Revised^7 Report on the Algorithmic Language Scheme

[アルゴリズム言語 Scheme 報告書 七訂版]

Alex Shinn, John Cowan, and Arthur A. Gleckler (*Editors*)

Steven Ganz
Aaron W. Hsu
Bradley Lucier
Emmanuel Medernach
Alexey Radul
Jeffrey T. Read
David Rush
Benjamin L. Russel
Olin Shivers
Alaric Snell-Pym
Gerald J. Sussman

Richard Kelsey, William Clinger, and Jonathan Rees
(Editors, Revised^5 Report on the Algorithmic Language Scheme)

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten
(Editors, Revised^6 Report on the Algorithmic Language Scheme)

John McCarthy と *Daniel Weinreb* のみたまにささぐ

2013 年 7 月 6 日

日本語訳 2014 年 1 月 22 日

要約

本報告書はプログラミング言語 Scheme を定義する一つの記述を与える。Scheme は Lisp プログラミング言語 [23] のうちの，静的スコープをもち，かつ真正に末尾再帰的である一方言であり，Guy Lewis Steele Jr. と Gerald Jay Sussman によって発明された。Scheme は並外れて明快で単純な意味論をもち，かつ式をつくる方法の種類がごく少数になるように設計された。命令型，関数型，およびオブジェクト指向型の各スタイルを含む広範囲のプログラミングパラダイムが Scheme によって手軽に表現できる。

序章は本言語と本報告書の歴史を簡潔に述べる。

最初の三つの章は本言語の基本的なアイディアを提示するとともに，本言語を記述するため，および本言語でプログラムを書くために用いられる表記上の規約を記述する。

第 4 章と第 5 章は式，定義，プログラム，およびライブラリの構文と意味を記述する。

第 6 章は Scheme の組込み手続きを記述する。これには本言語のデータ操作と入出力プリミティブのすべてが含まれる。

第 7 章は拡張 BNF で書かれた Scheme の形式的構文を，形式的な表示的意味論とともに定める。本言語の使用の一例が，この形式的な構文と意味論の後に続く。

付録 A は標準ライブラリおよびそれらが書き出す識別子のリストを提供する。

付録 B はオプションだが標準化された実装の機能名のリストを提供する。

本報告書の最後は参考文献一覧とアルファベット順の索引である。

注: R[5]RS および R[6]RS の報告書の編集者は，本報告書の実質的な部分が R[5]RS と R[6]RS から直接コピーされたものであるという認識で，本報告書の執筆者として記載されている。これらの編集者が，個人的あるいは団体的に，本報告書を支持または不支持することを意図するものではない。

目次

はじめに

プログラミング言語の設計は，機能の上に機能を積み重ねることによってではなく，余分な機能が必要であるように思わせている弱点と制限を取り除くことによってなされるべきである。式をつくるための規則が少数しかなくても，その組み合わせ方法に全く制限がなければ，今日使われている主要なプログラミングパラダイムのほとんどをサポートするのに十分なだけの柔軟性を備えた実用的で効率的なプログラミング言語を満足につくり上げられることを，Scheme は実証している。

Scheme は，ラムダ計算におけるようなファーストクラスの手続きを具体化した初めてのプログラミング言語の一つであった。そしてそれによって，動的に型付けされる言語での静的スコープ規則とブロック構造の有用性を証明した。Scheme は，手続きをラムダ式とシンボルから区別し，すべての変数に対し単一の字句的環境を使用し，手続き呼出しの演算子の位置をオペランドの位置と同じように評価する初めての主要な Lisp 方言であった。全面的に手続き呼出しによって繰返しを表現することで，Scheme は，末尾再帰の手続き呼出しが本質的には引数を渡す GOTO であるという事実を強調し，こうしてコヒーレントかつ効率的なプログラミングスタイルを可能にした。Scheme は，ファーストクラスの脱出手続きを採用した初めての広く使われたプログラミング言語であった。この脱出手続きから，すべての既知の逐次的制御構造が合成できる。後の版の Scheme は，正確数と不正確数の概念を導入した。これは Common Lisp の総称的算術演算の拡張である。最近になって Scheme は，保健的マクロ (hygienic macro) をサポートした初めてのプログラミング言語になった。保健的マクロは，ブロック構造をもった言語の構文を矛盾なく信頼性をもって拡張することを可能にする。

背景

Scheme の最初の記述は 1975 年に書かれた [35]。改訂報告書は 1978 年に発表された [31]。改訂報告書は，MIT の実装が革新的なコンパイラ [32] をサポートするために改良されたことに合わせて，その言語の進化を記述した。1981 年と 1982 年には三つの個別のプロジェクトが MIT，Yale および Indiana 大学で講座に Scheme 類を用いるために始まった [27, 24, 14]。Scheme を用いた計算機科学入門の教科書が 1984 年に出版された [1]。

Scheme が広まるにつれ，ローカルな方言が分かれはじめ，学生や研究者が他のサイトで書かれたコードの理解にしばしば困難をおぼえるまでになってきた。そこで，Scheme のより良い，より広く受け入れられる標準に向けて作業するため，Scheme の主要な実装の代表者 15 人が 1984 年 10 月に会合した。その報告書，RRRS [8] は 1985 年の夏に MIT と Indiana 大学で出版された。さらに 1986 年の春，R^3RS [29] として改訂が行われた。1988 年の春にもたらした R^4RS [10] は，1991 年に Scheme プログラミング言語のための IEEE 規格の基礎となった。1998 年には，高レベルの保健的なマクロ，複数の戻り値，および `eval` を含む IEEE 規格へのいくつかの追加が，R5RS として確定された。

2006 年秋，移植性向上の利益のために作られた，多くの新しい改良とより厳しい要件を含む，より大がかりな標準で活動が始まった。その結果となった標準，R^6RS は，2007 年 8 月 [33] に完成し，コア言語と必須の標準ライブラリのセットとしてまとめられた。それに準拠した Scheme のいくつかの新しい実装が作成された。しかし，ほとんどの既存 R^5RS 実装は (基本的にメンテナンスされていないものを除いて) R^6RS を採用しなかったか，選択された一部のみの採用でした。

この結果，Scheme 運営委員会は，2009 年 8 月，独立しているが互換性のある言語— 教育者，研究者，組み込み言語のユーザーに適し，R^5RS 互換性に焦点を当てた “小さな” 言語，および R^6RS の代替となることを目的とした，主流のソフトウェア開発の実用的なニーズに焦点を当てた “大きな” 言語の 2 つに標準を分割することを決定した。今回の報告書は，その努力の “小さな” 言語について説明する。それゆえ，R^6RS の後継として独立してみなすことはできない。

我々は，本報告書を Scheme コミュニティ全体に属するものとしようと思う。そのため我々は，これの全部または一部を無料で複写することを許可する。とりわけ我々は，Scheme の実装者が本報告書をマニュアルその他の文書のための出発点として，必要に応じて改変しつつ，利用することを奨励する。

謝辞

我々は運営委員会のメンバー，William Clinger, Marc Feeley, Chris Hanson, Jonathan Rees, ならびに Olin Shivers の支援と指導に感謝します。

本報告書は，非常に多くのコミュニティの取り組みであり，我々はコメントやフィードバックを提供してくれた，次のすべての人に感謝したいと思います: David Adler, Eli Barzilay, Taylan Ulrich Bayırlı/Kammer, Marco Benelli, Pierpaolo Bernardi, Peter Bex, Per Bothner, John Boyle, Taylor Campbell, Raffael Cavallaro, Ray Dillinger, Biep Durieux, Sztefan Edwards, Helmut Eller, Justin Ethier, Jay Reynolds Freeman, Tony Garnock-Jones, Alan Manuel Gloria, Steve Hafner, Sven Hartrumpf, Brian Harvey, Moritz Heidkamp, Jean-Michel Hufflen, Aubrey Jaffer, Takashi Kato, Shiro Kawai, Richard Kelsey, Oleg Kiselyov, Pjotr Kourzanov, Jonathan Kraut, Daniel Krueger, Christian Stigen Larsen, Noah Lavine, Stephen Leach, Larry D. Lee, Kun Liang, Thomas Lord, Vincent Stewart Manis, Perry Metzger, Michael Montague, Mikael More, Vitaly Magerya, Vincent Manis, Vassil Nikolov, Joseph Wayne Norton, Yuki Okumura, Daichi Oohashi, Jeronimo Pellegrini, Jussi Piitulainen, Alex Queiroz, Jim Rees, Grant Rettke, Andrew Robbins, Devon Schudy, Bakul Shah, Robert Smith, Arthur Smyles, Michael Sperber, John David Stone, Jay Sulzberger, Malcolm Tredinnick, Sam Tobin-Hochstadt, Andre van Tonder, Daniel Villeneuve,

Denis Washington, Alan Watson, Mark H. Weaver, Göran Weinholt, David A. Wheeler, Andy Wingo, James Wise, Jörg F. Wittenberger, Kevin A. Wortman, Sascha Ziemann.

また我々はすべての過去の編集者，ひいては彼らを支援した人々に感謝したいと思います: Hal Abelson, Norman Adams, David Bartley, Alan Bawden, Michael Blair, Gary Brooks, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Robert Findler, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Halstead, Robert Hieb, Paul Hudak, Morry Katz, Eugene Kohlbecker, Chris Lindblad, Jacob Matthews, Mark Meyer, Jim Miller, Don Oxley, Jim Philbin, Kent Pitman, John Ramsdell, Guillermo Rozas, Mike Shaff, Jonathan Shapiro, Guy Steele, Julie Sussman, Perry Wagle, Mitchel Wand, Daniel Weise, Henry Wu, and Ozan Yigit. Scheme 311 バージョン 4 リファレンスマニュアルからテキストを使用する許可してくれた，Carol Fessenden, Daniel Friedman, ならびに Christopher Haynes に感謝します。*TI Scheme* **言語リファレンスマニュアル** [37] からテキストを使用する許可してくれた，Cexas Instruments, Inc. に感謝します。我々は，MIT Scheme [24], T [28], Scheme 84 [15], Common Lisp [34] および Algol 60 [25] に加え，`http://srfi.schemers.org` で入手可能な以下の SRFI のマニュアルの影響を喜んで認めます: 0, 1, 4, 6, 9, 11, 13, 16, 30, 34, 39, 43, 46, 62, および 87。

言語の記述

1. Scheme の概観

1.1. 意味論

本節は Scheme の意味論の概観を与える。詳細な非形式的意味論は 3 章から 6 章で論ずる。典拠として，7.2 節は Scheme の形式的な意味論を定める。

Scheme は静的スコープをもつプログラミング言語である。変数の各使用は，その変数の字句的に見かけの束縛に関連している。

Scheme は動的型付き言語である。型は変数ではなく，値 (オブジェクトとも呼ばれる) に関連付けられている。静的型付き言語は，対照的に，値と同じように変数と式に型を関連付ける。

手続きや継続 (continuation) を含む，Scheme の計算過程で作成されたすべてのオブジェクトは，無期限の寿命 (extent) をもっている。Scheme のオブジェクトは，破壊されることはない。Scheme の実装が (通常は!) 記憶領域を使い果たさない理由は，あるオブジェクトが将来のいかなる計算にも関係ないことを証明できれば，オブジェクトが占める記憶領域を再利用することを許可されているためである。

Scheme の実装は真正に末尾再帰的 (properly tail-recursive) であることが必須である。これは，反復計算が構文的に再帰的な手続きで記述されている場合でも，一定の空間における反復計算の実行を可能にする。このように真正に末尾再帰的な実装では，特別な反復コンストラクトは，糖衣構文としてのみ有用であるように，反復は通常の手続き呼び出しの仕組みを使って表現することができる。3.5 節を見よ。

Scheme の手続きはそれ自体でオブジェクトである。手続きを動的に作成すること，データ構造の中に格納すること，手続きの結果として返すことなどが可能である。

Scheme の一つ際立った特徴は，他のほとんどの言語ではただ舞台裏での演算である継続が，“第一級” の地位も持っているということである。第一級の継続は，非局所的脱出，バックトラッキング，およびコルーチンを含むバラエティに富んだ高度な制御構造を実装するのに有用である。6.10 節を見よ。

Scheme の手続きへの引数は常に値によって渡される。これは，手続きが評価結果を必要とするか否かにかかわらず，手続きが制御を得る前に実引数の式が評価されることを意味する。

Scheme の算術モデルは計算機内部での数の特定の表現方法からできる限り独立を保つように設計されている。Scheme では，あらゆる整数は有理数であり，あらゆる有理数は実数であり，あらゆる実数は複素数である。したがって，整数算術と実数算術の区別は，多くのプログラミング言語にとってはとても重要であるが，Scheme には現れない。そのかわり，数学的理想に相当する正確算術 (exact arithmetic) と近似に基づく不正確算術の区別がある。正確算術は整数に限られない。

1.2. 構文

Scheme は，Lisp の大多数の方言と同じく，プログラムとそれ以外のデータを表すために，省略なくカッコでくくられた前置表記を使用する。Scheme の文法は，データを表すために使われる言語の，部分言語を生成する。この単純さの結果として，一様な表現は，Scheme プログラムやデータを容易に他の Scheme プログラムによって一様に扱うことができる。たとえば `eval` 手続きは，データとして表現された Scheme プログラムを評価する。

`read` 手続きは，字句だけでなく構文も解析してデータを読む。`read` 手続きは入力をプログラムとしてではなくデータ (7.1.2 節) としてパースする。

Scheme の形式的な構文は 7.1 節で記述する。

1.3. 表記と用語

1.3.1. 基本およびオプション機能

本報告書で定義されているすべての識別子は，1 つまたはいくつかのライブラリ (*libraries*) に表れる。base ライブラリ (*base library*) で定義された識別子は，報告書の本文に特別にマークされていない。このライブラリは，Scheme のコア構文と一般的に有用なデータ操作手続きを含んでいる。たとえば，変数 `abs` は数の絶対値を計算する 1 引数の手続きに束縛されており，変数 `+` は和を計算する手続きに束縛されている。それらがエクスポートするすべての標準ライブラリや識別子の完全なリストは，付録 A に記載されている。

Scheme のすべての実装において:

- base ライブラリと，そこからエクスポートされたすべての識別子を提供しなければならない。
- 本報告書に記載された他のライブラリを提供または省略してもよいが，各ライブラリは，追加の識別子をエクスポートすることなくその全体が提供される，または完全に省略される，のいずれかでなければならない。
- 本報告書に記載されていない他のライブラリを提供してもよい。
- また，本報告書中の任意の識別子の関数を拡張してもよく，提供される拡張機能はここで報告された言語と矛盾しない。
- 字句構文が本報告書に記載された字句構文と競合しないような動作モードを提供することにより，移植可能なコードをサポートしなければならない。

1.3.2. エラー状態と未規定の振舞い

エラー状態について言うとき，本報告書は “エラーが通知される” という表現を使って，実装がそのエラーを検出し報告しなければならないことを示す。エラーは， 6.11 節で記述された手続き `raise` のように継続不能な例外を発生させることで通知される。発生させたオブジェクトは実装依存であり，以前に同じ目的のために使われたオブジェクトとは別である必要はない。本報告書に記載された状況でのエラー通知に加えて，プログラマは自身のエラー通知し，通知されたエラーを処理することができる。

語句 “ **述語**を満たすエラーが通知される” とは，エラーが上記のように通知されることを意味する。さらに，通知されるオブジェクトが指定した (`file-error?` または `read-error?` のような) 述語に渡される場合，述語は `#t` を返す。

もしあるエラーの議論にそのような言い回しが現れないならば，実装がそのエラーを検出または報告することは，奨励されているが，必須ではない。そのような状況は時々，常にではないが，語句 “エラー” と呼ばれる。このような状況では，実装はエラーを通知してもよいし，しなくてもよい。もしエラーを通知する場合は，通知されるオブジェクトは述語 `error-object?`，`file-error?` または `read-error?` を満たしてもよいし，満たさなくてもよい。代わりに，実装は移植性のない拡張機能を提供してもよい。

たとえば，ある手続きにその手続きで処理することが明示的に規定されていない型の引数を渡すことは，たとえそのような定義域エラーが本報告書でほとんど言及されていなくても，エラーである。実装は，エラーを通知するか，そのような引数を含めるように手続きの定義域を拡張するか，壊滅的に失敗してもよい。

本報告書は “実装制限の違反を報告してもよい” という表現を使って，実装の課すなんらかの制限のために正当なプログラムの実行を続行できない，と報告することが実装に許可されている状況を示す。実装制限は望ましくないが，実装が実装制限の違反を報告することは奨励されている。

たとえば実装は，あるプログラムを走らせるだけの十分な記憶領域がないとき，あるいは，算術演算が実装において大きすぎて表現できないな正確数を生成するとき，実装制限の違反を報告してもよい。

もしある式の値が “未規定” (unspecified) であると述べられているならば，その式は，エラーが通知されることなく，なんらかのオブジェクトへと評価されなければならない。しかし，その値は実装に依存する。本報告書は，どんな値が返されるかを明示的には述べない。

最後に，単語やフレーズ “しなければならない (must),” “してはならない (must not),” “するものとする (shall),” “しないものとする (shall not),” “すべきである (should),” “すべきではない (should not),” “してもよい (may),” “必須である (required),” “推奨される (recommended),” ならびに “任意である (optional)” は，本報告書に大文字で書かれてはいないが，RFC 2119 [3] で説明されているように解釈されるべきである。それらは実装者あるいは実装の振舞いに関してのみ使用され，プログラマやプログラムの振舞いに関してではない。

1.3.3. エントリの書式

エントリごとにまとめられている。各エントリは一つの言語機能，または関連する複数の機能からなる一つのグループを記述する。ただし，ここで機能とは構文コンストラクトまたは手続きのことである。エントリは下記の書式の 1 行以上の見出しで始まる。base ライブラリ中の識別子ならば，

template *category*

そうでなければ，

template *name* library *category*

ここで *name* は付録 A で定義されているライブラリの短縮名である。

もし *category* が “構文” ならば，そのエントリは式型を記述し，template はその式型の構文を与える。式の構成要素は構文変数によって示される。構文変数は <式>，<変数> のように山形カッコを使って書かれる。構文変数はプログラムテキストのそれぞれの部分を表すことが意図されている。たとえば <式> は，構文的に妥当な一つの式である任意の文字の列を表す。表記

<なにか $_1$> ...

は <なにか> の 0 個以上の出現を示し，

<なにか $_1$> <なにか $_2$> ...

は <なにか> の 1 個以上の出現を示す。

もし *category* が “補助構文” ならば，そのエントリは特定の囲まれた式の一部としてのみ起こる構文束縛を記述する。独立した構文構造や変数としてのいかなる使用もエラーである。

もし *category* が “手続き” ならば，そのエントリは手続きを記述し，見出し行はその手続きの呼出しを表すテンプレートを与える。テンプレートの中の引数名は *italic* 体で書かれる。したがって見出し行

`(vector-ref` *vector k*`)` 手続き

は `vector-ref` に束縛された手続きが二つの引数，ベクタ *vector* と正確非負整数 *k* (下記参照) をとることを示す。見出し行

`(make-vector` *k*`)` 手続き
`(make-vector` *k fill*`)` 手続き

は `make-vector` 手続きが一つまたは二つの引数をとるように定義されなければならないことを示す。

ある演算に対し処理することが規定されていない引数を与えることはエラーである。簡潔のため我々は，もし引数名が 3.2 節に挙げられた型の名前でもあるならばその引数はその引数が名前付けられた型のものでない場合はエラーである，という規約に従う。たとえば上記の `vector-ref` の見出し行は，`vector-ref` の第 1 引数がベクタでなければならないことを命じている。以下の命名規約もまた型の制約を意味

する。

alist	連想リスト (ペアのリスト)
boolean	ブーリアン (`#t` or `#f`)
byte	正確整数 $0 \leq byte < 256$
bytevector	バイトベクタ
char	文字
end	正確非負整数
k, k_1, ... k_j, ...	正確非負整数
letter	英字
list, $list_1$, ... $list_j$, ...	リスト (6.4 節参照)
n, n_1, ... n_j, ...	整数
obj	任意のオブジェクト
pair	ペア
port	ポート
proc	手続き
q, q_1, ... q_j, ...	有理数
start	正確非負整数
string	文字列
symbol	シンボル
thunk	引数なしの手続き
vector	ベクタ
x, x_1, ... x_j, ...	実数
y, y_1, ... y_j, ...	実数
z, z_1, ... z_j, ...	複素数

名前 *start* および *end* は，文字列，ベクタ，およびバイトベクタの添字として使われる。これらの使用は，次のことを意味する:

- *start* が *end* より大きければエラーである。
- *end* が文字列，ベクタ，バイトベクタの長さよりも大きければエラーである。
- *start* が省略された場合，ゼロとみなす。
- *end* が省略された場合，文字列，ベクタ，バイトベクタの長さとみなす。
- 添字 *start* は常に含まれ，添字 *end* は常に含まない。例として，文字列を考える。もしも *start* と *end* が同じならば空の部分文字列のことになり，もしも *start* がゼロであって *end* が *string* の長さならば文字列全体のことになる。

1.3.4. 評価の例

プログラム例で使われる記号 "$\Longrightarrow$" は "へと評価される" と読む。たとえば，

```
(* 5 8)                    ⟹  40
```

は式 `(* 5 8)` がオブジェクト `40` へと評価されることを意味する。つまり，より正確には，文字の列 "`(* 5 8)`" によって与えられる式が，文字の列 "`40`" によって外部的に表現され得るあるオブジェクトへと，初期環境の中で評価されることを意味する。オブジェクトの外部表現の議論については 3.3 節を見よ。

1.3.5. 命名規約

規約により，`?` は常にブーリアン値を返す手続きの名前の最後の文字である。このような手続きは述語 (predicate) と呼ばれる。述語は，引数の型が間違って渡されたときに例外を起こしてもよいことを除いて，一般的に副作用がないと理解される。

同様に，`!` は，以前に割り当てられた場所の中へ値を格納する手続き (3.4 節参照) の名前の最後の文字である。このような手続きは書換え手続き (*mutation procedure*) と呼ばれる。書換え手続きが返す値は未規定である。

規約により，ある型のオブジェクトをとって別の型の相当するオブジェクトを返す手続きの名前の途中には "`->`" が現れる。たとえば `list->vector` は，一つのリストをとって，そのリストと同じ要素からなるベクタを返す。

コマンド (*command*) は，その継続に有用な値を返さない手続きである。

サンク (*thunk*) は，引数を受け取らない手続きである。

2. 字句規約

本節は Scheme プログラムを書くときに使われる字句的な規約のいくつかを非形式的に説明する。Scheme の形式的な構文については 7.1 節を見よ。

2.1. 識別子

識別子は，文字，数字，および有効な数である接頭辞を持っていないことを条件とする "拡張識別子文字" の任意のシーケンスである。しかし，リスト構文で使用される `.` トークン (単一ピリオド) は識別子ではない。

Scheme のすべての実装は以下の拡張識別子文字をサポートしなければならない:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

代わりに，識別子は垂直線で囲まれた 0 個以上の文字列で表すことができ，文字列リテラルに似ている。空白文字を含むが，バックスラッシュと垂直線の文字を除く任意の文字は，そのような識別子にそのまま表示することができる。また，文字は<インライン 16 進エスケープ>あるいは文字列内で使用可能な同一エスケープのいずれかを使用して指定することができる。

たとえば，識別子 `|H\x65;llo|` は `Hello` と同一の識別子であり，適切な Unicode 文字をサポートする実装では，識別子 `|\x3BB;|` は識別子 λ と同一である。しかも，`|\t\t|` と `|\x9;\x9;|` は同一である。`||` は他の識別子とは異なる有効な識別子であることに注意せよ。

ここでは識別子の例をいくつか示す。

```
...                     +
+soup+                  <=?
->string                a34kTMNs
```

```
lambda                          list->vector
q                               V17a
|two words|                     |two\x20;words|
the-word-recursion-has-many-meanings
```

識別子の形式的な構文については 7.1.1 節を見よ。

識別子には Scheme プログラムの中で二つの用途がある。

- どの識別子も変数として，または構文キーワードとして使える (3.1 節および 4.3 節参照)。
- 識別子がリテラルとして，またはリテラルの中に現れたとき (4.1.2 節参照)，それは**シンボル**を表すために使われている (6.5 節参照)。

報告書の以前の改訂 [20] とは対照的に，構文は，識別子およびその名前を使用して指定された文字で，大文字と小文字を区別する。しかし，数字あるいは識別子，文字，文字列の構文で使用される<インライン 16 進エスケープ>では大文字と小文字を区別しない。本報告書で定義された識別子はいずれも，英語の規約の結果として文の最初の単語を大文字で始めるように表示された場合でも，大文字が含まれていない。

次のディレクティブは，大文字小文字変換に明示的な制御を与える。

```
#!fold-case
#!no-fold-case
```

これらのディレクティブは，コメントが許可されている任意の場所に表示できる (2.2 参照) が，区切り文字の後に続かなければならない。それらは同じポートから後続のデータの読み出しに影響を与えることを除いては，コメントとして扱われる。`#!fold-case` ディレクティブは，後続の識別子や文字の名前が `string-foldcase` (6.7 節参照) によってであるかのように大文字小文字変換を起こす。これは，文字リテラルには影響しない。`#!no-fold-case` ディレクティブは，デフォルトの大文字小文字変換を行わない振舞いへの戻りを起こす。

2.2. 空白と注釈

空白文字 (*whitespace* character) は，スペース，タブおよび改行を含む。(実装は改ページなどの付加的な空白文字を定めてもよい。) 空白は，可読性を向上するために使われ，そしてトークンどうしを分離するために欠かせないものとして使われるが，それ以外の意味はない。ここでトークンとは，識別子または数などの，ある分割できない字句単位である。空白はどの二つのトークンのあいだに出現してもよいが，一つのトークンの途中に出現してはならない。文字列の内部または縦線で区切られたシンボルの内部に出現する空白は重要である。

字句構文は、いくつかのコメント形式が含まれている。コメントは正確に空白のように扱われる。

セミコロン (`;`) は注釈 (comment) 行の開始を示す。 注釈はそのセミコロンが現れた行の行末まで続く。

コメントを示すもうひとつの方法は，<データ>(7.1.2 節参照) に`#;` と省略可能な <空白> を接頭辞として付けることである。コメントは，コメント接頭辞 `#;`，スペース，および <データ> と共に構成されている。この表記は，コードのセクションを “コメントアウト” するのに有用である。

ブロックコメントは，適切なネスト`#|` と `|#` のペアで示される。

```
#|
   FACT 手続き
   非負整数の階乗 (factorial) を計算する
|#
(define fact
  (lambda (n)
    (if (= n 0)
        #;(= n 1)
        1         ; 再帰の底: 1 を返す
        (* n (fact (- n 1))))))
```

2.3. その他の表記

数に対して使われる表記の記述については 6.2 節を見よ。

`.` `+` `-` これらは数で使われ，そしてまた識別子のどこにでも出現する可能性がある。1 個の孤立した正符号または負符号も識別子である。1 個の孤立した (数や識別子の中に出現するのではない) ピリオドは，ペアの表記 (6.4 節) で，および仮パラメタ並びの残余パラメタ (4.1.4 節) を示すために使われる。2 つ以上のピリオドからなる列は識別子**である**ことに注意せよ。

`(` `)` 丸カッコはグループ化とリストの表記のために使われる (6.4 節)。

`'` アポストロフィー (単一引用符) はリテラルデータを示すために使われる (4.1.2 節)。

`` ` `` グレイヴアクセント (逆引用符) は部分的定数データを示すために使われる (4.2.8 節)。

`,` `,@` 文字「コンマ」と列「コンマ」「アットマーク」は準引用と組み合わせて使われる (4.2.8 節)。

`"` 引用符は文字列を区切るために使われる (6.7 節)。

`\` 逆スラッシュは文字定数を表す構文で使われ (6.6 節)，そして文字列定数 (6.7 節) および識別子 (section 7.1.1) の中のエスケープ文字として使われる。

`[` `]` `{` `}` 左右の角カッコと波カッコ (ブレース) は，言語にとって将来あり得る拡張のために予約されている。

`#` 番号記号は下記のようにその直後に続く文字によって決まる様々な目的のために使われる。

`#t #f` これらは代替 `#true` および `#false` とともに，ブーリアン定数である (6.3 節)。

`#\` これは文字定数の先頭となる (6.6 節)。

`#(` これはベクタ定数の先頭となる (6.8 節)。ベクタ定数は `)` を末尾とする。

`#u8(` これはバイトベクタ定数 (6.9 節) を導入する。バイトベクタ定数は `)` で終わる。

`#e #i #b #o #d #x` これらは数の表記に使われる (6.2.5 節)。

`#<n>= #<n>#` これらはラベリングおよび他のリテラルデータを参照するために使われている (2.4 節)。

2.4. データラベル

`#<n>=<データ>` 字句構文
`#<n>#` 字句構文

字句構文`#<n>=<データ>` は `<データ>` と同じ読み込みだけでなく，`<データ>` が `<n>`でラベリングされていることになる。`<n>` が数字の列でない場合はエラーである。

字句構文 `#<n>#` は `#<n>=` でラベル付けされたあるオブジェクトへの参照として機能する。結果は， `#<n>=` と同じオブジェクトである (6.1 節参照)。

合わせて，これらの構文は，共有または環状の部分構造を有する構造の表記を可能にする。

```
(let ((x (list 'a 'b 'c)))
  (set-cdr! (cddr x) x)
  x)                          ⟹ #0=(a b c . #0#)
```

データラベルの範囲は，そのラベルの右側に表われる最も外側のデータの一部である。その結果，参照 `#<n>#` はラベル `#<n>=` の後にのみ現れることができる。それは前方参照を試行することはエラーである。加えて，参照が (`#<n>= #<n>#` のように)，ラベル付けされたオブジェクト自身として現れる場合は `#<n>=` でラベル付けされたオブジェクトがこの場合にも定義されていないため，エラーである。

リテラルを除いて，循環参照を含むことは `<プログラム>` または `<ライブラリ>` のエラーである。特に，それらを含むことは，準クォート (4.2.8 節) のエラーである。

```
#1=(begin (display #\x) #1#)
                              ⟹ エラー
```

3. 基本概念

3.1. 変数，構文キーワード，および領域

識別子は，構文の型または値を格納できる場所のどちらかに名前をつけることができる。構文の型の名前になっている識別子は，**構文キーワード** (*syntactic keyword*) と呼ばれ，場所の名前になっている識別子は，**変数** (*variable*) と呼ばれ，その場所に**束縛**されていると言われる。プログラムの中のある地点で有効なすべての可視な束縛からなる集合は，その地点で有効な**環境** (*environment*) として知られる。変数が束縛されている場所に格納されている値は，その変数の値と呼ばれる。用語の誤用によって，変数が値の名前になるとか，値に束縛されるとか言われることがある。これは必ずしも正しくはないが，こうした用法から混乱がもたらされることは滅多にない。

ある種の式型が，新しい種類の構文を作成して構文キーワードをその新しい構文に束縛するために使われる一方，別の式型は新しい場所を作成して変数をその場所に束縛する。これらの式型を**束縛コンストラクト** (*binding construct*) と呼ぶ。構文キーワードを束縛するコンストラクトは 4.3 節で挙げる。最も基本的な変数束縛コンストラクトは `lambda` 式である。なぜなら他の変数束縛コンストラクトはすべて `lambda` 式によって説明できるからである。他の変数束縛コンストラクトは `let`, `let*`, `letrec`, `letrec*`, `let-values`, `let*-values`, `do` の各式である (4.1.4 節, 4.2.2 節, 4.2.4 節参照)。

Scheme はブロック構造をもつ言語である。プログラムの中で識別子が束縛される箇所にはそれぞれ，それに対応して，その束縛が可視であるようなプログラムテキストの**領域** (*region*) がある。領域は，束縛を設けたまさにその束縛コンストラクトによって決定される。たとえば，もし束縛が `lambda` 式によって設けられたならば，その束縛の領域はその `lambda` 式全体である。識別子への言及はそれぞれ，その識別子の使用を囲む領域のうち最内のものを設けた束縛を参照する。もしも使用を囲むような領域をとる束縛がその識別子にないならば，その使用は変数に対する大域環境での束縛を，もしあれば，参照する (4 章および 6 章)。もしもその識別子に対する束縛が全くないならば，その識別子は**未束縛** (*unbound*) であると言われる。

3.2. 型の分離性

どのオブジェクトも下記の述語を二つ以上満たすことはない。

`boolean?`	`bytevector?`
`char?`	`eof-object?`
`null?`	`number?`
`pair?`	`port?`
`procedure?`	`string?`
`symbol?`	`vector?`

かつすべての述語が `define-record-type` によって作られる。

これらの述語が **ブーリアン，バイトベクタ，文字**，空リストオブジェクト，*EOF* **オブジェクト，数，ペア，ポート，手続き，文字列，シンボル，ベクタ**，およびすべてのレコード型が型を定義する。

単独のブーリアン型というものはあるが，あらゆる Scheme 値を条件テストの目的のためにブーリアン値として使うこ

とができる。6.3 節で説明するように，`#f` を除くすべての値はこのようなテストで真と見なされる。本報告書は “真” (true) という言葉を `#f` を除くあらゆる Scheme 値のことを言うために使い，“偽” (false) という言葉を `#f` のことを言うために使う。

3.3. 外部表現

Scheme (および Lisp) の一つの重要な概念は，文字の列としてのオブジェクトの**外部表現** (*external representation*) という概念である。たとえば，整数 28 の外部表現は文字の列 “28” であり，整数 8 と 13 からなるリストの外部表現は文字の列 “`(8 13)`” である。

オブジェクトの外部表現は必ずしも一意的ではない。整数 28 には “`#e28.000`” や “`#x1c`” という表現もあり，上の段落のリストには “`( 08 13 )`” や “`(8 . (13 . ()))`” という表現もある (6.4 節参照)。

多くのオブジェクトには標準的な外部表現があるが，手続きなど，標準的な表現がないオブジェクトもある (ただし，個々の実装がそれらに対する表現を定義してもよい)。

対応するオブジェクトを得るために，外部表現をプログラムの中に記述することができる (4.1.2 節， `quote` 参照)。

外部表現は入出力のために使うこともできる。手続き `read` (6.13.2 節) は外部表現をパースし，手続き `write` (6.13.3 節) は外部表現を生成する。これらは共同してエレガントで強力な入出力の手段を与えている。

文字の列 “`(+ 2 6)`” は，整数 8 へと評価される式で**ある**が，整数 8 の外部表現では**ない**ことに注意せよ。正しくは，それはシンボル `+` と整数 2 と 6 からなる 3 要素のリストの外部表現である。Scheme の構文の性質として，1 個の式であるような文字の列はなんであれ，1 個のなんらかのオブジェクトの外部表現でもある。これは混乱をもたらしかねない。なぜなら，与えられた文字の列がデータを表そうとしているのか，それともプログラムを表そうとしているのか，文脈から必ずしも明らかではないからである。しかし，これは強力さの源でもある。なぜなら，これはインタプリタやコンパイラなど，プログラムをデータとして (あるいは逆にデータをプログラムとして) 扱うプログラムを書くことを容易にするからである。

様々な種類のオブジェクトの外部表現の構文については，そのオブジェクトを操作するためのプリミティブの，6 章の該当する節における記述の中で述べる。

3.4. 記憶モデル

変数と，ペアや文字列やベクタやバイトベクタなどのオブジェクトは，暗黙のうちに場所または場所の列を表している。たとえば，文字列はその文字列の中にある文字数と同じだけの場所を表している。新しい値はこれらの場所の一つに `string-set!` 手続きを使って格納できるが，文字列はあいかわらず同じ場所を指し続ける。

変数参照または `car`，`vector-ref`，`string-ref` などの手続きによって，ある場所からオブジェクトを取り出したとき，そのオブジェクトは，取出しの前にその場所に最後に格納されたオブジェクトと，`eqv?` (6.1 節) の意味で等価である。

場所にはそれぞれ，その場所が使用中かどうかを示すマークが付けられている。使用中でない場所を変数またはオブジェクトが参照することは決してない。

本報告書が，記憶領域を変数またはオブジェクトのために新たに割り当てる，と述べるとき常にそれが意味することは，使用中でない場所からなる集合から適切な個数の場所を選び出し，それらの場所に現在使用中であることを示すマークを付けてから，それらの場所を指すように変数またはオブジェクトを整える，ということである。にもかかわらず，空リストが新たに割り当てることができないことがるわかるのは，それが固有のオブジェクトだからである。場所を含んでいない空文字列，空のベクタ，および空のバイトベクタは，新たに割り当てをしてもしなくてもよいということもわかる。

場所を表すすべてのオブジェクトは，書換え可能 (mutable) か書換え不可能 (immutable) のいずれかである。リテラル定数，`symbol->string` で返される文字列，そしておそらく `scheme-report-environment` で返される環境は書換え不可能オブジェクトである。本報告書に挙げられている他の手続きが作成するオブジェクトはすべて書換え可能である。書換え不可能オブジェクトが表している場所に新しい値を格納しようとすることはエラーである。

これらの場所は概念的に理解すべきであり，物理的ではない。したがって，これらは必ずしもメモリアドレスに対応しておらず，もしそうであったとしても，メモリアドレスが一定でない可能性がある。

根拠: 多くのシステムでは，定数 (すなわちリテラル式の値) については，読み出し専用メモリ内に常駐することが望ましい。書換え可能オブジェクトと書換え不可能オブジェクトを区別するために他の仕組みを必要としないのと同時に，定数の変更をエラーにすることは，この実装戦略を可能にする。

3.5. 真正な末尾再帰

Scheme の実装は**真正に末尾再帰的**であることが必須である。下記で定義する特定の構文的文脈に出現している手続き呼出しは**末尾呼出し**である。もし無限個のアクティブな末尾呼出しを Scheme の実装がサポートしている場合，その実装は真正に末尾再帰的である。呼出しが**アクティブ**であるとは，呼び出した手続きから戻る可能性がまだあるということである。この，戻る可能性のある呼出しには，現在の継続によるものと，`call-with-current-continuation` によって以前に取り込まれ，後になって呼び起こされた継続によるものがあることに注意せよ。取り込まれた継続がなかった場合，呼出しは高々 1 回戻ることができるだけであり，アクティブな呼出しとはまだ戻っていない呼出しのことである，となっただろう。真正な末尾再帰の形式的定義は [6] に見られる。

根拠:

直観的に，アクティブな末尾呼出しに対して空間は不必要である。なぜなら末尾呼出しで使われる継続は，その末尾呼出しを収める手続きに渡されている継続と，同一の意味論をもつからである。たとえ非真正な実装が末尾呼出しで新しい継続を使ったとしても，この新しい継続への戻りは，ただちに，もともと手続きに渡されていた継続への戻りへと続くことになる。真正に末尾再帰的な実装は，直接その継続へと戻るのである。

真正な末尾再帰は，Steele と Sussman のオリジナル版 Scheme にあった中心的なアイディアの一つだった。彼らの最初の Scheme インタプリタは関数とアクタ (actor) の両方を実装した。制御フローはアクタを使って表現された。アクタは，結果を呼出し元に返すかわりに別のアクタへ渡すという点で，関数と異なっていた。本節の用語で言えば，各アクタは別のアクタへの末尾呼出しで終わったわけである。

Steele と Sussman はその後，彼らのインタプリタの中のアクタを扱うコードが関数のそれと同一であり，したがって言語の中に両方を含める必要がないことに気づいた。

末尾呼出し (*tail call*) とは，**末尾文脈** (*tail context*) に出現している手続き呼出しである。末尾文脈は帰納的に定義される。末尾文脈は常に個々のラムダ式に関して決定されることに注意せよ。

- 下記に <末尾式> として示されているラムダ式本体内の最後の式は末尾文脈に出現している。同じことが `case-lambda` 式のすべての本体についても真である。

```
(lambda <仮引数部>
  <定義>* <式>* <末尾式>)

(case-lambda (<仮引数部> <末尾本体>)*)
```

- もし以下の式の一つが末尾文脈にあるならば，<末尾式> として示されている部分式は末尾文脈にある。これらは7章で与える文法の規則から <本体> のいくつかの出現を <末尾本体> で，<式> のいくつかの出現を <末尾式> で，<シーケンス> のいくつかの出現を <末尾シーケンス> で，それぞれ置き換えることによって導出された。それらのうち末尾文脈をもつ規則だけが，ここに示されている。

```
(if <式> <末尾式> <末尾式>)
(if <式> <末尾式>)

(cond <cond 節>+)
(cond <cond 節>* (else <末尾列>))

(case <式>
  <case 節>+)
(case <式>
  <case 節>*
  (else <末尾列>))

(and <式>* <末尾式>)
(or <式>* <末尾式>)

(when <テスト> <末尾列>)
(unless <テスト> <末尾列>)

(let (<束縛仕様>*) <末尾本体>)
(let <変数> (<束縛仕様>*) <末尾本体>)
(let* (<束縛仕様>*) <末尾本体>)
(letrec (<束縛仕様>*) <末尾本体>)
(letrec* (<束縛仕様>*) <末尾本体>)
(let-values (<mv 束縛仕様>*) <末尾本体>)
(let*-values (<mv 束縛仕様>*) <末尾本体>)

(let-syntax (<構文仕様>*) <末尾本体>)
(letrec-syntax (<構文仕様>*) <末尾本体>)

(begin <末尾列>)

(do (<繰返し仕様>*)
    (<テスト> <末尾列>)
  <式>*)
```

ただし

```
<cond 節> ⟶ (<テスト> <末尾列>)
<case 節> ⟶ ((<データ>*) <末尾列>)

<末尾本体> ⟶ <定義>* <末尾列>
<末尾列> ⟶ <式>* <末尾式>
```

