Revised^5 Report on the Algorithmic Language Scheme

[アルゴリズム言語 Scheme 報告書 五訂版]

RICHARD KELSEY, WILLIAM CLINGER, AND JONATHAN REES (*Editors*)

H. ABELSON	R. K. DYBVIG	C. T. HAYNES	G. J. ROZAS
N. I. ADAMS IV	D. P. FRIEDMAN	E. KOHLBECKER	G. L. STEELE JR.
D. H. BARTLEY	R. HALSTEAD	D. OXLEY	G. J. SUSSMAN
G. BROOKS	C. HANSON	K. M. PITMAN	M. WAND

Robert Hieb のみたまにささぐ

1998 年 2 月 20 日

私家訳* 1999 年 3 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，およびメッセージ渡し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記法上の規約を記述する。

第 4 章と第 5 章は式，プログラム，および定義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。

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。

目次

---

*Translated into Japanese by Hisao Suzuki <`suzuki@acm.org`>

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，λ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λ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 [28]。改訂報告書は 1978 年に現れた [25]。改訂報告書は，MIT の実装が革新的なコンパイラ [26] 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 [21, 17, 10]。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 [4] は 1985 年の夏に MIT と Indiana 大学で出版された。さらに 1986 年の春 [23] と，1988 年の春 [6] に改訂が行われた。本報告書は 1992 年 6 月の Xerox PARC での会合で合意された追加的な改訂を反映している。

謝辞

次の人々の協力に感謝します: Alan Bawden, Michael Blair, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Marc Feeley, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Hieb, Paul Hudak, Morry Katz, Chris Lindblad, Mark Meyer, Jim Miller, Jim Philbin, John Ramsdell, Mike Shaff, Jonathan Shapiro, Julie Sussman, Perry Wagle, Daniel Weise, Henry Wu, Ozan Yigit。我々は Scheme 311 version 4 参照マニュアルから文章を使用する許可をいただいたことに対して Carol Fessenden, Daniel Friedman, Christopher Haynes に感謝します。*TI Scheme Language Reference Manual* [30] から文章を使用する許可をいただいたことに対して Texas Instruments, Inc. に感謝します。我々は MIT Scheme[17], T[22], Scheme 84[11],Common Lisp[27], Algol 60[18] の各マニュアルからの影響を喜んで認めます。

我々はまたこの報告書を TEX で組むために払った多大な努力に対して Betty Dexter に，そして彼女の苦労の元となったそのプログラムを設計したことに対して Donald Knuth に感謝します。

Massachusetts Institute of Technology 人工知能研究所，Indiana 大学 計算機科学科，Oregon 大学 計算機及び情報科学科，NEC Research Institute は，この報告書の準備をサポートしました。MIT の作業に対するサポートの一部は国防省 Advanced Research Projects Agency によって海軍研究局 契約 N00014-80-C-0505 のもとで与えられました。Indiana 大学に対するサポートは NFS グラント NCS 83-04567，NCS 83-03325 によって与えられました。

言語の記述

1. Scheme の概観

1.1. 意味論

この節は 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。

Algol にならい，Scheme は静的スコープをもつプログラミング言語である。ある変数のそれぞれの使用は，その変数の字句的に見掛けどおりの束縛と結合される。

Scheme のもつ型は顕在的ではなく潜在的である。型は変数とではなく値と結合される (値はオブジェクトとも呼ばれる)。(著者によっては潜在的な型をもつ言語のことを，弱く型付けされた言語または動的に型付けされる言語と呼ぶことがある)。潜在的な型をもつ言語には他に APL，Snobol および他の Lisp 方言がある。顕在的な型をもつ言語 (強く型付けされた言語または静的に型付けされた言語といわれることもある) には Algol 60，Pacal, C がある。

Scheme の計算の途上で造られたオブジェクトは，手続きと継続 (continuation) を含め，すべて無期限の寿命 (extent) をもつ。どの Scheme オブジェクトも決して消滅させられない。ただし実装は，もしあるオブジェクトが将来のいかなる計算にも関係し得ないことを証明できるならば，そのオブジェクトが占める記憶領域を再利用してよい。これが，Scheme の実装が (たいていの場合には!) 記憶領域を使い果たさない理由である。大多数のオブジェクトが無期限の寿命をもつ言語には他に APL や他の Lisp 方言がある。

Scheme の実装は真正に末尾再帰的 (properly tail-recursive) であることが要求されている。これは，たとえ繰返し計算が構文的に再帰的手続きで記述されているときでも，定数空間でその繰返し計算を実行することを可能にする。このように真正に末尾再帰的な実装では，繰返しは通常の手続き呼出しの機構を使って表現できるから，特別な繰返しコンストラクトはただシンタックス・シュガーとして有用であるにすぎない。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。手続きを動的に造ること，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等々が可能である。これらの性質をもつ言語には他に Common Lisp や ML がある。

Scheme の一つの特長は，継続もまた “ファースト・クラス” の地位にあることである。他の大多数の言語では，継続はただ舞台裏で暗躍するにすぎない。継続は非局所的脱出，バックトラッキング，コルーチンなど広範囲の高度な制御構造を実装することに有用である。6.4 節を見よ。

