォルトの振舞いには影響を与えない．

● 構　文：

update-instance-for-different-class *previous current* &rest *initargs*　　[総称関数]

● 既定義メソッド：

update-instance-for-different-class (*previous* standard-object)　　[基本メソッド]
(*current* standard-object)
&rest *initargs*

● 引　数：

update-instance-for-different-class の引数は change-class によって計算される．change-class があるインスタンスを引数にして起動されたとき，まずそのインスタンスのコピーが作られる．次に change-class はオリジナルのインスタンスを破壊的に変更する．update-instance-for-different-class の引数 *previous* はコピーのほうであり．スロットの古い値を一時的にもっている．この引数は change-class の中でだけ有効な動的エクステントをもっていて，update-instance-for-different-class から戻った後に参照されると，結果は未定義である．update-instance-for-different-class の第二引数 *current* は，変更されたオリジナルのインスタンスである．

引数 *previous* は slot-value, with-slots, リーダ総称関数, またはオリジナルのクラスのインスタンスに適用可能なメソッドを使って古いスロットの値を読むために使われることを意図している.

引数 *initargs* は, 初期化引数名と値が交互に現われるリストである.

● 値:

update-instance-for-different-class が返す値を change-class は無視する.

● 例:

change-class の例を参照せよ.

● 注 意:

初期化引数の妥当性は, defclass のオプション :initarg により, または update-instance-for-different-class か shared-initialize にメソッドを定義することにより行なわれる. update-instance-for-different-class または shared-initialize 上に定義されたメソッドのラムダリストのキーワード引数の名前は, そのメソッドが適用可能なすべてのクラスの初期化引数として有効になる.

update-instance-for-different-class にメソッドを定義することで change-class とは異なったようにスロットを初期化できる. change-class のデフォルトの振舞いについては「12. インスタンスのクラスの変更」で説明している.

● 参 照:

「12. インスタンスのクラスの変更」
「10. 4 初期化引数の規則」
「10. 2 初期化引数の妥当性の宣言」
change-class
shared-initialize

update-instance-for-redefined-class 標準総称関数

● 目 的:

総称関数 update-instance-for-redefined-class はプログラマによって呼ばれることは意図していない. プログラマはこの総称関数にメソッドを書くことが期待されている. 総称関数 update-instance-for-redefined-class は make-instances-obsolete によって起動されるメカニズムによって呼ばれる.

システム定義の update-instance-for-redefined-class の基本メソッドは, 初期化引数の妥当性をチェックし, もし妥当性の宣言されていない初期化引数があればエラーを発する. 次に初期化引数に従ってスロットを初期化し, 新しく追加されたスロットは :initform フォームに従って初期化する. update-

instance-for-redefined-class は，インスタンス，新しく追加されたスロットの名前のリスト，このメソッドに与えられた初期化引数を引数として shared-initialize を呼ぶ．新しく追加されたスロットとは，古いクラスの定義にその名前が無い局所スロットである．

● 構　文：

update-instance-for-redefined-class *instance* [総称関数]
added-slots discarded-slots
property-list
&rest *initargs*

● 既定義メソッド：

update-instance-for-redefined-class (*instance* standard-object) [基本メソッド]
added-slots discarded-slots
property-list
&rest *initargs*

● 引　数：

make-instances-obsolete が起動されたとき，もしくはクラスが再定義されインスタンスが更新される場合に，元のインスタンスのすべての破棄されたスロットの名前と，そのスロットの以前の値をもったプロパティリストが作られる．インスタンスの構造は，現在のクラス定義に合うように変換される．update-instance-for-redefined-class の引数は，変換されたインスタンス，インスタンスに新しく付け加えられたスロット名のリスト，インスタンスから破棄された古いスロット名のリスト，そして破棄されたスロットの中で値をもっていたスロットの名前と値をもつプロパティリストである．この破棄されたスロットのリストには，古いクラスでは局所スロットであったが，新しいクラスでは共有スロットになったスロットも含まれる．

引数 *initargs* は，初期化引数名と値が交互に現われるリストである．

● 値：

update-instance-for-redefined-class が返す値は無視される．

● 注　意：

初期化引数の妥当性は，defclass の :initarg オプションにより，または update-instance-for-redefined-class か shared-initialize にメソッドを定義することにより行なわれる．update-instance-for-redefined-class または shared-initialize 上に定義されたメソッドのラムダリストのキーワード引数の名前は，そのメソッドが適用可能なすべてのクラスの初期化引数として有効になる．