- もし `cond` または `case` 式が末尾文脈にあって，(<式 $_1$> `=>` <式 $_2$>) という形式の節をもつならば，<式 $_2$> の評価から結果として生ずる (暗黙の) 手続き呼出しは末尾文脈にある。<式 $_2$> それ自身は末尾文脈にはない。

本報告書で定義された特定の手続きも末尾呼出しを行うことが必須である。`apply` および `call-with-current-continuation` へ渡される第1引数と，`call-with-values` へ渡される第2引数は，末尾呼出しによって呼び出されなければならない。同様に，`eval` はその引数を，あたかもそれが `eval` 手続きの中の末尾位置にあるかのように，評価しなければならない。

以下の例で末尾呼出しは `f` の呼出しだけである。`g` と `h` の呼出しは末尾呼出しではない。`x` の参照は末尾文脈にあるが，呼出しではないから末尾呼出しではない。

```
(lambda ()
  (if (g)
      (let ((x (h)))
        x)
      (and (g) (f))))
```

注: 実装は，上記の `h` の呼出しなどの，あたかも末尾呼出しであるかのように評価できるいくつかの非末尾呼出しを認識してもよい。上記の例では，`let` 式を `h` の末尾呼出しとしてコンパイルできるだろう。(`h` が予期しない個数の値を返す可能性は無視してよ

い。なぜなら，その場合 `let` の効果は明示的に未規定であり実装依存だからである。)

4. 式

式型は，**原始**式型 (*primitive* expression type) と，**派生**式型 (*derived* expression type) に大別される。原始式型は変数と手続き呼出しを含む。派生式型は意味論的に原始的ではなく，マクロとして定義できる。派生式のいくつかの適切な構文定義は，7.3 節に記載されている。

手続き `force`，`promise?`，`make-promise`，および `make-parameter` はまた `delay`，`delay-force`，および `parameterize` 式型と密接に関連しているので本章に記載されている。

4.1. 原始式型

4.1.1. 変数参照

<変数> 構文

一つの変数 (3.1 節) からなる式は，変数参照である。変数参照の値は，その変数が束縛されている場所に格納されている値である。未束縛の変数を参照することはエラーである。

```
(define x 28)
x                              ⟹  28
```

4.1.2. リテラル式

(quote <データ>) 構文
'<データ> 構文
<定数> 構文

(quote <データ>) は <データ> へと評価される。<データ> は 任意の Scheme オブジェクトの外部表現がなりうる (3.3 節参照)。この表記はリテラル定数を Scheme コードの中に含めるために使われる。

```
(quote a)                      ⟹  a
(quote #(a b c))               ⟹  #(a b c)
(quote (+ 1 2))                ⟹  (+ 1 2)
```

(quote <データ>) は '<データ> と略記できる。この二つの表記はすべての点で等価である。

```
'a                             ⟹  a
'#(a b c)                      ⟹  #(a b c)
'()                            ⟹  ()
'(+ 1 2)                       ⟹  (+ 1 2)
'(quote a)                     ⟹  (quote a)
''a                            ⟹  (quote a)
```

数値定数，文字列定数，文字定数，ベクタ定数，バイトベクタ定数，ブーリアン定数はそれ自身へと評価される。それらはクォート (quote) しなくてもよい。

```
'145932                        ⟹  145932
145932                         ⟹  145932
'"abc"                         ⟹  "abc"
"abc"                          ⟹  "abc"
'#                             ⟹  #
#                              ⟹  #
'#(a 10)                       ⟹  #(a 10)
#(a 10)                        ⟹  #(a 10)
'#u8(64 65)                    ⟹  #u8(64 65)
#u8(64 65)                     ⟹  #u8(64 65)
'#t                            ⟹  #t
#t                             ⟹  #t
```

3.4 節で注意したように，定数 (つまりリテラル式の値) を，`set-car!` や `string-set!` のような書換え手続きを使って書き換えようとすることはエラーである。

4.1.3. 手続き呼出し

(<演算子> <オペランド $_1$> ...) 構文

手続き呼出しは，1 つの呼び出される手続きとしての式に続いて，その手続きに渡される引数としての各式を，丸カッコでくくることによって書かれる。演算子とオペランドの各式が (未規定の順序で) 評価され，そしてその結果として得られた手続きに，結果として得られた引数が渡される。

```
(+ 3 4)                        ⟹  7
((if #f + *) 3 4)              ⟹  12
```

本書中の手続きは，標準ライブラリによりエクスポートされた変数の値として利用可能である。たとえば，上の例の加算と乗算の手続きは base ライブラリ中の変数 `+` と `*` の値である。新しい手続きは `lambda` 式を評価することによって作成される (4.1.4 節参照)。

手続き呼出しは任意個数の値を返すことができる (6.10 節 `values` 参照)。本報告書で定義されているほとんどの手続きは 1 個の値を返すかまたは，`apply` のような手続きについていえば，手続きの引数のうちの一つへの呼出しが返した (複数個の) 値をそのまま受け取って次に渡す。例外は個々の説明に記載されている。

注: ほかの Lisp の方言とは対照的に，評価の順序は未規定であり，かつ演算子式とオペランド式は常に同じ評価規則で評価される。

注: たしかにその他の点では評価の順序は未規定だが，ただし，演算子式とオペランド式を同時並行に評価するときは，その効果はなんらかの逐次的な評価順序と一致しなくてはならない。評価の順序はそれぞれの手続き呼出しごとに別々に選択されてもよい。

注: 多くの Lisp の方言では，空のリスト `()` はそれ自身に評価される正当な式である。Scheme では，エラーである。

4.1.4. 手続き

(lambda <仮引数部> <本体>) 構文

構文: <仮引数部> は後述のような仮引数並びである。<本体> は 0 個以上の定義に続く 1 個以上の式からなる列である。

意味: lambda 式は手続きへと評価される。lambda 式が評価される時に有効だった環境は，その手続きの一部として記憶される。その手続きを後からなんらかの実引数とともに呼び出すと，lambda 式が評価された時の環境が，仮引数並びの中の各変数を新しい場所へ束縛することによって拡張され，そして対応する実引数値がそれらの場所へ格納される。(新鮮な (*fresh*) 場所は，すべての既存の場所とは異なるものである。) 次に，ラムダ式の本体の各式 (それが定義を含んでいる場合，letrec* 形式を表す — 4.2.2 節参照) が，その拡張された環境の中で逐次的に評価される。本体の最後の式の結果が，その手続き呼出しの 結果として返される。

```
(lambda (x) (+ x x))          ⟹  一つの手続き
((lambda (x) (+ x x)) 4)      ⟹  8

(define reverse-subtract
  (lambda (x y) (- y x)))
(reverse-subtract 7 10)       ⟹  3

(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                      ⟹  10
```

<仮引数部> は次の形式の一つをとる。

- (<変数 $_1$> ...): 手続きは固定個数の引数をとる。手続きが呼び出される時には，各引数がそれぞれ対応する変数に束縛された新しい場所に格納される。
- <変数>: 手続きは任意個数の引数をとる。手続きが呼び出される時には，実引数の列が，一つの新たに割り当てられたリストへと変換され，そしてそのリストが <変数> に束縛された新しい場所に格納される。
- (<変数 $_1$> ... <変数 $_n$> . <変数 $_{n+1}$>): もしスペースで区切られたピリオドが最後の変数の前にあるならば，手続きは n 個以上の引数をとる。ここで n はピリオドの前にある仮引数の個数である (1 以上でなければエラーである)。まず最後の変数以外の仮引数に対して実引数がそれぞれ対応づけられる。そのあと残余の実引数からなる一つの新たに割り当てられたリストが，最後の変数の束縛に格納される値になる。

一つの <変数> が <仮引数部> に複数回現れることはエラーである。

```
((lambda x x) 3 4 5 6)        ⟹  (3 4 5 6)
((lambda (x y . z) z)
 3 4 5 6)                     ⟹  (5 6)
```

手続きに対して eqv? と eq? が機能するようにするため，lambda 式の評価結果として作成される手続きはそれぞれ (概念的には) 記憶場所をタグとしてタグ付けされる (6.1 節参照)。

4.1.5. 条件式

(if <テスト> <帰結部> <代替部>) 構文
(if <テスト> <帰結部>) 構文

構文: <テスト>，<帰結部>，<代替部> は式である。

意味: if 式は次のように評価される。まず <テスト> が評価される。もしそれが真値 (6.3 節参照) をもたらすならば，<帰結部> が評価されてその値が返される。そうでなければ <代替部> が評価されてその値が返される。もし <テスト> が偽値をもたらし，かつ <代替部> が規定されていなければ，式の結果は未規定である。

```
(if (> 3 2) 'yes 'no)         ⟹  yes
(if (> 2 3) 'yes 'no)         ⟹  no
(if (> 3 2)
    (- 3 2)
    (+ 3 2))                  ⟹  1
```

4.1.6. 代入

(set! <変数> <式>) 構文

意味: <式> が評価され，<変数> が束縛されている場所に結果の値が格納される。<変数> が set! 式を取り囲むなんらかの領域で，または大域的に束縛されていなければエラーである。set! 式の結果は未規定である。

```
(define x 2)
(+ x 1)                       ⟹  3
(set! x 4)                    ⟹  未規定
(+ x 1)                       ⟹  5
```

4.1.7. 包含 (Inclusion)

(include <文字列 $_1$> <文字列 $_2$> ...) 構文
(include-ci <文字列 $_1$> <文字列 $_2$> ...) 構文

意味: include および include-ci はどちらも文字列リテラルとして 1 つ以上のファイル名をとり，対応するファイルを検索する実装固有のアルゴリズムを適用して read の繰り返しのアプリケーションであるかのように指定された順序でファイルの内容を読み，効果的に include または include-ci 式を，ファイルから読み込まれたものを含む begin 式に置き換える。両者の違いは，include-ci は #!fold-case ディレクティブで始まったかのように，また include はそうせずに各ファイルを読み込む。

注: 実装は，包含しているファイルがあるディレクトリ内のファイルを検索すること，および，検索する他のディレクトリを指定する方法をユーザに提供することが奨励される。

4.2. 派生式型

本節のコンストラクトは，4.3 節で論ずるような意味で，保健的である。典拠として，7.3 節は，本節で記述するコンストラクトの大部分をさきの節で記述した原始コンストラクトへと変換する構文定義を与える。

4.2.1. 条件式

`(cond <節 1> <節 2> ...)` 構文
`else` 補助構文
`=>` 補助構文

構文: <節> は二つの形式のいずれか一つをとる。

`(<テスト> <式 1> ...)`

ここで <テスト> は任意の式である。あるいは

`(<テスト> => <式>)`

最後の <節> は "else 節" にもなりうる。これは次の形式をとる。

`(else <式 1> <式 2> ...)`.

意味: cond 式は，一連の <節> の <テスト> 式を，その一つが真値 (6.3 節参照) へと評価されるまで順に評価することによって評価される。ある <テスト> が真値へと評価されると，その <節> の残りの各 <式> が順に評価され，そしてその <節> の最後の <式> の結果がその cond 式全体の結果として返される。

もし選択された <節> に <テスト> だけあって <式> がなければ，<テスト> の値が結果として返される。もし選択された <節> が => 代替形式を使っているならば，その <式> が評価される。値が 1 引数をとる手続きでなければエラーである。それからこの手続きが <テスト> の値を引数として呼び出され，そしてこの手続きが返した値が cond 式によって返される。

もしすべての <テスト> が #f へと評価されたとき，else 節がなければその条件式の結果は未規定であるが，else 節があればその各 <式> が順に評価されてその最後の式の値が返される。

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))      ⟹  greater
(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))        ⟹  equal
(cond ((assv 'b '((a 1) (b 2))) => cadr)
      (else #f))            ⟹  2
```

`(case <キー> <節 1> <節 2> ...)` 構文

構文: <キー> は任意の式がなりうる。各 <節> は次の形式をとる。

`((<データ 1> ...) <式 1> <式 2> ...)`,

ここで各 <データ> は，なんらかのオブジェクトの外部表現である。式のどの任意の <データ> も異なっていなければエラーである。代わりに，<節>は次の形式をとる。

`((<データ 1> ...) => <式>)`

最後の <節> は "else 節" にもなりうる。これは次の形式をとる。

`(else <式 1> <式 2> ...)`

または

`(else => <式>)`.

意味: case 式は次のように評価される。<キー> が評価され，その結果が各 <データ> と比較される。もし <キー> を評価した結果が，ある <データ> と (eqv? の意味で) 同じならば (6.1 節参照)，対応する <節> の各式が左から右へと順に評価され，そしてその <節> の最後の式の結果が case 式の結果として返される。

もし <キー> を評価した結果がどの <データ> とも異なるとき，else 節があれば，その各式が評価されてその最後の結果が case 式の結果になるが，なければ case 式の結果は未規定である。

もし選択された <節> または else 節が => の代替形式を使っていれば，<式> が評価される。値が 1 引数をとる手続きでなければエラーである。この手続きはこのとき，<キー> の値で呼ばれ，この手続きによって返される値は， case 式によって返される。

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⟹  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                 ⟹  未規定
(case (car '(c d))
  ((a e i o u) 'vowel)
  ((w y) 'semivowel)
  (else => (lambda (x) x))) ⟹  c
```

`(and <テスト 1> ...)` 構文

意味: 各 <テスト> 式が左から右へと評価され，もし任意の式が#f (6.3 節参照) へと評価されたとき，#f が返される。残りの式は評価されない。もしすべての式が真値へと評価されたならば，最後の式の値が返される。もし式が一つもなければ #t が返される。

```
(and (= 2 2) (> 2 1))       ⟹  #t
(and (= 2 2) (< 2 1))       ⟹  #f
(and 1 2 'c '(f g))         ⟹  (f g)
(and)                       ⟹  #t
```

`(or <テスト 1> ...)` 構文

意味: 各 <テスト> 式が左から右へと評価され，真値 (6.3 節参照) へと評価された最初の式の値が返される。残りの式は評価されない。もしすべての式が #f へと評価されたか，もし式が一つもなければ #f が返される。

```
(or (= 2 2) (> 2 1))        ⟹  #t
(or (= 2 2) (< 2 1))        ⟹  #t
(or #f #f #f)               ⟹  #f
(or (memq 'b '(a b c))
    (/ 3 0))                ⟹  (b c)
```

`(when <テスト> <式 1> <式 2> ...)` 構文

構文: <テスト> は一つの式である。

意味: テストが評価され，それが真値へと評価されたならば，式が順に評価される。when 式の結果は未規定である。

```
(when (= 1 1.0)
  (display "1")
  (display "2"))            ⟹  未規定
          and prints  12
```

(unless <テスト> <式 $_1$> <式 $_2$> ...) 構文

構文: <テスト> は一つの式である。

意味: テストが評価され，それが #f へと評価されたならば，式が順に評価される。unless 式の結果は未規定である。

```
(unless (= 1 1.0)
  (display "1")
  (display "2"))            ⟹  未規定
          and prints nothing
```

(cond-expand <ce 節 $_1$> <ce 節 $_2$> ...) 構文

構文: cond-expand 式型は実装に依存する異なる式を静的にする展開方法を提供する。<ce 節> は次の形式をとる。

(<機能要件> <式> ...)

最後の <節> は “else 節” にもなりうる。これは次の形式をとる。

(else <式> ...)

<機能要件> は次の形式のいずれか一つをとる。

- <機能識別子>
- (library <ライブラリ名>)
- (and <機能要件> ...)
- (or <機能要件> ...)
- (not <機能要件>)

意味: 各実装は，インポートすることができるライブラリのリストと同様に，存在している機能識別子のリストを管理する。<機能要件> の値は，実装のリスト上の各 <機能識別子> および (library <ライブラリ名>) を #t に，他の機能識別子とライブラリ名を #f に置き換えることよって決まる。このとき，and, or, および not の通常の解釈のもとで Scheme のブーリアン式として結果の式が評価される。

cond-expand はこのとき，それらの一つが #t を返すまで連続した <ce 節>の<機能要件>を順に評価することによって展開される。真の句が見つかったとき，対応する <式> が begin に展開され，残りの句は無視される。<機能要件> が一つも #t に評価されない場合，もし else 句があれば，その <式>が含まれる。そうでない場合， cond-expand の振舞いは未規定である。cond と違って， cond-expand は任意の変数の値に依存しない。

提供される正確な機能は実装定義であるが，移植性のために機能のコア集合は付録 B に記載されている。

4.2.2. 束縛コンストラクト

束縛コンストラクト let, let*, letrec letrec*, let-values, および let*-values は，Scheme に Algol 60 のようなブロック構造を与える。最初の四つのコンストラクトの構文は同一だが，その変数束縛のためにそれらが設ける領域が異なる。let 式は，各初期値をすべて計算してから変数を束縛する。let* 式は，束縛と評価を一つ一つ逐次的に行う。しかるに letrec と letrec* 式では，初期値を計算している間すべての束縛が有効であり，したがって相互再帰的な定義が可能である。let-values および let*-values コンストラクトはそれぞれ let と let* と似ているが，複数値の式を操作するために設計されており，異なる識別子を戻り値に束縛する。

(let <束縛部> <本体>) 構文

構文: <束縛部> は次の形式をとること。

((<変数 $_1$> <初期値 $_1$>) ...),

ここで各 <初期値> は式で， <本体> は 4.1.4 節に記述されているような 1 個以上の式からなる列に続く 0 個以上の定義からなる列である。束縛される変数のリストの中に一つの <変数> が複数回現れることはエラーである。

意味: 各 <初期値> が現在の環境の中で (ある未規定の順序で) 評価される。その結果を保持する新しい場所へ，それぞれの <変数> が束縛される。その拡張された環境の中で <本体> が評価される。そして <本体> の最後の式の値が返される。<変数> のどの束縛も <本体> をその領域とする。

```
(let ((x 2) (y 3))
  (* x y))                  ⟹  6

(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))               ⟹  35
```

4.2.4 節の “名前付き let” も見よ。

(let* <束縛部> <本体>) 構文

構文: <束縛部> は次の形式をとる。

((<変数 $_1$> <初期値 $_1$>) ...),

かつ <本体> は，4.1.4 節に記述されているような 1 個以上の式に続く 0 個以上の定義からなる列である。

意味: let* 束縛コンストラクトは let と似ているが，束縛は左から右へと順に実行され，(<変数> <初期値>) で示される束縛の領域は，束縛の右側の let* 式の一部である。したがって 2 番目の束縛は 1 番目の束縛が可視である環境でなされる等々である。<変数> は別個である必要はない。

```
(let ((x 2) (y 3))
  (let* ((x 7)
         (z (+ x y)))
    (* z x)))               ⟹  70
```

(letrec <束縛部> <本体>) 構文

構文: <束縛部> は次の形式をとる。

((<変数 $_1$> <初期値 $_1$>) ...),

かつ <本体> は，4.1.4 節に記述されているような 1 個以上の式に続く 0 個以上の定義からなる列である。束縛される変数のリストの中に一つの <変数> が複数回現れることはエラーである。

意味: 未規定値を保持する新しい場所へ，それぞれの <変数> が束縛される。その結果として得られた環境で，各 <初期値> が (ある未規定の順序で) 評価される。各 <変数> にそれぞれ対応する <初期値> の結果が代入される。その結果として得られた環境で， <本体> が評価される。そして <本体> の最後の式の値が返される。<変数> のどの束縛も letrec 式全体をその領域とする。これは相互再帰的な手続きを定義することを可能にしている。

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                                ⟹  #t
```

letrec の一つの制限はとても重要である。各 <初期値> は，どの <変数> の値も参照ないし代入することなく評価することが可能でなければエラーである。この制限が必要なのは，letrec が手続き呼び出しの意味で定義されるからであり，ここで lambda 式は変数を <初期値> の値に束縛する。最もありふれた letrec の用途では，<初期値> はすべて lambda 式であり，この制限は自動的に満たされる。

(letrec* <束縛部> <本体>) 構文

構文: <束縛部> は次の形式をとる。

((<変数 $_1$> <初期値 $_1$>) ...),

かつ <本体> は，4.1.4 節に記述されているような 1 個以上の式に続く 0 個以上の定義からなる列である。束縛される変数の並びの中に一つの <変数> が複数回現れることはエラーである。

意味: 新しい場所へ，それぞれの <変数> が束縛される。各 <変数> にそれぞれ対応する <初期値> の評価結果が左から右へと順に代入される。その結果として得られた環境で <本体> が評価される。そして <本体> の最後の式の値が返される。左から右への評価と割り当ての順序にもかかわらず，<変数> のどの束縛も letrec* 式全体をその領域とする。これは相互再帰的な手続きを定義することを可能にしている。

対応する <変数> の値を代入または参照なしに各 <初期値> または <束縛部> でその後に続く任意の束縛の<変数>を評価できない場合，エラーである。もう一つの制約は，一回以上の <初期値> の継続を呼び出すことはエラーである。

```
(letrec* ((p
           (lambda (x)
             (+ 1 (q (- x 1)))))
          (q
           (lambda (y)
             (if (zero? y)
                 0
                 (+ 1 (p (- y 1))))))
          (x (p 5))
          (y x))
  y)
                                ⟹  5
```

(let-values <mv 束縛仕様> <本体>) 構文

構文: <mv 束縛仕様> は次の形式をとる。

((<仮引数部 $_1$> <初期値 $_1$>) ...),

ここで，各 <初期値> は式であり，<本体> は，4.1.4 節に記述されているような 1 個以上の式からなる列に続く 0 個以上の定義である。<仮引数部> の中に一つの変数が複数回現れることはエラーである。

意味: <初期値> が，call-with-values を呼び出すかのように現在の環境で (ある未規定の順序で) 評価される。<仮引数部> で起こる変数は，<初期値>によって返される値を保持する新しい場所へ束縛される。ここで， <仮引数部> は，lambda 式内の <仮引数部> が手続き呼び出し中の引数に照合されるのと同じ方法で戻り値に照合する。このとき，<本体> が拡張された環境の中で評価され， <本体> の最後の式の値が返される。<変数> のどの束縛も <本体> をその領域とする。

<仮引数部> が，対応する <初期値> によって返される値の数と一致しない場合，エラーである。

```
(let-values (((root rem) (exact-integer-sqrt 32)))
  (* root rem))                 ⟹  35
```

(let*-values <mv 束縛仕様> <本体>) 構文

構文: <mv 束縛仕様> は次の形式をとる。

((<仮引数部> <初期値>) ...),

そして <本体> は，4.1.4 節に記述されているような 1 個以上の式に続く 0 個以上の定義からなる列である。各 <仮引数部> の中に一つの変数が複数回現れることはエラーである。

意味: let*-values コンストラクトは let-values と似ているが，<初期値> は評価され，束縛は <本体> と同様にその右側に<初期値>を含んでいる各<仮引数部>の束縛の領域とともに，左から右へと順に作成される。したがって，二番目の <初期値> は束縛の最初の集合が可視で初期化されているような環境で評価される，などである。

```
(let ((a 'a) (b 'b) (x 'x) (y 'y))
  (let*-values (((a b) (values x y))
                ((x y) (values a b)))
    (list a b x y)))        ⟹ (x y x y)
```

4.2.3. 逐次式

Scheme 逐次コンストラクトは両方とも begin と名付けられるが，二つはわずかに形式と用途が異なる。

(begin <式または定義> ...) 構文

begin の形式は，<本体> の一部として，または <プログラム> の最も外側のレベルに，または REPL に，またはこの形式自身が begin 内に直接入れ子になって現れることができる。

根拠: この形式は，複数の定義を生成するのとそれらが展開されている文脈に継ぎ合せるのに必要なマクロ (4.3 節参照) の出力で一般的に使われる。

(begin <式 $_1$> <式 $_2$> ...) 構文

begin のこの形式は，通常の式として使うことができる。各 <式> が左から右へと順に評価され，最後の <式> の値が返される。この式型は，代入や入出力などの副作用を順序どおりに起こすために使われる。

```
(define x 0)

(and (= x 0)
     (begin (set! x 5)
            (+ x 1)))       ⟹  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))   ⟹  未規定
           および  4 plus 1 equals 5  を印字する
```

ここで，ライブラリ宣言として使われる begin の第三の形式があることに注意せよ。5.6.1 節参照。

4.2.4. 繰返し

(do ((<変数 $_1$> <初期値 $_1$> <ステップ $_1$>) 構文
    ...)
   (<テスト> <式> ...)
 <コマンド> ...)

構文: <初期値>, <ステップ>, <テスト>, および <コマンド> はすべて式である。

意味: do 式は繰返しのコンストラクトである。これは束縛すべき変数の集合と，それらを繰返し開始時にどう初期化するか，そして繰返しごとにどう更新するかを規定する。終了条件が満たされた時，ループは各 <式> を評価して終わる。

do 式は次のように評価される。各 <初期値> 式が (ある未規定の順序で) 評価され，各 <変数> が新しい場所に束縛され，各 <初期値> 式の結果が各 <変数> の束縛に格納され，そして繰返しが始まる。

それぞれの繰返しは <テスト> を評価することで始まる。もしその結果が偽 (6.3 節参照) ならば，副作用を期待して各 <コマンド> 式が順に評価され，各 <ステップ> 式がある未規定の順序で評価され，各 <変数> が新しい場所に束縛され，各 <ステップ> の結果が各 <変数> の束縛に格納され，そして次の繰返しが始まる。

もし <テスト> が真値へと評価されたならば，各 <式> が左から右へと評価され，最後の <式> の値が返される。もし <式> がないならば，その do 式の値は未規定である。

<変数>の束縛の領域は，全 <初期値> を除く do 式全体である。do 変数のリストの中に一つの <変数> が複数回現れることはエラーである。

<ステップ> を省略してもよい。このとき，その効果は，(<変数> <初期値>) ではなく (<変数> <初期値> <変数>) と書いた場合と同じである。

```
(do ((vec (make-vector 5))
     (i 0 (+ i 1)))
    ((= i 5) vec)
  (vector-set! vec i i))    ⟹  #(0 1 2 3 4)

(let ((x '(1 3 5 7 9)))
  (do ((x x (cdr x))
       (sum 0 (+ sum (car x))))
      ((null? x) sum)))     ⟹  25
```

(let <変数> <束縛部> <本体>) 構文

意味: "名前付き let" は let の構文の一変種である。これは do よりも一般性の高い繰返しコンストラクトを定めており，再帰を表現するために使うこともできる。その構文と意味は，<変数> が <本体> の中で，束縛変数を仮引数とし <本体> を本体とする手続きへと束縛されることを除いては，通常の let と同じである。したがって <変数> という名前の手続きを呼び出すことによって，<本体> の実行を繰り返すことができる。

```
(let loop ((numbers '(3 -2 1 6 -5))
           (nonneg '())
           (neg '()))
  (cond ((null? numbers) (list nonneg neg))
        ((>= (car numbers) 0)
         (loop (cdr numbers)
               (cons (car numbers) nonneg)
               neg))
        ((< (car numbers) 0)
         (loop (cdr numbers)
               nonneg
               (cons (car numbers) neg)))))
          ⟹  ((6 1 3) (-5 -2))
```

4.2.5. 遅延評価

(delay <式>) lazy ライブラリ構文

意味: delay コンストラクトは，**遅延評価** (*lazy evaluation*) または *call by need* を実装するために，手続き force とともに使われる。(delay <式>) は**約束** (*promise*) と呼ばれるオブジェクトを返す。このオブジェクトは未来のある時点で <式> を評価して結果の値を (force 手続きによって) 求め

ることができ，結果として得られた値を受渡す。 複数個の値を返す <式> の効果は未規定である。

(delay-force <式>) lazy ライブラリ構文

意味: 式 (delay-force *variables*) は，概念的に (delay (force *expression*)) と似ているが，delay-force の結果を強制することは事実上 (force *expression*) に対する末尾呼び出しになるが，(delay (force *expression*)) の結果を強制することは，そうではないかもしれない，という違いがある。したがって，繰り返し遅延アルゴリズムは delay と force の長い連鎖となる可能性があり，評価中に無制限のスペースを消費しないように delay-force を使って書き換えることができる。

(force *promise*) lazy ライブラリ手続き

force 手続きは， delay, delay-force, または make-promise で作られた *promise* の値を強制する。もしも約束に対してなんら値が計算されていなければ，一つの値が計算されて返される。約束の値は，もしそれが二度目に強制された場合，以前に計算された値が返されるように，キャッシュ (または “メモ化”) されなければならない。その結果，遅延式はパラメータ値と最初にその値を要求された force への呼び出しの例外ハンドラを用いて評価される。*promise* が約束でない場合，そのまま返してもよい。

```
(force (delay (+ 1 2)))     ⟹  3
(let ((p (delay (+ 1 2))))
  (list (force p) (force p)))
                            ⟹  (3 3)

(define integers
  (letrec ((next
            (lambda (n)
              (delay (cons n (next (+ n 1)))))))
    (next 0)))
(define head
  (lambda (stream) (car (force stream))))
(define tail
  (lambda (stream) (cdr (force stream))))

(head (tail (tail integers)))
                            ⟹  2
```

次の例は，Scheme への遅延ストリームフィルタリングアルゴリズムの機械的変換である。コンストラクタへの各呼び出しは delay で覆われ，デコンストラクタに渡される各引数は force で覆われる。手続きの本体周辺で (delay-force ...) を (delay (force ...)) の代わりに使用することは，force が事実上そのような列を反復的に強制するので，増え続ける保留中の約束のシーケンスが使用可能な記憶領域を排出しないことを保証する。

```
(define (stream-filter p? s)
  (delay-force
   (if (null? (force s))
       (delay '())
       (let ((h (car (force s)))
             (t (cdr (force s))))
         (if (p? h)
             (delay (cons h (stream-filter p? t)))
             (stream-filter p? t))))))

(head (tail (tail (stream-filter odd? integers))))
                            ⟹ 5
```

delay, force, および delay-force は主に機能的なスタイルで書かれたプログラムを対象としているので，以下の例は良いプログラミングスタイルを説明する意図はない。しかしそれらは，それが強制された回数に関係なくただ一つの値が約束のために計算されるという特性を示している。

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                           ⟹  a promise
(force p)                   ⟹  6
p                           ⟹  a promise, still
(begin (set! x 10)
       (force p))           ⟹  6
```

delay, force および delay-force のこの意味論へのさまざまな拡張が，いくつかの実装でサポートされている:

- 約束ではないオブジェクトに対する force の呼出しは，単にそのオブジェクトを返してよい。
- 約束をその強制された値から操作的に区別する手段が全くないというケースがあってもよい。つまり，次のような式は，実装に依存して，#t と #f のどちらに評価されてもよい:

```
(eqv? (delay 1) 1)          ⟹  未規定
(pair? (delay (cons 1 2)))  ⟹  未規定
```

- 実装は，cdr や * のように，約束の値が特定の型の引数でのみで演算する手続きによって強制されるという “暗黙の強制” (implicit forcing) を実装してもよい。しかし，list のように，引数に均一に演算する手続きはそれらを強制してはいけない。

```
(+ (delay (* 3 7)) 13)      ⟹  未規定
(car
  (list (delay (* 3 7)) 13))⟹  a promise
```

(promise? *obj*) lazy ライブラリ手続き

promise? 手続きは，その引数が約束の場合 #t を返し，それ以外の場合 #f を返す。約束は，手続きのような他の Scheme 型から必ずしも分離的ではないことに注意せよ。

(make-promise *obj*) lazy ライブラリ手続き

make-promise 手続きは，強制されたときに *obj* を返すような約束を返す。これは delay と似ているが，その引数を遅延しない: それは構文よりも手続きである。*obj* が既に約束であれば，それが返される。

4.2.6. 動的束縛

手続き呼び出しの動的寿命 (*dynamic extent*) は，それが開始されてから返されるまでの時間である。Scheme では，call-with-current-continuation (6.10 節) はその手続き呼び出しが返された後に動的寿命に再び入ることを許可する。したがって，呼び出しの動的寿命は単一の連続した時間帯ではないかもしれない。

本節では，動的寿命の期間中，新しい値に束縛できるパラメータオブジェクト (*parameter objects*) を紹介する。与えられた時間におけるすべてのパラメータ束縛の集合は，動的環境 (*dynamic environment*) と呼ばれる。

(make-parameter *init*) 手続き
(make-parameter *init converter*) 手続き

新たに割り当てられたパラメータオブジェクトを返す。それは 0 個の引数を受け入れてパラメータオブジェクトに関連付けられた値を返す手続きである。まず，変換手続き *converter* が指定されていない場合，この値は (*converter init*) または *init*の値である。関連付けられた値は，以下に記載された parameterize を用いて，一時的に変更できる。

パラメータオブジェクトに引数を渡す効果は実装依存である。

(parameterize ((<パラメータ $_1$> <値 $_1$>) ...) 構文
  <本体>)

構文: <パラメータ $_1$> および <値 $_1$> は，両方とも式である。

任意の <パラメータ> 式の値がパラメータオブジェクトでなければ，エラーである。

意味: parameterize 式は，本体の評価時に指定したパラメータオブジェクトによって返される値を変更するのに使われる。

<パラメータ $_1$> および <値 $_1$> 式は未規定の順序で評価される。<本体> は，パラメータ作成時に，対応する値を指定された変換手続きに渡した結果をパラメータが返すように呼び出している動的環境で評価される。このとき，パラメータの前の値は，それらを変換手続きへ渡すことなく復元される。<本体> 中の最後の式の結果が，parameterize 式全体の結果として返される。

注: 変換手続きが冪等でない場合，パラメータ x をその現在値に束縛することを表す (parameterize ((x (x))) ...) の結果は，ユーザが期待するものではないかもしれない。

実装が複数の実行スレッドをサポートする場合，parameterize は，現在のスレッドと <本体> 内で作られたスレッド以外のスレッドで任意のパラメータの関連付けられた値を変更してはいけない。

パラメータオブジェクトは，明示的に呼び出しチェーン内のすべての手続きに値を渡すことを必要としないで計算のための構成可能な設定を指定するのに使うことができる。

```
(define radix
  (make-parameter
   10
   (lambda (x)
     (if (and (exact-integer? x) (<= 2 x 16))
         x
         (error "invalid radix")))))

(define (f n) (number->string n (radix)))

(f 12)                          ⟹ "12"
(parameterize ((radix 2))
  (f 12))                       ⟹ "1100"
(f 12)                          ⟹ "12"

(radix 16)                      ⟹ 未規定

(parameterize ((radix 0))
  (f 12))                       ⟹ エラー
```

4.2.7. 例外処理

(guard (<変数>
        <cond 節 $_1$> <cond 節 $_2$> ...)
  <本体>) 構文

構文: 各 <cond 節> は， cond の指定と同様である。

意味: <本体> は，その束縛スコープ内で発生させたオブジェクト (6.11 節の raise 参照) を<変数> に束縛して cond 式の節であるかのように節を評価する例外ハンドラで評価される。暗黙の cond 式は，継続と guard 式の動的環境で評価される。すべての <cond 節> の <テスト> が #f に評価され，else 節がない場合，raise-continuable は，現在の例外ハンドラが guard 式のものであることを除いて，raise または raise-continuable の元の呼び出しの動的環境で発生させたオブジェクトに対して呼び出される。

例外のより完全な議論については，6.11 節を見よ。

```
(guard (condition
        ((assq 'a condition) => cdr)
        ((assq 'b condition)))
  (raise (list (cons 'a 42))))
                                ⟹ 42

(guard (condition
        ((assq 'a condition) => cdr)
        ((assq 'b condition)))
  (raise (list (cons 'b 23))))
                                ⟹ (b . 23)
```

4.2.8. 準引用

(quasiquote <qq テンプレート>)	構文
`<qq テンプレート>	構文
unquote	補助構文
,	補助構文
unquote-splicing	補助構文
,@	補助構文

“準引用” 式は，リストまたはベクタのあるべき構造が前もっていくつか分かっているが，ただし完全にではない場合に，そのリストまたはベクタ構造を構築するのに有用である。もしコンマが <qq テンプレート> の中に一つも現れていなければ，`<qq テンプレート> を評価した結果は ’<qq テンプレート> を評価した結果と等価である。しかし，もしコンマが <qq テンプレート> の中に現れているならば，そのコンマに続く式が評価され (“unquote” され)，そしてその結果がコンマと式のかわりに構造の中に挿入される。もしコンマの後に空白が間に入ることなくアットマーク (@) が続くならば，それに続く式はリストへと評価されなければならない。この場合，そのリストの両端の丸カッコが “はぎとられ”，リストの各要素が コンマ-アットマーク-式 の列の位置に挿入される。コンマ-アットマークは，通常リストまたはベクタの <qq テンプレート> の中にだけ現れる。

注: コンマ-アットマーク 列との衝突を避けるため，@ で始まる識別子を unquote するのに明示的な unquote の使用，あるいは コンマの後に空白を置くことが必要である。

```
`(list ,(+ 1 2) 4)          ⟹  (list 3 4)
(let ((name 'a)) `(list ,name ',name))
          ⟹  (list a (quote a))
`(a ,(+ 1 2) ,@(map abs '(4 -5 6)) b)
          ⟹  (a 3 4 5 6 b)
`(( foo ,(- 10 3)) ,@(cdr '(c)) . ,(car '(cons)))
          ⟹  ((foo 7) . cons)
`#(10 5 ,(sqrt 4) ,@(map sqrt '(16 9)) 8)
          ⟹  #(10 5 2 4 3 8)
(let ((foo '(foo bar)) (@baz 'baz))
  `(list ,@foo , @baz))
          ⟹  (list foo bar baz)
```

準引用式は入れ子にすることができる。置換は，最も外側の準引用と同じ入れ子レベルで現れる被 unquote 要素に対してだけ行われる。入れ子レベルは，準引用に入って行くたびに 1 だけ増え，unquote される要素に入って行くたびに 1 だけ減る。

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
          ⟹  (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
(let ((name1 'x)
      (name2 'y))
  `(a `(b ,,name1 ,',name2 d) e))
          ⟹  (a `(b ,x ,'y d) e)
```

準引用式は，新たに割り当てられた書換え可能オブジェクト，または式の評価中に実行時に構築された任意の構造のリテラル構造を返してもよい。再ビルドが必要ない部分は，常にリテラルである。したがって，

```
(let ((a 3)) `((1 2) ,a ,4 ,'five 6))
```

は，次の式のいずれかと等価に扱われてもよい:

```
`((1 2) 3 4 five 6)

(let ((a 3))
  (cons '(1 2)
        (cons a (cons 4 (cons 'five '(6))))))
```

しかし，この式とは等価ではない:

```
(let ((a 3)) (list (list 1 2) a 4 'five 6))
```

`<qq テンプレート> と (quasiquote <qq テンプレート>) の二つの表記はすべての点で同一である。,<式> は (unquote <式>) と同一である。,@<式> は (unquote-splicing <式>) と同一である。write 手続きは，どちらの形式を出力してもよい。

```
(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  (list 3 4)
'(quasiquote (list (unquote (+ 1 2)) 4))
          ⟹  `(list ,(+ 1 2) 4)
     i.e., (quasiquote (list (unquote (+ 1 2)) 4))
```

もし識別子 quasiquote，unquote，unquote-splicing のどれかが，上に述べたような位置以外のところで <qq テンプレート> の中に現れている場合は，エラーである。

4.2.9. case-lambda

(case-lambda <節> ...)　case-lambda ライブラリ構文

構文: 各 <節> は次の形式である。(<仮引数部> <本体>), ここで， <仮引数部> および <本体> は lambda 式と同じ構文をもつ。

意味: case-lambda 式は，可変個の引数を受け入れ，lambda 式に起因する手続きと同様に字句的にスコープされる手続きに評価する。手続きが呼び出されると，<仮引数部> に一致する引数をもつ最初の <節> が選択される。ここで一致は lambda 式の <仮引数部> として指定される。<仮引数部> の変数は新しい場所に束縛され，引数の値はそれらの場所に格納され，<本体> は拡張された環境で評価され，<本体> の結果は，手続き呼び出しの結果として返される。

任意の節に <仮引数部> が一致しない場合は，引数のエラーである。

```
(define range
  (case-lambda
   ((e) (range 0 e))
   ((b e) (do ((r '() (cons e r))
               (e (- e 1) (- e 1)))
              ((< e b) r)))))

(range 3)                  ⟹ (0 1 2)
(range 3 5)                ⟹ (3 4)
```

4.3. マクロ

Scheme プログラムは新しい派生式型を定義して使用することができる。この式型を**マクロ** (*macro*) と呼ぶ。プログラムで定義される式型は次の構文をとる。