Scheme の手続きへの引数はつねに値によって渡される。これは，手続きが実引数の評価結果を必要とするか否かにかかわりなく，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。ML，C，APL はつねに値によって引数を渡す他の三つの言語である。これは Haskell の遅延評価の意味論や Algol 60 の名前呼びの意味論 —それらは手続きが値を必要としない限り，引数式を評価しない— とは明確に区別される。

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，数学的理想に一致する正確算術 (exact arithmetic) と近似に基づく不正確算術の区別がある。Common Lisp と同じく，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記法を使用する。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。この単純で一様な表現の一つの重要な帰結は，他の Scheme プログラムによる取扱いを，Scheme のプログラムに対してもデータに対しても一様化できることである。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. 記法と用語

1.3.1. 原始機能，ライブラリ機能，省略可能機能

Scheme のどの実装も，省略可能 (*optional*) とマークされていない機能をすべてサポートすることが要求されている。実装は自由に，Scheme の省略可能機能を省略すること，または拡張を追加することができる。ただし，その拡張がここで報告される言語と衝突してはならない。とりわけ実装は，この報告書のどの字句規約も無効にしない構文モードを用意することによって，ポータブルなコードをサポートしなければならない。

Scheme の理解と実装を助けるため，いくつかの機能はライブラリ (*library*) とマークされている。これらはそれ以外の，原始的 (primitive) な機能によって容易に実装できる。これらは言葉の厳密な意味において冗長であるが，ありふれた慣用法のパターンをとらえており，それゆえ便利な略記法として用意されている。

1.3.2. エラー状態と未規定の振舞

エラー状態について言うとき，この報告書は “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。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，奨励されているけれども，要求されていない。実装による検出が要求されていないエラー状態は，普通ただ単に “エラー” と呼ばれる。

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引数を渡すことは，たとえそのような定義域エラーがこの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，そのような引数を含めるように手続きの定義域を拡張してもよい。

この報告書は “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せいで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されている状況を示す。もちろん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，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。この報告書は，どんな値が返されるべきかを明示的には述べない。

1.3.3. エントリの書式

4 章と 6 章はエントリごとに編成されている。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組込み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。その機能が必須の原始機能ならば，

template *category*

そうでなければ，

template *qualifier category*

ただし，ここで *qualifier* は，1.3.1 節で定義したような “ライブラリ” または “省略可能” のどちらかである。

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と理解すべきである。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。記法

<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 “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

は組込み手続き `vector-ref` 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名前が示す型でなくてはならない，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。

obj	任意のオブジェクト
$list$, $list_1$, ... $list_j$, ...	リスト (6.3.2 節参照)
z, z_1, ... z_j, ...	複素数
x, x_1, ... x_j, ...	実数
y, y_1, ... y_j, ...	実数
q, q_1, ... q_j, ...	有理数
n, n_1, ... n_j, ...	整数
k, k_1, ... k_j, ...	正確非負整数

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” と読むべきである。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，より精確には，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，つねにブーリアン値を返す手続きの名前は普通 “`?`” で終わる。このような手続きは述語 (predicate) と呼ばれる。

規約により，以前に割り付けられた場所の中へ値を格納する手続き (3.4 節参照) の名前は普通 “`!`” で終わる。このような手続きは書換え手続き (mutation procedure) と呼ばれる。規約により，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。

2. 字句規約

この節は 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。

英字の大文字と小文字は，文字定数と文字列定数の中を除いて，決して区別されない。たとえば `Foo` は `FOO` と同じ識別子であり，`#x1AB` は `#X1ab` と同じ数である。

2.1. 識別子

他のプログラミング言語で許される大多数の識別子は Scheme にも受理される。識別子をつくる詳細な規則は Scheme の実装によって異なるが，英字と数字と “拡張アルファベット文字” からなる列であって，数の先頭になり得ない文字で始まるものは，どの実装でも識別子である。さらに `+` と `-` と `...` も識別子である。下記は識別子の例である。

```
lambda                    q
list->vector              soup
+                         V17a
<=?                       a34kTMNs
the-word-recursion-has-many-meanings
```

拡張アルファベット文字は，識別子の中であたかも英字であるかのように使ってよい。以下は拡張アルファベット文字である。

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

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ってよい (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 (6.3.3 節参照)。

2.2. 空白と注釈

空白文字 (*whitespace* character) は，スペースと改行である。(実装は典型的にはタブまたは改ページなどの付加的な空白文字を定める)。空白は，可読性を改善するために使われ，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。空白は文字列の内部に出現してもよく，そこでは空白は無視されない。

セミコロン (`;`) は注釈 (comment) 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。注釈は Scheme にとって不可視だが，その行末は空白として可視である。したがって，注釈が一つの識別子または数の途中に現れることはできない。

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

2.3. その他の記法

数に対して使われる記法の記述については 6.2 節を見よ。

`. + -` これらは数で使われ，そしてまた最初の文字を除き識別子のどこにでも出現できる。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 (6.3.2 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。3 個の連続したピリオドからなる孤立した列も識別子である。