● 例：

```
(defclass position () ())

(defclass x-y-position (position)
   ((x :init-form 0 :accessor position-x)
    (y :init-form 0 :accessor position-y)))

;;;直交座標よりも極座標のほうが便利なことが判明したので，表現を変更し
;;;幾つかのアクセサメソッドを付け加える

(defmethod update-instance-for-redefined-class :before
   ((pos x-y-position) added deleted plist &key)
   ;; x-y座標を極座標に変換して新しいスロットに代入する.
   (let ((x (getf plist 'x))
         (y (getf plist 'y)))
     (setf (position-rho pos) (sqrt (+ (* x x) (* y y)))
           (position-theta pos) (atan y x))))

(defclass x-y-position (position)
   ((rho :initform 0 :accessor position-rho)
    (theta :initform 0 :accessor position-theta)))

;;;すべての古い x-y-position クラスのインスタンスは自動的に
;;;更新される

;;;新しい表現でも古い表現の見かけと印象を与える

(defmethod position-x ((pos x-y-position))
    (with-slots (rho theta) pos (* rho (cos theta))))

(defmethod (setf position-x) (new-x (pos x-y-position))
    (with-slots (rho theta) pos
      (let ((y (position-y pos)))
        (setq rho (sqrt (+ (* new-x new-x) (* y y)))
              theta (atan y new-x))
        new-x)))

(defmethod position-y ((pos x-y-position))
    (with-slots (rho theta) pos (* rho (sin theta))))

(defmethod (setf position-y) (new-y (pos x-y-position))
    (with-slots (rho theta) pos
      (let ((x (position-x pos)))
        (setq rho (sqrt (+ (* x x) (* new-y new-y)))
```

```
            theta (atan new-y x))
      new-y)))
```

● 参　照：

「11. クラスの再定義」
「10.4 初期化引数の規則」
「10.2 初期化引数の妥当性の宣言」
make-instances-obsolete
shared-initialize

---

with-accessors　　マクロ

● 目　的：

マクロ with-accessors は，スロットがあたかも変数であるかのような静的コンテキストを生成する．マクロ with-accessors はスロットをアクセスするために適当なアクセサを呼ぶ．setf と setq は共にスロットに値を設定するのに使える．

● 構　文：

with-accessors ({*slot-entry*}*) *instance-form* &body *body*　　[マクロ]

slot-entry ::= (*variable-name accessor-name*)

● 値：

結果として返す値は，*body* で示されるフォームを実行して得られるものである．

● 例：

```
(with-accessors ((x position-x)
                 (y position-y))
                 p1
       (setq x y))
```

● 注　意：

(with-accessors (*slot-entry*$_1$ ... *slot-entry*$_n$) *instance form*$_1$ ... *form*$_k$)

は次の式と等価なものに展開される．

```
(let ((in instance))
  (symbol-macrolet (Q1 ... Qn) form1 ... formk))
```

ここで Q_i は

(*variable-name*$_i$ (*accessor-name*$_i$ *in*))

である.

● 参 照:

with-slots
symbol-macrolet

with-added-methods 特殊形式

● 目 的:

特殊形式 with-added-methods は新しい総称関数を生成し, 新しい関数定義を束縛したレキシカルな環境を作り出す. with-added-methods は, レキシカルに見える同じ名前の総称関数をそのメソッドと共にコピーしたものに, メソッド定義で指定されたメソッドを追加して新しい総称関数を生成する. もし同じ名前の総称関数が存在しなかったら, 新しい総称関数を生成する. この総称関数はレキシカルなスコープをもつ.

特殊形式 with-added-methods は局所的に意味のある名前をもった総称関数を定義し, これらの関数束縛のもとで一連のフォームを実行するために用いる.

with-added-methods によって定義された総称関数の名前はレキシカルなスコープをもつ. すなわち, その名前は with-added-methods のボディの中だけの局所的な定義である. with-added-methods のボディの中から関数を参照したとき, その名前の関数が with-added-methods によって局所的な束縛をもっているならば, たとえ同じ名前の関数が大域的な関数として定義されていても, 局所的な関数のほうを参照する. このような総称関数名の束縛によって作られるスコープは with-added-methods のボディの中だけでなく, 局所的な総称関数に属するメソッドのボディの中でも有効である.