```
(<キーワード> <データ> ...)
```

ここで <キーワード> はその式型を一意的に決定する識別子である。この識別子をマクロの**構文キーワード** または単に**キーワード**と呼ぶ。<データ> の個数とその構文はその式型に依存する。

マクロのインスタンスはそれぞれ，そのマクロの**使用** (*use*) と呼ばれる。あるマクロの使用がより原始的な式へとどのように変換されるのかを規定する規則の集合を，そのマクロの**変換子** (*transformer*) と呼ぶ。

マクロ定義の手段は二つの部分からなる。

- 特定の識別子がマクロキーワードであることを確立し，それらをマクロ変換子と結合させ，そしてマクロが定義されるスコープを制御する，ということのために使われる式の集合
- マクロ変換子を規定するためのパターン言語

マクロの構文キーワードが変数束縛を隠蔽してもよいし，ローカル変数束縛がキーワード構文束縛を隠蔽してもよい。二つのメカニズムは意図しない衝突を防止するために設けられている。

- マクロ変換子が，ある識別子 (変数またはキーワード) に対する束縛を挿入するとき，その識別子は，他の識別子との衝突を避けるために，そのスコープ全体にわたって実質的に改名される。グローバル変数定義は束縛を導入してもしなくてもよいことに注意せよ (5.3 節参照)。
- マクロ変換子が，ある識別子への自由な参照を挿入するとき，その参照は，そのマクロ使用を取り囲むローカル束縛には一切関係なく，もともとその変換子が規定された箇所で可視だった束縛を参照する。

結果として，パターン言語を使って定義されるすべてのマクロは “保健的” (hygienic) かつ “参照透過的” (referentially transparent) であり，したがって Scheme の字句的スコーピングを侵さない。[21, 22, 2, 9, 12]

実装は，他の型のマクロ手段を提供してもよい。

4.3.1. 構文キーワードのための束縛コンストラクト

`let-syntax` と `letrec-syntax` 束縛コンストラクトは，`let` と `letrec` に相当するが，変数を値の保持場所に束縛するかわりに，構文キーワードをマクロ変換子に束縛する。構文キーワードを大域的または `define-syntax` で局所的に束縛することもできる。5.4 節を見よ。

`(let-syntax <束縛部> <本体>)` 構文

構文: <束縛部> は次の形式をもつ。

```
((<キーワード> <変換子仕様>) ...)
```

各 <キーワード> は識別子である。各 <変換子仕様> は `syntax-rules` のインスタンスである。<本体> は 1 個以上の定義に続く 1 個以上の式からなる列であること。束縛されるキーワードの並びの中に一つの <キーワード> が複数回現れることはエラーである。

意味: `let-syntax` 式の構文環境をマクロで拡張することによって得られる構文環境の中で，<本体> が展開される。ただし，それらのマクロはそれぞれ <キーワード> をキーワードとし，仕様が規定する変換子に束縛されている。各 <キーワード> の束縛は <本体> をその領域とする。

```
(let-syntax ((given-that (syntax-rules ()
                     ((given-that test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (given-that if (set! if 'now))
    if))                    ⟹  now

(let ((x 'outer))
  (let-syntax ((m (syntax-rules () ((m) x))))
    (let ((x 'inner))
      (m))))                ⟹  outer
```

`(letrec-syntax <束縛部> <本体>)` 構文

構文: `let-syntax` と同じである。

意味: `letrec-syntax` 式の構文環境をマクロで拡張することによって得られる構文環境の中で，<本体> が展開される。ただし，それらのマクロはそれぞれ <キーワード> をキーワードとし，仕様が規定する変換子に束縛されている。各 <キーワード> の束縛は，<本体> ばかりでなく <変換子仕様> をもその領域とするから，変換子は式を，`letrec-syntax` 式が導入するマクロの使用へと変換できる。

```
(letrec-syntax
    ((my-or (syntax-rules ()
              ((my-or) #f)
              ((my-or e) e)
              ((my-or e1 e2 ...)
               (let ((temp e1))
                 (if temp
                     temp
                     (my-or e2 ...)))))))
  (let ((x #f)
        (y 7)
        (temp 8)
        (let odd?)
        (if even?))
    (my-or x
           (let temp)
           (if y)
           y)))             ⟹  7
```

4.3.2. パターン言語

<変換子仕様> は次のうちいずれか一つの形式をとる。

```
(syntax-rules (<リテラル部> ...)                    構文
  <構文規則> ...)
(syntax-rules <省略符号> (<リテラル部> ...)          構文
  <構文規則> ...)
_                                                   補助構文
...                                                 補助構文
```

構文: 任意の <リテラル部>, または二番目の形式の <省略符号> が識別子でなければエラーである。また, <構文規則> が

```
(<パターン> <テンプレート>)
```

の形式でなければエラーである。<構文規則> におかれる <パターン> は, 最初の要素が識別子であるリスト <パターン> である。

<パターン> は識別子, 定数, または次の一つである。

```
(<パターン> ...)
(<パターン> <パターン> ... . <パターン>)
(<パターン> ... <パターン> <省略符号> <パターン> ...)
(<パターン> ... <パターン> <省略符号> <パターン> ...
   . <パターン>)
#(<パターン> ...)
#(<パターン> ... <パターン> <省略符号> <パターン> ...)
```

そして <テンプレート> は識別子, 定数, または次の一つである。

```
(<要素> ...)
(<要素> <要素> ... . <テンプレート>)
(<省略符号> <テンプレート>)
#(<要素> ...)
```

ここで, <要素> は任意の <省略符号> が続く一つの <テンプレート> である。<省略符号> は, syntax-rules の二番目の形式で指定された識別子, あるいはそれ以外の場合, デフォルトの識別子 "..." (三つの連続するピリオド) である。

意味: syntax-rules のインスタンスは, 保健的な書換え規則の列を規定することによって, 新しいマクロ変換子を生成する。マクロの使用は, そのキーワードと結合している変換子を規定する syntax-rules の, 各 <構文規則> に収められたパターンに対して照合される。照合は最左の <構文規則> から開始される。照合が見つかった時, マクロ使用は, テンプレートに従って保健的に変換される。

パターン内に現れる識別子は, アンダースコア (_), <リテラル部>のリストに記載されているリテラル識別子, または<省略符号>のいずれかにできる。<パターン> に現れる他の識別子はすべて**パターン変数**である。

<構文規則> のパターンの先頭にあるキーワードは, 照合に関与せず, それはパターン変数ともリテラル識別子とも見なされない。

パターン変数は, 任意の入力要素と照合し, 入力の各要素をテンプレートの中で参照するために使われる。<パターン> の中に同じパターン変数が複数回現れることはエラーである。

アンダースコアもまた, 任意の入力要素と照合するが, パターン変数ではなく, これらの要素を参照するためには使われない。もしアンダースコアが <リテラル部> のリストに現れた場合は, 順位をとり<パターン> の中のアンダースコアは, リテラルとして照合する。複数のアンダースコアは <パターン> の中に現れることができる。

(<リテラル部> ...) に現れる識別子は, リテラル識別子と解釈される。これは入力からの対応する要素と照合されることになる。入力における要素がリテラル識別子と照合するための必要十分条件は, それが識別子であって, かつそれのマクロ式における出現とマクロ定義における出現がともに同じ字句的束縛をもつか, あるいは二つの識別子が同じかつどちらも字句的束縛をもたないことである。

<省略符号> が後続する部分形式は, リテラルとして照合する <省略符号> が <リテラル部> の中に現れない限り, 入力の 0 個以上の要素と照合できる。

より形式的には, 入力式 E はパターンの P が次の場合に限り照合する。

- P がアンダースコア (_) である。
- P がリテラル識別子でない識別子である。または
- P がリテラル識別子であり, かつ E が同じ束縛をもつ識別子である。または
- P がリスト (P_1 ... P_n) であり, かつ E が, P_1 から P_n までそれぞれ照合する n 個の要素からなるリストである。または
- P が非真正リスト (P_1 P_2 ... P_n . P_{n+1}) であり, かつ E が, P_1 から P_n までそれぞれ照合する n 個以上の要素からなるリストまたは非真正リストであって, かつその n 番目の末尾が P_{n+1} に照合する。または
- P が次の形式である。(P_1 ... P_k P_e <省略符号> P_{m+1} ... P_n) ここで E は最初の k 個がそれぞれ P_1 から P_k まで照合し, 次の $m-k$ 個の要素がそれぞれ P_e に照合し, 残りの $n-m$ 個の要素がそれぞれ P_{m+1} から P_n までそれぞれ照合する, n 個の要素からなる真正リストである。または
- P が次の形式である。(P_1 ... P_k P_e <省略符号> P_{m+1} ... P_n . P_x) ここで E は最初の k 個がそれぞれ P_1 から P_k まで照合し, 次の $m-k$ 個の要素がそれぞれ P_e に照合し, 残りの $n-m$ 個の要素がそれぞれ P_{m+1} から P_n までそれぞれ照合し, かつ, その n 番目 と最後の cdr が P_x に照合する, n 個の要素からなる非真正リストである。または
- P が #(P_1 ... P_n) という形式のベクタであり, かつ E が, P_1 から P_n まで照合する n 個の要素からなるベクタである。または
- P が次の形式である。#(P_1 ... P_k P_e <省略符号> P_{m+1} ...P_n) ここで E は最初の k 個がそれぞれ P_1

から P_k まで照合し，次の $m-k$ 個の要素がそれぞれ P_e に照合し，残りの $n-m$ 個の要素がそれぞれ P_{m+1} から P_n までそれぞれ照合する，n 個の要素からなるベクタである。または

- P が定数であり，かつ E が P と equal? 手続きの意味で等しい。

マクロキーワードを，その束縛のスコープの内部で，どのパターンとも照合しない式に使用することは，エラーである。

マクロ使用がそれと照合する <構文規則> のテンプレートに従って変換される時，テンプレートに出現するパターン変数は，入力において照合した要素で置き換えられる。識別子 <省略符号> のインスタンスが 1 個以上後続している部分パターンの中に出現するパターン変数は，同数の <省略符号> のインスタンスが後続している部分テンプレートの中にだけ出現を許可される。出力においてそれらは，入力においてそれらが照合した要素全部によって，ただし指示どおりに分配されて，置き換えられる。仕様どおりに出力を形成できない場合はエラーである。

テンプレートに現れているがパターン変数でも識別子 <省略符号> でもない識別子は，リテラル識別子として出力に挿入される。もしリテラル識別子が自由識別子として挿入されるならば，その場合，その識別子の束縛であって，かつ元々の syntax-rules のインスタンスが現れている箇所を含んでいるようなスコープの束縛が参照される。もしリテラル識別子が束縛識別子として挿入されるならば，その場合，その識別子は，自由識別子の不慮の取り込みを防ぐために実質的に改名される。

形式のテンプレート (<省略符号> <テンプレート>) は，テンプレート内の省略符号が特別な意味を持たないことを除いて<テンプレート>と同じである。すなわち，<テンプレート> 内に含まれるの任意の省略符号は，通常の識別子として扱われる。特に，テンプレート (<省略符号> <省略符号>) は単一の <省略符号> を生成する。これは省略符号を含むコードに拡張するための構文上の抽象化を許可する。

```
(define-syntax be-like-begin
  (syntax-rules ()
    ((be-like-begin name)
     (define-syntax name
       (syntax-rules ()
         ((name expr (... ...))
          (begin expr (... ...))))))))

(be-like-begin sequence)
(sequence 1 2 3 4)          ⟹ 4
```

一例として，もし let と cond が 7.3 節のように定義されているならば，そのときそれらは (要求どおり) 保健的であり，下記はエラーにならない。

```
(let ((=> #f))
  (cond (#t => 'ok)))       ⟹ ok
```

cond のマクロ変換子は => をローカル変数として，したがって一つの式として認識する。マクロ変換子が構文キーワードとして扱う基本識別子 => としては認識しない。したがって，この例は

```
(let ((=> #f))
  (if #t (begin => 'ok)))
```

のように展開されるのであって，不正な手続き呼出しに終わることになる

```
(let ((=> #f))
  (let ((temp #t))
    (if temp ('ok temp))))
```

とはならない。

4.3.3. マクロ変換子でのエラー通知

(syntax-error <メッセージ> <引数> ...) 構文

syntax-error は，評価とは別の拡張パスでの実装がすぐに syntax-error 展開されているようにエラーを通知すべきであることを除いて，error (6.11) と同じように振舞う。これはマクロの不正使用である <パターン> のために syntax-rules <テンプレート>として使することができ，より説明的なエラーメッセージを提供することができる。<メッセージ> は文字列リテラルであり，<引数> は追加情報を提供する任意の式である。アプリケーションでは，例外ハンドラやガードで構文エラーを捕捉することができることを頼りにすることはできない。

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail)
        body1 body2 ...)
     (syntax-error
      "expected an identifier but got"
      (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
       val ...))))
```

5. プログラム構造

5.1. プログラム

Scheme プログラムは一つ以上のインポート宣言に続く式および定義の列からなる。インポート宣言は，プログラムやライブラリが依存するライブラリを指定する。ライブラリによってエクスポートされた識別子のサブセットは，プログラムで使用可能になる。式は 4 章で記述している。定義は，変数定義，構文定義，またはレコード型の定義のいずれかであり，すべて本章で説明している。これらは式が許可されているコンテキスト，特に <プログラム> の最も外側のレベルおよび <本体> の先頭では，全部ではないが，一部で有効である。

プログラムの最も外側のレベルでは，(begin <式または定義 $_1$> ...) は，begin の式および定義からなる列と等価である。同様に，<本体> では，(begin <定義 $_1$> ...) は，列 <定義 $_1$> ... と等価である。マクロは，このような begin の形式に展開することができる。正式な定義については，4.2.3 を参照せよ。

インポート宣言および定義は，グローバル環境で束縛の作成を起こすか，既存のグローバル束縛の値を変更させる。プログラムの初期環境は空なので，少なくとも一つのインポート宣言が初期束縛を導入するために必要とされる。

いくつかの実装形態では，実行中の Scheme システム内に対話的に入力することができるが，プログラムとライブラリは一般的にはファイルに格納される。他のパラダイムも可能である。ファイル内のライブラリを保存する実装は，ライブラリの名前からファイルシステム内の場所へのマッピングをドキュメント化すべきである。

5.2. インポート宣言

インポート宣言は次の形式をとる:

```
(import <インポート集合> ...)
```

インポート宣言は，ライブラリによってエクスポートされた識別子をインポートする方法を提供する。各 <インポート集合> はライブラリからの束縛の集合を指定し，おそらくインポートされた束縛のローカル名を指定する。それは，次のいずれかの形式をとる:

- <ライブラリ名>
- (only <インポート集合> <識別子> ...)
- (except <インポート集合> <識別子> ...)
- (prefix <インポート集合> <識別子>)
- (rename <インポート集合>
  (<識別子 $_1$> <識別子 $_2$>) ...)

最初の形式では，指定されたライブラリのエクスポート句内の識別子のすべてが同じ名前で (あるいは，rename でエクスポートされる場合はエクスポートされた名前で) インポートされる。追加の <インポート集合> 形式は，次のようにこの集合を変更する:

- only は，記載された識別子のみを含む，与えられた <インポート集合> の部分集合を生成する (任意の名前変更後)。任意のリストされた識別子が，オリジナルの集合に見つからない場合はエラーである。
- except は，記載された識別子を除いた，与えられた <インポート集合> の部分集合を生成する (任意の名前変更後)。任意のリストされた識別子が，オリジナルの集合に見つからない場合はエラーである。
- rename は，<識別子 $_1$> のインスタンスを <識別子 $_2$> に置き換えて，与えられた <インポート集合> を変更する。任意のリストされた <識別子 $_1$> が，オリジナルの集合に見つからない場合はエラーである。
- prefix は，指定された <識別子> をそれぞれに接頭辞を付けて，与えられた <インポート集合> のすべての識別子を自動的に名前変更する。

プログラムやライブラリの宣言では，異なる束縛で同じ識別子を複数回インポートすること，インポートされた束縛を再定義したり定義もしくは set!で変異させること，インポートされる前に識別子を参照することはエラーである。しかし，REPL は，これらの動作を許可すべきである。

5.3. 変数定義

変数定義は一つ以上の識別子を束縛し，それらの各々の初期値を指定する。変数定義のもっとも単純な種類は以下の形式の一つをとる。

- (define <変数> <式>)
- (define (<変数> <仮引数部>) <本体>)

  <仮引数部> は，０個以上の変数からなる列か，または１個以上の変数からなる列に対しスペースで区切られたピリオドともう１個の変数を続けたものである (ラムダ式の場合と同様である)。この形式は次と等価である。

  ```
  (define <変数>
    (lambda (<仮引数部>) <本体>)).
  ```

- (define (<変数> . <仮引数>) <本体>)

  <仮引数> は単一の変数である。この形式は次と等価である。

  ```
  (define <変数>
    (lambda <仮引数> <本体>)).
  ```

5.3.1. トップレベル定義

プログラムの最も外側のレベルでは，定義

```
(define <変数> <式>)
```

は，もし <変数> が一つの非構文値に束縛されているならば，本質的に代入式

```
(set! <変数> <式>)
```

と同じ効果をもつ。しかし，もし <変数> が未束縛あるいは構文キーワードならば，この定義は代入を実行する前にまず <変数> を新しい場所に束縛する。しかるに set! を未束縛の変数に対して実行するとエラーになるだろう。

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                    ⟹  6
(define first car)
(first '(1 2))              ⟹  1
```

5.3.2. 内部定義

定義は <本体> (つまり lambda, let, let*, letrec, letrec*, let-values, let*-values, let-syntax, letrec-syntax, parameterize, guard, case-lambda の各式の本体，または適切な形式の定義の本体) の先頭に出現可能である。このような定義を，上述のグローバル定義に対抗して**内部定義** (*internal definition*) という。内部定義で定義される変数は <本体> にローカルである。つまり，<変数> は代入されるのではなく束縛されるのであり，その束縛の領域は <本体> 全体である。たとえば，

```
(let ((x 5))
  (define foo (lambda (y) (bar x y)))
  (define bar (lambda (a b) (+ (* a b) a)))
  (foo (+ x 3)))            ⟹  45
```

内部定義をもつ拡張された <本体> は常にそれと完全に等価な letrec* 式に変換できる。たとえば，上の例の let 式は次と等価である。

```
(let ((x 5))
  (letrec* ((foo (lambda (y) (bar x y)))
            (bar (lambda (a b) (+ (* a b) a))))
    (foo (+ x 3))))
```

等価な letrec 式がそうであるように，<本体> の中の内部定義のそれぞれの <式> は，対応する <変数> の値，または，<本体> 内でそれに続く定義のいずれかの変数への代入も参照もすることなしに評価不能であればエラーである。

同一の <本体> において複数回同じ識別子を定義することはエラーである。

どこに内部定義が出現しようとも，(begin <定義 $_1$> ...) は，その begin の本体をつくる定義からなる列と等価である。

5.3.3. 複数の値の定義

別の定義の種類は define-values で提供され，これは複数の値を返す単一の式から，複数の定義が作られる。これは define が許可されているところで許可される。

(define-values <仮引数部> <式>) 構文

<仮引数部> の集合に複数回変数が出現する場合はエラーである。

意味: <式> が評価され，<仮引数部> は，lambda 式の <仮引数部> が手続き呼び出しの引数に照合しているのと同じ方法で，戻り値に束縛される。

```
(define-values (x y) (integer-sqrt 17))
(list x y)                  ⟹  (4 1)

(let ()
  (define-values (x y) (values 1 2))
  (+ x y))                  ⟹  3
```

5.4. 構文定義

構文定義はこの形式をとる。

(define-syntax <キーワード> <変換子仕様>)

<キーワード> は識別子であり，<変換子仕様> は syntax-rules のインスタンスである。変数定義と同様に，構文定義は最も外側のレベルに現れる，あるいは本体 (body) 内にネストすることが可能である。

define-syntax が最も外側のレベルで出現した場合，グローバル構文環境は指定された変換子に <キーワード> を束縛することによって拡張されるが，<キーワード> のグローバル束縛の以前の展開はいずれも変更されない。それ以外の場合は，内部構文定義であり，それが定義されている本体に対してローカルである。構文キーワードに対応する定義の前にそれを使用することはエラーである。特に，内側の定義に先行した使用は，外側の定義を適用しない。

```
(let ((x 1) (y 2))
  (define-syntax swap!
    (syntax-rules ()
      ((swap! a b)
       (let ((tmp a))
         (set! a b)
         (set! b tmp)))))
  (swap! x y)
  (list x y))               ⟹  (2 1)
```

定義を許可する任意の文脈においてマクロは定義や構文定義へと展開してもよい。しかし，定義自体の，または，内部定義の同じグループに属している先行するいずれかの定義の意味を決定するために，束縛が知っているべき識別子を定義することは，定義についてのエラーである。同様に，内部定義と，それが属する本体の式の境界を決定するために束縛が知っているべき識別子を定義することは，内部定義についてのエラーである。たとえば，下記はそれぞれエラーである。

```
(define define 3)

(begin (define begin list))

(let-syntax
    ((foo (syntax-rules ()
            ((foo (proc args ...) body ...)
             (define proc
               (lambda (args ...)
                 body ...))))))
  (let ((x 3))
    (foo (plus x y) (+ x y))
    (define foo x)
    (plus foo x)))
```

5.5. レコード型定義

レコード型定義 (*record-type definitions*) は，レコード型 (*record types*) と呼ばれる新しいデータ型を導入するために使用される。他の定義のように，最も外側のレベルまたは本体内に現れることができる。レコード型の値はレコード (*records*) と呼ばれ，ゼロ個以上のフィールド (*fields*) の集合体でありそれぞれが単一の場所を保持している。述語，コンストラクタ，フィールドアクセサ，およびミューテータは各レコード型のために定義される。

`(define-record-type` <名前> 構文
`(`<コンストラクタ> <述語> <フィールド> ...`)`

構文: <名前> と <述語> は識別子である。<コンストラクタ> は次の形式である。

`(`<コンストラクタ名> <フィールド名> ...`)`

そして，各 <フィールド> は，次のいずれかの形式である。

`(`<フィールド名> <アクセサ名>`)`

または

`(`<フィールド名> <アクセサ名> <修飾子名>`)`

フィールド名として同じ識別子が複数回出現することはエラーである。また，同じ識別子が一度アクセサまたはミューテータの名前として複数回出現することもエラーである。

`define-record-type` コンストラクトは，生成的である。それぞれの使用は，Scheme の定義済みの型や他のレコード型を含む，すべての既存の型とは異なる新しいレコード型を，それが同じ名前または構造のレコード型であっても作成する。

`define-record-type` のインスタンスは以下の定義と等価である。

- <名前> は，レコード型自体の表現に束縛されている。これは，実行時オブジェクトまたは純粋な構文表現であってもよい。表現は，本報告書で利用されていないが，さらなる言語拡張による使用のためにレコード型を識別するための手段として機能する。
- <コンストラクタ名> は，部分式 (<コンストラクタ名> ...) における <フィールド名> の数と同じ数の引数を取り，<名前> 型の新しいレコードを返す手続きに束縛されている。名前が <コンストラクタ名> と記載されているフィールドは，初期値として対応する引数がある。他のすべてのフィールドの初期値は未規定である。フィールド名が <フィールド名> としてでなく <コンストラクタ> に現れることはエラーである。
- <述語> は，<コンストラクタ名> に束縛された手続きによって返される値が与えられたときに `#t` を返し，それ以外すべての場合 `#f` を返す述語に束縛される。
- 各 <アクセサ名> は，<名前> 型のレコードをとり，対応するフィールドの現在値を返す手続きに束縛されている。アクセサに適切な型のレコードではない値を渡すことはエラーである。
- 各 <修飾子名> は，<名前> 型のレコードと，対応するフィールドの新しい値になる値をとる手続きに束縛されている。返される値は未規定である。修飾子に適切な型のレコードではない第一引数を渡すことはエラーである。

たとえば，次のレコード型の定義

```
(define-record-type <pare>
  (kons x y)
  pare?
  (x kar set-kar!)
  (y kdr))
```

は，`kons` をコンストラクタとして，`kar` と `kdr` をアクセサとして，`set-kar!` を修飾子として，`pare?` を<pare> のインスタンスの述語として定義している。

```
(pare? (kons 1 2))        ⟹ #t
(pare? (cons 1 2))        ⟹ #f
(kar (kons 1 2))          ⟹ 1
(kdr (kons 1 2))          ⟹ 2
(let ((k (kons 1 2)))
  (set-kar! k 3)
  (kar k))                ⟹ 3
```

5.6. ライブラリ

ライブラリは，プログラムの残りの部分を明示的に定義されたインタフェースを備えた Scheme プログラムを再利用可能な部分に組織する方法を提供する。本節ではライブラリの表記および構文を定義する。

5.6.1. ライブラリ構文

ライブラリ定義は以下の形式をとる。

```
(define-library <ライブラリ名>
  <ライブラリ宣言> ...)
```

<ライブラリ名> は，識別子と非負の整数をメンバにもつリストである。それは他のプログラムまたはライブラリからインポートするときに，ライブラリを一意に識別するために使用される。最初の識別子が scheme であるライブラリは，本報告書および将来のバージョンで使用するために予約されている。最初の識別子が srfi であるライブラリは，Scheme のための実装要求 (Scheme Requests for Implementation : SRFI) を実装するライブラリのために予約されている。ライブラリ名の識別子に，エスケープ展開後の任意の文字または制御文字| \ ? * < " : > + [ ] / を含めることは，エラーではないが非推奨である。

<ライブラリ宣言> は以下のいずれかである:

- (export <エクスポート仕様> ...)
- (import <インポート集合> ...)
- (begin <コマンドまたは定義> ...)
- (include <ファイル名 $_1$> <ファイル名 $_2$> ...)
- (include-ci <ファイル名 $_1$> <ファイル名 $_2$> ...)
- (include-library-declarations <ファイル名 $_1$> <ファイル名 $_2$> ...)
- (cond-expand <ce 節 $_1$> <ce 節 $_2$> ...)

export 宣言は，他のライブラリまたはプログラムに見えるようにすることができる識別子のリストを指定する。<エクスポート仕様> は以下のいずれかの形式をとる。

- <識別子>
- (rename <識別子 $_1$> <識別子 $_2$>)

<エクスポート仕様> では，<識別子> はライブラリで定義されているか，ライブラリにインポートされている単一の束縛を名付ける。ここで，エクスポートのための外部名は，ライブラリでの束縛の名前と同じである。rename 仕様は，外部の名前として <識別子 $_2$> を使用して，ライブラリで定義されているか，ライブラリにインポートされており，各 (<識別子 $_1$> <識別子 $_2$>) ペアリングしている，<識別子 $_1$> で名付けられた束縛をエクスポートする。

import 宣言は，他のライブラリによってエクスポートされた識別子をインポートする方法を提供する。それはプログラムまたは REPL で使われるインポート宣言と同じ構文および意味を持つ (5.2 節参照)。

begin, include, および include-ci 宣言は，ライブラリ本体を指定するのに使われる。それらは対応する式型と同じ構文および意味をもつ。この begin の形式は，同じではないが，4.2.3 節で定義された begin の二つの型に似ている。

include-library-declarations 宣言は，ファイルの内容が現在のライブラリ定義に直接指定されることを除いて include に似ている。これはたとえば，複数のライブラリ内の同じ export 宣言を共有するのにライブラリインタフェースの単純な形式として使うことができる。

cond-expand 宣言は，begin で囲まれた式ではなく，継ぎ合わせライブラリ宣言に展開することを除いて，cond-expand 式型と同じ構文および意味をもつ。

ライブラリの可能な実装の一つを以下に示す。すべての cond-expand ライブラリ宣言が展開された後，新しい環境がすべてのインポートされた束縛を含んでいるライブラリのために構築される。すべての begin, include および include-ci ライブラリ宣言からの式は，そのライブラリに発生した順に，その環境で展開される。別の方法として，cond-expand と import 宣言は，インポートされた束縛が各 import 宣言によってそれに追加されるような生育環境で，散在する他の宣言の処理と合わせて左から右へと順に処理されてもよい。

ライブラリがロードされるとき，その式は記述順に実行される。ライブラリの定義がプログラムまたはライブラリ本体の展開された形式で参照される場合は，展開されたプログラムやライブラリ本体が評価される前にロードされなければならない。このルールは推移的に適用する。ライブラリが複数のプログラムやライブラリでインポートされる場合，それは追加時にロードされてもよい。

同様に，ライブラリ (foo) の展開中，ライブラリを展開するのに必要とされる別のライブラリ (bar) から任意の構文キーワードがインポートされる場合，ライブラリ (bar) は展開され，その構文定義は (foo) の展開前に評価されればならない。

ライブラリがロードされる回数に関係なく，ライブラリから束縛をインポートする各プログラムやライブラリはインポート宣言の現れる回数に関係なく，そのライブラリの単一のロードから行わなければならない。すなわち，(import (only (foo) a)) の後に続く (import (only (foo) b)) は，(import (only (foo) a b)) と同じ効果をもたらす。

5.6.2. ライブラリの例

以下の例は，プログラムがライブラリと比較的小さいメインプログラムに分割できる方法を示す [16]。メインプログラムが REPL に入力された場合， base ライブラリをインポートする必要はない。

```
(define-library (example grid)
  (export make rows cols ref each
          (rename put! set!))
  (import (scheme base))
  (begin
```

```
    ;; NxM のグリッドを作成する。
    (define (make n m)
      (let ((grid (make-vector n)))
        (do ((i 0 (+ i 1)))
            ((= i n) grid)
          (let ((v (make-vector m #false)))
            (vector-set! grid i v)))))
    (define (rows grid)
      (vector-length grid))
    (define (cols grid)
      (vector-length (vector-ref grid 0)))
    ;; 範囲外ならば #false を返す。
    (define (ref grid n m)
      (and (< -1 n (rows grid))
           (< -1 m (cols grid))
           (vector-ref (vector-ref grid n) m)))
    (define (put! grid n m v)
      (vector-set! (vector-ref grid n) m v))
    (define (each grid proc)
      (do ((j 0 (+ j 1)))
          ((= j (rows grid)))
        (do ((k 0 (+ k 1)))
            ((= k (cols grid)))
          (proc j k (ref grid j k)))))))

(define-library (example life)
  (export life)
  (import (except (scheme base) set!)
          (scheme write)
          (example grid))
  (begin
    (define (life-count grid i j)
      (define (count i j)
        (if (ref grid i j) 1 0))
      (+ (count (- i 1) (- j 1))
         (count (- i 1) j)
         (count (- i 1) (+ j 1))
         (count i (- j 1))
         (count i (+ j 1))
         (count (+ i 1) (- j 1))
         (count (+ i 1) j)
         (count (+ i 1) (+ j 1))))
    (define (life-alive? grid i j)
      (case (life-count grid i j)
        ((3) #true)
        ((2) (ref grid i j))
        (else #false)))
    (define (life-print grid)
      (display "\x1B;[1H\x1B;[J")  ; vt100 をクリア
      (each grid
       (lambda (i j v)
         (display (if v "*" " "))
         (when (= j (- (cols grid) 1))
           (newline)))))
    (define (life grid iterations)
      (do ((i 0 (+ i 1))
           (grid0 grid grid1)
           (grid1 (make (rows grid) (cols grid))
                  grid0))
          ((= i iterations))
        (each grid0
         (lambda (j k v)
           (let ((a (life-alive? grid0 j k)))
             (set! grid1 j k a))))
        (life-print grid1)))))

;; メインプログラム。
(import (scheme base)
        (only (example life) life)
        (rename (prefix (example grid) grid-)
                (grid-make make-grid)))

;; glider でグリッドを初期化する。
(define grid (make-grid 24 24))
(grid-set! grid 1 1 #true)
(grid-set! grid 2 2 #true)
(grid-set! grid 3 0 #true)
(grid-set! grid 3 1 #true)
(grid-set! grid 3 2 #true)

;; 80 回繰り返す。
(life grid 80)
```

5.7. REPL

実装は，インポート宣言，式および定義が，一度に入力および評価できる *REPL* (read-eval-print ループ) と呼ばれる対話型セッションを提供してもよい。利便性と使いやすさのために， REPL のグローバル Scheme 環境は空であってはならないが，少なくとも， base ライブラリによって提供される束縛から始めなければならない。このライブラリは，Scheme のコア構文と，データを操作する一般的に有用な手続きを含んでいる。たとえば，変数 `abs` は数の絶対値を計算する，一引数の手続きに束縛されており，変数 `+` は合計を計算する手続きに束縛されている。(`scheme base`) 束縛の完全なリストは，付録 A に記載されている。

実装は，すべての可能な変数が場所に束縛されていて，そのほとんどが未規定の値を含んでいるかのように振舞うような初期 REPL 環境を提供してもよい。このような実装におけるトップレベルの REPL 定義は，識別子が構文キーワードとして定義されない限り，真に代入と等価である。

実装は， REPL が入力をファイルから読み取る操作モードを提供してもよい。このようなファイルは，先頭以外の場所にインポート宣言を含めることができるので，一般的には，プログラムと同じではない。

6. 標準手続き

本章は Scheme の組込み手続きを記述する。

手続き `force`, `promise?`, および `make-promise` は式型 `delay` および `delay-force` と密接に関連付けられ，4.2.5 節でともに記載されている。同様に，手続き `make-parameter`

は式型 parameterize と密接に関連付けられ，4.2.6 節でともに記載されている。

プログラムはグローバル変数定義を使って任意の変数を束縛できる。そして後からその束縛を代入によって変更してよい (4.1.6 節参照)。これらの演算は本報告書で定義されている，またはライブラリからインポートされた手続きの振舞いを改変しない (5.6 節参照)。定義によって導入されていないグローバルの束縛を変更することは，本章で定義された手続きの振舞いに対して未規定の効果をもつ。

手続きが新たに割り当てられた (*newly allocated*) オブジェクトを返すように言われたとき，それはオブジェクトの場所が新鮮であることを意味する。

6.1. 等価性述語

述語 (*predicate*) とは，常にブーリアン値 (#t または #f) を返す手続きである。**等価性述語** (*equivalence predicate*) とは，計算機上で数学的な等価関係に相当するものである; 対称律と反射律と推移律が成り立つ。本節で記述する等価性述語のうち，eq? が最も細かく (つまり最も識別が厳しく)，equal? が最も粗く，eqv? は eq? よりもわずかに甘く識別する。

(eqv? obj_1 obj_2) 手続き

eqv? 手続きはある有用な等価関係をオブジェクトに対して定義する。簡単にいえば，もしも obj_1 と obj_2 が通常同じオブジェクトだと見なすならば，#t を返す。この関係にはいささか解釈の余地があるが，下記の部分的な eqv? の規定は Scheme のすべての実装に対して成り立つ。

次が成り立つならば eqv? 手続きは #t を返す:

- obj_1 と obj_2 がともに #t またはともに #f である。
- obj_1 と obj_2 がともにシンボルであって，かつ symbol=? 手続き (6.5 節) に従って同じシンボルである。
- obj_1 と obj_2 がともに正確数であって，かつ数値的に (= の意味において) 等しい。
- obj_1 と obj_2 がともに数値的に (= の意味において) 等しくなるような不正確数であって, かつ，Scheme の標準算術演算手続きの有限組成物として定義可能で NaN 値を生じない提供ができる他の手続きに引数として渡されたときに，(eqv? の意味において) 同じ結果が得られる。
- obj_1 と obj_2 がともに文字であって，かつ char=? 手続き (6.6 節) で同じ文字だと判定される。
- obj_1 と obj_2 がともに空リストである。
- obj_1 と obj_2 がともにペア，ベクタ，バイトベクタ，レコード，または文字列であって，記憶領域の中の同じ場所を表す (3.4 節)。
- obj_1 と obj_2 がともに手続きであって，場所タグが等しい (4.1.4 節)。

次が成り立つならば eqv? 手続きは #f を返す:

- obj_1 と obj_2 の型が異なる (3.2 節)。
- obj_1 と obj_2 の一方が #t で他方が #f である。
- obj_1 と obj_2 がともにシンボルだが symbol=? 手続き (6.5 節) に従って同じシンボルではない。
- obj_1 と obj_2 の一方が正確数で他方が不正確数である。
- obj_1 と obj_2 がともに正確数であって，かつ数値的に (= の意味において) 等しくない。
- obj_1 と obj_2 がともに〜であるような不正確数であって，それらが (=の意味において) 数値的に等しくない，または Scheme の標準算術演算手続きの有限組成物として定義可能で NaN 値を生じない提供ができる他の手続きに引数として渡されたときに，それらが (eqv?の意味において) 同じ結果が得られない不正確数である。例外として，obj_1 と obj_2 がともに NaN である場合は，eqv? の振舞いは未規定である。
- obj_1 と obj_2 がともに文字であって，char=? 手続きが #f を返す。
- obj_1 と obj_2 の一方が空リストだが他方がそうではない。
- obj_1 と obj_2 がともにペア，ベクタ，バイトベクタ，レコード，または文字列であって，別々の場所を表している。
- obj_1 と obj_2 がともに手続きであって，なんらかの引数に対して異なった振舞いをする (異なった値を返す，または異なった副作用をもつ) だろう。

```
(eqv? 'a 'a)                    ⟹  #t
(eqv? 'a 'b)                    ⟹  #f
(eqv? 2 2)                      ⟹  #t
(eqv? 2 2.0)                    ⟹  #f
(eqv? '() '())                  ⟹  #t
(eqv? 100000000 100000000)      ⟹  #t
(eqv? 0.0 +nan.0)               ⟹  #f
(eqv? (cons 1 2) (cons 1 2))    ⟹  #f
(eqv? (lambda () 1)
      (lambda () 2))            ⟹  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                   ⟹  #t
(eqv? #f 'nil)                  ⟹  #f
```

下記の例は，上記の規則が eqv? の振舞いを完全には規定していないという実例を示している。このような場合について言えることは，eqv? の返す値はブーリアンでなければならないということだけである。

```
(eqv? "" "")                  ⟹  未規定
(eqv? '#() '#())              ⟹  未規定
(eqv? (lambda (x) x)
      (lambda (x) x))         ⟹  未規定
(eqv? (lambda (x) x)
      (lambda (y) y))         ⟹  未規定
(eqv? 1.0e0 1.0f0)            ⟹  未規定
(eqv? +nan.0 +nan.0)          ⟹  未規定
```

(eqv? 0.0 -0.0) は，負のゼロが区別されている場合 #f を返し，負のゼロが区別されない場合は， #t を返すことに注意せよ。

次の例は，ローカルな状態をもった手続きに対する eqv? の使用を示している。gen-counter 手続きは毎回別々の手続きを返しているはずである。なぜなら，各手続きはそれ自身の内部カウンタをもっているからである。一方，gen-loser 手続きは毎回動作上等価な手続きを返している。なぜなら，そのローカル状態は，手続きの値または副作用に影響しないからである。しかし， eqv? はこの等価性を検出できるかも知れないし，そうでないかも知れない。

```
(define gen-counter
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) n))))
(let ((g (gen-counter)))
  (eqv? g g))                 ⟹  #t
(eqv? (gen-counter) (gen-counter))
                              ⟹  #f
(define gen-loser
  (lambda ()
    (let ((n 0))
      (lambda () (set! n (+ n 1)) 27))))
(let ((g (gen-loser)))
  (eqv? g g))                 ⟹  #t
(eqv? (gen-loser) (gen-loser))
                              ⟹  未規定

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                              ⟹  未規定

(letrec ((f (lambda () (if (eqv? f g) 'f 'both)))
         (g (lambda () (if (eqv? f g) 'g 'both))))
  (eqv? f g))
                              ⟹  #f
```

定数オブジェクト (リテラル式が返すオブジェクト) を改変することはエラーであるため，適切なところで実装が定数どうしの構造を共有化することがある。したがって定数についての eqv? の値はときどき実装依存である。

```
(eqv? '(a) '(a))              ⟹  未規定
(eqv? "a" "a")                ⟹  未規定
(eqv? '(b) (cdr '(a b)))      ⟹  未規定
(let ((x '(a)))
  (eqv? x x))                 ⟹  #t
```

eqv? の上記の定義は，手続きとリテラルの取扱いにおいて実装に自由度を許可している。つまり，実装は，二つの手続きないし二つのリテラルが互いに等価であることを検出することも検出しそこなってもよく，両者の表現に使用するポインタまたはビットパターンが同一であることを以て等価なオブジェクトであることと同一視することにしてもしなくてもよい。

注: もし正確整数が IEEE バイナリ浮動小数点数として表現されるならば，同じサイズの不正確数をビット単位で等しいかどうか単純比較する eqv? の実装は，上記の定義によって正しい。

(eq? obj_1 obj_2) 手続き

eq? 手続きは，いくつかの場合において eqv? よりも細かく差異を見分けられることを除けば，eqv? と同様である。eqv? がそう (#f を返す) であろうとき，それは常に #f を返す必要があるが，場合によっては eqv? が #t を返すときに #f を返してもよい。

シンボル，ブーリアン，空リスト，ペア，レコード，および空でない文字列，ベクタ，バイトベクタに対して eq? と eqv? は同じ振舞いをすることが保証されている。手続きに対して，引数のロケーションタグが同じならば，eq? は真を返さなければならない。数と文字に対して， eq? の振舞いは実装依存だが，常に真か偽を返す。空文字列，空ベクタ，空バイトベクタに対しても， eq? は eqv? と違う振舞いをしてよい。

```
(eq? 'a 'a)                   ⟹  #t
(eq? '(a) '(a))               ⟹  未規定
(eq? (list 'a) (list 'a))     ⟹  #f
(eq? "a" "a")                 ⟹  未規定
(eq? "" "")                   ⟹  未規定
(eq? '() '())                 ⟹  #t
(eq? 2 2)                     ⟹  未規定
(eq? #\A #\A)                 ⟹  未規定
(eq? car car)                 ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                  ⟹  未規定
(let ((x '(a)))
  (eq? x x))                  ⟹  #t
(let ((x '#()))
  (eq? x x))                  ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                  ⟹  #t
```

根拠: 普通，eq? は eqv? よりもずっと効率的に—たとえば，なにか複雑な演算ではなく単純なポインタ比較として—実装することが可能だろう。一つの理由として，eq? をポインタ比較として実装すると常に定数時間で終了するだろうが，二つの数の eqv? を定数時間で計算することは常に可能ではないからである。

(equal? obj_1 obj_2) 手続き

equal? 手続きはペア，ベクタ，文字列およびバイトベクタに適用されるとき，再帰的にそれらを比較し，その引数の木構造への (場合によっては無限の) 展開が，順序木として (equal?の意味で) 等しい場合には #t を返し，そうでない場合には #f を返す。ブーリアン，シンボル，数，文字，ポート，手続き，および空リストに適用された場合には，eqv?

と同じように返す。2 つのオブジェクトが eqv? ならば，それらは同様に equal? でなければならない。それ以外の場合は，equal? は #t または #f のどちらかを返す。

その引数が環状データ構造であっても， equal? は常に停止しなければならない。