`( )` 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 (6.3.2 節)。

`'` 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)。

`` ` `` 逆引用符は準定数データを示すために使われる (4.2.6 節)。

`, ,@` 文字「コンマ」と列「コンマ」「アットマーク」は逆引用符と組み合わせて使われる (4.2.6 節)。

`"` 二重引用符は文字列を区切るために使われる (6.3.5 節)。

`\` 逆スラッシュは文字定数を表す構文で使われ (6.3.4 節)，そして文字列定数の中のエスケープ文字として使われる (6.3.5 節)。

`[ ] { } |` 左右の角カッコと波カッコと垂直バーは，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。

`#` シャープ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` これらはブーリアン定数である (6.3.1 節)。

`#\` これは文字定数の先頭となる (6.3.4 節)。

`#(` これはベクタ定数の先頭となる (6.3.6 節)。ベクタ定数は `)` を末尾とする。

#e #i #b #o #d #x これらは数の表記に使われる (6.2.4 節)。

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，構文の型の名前に，または値を格納できる場所の名前になり得る。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，その構文に束縛 (*bound*) されていると言われる。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造っ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造っ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`, `do` の各式である (4.1.4 節, 4.2.2 節, 4.2.4 節参照)。

Algol や Pascal と同じく，そして Common Lisp を除く他の大多数の Lisp の方言とは異なり，Scheme はブロック構造をもって静的にスコープが決まる言語である。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トップ・レベル環境での束縛を，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?          pair?
symbol?           number?
char?             string?
vector?           port?
procedure?
```

これらの述語が ブーリアン，ペア，シンボル，数，文字，文字列，ベクタ，ポート，手続き の各型を定義する。空リストは，それ独自の型に属する一つの特殊なオブジェクトであり，上記の述語のどれをも満たさない。

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。6.3.1 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。この報告書は “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 (6.3.2 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書いてもよい (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` (6.6.2 節) は外部表現をパースし，手続き `write` (6.6.3 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，文脈から離れては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，これはインター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，ペアやベクタや文字列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。(これらの場所は必ずしもマシンの 1 語長に対応しているとは限らない)。たとえ新しい値をこれらの場所の一つに `string-set!` 手続きを使って格納しても，文字列はあいかわらず同じ場所を表し続ける。

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。この報告書が，記憶領域を変数またはオブジェクトのために割り付ける，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，それらの場所に今や使用中であることを示すマークを付けてから，それらの場所を表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。

多くのシステムにおいて，定数 (つまりリテラル式の値) を読出し専用メモリに置くことが望ましい。これを表現するため，場所を表すオブジェクトにそれぞれ，そのオブジェクトが書換え可能 (mutable) かそれとも書換え不可能 (immutable) かを示すフラグが付けられていると想像すると便利である。このようなシステムにおいて，リテラル定数と `symbol->string` の返す文字列が書換え不可能オブジェクトである一方，この報告書に挙げられている他の手続きが造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。

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要求されている。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。もし無制限個数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ならば，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` によって以前にとらえられ，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。とらえられた継続というものがなかったならば，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 [8] に見られる。

根拠:

直観的に言って，アクティブな末尾呼出しに空間は全く必要ない。なぜなら末尾呼出しで使われる継続は，その末尾呼出しをする手続きに渡されている継続と，同じ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，ただちに元々手続きに渡されている継続への戻りへと引き継がれることになっただろう。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 インター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。この節の用語で言えば，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，彼らのインター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。末尾文脈はつねに個々のλ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λ式本体内の最後の式は末尾文脈に出現している。

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

- 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 <式> のいくつかの出現を <末尾式> で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>)

(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>)

(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` 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。

特定の組込み手続きも末尾呼出しを行うことが要求されている。`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することが，要求されていないが，許されている。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。マクロ定義が複雑な `quasiquote` を例外として，派生式はライブラリ機能に分類される。相応する定義は 7.3 節で与えられている。

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 Scheme オブジェクトの外部表現ならなんでもよい (3.3 節参照)。この記法はリテラル定数を 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> と略記してもよい。この二つの記法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，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。

```
'"abc"               ⟹  "abc"
"abc"                ⟹  "abc"
'145932              ⟹  145932
145932               ⟹  145932
'#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，呼び出される手続きとしての式と，その手続きに渡される引数としての各式を，単純に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
```

数多くの手続きが初期環境において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造られる (4.1.4 節参照)。

手続き呼出しは任意個数の値を返してよい (6.4 節 values 参照)。values を例外として，初期環境において利用可能な手続きは 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返す。

手続き呼出しはコンビネーションとも呼ばれる。

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つね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コンカレントに評価するときは，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，空のコンビネーション () は正当な式である。Scheme では，コンビネーションに少なくとも 1 個の部分式がなければならないから，() は構文的に妥当な式ではない。

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こと。<本体> は 1 個以上の式からなる列であること。

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，対応する実引数値がそれらの場所へ格納され，そして lambda 式の本体の各式が，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こと。

- (<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の束縛に格納される。
- <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，一つの新しく割り付けられたリストへと変換され，そしてそのリストが <変数> の束縛に格納される。
- (<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 (これは 1 以上でなければならない)。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しく割り付けられたリストが，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造られる手続きはそれぞれ (概念的に) 記憶場所を以 (もっ) 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> は任意の式でよい。

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 (6.3.1 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

<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> は set! 式を取り囲むなんらかの領域で，またはトップ・レベルで束縛されていなければならない。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
```

4.2. 派生式型

この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。参照のため，7.3 節は，この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マクロ定義を与えている。

4.2.1. 条件式

(cond <節$_1$> <節$_2$> ...) ライブラリ構文

構文*:* 各 <節> は次の形式であること。

(<テスト> <式$_1$> ...)