● 構 文:

with-added-methods (*function-specifier lambde-list* [↓ *option* | *method-description**]) {*form*}* [特殊形式]

function-specifier ::= {*symbol* | (setf *symbol*)}

option ::= (:argument-precedence-order {*parameter-name*}+) |
(declare {*declaration*}+) |
(:documentation *string*) |
(:method-combination *symbol* {*arg*}*) |

(:generic-function-class *class-name*) |
(:method-class *class-name*)

method-description ::= (:method {*method-qualifier*}* *specialized-lambda-list*
{*declaration* | *documentation*}* {*form*}*)

● 引　数：

引数 *function-specifier*, *option*, *method-qualifier*, そして *specialized-lambda-list* は defgeneric のものと同様である.

おのおののメソッドのボディは暗黙的なブロックに囲まれる. もし *function-specifier* がシンボルならこのブロックは総称関数と同じ名前をもつ. もし *function-specifier* がリスト (setf *symbol*) ならブロックの名前は *symbol* となる.

● 値：

特殊形式 with-added-methods は結果として最後に実行されたフォームの返す値または多値を返す. フォームをもっていなかったら nil を返す.

● 注　意：

もし同じ名前の総称関数がすでにあれば, with-added-methods フォームに指定されたラムダリストはその総称関数上のすべてのメソッドのラムダリストと適合しなければならない. また with-added-methods で定義するすべてのメソッドのラムダリストとも適合しなければならない. もしそうでなければ, エラーを発する.

もし *function-specifier* に既存の総称関数を指定し, 次のいずれかのオプションの値に相違があれば, 総称関数のコピーは新しく指定した値をもつように変更される. これらのオプションは, :argument-precedence-order, declare, :documentation, :generic-function-class, :method-combination である.

もし *function-specifier* に既存の総称関数を指定し, :method-class オプションの値に相違があれば, 総称関数のコピーのもっているオプションの値は変更されるが, コピーされたメソッド自体のクラスは変更されない.

もし同じ名前の関数がすでにあれば, その関数はコピーされ総称関数のデフォルトメソッドとなる. この点は defgeneric と異なるので注意せよ.

もし同じ名前のマクロや特殊形式がすでにあれば, エラーが発せられる.

もし同じ名前の総称関数が存在しないなら, オプションのデフォルト値は defgeneric のデフォルト値と同じである.

● 参 照：

generic-labels
generic-flet
defmethod
defgeneric
ensure-generic-function

with-slots

マクロ

● 目 的：

マクロ with-slots は，スロットがあたかも変数であるかのような静的コンテキストを生成する．このコンテキストの中では，スロットの値を静的に束縛された変数のようにスロット名で参照できる．setf と setq は共にスロットに値を設定するのに使える．

マクロ with-slots は変数名として現われたスロット名を slot-value の呼出しに展開する．

● 構 文：

with-slots ({*slot-entry*}*) *instance-form* &body *body* [マクロ]

slot-entry ::= *slot-name* | (*variable-name slot-name*)

● 値：

結果として返す値は，*body* で示されるフォームを実行して得られるものである．

● 例：

```
(with-slots (x y) position-1
  (sqrt (+ (* x x) (* y y))))

(with-slots ((x1 x) (y1 y)) position-1
  (with-slots ((x2 x) (y2 y)) position-2
    (psetf x1 x2
           y1 y2)))

(with-slots (x y) position
  (setq x (1+ x)
        y (1+ y)))
```

● 注　意：

(with-slots (*slot-entry*$_1$ ... *slot-entry*$_n$) *instance* *form*$_1$ ... *form*$_k$)

は次の式と等価なものに展開される．

```
(let ((in instance))
  (symbol-macrolet (Q1 ... Qn)  form1 ... formk))
```

ここで，*slot-entry*$_i$ がシンボルなら Q_i は

(*slot-entry*$_i$ (slot-value *in* '*slot-entry*$_i$))

である．*slot-entry*$_i$ が (*variable-name*$_i$ *slot-name*$_i$) なら Q_i は

(*variable-name*$_i$ (slot-value *in* '*slot-name*$_i$))

である．

● 参

with-accessors
symbol-macrolet

第II部

CLOSの仕様

Common Lisp Object System

訳　者

（五十音順）

井田昌之　大久保清貴　岡本泰次　川辺治之　小島泰三

西田大治　前田哲司　元吉文男　横尾　真　吉川昌澄