```
(equal? 'a 'a)                ⟹  #t
(equal? '(a) '(a))            ⟹  #t
(equal? '(a (b) c)
        '(a (b) c))           ⟹  #t
(equal? "abc" "abc")          ⟹  #t
(equal? 2 2)                  ⟹  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))   ⟹  #t
(equal? '#1=(a b . #1#)
        '#2=(a b a b . #2#))  ⟹  #t
(equal? (lambda (x) x)
        (lambda (y) y))       ⟹  未規定
```

注: おおざっぱな原則として，同じように印字されるオブジェクトどうしは一般に equal? である。

6.2. 数

数学上の数，それをモデル化しようとしている Scheme の数，Scheme の数の実装に使われる機械表現，および数を書くために使われる表記，この四者を区別することが大切である。本報告書は *number*, *complex*, *real*, *rational*, および *integer* という型を使って，数学上の数と Scheme の数の両方を言い表す。

6.2.1. 数値型

数学的には，数は，各レベルがそれぞれその上位レベルの部分集合であるような部分型の塔にまとめられる:

number (数)
complex number (複素数)
real number (実数)
rational number (有理数)
integer (整数)

たとえば， 3 は整数である。したがって 3 は有理数でも，実数でも，複素数でもある。同じことが 3 をモデル化した Scheme の数についても成り立つ。Scheme の数に対して，これらの型は述語 number?, complex?, real?, rational?, および integer? によって定義される。

数の型とその計算機内部での表現との間に単純な関係はない。たいていの Scheme の実装は少なくとも二種類の 3 の表現を提供するだろうが，これら様々な表現はみな同じ整数を表すというわけである。

Scheme の数値演算は数を—可能な限りその表現から独立した—抽象データとして取り扱う。たとえ Scheme の実装が数の複数の内部表現を使うとしても，このことは単純なプログラムを書くカジュアルなプログラマの目に明らかであるべきではない。

6.2.2. 正確性

正確に表現された数と，そうとは限らない数とを区別することは有用である。たとえば，データ構造への添字は，記号代数系における多項式係数と同様，正確に知られなければならない。他方，計測の結果というものは本質的に不正確である。また，無理数は有理数によって近似され得るが，そのときは不正確な近似になる。正確数が要求される箇所での不正確数の使用を捕捉するため，Scheme は正確数を不正確数から明示的に区別する。この区別は型の次元に対して直交的である。

もしも Scheme の数が正確定数として書かれたか，または正確数から正確演算だけを使って導出されたならば，その数は *exact* (正確) である。もしも数が不正確定数として書かれたか，不正確な材料を使って導出されたか，あるいは不正確演算を使って導出されたならば，その数は *inexact* (不正確) である。このように不正確性は，数のもつ伝搬性の性質である。具体的には，正確複素数 (*exact complex number*) には正確実部と正確虚部があり，他のすべての複素数は不正確複素数 (*inexact complex numbers*) である。

二つの実装がある計算に対して正確な結果を算出するとき，そこで不正確な中間結果とかかわらなかったならば，二つの最終結果は数学的に等しいだろう。これは一般に，不正確数とかかわる計算には成り立たない。なぜなら浮動小数点演算のような近似的方法が使われ得るからである。しかし結果を数学上の理想的な結果に現実的な範囲で近付けることは各実装の義務である。

+ のような有理演算は，正確な引数が与えられたときは，常に正確な結果を算出するべきである。もし演算が正確な結果を算出できないならば，実装制限の違反を報告してもよいし，黙ってその結果を不正確値に変換してもよい。しかし，(/ 3 4) は，数学的に正しくない値 0 を返してはいけない。6.2.3 節を見よ。

exact を除いて，本節で記述される演算は一般に，なんであれ不正確な引数が与えられたときは，不正確な結果を返さなければならない。ただし，もしも引数の不正確性が結果の値に影響しないことが証明できるならば，その演算は正確な結果を返してよい。たとえば，正確なゼロとの乗算は，たとえ他方の引数が不正確であっても，正確なゼロを結果として算出してよい。

具体的には，式 (* 0 +inf.0) は 0 または +nan.0 を返す，または不正確数がサポートされていないことを報告する，または非有理実数がサポートされていないことを報告する，または実装固有の方法で黙ってもしくは騒々しく失敗する，のどれでもよい。

6.2.3. 実装制限

Scheme の実装は，6.2.1 節で与えられた部分型の塔全体を実装することは必須ではないが，実装の目的と Scheme 言語の精神にともに合致した一貫性のあるサブセットを実装しなければならない。たとえば，すべての数が実数，または非実数が常に不正確，または正確数が常に整数，であるような実装でも依然，かなり有用であり得る。

実装はまた，本節の要求を満たす限り，どの型であれ，それに対してある限定された値域の数をサポートするだけでよい。どの型であれ，その正確数に対してサポートする値域が，その不正確数に対してサポートする値域と異なっていてもよい。たとえば， IEEE バイナリ倍精度浮動小数点数を使ってすべての不正確実数を表現するような実装もまた，不正確実数の値域を (したがって不正確整数と不正確有理数の値域をも) IEEE バイナリ倍精度形式のダイナミックレンジに限定する一方で，正確整数と正確有理数に対しては事実上無制限の値域をサポートしてもよい。なお，このような実装では，表現可能な不正確整数および不正確有理数の数の間隔は，この値域の限界に接近するにつれて非常に大きくなりやすい。

Scheme の実装は，リストやベクタ，バイトベクタや文字列の添字として許可されているか，またはそれらの一つの長さの計算から結果として得られ得るような数の値域にわたって正確整数をサポートしなければならない。`length`，`vector-length`，`bytevector-length`，および `string-length` の各手続きは正確整数を返さなければならない。正確整数以外のものを添字として使うことはエラーである。なお，添字値域内の整数定数が，正確整数の構文で表現されているならば，たとえいかなる実装制限がこの値域外に適用されていようとも，その定数は実際に正確整数として読み込まれなければならない。最後に，下記に挙げる手続きは，もしも引数がすべて正確整数であって，かつ数学的に期待される結果が実装において正確整数として表現可能ならば，常に正確整数の結果を返すものとする:

```
-                       *
+                       abs
ceiling                 denominator
exact-integer-sqrt      expt
floor                   floor/
floor-quotient          floor-remainder
gcd                     lcm
max                     min
modulo                  numerator
quotient                rationalize
remainder               round
square                  truncate
truncate/               truncate-quotient
truncate-remainder
```

実装が事実上無制限のサイズと精度をもった正確整数と正確有理数をサポートすることと，上記の手続きと `/` 手続きを正確な引数が与えられたときは常に正確な結果を返すように実装することは，必須ではないが，推奨されている。もしもこれらの手続きの一つが，正確引数が与えられたときに正確な結果をかなえられないならば，実装制限の違反を報告してもよいし，黙ってその結果を不正確数に変換してもよい。このような変換は後にエラーが発生することがある。にもかかわらず，正確有理数を提供していない実装では，実装上の制約を報告するのではなく，不正確有理数を返さなければならない。

実装は浮動小数点やその他の近似的な表現戦略を不正確数を表すために使ってよい。本報告書は，必須ではないが，浮動小数点表現を使う実装が IEEE754 標準に準拠すること，そして他の表現を使った実装がこれらの浮動小数点規格 [17] を使って達成可能な精度と互角以上であることを，推奨する。特に，そのような実装は，特に無限大や NaN に関して，IEEE754-2008 にある超越関数の記述に従わなければならない。

Scheme は数に対して様々な表記を許可しているが，個々の実装はその一部をサポートするだけでよい。たとえば，すべての数が実数であるような実装が，複素数に対する直交座標系および極座標系の表記をサポートする必要はない。実装が，正確数として表現できないような正確数値定数に遭遇した場合，実装は実装制限の違反を報告してもよいし，黙ってその定数を不正確数で表現してもよい。

6.2.4. 実装の拡張

実装は，異なる精度で浮動小数点数の一つ以上の表現を提供してもよい。そのような実装では，不正確な結果は少なくとも，その演算における不正確引数のいずれかを表現するのに使われているものと同程度の精度で表現されなければならない。それは正確な引数に適用された場合に正確な答えを生成する `sqrt` のように，潜在的に不正確な演算のために望ましいが，正確数が不正確な結果を生成するように演算されるならば，利用可能な最も正確な表現を使われなければならない。たとえば，(`sqrt 4`) の値は 2 であるべきだが，単精度および倍精度浮動小数点数の両方の精度を提供する実装では，後者であってもよいが，前者であってはならない。

実装で表現するには大きすぎる絶対値や仮数を持つ不正確数オブジェクトの使用を避けるのは，プログラマの責任である。

加えて，実装は 正の無限大, 負の無限大, NaN, 負のゼロと呼ばれる特別な数を区別することができる。

正の無限大は，すべての有理数で表される数よりも大きな不定値を表す不正確実数 (だが有理数ではない) とみなされている。負の無限大は，すべての有理数で表される数字りも小さい不定値を表す不正確実数 (だが有理数ではない) とみなされている。

任意の有限の実数値に無限大を加算または乗算すると，適切に符号付き無限大になる。しかし，正の無限大と負の無限大の和は NaN である。正の無限大はゼロの逆数であり，負の無限大は負のゼロの逆数である。超越関数の振舞いは IEEE 754 に準拠して，無限大に敏感である。

NaN は，正または負の無限大を含み，かつ正の無限大より大きいか負の無限大よりも小さいかもしれない任意の実数値を表す不定の不正確実数 (だが有理数ではない) とみなされている。非実数をサポートしていない実装では，(`sqrt -1.0`) および (`asin 2.0`) のような非実数値を表すために NaN を使ってもよい。

NaN は常に NaN を含む任意の数との比較でに偽を返す。NaN が任意の有理数に置き換えられた場合の結果が同じであると実装が証明できない限り，一方のオペランドが NaN の算術演算は NaN を返す。両方のゼロが正確数でない限り，ゼロをゼロで割った結果は NaN である。

負のゼロは，不正確実数値で `-0.0` と書かれ，(`eqv?` の意味において) `0.0` とは異なる。Scheme の実装は，負のゼロを区別する必要はない。しかし，それを区別する場合は，超越関数の振舞いは， IEEE 754 に基づいて区別に敏感である。特に，複素数と負のゼロの両方を実装する Scheme では，複素対数関数の分岐截線はつまり，`(imag-part (log -1.0-0.0i))` は π ではなく $-\pi$ である。

さらに，負のゼロの否定は通常のゼロとなり，逆もまた同様である。これは，2 つ以上の負のゼロの和は負であり，負のゼロから (正の) ゼロを減算した結果は，同様に負であることを意味する。ただし，数値の比較は負のゼロをゼロと等しく扱う。

複素数の実部と虚部の両方が無限大，NaN，または負のゼロになりうることに注意せよ。

6.2.5. 数値定数の構文

数に対する表記表現の構文は 7.1.1 節で形式的に記述される。英字の大文字と小文字は数値定数において区別されないことに注意せよ。

数は，基数接頭辞 (radix prefix) の使用により二進，八進，十進，または十六進で書くことができる。基数接頭辞は `#b` (binary, 二進), `#o` (octal, 八進), `#d` (decimal, 十進), および `#x` (hexadecimal, 十六進) である。基数接頭辞がなければ，十進数での表現と仮定される。

数値定数は，接頭辞により正確か不正確かを指定することができる。その接頭辞は正確 (exact) が `#e`，不正確 (inexact) が `#i` である。基数接頭辞を使うとき，正確性接頭辞 (exactness prefix) はその前にも後ろにも現れることができる。もし数の表記表現に正確性接頭辞がなければ，その定数が小数点または指数部が含まれていれば不正確数である。そうでなければ正確数である。

様々な精度の不正確数をもったシステムでは，定数の精度を指定することが有用であり得る。この目的のため，実装はその不正確表現の望ましい精度を示す指数部マーカとともに書かれた数値定数を受け入れてよい。英字 `s`, `f`, `d`, および `l` はそれぞれ *short*, *single*, *double*, および *long* 精度の使用を指定するのに，`e` の代わりに使用することができる。デフォルト精度は少なくとも *double* と同等の精度だが，実装はこのデフォルトを利用者が設定できるようにしてもよい。

```
3.14159265358979F0
        single に丸めて — 3.141593
0.6L0
        long に拡張して — .600000000000000
```

正の無限大，負の無限大，および NaN の数はそれぞれ `+inf.0`, `-inf.0` および `+nan.0` と書かれる。NaN はまた，`-nan.0` とも書かれる。書かれる表現の符号の使用は，もしあっても，必ずしも NaN 値の基盤となる符号を反映していない。実装はこれらの数をサポートする必要はないが，実装がサポートしている場合，それらは，IEEE 754 に一般的に準拠しなければならない。しかし，実装は NaN 値の通知をサポートする必要はなければ，別の NaN 値を区別するための方法を提供する必要もない。

非実複素数のために 2 つの表記が提供される。直交座標系表記 (*rectangular notation*) *a*`+`*b*`i` ここで a は実部， b は虚部である。および極座標系表記 (*polar notation*) r`@`θ ここで r は絶対値，θ はラジアン単位の位相 (角度) である。これらは次式のような関係を持っている。$a+b\mathrm{i} = r\cos\theta + (r\sin\theta)\mathrm{i}$. a, b, r, および θ はすべて実数である。

6.2.6. 数値演算

読者は 1.3.3 節を参照して，数値ルーチンの引数の型についての制限を指定するために使われた命名規約のまとめを読まれたい。本節における用例は，正確表記を使って書かれた数値定数がどれも実際に正確数として表現されるということを仮定している。例によっては，不正確表記を使って書かれた数値定数のいくつかが精度を失うことなく表現可能であるということも仮定しており，その不正確定数には IEEE バイナリ倍精度を使って不正確数を表現する実装においてこれが成り立ちそうなものが選ばれている。

`(number?` *obj*`)` 手続き
`(complex?` *obj*`)` 手続き
`(real?` *obj*`)` 手続き
`(rational?` *obj*`)` 手続き
`(integer?` *obj*`)` 手続き

これらの数値型述語は，数に限らず，どの種類の引数にも適用できる。これらは，もしオブジェクトが述語の名前の型ならば `#t` を返し，そうでなければ `#f` を返す。一般に，もしある型述語がある数について真ならば，すべての上位の型述語もまたその数について真である。したがって，もしある型述語がある数について偽ならば，すべての下位の型述語もまたその数について偽である。

z を複素数とするとき，`(real?` z`)` が真である必要十分条件は `(zero? (imag-part` z`))` が真であることである。x を不正確実数とするとき，`(integer?` x`)` が真である必要十分条件は `(=` x `(round` x`))` である。

数 `+inf.0`, `-inf.0`, および `+nan.0` は実数だが有理数ではない。

```
(complex? 3+4i)            ⟹  #t
(complex? 3)               ⟹  #t
(real? 3)                  ⟹  #t
(real? -2.5+0i)            ⟹  #t
(real? -2.5+0.0i)          ⟹  #f
(real? #e1e10)             ⟹  #t
(real? +inf.0)             ⟹  #t
(real? +nan.0)             ⟹  #t
(rational? -inf.0)         ⟹  #f
(rational? 3.5)            ⟹  #t
(rational? 6/10)           ⟹  #t
(rational? 6/3)            ⟹  #t
(integer? 3+0i)            ⟹  #t
(integer? 3.0)             ⟹  #t
(integer? 8/4)             ⟹  #t
```

注: 不正確数に対するこれらの型述語の振舞いは信頼できない。なぜなら，いかなる誤差が結果に影響するかもしれないからである。

注: 多くの実装では `complex?` 手続きは `number?` と同一になるだろう。しかし，普通でない実装ではいくつかの無理数を正確に表現してもよいし，あるいは数の体系を拡張してある種の非複素数をサポートしてもよい。

`(exact?` z`)` 手続き
`(inexact?` z`)` 手続き

これらの数値述語は数量の正確性についてのテストを定めている。どの Scheme 数についても，これらの述語のうちの正確に一つが真である。

```
(exact? 3.0)          ⟹  #f
(exact? #e3.0)        ⟹  #t
(inexact? 3.)         ⟹  #t
```

`(exact-integer?` z`)` 手続き

z が正確かつ整数でもある場合は `#t` を返す。そうでない場合は `#f` を返す。

```
(exact-integer? 32)       ⟹ #t
(exact-integer? 32.0)     ⟹ #f
(exact-integer? 32/5)     ⟹ #f
```

`(finite?` z`)` inexact ライブラリ手続き

`finite?` 手続きは， `+inf.0`, `-inf.0`, および `+nan.0` を除くすべての実数と，実部と虚部がどちらも有限である虚数で `#t` を返す。そうでない場合， `#f` を返す。

```
(finite? 3)               ⟹  #t
(finite? +inf.0)          ⟹  #f
(finite? 3.0+inf.0i)      ⟹  #f
```

`(infinite?` z`)` inexact ライブラリ手続き

`infinite?` 手続きは，実数 `+inf.0` および `-inf.0` と，実部と虚部の一方または両方がで無限である虚数で `#t` を返す。そうでない場合， `#f` を返す。

```
(infinite? 3)             ⟹  #f
(infinite? +inf.0)        ⟹  #t
(infinite? +nan.0)        ⟹  #f
(infinite? 3.0+inf.0i)    ⟹  #t
```

`(nan?` z`)` inexact ライブラリ手続き

`nan?` 手続きは，`+nan.0` と，実部と虚部の一方または両方が `+nan.0` である虚数で `#t` を返す。そうでない場合， `#f` を返す。

```
(nan? +nan.0)             ⟹  #t
(nan? 32)                 ⟹  #f
(nan? +nan.0+5.0i)        ⟹  #t
(nan? 1+2i)               ⟹  #f
```

`(=` z_1 z_2 z_3 `...)` 手続き
`(<` x_1 x_2 x_3 `...)` 手続き
`(>` x_1 x_2 x_3 `...)` 手続き
`(<=` x_1 x_2 x_3 `...)` 手続き
`(>=` x_1 x_2 x_3 `...)` 手続き

これらの手続きは引数が (それぞれ) 等しい，狭義単調増加，狭義単調減少，広義単調増加，または広義単調減少ならば `#t` を返し，またそれ以外の場合は `#f` を返す。

すべての述語は，いずれかの引数が `+nan.0` であれば `#f` を返す。それらは不正確のゼロと不正確の負のゼロを区別しない。

これらの述語は推移的であることが必須である。

注: 任意の引数が不正確数である場合にすべての引数を不正確数に変換する実装アプローチは，推移的ではない。たとえば，`big` を `(expt 2 1000)` とし， `big` は正確数であり，その不正確数が 64 ビット IEEE バイナリ浮動小数点数で表されていると仮定する。このとき，`(= (- big 1) (inexact big))` および `(= (inexact big) (+ big 1))` は，大きな整数の IEEE 表現の制限により，このアプローチでは両方とも真になるだろうが，`(= (- big 1) (+ big 1))` は偽である。無限大の扱いで特別な注意が必要ではあるが，不正確な値を正確数にする (`=` の意味において) 等しい変換は，この問題を回避する。

注: 不正確数をこれらの述語を使って比較することはエラーではないが，その結果は信頼できない。なぜなら小さな誤差が結果に影響しうるからである。とりわけ `=` と `zero?` が影響されやすい。疑わしいときは，数値解析の専門家に相談されたい。

`(zero?` z`)` 手続き
`(positive?` x`)` 手続き
`(negative?` x`)` 手続き
`(odd?` n`)` 手続き
`(even?` n`)` 手続き

これらの数値述語は特定の性質について数をテストし，`#t` または `#f` を返す。上記の注を見よ。

`(max` x_1 x_2 `...)` 手続き
`(min` x_1 x_2 `...)` 手続き

これらの述語は引数のうちの最大値 (maximum) または最小値 (minimum) を返す。

```
(max 3 4)                 ⟹  4    ; 正確数
(max 3.9 4)               ⟹  4.0  ; 不正確数
```

注: もし引数のどれかが不正確数ならば，結果も不正確数になる (ただし，その誤差が結果に影響するほど大きくないことを手続きが証明できるならば，その限りではないが，そのようなことは普通でない実装においてのみ可能である)。もし `min` または `max` が正確数と不正確数を比較するために使われ，かつ精度を失うことなく不正確数としてその結果である数値を表現できないならば，そのとき手続きは実装制限の違反を報告してよい。

`(+` z_1 `...)` 手続き
`(*` z_1 `...)` 手続き

これらの手続きは引数の和または積を返す。

```
(+ 3 4)                    ⟹  7
(+ 3)                      ⟹  3
(+)                        ⟹  0
(* 4)                      ⟹  4
(*)                        ⟹  1
```

(- z) 手続き
(- z_1 z_2 ...) 手続き
(/ z) 手続き
(/ z_1 z_2 ...) 手続き

2個以上の引数に対し，これらの手続きは引数の—左結合での—差または商を返す。1引数に対しては，引数の加法または乗法での逆元を返す。

/ の1番目以外の任意の引数が正確なゼロであればエラーである。第1引数が正確なゼロの場合，実装は他の引数うちの1つが NaN でない限り，正確なゼロを返してもよい。

```
(- 3 4)                    ⟹  -1
(- 3 4 5)                  ⟹  -6
(- 3)                      ⟹  -3
(/ 3 4 5)                  ⟹  3/20
(/ 3)                      ⟹  1/3
```

(abs x) 手続き

abs 手続きは引数の絶対値 (absolute value) を返す。

```
(abs -7)                   ⟹  7
```

(floor/ n_1 n_2) 手続き
(floor-quotient n_1 n_2) 手続き
(floor-remainder n_1 n_2) 手続き
(truncate/ n_1 n_2) 手続き
(truncate-quotient n_1 n_2) 手続き
(truncate-remainder n_1 n_2) 手続き

これらの手続きは数論的な (整数の) 除算を実装する。n_2 がゼロの場合はエラーである。/ で終わる手続きは，2つの整数を返す。他の手続きは1つの整数を返す。すべての手続きは，次のような商 n_q ならびに剰余 n_r を計算する。$n_1 = n_2 n_q + n_r$ 除算演算子のそれぞれについて，以下のように定義された3つの手続きがある:

```
(<演算子>/ n1 n2)          ⟹  nq nr
(<演算子>-quotient n1 n2)  ⟹  nq
(<演算子>-remainder n1 n2) ⟹  nr
```

剰余 n_r は n_q: $n_r = n_1 - n_2 n_q$ であるような整数の選択によって決まる。演算子の各セットは，n_q の別の選択を使用している:

floor $n_q = \lfloor n_1/n_2 \rfloor$
truncate $n_q = \text{truncate}(n_1/n_2)$

任意の演算子および整数 n_1 と整数 n_2 ($\neq 0$) に対して

```
(= n1 (+ (* n2 (<演算子>-quotient n1 n2))
         (<演算子>-remainder n1 n2)))
                           ⟹  #t
```

ただし，この計算に関係したすべての数が正確数であるとする。

例:

```
(floor/ 5 2)               ⟹  2 1
(floor/ -5 2)              ⟹  -3 1
(floor/ 5 -2)              ⟹  -3 -1
(floor/ -5 -2)             ⟹  2 -1
(truncate/ 5 2)            ⟹  2 1
(truncate/ -5 2)           ⟹  -2 -1
(truncate/ 5 -2)           ⟹  -2 1
(truncate/ -5 -2)          ⟹  2 -1
(truncate/ -5.0 -2)        ⟹  2.0 -1.0
```

(quotient n_1 n_2) 手続き
(remainder n_1 n_2) 手続き
(modulo n_1 n_2) 手続き

quotient および remainder 手続きはそれぞれ truncate-quotient および truncate-remainder と等価であり，modulo は floor-remainder と等価である，

注: これらの手続きは，本報告書の以前のバージョンとの下位互換性のために提供されている。

(gcd n_1 ...) 手続き
(lcm n_1 ...) 手続き

これらの手続きは引数の最大公約数 (greatest common divisor) または最小公倍数 (least common multiple) を返す。結果は常に非負である。

```
(gcd 32 -36)               ⟹  4
(gcd)                      ⟹  0
(lcm 32 -36)               ⟹  288
(lcm 32.0 -36)             ⟹  288.0  ; inexact
(lcm)                      ⟹  1
```

(numerator q) 手続き
(denominator q) 手続き

これらの手続きは引数の分子 (numerator) または分母 (denominator) を返す。結果は，引数があたかも既約分数として表現されているかのように計算される。分母は常に正である。 0 の分母は 1 であると定義されている。

```
(numerator (/ 6 4))        ⟹  3
(denominator (/ 6 4))      ⟹  2
(denominator
  (inexact (/ 6 4)))       ⟹  2.0
```

(floor x) 手続き
(ceiling x) 手続き
(truncate x) 手続き
(round x) 手続き

これらの手続きは整数を返す。floor 手続きは x 以下の最大の整数を返す。ceiling 手続きは x 以上の最小の整数を返す。truncate は絶対値が x の絶対値以下であって x に

最も近い整数を返す。そして `round` は x に最も近い整数を返す，ただし x が二つの整数の中間ならば偶数へと丸める。

根拠: `round` 手続きが偶数へと丸めることは，IEEE 754 IEEE floating-point standard で規定されたデフォルト丸めモードと一致する。

注: もしもこれらの手続きへの引数が不正確数ならば，その結果もまた不正確数になる。もし正確値が必要ならば，結果を `exact` 手続きに渡すことができる。引数が無限大や NaN の場合，それが返される。

```
(floor -4.3)                ⟹  -5.0
(ceiling -4.3)              ⟹  -4.0
(truncate -4.3)             ⟹  -4.0
(round -4.3)                ⟹  -4.0

(floor 3.5)                 ⟹  3.0
(ceiling 3.5)               ⟹  4.0
(truncate 3.5)              ⟹  3.0
(round 3.5)                 ⟹  4.0   ; 不正確数

(round 7/2)                 ⟹  4     ; 正確数
(round 7)                   ⟹  7
```

`(rationalize` x y`)` 手続き

`rationalize` 手続きは，x に対して y 以内の誤差で等しい**最も単純な**有理数を返す。有理数 r_1, r_2 について $r_1 = p_1/q_1$, $r_2 = p_2/q_2$ (ただし既約分数) とするとき，$|p_1| \le |p_2|$ かつ $|q_1| \le |q_2|$ ならば，有理数 r_1 は有理数 r_2 **より単純**である。たとえば 3/5 は 4/7 より単純である。すべての有理数がこの順序で比較できるわけではないが (たとえば 2/7 と 3/5)，数直線上のいかなる区間も，その区間内の他のどの有理数よりも単純であるような有理数を 1 個含んでいる (2/7 と 3/5 の間にはより単純な 2/5 が存在する)。$0 = 0/1$ がすべての有理数のうちで最も単純であることに注意せよ。

```
(rationalize
  (exact .3) 1/10)          ⟹  1/3     ; 正確数
(rationalize .3 1/10)       ⟹  #i1/3   ; 不正確数
```

`(exp` z`)`	inexact ライブラリ手続き
`(log` z`)`	inexact ライブラリ手続き
`(log` z_1 z_2`)`	inexact ライブラリ手続き
`(sin` z`)`	inexact ライブラリ手続き
`(cos` z`)`	inexact ライブラリ手続き
`(tan` z`)`	inexact ライブラリ手続き
`(asin` z`)`	inexact ライブラリ手続き
`(acos` z`)`	inexact ライブラリ手続き
`(atan` z`)`	inexact ライブラリ手続き
`(atan` y x`)`	inexact ライブラリ手続き

これらの手続きは，おなじみの超越関数を計算する。`log` 手続きは，一引数の場合は z の自然対数 (常用対数ではない) を，二引数の場合は z_2 を底とする z_1 の対数を計算する。`asin`, `acos`, および `atan` 手続きはそれぞれ arcsine ($\sin^{-1}$), arc-cosine ($\cos^{-1}$), および arctangent ($\tan^{-1}$) を計算する。

二引数の `atan` は，たとえ一般的な複素数をサポートしない実装であっても，`(angle (make-rectangular` x y`))` を計算する (下記参照)。

一般に，数学上の関数 log, arcsine, arc-cosine, および arctangent は多価関数として定義される。

$\log z$ の値は一つになるように，その虚部は $-\pi$ (`-0.0` が区別されている場合は含み，そうでない場合は含まない) から π (含む) までの区間で定義されている。このように log を定義したとき，$\sin^{-1} z$, $\cos^{-1} z$, および $\tan^{-1} z$ の値は以下の公式によって決まる:

$$\sin^{-1} z = -i \log(iz + \sqrt{1 - z^2})$$

$$\cos^{-1} z = \pi/2 - \sin^{-1} z$$

$$\tan^{-1} z = (\log(1 + iz) - \log(1 - iz))/(2i)$$

しかし，実装が無限大 (および `-0.0`) をサポートしていれば `(log 0.0)` は `-inf.0` を返す (そして `(log -0.0)` は `-inf.0+`π`i` を返す)。

`(atan` y x`)` の区間は次表のとおりである。アスタリスク (*) は，エントリが負のゼロを区別する実装に適用していることを示している。

	y の条件	x の条件	結果 r の値域
	$y = 0.0$	$x > 0.0$	0.0
*	$y = +0.0$	$x > 0.0$	$+0.0$
*	$y = -0.0$	$x > 0.0$	-0.0
	$y > 0.0$	$x > 0.0$	$0.0 < r < \frac{\pi}{2}$
	$y > 0.0$	$x = 0.0$	$\frac{\pi}{2}$
	$y > 0.0$	$x < 0.0$	$\frac{\pi}{2} < r < \pi$
	$y = 0.0$	$x < 0$	π
*	$y = +0.0$	$x < 0.0$	π
*	$y = -0.0$	$x < 0.0$	$-\pi$
	$y < 0.0$	$x < 0.0$	$-\pi < r < -\frac{\pi}{2}$
	$y < 0.0$	$x = 0.0$	$-\frac{\pi}{2}$
	$y < 0.0$	$x > 0.0$	$-\frac{\pi}{2} < r < 0.0$
	$y = 0.0$	$x = 0.0$	未定義
*	$y = +0.0$	$x = +0.0$	$+0.0$
*	$y = -0.0$	$x = +0.0$	-0.0
*	$y = +0.0$	$x = -0.0$	π
*	$y = -0.0$	$x = -0.0$	$-\pi$
*	$y = +0.0$	$x = 0$	$\frac{\pi}{2}$
*	$y = -0.0$	$x = 0$	$-\frac{\pi}{2}$

上記の規定は [34] に従っており，そしてそれは翻って [26] を引用している。これらの関数の分岐截線，境界条件，および実装のより詳細な議論についてはこれらの原典を参照せよ。可能な場合，これらの手続きは実数引数から実数結果を算出する。

`(square` z`)` 手続き

z の 2 乗を返す。これは `(*` z z`)` と等価である。

```
(square 42)                 ⟹  1764
(square 2.0)                ⟹  4.0
```

(sqrt z) inexact ライブラリ手続き

z の平方根の主値を返す。結果は正の実部をもつか，あるいはゼロの実部と非負の虚部をもつ。

```
(sqrt 9)                ⟹ 3
(sqrt -1)               ⟹ +i
```

(exact-integer-sqrt k) 手続き

$k = s^2 + r$ and $k < (s+1)^2$ が成り立つ非負の正確性数 s と r を返す。

```
(exact-integer-sqrt 4)  ⟹ 2 0
(exact-integer-sqrt 5)  ⟹ 2 1
```

(expt z_1 z_2) 手続き

z_1 の z_2 乗を返す。非ゼロの z_1 に対して

$$z_1{}^{z_2} = e^{z_2 \log z_1}$$

0^z の値は，(zero? z) の場合 1，(real-part z) が正の場合 0，それ以外の場合エラーである。同様に 0.0^z に対して，不正確数の結果となる。

(make-rectangular x_1 x_2)	complex ライブラリ手続き
(make-polar x_3 x_4)	complex ライブラリ手続き
(real-part z)	complex ライブラリ手続き
(imag-part z)	complex ライブラリ手続き
(magnitude z)	complex ライブラリ手続き
(angle z)	complex ライブラリ手続き

x_1, x_2, x_3, および x_4 が実数であり，z が複素数としたとき，

$$z = x_1 + x_2 i = x_3 \cdot e^{ix_4}$$

このとき，すべての

```
(make-rectangular x1 x2)  ⟹ z
(make-polar x3 x4)        ⟹ z
(real-part z)             ⟹ x1
(imag-part z)             ⟹ x2
(magnitude z)             ⟹ |x3|
(angle z)                 ⟹ x_angle
```

が真となる。ここで，ある整数 n に対して $-\pi \leq x_{angle} \leq \pi$ with $x_{angle} = x_4 + 2\pi n$ である。

make-polar 手続きは，引数が正確数であっても不正確虚数を返してよい。real-part および imag-part 手続きが不正確虚数に適用されたときには，make-rectangular に渡された対応する引数が正確数の場合は，正確実数を返してよい。

根拠: magnitude 手続きは実数引数に対して abs と同一である。しかし，abs が base ライブラリである一方，magnitude は，オプションの complex ライブラリである。

(inexact z) 手続き
(exact z) 手続き

inexact 手続きは z の不正確表現を返す。返される値は，引数に数値的に最も近い不正確数である。不正確数の引数に対して，結果は引数と同じである。正確複素数に対して，結果は，実部と虚部がそれぞれ引数の実部と虚部に inexact を適用した複素数である。もしも正確引数にほどよく近い不正確数が (= の意味において) なければ，実装制限の違反を報告してもよい。

exact 手続きは z の正確表現を返す。返される値は，引数に数値的に最も近い正確数である。正確数の引数に対して，結果は引数と同じである。非整数の不正確実数引数に対して，実装は，合理的な近似値を返してもよいし，実装違反を報告してもよい。不正確複素引数に対して，結果は，実部と虚部がそれぞれ引数の実部と虚部に exact を適用した複素数である。もしも不正確引数にほどよく近い正確数がなければ，実装制限の違反を報告してもよい。

これらの手続きは，正確整数と不正確整数のあいだの自然な一対一の対応を，ある実装依存の値域にわたって実装する。6.2.3 節を見よ。

注: これらの手続きは R^{5}RS でそれぞれ exact->inexact および inexact->exact として知られているが，それらは常に任意の正確の引数を受け取っている。

6.2.7. 数値入出力

(number->string z) 手続き
(number->string z $radix$) 手続き

$radix$ は 2, 8, 10, または 16 のいずれかでなければエラーである。

手続き number->string は数 ($number$) と基数 ($radix$) を取り，その数のその基数での外部表現を，次式が真である文字列として返す。

```
(let ((number number)
      (radix radix))
  (eqv? number
        (string->number (number->string number
                                        radix)
                        radix)))
```

もしこの式を真にする結果があり得なければエラーである。$radix$ は省略時， 10 と見なされる。

もし z が不正確であり，基数が 10 であって，かつ上式が小数点付きの結果によって充足され得るならば，そのとき結果は小数点付きであり，上式を真にするために必要な最小個数の数字 (指数部と末尾のゼロを計算に入れないで) を使って表現される [4, 5]。それ以外，結果の書式は未規定である。

number->string が返す結果は決して明示的な基数接頭辞を含まない。

注: エラーが起こり得るのは，z が複素数でないか，または有理数でない実部または虚部をもつ複素数であるときだけである。

根拠: もし z が不正確数であり，基数が 10 ならば，そのとき上式は通常，小数点付きの結果によって充足される。未規定の場合としては無限大，NaN, および異常な表現が考えられている。

(string->number *string*) 手続き
(string->number *string radix*) 手続き

与えられた文字列 (*string*) によって表される最大限に精度の高い表現の数を返す。*radix* は 2, 8, 10, または 16 のいずれかでなければエラーである。

radix を与えるとそれがデフォルトの基数になるが，それよりも *string* 内の明示的な基数 (たとえば "#o177") が優先される。*radix* の省略時，デフォルトの基数は 10 である。*string* が構文的に妥当な数の表記でない，または *string* が実装が表現できない数になる場合，string->number は #f を返す。*string* の内容に起因するエラーが通知されることはない。

```
(string->number "100")      ⟹  100
(string->number "100" 16)   ⟹  256
(string->number "1e2")      ⟹  100.0
```

注: string->number の定義域を実装は次のように制限してもよい。もしも実装がサポートする数がすべて実数ならば，string->number は，*string* が複素数に対する極座標系または直交座系表記を使った場合に対して常に #f を返すことが許可されている。もしもすべての数が整数ならば，string->number は，分数表記が使われた場合に対して常に #f を返してよい。もしもすべての数が正確数ならば，string->number は，指数部マーカまたは明示的な正確性接頭辞が使われた場合に対して常に#f を返してよい。もしもすべての不正確数が整数ならば，string->number は，小数点が使われた場合に対して常に #f を返してよい。

string->number の特定の実装で使用されるルールはまた，内部数値処理，I/O，およびプログラムの処理との整合性を維持するために，read およびプログラムを読み出すルーチンに適用されなければならない。結果として，*string* が明示的な基数接頭辞を持つ場合は #f を返すように R[5]RS 許可が撤回された。

6.3. ブーリアン

真と偽をあらわす標準的なブーリアンオブジェクトは #t および #f と書かれる。あるいは，それらは，それぞれ #true および #false と書くことができる。しかし，実際に問題になるのは，Scheme の条件式 (if, cond, and, or, when, unless, do) が真または偽として扱うオブジェクトである。“真値” (または単に “真”) という表現は条件式が真として扱ういかなるオブジェクトのことも意味し，“偽値” (または “偽”) という表現は条件式が偽として扱ういかなるオブジェクトのことも意味する。

すべての Scheme 値のうちで，ただ #f だけが条件式において偽と見なされる。他のすべての Scheme 値は #t を含め，真と見なされる。

注: Lisp の他のいくつかの方言とは違って，Scheme は #f と空リストを互いに区別し，シンボル nil とも区別していることに注意すべきである。

ブーリアン定数はそれ自身へと評価されるから，プログラムの中でそれらをクォートする必要はない。

```
#t      ⟹  #t
#f      ⟹  #f
'#f     ⟹  #f
```

(not *obj*) 手続き

not 手続きは *obj* が偽ならば #t を返し，そうでなければ #f を返す。

```
(not #t)          ⟹  #f
(not 3)           ⟹  #f
(not (list 3))    ⟹  #f
(not #f)          ⟹  #t
(not '())         ⟹  #f
(not (list))      ⟹  #f
(not 'nil)        ⟹  #f
```

(boolean? *obj*) 手続き

boolean? 述語は *obj* が #t か #f ならば #t を返し，そうでなければ #f を返す。

```
(boolean? #f)     ⟹  #t
(boolean? 0)      ⟹  #f
(boolean? '())    ⟹  #f
```

(boolean=? $boolean_1$ $boolean_2$ $boolean_3$ ...) 手続き

すべての引数がブーリアンかつそれらがすべて #t あるいはすべてが #f ならば #t を返す。

6.4. ペアとリスト

ペア *pair* (ときには**ドット対** *dotted pair* と呼ばれる) とは，(歴史的な理由から) car 部，cdr 部と呼ばれる二つのフィールドをもったレコード構造である。ペアは手続き cons によって作成される。car 部と cdr 部は手続き car と cdr によってアクセスされる。car 部と cdr 部は手続き set-car! と set-cdr! によって代入される。

ペアはまずリストを表現するために使われる。リスト (*list*) は，空リスト (the empty list)，または cdr 部がリストであるようなペア，として再帰的に定義できる。より正確には，リストの集合は，下記を満たす最小の集合 X として定義される。

- (ただ一つの) 空リストは X にある。
- もし *list* が X にあるならば，cdr 部が *list* を収めているどのペアもまた X にある。

一つのリストの一連のペアの各 car 部にあるオブジェクトは，そのリストの要素である。たとえば，2 要素リストとは，car が第 1 要素であって cdr がペアであり，そしてそのペアの car が第 2 要素であって cdr が空リストであるようなペアである。リストの長さとは要素の個数であり，それはペアの個数と同じである。

空リストは，それ自身の型に属する一つの特殊なオブジェクトである。これはペアではなく，要素を持っておらず，その長さはゼロである。

注: 上記の定義は，すべてのリストが有限の長さをもち，空リストを終端とすることを，暗黙のうちに意味している。

Scheme のペアを表す最汎の表記 (外部表現) は，“ドット対” 表記 (dotted notation) (c_1 . c_2) である。ここで c_1 は car 部の値であり，c_2 は cdr 部の値である。たとえば (4 . 5) は，car が 4 であり cdr が 5 であるペアである。(4 . 5) はペアの外部表現であるが，ペアへと評価される式ではないことに注意せよ。

リストに対しては，より能率的な表記が使える。リストの各要素をスペースで区切って丸カッコで囲むだけでよい。空リストは () と書かれる。たとえば，

```
(a b c d e)
```

と

```
(a . (b . (c . (d . (e . ()))))))
```

は，シンボルからなるあるリストを表す等価な表記である。

ペアの連鎖であって，空リストで終わらないものは，**非真正リスト** (*improper list*) と呼ばれる。非真正リストはリストではないことに注意せよ。リスト表記とドット対表記を組み合わせて，非真正リストを表現してよい:

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

は次と等価である。

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

あるペアがリストかどうかは，何が cdr 部に格納されているかに依存する。set-cdr! 手続きを使えば，あるオブジェクトがある時点ではリストであり，かつ次の時点ではそうでない，ということが可能である:

```
(define x (list 'a 'b 'c))
(define y x)
y                          ⟹  (a b c)
(list? y)                  ⟹  #t
(set-cdr! x 4)             ⟹  未規定
x                          ⟹  (a . 4)
(eqv? x y)                 ⟹  #t
y                          ⟹  (a . 4)
(list? y)                  ⟹  #f
(set-cdr! x x)             ⟹  未規定
(list? x)                  ⟹  #f
```

リテラル式の中と，read 手続きが読むオブジェクトの表現の中で，形式 '<データ> と `<データ> と ,<データ> と ,@<データ> はそれぞれ 2 要素リストを表しており，その第 1 要素はそれぞれシンボル quote，quasiquote，unquote，unquote-splicing である。各場合とも第 2 要素は <データ> である。この規約は，任意の Scheme プログラムをリストとして表現できるようにと，サポートされている。つまり，Scheme の文法に従えば，どの <式> も <データ> である (7.1.2 節参照)。とりわけ，このことは，read 手続きを使って Scheme プログラムをパースすることを可能にしている。3.3 節を見よ。

(pair? *obj*) 手続き

pair? 述語は，もし *obj* がペアならば #t を返し，そうでなければ #f を返す。

```
(pair? '(a . b))           ⟹  #t
(pair? '(a b c))           ⟹  #t
(pair? '())                ⟹  #f
(pair? '#(a b))            ⟹  #f
```

(cons obj_1 obj_2) 手続き

一つの新たに割り当てられたペアを返す。ペアの car は obj_1 であり，cdr は obj_2 である。このペアは，存在するどのオブジェクトからも (eqv? の意味で) 異なっていることが保証されている。

```
(cons 'a '())              ⟹  (a)
(cons '(a) '(b c d))       ⟹  ((a) b c d)
(cons "a" '(b c))          ⟹  ("a" b c)
(cons 'a 3)                ⟹  (a . 3)
(cons '(a b) 'c)           ⟹  ((a b) . c)
```

(car *pair*) 手続き

pair の car 部の内容を返す。空リストの car を取ることはエラーであることに注意せよ。

```
(car '(a b c))             ⟹  a
(car '((a) b c d))         ⟹  (a)
(car '(1 . 2))             ⟹  1
(car '())                  ⟹  エラー
```

(cdr *pair*) 手続き

pair の cdr 部の内容を返す。空リストの cdr を取ることはエラーであることに注意せよ。

```
(cdr '((a) b c d))         ⟹  (b c d)
(cdr '(1 . 2))             ⟹  2
(cdr '())                  ⟹  エラー
```

(set-car! *pair obj*) 手続き

pair の car 部に *obj* を格納する。

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)           ⟹  未規定
(set-car! (g) 3)           ⟹  エラー
```

(set-cdr! *pair obj*) 手続き

pair の cdr 部に *obj* を格納する。

(caar *pair*) 手続き
(cadr *pair*) 手続き
(cdar *pair*) 手続き
(cddr *pair*) 手続き

これらの手続きは次のように car と cdr の合成である。

```
(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))
```

(caaar *pair*) cxr ライブラリ手続き
(caadr *pair*) cxr ライブラリ手続き
⋮ ⋮
(cdddar *pair*) cxr ライブラリ手続き
(cddddr *pair*) cxr ライブラリ手続き

これら 24 個の手続きは同じ原理による car と cdr の更なる合成である。例えば， caddr は次のように定義できる。

```
(define caddr (lambda (x) (car (cdr (cdr x))))).
```

深さ 4 までの任意の合成が用意されている。

(null? *obj*) 手続き

もし *obj* が空リストならば #t を返し，そうでなければ #f を返す。

(list? *obj*) 手続き

もし *obj* がリストならば #t を返す。そうでなければ #f を返す。定義により，すべてのリストは有限の長さをもち，空リストを終端とする。

```
(list? '(a b c))    ⟹  #t
(list? '())         ⟹  #t
(list? '(a . b))    ⟹  #f
(let ((x (list 'a)))
  (set-cdr! x x)
  (list? x))        ⟹  #f
```

(make-list *k*) 手続き
(make-list *k fill*) 手続き

k 個の要素をもつ，新たに割り当てられたリストを返す。二つ目の引数が与えられた場合，各要素は *fill* で初期化される。そうでなければ各要素の初期値は不定である。

```
(make-list 2 3)          ⟹  (3 3)
```

(list *obj* ...) 手続き

その引数からなる，一つの新たに割り当てられたリストを返す。

```
(list 'a (+ 3 4) 'c)     ⟹  (a 7 c)
(list)                   ⟹  ()
```

(length *list*) 手続き

list の長さを返す。

```
(length '(a b c))        ⟹  3
(length '(a (b) (c d e)))⟹  3
(length '())             ⟹  0
```

(append *list* ...) 手続き

最後の引数は，一つある場合，任意の型であってもよい。

引数がない場合，空のリストを返す。正確に一つの引数が存在する場合には，それが返される。そうでない場合は結果のリストは，最後の引数と構造を共有することを除いて，常に新たに割り当てられる。最後の引数が適切なリストではない場合には，不適切なリストになる。

```
(append '(x) '(y))          ⟹  (x y)
(append '(a) '(b c d))      ⟹  (a b c d)
(append '(a (b)) '((c)))    ⟹  (a (b) (c))

(append '(a b) '(c . d))    ⟹  (a b c . d)
(append '() 'a)             ⟹  a
```

(reverse *list*) 手続き

逆順にした *list* の要素で構成された，一つの新たに割り当てられたリストを返す。

```
(reverse '(a b c))          ⟹  (c b a)
(reverse '(a (b c) d (e (f))))
          ⟹  ((e (f)) d (b c) a)
```

(list-tail *list k*) 手続き

list の要素数が *k* より少なければエラーである。

list から最初の *k* 要素を省いて得られる部分リストを返す。list-tail 手続きは次のように定義できる。

```
(define list-tail
  (lambda (x k)
    (if (zero? k)
        x
        (list-tail (cdr x) (- k 1)))))
```

(list-ref *list k*) 手続き

list 引数は環状になりうるが，もしも *list* の要素数が *k* より少なければエラーである。

list の第 *k* 要素を返す。(これは (list-tail *list k*) の car と同じである。)

```
(list-ref '(a b c d) 2)     ⟹  c
(list-ref '(a b c d)
          (exact (round 1.8)))
          ⟹  c
```

(list-set! *list k obj*) 手続き

k が *list* の有効なインデックスでなければエラーである。

list-set! 手続きは， *list* の第 *k* 要素に *obj* を格納する。

```
(let ((ls (list 'one 'two 'five!)))
  (list-set! ls 2 'three)
  ls)
          ⟹  (one two three)

(list-set! '(0 1 2) 1 "oops")
          ⟹  エラー  ; 定数リスト
```

(memq *obj list*) 手続き
(memv *obj list*) 手続き
(member *obj list*) 手続き
(member *obj list compare*) 手続き

これらの手続きは，*list* の部分リストのうち，car が *obj* である最初のものを返す。ここで *list* の部分リストとは，$k <$ *list*の長さ に対して (list-tail *list k*) が返す空でないリストである。もしも *obj* が *list* に出現しないならば，(空リストではなく) #f が返される。memq 手続きは eq? を使って *obj* を *list* の各要素と比較するが，memv は eqv? を使い，member は与えられれば *compare* を使い，そうでない場合は equal? を使う。

```
(memq 'a '(a b c))          ⟹  (a b c)
(memq 'b '(a b c))          ⟹  (b c)
(memq 'a '(b c d))          ⟹  #f
(memq (list 'a) '(b (a) c)) ⟹  #f
(member (list 'a)
        '(b (a) c))         ⟹  ((a) c)
(member "B"
        '("a" "b" "c")
        string-ci=?)        ⟹  ("b" "c")
(memq 101 '(100 101 102))   ⟹  未規定
(memv 101 '(100 101 102))   ⟹  (101 102)
```

(assq *obj alist*) 手続き
(assv *obj alist*) 手続き
(assoc *obj alist*) 手続き
(assoc *obj alist compare*) 手続き

alist (つまり，連想リスト “association list”) が，ペアからなるリストでなければエラーである。

これらの手続きは，*alist* の中のペアのうち，car 部が *obj* である最初のものを見つけて，そのペアを返す。もしも *alist* のどのペアも *obj* を car としないならば，(空リストではなく) #f が返される。assq 手続きは eq? を使って *obj* を *alist* の中の各ペアの car 部と比較するが，assv は eqv? を使い，assoc は与えられれば *compare* を使い，そうでない場合は equal? を使う。

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                 ⟹  (a 1)
(assq 'b e)                 ⟹  (b 2)
(assq 'd e)                 ⟹  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  ((a))
(assoc 2.0 '((1 1) (2 4) (3 9)) =)
                            ⟹  (2 4)
(assq 5 '((2 3) (5 7) (11 13)))
                            ⟹  未規定
(assv 5 '((2 3) (5 7) (11 13)))
                            ⟹  (5 7)
```

根拠: memq, memv, member, assq, assv, および assoc は，しばしば述語として使われるが，単なる #t や #f だけではない潜在的に有用な値を返すため，名前に疑問符を付けない。

(list-copy *obj*) 手続き

*obj*がリストの場合に，その新たに割り当てられたコピーを返す。ペア自身がコピーされる。結果の car は，*list* の car と (eqv?の意味において) 同じである。*obj* が不適切なリストの場合，それが結果となり，最後のな cdr は eqv? の意味において同じである。リストではない *obj* が返すは変わらない。*obj*が環状リストの場合はエラーである。

```
(define a '(1 8 2 8)) ; a may be immutable
(define b (list-copy a))
(set-car! b 3)        ; b is mutable
b                           ⟹  (3 8 2 8)
a                           ⟹  (1 8 2 8)
```

6.5. シンボル

シンボルとは，二つのシンボルが (eqv? の意味で) 同一なのは名前が同じようにつづられるときかつそのときに限られるという事実に，その有用性がかかっているオブジェクトである。たとえば，列挙値を他の言語で利用するのと同じ用途に，シンボルを利用してもよい。

シンボルを書くための規則は，識別子を書くための規則と正確に同じである。2.1 節と 7.1.1 節を見よ。

どのシンボルであれ，リテラル式の一部として返されたか，または read 手続きを使って読まれたシンボルを，write 手続きを使って外に書いた場合，それは (eqv? の意味で) 同一のシンボルとして再び読み込まれることが保証されている。

注: これは write/read 不変性をくつがえし，かつまた，二つのシンボルが同じなのは名前のつづりが同じときかつそのときに限られるという規則に違反する。本報告書では，実装依存の拡張の振舞いを指定しない。

(symbol? *obj*) 手続き

もし *obj* がシンボルならば #t を返し，そうでなければ #f を返す。

```
(symbol? 'foo)              ⟹  #t
(symbol? (car '(a b)))      ⟹  #t
(symbol? "bar")             ⟹  #f
(symbol? 'nil)              ⟹  #t
(symbol? '())               ⟹  #f
(symbol? #f)                ⟹  #f
```

(symbol=? $symbol_1$ $symbol_2$ $symbol_3$ ...) 手続き

もしすべての引数がシンボルかつ，すべてが string=? の意味において同じ名前を持つならば#t を返す。

注: 上記の定義は，インターンしてないシンボルが引数にないことを仮定している。

(symbol->string *symbol*) 手続き

symbol の名前を，エスケープを追加することなく一つの文字列として返す。この手続きが返す文字列に，string-set! のような書換え手続きを適用することはエラーである。

```
(symbol->string 'flying-fish)
                                ⟹  "flying-fish"
(symbol->string 'Martin)        ⟹  "Martin"
(symbol->string
   (string->symbol "Malvina"))
                                ⟹  "Malvina"
```

(string->symbol *string*) 手続き

名前が *string* であるシンボルを返す。この手続きは，特殊文字を含んでいる名前をもったシンボルを作成することができ，特殊文字が書かれた時にはエスケープが必要となるが，その入力ではエスケープを解釈しない。

```
(string->symbol "mISSISSIppi")
          ⟹  mISSISSIppi
(eqv? 'bitBlt (string->symbol "bitBlt"))
          ⟹  #t
(eqv? 'LollyPop
     (string->symbol
       (symbol->string 'LollyPop)))
          ⟹  #t
(string=? "K. Harper, M.D."
          (symbol->string
            (string->symbol "K. Harper, M.D.")))
          ⟹  #t
```

6.6. 文字

文字とは，英字や数字のような印字文字 (printed character) を表現するオブジェクトである。すべての Scheme の実装は，少なくとも ASCII 文字レパートリをサポートしている必要がある。つまり，Unicode 文字の U+0000 から U+007F までである。実装は，適当と考えられる他の Unicode 文字をサポートしてもよいし，同様に非 Unicode 文字をサポートしてもよい。別途記載のものを除いて，次の手続きのいずれかを非 Unicode 文字に適用した結果は，実装依存である。

文字は #\<文字> または #\<文字名> または #\x<16 進スカラ値> という表記を使って書かれる。

次の文字名は，指定された値で，すべての実装でサポートされなければならない。実装は，x で始まる 16 進のスカラ値として解釈されないような提供されている他の名前を追加してもよい。

```
#\alarm      ; U+0007
#\backspace  ; U+0008
#\delete     ; U+007F
#\escape     ; U+001B
#\newline    ; 改行文字, U+000A
#\null       ; ヌル文字, U+0000
#\return     ; 復帰文字, U+000D
#\space      ; スペースを書くための好ましい方法
#\tab        ; タブ文字, U+0009
```

ここではいくつかの追加的な例を示す:

```
#\a       ; 英小文字
#\A       ; 英大文字
#\(       ; 左丸カッコ
#\        ; スペース文字
#\x03BB   ; λ (文字がサポートされている場合)
#\iota    ; ι (文字および名前がサポートされている場合)
```

大文字と小文字の違いは #\<文字>，ならびに#\⟨ 文字名 ⟩ では意味をもつが，#\x<16 進スカラ値> では意味をもたない。#\<文字> の <文字> がアルファベットの場合，<文字> の直後の任意の文字は，識別子内に表れるそれであってはならない。この規則は，たとえば，文字の列 “#\space” が 1 個のスペース文字の表現であるとも取れるし，あるいは文字 “#\s” の表現の後にシンボル “pace” の表現が続いたものとも取れる，というあいまいなケースを解決する。

#\ 表記で書かれた文字は自己評価的である。すなわち，プログラムの中でそれらをクォートする必要はない。

文字について演算する手続きのいくつかは，大文字と小文字の違いを無視する。大文字と小文字の違いを無視する手続きは，その名前に “-ci” (つまり “case insensitive”) を埋め込んでいる。

(char? *obj*) 手続き

もし *obj* が文字ならば #t を返し，そうでなければ #f を返す。

(char=? $char_1$ $char_2$ $char_3$ ...) 手続き
(char<? $char_1$ $char_2$ $char_3$ ...) 手続き
(char>? $char_1$ $char_2$ $char_3$ ...) 手続き
(char<=? $char_1$ $char_2$ $char_3$ ...) 手続き
(char>=? $char_1$ $char_2$ $char_3$ ...) 手続き

これらの手続きは，char->integer に各引数を渡したときの結果がそれぞれ，等しい，単調増加 (より小)，単調減少 (より大)，単調非減少 (より小さいか又は等しい)，または単調非増加 (より大きいか又は等しい) の場合に #t を返す。

これらの述語は推移的であることが必須である。

(char-ci=? $char_1$ $char_2$ $char_3$ ...)
char ライブラリ手続き
(char-ci<? $char_1$ $char_2$ $char_3$ ...)
char ライブラリ手続き
(char-ci>? $char_1$ $char_2$ $char_3$ ...)
char ライブラリ手続き
(char-ci<=? $char_1$ $char_2$ $char_3$ ...)
char ライブラリ手続き
(char-ci>=? $char_1$ $char_2$ $char_3$ ...)
char ライブラリ手続き

これらの手続きは char=? 等と同様だが，英大文字と英小文字を同じものとして扱う。たとえば，(char-ci=? #\A #\a) は #t を返す。

具体的には，これらの手続きは引数を比較する前に char-foldcase がそれらに適用されたかのように振舞う。

(char-alphabetic? *char*) char ライブラリ手続き
(char-numeric? *char*) char ライブラリ手続き
(char-whitespace? *char*) char ライブラリ手続き
(char-upper-case? *letter*) char ライブラリ手続き
(char-lower-case? *letter*) char ライブラリ手続き

これらの手続きは，それぞれ，その引数がアルファベット文字，数値文字，空白文字，大文字，または小文字ならば #t を返し，そうでなければ #f を返す。

具体的には，Unicode 文字プロパティがそれぞれ Alphabetic, Numeric_Digit, White_Space, Uppercase および Lowercase の場合に #t を返し，他の Unicode 文字に適用された場合には #f を返さなければならない。多くの Unicode 文字は Alphabetic だが大文字でも小文字でないことに注意せよ。

(digit-value *char*) char ライブラリ手続き

この手続きは，数字桁の (すなわち，char-numeric? が #t を返す) 場合にはその数値 (0 から 9) を返し，他の任意の文字の場合は #f を返す。

```
(digit-value #\3)       ⟹ 3
(digit-value #\x0664)   ⟹ 4
(digit-value #\x0AE6)   ⟹ 0
(digit-value #\x0EA6)   ⟹ #f
```

(char->integer *char*) 手続き
(integer->char *n*) 手続き

Unicode 文字が与えられたとき，char->integer はその文字の Unicode スカラ値に等しい 0 から #xD7FF の間，または #xE000 から #x10FFFF の間の正確整数を返す。非 Unicode 文字が与えられたときには，#x10FFFF より大きい正確整数を返す。これは，実装が内部的に Unicode 表現を使用するかどうかとは真に独立している。

char->integer が適用されたときにその文字が返す値の正確整数が与えられたとき，integer->char はその文字を返す。

(char-upcase *char*) char ライブラリ手続き
(char-downcase *char*) char ライブラリ手続き
(char-foldcase *char*) char ライブラリ手続き

char-upcase 手続きは， Unicode のケーシング対 (大文字と小文字のペア) の大文字部分が引数に与えられると，両方の文字が Scheme の実装によってサポートされていれば，対の小文字のメンバを返す。言語に依存するケーシング対が使用されていないことに注意せよ。引数がそのような対の大文字メンバではない場合，それが返される。

char-downcase 手続きは， Unicode のケーシング対の小文字部分で引数が与えられると，両方の文字が Scheme の実装によってサポートされていれば，対の大文字のメンバを返す。言語に依存するケーシング対が使用されていないことに注意せよ。引数がそのような対の小文字メンバではない場合，それが返される。

char-foldcase 手続きでは，その引数に Unicode の単純な大文字小文字変換アルゴリズムを適用し，その結果を返す。言語に依存する変換が使用されていないことに注意せよ。引数が大文字であれば，結果は小文字，あるいは小文字が存在しないか実装によってサポートされていなければ引数と同じ，のいずれかである。詳細は，UAX #29 [11] (Unicode 標準の一部) を参照のこと。

多くの Unicode の小文字は，等価な大文字を持たないことに注意せよ。

6.7. 文字列

文字列 (string) とは，文字の列 (sequence of characters) である。文字列は引用符 (") で囲まれた文字の列として書かれる。文字列リテラル内部では，さまざまなエスケープシーケンスがそれ自身以外の文字を表す。エスケープシーケンスは常にバックスラッシュ(\) で始まる。

- \a : alarm, U+0007
- \b : backspace, U+0008
- \t : character tabulation, U+0009
- \n : linefeed, U+000A
- \r : return, U+000D
- \" : double quote, U+0022
- \\ : backslash, U+005C
- \| : vertical line, U+007C
- \<行内空白>*<行末><行内空白>* : nothing
- \x<16 進スカラ値>; : specified character (note the terminating semi-colon).

文字列内で他の文字がバックスラッシュの後に現れた場合，結果は未規定である。

行末を除いて，エスケープシーケンスの外部の任意の文字は，リテラル文字列で，それ自体を意味する。\<行内空白> が付いている行末は，(末尾のライン内空白と一緒に) 空に展開され，改善された読みやすさのためにインデント文字列に使用することができる。他の行末は，文字列に \n 文字を挿入するのと同じ効果を持っている。

例:

```
"The word \"recursion\" has many meanings."
"Another example:\ntwo lines of text"
"Here's text \
   containing just one line"
"\x03B1; is named GREEK SMALL LETTER ALPHA."
```

文字列の**長さ** (*length*) とは，文字列が含む文字の個数である。この個数は，文字列が作成されるときに固定される正確非負整数である。文字列の**妥当な添字** (*valid index*) は，文字列の長さより小さい正確非負整数である。文字列の最初の文字の添字は 0 であり，その次の文字の添字は 1 である等々である。

文字列について演算する手続きのいくつかは，大文字と小文字の違いを無視する。大文字と小文字の違いを無視するバージョンの名前は，“`-ci`” (つまり “case insensitive”) で終わる。

実装は，文字列に表示されない特定の文字を禁止してもよい。しかし，`#\null` 例外として， ASCII 文字が禁止されてはならない。例えば，実装は，Unicode レパートリー全体をサポートしてもよいが，文字列中の文字 U+0001 から U+00FF(`#\null` 以外の Latin-1 レパートリー) だけを許可する。

`make-string`, `string`, `string-set!`, または `string-fill!` に，`list->string` に渡されたリストの一部として，または `vector->string` に渡されたベクタの一部として (6.8 節参照)，または `utf8->string` (6.9 節参照) に渡されたバイトベクタ内の UTF-8 エンコードされた形式で，このような禁止文字を渡すことはエラーである。また，禁止された文字を返すために `string-map` (6.10 節参照) に渡された，またはそれを読み取ろうとする `read-string` (6.13.2 節参照) も手続きのエラーである。

`(string?` *obj*`)` 手続き

もし *obj* が文字列ならば `#t` を返し，そうでなければ `#f` を返す。

`(make-string` *k*`)` 手続き
`(make-string` *k char*`)` 手続き

`make-string` は，一つの新たに割り当てられた，長さ *k* の文字列を返す。もしも *char* が与えられたならば文字列のすべての文字が *char* に初期化されるが，そうでなければ文字列の内容は未規定である。

`(string` *char* ...`)` 手続き

引数から構成された，一つの新たに割り当てられた文字列を返す。これは `list` に似ている。

`(string-length` *string*`)` 手続き

与えられた *string* の中の文字の個数を返す。

`(string-ref` *string k*`)` 手続き

k が *string* の妥当な添字でなければエラーである。

`string-ref` 手続きは *string* の，ゼロから数えて第 *k* の文字を返す。 一定時間で実行するための，この手続きへの要件はない。

`(string-set!` *string k char*`)` 手続き

k が *string* の妥当な添字でなければエラーである。

`string-set!` は *string* の要素 *k* に *char* を格納する。一定時間で実行するための，この手続きへの要件はない。

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)      ⟹  未規定
(string-set! (g) 0 #\?)      ⟹  エラー
(string-set! (symbol->string 'immutable)
             0
             #\?)            ⟹  エラー
```

`(string=?` $string_1$ $string_2$ $string_3$ ...`)` 手続き

もしもすべての文字列が同じ長さであって同じ位置に正確に同じ文字を含んでいるならば `#t` を返し，そうでなければ `#f` を返す。

`(string-ci=?` $string_1$ $string_2$ $string_3$ ...`)` char ライブラリ手続き

もしも大文字小文字変換の後，すべての文字列が同じ長さであって同じ位置に同じ文字を含んでいるならば `#t` を返し，そうでなければ `#f` を返す。具体的には，これらの手続きは引数を比較する前にそれらに `string-foldcase` が適用されたかのように振舞う。

`(string<?` $string_1$ $string_2$ $string_3$ ...`)` 手続き
`(string-ci<?` $string_1$ $string_2$ $string_3$ ...`)` char ライブラリ手続き
`(string>?` $string_1$ $string_2$ $string_3$ ...`)` 手続き
`(string-ci>?` $string_1$ $string_2$ $string_3$ ...`)` char ライブラリ手続き
`(string<=?` $string_1$ $string_2$ $string_3$ ...`)` 手続き
`(string-ci<=?` $string_1$ $string_2$ $string_3$ ...`)` char ライブラリ手続き
`(string>=?` $string_1$ $string_2$ $string_3$ ...`)` 手続き
`(string-ci>=?` $string_1$ $string_2$ $string_3$ ...`)` char ライブラリ手続き

これらの手続きは，その引数が (それぞれ) 以下の場合に `#t` を返す: 単調減少，単調増加，単調非減少，または単調非増加。

これらの述語は推移的であることが必須である。

これらの手続きは，実装定義の方法で文字列を比較する。一つのアプローチは，文字についての対応する順序付けの，文字列への辞書的な拡張を作成することである。その場合，`string<?` は，文字の順序付け `char<?` によって帰納される辞書的な文字列の順序付けであり，もしも二つの文字列が，長さの点で異なっているが，短い方の文字列の長さまでは同じならば，短い方の文字列が長い方の文字列よりも辞書的に小さいと見なされるだろう。しかし，それはまた，実装の文字列の内部表現や，より複雑なロケール固有の順序によって課された自然な順序付けを使用することが許可されている。

すべての場合において，文字列の対は，正しく string<?, string=?, および string>? のいずれかを満たす必要があり，なおかつ，string>? を満たしていない場合に限り string<=? を満たし，string<? を満たしていない場合に限り string>=? を満たす必要がある。
“-ci”

手続きは，“-ci” がついていない対応する手続きを呼び出す前に，引数に string-foldcase が適用されたかのように振舞う。

(string-upcase *string*) char ライブラリ手続き
(string-downcase *string*) char ライブラリ手続き
(string-foldcase *string*) char ライブラリ手続き

これらの手続きは，引数に Unicode 文字列全体に大文字化，小文字化，および大文字小文字変換アルゴリズムを適用し，結果を返す。特定の場合において，結果は引数と長さが異なる。結果が string=? の意味において引数と等しい場合は，引数を返してもよい。言語に依存するマッピングおよび変換は使用されていないことに注意せよ。

Unicode 標準は，ギリシャ文字 Σ の特別な扱いを規定しており，その通常の小文字表記は σ だが，単語の末尾ではそれが ς になる。詳細は UAX #29 [11] (Unicode 標準の一部) を参照のこと。しかし，string-downcase の実装はこの振舞いを提供する必要はないし，すべての場合において Σ を σ に変更する選択をしてもよい。

(substring *string start end*) 手続き

substring は，*string* の，添字 *start* で始まり，添字 *end* で終わる文字から構成された，一つの新たに割り当てられた文字列を返す。これは同じ引数で string-copy を呼び出すことと等価であるが，下位互換性と文体の柔軟性のために提供される。

(string-append *string* ...) 手続き

その文字が与えられた文字列内の文字の連結であるような，一つの新たに割り当てられた文字列を返す。

(string->list *string*) 手続き
(string->list *string start*) 手続き
(string->list *string start end*) 手続き
(list->string *list*) 手続き

list の任意の要素が文字でない場合は，エラーである。

string->list 手続きは，*start* と *end* の間の *string* の新たに割り当てられた文字からなるリストを返す。list->string は，リスト *list* の要素から構成された，一つの新たに割り当てられた文字列を返す。どちらの手続きも，順序が保持される。string->list と list->string は，equal? に関する限りにおいて逆関数どうしである。

(string-copy *string*) 手続き
(string-copy *string start*) 手続き
(string-copy *string start end*) 手続き

与えられた *string* の *start* から *end* までの部分の，一つの新たに割り当てられたコピーを返す。

(string-copy! *to at from*) 手続き
(string-copy! *to at from start*) 手続き
(string-copy! *to at from start end*) 手続き

at がゼロより小さいか，*to* の長さより大きければエラーである。(- (string-length *to*) *at*) が (- *end start*) より小さい場合もエラーである。

文字列 *from* の *start* と *end* 間の文字を文字列 *to*に，*at* から始めてコピーする。文字がコピーされる順序は，次の場合を除いて未規定である。もしコピー元とコピー先が重なっている場合は，コピー元は最初に一時文字列にコピーした後，コピー先にコピーされているかのようにコピーが行われる。これは，このような状況において必ず正しい方向にコピーすることにより，記憶領域を割り当てることなく実現できる。

```
(define a "12345")
(define b (string-copy "abcde"))
(string-copy! b 1 a 0 2)
b                                ⟹ "a12de"
```

(string-fill! *string fill*) 手続き
(string-fill! *string fill start*) 手続き
(string-fill! *string fill start end*) 手続き

fill が文字でなければエラーである。

string-fill! 手続きは，*string* 内の *start* と *end* の間の要素に *fill* を格納する。

6.8. ベクタ

ベクタとは，その要素が整数によって添字付けられる不均質な (訳注: つまり要素どうしが同じ型とは限らない) 構造である。ベクタは一般的には同じ長さのリストよりも少ない空間しか占めず，かつランダムに選んだ要素をアクセスするために必要な平均時間は一般的にはベクタの方がリストよりも短い。

ベクタの**長さ** (*length*) とは，ベクタが含む要素の個数である。この個数は，ベクタが作成されるときに固定される非負整数である。ベクタの**妥当な添字** (*valid index*) は，ベクタの長さより小さい正確非負整数である。ベクタの最初の要素はゼロで添字付けられ，最後の要素の添字はベクタの長さより 1 だけ小さく添字付けられる。

ベクタは表記 #(*obj* ...) を使って書かれる。たとえば，長さが 3 であって，要素 0 に数ゼロを，要素 1 にリスト (2 2 2 2) を，そして要素 2 に文字列 "Anna" を含んでいるベクタは下記のように書ける:

```
#(0 (2 2 2 2) "Anna")
```

ベクタ定数は自己評価しているので，プログラム中でクォートする必要はない。

(vector? *obj*) 手続き

もし *obj* がベクタならば #t を返す。そうでなければ #f を返す。

(make-vector *k*) 手続き
(make-vector *k fill*) 手続き

k 個の要素からなる，一つの新たに割り当てられたベクタを返す。もしも第２引数が与えられたならば，各要素は *fill* に初期化される。そうでなければ各要素の初期内容は未規定である。

(vector *obj* ...) 手続き

与えられた引数を要素として持つ，一つの新たに割り当てられたベクタを返す。それは list に相当する。

```
(vector 'a 'b 'c)           ⟹  #(a b c)
```

(vector-length *vector*) 手続き

vector の中の要素の個数を正確整数として返す。

(vector-ref *vector k*) 手続き

k が *vector* の妥当な添字でなければエラーである。

vector-ref は *vector* の要素 *k* の内容を返す。

```
(vector-ref '#(1 1 2 3 5 8 13 21)
            5)
          ⟹  8
(vector-ref '#(1 1 2 3 5 8 13 21)
            (exact
             (round (* 2 (acos -1)))))
          ⟹ 13
```

(vector-set! *vector k obj*) 手続き

k が *vector* の妥当な添字でなければエラーである。

vector-set! は *vector* の要素 *k* に *obj* を格納する。

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec)
          ⟹  #(0 ("Sue" "Sue") "Anna")

(vector-set! '#(0 1 2) 1 "doe")
          ⟹  エラー  ; 定数ベクタ
```

(vector->list *vector*) 手続き
(vector->list *vector start*) 手続き
(vector->list *vector start end*) 手続き
(list->vector *list*) 手続き

vector->list 手続きは，*vector* の *start* と *end* の間を要素に含んでいる，一つの新たに割り当てられたオブジェクトのリストを返す。list->vector 手続きは，リスト *list* の要素で初期化された，一つの新たに生成されたベクタを返す。

どちらの手続きも，順序が保持される。

```
(vector->list '#(dah dah didah))
          ⟹  (dah dah didah)
(vector->list '#(dah dah didah) 1 2)
          ⟹ (dah)
(list->vector '(dididit dah))
          ⟹  #(dididit dah)
```

(vector->string *vector*) 手続き
(vector->string *vector start*) 手続き
(vector->string *vector start end*) 手続き
(string->vector *string*) 手続き
(string->vector *string start*) 手続き
(string->vector *string start end*) 手続き

vector の *start* と *end* の間の任意の要素が文字でなければエラーである。

vector->string 手続きは，*vector* の *start* と *end* の間を要素に含んでいる，一つの新たに割り当てられた文字列を返す。string->vector 手続きは，リスト *string* の *start* と *end* の間の要素で初期化された，一つの新たに生成されたベクタを返す。

どちらの手続きも，順序が保持される。

```
(string->vector "ABC")      ⟹   #(#\A #\B #\C)
(vector->string
  #(#\1 #\2 #\3)            ⟹ "123"
```

(vector-copy *vector*) 手続き
(vector-copy *vector start*) 手続き
(vector-copy *vector start end*) 手続き

vector の *start* と *end* の間に与えられた要素の一つの新たに割り当てられたコピーを返す。新しいベクタの要素は，古い要素と (eqv? の意味において) 同一である。

```
(define a #(1 8 2 8)) ; a は書換え不可能である
(define b (vector-copy a))
(vector-set! b 0 3)   ; b は書換え可能である
b                           ⟹ #(3 8 2 8)
(define c (vector-copy b 1 3))
c                           ⟹ #(8 2)
```

(vector-copy! *to at from*) 手続き
(vector-copy! *to at from start*) 手続き
(vector-copy! *to at from start end*) 手続き

at がゼロより小さいか，*to* の長さより大きければエラーである。(- (vector-length *to*) *at*) が (- *end start*) より小さい場合もエラーである。

ベクタ *from* の *start* と *end* 間の要素をベクタ *to*に，*at* から始めてコピーする。要素がコピーされる順序は，次の場合を除いて未規定である。もしコピー元とコピー先が重なっている場合は，コピー元は最初に一時ベクタにコピーした後，コピー先にコピーされているかのようにコピーが行われる。これは，このような状況において必ず正しい方向にコピーすることにより，記憶領域を割り当てることなく実現できる。

```
(define a (vector 1 2 3 4 5))
(define b (vector 10 20 30 40 50))
(vector-copy! b 1 a 0 2)
b                              ⟹ #(10 1 2 40 50)
```

(vector-append *vector* ...) 手続き

その要素が与えられたベクタ内の要素の連結であるような，一つの新たに割り当てられたベクタを返す。

```
(vector-append #(a b c) #(d e f))
          ⟹ #(a b c d e f)
```

(vector-fill! *vector fill*) 手続き
(vector-fill! *vector fill start*) 手続き
(vector-fill! *vector fill start end*) 手続き

vector-fill! 手続きは，*vector* 内の *start* と *end* の間の要素に *fill* を格納する。

```
(define a (vector 1 2 3 4 5))
(vector-fill! a 'smash 2 4)
a
          ⟹ #(1 2 smash smash 5)
```

6.9. バイトベクタ

バイトベクタは，バイナリデータのブロックを表す。それらはバイトの固定長配列で，バイトとは 0 から 255 までの範囲の正確整数である。バイトベクタは一般的に，同じ値を含んだベクタよりもよりスペース効率的である。

バイトベクタの**長さ** (*length*) とは，バイトベクタが含む要素の個数である。この個数は，バイトベクタが作成されるときに固定される非負整数である。バイトベクタの**妥当な添字** (*valid index*) は，バイトベクタの長さより小さい正確非負整数であり，バイトベクタと同様に添字はゼロから始まる。

バイトベクタは表記 #u8(*byte* ...) を使って書かれる。たとえば，長さが 3 であって，要素 0 にバイト 0 を，要素 1 にバイト 10 を，そして要素 2 にバイト 5 を含んでいるバイトベクタは下記のように書ける:

```
#u8(0 10 5)
```

バイトベクタ定数は自己評価しているので，プログラム中でクォートする必要はない。

(bytevector? *obj*) 手続き

もし *obj* がバイトベクタならば #t を返す。そうでなければ， #f が返される。

(make-bytevector *k*) 手続き
(make-bytevector *k byte*) 手続き

長さ *k* の，一つの新たに割り当てられたバイトベクタを返す。もしも *byte* が与えられたならば，すべての要素は *byte* で初期化され，そうでなければ各要素の内容は未規定である。

```
(make-bytevector 2 12)      ⟹ #u8(12 12)
```

(bytevector *byte* ...) 手続き

その引数を持つ，一つの新たに割り当てられたバイトベクタを返す。

```
(bytevector 1 3 5 1 3 5)    ⟹ #u8(1 3 5 1 3 5)
(bytevector)                ⟹ #u8()
```

(bytevector-length *bytevector*) 手続き

bytevector のバイト単位の長さを正確整数として返す。

(bytevector-u8-ref *bytevector k*) 手続き

k が *bytevector* の妥当な添字でなければエラーである。

bytevector の *k* 番目のバイトを返す。

```
(bytevector-u8-ref '#u8(1 1 2 3 5 8 13 21)
                   5)
          ⟹ 8
```

(bytevector-u8-set! *bytevector k byte*) 手続き

k が *bytevector* の妥当な添字でなければエラーである。

vector-set! は *bytevector* の *k* 番目のバイトに *byte* を格納する。

```
(let ((bv (bytevector 1 2 3 4)))
  (bytevector-u8-set! bv 1 3)
  bv)
          ⟹ #u8(1 3 3 4)
```

(bytevector-copy *bytevector*) 手続き
(bytevector-copy *bytevector start*) 手続き
(bytevector-copy *bytevector start end*) 手続き

bytevector の *start* と *end* の間のバイトを含む一つの新たに割り当てられたバイトベクタを返す。

```
(define a #u8(1 2 3 4 5))
(bytevector-copy a 2 4))    ⟹ #u8(3 4)
```

(bytevector-copy! *to at from*) 手続き
(bytevector-copy! *to at from start*) 手続き
(bytevector-copy! *to at from start end*) 手続き

at がゼロより小さいか，*to* の長さより大きければエラーである。(- (bytevector-length *to*) *at*) が (- *end start*) より小さい場合もエラーである。

バイトベクタ *from* の *start* と *end* 間のバイトをバイトベクタ *to*に，*at* から始めてコピーする。バイトがコピーされる順序は，次の場合を除いて未規定である。もしコピー元とコピー先が重なっている場合は，コピー元は最初に一時バイトベクタにコピーした後，コピー先にコピーされているかのようにコピーが行われる。これは，このような状況において必ず正しい方向にコピーすることにより，記憶領域を割り当てることなく実現できる。

```
(define a (bytevector 1 2 3 4 5))
(define b (bytevector 10 20 30 40 50))
(bytevector-copy! b 1 a 0 2)
b                           ⟹ #u8(10 1 2 40 50)
```

注: この手続きは R[6]RS に表れるが，Scheme における他のそのような手続きに反して，コピー先の前にコピー元を配置する。

(bytevector-append *bytevector* ...) 手続き

その要素が与えられたバイトベクタ内の要素の連結であるような，一つの新たに割り当てられたバイトベクタを返す。

```
(bytevector-append #u8(0 1 2) #u8(3 4 5))
          ⟹ #u8(0 1 2 3 4 5)
```

(utf8->string *bytevector*) 手続き
(utf8->string *bytevector start*) 手続き
(utf8->string *bytevector start end*) 手続き
(string->utf8 *string*) 手続き
(string->utf8 *string start*) 手続き
(string->utf8 *string start end*) 手続き

bytevector に無効な UTF-8 のバイト列を含むとエラーになる。

これらの手続きは，文字列と，UTF-8 エンコーディングを使用してこれらの文字列をエンコードしたバイトベクタとの間の変換を行う。utf8->string 手続きは *start* と *end* の間のバイトベクタのバイトをデコードし，対応する文字列を返す。string->utf8 手続きは *start* と *end* の間の文字列の文字をエンコードし，対応するバイトベクタを返す。

```
(utf8->string #u8(#x41))    ⟹ "A"
(string->utf8 "λ")          ⟹ #u8(#xCE #xBB)
```

6.10. 制御機能

本節はプログラム実行の流れを特殊な方法で制御する様々なプリミティブ手続きを記述する。手続きの引数を呼び出す本節の手続きは，常に元の手続きの呼び出しと同じ動的環境で行うこと。procedure? 述語もここで記述される。

(procedure? *obj*) 手続き

もし *obj* が手続きならば #t を返し，そうでなければ #f を返す。

```
(procedure? car)            ⟹  #t
(procedure? 'car)           ⟹  #f
(procedure? (lambda (x) (* x x)))
                            ⟹  #t
(procedure? '(lambda (x) (* x x)))
                            ⟹  #f
(call-with-current-continuation procedure?)
                            ⟹  #t
```

(apply *proc* arg_1 ... *args*) 手続き

proc を，リスト (append (list arg_1 ...) *args*) の各要素を各実引数として呼び出す。

```
(apply + (list 3 4))        ⟹  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)    ⟹  30
```

(map *proc* $list_1$ $list_2$ ...) 手続き

proc が *list* の個数と同じだけの個数の引数をとり，単一の値を返さなければエラーである。

map 手続きは *proc* を各 *list* の要素に要素ごとに適用し，その結果を順序どおりに並べたリストを返す。もしも複数の *list* が与えられ，それらがすべて同じ長さでなければ，map は最短のリストがなくなった時点で終了する。*list* は環状構造であってもよいが，それらのすべてが環状構造である場合はエラーである。*proc* が任意のリストを変異させることはエラーである。*proc* が各 *list* の要素に適用される動的な順序は未規定である。複数の戻り値が map から発生した場合，それ以前の戻り値によって返される値は変異しない。

```
(map cadr '((a b) (d e) (g h)))
          ⟹  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
          ⟹  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6 7)) ⟹  (5 7 9)
```