ここで <テスト> は任意の式である。あるいは，<節> は次の形式でもよい。

(<テスト> => <式>)

最後の <節> は “else 節” でもよい。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 (6.3.1 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。その値は，1 引数をとる手続きでなければならない。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> が偽値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$_2$> ...) ライブラリ構文

構文*:* <キー> はどんな式でもよい。各 <節> は次の形式をとること。

((<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。<データ> はすべて異なっていなければならない。最後の <節> は “else 節” でもよい。これは次の形式をとる。

(else <式$_1$> <式$_2$> ...).

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) 等価ならば (6.1 節参照)，対応する <節> の各式が左から右へと評価され，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。

```
(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 'consonant))         ⟹  consonant
```

(and <テスト$_1$> ...) ライブラリ構文

各 <テスト> 式が左から右へと評価され，偽値 (6.3.1 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$_1$> ...) ライブラリ構文

各 <テスト> 式が左から右へと評価され，真値 (6.3.1 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。もしすべての式が偽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 #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)
```

4.2.2. 束縛コンストラクト

三つの束縛コンストラクト `let`, `let*`, `letrec` は，Scheme に Algol 60 のようなブロック構造を与える。この三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` 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 <束縛部> <本体>)　　ライブラリ構文

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> は式である。かつ <本体> は 1 個以上の式からなる列であること。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*:* 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>)　　ライブラリ構文

構文*:* <束縛部> は次の形式をとること。

```
((<変数_1> <初期値_1>) ...),
```

かつ <本体> は 1 個以上の式からなる列であること。

意味*:* `let*` は `let` と似ているが，束縛を逐次的に左から右へと行うから，一つの (<変数> <初期値>) が示す束縛の領域は `let*` 式でその束縛から右の部分である。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。

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

(letrec <束縛部> <本体>)　　ライブラリ構文

構文*:* <束縛部> は次の形式をとること。

```
((<変数_1> <初期値_1>) ...),
```

かつ <本体> は 1 個以上の式からなる列であること。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*:* 未定義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の中で各 <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 `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ならない。もしこの制限に違反するならば，エラーである。この制限が必要なのは，Scheme が引数を名前でなく値で渡すからである。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。

4.2.3. 逐次式

(begin <式 $_1$> <式 $_2$> ...)　　ライブラリ構文

各 <式> が逐次的に左から右へと評価され，最後の <式> の値が返される。この式型は，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)

(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
```

4.2.4. 繰返し

(do ((<変数 $_1$> <初期値 $_1$> <ステップ $_1$>)　　ライブラリ構文

```
     ...)
    (<テスト> <式> ...)
  <コマンド> ...)
```

do 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 (6.3.1 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。

<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>) ライブラリ構文

“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。その構文と意味論は通常の let と同じだが，ただし，<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，という点が異なる。したがって <変数> を名前としている手続きを呼び起こすことによって，<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>) ライブラリ構文

delay コンストラクトは，遅延評価 (*lazy evaluation*) つまり *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かなえることを (force 手続きによって) 求められるかもしれない。複数個の値を返す <式> の効果は未規定である。

delay のより完全な記述については force の記述 (6.4 節) を見よ。

4.2.6. 準引用

(quasiquote <qq テンプレート>) 構文

\`<qq テンプレート> 構文

“バッククォート” 式 つまり “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ほとんど分かっているが，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。もしコンマの直後がアットマーク (@) 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，リストまたはベクタの <qq テンプレート> の中にだけ現れることとする。

```
`(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)
```

準引用形式は入れ子にしてもよい。置換は，最外のバッククォート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)
```

`<qq テンプレート> と (quasiquote <qq テンプレート>) の二つの記法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。これらのシンボルの一つを car とする 2 要素リストに対して write が生成する外部構文は，実装ごとに異なってもよい。

```
(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))
```

もしシンボル 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，予測できない振舞が結果として得られるかもしれない。

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束縛を隠蔽してもよい。 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 [14, 15, 2, 7, 9]。つまり，

- 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。トップ・レベルでの define は束縛を導入するかもしれないし，しないかもしれないことに注意せよ (5.2 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。

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 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。構文キーワードをトップ・レベルで束縛することもできる。5.3 節を見よ。

(let-syntax <束縛部> <本体>) 構文

構文: <束縛部> は次の形式をとること。

```
((<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 ((when (syntax-rules ()
                     ((when test stmt1 stmt2 ...)
                      (if test
                          (begin stmt1
                                 stmt2 ...))))))
  (let ((if #t))
    (when 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 <束縛部> 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> は次の形式をとる。

(syntax-rules <リテラル部> <構文規則> ...)

構文*:* <リテラル部> は識別子の並びである。各 <構文規則> は次の形式であること。

```
(<パターン> <テンプレート>)
```

<構文規則> におかれる <パターン> は，定義されるマクロのキーワードで始まるリスト <パターン> である。

一般に <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>)
```

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>)
#(<要素> ...)
```

ここで <要素> は一つの <テンプレート>，またはそれに一つの <省略符号> を続けたものである。<省略符号> は識別子 “`...`” である (これはテンプレートやパターンの中で識別子として使うことはできない)。

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。

<構文規則> のパターンに現れる識別子はパターン変数である。ただし，それがそのパターンを始めるキーワードであるか，<リテラル部> に挙げられているか，または識別子 “`...`” である場合を除く。パターン変数は，任意の入力要素と照合する。パターン変数は，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。

<構文規則> のパターンの先頭にあるキーワードは，照合に関与しない。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。

根拠*:* キーワードのスコープは，キーワードをマクロ変換子に束縛して結合させる式ないし構文定義によって決定される。もしも仮にキーワードがパターン変数またはリテラル識別子だとすると，`let-syntax` と `letrec-syntax` のどちらがキーワードを束縛するかに関係なく，そのスコープの中に，パターンに後続するテンプレートが含まれることになっただろう。

<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部分形式と照合されることになる。入力における部分形式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等しくかつどちらも字句的束縛をもたないことである。

部分形式に `...` が後続するとき，その部分形式は入力の 0 個以上の要素と照合できる。`...` が <リテラル部> に現れることはエラーである。パターンの内部では識別子 `...` は，部分パターンの非空な列の最後の要素に後続しなければならない。

より形式的には，入力形式 F がパターン P と照合するのは，次が成り立つときかつそのときに限る。

- 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 F 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 F が，P_1 から P_n までそれぞれ照合する n 個の形式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 F が，P_1 から P_n までそれぞれ照合する n 個以上の形式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 “cdr” が P_{n+1} に照合する。または
- P が (P_1 ... P_n P_{n+1} <省略符号>)，ただしここで <省略符号> は識別子 `...` とする，という形式をとり，かつ F が，最初の n 個が P_1 から P_n までそれぞれ照合する少なくとも n 個の形式からなる真正リストであって，かつ F の残りの要素がおのおの P_{n+1} に照合する。または

- P が #(P_1 ... P_n) という形式のベクタであり，かつ F が，P_1 から P_n まで照合する n 個の形式からなるベクタである。または
- P が #(P_1 ... P_n P_{n+1} <省略符号>)，ただしここで <省略符号> は識別子 ... とする，という形式をとり，かつ F が，最初の n 個が P_1 から P_n までそれぞれ照合する少なくとも n 個の形式からなるベクタであって，かつ F の残りの要素がおのおの P_{n+1} に照合する。または
- P がデータであり，かつ F 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部分形式で置き換えられる。識別子 ... 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 ... 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部分形式全部によって，ただし指示どおりに分配されて，置き換えられる。もし仕様どおりに出力を形成できないならばエラーである。

テンプレートに現れているがパターン変数でも識別子 ... 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捕獲を防ぐために実質的に改名される。

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トップ・レベル識別子 => 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。

5. プログラム構造

5.1. プログラム

Scheme プログラムは式，定義，および構文定義の列からなる。式は 4 章で記述している。定義と構文定義は本章の残りで論ずる。

プログラムは典型的にはファイルに格納されるか，または実行中の Scheme システムに対話的に入力されるが，他のパラダイムも可能である。しかし，ユーザ・インタフェースの問題はこの報告書の範囲外である。(実際，Scheme はたとえ機械上の実装がなくても計算方法を表現する記法として依然有用だろう)。

プログラムのトップ・レベルに出現する定義と構文定義は，宣言的に解釈され得る。それらは束縛をトップ・レベル環境の中に造らせるか，または既存のトップ=レベル束縛の値を改変する。プログラムのトップ・レベルに出現する式は，命令的に解釈される。それらは，プログラムが起動またはロードされたときに順に実行され，そして典型的にはなんらかの種類の初期化を行う。

プログラムのトップ・レベルでは (begin <形式$_1$> ...) は，その begin の本体をつくる式，定義，および構文定義からなる列と等価である。

5.2. 定義

定義は，式が許される文脈の，全部ではなく一部において妥当である。定義は，<プログラム> のトップ・レベルと <本体> の先頭でだけ妥当である。

定義は次の形式の一つをとること。

- (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こと (λ 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と。この形式は次と等価である。