```
(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))                    ⟹  (1 2) または (2 1)
```

(string-map *proc* *string*$_1$ *string*$_2$ ...) 手続き

proc が *string* の個数と同じだけの個数の引数をとり，単一の値を返さなければエラーである。

string-map 手続きは *proc* を各 *string* の要素に要素ごとに適用し，その結果を順序どおりに並べた文字列を返す。もしも複数の *string* が与えられ，それらがすべて同じ長さでなければ，string-map は最短の文字列がなくなった時点で終了する。*proc* が各 *string* の要素に適用される動的な順序は未規定である。複数の戻り値が string-map から発生した場合，それ以前の戻り値によって返される値は変異しない。

```
(string-map char-foldcase "AbdEgH")
          ⟹  "abdegh"

(string-map
 (lambda (c)
   (integer->char (+ 1 (char->integer c))))
 "HAL")
          ⟹  "IBM"

(string-map
 (lambda (c k)
   ((if (eqv? k #\u) char-upcase char-downcase)
    c))
 "studlycaps xxx"
 "ululululul")
          ⟹   "StUdLyCaPs"
```

(vector-map *proc* *vector*$_1$ *vector*$_2$ ...) 手続き

proc が *vector* の個数と同じだけの個数の引数をとり，単一の値を返さなければエラーである。

vector-map 手続きは *proc* を各 *vector* の要素に要素ごとに適用し，その結果を順序どおりに並べたベクタを返す。もしも複数の *string* が与えられ，それらがすべて同じ長さでなければ，vector-map は最短のベクタがなくなった時点で終了する。*proc* が各 *string* の要素に適用される動的な順序は未規定である。複数の戻り値が vector-map から発生した場合，それ以前の戻り値によって返される値は変異しない。

```
(vector-map cadr '#((a b) (d e) (g h)))
          ⟹  #(b e h)

(vector-map (lambda (n) (expt n n))
            '#(1 2 3 4 5))
          ⟹  #(1 4 27 256 3125)

(vector-map + '#(1 2 3) '#(4 5 6 7))
          ⟹  #(5 7 9)

(let ((count 0))
  (vector-map
   (lambda (ignored)
     (set! count (+ count 1))
     count)
   '#(a b)))                   ⟹  #(1 2) or #(2 1)
```

(for-each *proc* *list*$_1$ *list*$_2$ ...) 手続き

proc が *list* の個数と同じだけの個数の引数をとらなければ，エラーである。

for-each の引数は map の引数と同様だが，for-each は *proc* をその値ではなくその副作用を求めて呼び出す。map と異なり，for-each は *proc* を各 *list* の要素に対して最初の要素から最後の要素へという順序で呼び出すことが保証されており，かつ for-each が返す値は未規定である。もしも複数の *list* が与えられ，それらがすべて同じ長さでなければ，for-each は最短のリストがなくなった時点で終了する。*list* は環状構造であってもよいが，それらのすべてが環状構造である場合はエラーである。

proc が任意のリストを変異させることはエラーである。

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                           ⟹  #(0 1 4 9 16)
```

(string-for-each *proc* *string*$_1$ *string*$_2$ ...) 手続き

proc が *string* の個数と同じだけの個数の引数をとらなければ，エラーである。

string-for-each の引数は string-map の引数と同様だが，string-for-each は *proc* をその値ではなくその副作用を求めて呼び出す。string-map と異なり，string-for-each は *proc* を各 *string* の要素に対して最初の要素から最後の要素へという順序で呼び出すことが保証されており，かつ string-for-each が返す値は未規定である。もしも複数の *string* が与えられ，それらがすべて同じ長さでなければ，string-for-each は最短のリストがなくなった時点で終了する。*proc* が任意のリストを変異させることはエラーである。

```
(let ((v '()))
  (string-for-each
   (lambda (c) (set! v (cons (char->integer c) v)))
   "abcde")
  v)                           ⟹  (101 100 99 98 97)
```

(vector-for-each *proc* *vector*$_1$ *vector*$_2$ ...) 手続き

proc が *vector* の個数と同じだけの個数の引数をとらなければ，エラーである。

vector-for-each の引数は vector-map の引数と同様だが，vector-for-each は *proc* をその値を求めてではなくその副作用を求めて呼び出す。vector-map と異なり，vector-for-each は *proc* を各 *vector* の要素に対して最初の要素から最後の要素へという順序で呼び出すことが保

証されており，かつ vector-for-each が返す値は未規定である。もしも複数の *vector* が与えられ，それらがすべて同じ長さでなければ，vector-for-each は最短のリストがなくなった時点で終了する。*proc* が任意のリストを変異させることはエラーである。

```
(let ((v (make-list 5)))
  (vector-for-each
   (lambda (i) (list-set! v i (* i i)))
   '#(0 1 2 3 4))
  v)                          ⟹  (0 1 4 9 16)
```

(call-with-current-continuation *proc*)　　手続き
(call/cc *proc*)　　手続き

proc が 1 引数の手続きでなければエラーである。

手続き call-with-current-continuation (あるいはそれと同等の省略形 call/cc) は，現在の継続 (下記の根拠を見よ) を "脱出手続き" (escape procedure) としてパッケージし，それを *proc* に引数として渡す。脱出手続きとは，もし後でこれを呼び出すと，その時たとえどんな継続が有効であっても捨て去り，そのかわりに脱出手続きが作成された時に有効だった継続を使うことになる Scheme 手続きである。

脱出手続きの呼出しは，dynamic-wind を使ってインストールされた *before* および *after* サンクの起動を発生してもよい。脱出手続きは，call-with-current-continuation へのもともとの呼出しに対する継続と同じだけの個数の引数を受理する。ほとんどの継続はただ一つの値をとる。call-with-values 手続き (define-values, let-values, および let*-values 式の初期化式を含む) が作成した継続は，消費者が期待する値の数を受け取る。lambda, case-lambda, begin, let, let*, letrec, letrec*, let-values, let*-values, let-syntax, letrec-syntax, parameterize, guard, case, cond, when, および unless のような，式のシーケンス内のすべての非終端式の継続は任意のイベントでそれらに渡した値を破棄しているので，任意の個数の値を取る。これらのいずれかの方法で作成したのではない継続へ，ゼロ個または複数個の値を渡すことの効果は未規定である。

proc へ渡される脱出手続きは，Scheme の他のすべての手続きと同じように無期限の寿命をもつ。脱出手続きを変数やデータ構造の中に格納して，何回でも望むだけの回数呼び出しができる。しかし，raise や error 手続きのように，その呼び出し元に戻ることはない。

下記の例は call-with-current-continuation を使う最もシンプルな方法だけを示している。もしも，実際の用法がすべて，これらの例と同じように単純だったならば，call-with-current-continuation のような強力さを備えた手続きはなんら必要なかっただろう。

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                      ⟹  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                  (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj)
                           (+ (r (cdr obj)) 1))
                          (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))    ⟹  4

(list-length '(a b . c))    ⟹  #f
```

根拠:

call-with-current-continuation の一般的な用途は，ループまたは手続き本体からの構造化された非局所的脱出である。しかし実のところ，call-with-current-continuation は広範囲の高度な制御構造を実装することにも有用である。実際には，raise と guard は非ローカルの脱出のためにより構造化されたメカニズムを提供する。

一つの Scheme 式が評価される時は常に，その式の結果を欲している一つの**継続** (*continuation*) が存在する。継続は，計算に対する (デフォルトの) 未来全体を表現する。たとえば，もしも式が REPL で評価されるならば，そのとき継続は結果を受け取り，それを画面に印字し，次の入力を催促し，それを評価し，等々を永遠に続けることだろう。たいていの場合，継続は，ユーザのコードが規定する動作を含んでいる。たとえば，結果を受け取り，あるローカル変数に格納されている値でそれを乗算し，7 を加算し，その答えを REPL の継続に与えて印字させる，という継続におけるようにである。通常，これらの遍在する継続は舞台裏に隠されており，プログラマはそれについてあまり考えない。しかし，まれにはプログラマが継続を明示的に取り扱う必要がある。call-with-current-continuation 手続きは， Scheme プログラマが現在の継続と同じようにはたらく手続きを作成することを可能にする。

(values *obj* ...)　　手続き

その継続に (どれだけの個数であれ) 引数をすべて引き渡す。values 手続きは次のように定義され得る:

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

(call-with-values *producer consumer*)　　手続き

producer 引数を引数なしで呼び出し，そしてある継続を呼び出す。この継続は，(複数個の) 値が渡されると，それらの値を引数として *consumer* 手続きを呼び出す。*consumer* への呼出しに対する継続は，call-with-values への呼出しの継続である。

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                            ⟹  5

(call-with-values * -)      ⟹  -1
```

(dynamic-wind *before thunk after*) 手続き

thunk を引数なしで呼び出し，その呼出しの (1 個または複数個の) 結果を返す。*before* と *after* は，これもまた引数なしで，次の規則によって要求されるとおりに呼び出される。call-with-current-continuation を使って取り込まれた継続への呼出しというものがなければ，この三つの引数はそれぞれ 1 回ずつ順に呼び出されることに注意せよ。*before* は，いつであれ実行が *thunk* への呼出しの動的寿命に入る時に呼び出され，*after* は，いつであれその動的寿命が終わる時に呼び出される。手続き呼出しの動的寿命 (dynamic extent) とは，呼出しが開始された時から呼出しが戻る時までの期間である。*before* と *after* サンクは，dynamic-wind への呼び出しと同じ動的環境で呼び出される。Scheme では，call-with-current-continuation があるため，呼出しの動的寿命は必ずしも単一の，連続した時間の期間ではない。それは次のように定義される:

- 呼び出された手続きの本体が始まる時，その動的寿命に入る。
- (call-with-current-continuation を使って) 動的寿命の期間中に取り込まれた継続が，実行がその動的寿命の中にないときに起動されたならば，その時もまた，その動的寿命に入る。
- 呼び出された手続きが戻るときに終了する。
- 実行がその動的寿命の中にあり，動的寿命の期間外に取り込まれた継続が呼び出されるときときにも終了する。

もしも dynamic-wind への二度目の呼出しが，*thunk* への呼出しの動的寿命の期間中に出現し，そしてこれら二つの dynamic-wind の起動に由来する二つの *after* がどちらも呼び出されるように継続が起動されたならば，そのときは，dynamic-wind への二度目の (内側の) 呼出しに対応する *after* の方が最初に呼び出される。

もしも dynamic-wind への二度目の呼出しが，*thunk* への呼出しの動的寿命の期間中に出現し，そしてこれら二つの dynamic-wind の起動に由来する二つの *before* がどちらも呼び出されるように継続が起動されたならば，そのときは，dynamic-wind への一度目の (外側の) 呼出しに対応する *before* の方が最初に呼び出される。

もしも，ある継続の呼出しが，dynamic-wind への一つの呼出しに由来する *before* と，もう一つの呼出しに由来する *after* の，二つの呼出しを要求するならば，そのときは，*after* の方が最初に呼び出される。

取り込まれた継続を使って，*before* または *after* への呼出しの動的寿命に入るまたはそれを終えることの効果は，未規定である。

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

      ⟹ (connect talk1 disconnect
          connect talk2 disconnect)
```

6.11. 例外

本節では，Scheme の例外処理および例外発生手続きについて説明する。Scheme の例外の概念については，1.3.2 節を参照のこと。ガード構文のための 4.2.7 についても参照のこと。

例外ハンドラ (*exception handler*) は，例外的な状況が通知されたときにプログラムが実行するアクションを決定する引数が 1 個の手続きである。システムは，暗黙的に動的環境における現在の例外ハンドラを保持している。

プログラムは例外に関する情報をカプセル化したオブジェクトを渡し，現在の例外ハンドラを呼び出すことによって例外を発生させる。1 個の引数を受け取る任意の手続きでは例外ハンドラとして機能することができ，任意のオブジェクトは例外を表すために使用することができる。

(with-exception-handler *handler thunk*) 手続き

handler が 1 個の引数を受け付けない場合はエラーである。*thunk* が 0 個の引数を受け入れない場合もエラーである。

with-exception-handler 手続きは *thunk* を呼び出した結果を返す。*handler* は， *thunk* の呼び出しに使用される，動的環境における現在の例外ハンドラとしてインストールされている。

```
(call-with-current-continuation
 (lambda (k)
  (with-exception-handler
   (lambda (x)
    (display "condition: ")
    (write x)
    (newline)
    (k 'exception))
   (lambda ()
    (+ 1 (raise 'an-error))))))
                              ⟹ exception
          and prints  condition: an-error
```

```
(with-exception-handler
 (lambda (x)
  (display "something went wrong\n"))
 (lambda ()
  (+ 1 (raise 'an-error))))
          prints  something went wrong
```

印字の後，二番目の例は別の例外が発生する。

(raise *obj*) 手続き

obj で現在の例外ハンドラを呼び出すことによって例外を発生させる。ハンドラは，現在の例外ハンドラが呼び出されているハンドラが，インストールされたときに所定の位置にあったものであることを除いて，raise への呼び出しとして同じ動的環境で呼び出される。ハンドラが返された場合，二次例外がハンドラとして同じ動的環境で発生する。*obj* と二次例外によって発生させたオブジェクトとの関係は未規定である。

(raise-continuable *obj*) 手続き

obj で現在の例外ハンドラを呼び出すことによって例外を発生させる。ハンドラは以下を除いて，raise-continuable への呼び出しとして同じ動的環境で呼び出される: (1) 現在の例外ハンドラは，呼び出されたハンドラをインストールしたときに，それは所定の位置にあったものである。(2) 呼び出されたハンドラが返された場合，それは再び現在の例外ハンドラになる。ハンドラが戻ると，それが返す値は raise-continuable への呼び出しによって返された値にになる。

```
(with-exception-handler
  (lambda (con)
    (cond
      ((string? con)
       (display con))
      (else
       (display "a warning has been issued")))
    42)
  (lambda ()
    (+ (raise-continuable "should be a number")
       23)))
    prints: should be a number
                              ⟹ 65
```

(error *message obj* ...) 手続き

message は文字列でなければならない。

新たに割り当てられた処理系定義のオブジェクト上で raise を呼び出すかのように例外を発生させ，このオブジェクトは任意の *obj* 同様，刺激物 (*irritants*) として知られている *message* によって提供される情報をカプセル化している。error-object? 手続きは，このようなオブジェクト上で #t を返さなければならない。

```
(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else
          (error
            "null-list?: argument out of domain"
            l))))
```

(error-object? *obj*) 手続き

もし *obj* が error で作成されたオブジェクト，またはオブジェクトの実装定義セットの一つのいずれかの場合は #t を返し，それ以外の場合は #f を返す。オブジェクトはエラーを通知するために使われ，述語 file-error? および read-error? を満たすものまたは error-object? を満たさないものを含んでいる。

(error-object-message *error-object*) 手続き

error-object によってカプセル化されたメッセージを返す。

(error-object-irritants *error-object*) 手続き

error-object によってカプセル化された刺激物のリストを返す。

(read-error? *obj*) 手続き
(file-error? *obj*) 手続き

エラー型述語。obj が read 手続きよって発生したオブジェクト，またはファイルの入力または出力ポートを開けないことによって発生したオブジェクトのいずれかであれば #t を返し，それ以外の場合は #f を返す。

6.12. 環境および評価

(environment *list*$_1$ ...) eval ライブラリ手続き

この手続きでは，各 *list* をインポートセットとして考え，空の環境から開始してそこにインポートすることで得られる環境の指定子を返す。(インポートセットの説明については 5.6 を参照。) 指定子によって表される環境の束縛は，環境自体はそのままで不変である。

(scheme-report-environment *version*)
r5rs ライブラリ手続き

version が 5 と等しければ，R^{5}RS に従い，scheme-report-environment は R^{5}RS ライブラリで定義されている束縛のみを含む環境の指定子 (specifier) を返す。実装は *version* の値をサポートしなければならない。

報告書の指定バージョンに対応する束縛を含む環境の指定子を返す場合，実装はまた， *version* の他の値をサポートしてもよい。もしも *version* が 5 でもなく，実装によって

サポートされている他の値でもなければ，エラーが通知される。

scheme-report-environment において束縛されている識別子 (たとえば car) へ，(eval を使って) 定義あるいは代入をすることの効果は未規定である。したがって，それが含む環境および束縛は，書換え不可能であってもよい。

(null-environment *version*)　　r5rs ライブラリ手続き

version が 5 と等しければ，R^{5}RS に従い，null-environment 手続きは R^{5}RS ライブラリで定義されているすべての構文キーワードの束縛のみを含む環境の指定子を返す。実装は *version* の値をサポートしなければならない。

報告書の指定バージョンに対応する適切な束縛を含む環境の指定子を返す場合，実装はまた， *version* の他の値をサポートしてもよい。もしも *version* が 5 でもなく，実装によってサポートされている他の値でもなければ，エラーが通知される。

scheme-report-environment において束縛されている識別子 (たとえば car) へ，(eval を使って) 定義あるいは代入をすることの効果は未規定である。したがって，それが含む環境および束縛は，書換え不可能であってもよい。

(interaction-environment)　　repl ライブラリ手続き

この手続きは，実装定義の束縛集合を含む不変環境—一般的には (scheme base) によってエクスポートされた環境の，あるスーパーセット—の指定子を返す。その意図は，ユーザによって REPL に入力された式を実装が評価するときの環境を，この手続きが返すことである。

(eval *expr-or-def environment-specifier*)
eval ライブラリ手続き

expr-or-def が式である場合は，指定された環境において評価し，その値を返す。それが定義である場合は，指定された識別子は，指定された環境において定義され，提供される環境は不変ではない。実装は，他のオブジェクトを許可するように eval を拡張してもよい。

```
(eval '(* 7 3) (environment '(scheme base)))
                                ⟹  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                ⟹  20
(eval '(define foo 32)
      (environment '(scheme base)))
                                ⟹  error is signaled
```

6.13. 入出力

6.13.1. ポート

ポート (port) は，入力 (input) と出力 (output) の装置を表現する。Scheme にとって，入力ポートとはデータをコマンドへ配送できる Scheme オブジェクトであり，出力ポートとはデータを受理できる Scheme オブジェクトである。 入力と出力のポート型が互いに素であるかどうかは実装依存である。

異なる**ポート型**は異なるデータを操作する。Scheme 実装は，**テキストポート** と **バイナリポート** をサポートすることが必須であるが，他のポート型も提供してよい。

テキストポートは，以下の read-char と write-char を用いて文字を含んだ補助記憶との個々の文字の読み書きをサポートし，read と write のような文字単位で定義された操作をサポートする。

バイナリポートは，以下の read-u8 と write-u8 を用いてバイトを含んだ補助記憶との個々のバイトの読み書きを，バイト単位で定義された操作と同様にサポートする。テキストおよびバイナリのポート型が互いに素であるかどうかは，実装依存である。

ポートは，ファイル，デバイス，および Scheme プログラムが動作しているホストシステム上の同様のものへアクセスするために使用することができる。

(call-with-port *port proc*)　　手続き

proc が 1 個の引数を受け取らなければエラーである。

call-with-port 手続きは *port* を引数として受け取り，*proc* を呼び出す。*proc* が戻るときには，ポートが自動的に閉じられ，そして *proc* がもたらした値が返される。もしも *proc* が戻らないならば，ポートは，それが read または write 演算に二度と再び使われないと証明できない限り，自動的に閉じられることはない。

根拠: Scheme の脱出手続きは無期限の寿命をもつから，現在の継続から脱出した後でも，再開することができる。もしも実装が，現在の継続からのどの脱出においてポートを閉じることにしてもよかったとしたら，call-with-current-continuation と call-with-port の両方を使用して移植可能なコードを書くことは不可能であろう。

(call-with-input-file *string proc*)
file ライブラリ手続き
(call-with-output-file *string proc*)
file ライブラリ手続き

proc が 1 個の引数を受け取らなければエラーである。

これらの手続きは，open-input-file または open-output-file であるかのように名前のファイルを入力または出力のために開くことでによって得られたテキストポート取得する。ポートと *proc* はこのとき，call-with-port と等価な手続きに渡される。

(input-port? *obj*) 手続き
(output-port? *obj*) 手続き
(textual-port? *obj*) 手続き
(binary-port? *obj*) 手続き
(port? *obj*) 手続き

これらの手続きは，それぞれ，もし *obj* が入力ポート，出力ポート，テキストポート，バイナリポート，または任意の種類のポートの場合は `#t` を返し，そうでなければ `#f` を返す。

(input-port-open? *port*) 手続き
(output-port-open? *port*) 手続き

port がまだに開いていて，それぞれ入力または出力を行うことができる場合は `#t` を返し，そうでなければ `#f` を返す。

(current-input-port) 手続き
(current-output-port) 手続き
(current-error-port) 手続き

それぞれ，現在のデフォルト入力ポート，出力ポート，またはエラーポート (出力ポート) を返す。これらの手続きは，`parameterize` (4.2.6) 節参照) で上書き可能なパラメータオブジェクトである。これらの初期束縛は，実装定義のテキストポートである。

(with-input-from-file *string thunk*)
file ライブラリ手続き
(with-output-to-file *string thunk*)
file ライブラリ手続き

`open-input-file` または `open-output-file` のように，入力または出力のためにファイルが開かれ，`current-input-port` または `current-output-port` で返される (`(read)`，`(write` *obj*`)` などで使われる) 新しいポートが値とされている。*thunk* はこのとき，引数なしで呼び出される。*thunk* が戻るときには，ポートが閉じられて以前のデフォルトが回復される。*thunk* は，引数をとらない手続きでなければエラーである。どちらの手続きも，*thunk* がもたらした値を返す。もしも脱出手続きが，これらの手続きの継続から脱出するために使われる場合は，これらは現在の入力または出力ポートが `parameterize` で動的に束縛されていたかのように正確に振舞う。

(open-input-file *string*) file ライブラリ手続き
(open-binary-input-file *string*) file ライブラリ手続き

既存のファイルの *string* をとり，そのファイルからのデータを配送できるテキスト入力ポートまたはバイナリ入力ポートを返す。ファイルが存在しないか，開くことができない場合，`file-error?` を満たすエラーが通知される。

(open-output-file *string*) file ライブラリ手続き
(open-binary-output-file *string*)
file ライブラリ手続き

作成される出力ファイルにつける名前の *string* をとり，その名前をで新しいファイルにデータを書き込むことが可能なテキスト出力ポートまたはバイナリ出力ポートを返す。与えられた名前をのファイルがすでに存在する場合，その効果は未規定である。ファイルを開くことができない場合，`file-error?` を満たすエラーが通知される。

(close-port *port*) 手続き
(close-input-port *port*) 手続き
(close-output-port *port*) 手続き

port に関連付けられたしたリソースを閉じて，*port* がデータを配送または受理できないようにする。入力または出力ではないポートを最後の二つの手続きにそれぞれ適用することは，エラーである。Scheme 実装は，ソケットのような，同時に入力と出力であるポートを提供してもよい。`close-input-port` および `close-output-port` 手続きはこのとき，ポートの入力側と出力側を独立して閉じるのに使うことができる。

これらのルーチンは，もしもファイルが既に閉じられているならば，なんの効果もない。

(open-input-string *string*) 手続き

文字列をとり，文字列から文字を配送するテキスト入力ポートを返す。文字が変更された場合，その効果は未規定である。

(open-output-string) 手続き

`get-output-string` で検索のための文字を蓄積するテキスト出力ポートを返す。

(get-output-string *port*) 手続き

port が `open-output-string` によって作られていない場合，エラーである。

それらが出力された順に，これまでのポートに出力された文字で構成される文字列を返す。結果の文字が変更された場合，その効果は未規定である。

```
(parameterize
    ((current-output-port
      (open-output-string)))
    (display "piece")
    (display " by piece ")
    (display "by piece.")
    (newline)
    (get-output-string (current-output-port)))

        ⟹ "piece by piece by piece.\n"
```

(open-input-bytevector *bytevector*) 手続き

バイトベクタをとり，バイトベクタからバイトを配送するバイナリ入力ポートを返す。

(open-output-bytevector) 手続き

`get-output-bytevector` による検索のためにバイトを蓄積したバイナリ出力ポートを返す。

(get-output-bytevector *port*) 手続き

port が open-output-bytevector によって作られていない場合，エラーである。

それらが出力された順に，これまでのポートに出力されたバイトで構成されるバイトベクタを返す。

6.13.2. 入力

port が任意の入力手続きで省略された場合，(current-input-port) の返す値をデフォルトとする。閉じているポートに対して入力操作を試みた場合はエラーである。

(read) read ライブラリ手続き
(read *port*) read ライブラリ手続き

read 手続きは， Scheme オブジェクトの外部表現をオブジェクト自身へと変換する。つまり，これは，非終端記号 <データ> に対するパーサである (7.1.2 節と 6.4 節を参照)。これは，与えられたテキスト入力 *port* から次にパース可能なオブジェクトを返し，そしてそのオブジェクトの外部表現の終わりを過ぎて最初の文字を指すように *port* を更新する。

実装は，レコード型やデータ表現をもたない他の型を表すために拡張構文をサポートしてもよい。

オブジェクトを開始できる任意の文字が見つかる前に入力でファイル終端に遭遇した場合は，end-of-file オブジェクトが返される。ポートは開いたままであり，更に読み込みを試みた場合もまた end-of-file オブジェクトを返す。オブジェクトの外部表現の開始の後でファイル終端に遭遇したが，その外部表現が不完全なため解析可能でない場合，read-error? を満たすエラーが通知される。

(read-char) 手続き
(read-char *port*) 手続き

テキスト入力 *port* から次に入手可能な文字を返し，その次に来る文字を指すように *port* を更新する。それ以上文字が入手可能でない場合，end-of-file オブジェクトが返される。

(peek-char) 手続き
(peek-char *port*) 手続き

テキスト入力 *port* から次に入手可能な文字を返すが，その次に来る文字を指すように *port* を更新することは**しない**。それ以上文字が入手可能でない場合，end-of-file オブジェクトが返される。

注: peek-char を呼び出すと，それと同じ *port* を引数とする read-char への呼出しが返したはずの値と，同じ値が返される。唯一の違いは，次回その *port* で read-char や peek-char を呼び出すと，それに先行する peek-char への呼出しが返した値が返される，という点である。とりわけ，対話的ポートでの peek-char への呼出しは，そのポートでの read-char への呼出しが入力待ちでハングすることになるときは常にハングするだろう。

(read-line) 手続き
(read-line *port*) 手続き

テキスト入力 *port* から次に入手可能なテキストの次の行を返し，その次に来る文字を指すように *port* を更新する。行末が読み込まれた場合，行末 (は含まない) までのテキストすべてを含む文字列が返され，ポートが行末の直後を指すように更新される。任意の行末が読み込まれる前にファイル終端に遭遇した場合，いくつかの文字が読み込まれていれば，それらの文字を含む文字列が返される。任意の文字が読み込まれる前にファイル終端に遭遇した場合， end-of-file オブジェクトが返される。この手続きのために，行末は，改行文字，復改文字，または復改文字の後に続く改行文字の列からなる。実装はまた，他の行末文字または列を認識してもよい。

(eof-object? *obj*) 手続き

もし *obj* が end-of-file オブジェクトならば #t を返し，そうでなければ #f を返す。end-of-file オブジェクトの正確な集合は実装によって異なるだろうが，いずれにせよ，どの end-of-file オブジェクトも，read を使って読むことのできるオブジェクトであることは決してない。

(eof-object) 手続き

必ずしも一意ではない， end-of-file オブジェクトを返す。

(char-ready?) 手続き
(char-ready? *port*) 手続き

テキスト入力 *port* で文字が準備できている場合 #t を返し，そうでなければ #f を返す。char-ready? が #t を返す場合，そのときその *port* での次回の read-char 演算はハングしないと保証されている。*port* がファイル終端にある場合， char-ready? は #t を返す。

根拠: char-ready? 手続きは，プログラムが入力待ちで止まってしまうことなく対話的ポートから文字を受理できるようにするために存在する。なんであれそのようなポートに結合している入力エディタは，いったん char-ready? によって存在を表明された文字が入力から削除され得ないことを，保証しなければならない。もし仮に char-ready? がファイル終端で #t を返すとしたならば，ファイル終端にあるポートは，文字が準備できていない対話的ポートと見分けがつかなくなってしまったことだろう。

(read-string *k*) 手続き
(read-string *k port*) 手続き

次の *k* 文字，またはファイル終端までの入手可能な限りの文字をテキスト入力 *port* から読み，新たに割り当てられた文字列に左から右へと順に格納し，その文字列を返す。ファイル終端までに入手可能な文字がなければ， end-of-file オブジェクトが返される。

(read-u8) 手続き
(read-u8 *port*) 手続き

バイナリ入力 *port* から次に入手可能なバイトを返し，その次に来るバイトを指すように *port* を更新する。それ以上バ

イトが入手可能でない場合， end-of-file オブジェクトが返される。

(peek-u8) 手続き
(peek-u8 *port*) 手続き

バイナリ入力 *port* から次に入手可能なバイトを返すが，その次に来るバイトを指すように *port* を更新することは**しない**。それ以上バイトが入手可能でない場合， end-of-file オブジェクトが返される。

(u8-ready?) 手続き
(u8-ready? *port*) 手続き

バイナリ入力 *port* でバイトが準備できている場合 #t を返し，そうでなければ #f を返す。u8-ready? が #t を返す場合，そのときその *port* での次回の read-u8 演算はハングしないと保証されている。*port* がファイル終端にある場合，u8-ready? は #t を返す。

(read-bytevector *k*) 手続き
(read-bytevector *k port*) 手続き

次の *k* バイト，またはファイル終端までの入手可能な限りのバイトをバイナリ入力 *port* から読み，新たに割り当てられたバイトベクタに左から右へと順に格納し，そのバイトベクタを返す。ファイル終端までに入手可能なバイトがなければ， end-of-file オブジェクトが返される。

(read-bytevector! *bytevector*) 手続き
(read-bytevector! *bytevector port*) 手続き
(read-bytevector! *bytevector port start*) 手続き
(read-bytevector! *bytevector port start end*) 手続き

次の *end*−*start* バイト，またはファイル終端までの入手可能な限りのバイトを バイナリ入力 *port* から読み，*bytevector* に *start* から始めて左から右へと順に格納する。*end* が指定されていない場合， *bytevector* の終端に達するまで読む。*start* が指定されていない場合，位置 0 から読み始める。読んだバイト数を返す。入手可能なバイトがなければ，end-of-file オブジェクトが返される。

6.13.3. 出力

port が任意の出力手続きで省略された場合，(current-output-port) の返す値をデフォルトとする。閉じているポートに対して出力操作を試みた場合はエラーである。

(write *obj*) write ライブラリ手続き
(write *obj port*) write ライブラリ手続き

obj の表現を，与えられたテキスト出力 *port* に書く。この表記表現の中に現れる文字列は引用符に囲まれ，そしてその文字列の内部では逆スラッシュと引用符文字は逆スラッシュでエスケープされる。非 ASCII 文字を含むシンボルは，縦線で囲まれる。文字オブジェクトは #\ 表記を使って書かれる。

obj が，通常の記述表現を使って無限ループの原因となるであろう環状を含む場合，少なくとも環状の一部を構成するオブジェクトは，2.4 節で説明されているようなデータラベルを使って表されなければならない。データラベルは，環状が無ければ使ってはならない。

実装は，レコード型やデータ表現をもたない他の型を表すために拡張構文をサポートしてもよい。

write 手続きは未規定の値を返す。

(write-shared *obj*) write ライブラリ手続き
(write-shared *obj port*) write ライブラリ手続き

write-shared 手続きは，共有構造が出力に一回以上現れるすべてのペアとベクタがデータラベルを使って表さなければならないことを除いて， write と同じである。

(write-simple *obj*) write ライブラリ手続き
(write-simple *obj port*) write ライブラリ手続き

write-simple 手続きは，共有構造がデータラベルを使って表されることがないという点を除いて， write と同じである。これは， *obj* が環状構造を含む場合は，終了しない write-simple を起こすことができる。

(display *obj*) write ライブラリ手続き
(display *obj port*) write ライブラリ手続き

obj の表現を，与えられたテキスト出力 *port* に書く。表記表現の中に現れる文字列はあたかも write ではなく write-string で書かれたかのように出力される。シンボルはエスケープされない。文字オブジェクトはこの表現の中にあたかも write ではなく write-char で書かれたかのように現れる。

他のオブジェクトの display 表現は，未規定である。しかし， display は自己参照のペア，ベクタ，レコードで無限ループしてはならない。したがって，通常の write 表現が使われた場合，データラベルが write のように環状を表現するのに必要とされる。

実装は，レコード型やデータ表現をもたない他の型を表すために拡張構文をサポートしてもよい。

display 手続きは未規定の値を返す。

根拠: write 手続きは機械が読み取り可能な出力を生成するためのものあり，display は人間が読み取り可能な出力を生成するためのものである。

(newline) 手続き
(newline *port*) 手続き

行末をテキスト出力 *port* に書く。正確なところ，これがどのようになされるのかはオペレーティングシステムごとに異なる。未規定の値を返す。

(write-char *char*) 手続き
(write-char *char port*) 手続き

文字 *char* (文字の外部表現ではなく文字自体) を，与えられたテキスト出力 *port* に書き，未規定の値を返す。

(write-string *string*) 手続き
(write-string *string port*) 手続き
(write-string *string port start*) 手続き
(write-string *string port start end*) 手続き

string の文字を，*start* から *end* まで左から右へ順に，テキスト出力 *port* に書く。

(write-u8 *byte*) 手続き
(write-u8 *byte port*) 手続き

byte を与えられたバイナリ出力 *port* に書き，未規定の値を返す。

(write-bytevector *bytevector*) 手続き
(write-bytevector *bytevector port*) 手続き
(write-bytevector *bytevector port start*) 手続き
(write-bytevector *bytevector port start end*) 手続き

bytevector のバイトを，*start* から *end* まで右から左へ順に，バイナリ出力 *port* に書く。

(flush-output-port) 手続き
(flush-output-port *port*) 手続き

出力ポートのバッファから元になるファイルまたはデバイスへバッファされた出力をフラッシュし，未規定の値を返す。

6.14. システムインタフェース

システムインタフェースの問題は一般に本報告書の範囲を越えている。しかし，下記の演算はここに記述するだけの重要性がある。

(load *filename*) load ライブラリ手続き
(load *filename environment-specifier*) load ライブラリ手続き

filename が文字列でない場合はエラーである。

実装依存の操作は，*filename* を Scheme のソースコードを含む既存のファイル名に変換するために使用されている。load 手続きは，*environment-specifier*で指定された環境内でそのファイルから式と定義を読んで逐次的に評価する。もし *environment-specifier* が省略された場合，(interaction-environment) とみなされる。

式の結果が印字されるかどうかは未規定である。load 手続きは current-input-port と current-output-port のどちらの返す値にも影響をおよぼさない。それは未規定の値を返す。

根拠: 互換性のため，load はソースファイルについて演算しなければならない。他の種類のファイルについての load 演算は必然的に実装間で異なる。

(file-exists? *filename*) file ライブラリ手続き

filename が文字列でない場合はエラーである。

file-exists? 手続きは，もし手続きが呼び出された時点でその名前のファイルが存在する場合は #t を返し，そうでない場合は #f を返す。

(delete-file *filename*) file ライブラリ手続き

filename が文字列でない場合はエラーである。

delete-file 手続きは，その名前のファイルが存在し，かつ削除可能な場合は削除し，その戻り値は未規定である。ファイルが存在しない，または削除できない場合は，file-error? を満たすエラーが通知される。

(command-line) process-context ライブラリ手続き

文字列のリストとしてプロセスに渡されたコマンドラインを返す。最初の文字列はコマンド名に対応し，実装依存である。これらの文字列のいずれかを変異させることはエラーである。

(exit) process-context ライブラリ手続き
(exit *obj*) process-context ライブラリ手続き

すべての未解決の dynamic-wind *after* 手続きを実行し，実行中のプログラムを終了し，オペレーティングシステムに終了値を通信する。引数が指定されていない，または *obj* が #t の場合，exit 手続きはプログラムが正常終了したことをオペレーティングシステムに通信する必要がある。もし *obj* が #f の場合は，exit 手続きはプログラムが異常終了したことをオペレーティングシステムに通信する必要がある。そうでない場合は，可能であれば exit は *obj* をオペレーティングシステムの適切な終了値に変換する必要がある。

exit 手続きは，例外通知または継続への返却をしてはならない。

注: ハンドラを実行するための要件のためであって，この手続きは単なるオペレーティングシステムの終了処理ではない。

(emergency-exit) process-context ライブラリ手続き
(emergency-exit *obj*) process-context ライブラリ手続き

すべての未解決の dynamic-wind *after* 手続きを実行せずにプログラムを終了し，exit と同様にオペレーティングシステムに終了値を通信する。

注: emergency-exit 手続きは Windows および Posix の _exit に相当する。

`(get-environment-variable` *name*`)`
process-context ライブラリ手続き

多くのオペレーティングシステムは，環境変数 (*environment variables*) で構成される環境 (*environment*) で実行中の各プロセスを提供する。(この環境は `eval` に渡すことができる Scheme 環境と混同してはならない。6.12 節を参照のこと。) 環境変数の名前および値は両方とも文字列である。`get-environment-variable` 手続きは，環境変数 *name* の値を返し，その名前の環境変数が見つからない場合は `#f` を返す。名前をエンコードしたり環境変数の値をデコードしたりするのにロケール情報を使用してもよい。`get-environment-variable` がデコードできない場合はエラーである。また，結果の文字列を変異させることもエラーである。

```
(get-environment-variable "PATH")
          ⟹ "/usr/local/bin:/usr/bin:/bin"
```

`(get-environment-variables)`
process-context ライブラリ手続き

すべての環境変数の名前と値を連想リストとして返す。各エントリの car は環境変数の名前で， cdr は値であり，いずれも文字列である。リストの順序は規定されていない。これらの文字列や連想リスト自体のいずれかを変異させることはエラーである。

```
(get-environment-variables)
          ⟹ (("USER" . "root") ("HOME" . "/"))
```

`(current-second)` time ライブラリ手続き

国際原子時 (TAI) スケールで，現在時刻を表す不正確数を返す。値 0.0 は TAI の 1970 年 1 月 1 日の真夜中を表し (世界時で真夜中の 10 秒前に相当)，値 1.0 は TAI の 1 秒後を表す。高い正確度も高い精度も必要でない; とりわけ，協定世界時に適切な定数を加えて返すのが実装ができる最善かもしれない。

`(current-jiffy)` time ライブラリ手続き

任意の，実装定義エポックから経過した正確整数として， jiffy 単位の数を返す。jiffy は秒の実装定義分数で，`jiffies-per-second` 手続きの戻り値によって定義されている。開始エポックはプログラムの実行中に一定であることが保証されるが，実行間で異なる場合がある。

根拠: jiffy 単位は `current-jiffy` が最小のオーバーヘッドで実行できるように，実装依存であることが許可されている。それは，コンパクトに表現された整数が戻り値として十分可能性が高くなければならない。任意の特定の jiffy サイズはいくつかの実装のために不適切になる。ほとんどの呼び出しに割り当てられなければならない整数を返すためにはるかに小さい単位が多くの実装を強制するとき，マイクロ秒は非常に高速なマシンには長すぎるので，`current-jiffy` レンダリングは正確な時間測定にはあまり有用でない。

`(jiffies-per-second)` time ライブラリ手続き

SI 秒あたり の jiffy の数を表す正確整数を返す。この値は実装指定の定数である。

```
(define (time-length)
  (let ((list (make-list 100000))
        (start (current-jiffy)))
    (length list)
    (/ (- (current-jiffy) start)
       (jiffies-per-second))))
```

`(features)` 手続き

`cond-expand` を真として扱う機能識別子のリストを返す。このリストを変更した場合エラーである。`features` が返してもよいものの例を以下に示す:

```
(features)                    ⟹
  (r7rs ratios exact-complex full-unicode
   gnu-linux little-endian
   fantastic-scheme
   fantastic-scheme-1.0
   space-ship-control-system)
```

7. 形式的な構文と意味論

本章は，本報告書のこれまでの章で非形式的に記述されてきたことの形式的な記述を与える。

7.1. 形式的構文

本節は拡張 BNF で書かれた Scheme の形式的構文を定める。

文法におけるすべてのスペースは読みやすさのためにある。英大文字と英小文字は，<英字>, <文字名> および <ニモニックエスケープ>の定義を除いて区別されない。たとえば， `#x1A` と `#X1a` は等価であるが，`foo` と `Foo` ならびに `#\space` と `#\Space` は独立している。<空> は空文字列を表す。

記述を簡潔にするため BNF への下記の拡張を使う: <なにか>* は <なにか> の 0 個以上の出現を意味し，<なにか>$^+$ は少なくとも 1 個の <なにか> を意味する。

7.1.1. 字句構造

本節は個々のトークン (識別子や数など) が文字の列からどのようにつくられるかを記述する。続く各節は，式とプログラムがトークンの列からどのようにつくられるかを記述する。

<トークン間スペース> は任意のトークンの左右どちらの側にも出現できるが，トークンの内側には出現できない。

縦線で始まらない識別子は，<区切り文字>によって，あるいは入力の終わりで終わる。それらはドット，数，文字，およびブーリアンである。縦線で始まる識別子は，もう一つの縦線で終わる。

次に挙げる ASCII レパートリーからの四つの文字は言語への将来の拡張のために予約されている: `[ ] { }`

下記で指定された ASCII レパートリーの識別子文字に加えて，Scheme 実装は，そのような各文字が Unicode 一般カテゴリ Lu, Ll, Lt, Lm, Lo, Mn, Mc, Me, Nd, Nl, No, Pd, Pc, Po, Sc, Sm, Sk, So, または Co をもつ，または U+200C または U+200D (それぞれ，幅がゼロの非結合子および結合子で，ペルシャ語，ヒンディー語，およびその他の言語で正しい綴りのために必要とされる) であるという条件で，識別子に採用される Unicode 文字の任意の追加レパートリーを許可してもよい。しかし，最初の文字が一般カテゴリ Nd, Mc, または Me をもつことはエラーである。非 Unicode 文字をシンボルまたは識別子に使用することもエラーである。

すべての Scheme 実装は，縦線で括られている Scheme 識別子で現れるためにエスケープシーケンス `\x<hexdigits>;` を許可しなければならない。もし与えられた Unicode スカラ値をもつ文字が実装でサポートされている場合は，そのような列を含んでいる識別子は対応する文字を含んでいる識別子と等価である。