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

5.2.1. トップ・レベル定義

プログラムのトップ・レベルでは，定義

```
(define <変数> <式>)
```

は，もし <変数> が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
```

Scheme の実装によっては，初期環境として，可能なあらゆる変数がなんらかの場所に束縛されており，そしてその場所の大多数が未定義値を入れている，という環境を使う。このような実装ではトップ・レベル定義は代入と全く等価である。

5.2.2. 内部定義

定義は <本体> (つまり lambda, let, let*, letrec let-syntax, letrec-syntax の各式の本体，または適切な形式の定義の本体) の先頭に出現してもよい。このような定義を，上述のトップ・レベ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 <本体> はつね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> の中の各内部定義のそれぞれの <式> は，そこで定義されるどの <変数> の値の代入も参照もすることなく評価可能でなければならない。

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. 構文定義

構文定義は <プログラム> のトップ・レベルでだけ妥当である。 構文定義は次の形式をとる。

```
(define-syntax <キーワード> <変換子仕様>)
```

<キーワード> は識別子であり，そして <変換子仕様> は syntax-rules のインスタンスであること。<キーワード> を，仕様が規定する変換子へと束縛することによって，トップ・レベル構文環境が拡張される。

define-syntax には，内部定義に相当するものはない。

マクロは，定義や構文定義を許す文脈の中ならばどこでも，定義や構文定義へと展開してよい。ただし，定義や構文定義が構文キーワードを隠蔽することは，隠蔽するほうの定義を収めている形式のグループの中のどれかの形式が実際に定義であるかどうかを決定するために，あるいは，内部定義に対して，そのグループとそのグループに続く式との境界を決定するために，隠蔽される構文キーワードの意味が必要とされる場合には，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)))
```

6. 標準手続き

この章は Scheme の組込み手続きを記述する。初期の (つまり “トップ・レベル” の) Scheme 環境は，有用な値を収めている場所へと束縛された数多くの変数とともに始まる。そして，その値の大多数はデータを操作するプリミティブ手続きである。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 (を初期値として収めている場所) へと束縛され，変数 + は数の和を計算する手続きへと束縛される。他の組込み手続きを使って容易に書くことのできる組込み手続きは，“ライブラリ手続き” として分類される。

プログラムはトップ=レベル定義を使って任意の変数を束縛してよい。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 Scheme の組込み手続きの振舞を改変しない。定義によって導入されていないトップ=レベルの束縛を変更することは，組込み手続きの振舞に対して未規定の効果をもつ。

6.1. 等価性述語

述語 (*predicate*) とは，いつでもブーリアン値 (`#t` または `#f`) を返す手続きである。等価性述語 (*equivalence predicate*) とは，数学的な等価関係の計算機上の相当物である (対称律と反射律と推移律が成り立つ)。この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?` が最も粗い。`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してよいものならば，`#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

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                            ⟹  #t
```

  注: これは obj_1 と obj_2 のどちらも，6.3.3 節でほのめかされているような “未インターンのシンボル” ではないことを仮定している。この報告書は，実装依存の拡張に対する `eqv?` の振舞をあえて規定するようなことはしない。

- obj_1 と obj_2 がともに数であって, 数値的に等しく (6.2 節の `=` を見よ)，かつともに正確数またはともに不正確数である。
- obj_1 と obj_2 がともに文字であって，かつ `char=?` 手続き (6.3.4 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，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

```
(string=? (symbol->string obj1)
          (symbol->string obj2))
                            ⟹  #f
```

- 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 がともに数であって，`=` 手続きが `#f` を返す。
- 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ない。
- obj_1 と obj_2 がともにペア，ベクタ，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? '() '())               ⟹  #t
(eqv? 100000000 100000000)   ⟹  #t
(eqv? (cons 1 2) (cons 1 2)) ⟹  #f
(eqv? (lambda () 1)
      (lambda () 2))         ⟹  #f
(eqv? #f 'nil)               ⟹  #f
(let ((p (lambda (x) x)))
  (eqv? p p))                ⟹  #t
```

下記の例は，上記の規則が `eqv?` の振舞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
```

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` 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` は毎回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。

```
(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だから，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は，要求はされていないけれども，許されている。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
```

根拠: 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うことも自由であり，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。

(eq? *obj*$_1$ *obj*$_2$) 手続き

eq? 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。

eq? と eqv? はシンボル，ブーリアン，空リスト，ペア，手続き，および非空の文字列とベクタに対して同じ振舞をすることが保証されている。数と文字に対する eq? の振舞は実装依存だが，いつでも真か偽を返し，そして真を返すようなときは，eqv? もまた真を返す。eq? は空ベクタと空文字列に対しても eqv? と違う振舞をしてよい。

```
(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いつでも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必ずしも可能とは限らないからである。手続きを使って状態付きオブジェクトを実装するような応用においては，eq? は eqv? と同じ制約に従うから，eq? を eqv? のように使ってもよい。

(equal? *obj*$_1$ *obj*$_2$) ライブラリ手続き

equal? は再帰的にペア，ベクタ，および文字列の内容を比較し，それ以外の数やシンボルなどのオブジェクトに対しては eqv? を適用する。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。もし引数が循環データ構造ならば，equal? は停止しないかもしれない。

```
(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? (lambda (x) x)
        (lambda (y) y))       ⟹  未規定
```

6.2. 数

数値計算は伝統的に Lisp コミュニティから無視されてきた。Common Lisp 以前には数値計算を体系化するための熟慮された戦略は皆無だった。そして MacLisp システム [20] を例外として，数値コードを効率的に実行するための努力はほとんどなされなかった。この報告書は Common Lisp 委員会の労作を高く認め，彼らの勧告の多くを受け入れる。見方によっては，この報告書は彼らの提案を Scheme の目的に沿うように単純化し一般化している。

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法，この四者を区別することが大切である。この報告書は *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。固定小数点数や浮動小数点数のような機械表現は，*fixnum* や *flonum* のような名前によって言い表される。

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編成できる:

number (数)
complex (複素数)
real (実数)
rational (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 fixnum や flonum，あるいはその他の数の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。

しかし，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必要である。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。

6.2.2. 正確性

Scheme の数は *exact* (正確) か *inexact* (不正確) かのいずれかである。もしも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，その数は正確数である。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，その数は不正確数である。このように不正確性は，数のもつ伝染性の性質である。

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価になるだろう。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，いつでも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。6.2.3 節を見よ。

`inexact->exact` を例外として，この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。

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要求されていないが，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，この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，flonum を使ってすべての不正確実数を表現するような実装が，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) flonum 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，このレンジ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，リストやベクタや文字列の添字として使われ得るか，またはリストやベクタや文字列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`, 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るものとする。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，いつでも正確整数の結果を返すものとする:

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

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，正確な引数が与えられたときはいつでも正確な結果を返すように上記の手続きと / 手続きを実装すること，この二つが実装に対して—要求はされていないが—奨励されている。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からエラーの原因となるかもしれない。

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。この報告書は，flonum 表現を使う実装が IEEE 32-bit and 64-bit floating point standards に従うこと，そして他の表現を使う実装はこれらの浮動小数点規格 [12] を使って達成可能な精度と互角以上であることを—要求はしないが—推奨する。

とりわけ，flonum 表現を使う実装は次の規則に従わなければならない: flonum 結果はその演算への不正確引数のどれを表現するために使われた精度とも，少なくとも同等の精度で表現されなければならない。`sqrt` のような潜在的に不正確な演算は，正確引数に適用されたとき，可能ならばいつでも正確な答えを算出する (たとえば，正確数 4 の平方根—square root—は正確数 2 になるべきである) ことが望ましい (が要求はされていない)。しかしながら，もしも (`sqrt` でのごとく) 正確数を演算した結果が不正確数になってしまうならば，そしてその不正確数が flonum として表現されるならば，そのときは，利用可能な最も高精度の flonum 形式が使われなければならない。しかし，もしも結果がなにか別の方法で表現されるならば，そのときは，その表現は利用可能な最も高精度の flonum 形式と少なくとも同等の精度でなければならない。

Scheme は数に対して様々な表記法を許しているが，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および極座標の記法をサポートする必要はない。もし実装が，正確数として表現できないような正確数値定数に出会ったならば，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. 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，または十六進で書かれ得る。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，十進数での表現だと仮定される。

数値定数は，接頭辞により正確か不正確かを指定され得る。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) はその前に現れても後ろに現れてもよい。もし数の表記表現に正確性接頭辞がなければ，その定数は不正確でも正確でもあり得る。もしそれが小数点，指数部，または数字代わりの “#” 文字を含んでいるならば不正確数であり，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，数値定数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得る。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。(不正確数の内部表現が四種類に満たないとき，この四つのサイズ指定は利用可能な指定の上へと写像される。たとえば，二種類の内部表現をもった実装は short と single，および long と double をそれぞれ一つに写してよい。) なお，指数部マーカ `e` は実装のデフォルト精度を指定する。デフォルト精度は *double* と少なくとも同等の精度だが，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
```

6.2.5. 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。この節における用例は，正確記法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，不正確記法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 flonum 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 を不正確複素数とするとき，(`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。

```
(complex? 3+4i)        ⟹  #t
(complex? 3)           ⟹  #t
(real? 3)              ⟹  #t
(real? -2.5+0.0i)      ⟹  #t
(real? #e1e10)         ⟹  #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 `rational?` 手続きは `real?` と同一になり，`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できるかもしれないし，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いるかもしれない。

`(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。

`(= z1 z2 z3 ...)` 手続き
`(< x1 x2 x3 ...)` 手続き
`(> x1 x2 x3 ...)` 手続き
`(<= x1 x2 x3 ...)` 手続き
`(>= x1 x2 x3 ...)` 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` を返す。

これらの述語は推移的であることが要求されている。

注: Lisp ライクな言語におけるこれらの述語の伝統的な実装は推移的ではない。

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かもしれない。なぜなら小さな誤差が結果に影響するかもしれないからである。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)` ライブラリ手続き
`(positive? x)` ライブラリ手続き
`(negative? x)` ライブラリ手続き
`(odd? n)` ライブラリ手続き
`(even? n)` ライブラリ手続き

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 x1 x2 ...)` ライブラリ手続き
`(min x1 x2 ...)` ライブラリ手続き

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。

`(+ z1 ...)` 手続き
`(* z1 ...)` 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
```

`(- z1 z2)` 手続き
`(- z)` 手続き
`(- z1 z2 ...)` 省略可能手続き
`(/ z1 z2)` 手続き
`(/ z)` 手続き
`(/ z1 z2 ...)` 省略可能手続き

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。

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

`(abs x)` ライブラリ手続き

`abs` は引数の絶対値 (absolute value) を返す。