```
<トークン> ⟶ <識別子> | <ブーリアン> | <数>
    | <文字> | <文字列>
    | ( | ) | #( | #u8( | ’ | ` | , | ,@ | .
<区切り文字> ⟶ <空白> | <縦線>
    | ( | ) | " | ;
<行内空白> ⟶ <スペースまたはタブ>
<空白> ⟶ <行内空白> | <行末>
<縦線> ⟶ |
<行末> ⟶ <改行> | <復帰> <改行>
    | <復帰>
<注釈> ⟶ ; ⟨行末までのすべての後続する文字⟩
              | <ネストしたコメント>
              | #; <トークン間スペース> <データ>
<ネストしたコメント> ⟶ #| <コメント文>
                              <継続コメント>* |#
<コメント文> ⟶ ⟨#| または |# を含まない文字の並び⟩
<継続コメント> ⟶ <ネストしたコメント> <コメント文>
<ディレクティブ> ⟶ #!fold-case | #!no-fold-case
```

<ディレクティブ> の後に <区切り文字> 以外の何か，または EOF が続くことは，非文法的であることに注意せよ。

```
<アトモスフィア> ⟶ <空白> | <注釈>
    | <ディレクティブ>
<トークン間スペース> ⟶ <アトモスフィア>*
```

以下の +i, -i および <無限大および無効値> は <独特な識別子> 規則の例外であることに注意せよ。それらは識別子としてではなく，数として解釈される。

```
<識別子> ⟶ <頭文字> <後続文字>*
    | <縦線> <シンボル要素>* <縦線>
    | <独特な識別子>
<頭文字> ⟶ <英字> | <特殊頭文字>
<英字> ⟶ a | b | c | ... | z
    | A | B | C | ... | Z
<特殊頭文字> ⟶ ! | $ | % | & | * | / | : | < | =
    | > | ? | ^ | _ | ~
<後続文字> ⟶ <頭文字> | <数字> | <特殊後続文字>
<数字> ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<16 進数字> ⟶ <数字> | a | b | c | d | e | f
<明示的符号> ⟶ + | -
<特殊後続文字> ⟶ <明示的符号> | . | @
<インライン 16 進エスケープ> ⟶ \x<16 進スカラ値>;
<16 進スカラ値> ⟶ <16 進数字>+
<ニモニックエスケープ> ⟶ \a | \b | \t | \n | \r
<独特な識別子> ⟶ <明示的符号>
    | <明示的符号> <符号後続文字> <後続文字>*
    | <明示的符号> . <ドット後続文字> <後続文字>*
    | . <ドット後続文字> <後続文字>*
<ドット後続文字> ⟶ <符号後続文字> | .
<符号後続文字> ⟶ <頭文字> | <明示的符号> | @
<記号要素> ⟶
    <any character other than <縦線> or \>
    | <インライン 16 進エスケープ>
    | <ニモニックエスケープ> | \|

<ブーリアン> ⟶ #t | #f | #true | #false
```

<文字> ⟶ #\ <任意の文字>
　　| #\ <文字名>
　　| #\x<16 進スカラ値>
<文字名> ⟶ alarm | backspace | delete | escape
　　| newline | null | return | space | tab

<文字列> ⟶ " <文字列要素>* "
<文字列要素> ⟶ <" と \を除く任意の文字>
　　| <ニモニックエスケープ> | \" | \\
　　| \<行内空白>*<行末>
　　<行内空白>*
　　| <インライン 16 進エスケープ>
<バイトベクタ> ⟶ #u8(<バイト>*)
<バイト> ⟶ <0 から 255 までの任意の整数>

<数> ⟶ <2 進数> | <8 進数>
　　| <10 進数> | <16 進数>

<R 進数>，<R 進複素数>，<R 進実数>，
<R 進符号なし実数>，<R 進符号なし整数>，お　よ　び
<R 進接頭辞>　に対する下記の規則は　$R = 2, 8, 10, 16$ に対して暗黙的に複製されるものとする。<2 進小数>，<8 進小数>，および <16 進小数> に対する規則はなく，これは小数点や指数部をもつ数は常に 10 進小数であることを意味する。以下には示されていないが，数の文法で使用されているすべてのアルファベット文字は，大文字または小文字のどちらでも書ける。

<R 進数> ⟶ <R 進接頭辞> <R 進複素数>
<R 進複素数> ⟶ <R 進実数>
　　| <R 進実数> @ <R 進実数>
　　| <R 進実数> + <R 進符号なし実数> i
　　| <R 進実数> - <R 進符号なし実数> i
　　| <R 進実数> + i | <R 進実数> - i
　　| <R 進実数> <無限大および無効値> i
　　| + <R 進符号なし実数> i | - <R 進符号なし実数> i
　　| <無限大および無効値> i | + i | - i
<R 進実数> ⟶ <符号> <R 進符号なし実数>
　　| <無限大および無効値>
<R 進符号なし実数> ⟶ <R 進符号なし整数>
　　| <R 進符号なし整数> / <R 進符号なし整数>
　　| <R 進小数>
<10 進小数> ⟶ <10 進符号なし整数> <接尾辞>
　　| . <10 進数字>$^+$ <接尾辞>
　　| <10 進数字>$^+$. <10 進数字>* <接尾辞>
<R 進符号なし整数> ⟶ <R 進数字>$^+$
<R 進接頭辞> ⟶ <R 進基数接頭辞> <正確性接頭辞>
　　| <正確性接頭辞> <R 進基数接頭辞>
<無限大および無効値> ⟶ +inf.0 | -inf.0 | +nan.0
　　| -nan.0

<接尾辞> ⟶ <空>
　　| <指数部マーカ> <符号> <10 進数字>$^+$
<指数部マーカ> ⟶ e
<符号> ⟶ <空> | + | -

<正確性接頭辞> ⟶ <空> | #i | #e
<2 進基数接頭辞> ⟶ #b
<8 進基数接頭辞> ⟶ #o
<10 進基数接頭辞> ⟶ <空> | #d
<16 進基数接頭辞> ⟶ #x
<2 進数字> ⟶ 0 | 1
<8 進数字> ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<10 進数字> ⟶ <数字>
<16 進数字> ⟶ <10 進数字> | a | b | c | d | e | f

7.1.2. 外部表現

<データ> とは read 手続き (6.13.2 節) がパースに成功するものである。<式> としてパースできる文字列はすべて，<データ> としてもパースできることになる点に注意せよ。

<データ> ⟶ <単純データ> | <複合データ>
　　| <ラベル> = <データ> | <ラベル> #
<単純データ> ⟶ <ブーリアン> | <数>
　　| <文字> | <文字列> | <シンボル> | <バイトベクタ>
<シンボル> ⟶ <識別子>
<複合データ> ⟶ <リスト> | <ベクタ> | <略記形>
<リスト> ⟶ (<データ>*) | (<データ>$^+$. <データ>)
<略記形> ⟶ <略記接頭辞> <データ>
<略記接頭辞> ⟶ ' | ` | , | ,@
<ベクタ> ⟶ #(<データ>*)
<ラベル> ⟶ # <10 進符号なし整数>

7.1.3. 式

本項および後続の項での定義は，本報告書内のすべての構文キーワードは，それらのライブラリから適切にインポートされ，いずれも再定義または隠蔽されていないものと仮定する。

<式> ⟶ <識別子>
　　| <リテラル式>
　　| <手続き呼出し>
　　| <ラムダ式>
　　| <条件式>
　　| <代入>
　　| <派生式>
　　| <マクロ使用>
　　| <マクロブロック>
　　| <インクルーダ>

<リテラル式> ⟶ <引用> | <自己評価的データ>
<自己評価的データ> ⟶ <ブーリアン> | <数>
　　| <ベクタ> | <文字> | <文字列> | <バイトベクタ>
<引用> ⟶ '<データ> | (quote <データ>)
<手続き呼出し> ⟶ (<演算子> <オペランド>*)
<演算子> ⟶ <式>
<オペランド> ⟶ <式>

<ラムダ式> ⟶ (lambda <仮引数部> <本体>)

<仮引数部> ⟶ (<識別子>*) | <識別子>
    | (<識別子>+ . <識別子>)
<本体> ⟶ <定義>* <列>
<列> ⟶ <コマンド>* <式>
<コマンド> ⟶ <式>

<条件式> ⟶ (if <テスト> <帰結部> <代替部>)
<テスト> ⟶ <式>
<帰結部> ⟶ <式>
<代替部> ⟶ <式> | <空>

<代入> ⟶ (set! <識別子> <式>)

<派生式> ⟶
      (cond <cond 節>+)
    | (cond <cond 節>* (else <列>))
    | (case <式>
        <case 節>+)
    | (case <式>
        <case 節>*
        (else <列>))
    | (case <式>
        <case 節>*
        (else => <レシピエント>))
    | (and <テスト>*)
    | (or <テスト>*)
    | (when <テスト> <列>)
    | (unless <テスト> <列>)
    | (let (<束縛仕様>*) <本体>)
    | (let <識別子> (<束縛仕様>*) <本体>)
    | (let* (<束縛仕様>*) <本体>)
    | (letrec (<束縛仕様>*) <本体>)
    | (letrec* (<束縛仕様>*) <本体>)
    | (let-values (<mv 束縛仕様>*) <本体>)
    | (let*-values (<mv 束縛仕様>*) <本体>)
    | (begin <列>)
    | (do (<繰返し仕様>*)
          (<テスト> <do 結果>)
        <コマンド>*)
    | (delay <式>)
    | (delay-force <式>)
    | (parameterize ((<式> <式>)*)
        <本体>)
    | (guard (<識別子> <cond 節>*) <本体>)
    | <準引用>
    | (case-lambda <case-lambda 節>*)

<cond 節> ⟶ (<テスト> <列>)
    | (<テスト>)
    | (<テスト> => <レシピエント>)
<レシピエント> ⟶ <式>
<case 節> ⟶ ((<データ>*) <列>)
    | ((<データ>*) => <レシピエント>)
<束縛仕様> ⟶ (<識別子> <式>)
<mv 束縛仕様> ⟶ (<仮引数部> <式>)

<繰返し仕様> ⟶ (<識別子> <初期値> <ステップ>)
    | (<識別子> <初期値>)
<case-lambda 節> ⟶ (<仮引数部> <本体>)
<初期値> ⟶ <式>
<ステップ> ⟶ <式>
<do 結果> ⟶ <列> | <空>

<マクロ使用> ⟶ (<キーワード> <データ>*)
<キーワード> ⟶ <識別子>

<マクロブロック> ⟶
    (let-syntax (<構文仕様>*) <本体>)
    | (letrec-syntax (<構文仕様>*) <本体>)
<構文仕様> ⟶ (<キーワード> <変換子仕様>)

<インクルーダ> ⟶
    | (include <文字列>+)
    | (include-ci <文字列>+)

7.1.4. 準引用

準引用式に対する下記の文法は文脈自由ではない。これは無限個の生成規則を生み出すためのレシピとして与えられている。$D = 1, 2, 3, \ldots$ に対する下記の規則のコピーを想像せよ。ここで D は入れ子の深さ (depth) である。

<準引用> ⟶ <準引用 1>
<qq テンプレート 0> ⟶ <式>
<準引用 D> ⟶ `<qq テンプレート D>
    | (quasiquote <qq テンプレート D>)
<qq テンプレート D> ⟶ <単純データ>
    | <リスト qq テンプレート D>
    | <ベクタ qq テンプレート D>
    | <脱引用 D>
<リスト qq テンプレート D> ⟶
      (<qq テンプレートまたは継ぎ合わせ D>*)
    | (<qq テンプレートまたは継ぎ合わせ D>+
       . <qq テンプレート D>)
    | '<qq テンプレート D>
    | <準引用 $D+1$>
<ベクタ qq テンプレート D> ⟶
      #(<qq テンプレートまたは継ぎ合わせ D>*)
<脱引用 D> ⟶ ,<qq テンプレート $D-1$>
    | (unquote <qq テンプレート $D-1$>)
<qq テンプレートまたは継ぎ合わせ D> ⟶
      <qq テンプレート D>
    | <継ぎ合わせ脱引用 D>
<継ぎ合わせ脱引用 D> ⟶ ,@<qq テンプレート $D-1$>
    | (unquote-splicing <qq テンプレート $D-1$>)

<準引用> において，<リスト qq テンプレート D> はときどき <脱引用 D> または <継ぎ合わせ脱引用 D> と紛らわしい。<脱引用 D> または <継ぎ合わせ脱引用 D> としての解釈が優先される。

7.1.5. 変換子

```
<変換子仕様> ⟶
    (syntax-rules (<識別子>*) <構文規則>*)
    | (syntax-rules <識別子> (<識別子>*)
        <構文規則>*)
<構文規則> ⟶ (<パターン> <テンプレート>)
<パターン> ⟶ <パターン識別子>
    | <アンダースコア>
    | (<パターン>*)
    | (<パターン>+ . <パターン>)
    | (<パターン>* <パターン> <省略符号> <パターン>*)
    | (<パターン>* <パターン> <省略符号> <パターン>*
        . <パターン>)
    | #(<パターン>*)
    | #(<パターン>* <パターン> <省略符号> <パターン>*)
    | <パターンデータ>
<パターンデータ> ⟶ <文字列>
    | <文字>
    | <ブーリアン>
    | <数>
<テンプレート> ⟶ <パターン識別子>
    | (<テンプレート要素>*)
    | (<テンプレート要素>+ . <テンプレート>)
    | #(<テンプレート要素>*)
    | <テンプレートデータ>
<テンプレート要素> ⟶ <テンプレート>
    | <テンプレート> <省略符号>
<テンプレートデータ> ⟶ <パターンデータ>
<パターン識別子> ⟶ <... を除く任意の識別子>
<省略符号> ⟶ <... にデフォルト設定された識別子>
<アンダースコア> ⟶ <識別子 _>
```

7.1.6. プログラムと定義

```
<プログラム> ⟶
    <インポート宣言>+
    <コマンドまたは定義>+
<コマンドまたは定義> ⟶ <コマンド>
    | <定義>
    | (begin <コマンドまたは定義>+)
<定義> ⟶ (define <識別子> <式>)
    | (define (<識別子> <def 仮引数部>) <本体>)
    | <構文定義>
    | (define-values <仮引数部> <本体>)
    | (define-record-type <識別子>
        <コンストラクタ> <識別子> <フィールド仕様>*)
    | (begin <定義>*)
<def 仮引数部> ⟶ <識別子>*
    | <識別子>* . <識別子>
<コンストラクタ> ⟶ (<識別子> <フィールド名>*)
<フィールド仕様> ⟶ (<フィールド名> <アクセサ>)
    | (<フィールド名> <アクセサ> <ミューテータ>)
<フィールド名> ⟶ <識別子>
<アクセサ> ⟶ <識別子>
<ミューテータ> ⟶ <識別子>
<構文定義> ⟶
    (define-syntax <キーワード> <変換子仕様>)
```

7.1.7. ライブラリ

```
<ライブラリ> ⟶
    (define-library <ライブラリ名>
      <ライブラリ定義>*)
<ライブラリ名> ⟶ (<ライブラリ名部>+)
<ライブラリ名部> ⟶ <識別子> | <10 進符号なし整数>
<ライブラリ宣言> ⟶ (export <エクスポート仕様>*)
    | <インポート宣言>
    | (begin <コマンドまたは定義>*)
    | <インクルーダ>
    | (include-library-declarations <文字列>+)
    | (cond-expand <cond-expand 節>+)
    | (cond-expand <cond-expand 節>+
                (else <ライブラリ宣言>*))
<import 宣言> ⟶ (import <インポート集合>+)
<エクスポート仕様> ⟶ <識別子>
    | (rename <識別子> <識別子>)
<インポート集合> ⟶ <ライブラリ名>
    | (only <インポート集合> <識別子>+)
    | (except <インポート集合> <識別子>+)
    | (prefix <インポート集合> <識別子>)
    | (rename <インポート集合> (<識別子> <識別子>)+)
<cond-expand 節> ⟶
    (<機能要件> <ライブラリ宣言>*)
<機能要件> ⟶ <識別子>
    | <ライブラリ名>
    | (and <機能要件>*)
    | (or <機能要件>*)
    | (not <機能要件>)
```

7.2. 形式的意味論

本節は，Scheme の原始式といくつか選んだ組込み手続きに対する形式的な表示的意味論を定める。ここで使う概念と表記は [36] に記述されている。dynamic-wind の定義は [39] から得た。表記を下記に要約する:

$\langle \ldots \rangle$	列の形成
$s \downarrow k$	列 s の第 k 要素 (1 から数えて)
$\#s$	列 s の長さ
$s \S t$	列 s と列 t の連結
$s \dagger k$	列 s の最初の k 個の要素を落とす
$t \to a, b$	McCarthy の条件式 “if t then a else b”
$\rho[x/i]$	置換 “i の代わりに x を持った ρ”
x in D	定義域 D の中への x の単射
$x \,\vert\, D$	定義域 D への x の射影

式の継続が，単一の値ではなく (複数個の) 値からなる列を受け取る理由は，手続き呼出しと多重個の戻り値の形式的な取扱いを単純化するためである。

ペア，ベクタ，および文字列に結合したブーリアンのフラグは，書換え可能オブジェクトならば真に，書換え不可能オブジェクトならば偽になる。

一つの呼出しにおける評価の順序は未規定である。我々はここで，呼出しにおける引数を評価する前と評価した後にそれらの引数に恣意的な順列並べ替え関数 *permute* とその逆関数 *unpermute* を適用することによってそれを模倣する。これは評価の順序が (任意の数の引数について) プログラム全体で一定であると間違って示唆しているのでまったく適切ではないが，それは左から右への評価よりも意図された意味論に近い近似である。

記憶領域を割り当てる関数 *new* は実装依存だが，次の公理に従わなければならない: もし $new\,\sigma \in \mathtt{L}$ ならば $\sigma(new\sigma|\mathtt{L}) \downarrow 2 = \mathit{false}$.

$\mathcal{K}$ の定義は省略する。なぜなら $\mathcal{K}$ の精密な定義は，あまりおもしろみなく意味論を複雑化するだろうからである。

もしも P がプログラムであって，そのすべての変数が参照または代入される前に定義されているならば，P の意味は

$$\mathcal{E}[\![\texttt{((lambda (I*) P')} \texttt{<未規定>} \ldots\texttt{)}]\!]$$

である。ここで I* は P で定義されている変数の列であり，P′ は P の中の定義をそれぞれ代入で置き換えることによって得られる式の列であり，<未規定> は *undefined* (未定義値) へと評価される式であり，そして $\mathcal{E}$ は意味を式に割り当てる意味関数 (semantic function) である。

7.2.1. 抽象構文

$\mathrm{K} \in \mathrm{Con}$	引用を含む定数
$\mathrm{I} \in \mathrm{Ide}$	識別子 (変数)
$\mathrm{E} \in \mathrm{Exp}$	式
$\Gamma \in \mathrm{Com} = \mathrm{Exp}$	コマンド

$$\begin{aligned}\mathrm{Exp} \longrightarrow\ & \mathrm{K} \mid \mathrm{I} \mid (\mathrm{E}_0\ \mathrm{E}^*)\\ \mid\ & \texttt{(lambda (I*) } \Gamma^*\ \mathrm{E}_0\texttt{)}\\ \mid\ & \texttt{(lambda (I* . I) } \Gamma^*\ \mathrm{E}_0\texttt{)}\\ \mid\ & \texttt{(lambda I } \Gamma^*\ \mathrm{E}_0\texttt{)}\\ \mid\ & \texttt{(if } \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)} \mid \texttt{(if } \mathrm{E}_0\ \mathrm{E}_1\texttt{)}\\ \mid\ & \texttt{(set! I E)}\end{aligned}$$

7.2.2. ドメイン等式

$\alpha \in \mathtt{L}$		場所 (locations)
$\nu \in \mathtt{N}$		自然数
$\mathtt{T}$	$= \{\mathit{false}, \mathit{true}\}$	ブーリアン
$\mathtt{Q}$		シンボル
$\mathtt{H}$		文字
$\mathtt{R}$		数
$\mathtt{E}_\mathrm{p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	ペア
$\mathtt{E}_\mathrm{v}$	$= \mathtt{L}^* \times \mathtt{T}$	ベクタ
$\mathtt{E}_\mathrm{s}$	$= \mathtt{L}^* \times \mathtt{T}$	文字列
$\mathtt{M}$	$=$ { 偽値, 真値, *null*, 未定義値, 未規定値 }	雑値 (miscellaneous)
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	手続き値
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E}_\mathrm{p} + \mathtt{E}_\mathrm{v} + \mathtt{E}_\mathrm{s} + \mathtt{M} + \mathtt{F}$	式の値
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	記憶装置 (stores)
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	環境
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	式の継続
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	式の継続
$\mathtt{A}$		答え (answers)
$\mathtt{X}$		エラー
$\omega \in \mathtt{P}$	$= (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{\mathit{root}\}$	動的ポイント

7.2.3. 意味関数

$$\begin{aligned}\mathcal{K} &: \mathrm{Con} \to \mathtt{E}\\ \mathcal{E} &: \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\ \mathcal{E}^* &: \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\ \mathcal{C} &: \mathrm{Com}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}\end{aligned}$$

ここで $\mathcal{K}$ の定義は故意に省略する。

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\omega\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{aligned}\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\omega\kappa\,.\,&\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\ &(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \to\\ &\qquad \mathit{wrong}\ \text{“未定義変数”},\\ &\quad \mathit{send}\,\epsilon\,\kappa))\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![(\mathrm{E}_0\ \mathrm{E}^*)]\!] =\\ &\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle \mathrm{E}_0\rangle \S \mathrm{E}^*))\\ &\qquad\qquad \rho\\ &\qquad\qquad \omega\\ &\qquad\qquad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^* \downarrow 1)\,(\epsilon^* \dagger 1)\,\omega\kappa)\\ &\qquad\qquad\qquad (\mathit{unpermute}\,\epsilon^*)))\end{aligned}$$

$$\begin{aligned}&\mathcal{E}[\![\texttt{(lambda (I*) } \Gamma^*\ \mathrm{E}_0\texttt{)}]\!] =\\ &\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,.\\ &\qquad \mathit{new}\,\sigma \in \mathtt{L} \to\\ &\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma\,|\,\mathtt{L},\\ &\qquad\qquad\quad \lambda\epsilon^*\omega'\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to\\ &\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\omega'\kappa'))\\ &\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\\ &\qquad\qquad\qquad\quad \epsilon^*,\\ &\qquad\qquad\qquad \mathit{wrong}\ \text{“引数の個数違い”}\rangle\\ &\qquad\qquad\quad \text{in } \mathtt{E})\\ &\qquad\qquad \kappa\\ &\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathtt{L})\,\mathit{unspecified}\,\sigma),\\ &\qquad\quad \mathit{wrong}\ \text{“メモリ不足”}\ \sigma\end{aligned}$$

$\mathcal{E}[\![\texttt{(lambda (I* . I) Γ* E}_0\texttt{)}]\!] =$
$\lambda\rho\omega\kappa\,.\,\lambda\sigma\,.$
$new\,\sigma \in \mathtt{L} \to$
$send\,(\langle new\,\sigma\,|\,\mathtt{L},$
$\lambda\epsilon^*\omega'\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to$
$tievalsrest$
$(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\omega'\kappa'))$
$(extends\,\rho\,(\mathrm{I}^* \S \langle \mathrm{I} \rangle)\,\alpha^*))$
ϵ^*
$(\#\mathrm{I}^*),$
$wrong$ “引数が少な過ぎる”$\rangle$ in E)
κ
$(update\,(new\,\sigma\,|\,\mathtt{L})\,unspecified\,\sigma),$
$wrong$ “メモリ不足” σ

$\mathcal{E}[\![\texttt{(lambda I Γ* E}_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (. I) Γ* E}_0\texttt{)}]\!]$

$\mathcal{E}[\![\texttt{(if E}_0\ \texttt{E}_1\ \texttt{E}_2\texttt{)}]\!] =$
$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,$
$\mathcal{E}[\![\mathrm{E}_2]\!]\rho\omega\kappa))$

$\mathcal{E}[\![\texttt{(if E}_0\ \texttt{E}_1\texttt{)}]\!] =$
$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,$
$send\,unspecified\,\kappa))$

ここでも他のところでも，*undefined* (未定義値) を除く任意の式の値を，*unspecified* (未規定値) のところに使ってよい。

$\mathcal{E}[\![\texttt{(set! I E)}]\!] =$
$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,\omega\,(single(\lambda\epsilon\,.\,assign\,(lookup\,\rho\,\mathrm{I})$
ϵ
$(send\,unspecified\,\kappa)))$

$\mathcal{E}^*[\![\ ]\!] = \lambda\rho\omega\kappa\,.\,\kappa\langle\,\rangle$

$\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] =$
$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\omega\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle \S \epsilon^*))))$

$\mathcal{C}[\![\ ]\!] = \lambda\rho\omega\theta\,.\,\theta$

$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\omega\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\omega\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\omega\theta)$

7.2.4. 補助関数

$lookup : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$
$lookup = \lambda\rho\mathrm{I}\,.\,\rho\mathrm{I}$

$extends : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}$
$extends =$
$\lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \to \rho,$
$extends\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)$

$wrong : \mathtt{X} \to \mathtt{C}$ [実装依存]

$send : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$
$send = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle$

$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$
$single =$
$\lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$
$wrong$ “戻り値の個数違い”

$new : \mathtt{S} \to (\mathtt{L} + \{error\})$ [実装依存]

$hold : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$
$hold = \lambda\alpha\kappa\sigma\,.\,send\,(\sigma\alpha \downarrow 1)\kappa\sigma$

$assign : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}$
$assign = \lambda\alpha\epsilon\theta\sigma\,.\,\theta(update\,\alpha\epsilon\sigma)$

$update : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}$
$update = \lambda\alpha\epsilon\sigma\,.\,\sigma[\langle\epsilon, true\rangle/\alpha]$

$tievals : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$
$tievals =$
$\lambda\psi\epsilon^*\sigma\,.\,\#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma,$
$new\,\sigma \in \mathtt{L} \to tievals\,(\lambda\alpha^*\,.\,\psi(\langle new\,\sigma\,|\,\mathtt{L}\rangle \S \alpha^*))$
$(\epsilon^* \dagger 1)$
$(update(new\,\sigma\,|\,\mathtt{L})(\epsilon^* \downarrow 1)\sigma),$
$wrong$ “メモリ不足”σ

$tievalsrest : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C}$
$tievalsrest =$
$\lambda\psi\epsilon^*\nu\,.\,list\,(dropfirst\,\epsilon^*\nu)$
$(single(\lambda\epsilon\,.\,tievals\,\psi\,((takefirst\,\epsilon^*\nu) \S \langle\epsilon\rangle)))$

$dropfirst = \lambda ln\,.\,n = 0 \to l, dropfirst\,(l \dagger 1)(n-1)$

$takefirst = \lambda ln\,.\,n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle \S (takefirst\,(l \dagger 1)(n-1))$

$truish : \mathtt{E} \to \mathtt{T}$
$truish = \lambda\epsilon\,.\,\epsilon = false \to false, true$

$permute : \mathrm{Exp}^* \to \mathrm{Exp}^*$ [実装依存]

$unpermute : \mathtt{E}^* \to \mathtt{E}^*$ [$permute$ の逆関数]

$applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$applicate =$
$\lambda\epsilon\epsilon^*\omega\kappa\,.\,\epsilon \in \mathtt{F} \to (\epsilon\,|\,\mathtt{F} \downarrow 2)\epsilon^*\omega\kappa, wrong$ “無効手続き”

$onearg : (\mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$
$onearg =$
$\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\omega\kappa,$
$wrong$ “引数の個数違い”

$twoarg : (\mathtt{E} \to \mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$
$twoarg =$
$\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\omega\kappa,$
$wrong$ “引数の個数違い”

$threearg : (\mathtt{E} \to \mathtt{E} \to \mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$
$threearg =$
$\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 3 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)(\epsilon^* \downarrow 3)\omega\kappa,$
$wrong$ “引数の個数違い”

$list : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$list =$
$\lambda\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 0 \to send\,null\,\kappa,$
$list\,(\epsilon^* \dagger 1)(single(\lambda\epsilon\,.\,cons\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))$

$cons : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$cons =$
$twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\omega\sigma\,.\,new\,\sigma \in \mathtt{L} \to$
$(\lambda\sigma'\,.\,new\,\sigma' \in \mathtt{L} \to$
$send\,(\langle new\,\sigma\,|\,\mathtt{L}, new\,\sigma'\,|\,\mathtt{L}, true\rangle$
in E)
κ
$(update(new\,\sigma'\,|\,\mathtt{L})\epsilon_2\sigma'),$
$wrong$ “メモリ不足”σ')
$(update(new\,\sigma\,|\,\mathtt{L})\epsilon_1\sigma),$
$wrong$ “メモリ不足”σ)

$$
\begin{aligned}
&less : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&less =\\
&\quad twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
&\qquad\qquad send\,(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to true, false)\kappa,\\
&\qquad\qquad wrong\ \text{“}\texttt{<}\ \text{に非数値引数”})
\end{aligned}
$$

$$
\begin{aligned}
&add : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&add =\\
&\quad twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
&\qquad\qquad send\,((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R})\ \text{in}\ \mathtt{E})\kappa,\\
&\qquad\qquad wrong\ \text{“}\texttt{<}\ \text{に非数値引数”})
\end{aligned}
$$

$$
\begin{aligned}
&car : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&car =\\
&\quad onearg\,(\lambda\epsilon\omega\kappa\,.\,\epsilon \in \mathtt{E_p} \to car\text{-}internal\,\epsilon\kappa,\\
&\qquad\qquad wrong\ \text{“}\texttt{car}\ \text{に非ペア引数”})
\end{aligned}
$$

$$
\begin{aligned}
&car\text{-}internal : \mathtt{E} \to \mathtt{K} \to \mathtt{C}\\
&car\text{-}internal = \quad \lambda\epsilon\omega\kappa\,.\,hold\,(\epsilon \mid \mathtt{E_p} \downarrow 1)\kappa
\end{aligned}
$$

$$
cdr : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \qquad [car\ \text{と同様}]
$$

$$
cdr\text{-}internal : \mathtt{E} \to \mathtt{K} \to \mathtt{C} \qquad [car\text{-}internal\ \text{と同様}]
$$

$$
\begin{aligned}
&setcar : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&setcar =\\
&\quad twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,\epsilon_1 \in \mathtt{E_p} \to\\
&\qquad\qquad (\epsilon_1 \mid \mathtt{E_p} \downarrow 3) \to assign\,(\epsilon_1 \mid \mathtt{E_p} \downarrow 1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \epsilon_2\\
&\qquad\qquad\qquad\qquad\qquad\qquad (send\ unspecified\,\kappa),\\
&\qquad\qquad wrong\ \text{“}\texttt{set-car!}\ \text{に書換え不可能引数”},\\
&\qquad\qquad wrong\ \text{“}\texttt{set-car!}\ \text{に非ペア引数”})
\end{aligned}
$$

$$
\begin{aligned}
&eqv : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&eqv =\\
&\quad twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,(\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to\\
&\qquad\qquad send\,(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to true, false)\kappa,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to\\
&\qquad\qquad\quad send\,(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to true, false)\kappa,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to\\
&\qquad\qquad\quad send\,(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to true, false)\kappa,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
&\qquad\qquad\quad send\,(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to true, false)\kappa,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{E_p} \wedge \epsilon_2 \in \mathtt{E_p}) \to\\
&\qquad\qquad\quad send\,((\lambda p_1 p_2\,.\,((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge\\
&\qquad\qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to true,\\
&\qquad\qquad\qquad\qquad\quad false)\\
&\qquad\qquad\qquad\quad (\epsilon_1 \mid \mathtt{E_p})\\
&\qquad\qquad\qquad\quad (\epsilon_2 \mid \mathtt{E_p}))\\
&\qquad\qquad\qquad\quad \kappa,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{E_v} \wedge \epsilon_2 \in \mathtt{E_v}) \to \ldots,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{E_s} \wedge \epsilon_2 \in \mathtt{E_s}) \to \ldots,\\
&\qquad\qquad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \to\\
&\qquad\qquad\quad send\,((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \to true, false)\\
&\qquad\qquad\qquad\quad \kappa,\\
&\qquad\qquad\quad send\ false\,\kappa)
\end{aligned}
$$

$$
\begin{aligned}
&apply : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&apply =\\
&\quad twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,\epsilon_1 \in \mathtt{F} \to valueslist\,\epsilon_2(\lambda\epsilon^*\,.\,applicate\,\epsilon_1\epsilon^*\omega\kappa),\\
&\qquad\qquad wrong\ \text{“}\texttt{apply}\ \text{に無効手続き引数”})
\end{aligned}
$$

$$
\begin{aligned}
&valueslist : \mathtt{E} \to \mathtt{K} \to \mathtt{C}\\
&valueslist =\\
&\quad \lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \to\\
&\qquad\quad cdr\text{-}internal\,\epsilon\\
&\qquad\qquad\qquad (\lambda\epsilon^*\,.\,valueslist\\
&\qquad\qquad\qquad\qquad \epsilon^*\\
&\qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,car\text{-}internal\\
&\qquad\qquad\qquad\qquad\qquad\qquad \epsilon\\
&\qquad\qquad\qquad\qquad\qquad\qquad (single(\lambda\epsilon\,.\,\kappa(\langle\epsilon\rangle \S \epsilon^*))))),\\
&\qquad \epsilon = null \to \kappa\langle\,\rangle,\\
&\qquad\quad wrong\ \text{“}\texttt{values-list}\ \text{に非リスト引数”}
\end{aligned}
$$

$$
\begin{aligned}
&cwcc\text{:}\ \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&\qquad [\texttt{call-with-current-continuation}]\\
&cwcc =\\
&\quad onearg\,(\lambda\epsilon\omega\kappa\,.\,\epsilon \in \mathtt{F} \to\\
&\qquad\qquad (\lambda\sigma\,.\,new\,\sigma \in \mathtt{L} \to\\
&\qquad\qquad\qquad applicate\,\epsilon\\
&\qquad\qquad\qquad\qquad \langle\langle new\,\sigma \mid \mathtt{L},\\
&\qquad\qquad\qquad\qquad\quad \lambda\epsilon^*\omega'\kappa'\,.\,travel\,\omega'\omega(\kappa\epsilon^*)\rangle\\
&\qquad\qquad\qquad\qquad\quad \text{in}\ \mathtt{E}\rangle\\
&\qquad\qquad\qquad\qquad \omega\\
&\qquad\qquad\qquad\qquad \kappa\\
&\qquad\qquad\qquad\qquad (update\,(new\,\sigma \mid \mathtt{L})\\
&\qquad\qquad\qquad\qquad\qquad\quad unspecified\\
&\qquad\qquad\qquad\qquad\qquad\quad \sigma),\\
&\qquad\qquad\qquad wrong\ \text{“メモリ不足”}\,\sigma),\\
&\qquad\qquad wrong\ \text{“無効手続き引数”})
\end{aligned}
$$

$$
\begin{aligned}
&travel : \mathtt{P} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}\\
&travel =\\
&\quad \lambda\omega_1\omega_2\,.\,travelpath\,((pathup\,\omega_1(commonancest\,\omega_1\omega_2)) \S\\
&\qquad\qquad\qquad (pathdown\,(commonancest\,\omega_1\omega_2)\omega_2))
\end{aligned}
$$

$$
\begin{aligned}
&pointdepth : \mathtt{P} \to \mathtt{N}\\
&pointdepth =\\
&\quad \lambda\omega\,.\,\omega = root \to 0, 1 + (pointdepth\,(\omega \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))
\end{aligned}
$$

$$
\begin{aligned}
&ancestors : \mathtt{P} \to \mathcal{P}\mathtt{P}\\
&ancestors =\\
&\quad \lambda\omega\,.\,\omega = root \to \{\omega\}, \{\omega\} \cup (ancestors\,(\omega \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))
\end{aligned}
$$

$$
\begin{aligned}
&commonancest : \mathtt{P} \to \mathtt{P} \to \mathtt{P}\\
&commonancest =\\
&\quad \lambda\omega_1\omega_2\,.\,\text{the only element of}\\
&\qquad\quad \{\omega' \mid \omega' \in (ancestors\,\omega_1) \cap (ancestors\,\omega_2),\\
&\qquad\qquad pointdepth\,\omega' \geq pointdepth\,\omega''\\
&\qquad\qquad \forall\omega'' \in (ancestors\,\omega_1) \cap (ancestors\,\omega_2)\}
\end{aligned}
$$

$$
\begin{aligned}
&pathup : \mathtt{P} \to \mathtt{P} \to (\mathtt{P} \times \mathtt{F})^*\\
&pathup =\\
&\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
&\qquad\quad \langle(\omega_1, \omega_1 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 2)\rangle \S\\
&\qquad\quad (pathup\,(\omega_1 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3)\omega_2)
\end{aligned}
$$

$$
\begin{aligned}
&pathdown : \mathtt{P} \to \mathtt{P} \to (\mathtt{P} \times \mathtt{F})^*\\
&pathdown =\\
&\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
&\qquad\quad (pathdown\,\omega_1(\omega_2 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3)) \S\\
&\qquad\quad \langle(\omega_2, \omega_2 \mid (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 1)\rangle
\end{aligned}
$$

$$
\begin{aligned}
&travelpath : (\mathtt{P} \times \mathtt{F})^* \to \mathtt{C} \to \mathtt{C}\\
&travelpath =
\end{aligned}
$$

$\lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta,$
$((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1)$
$(\lambda\epsilon^*\,.\,travelpath\,(\pi^* \dagger 1)\theta)$

$dynamicwind : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$dynamicwind =$
$threearg\,(\lambda\epsilon_1\epsilon_2\epsilon_3\omega\kappa\,.\,(\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F} \wedge \epsilon_3 \in \mathtt{F}) \to$
$applicate\,\epsilon_1\langle\rangle\omega(\lambda\zeta^*\,.$
$applicate\,\epsilon_2\langle\rangle((\epsilon_1 \mid \mathtt{F}, \epsilon_3 \mid \mathtt{F}, \omega)\text{ in }\mathtt{P})$
$(\lambda\epsilon^*\,.\,applicate\,\epsilon_3\langle\rangle\omega(\lambda\zeta^*\,.\,\kappa\epsilon^*))),$
$wrong$ “無効手続き引数”)

$values : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$values = \lambda\epsilon^*\omega\kappa\,.\,\kappa\epsilon^*$

$cwv : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$ [call-with-values]
$cwv =$
$twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,applicate\,\epsilon_1\langle\,\rangle\omega(\lambda\epsilon^*\,.\,applicate\,\epsilon_2\,\epsilon^*\omega))$

7.3. 派生式型

本節は quasiquote を除く原始式型 (リテラル，変数，呼出し，lambda，if，set!) による派生式型の構文定義を与える。

条件付き派生構文型:

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))


(define-syntax case
  (syntax-rules (else =>)
    ((case (key ...)
       clauses ...)
     (let ((atom-key (key ...)))
       (case atom-key clauses ...)))
    ((case key
       (else => result))
     (result key))
    ((case key
       (else result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
       ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((case key
       ((atoms ...) => result))
     (if (memv key '(atoms ...))
         (result key)))
    ((case key
       ((atoms ...) => result)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (result key)
         (case key clause clauses ...)))
    ((case key
       ((atoms ...) result1 result2 ...)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)
         (case key clause clauses ...)))))


(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))


(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))


(define-syntax when
  (syntax-rules ()
    ((when test result1 result2 ...)
     (if test
         (begin result1 result2 ...)))))


(define-syntax unless
  (syntax-rules ()
    ((unless test result1 result2 ...)
     (if (not test)
         (begin result1 result2 ...)))))
```

束縛コンストラクト:

```
(define-syntax let
  (syntax-rules ()
```

```
    ((let ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))
    ((let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (lambda (name ...)
                      body1 body2 ...)))
        tag)
      val ...))))


(define-syntax let*
  (syntax-rules ()
    ((let* () body1 body2 ...)
     (let () body1 body2 ...))
    ((let* ((name1 val1) (name2 val2) ...)
       body1 body2 ...)
     (let ((name1 val1))
       (let* ((name2 val2) ...)
         body1 body2 ...)))))
```

下記の letrec マクロは，なにか場所に格納され，その場所に格納された値を取得しようとするとエラーになってしまうものを返す式の代わりとして，シンボル <undefined> を使っている。(このような式は Scheme では定義されない。) 値が評価される順序を規定することを避けるために必要な一時変数を生成するため，あるトリックが使われている。これは補助的なマクロを使用することによっても達成できただろう。

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 init1) ...) body ...)
     (letrec "generate_temp_names"
       (var1 ...)
       ()
       ((var1 init1) ...)
       body ...))
    ((letrec "generate_temp_names"
       ()
       (temp1 ...)
       ((var1 init1) ...)
       body ...)
     (let ((var1 <undefined>) ...)
       (let ((temp1 init1) ...)
         (set! var1 temp1)
         ...
         body ...)))
    ((letrec "generate_temp_names"
       (x y ...)
       (temp ...)
       ((var1 init1) ...)
       body ...)
     (letrec "generate_temp_names"
       (y ...)
       (newtemp temp ...)
       ((var1 init1) ...)
       body ...))))


(define-syntax letrec*
  (syntax-rules ()
    ((letrec* ((var1 init1) ...) body1 body2 ...)
     (let ((var1 <undefined>) ...)
       (set! var1 init1)
       ...
       (let () body1 body2 ...)))))

(define-syntax let-values
  (syntax-rules ()
    ((let-values (binding ...) body0 body1 ...)
     (let-values "bind"
         (binding ...) () (begin body0 body1 ...)))

    ((let-values "bind" () tmps body)
     (let tmps body))

    ((let-values "bind" ((b0 e0)
         binding ...) tmps body)
     (let-values "mktmp" b0 e0 ()
         (binding ...) tmps body))

    ((let-values "mktmp" () e0 args
         bindings tmps body)
     (call-with-values
       (lambda () e0)
       (lambda args
         (let-values "bind"
             bindings tmps body))))

    ((let-values "mktmp" (a . b) e0 (arg ...)
         bindings (tmp ...) body)
     (let-values "mktmp" b e0 (arg ... x)
         bindings (tmp ... (a x)) body))

    ((let-values "mktmp" a e0 (arg ...)
        bindings (tmp ...) body)
     (call-with-values
       (lambda () e0)
       (lambda (arg ... . x)
         (let-values "bind"
             bindings (tmp ... (a x)) body))))))


(define-syntax let*-values
  (syntax-rules ()
    ((let*-values () body0 body1 ...)
     (let () body0 body1 ...))

    ((let*-values (binding0 binding1 ...)
         body0 body1 ...)
     (let-values (binding0)
       (let*-values (binding1 ...)
         body0 body1 ...)))))


(define-syntax define-values
  (syntax-rules ()
    ((define-values () expr)
     (define dummy
       (call-with-values (lambda () expr)
                         (lambda args #f))))
```

```
((define-values (var) expr)
 (define var expr))
((define-values (var0 var1 ... varn) expr)
 (begin
   (define var0
     (call-with-values (lambda () expr)
                       list))
   (define var1
     (let ((v (cadr var0)))
       (set-cdr! var0 (cddr var0))
       v)) ...
   (define varn
     (let ((v (cadr var0)))
       (set! var0 (car var0))
       v))))
((define-values (var0 var1 ... . varn) expr)
 (begin
   (define var0
     (call-with-values (lambda () expr)
                       list))
   (define var1
     (let ((v (cadr var0)))
       (set-cdr! var0 (cddr var0))
       v)) ...
   (define varn
     (let ((v (cdr var0)))
       (set! var0 (car var0))
       v))))
((define-values var expr)
 (define var
   (call-with-values (lambda () expr)
                     list)))))
```

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp ...)
     ((lambda () exp ...)))))
```

下記の代替的な begin の展開は，ラムダ式の本体に複数の式を書けるという能力を利用していない。いずれにせよ，これらの規則が適用できるのは， begin の本体に定義が全く含まれていないときだけであることに注意せよ。

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp)
     exp)
    ((begin exp1 exp2 ...)
     (call-with-values
         (lambda () exp1)
       (lambda args
         (begin exp2 ...))))))
```

下記の do の構文定義は変数節を展開するためにあるトリックを使っている。上記の letrec と同じく，補助マクロでもよかっただろう。式 (if #f #f) は未規定の値を得るために使われている。

```
(define-syntax do
  (syntax-rules ()
    ((do ((var init step ...) ...)
         (test expr ...)
         command ...)
     (letrec
       ((loop
         (lambda (var ...)
           (if test
               (begin
                 (if #f #f)
                 expr ...)
               (begin
                 command
                 ...
                 (loop (do "step" var step ...)
                       ...))))))
         (loop init ...)))
    ((do "step" x)
     x)
    ((do "step" x y)
     y)))
```

ここで delay, force ならびに delay-force の可能な実装がある。我々は式

```
(delay-force <式>)
```

を次の手続き呼出しと同じ意味をもつものとして定義する:

```
(make-promise #f (lambda () <式>))
```

すなわち，次のように定義する:

```
(define-syntax delay-force
  (syntax-rules ()
    ((delay-force expression)
     (make-promise #f (lambda () expression)))))
```

そして式

```
(delay <式>)
```

を次と同じ意味をもつものとして定義する:

```
(delay-force (make-promise #t <式>))
```

すなわち，次のように定義する:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (delay-force (make-promise #t expression)))))
```

ここで make-promise は次のように定義される:

```
(define make-promise
  (lambda (done? proc)
    (list (cons done? proc))))
```

最後に，非遅延結果 (すなわち， delay-force の代わりに delay によって生成された値) が返されるまで次のトランポリン技術 [38] を用いて約束の中の手続き式を繰り返し呼び出すために， force を次のように定義する:

```
(define (force promise)
  (if (promise-done? promise)
      (promise-value promise)
      (let ((promise* ((promise-value promise))))
        (unless (promise-done? promise)
          (promise-update! promise* promise))
        (force promise))))
```

次の約束アクセサによって:

```
(define promise-done?
  (lambda (x) (car (car x))))
(define promise-value
  (lambda (x) (cdr (car x))))
(define promise-update!
  (lambda (new old)
    (set-car! (car old) (promise-done? new))
    (set-cdr! (car old) (promise-value new))
    (set-car! new (car old))))
```

次の make-parameter および parameterize の実装はスレッドなしの実装にとって適切である。パラメタオブジェクトは，二つの任意の異なるオブジェクト<param-set!>と<param-convert>を用いてここでは手続きとして実装される:

```
(define (make-parameter init . o)
  (let* ((converter
          (if (pair? o) (car o) (lambda (x) x)))
         (value (converter init)))
    (lambda args
      (cond
       ((null? args)
        value)
       ((eq? (car args) <param-set!>)
        (set! value (cadr args)))
       ((eq? (car args) <param-convert>)
        converter)
       (else
        (error "bad parameter syntax"))))))
```

このとき parameterize は，関連付けられた値を動的に再束縛するために dynamic-wind を用いる:

```
(define-syntax parameterize
  (syntax-rules ()
    ((parameterize ("step")
                   ((param value p old new) ...)
                   ()
                   body)
     (let ((p param) ...)
       (let ((old (p)) ...
             (new ((p <param-convert>) value)) ...)
         (dynamic-wind
          (lambda () (p <param-set!> new) ...)
          (lambda () . body)
          (lambda () (p <param-set!> old) ...)))))
    ((parameterize ("step")
                   args
                   ((param value) . rest)
                   body)
     (parameterize ("step")
                   ((param value p old new) . args)
                   rest
                   body))
    ((parameterize ((param value) ...) . body)
     (parameterize ("step")
                   ()
                   ((param value) ...)
                   body))))
```

guard の次の実装は，ここで呼ばれる補助的なマクロ guard-aux に依存している。

```
(define-syntax guard
  (syntax-rules ()
    ((guard (var clause ...) e1 e2 ...)
     ((call/cc
       (lambda (guard-k)
         (with-exception-handler
          (lambda (condition)
            ((call/cc
              (lambda (handler-k)
                (guard-k
                 (lambda ()
                   (let ((var condition))
                     (guard-aux
                       (handler-k
                         (lambda ()
                           (raise-continuable condition)))
                       clause ...))))))))
          (lambda ()
            (call-with-values
             (lambda () e1 e2 ...)
             (lambda args
               (guard-k
                 (lambda ()
                   (apply values args)))))))))))))

(define-syntax guard-aux
  (syntax-rules (else =>)
    ((guard-aux reraise (else result1 result2 ...))
     (begin result1 result2 ...))
    ((guard-aux reraise (test => result))
     (let ((temp test))
       (if temp
           (result temp)
           reraise)))
    ((guard-aux reraise (test => result)
                clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (guard-aux reraise clause1 clause2 ...))))
    ((guard-aux reraise (test))
     (or test reraise))
    ((guard-aux reraise (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (guard-aux reraise clause1 clause2 ...))))
    ((guard-aux reraise (test result1 result2 ...))
```

```
     (if test
         (begin result1 result2 ...)
         reraise))
    ((guard-aux reraise
                (test result1 result2 ...)
                clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (guard-aux reraise clause1 clause2 ...)))))


(define-syntax case-lambda
  (syntax-rules ()
    ((case-lambda (params body0 ...) ...)
     (lambda args
       (let ((len (length args)))
         (let-syntax
             ((cl (syntax-rules ::: ()
                    ((cl)
                     (error "no matching clause"))
                    ((cl ((p :::) . body) . rest)
                     (if (= len (length '(p :::)))
                         (apply (lambda (p :::)
                                  . body)
                                args)
                         (cl . rest)))
                    ((cl ((p ::: . tail) . body)
                         . rest)
                     (if (>= len (length '(p :::)))
                         (apply
                          (lambda (p ::: . tail)
                            . body)
                          args)
                         (cl . rest))))))
               (cl (params body0 ...) ...)))))))
```

cond-expand の定義は， features 手続きと対話しない。それは実装で提供される各機能識別子が明示的に言及されることを要求する。

```
(define-syntax cond-expand
  ;; すべての機能 ID とライブラリを記載するため，これを拡張する
  (syntax-rules (and or not else r7rs library scheme base)
    ((cond-expand)
     (syntax-error "Unfulfilled cond-expand"))
    ((cond-expand (else body ...))
     (begin body ...))
    ((cond-expand ((and) body ...) more-clauses ...)
     (begin body ...))
    ((cond-expand ((and req1 req2 ...) body ...)
                  more-clauses ...)
     (cond-expand
       (req1
         (cond-expand
           ((and req2 ...) body ...)
           more-clauses ...))
       more-clauses ...))
    ((cond-expand ((or) body ...) more-clauses ...)
     (cond-expand more-clauses ...))
    ((cond-expand ((or req1 req2 ...) body ...)
                  more-clauses ...)
     (cond-expand
       (req1
        (begin body ...))
       (else
        (cond-expand
           ((or req2 ...) body ...)
           more-clauses ...))))
    ((cond-expand ((not req) body ...)
                  more-clauses ...)
     (cond-expand
       (req
         (cond-expand more-clauses ...))
       (else body ...)))
    ((cond-expand (r7rs body ...)
                  more-clauses ...)
       (begin body ...))
    ;; サポートされている各機能識別子のため，
    ;; ここで句を追加する。
    ;; 例:
    ;; ((cond-expand (exact-closed body ...)
    ;;               more-clauses ...)
    ;;   (begin body ...))
    ;; ((cond-expand (ieee-float body ...)
    ;;               more-clauses ...)
    ;;   (begin body ...))
    ((cond-expand ((library (scheme base))
                   body ...)
                  more-clauses ...)
      (begin body ...))
    ;; 各ライブラリのため，ここで句を追加する
    ((cond-expand (feature-id body ...)
                  more-clauses ...)
       (cond-expand more-clauses ...))
    ((cond-expand ((library (name ...))
                   body ...)
                  more-clauses ...)
       (cond-expand more-clauses ...))))
```

Appendix A. 標準ライブラリ

本節では，標準ライブラリで提供されるエクスポートを一覧表示する。ライブラリは，すべての実装でサポートされないかもしれない，あるいは負荷が高いかもしれない機能を分離するように考慮されている。

scheme ライブラリ接頭辞にはすべての標準ライブラリで使用され，将来の規格で使用するために予約されている。

base ライブラリ

(scheme base) ライブラリは，伝統的に Scheme と関連付けられた多くの手続きと構文束縛をエクスポートする。base ライブラリと他の標準ライブラリの境界は，構造上ではなく，使用上に基づく。特に，他の標準手続きまたは構文の点でコンパイラや実行システムによって通常プリミティブとして実装されている一部の手段は，base ライブラリの一部としてではなく，分離したライブラリとして定義されている。同様に， base ライブラリの一部のエクスポートは，他のエクスポートの点で実装可能である。それらは厳密な意味で冗長ではあるが，使い方の共通パターンを取り込んでいるので，便利な略語として提供されている。

```
*                               +
-                               ...
/                               <
<=                              =
=>                              >
>=                              _
abs                             and
append                          apply
assoc                           assq
assv                            begin
binary-port?                    boolean=?
boolean?                        bytevector
bytevector-append               bytevector-copy
bytevector-copy!                bytevector-length
bytevector-u8-ref               bytevector-u8-set!
bytevector?                     caar
cadr
call-with-current-continuation
call-with-port                  call-with-values
call/cc                         car
case                            cdar
cddr                            cdr
ceiling                         char->integer
char-ready?                     char<=?
char<?                          char=?
char>=?                         char>?
char?                           close-input-port
close-output-port               close-port
complex?                        cond
cond-expand                     cons
current-error-port              current-input-port
current-output-port             define
define-record-type              define-syntax
define-values                   denominator
do                              dynamic-wind
else                            eof-object
eof-object?                     eq?
equal?                          eqv?
error                           error-object-irritants
error-object-message            error-object?
even?                           exact
exact-integer-sqrt              exact-integer?
exact?                          expt
features                        file-error?
floor                           floor-quotient
floor-remainder                 floor/
flush-output-port               for-each
gcd                             get-output-bytevector
get-output-string               guard
if                              include
include-ci                      inexact
inexact?                        input-port-open?
input-port?                     integer->char
integer?                        lambda
lcm                             length
let                             let*
let*-values                     let-syntax
let-values                      letrec
letrec*                         letrec-syntax
list                            list->string
list->vector                    list-copy
list-ref                        list-set!
list-tail                       list?
make-bytevector                 make-list
make-parameter                  make-string
make-vector                     map
max                             member
memq                            memv
min                             modulo
negative?                       newline
not                             null?
number->string                  number?
numerator                       odd?
open-input-bytevector           open-input-string
open-output-bytevector          open-output-string
or                              output-port-open?
output-port?                    pair?
parameterize                    peek-char
peek-u8                         port?
positive?                       procedure?
quasiquote                      quote
quotient                        raise
raise-continuable               rational?
rationalize                     read-bytevector
read-bytevector!                read-char
read-error?                     read-line
read-string                     read-u8
real?                           remainder
reverse                         round
set!                            set-car!
set-cdr!                        square
string                          string->list
string->number                  string->symbol
string->utf8                    string->vector
string-append                   string-copy
string-copy!                    string-fill!
string-for-each                 string-length
```

```
string-map                string-ref
string-set!               string<=?
string<?                  string=?
string>=?                 string>?
string?                   substring
symbol->string            symbol=?
symbol?                   syntax-error
syntax-rules              textual-port?
truncate                  truncate-quotient
truncate-remainder        truncate/
u8-ready?                 unless
unquote                   unquote-splicing
utf8->string              values
vector                    vector->list
vector->string            vector-append
vector-copy               vector-copy!
vector-fill!              vector-for-each
vector-length             vector-map
vector-ref                vector-set!
vector?                   when
with-exception-handler    write-bytevector
write-char                write-string
write-u8                  zero?
```

case-lambda ライブラリ

(scheme case-lambda) ライブラリは，case-lambda 構文をエクスポートする。

```
case-lambda
```

char ライブラリ

(scheme char) ライブラリは，すべての Unicode 文字をサポートする場合，潜在的に大きなテーブルを伴う文字を扱う手続きを提供する。

```
char-alphabetic?          char-ci<=?
char-ci<?                 char-ci=?
char-ci>=?                char-ci>?
char-downcase             char-foldcase
char-lower-case?          char-numeric?
char-upcase               char-upper-case?
char-whitespace?          digit-value
string-ci<=?              string-ci<?
string-ci=?               string-ci>=?
string-ci>?               string-downcase
string-foldcase           string-upcase
```

complex ライブラリ

(scheme complex) ライブラリは，通常非実数のみ有用な手続きをエクスポートする。

```
angle                     imag-part
magnitude                 make-polar
make-rectangular          real-part
```

CxR ライブラリ

(scheme cxr) ライブラリは，3 から 4 つの car と cdr 操作の合成からなる 24 個の手続きをエクスポートする。

```
(define caddar
  (lambda (x) (car (cdr (cdr (car x))))))
```

car と cdr 自身，およびこれら 2 つの合成からなる 4 個の手続きは，base ライブラリに含まれる。6.4 節参照。

```
caaaar                    caaadr
caaar                     caadar
caaddr                    caadr
cadaar                    cadadr
cadar                     caddar
cadddr                    caddr
cdaaar                    cdaadr
cdaar                     cdadar
cdaddr                    cdadr
cddaar                    cddadr
cddar                     cdddar
cddddr                    cdddr
```

eval ライブラリ

(scheme eval) ライブラリは，プログラムとして Scheme データを評価する手続きをエクスポートする。

```
environment               eval
```

file ライブラリ

(scheme file) ライブラリは，ファイルにアクセスするための手続きを提供する。

```
call-with-input-file      call-with-output-file
delete-file               file-exists?
open-binary-input-file    open-binary-output-file
open-input-file           open-output-file
with-input-from-file      with-output-to-file
```

inexact ライブラリ

(scheme inexact) ライブラリは，通常不正確値のみ有用な手続きをエクスポートする。

```
acos                      asin
atan                      cos
exp                       finite?
infinite?                 log
nan?                      sin
sqrt                      tan
```

lazy ライブラリ

(scheme lazy) ライブラリは，遅延評価のための手続きと構文キーワードをエクスポートする。

```
delay                     delay-force
force                     make-promise
promise?
```

load ライブラリ

(scheme load) ライブラリは，Scheme 式をファイルからロードする手続きをエクスポートする。

```
load
```

process-context ライブラリ

(scheme process-context) ライブラリは，プログラムの呼び出しコンテキストにアクセスする手続きをエクスポートする。

```
command-line                emergency-exit
exit
get-environment-variable
get-environment-variables
```

read ライブラリ

(scheme read) ライブラリは，Scheme オブジェクトを読み込むための手続きを提供する。

```
read
```

REPL ライブラリ

(scheme repl) ライブラリは，interaction-environment 手続きをエクスポートする。

```
interaction-environment
```

time ライブラリ

(scheme time) ライブラリは，時間に関連した値へのアクセスを提供する。

```
current-jiffy               current-second
jiffies-per-second
```

write ライブラリ

(scheme write) ライブラリは，Scheme オブジェクトを書き込むための手続きを提供する。

```
display                     write
write-shared                write-simple
```

R5RS ライブラリ

(scheme r5rs) ライブラリは，transcript-on と transcript-off が存在しないことを除いて，R^{5}RS で定義された識別子を提供する。exact および inexact 手続きは，それらは R^{5}RS のもとでは，それぞれ名前 inexact->exact および exact->inexact で現れることに注意せよ。しかし，もし実装が複素数ライブラリのような特定のライブラリを提供しなければ，対応する識別子はこのライブラリに現れなくなる。

```
*                           +
-                           /
<                           <=
=                           >
>=                          abs
acos                        and
angle                       append
apply                       asin
assoc                       assq
assv                        atan
begin                       boolean?
caaaar                      caaadr
caaar                       caadar
caaddr                      caadr
caar                        cadaar
cadadr                      cadar
caddar                      cadddr
caddr                       cadr
call-with-current-continuation
call-with-input-file        call-with-output-file
call-with-values            car
case                        cdaaar
cdaadr                      cdaar
cdadar                      cdaddr
cdadr                       cdar
cddaar                      cddadr
cddar                       cdddar
cddddr                      cdddr
cddr                        cdr
ceiling                     char->integer
char-alphabetic?            char-ci<=?
char-ci<?                   char-ci=?
char-ci>=?                  char-ci>?
char-downcase               char-lower-case?
char-numeric?               char-ready?
char-upcase                 char-upper-case?
char-whitespace?            char<=?
char<?                      char=?
char>=?                     char>?
char?                       close-input-port
close-output-port           complex?
cond                        cons
cos                         current-input-port
current-output-port         define
define-syntax               delay
denominator                 display
do                          dynamic-wind
eof-object?                 eq?
equal?                      eqv?
eval                        even?
exact->inexact              exact?
exp                         expt
floor                       for-each
force                       gcd
if                          imag-part
inexact->exact              inexact?
input-port?                 integer->char
integer?                    interaction-environment
lambda                      lcm
length                      let
```

```
let*                        let-syntax
letrec                      letrec-syntax
list                        list->string
list->vector                list-ref
list-tail                   list?
load                        log
magnitude                   make-polar
make-rectangular            make-string
make-vector                 map
max                         member
memq                        memv
min                         modulo
negative?                   newline
not                         null-environment
null?                       number->string
number?                     numerator
odd?                        open-input-file
open-output-file            or
output-port?                pair?
peek-char                   positive?
procedure?                  quasiquote
quote                       quotient
rational?                   rationalize
read                        read-char
real-part                   real?
remainder                   reverse
round
scheme-report-environment
set!                        set-car!
set-cdr!                    sin
sqrt                        string
string->list                string->number
string->symbol              string-append
string-ci<=?                string-ci<?
string-ci=?                 string-ci>=?
string-ci>?                 string-copy
string-fill!                string-length
string-ref                  string-set!
string<=?                   string<?
string=?                    string>=?
string>?                    string?
substring                   symbol->string
symbol?                     tan
truncate                    values
vector                      vector->list
vector-fill!                vector-length
vector-ref                  vector-set!
vector?                     with-input-from-file
with-output-to-file         write
write-char                  zero?
```

Appendix B. 標準機能識別子

実装は `cond-expand` および `features` による使用のために，下に列挙されている任意のあるいはすべての機能識別子を提供してもよいが，もし対応する機能を提供していなければ，その機能識別子は提供してはならない。

`r7rs`

すべての R[7]RS Scheme 実装はこの機能をもつ。

`exact-closed`

`/` を除くすべての代数演算は，与えられた正確入力に対して正確値をもたらす。

`exact-complex`

正確複素数が提供されている。

`ieee-float`

不正確数が IEEE 754 バイナリ浮動小数点数である。

`full-unicode`

Unicode version 6.0 におけるすべての Unicode 文字表現が， Scheme 文字としてサポートされている。

`ratios`

除数が非ゼロのとき，`/` に正確な引数を指定して，正確な結果をもたらす。

`posix`

この実装が，POSIX システム上で実行されている。

`windows`

この実装が，Windows 上で実行されている。

`unix, darwin, gnu-linux, bsd, freebsd, solaris, ...`

オペレーティングシステムフラグ (おそらく一つ以上)。

`i386, x86-64, ppc, sparc, jvm, clr, llvm, ...`

CPU アーキテクチャフラグ。

`ilp32, lp64, ilp64, ...`

C メモリモデルフラグ。

`big-endian, little-endian`

バイトオーダフラグ。

<名前>

この実装の名前。

<名前-バージョン>

この実装の名前およびバージョン。

言語変化

R^{5}RS との非互換性

本節は，本報告書と “Revised^5 report” [20] の間の非互換性を列挙する。

本リストは正式ではないが，正しく完全であると考えられる。

- 大文字と小文字の区別は現在，シンボルと文字の名前でデフォルトである。これは，次の仮定の下で書かれたコードを意味する。ある文脈で書かれた `FOO` または `Foo` というシンボルと，別の分脈で書かれた `foo` は，新たな `#!fold-case` ディレクティブでマークされる，または `include-ci` ライブラリ宣言を使ってライブラリに含められるのどちらかに変更することができる。すべての標準的な識別子は全体が小文字である。
- `syntax-rules` コンストラクトは現在，`_` (アンダースコア) をワイルドカードとして認識する。これは，構文変数として使用することができないことを意味する。それはまだ，リテラルとして使用することができる。
- R^{5}RS 手続き `exact->inexact` と `inexact->exact` は，これらの名前がより短く，より正確であるように，R^{6}RS の名前 `inexact` と `exact` にそれぞれ名称変更された。以前の名前は R^{5}RS ライブラリで引き続き利用可能である。
- (`string<?` および関連する述語による) 文字列比較は，(`char<?` および関連する述語による) 文字比較の辞書編集の拡張であることの保証は削除された。
- 数値リテラル内の # 文字のサポートは不要になった。
- 指数マーカーとしての `s`, `f`, `d`, および `l` 文字のサポートは不要になった。
- `string->number` の実装は，引数が明示的な基数接頭辞を含んでいる場合 `#f` を返すことはもはや許可されず，`read` およびプログラムの数値の構文との互換性がなくてはならない。
- 手続き `transcript-on` と `transcript-off` は削除された。

R^{5}RS 以降の他の言語変化

本節は，本報告書と “Revised^5 report” [20] の間の追加の相違点を列挙する。

本リストは正式ではないが，正しく完全であると考えられる。

- R^{5}RS における，様々なマイナーな曖昧さや不明瞭さは一掃された。
- ライブラリは，カプセル化とコードの共有を改善するための新たなプログラム構造として追加された。いくつかの既存および新規の識別子が，分離したライブラリに織り込まれている。ライブラリは，制御された表出および識別子の名前変更とともに，他のライブラリやメインプログラムにインポートすることができる。ライブラリの内容は，それが使用されるべき実装の機能を条件とすることができる。R^{5}RS 互換ライブラリが存在する。
- 式型 `include`, `include-ci`, および `cond-expand` は，base ライブラリに追加された; それらは対応するライブラリ宣言と同じ意味を持つ。
- 例外は現在，`raise`, `raise-continuable` または `error` で明示的に通知することができ，`with-exception-handler` と `guard` 構文で処理することができる。任意のオブジェクトは，エラー状態を指定することができる; `error` によって通知された実装定義の条件は，それらを検出するための述語と，`error` に渡される引数を取得するためのアクセス関数を持っている。`read` によって，およびファイルに関連した手続きによって通知される条件もまたそれらを検出するための述語を持っている。
- 複数のフィールドへのアクセスをサポートする新しい分離的な型は， SRFI 9 [19] の `define-record-type` を使用して生成することができる。
- パラメータオブジェクトは， `make-parameter` を使用して作成することができ， `parameterize` を使用して動的に再束縛される。手続き `current-input-port` と `current-output-port` は現在，新たに導入された `current-error-port` がそうであるように，パラメータオブジェクトである。
- 約束のサポートは， SRFI 45 [38] に基づいて拡張されている。
- 0 から 255 の範囲の正確整数のベクタであるバイトベクタは，新たな分離的な型として追加された。ベクタ手続きのサブセットが提供される。バイトベクタは，UTF-8 文字エンコーディングに基づいて文字列との相互変換ができる。バイトベクタはデータ表現を持っており，それ自身に評価される。
- ベクタ定数はそれ自身に評価される。
- 手続き `read-line` は，行指向のテキスト入力を簡単にするために提供された。
- 手続き `flush-output-port` は，出力ポートバッファリングの最小の制御を可能にするために提供された。
- ポートは現在，バイナリデータを読み書きするための新しい手続きとともに，**テキスト** または **バイナリ**ポートとして指定することができる。新しい述語 `input-port-open?` および `output-port-open?` は，ポートが開いているか閉じているかを返す。新しい手

続き close-port は現在のポートを閉じる; ポートが入力側と出力側の両方を有する場合，両方が閉じられる。

- **文字列ポート** は文字列の文字を読み書きする方法として，**バイトベクタポート** はバイトベクタのバイトを読み書きするために追加された。
- 現在文字列とバイトベクタに固有の I/O 手続きが存在する。
- write 手続きは現在，環状のオブジェクトに適用されたとき，データラベルを生成する。新しい手続き write-simple はラベルを生成することはない。write-shared は，すべての共有構造および環状構造にラベルを生成する。display 手続きは，環状オブジェクトでループしてはならない。
- R[6]RS の手続き eof-object が追加された。EOF オブジェクトは現在，分離的な型であることが必須である。
- 構文定義は現在，変数定義がどこにあっても許可されている。
- syntax-rules コンストラクトは現在，省略符号をデフォルト ... の代わりに明示的に指定することが許可され，テンプレートが省略符号が接頭辞についたリスト用いてエスケープすることが許可され，末尾パターンが省略符号パターンに続くことが許可されている。
- syntax-error 構文は，マクロが展開された時点で，すぐにより有益なエラーを通知するための手段として追加された。
- letrec* 束縛コンストラクトが追加され，その点で内部 define が指定された。
- 複数の値を取得するためのサポートが define-values, let-values, および let*-values で強化された。式の列を含んでいる標準的な式型は，現在，列のすべての非終端式の継続にゼロまたは 1 個以上の値を渡すことを許可する。
- case 条件は現在， cond に類似する => 構文を，正規句の中だけでなく， else 句の中も同様にサポートしている。
- 手続きに渡される引数の数での配布をサポートするために， case-lambda が自身のライブラリに追加された。
- 便利な条件 when と unless が追加された。
- eqv? の不正確数での振舞いは現在， R[6]RS の定義に準拠している。
- 手続きに適用された場合， eq? および eqv? は異なる答えを返すことが許可されている。
- R[6]RS の手続き boolean=? と symbol=? が追加された。
- 正の無限大，負の無限大，NaN，負の不正確ゼロが，不正確値としてそれぞれ表記表現 +inf.0, -inf.0, +nan.0, and -0.0 で数値塔に追加された。それらのサポートは必須ではない。表現 -nan.0 は +nan.0 と同義である。
- log 手続きは現在，対数の底を指定する第 2 引数を受け取る。
- 手続き map と for-each は現在，最も短い引数リストで終了する必要がある。
- 手続き member と assoc は現在，使用する等号述語を指定するオプションの第 3 引数をとる。
- 数値手続き finite?, infinite?, nan?, exact-integer?, square, and exact-integer-sqrt が追加された。
- - と / 手続き，文字および文字列比較述語は現在，二つ以上の引数をサポートする必要がある。
- 形式 #true と #false は現在， #t と #f 同様にサポートされている。
- 手続き make-list, list-copy, list-set!, string-map, string-for-each, string->vector, vector-append, vector-copy, vector-map, vector-for-each, vector->string, vector-copy!, and string-copy! は，列の操作を完成するために追加された。
- 一部の文字列およびベクタの手続きは，オプションの *start* と *end* 引数を使用して，文字列またはベクタの部分処理をサポートする。
- 一部のリスト手続きは現在，環状リストに定義されている。
- 実装は， ASCII を含む完全な Unicode レパートリーの任意のサブセットを提供してもよいが，実装は Unicode と一致する方法で，そのようなサブセットをサポートしなければならない。これに応じて様々な文字と文字列の手続きが拡張されており，文字列に対して，大文字と小文字の変換手続きが追加された。文字列の比較はもはや，単に Unicode のスカラ値に基づく文字比較と一致する必要はない。新しい digit-value 手続きが，数字文字の数値を取得するために追加された。
- 2 つの追加のコメント構文が存在する: 次のデータをスキップするための #; ，および入れ子が可能なブロックコメントのための #| ... |# 。
- データラベルが接頭辞に付いたデータ #<n>= は，共有構造を持つデータの読み書きを可能にし， #<n># で参照することができる。
- 文字列および記号は現在，ニモニックと数値のエスケープシーケンスを可能にし，名前付き文字のリストが拡張された。

- 手続き `file-exists?` と `delete-file` は，（`scheme file`）ライブラリで利用可能である。
- システム環境，コマンドライン，およびプロセス終了ステータスへのインタフェースは，（`scheme process-context`）ライブラリで利用可能である。
- 時間関連の値にアクセスするための手続きは，（`scheme time`）ライブラリで利用可能である。
- 異常が少ない整数除算演算子のセットは，新しくより明確な名前で提供されている。
- `load` 手続きは現在，ロード先の環境を指定する第 2 引数を受け取る。
- `call-with-current-continuation` 手続き現在，同義語 `call/cc` を持つ。
- read-eval-print ループの意味は現在，構文キーワードはしないが手続きの再定義を必要とする，遡及効果を持つことが部分的に規定されている。
- 形式的意味論は現在，`dynamic-wind` を操作する。

R^{6}RS との非互換性

本節では，R^{7}RS と “改訂 ^{6} レポート” [33] およびそれに付随する標準ライブラリのドキュメントとの非互換性を列挙する。

本リストは正式ではなく，おそらく不完全である。

- R^{7}RS ライブラリは，それらが構文的に R^{6}RS ライブラリから区別できるようにするために，キーワード `library` ではなく `define-library` で始まる。R^{7}RS の用語では， R^{6}RS ライブラリの本体は，単一のエクスポート宣言に続いて，単一のインポート宣言，それに続くコマンドおよび定義で構成されている。R^{7}RS では，コマンドおよび定義は，本体内で直接許可されていない: それらは `begin` ライブラリ宣言に覆われなければならない。
- `include`, `include-ci`, `include-library-declarations`, または `cond-expand` ライブラリ宣言に直接相当するものは， R^{6}RS にはない。一方， R^{7}RS ライブラリ構文は，位相またはバージョン仕様をサポートしていない。
- ライブラリに標準化された識別子のグループ分けは，R^{6}RS のアプローチとは異なる。特に， R^{5}RS ではオプションの手続き，いずれか明示または暗示されることにより， base ライブラリから削除された。base ライブラリ自体のみが絶対要件でである。
- 識別子構文の形式は提供されない。
- 内部構文定義は許可されているが，構文形式の使用は，その定義の前に現れることはできない。R^{6}RS で与えられた `even`/`odd` の例は許可されていない。
- R^{6}RS 例外システムはそのまま組み込まれたが，条件型が未規定のままになっている。特に， R^{6}RS が通知されるべき指定された型の条件を必要とする場合，通知の未解決問題を残して， R^{7}RS は “it is an error” とだけ示す。
- Unicode のフルサポートは必須ではない。正規化は提供されていない。文字の比較は， Unicode で定義されるが，文字列比較は実装依存である。非 Unicode 文字は許可されている。
- すべての数値塔は R^{5}RS では任意であるが， IEEE の無限大， NaN ，および -0.0 のオプションサポートは R^{6}RS から採用された。数値結果のほとんどの明確化も採用されたが， R^{6}RS の手続き `real-valued?`, `rational-valued?`, and `integer-valued?` はそうではない。R^{6}RS 除算演算子 `div`, `mod`, `div-and-mod`, `div0`, `mod0` および `div0-and-mod0` は提供されていない。
- 結果が未規定である場合，それはまだ単一の値であることが必須である。しかし，本体内の非終端式は，任意の数の値を返すことができる。
- `map` と `for-each` の意味は，SRFI 1 [30] 早期終了の振舞いを使用するように変更された。同様に，`assoc` と `member` は R^{6}RS の `assp` と `memp` 手続きを分離する代わりに，SRFI 1 のようにオプションの `equal?` 引数をとる。
- R^{6}RS`quasiquote` の明確化は，複数の引数 `unquote` と `unquote-splicing` の例外とともに採用された。
- 仮数部の幅を指定する R^{6}RS の方法は採用されなかった。
- 文字列ポートは， R^{6}RS ではなく SRFI 6 [7] と互換性がある。
- R^{6}RS 形式のバイトベクタが含まれているが，符号なしバイト (`u8`) 手続きだけが提供された。字句構文は，R^{6}RS `#vu8` 形式ではなく， SRFI 4 [13] との互換性のために `#u8` を使用する。
- ユーティリティマクロ `when` と `unless` が提供されているが，その結果は未規定のままである。
- 標準ライブラリドキュメントの残りの機能は，将来の標準化の努力に委ねられた。

追加資料

TScheme コミュニティウェブサイト `http://schemers.org` は，学習やプログラミング，仕事およびイベントのポスト，また Scheme ユーザグループ情報の追加資料を含んでいる。

Scheme 関連研究の参考文献 `http://library.readscheme.org` には，古典的論文

や最近の研究を含む， Scheme 言語に関する技術文書と論文へリンクしている。

オンライン上の Scheme の議論は，IRC を使用した irc.freenode.net の #scheme チャンネルおよび Usenet ディスカッショングループ comp.lang.scheme にて開催されている。

例

手続き integrate-system は微分方程式の系

$$y'_k = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

を Runge-Kutta 法で積分する。

パラメタ system-derivative は系の状態 (状態変数 $y_1, \ldots, y_n$ に対する値からなるベクタ) を受け取って系の微分係数 (値 $y'_1, \ldots, y'_n$) を算出する関数である。パラメタ initial-state は系の初期状態を定める。そして h は積分の刻み幅の初期推定値である。

integrate-system が返す値は，系の状態からなる無限ストリームである。

```
(define (integrate-system system-derivative
                          initial-state
                          h)
  (let ((next (runge-kutta-4 system-derivative h)))
    (letrec ((states
              (cons initial-state
                    (delay (map-streams next
                                        states)))))
      states)))
```

手続き runge-kutta-4 は，系の状態から系の微分係数を算出する関数 f を受け取る。これは，系の状態を受け取って新しい系の状態を算出する関数を算出する。

```
(define (runge-kutta-4 f h)
  (let ((*h (scale-vector h))
        (*2 (scale-vector 2))
        (*1/2 (scale-vector (/ 1 2)))
        (*1/6 (scale-vector (/ 1 6))))
    (lambda (y)
      ;; y is a system state
      (let* ((k0 (*h (f y)))
             (k1 (*h (f (add-vectors y (*1/2 k0)))))
             (k2 (*h (f (add-vectors y (*1/2 k1)))))
             (k3 (*h (f (add-vectors y k2)))))
        (add-vectors y
          (*1/6 (add-vectors k0
                             (*2 k1)
                             (*2 k2)
                             k3)))))))

(define (elementwise f)
  (lambda vectors
    (generate-vector
     (vector-length (car vectors))
     (lambda (i)
       (apply f
              (map (lambda (v) (vector-ref  v i))
                   vectors))))))

(define (generate-vector size proc)
  (let ((ans (make-vector size)))
    (letrec ((loop
              (lambda (i)
                (cond ((= i size) ans)
```

```
                    (else
                     (vector-set! ans i (proc i))
                     (loop (+ i 1)))))))
      (loop 0))))

(define add-vectors (elementwise +))

(define (scale-vector s)
  (elementwise (lambda (x) (* x s))))
```

map-streams 手続きは map に相当する: これはその第一引数 (手続き) を第二引数 (ストリーム) のすべての要素に適用する。

```
(define (map-streams f s)
  (cons (f (head s))
        (delay (map-streams f (tail s)))))
```

無限ストリームは，そのストリームの最初の要素を car が保持し，そしてそのストリームの残りをかなえる約束を cdr が保持するペアとして実装される。

```
(define head car)
(define (tail stream)
  (force (cdr stream)))
```

下記に示すのは減衰振動をモデル化した系

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

の積分に integrate-system を応用した例である。

```
(define (damped-oscillator R L C)
  (lambda (state)
    (let ((Vc (vector-ref state 0))
          (Il (vector-ref state 1)))
      (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
              (/ Vc L)))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

参考文献

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[3] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. http://www.ietf.org/rfc/rfc2119.txt, 1997. (日本語訳 translated by Mendoxi. http://www.cam.hi-ho.ne.jp/mendoxi/rfc/rfc2119j.html)

[4] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116.

[5] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

[6] William Clinger. Proper Tail Recursion and Space Efficiency. In *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, June 1998.

[7] William Clinger. SRFI 6: Basic String Ports. http://srfi.schemers.org/srfi-6/, 1999.

[8] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[9] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[10] William Clinger and Jonathan Rees, editors. The revised^4 report on the algorithmic language Scheme. In *ACM Lisp Pointers* 4(3), pages 1–55, 1991.

[11] Mark Davis. Unicode Standard Annex #29, Unicode Text Segmentation. http://unicode.org/reports/tr29/, 2010.

[12] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4):295–326, 1993.

[13] Marc Feeley. SRFI 4: Homogeneous Numeric Vector Datatypes. http://srfi.schemers.org/srfi-45/, 1999.

[14] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [15].

[15] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[16] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life." In *Scientific American*, 223:120–123, October 1970.

[17] *IEEE Standard 754-2008. IEEE Standard for Floating-Point Arithmetic.* IEEE, New York, 2008.

[18] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[19] Richard Kelsey. SRFI 9: Defining Record Types. `http://srfi.schemers.org/srfi-9/`, 1999.

[20] Richard Kelsey, William Clinger, and Jonathan Rees, editors. The revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7-105, 1998. (日本語訳 translated by Suzuki Hisao, published by Yusuke Shinyama. `http://www.unixuser.org/~euske/doc/r5rs-ja/`)

[21] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[22] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161.

[23] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* 3(4):184–195, April 1960.

[24] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[25] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[26] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[27] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

[28] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[29] Jonathan Rees and William Clinger, editors. The revised[3] report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[30] Olin Shivers. SRFI 1: List Library. `http://srfi.schemers.org/srfi-1/`, 1999.

[31] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[32] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[33] Michael Sperber, R. Kent Dybvig, Mathew Flatt, and Anton van Straaten, editors. *The revised[6] report on the algorithmic language Scheme.* Cambridge University Press, 2010. (日本語訳 translated by Shiro Kawai. `http://practical-scheme.net/wiliki/wiliki.cgi?R6RS`)

[34] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[35] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[36] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[37] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

[38] Andre van Tonder. SRFI 45: Primitives for Expressing Iterative Lazy Algorithms. `http://srfi.schemers.org/srfi-45/`, 2002.

[39] Martin Gasbichler, Eric Knauel, Michael Sperber and Richard Kelsey. How to Add Threads to a Sequential Language Without Getting Tangled Up. *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, November 2003.

概念の定義，キーワード，および手続きのアルファベット順索引

それぞれの用語，手続き，またはキーワードに対し，その主要エントリを，他のエントリからセミコロンで分けて，最初に挙げる。