```
(abs -7)     ⟹  7
```

`(quotient n1 n2)` 手続き
`(remainder n1 n2)` 手続き
`(modulo n1 n2)` 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 は非ゼロであること。三つすべての手続きが整数を返す。もしも n_1/n_2 が整数ならば:

```
(quotient n1 n2)     ⟹  n1/n2
(remainder n1 n2)    ⟹  0
(modulo n1 n2)       ⟹  0
```

もしも n_1/n_2 が整数でなければ:

```
(quotient n1 n2)     ⟹  nq
(remainder n1 n2)    ⟹  nr
(modulo n1 n2)       ⟹  nm
```

ここで n_q はゼロ方向に丸めた n_1/n_2, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r と n_m は n_1 から n_2 の倍数だけ異なる, n_r は n_1 と同符号, かつ n_m は n_2 と同符号とする。

このことから私たちは整数 n_1 と整数 n_2 ($\neq 0$) に対して次を結論できる:

```
(= n1 (+ (* n2 (quotient n1 n2))
      (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。

```
(modulo 13 4)                ⟹  1
(remainder 13 4)             ⟹  1

(modulo -13 4)               ⟹  3
(remainder -13 4)            ⟹  -1

(modulo 13 -4)               ⟹  -3
(remainder 13 -4)            ⟹  1

(modulo -13 -4)              ⟹  -1
(remainder -13 -4)           ⟹  -1

(remainder -13 -4.0)         ⟹  -1.0  ; 不正確数
```

(gcd n_1 ...) ライブラリ手続き
(lcm n_1 ...) ライブラリ手続き

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。結果はつねに非負である。

```
(gcd 32 -36)                 ⟹  4
(gcd)                        ⟹  0
(lcm 32 -36)                 ⟹  288
(lcm 32.0 -36)               ⟹  288.0  ; 不正確数
(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。分母はつねに正である。 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
  (exact->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` は x 以下の最大の整数を返す。`ceiling` 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。`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` が偶数へと丸めることは，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 `inexact->exact` 手続きに渡せばよい。

```
(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) ライブラリ手続き

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

(exp z) 手続き
(log z) 手続き
(sin z) 手続き
(cos z) 手続き
(tan z) 手続き
(asin z) 手続き
(acos z) 手続き
(atan z) 手続き
(atan y x) 手続き

これらの手続きは，一般的な実数をサポートするどの実装にも含まれる。これらはおなじみの超越関数を計算する。`log` は z の自然対数を計算する (常用対数ではない)。`asin`, `acos`, および `atan` はそれぞれ arcsine ($\sin^{-1}$), arccosine ($\cos^{-1}$), 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, arccosine, および arctangent は多価関数として定義される。しかし，ここでは $\log z$ の値として，虚部が区間 $(-\pi, \pi]$ 内に存在するような値を採ることにする。$\log 0$ は未定義とする。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:

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

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

上記の規定は [27] に従っており，そしてそれは翻って [19] を引用している。これらの関数の branch cut，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。可能ならばいつでも，これらの手続きは実数引数から実数結果を算出する。

(sqrt z) 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。

(expt z_1 z_2) 手続き

z_1 の z_2 乗を返す。$z_1 \neq 0$ に対して

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

0^z は，$z = 0$ ならば 1，そうでなければ 0 である。

(make-rectangular x_1 x_2) 手続き
(make-polar x_3 x_4) 手続き
(real-part z) 手続き
(imag-part z) 手続き
(magnitude z) 手続き
(angle z) 手続き

これらの手続きは，一般的な複素数をサポートするどの実装にも含まれる。x_1, x_2, x_3, および x_4 が実数であり，z が複素数であって，次のようであると仮定するとき，

$$z = x_1 + x_2 i = x_3 \cdot e^{i x_4}$$

次が成り立つ。

(make-rectangular x_1 x_2)	$\Longrightarrow z$
(make-polar x_3 x_4)	$\Longrightarrow z$
(real-part z)	$\Longrightarrow x_1$
(imag-part z)	$\Longrightarrow x_2$
(magnitude z)	$\Longrightarrow \lvert x_3 \rvert$
(angle z)	$\Longrightarrow x_{angle}$

ここで $-\pi < x_{angle} \leq \pi$，ただし，ある整数 n に対して $x_{angle} = x_4 + 2\pi n$ とする。

根拠: magnitude は実数引数に対して abs と同一である。しかし，abs がすべての実装になければならない一方，magnitude は一般的な複素数をサポートする実装にだけあればよい。

(exact->inexact z) 手続き
(inexact->exact z) 手続き

exact->inexact 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。もしも正確引数にほどよく近い不正確数がなければ，実装制限の違反を報告してもよい。

inexact->exact 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。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。

6.2.6. 数値入出力

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

radix は正確整数 2, 8, 10, または 16 でなければならない。省略時，*radix* は 10 と見なされる。手続き number->string は数 (*number*) と基数 (*radix*) を取って，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。

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 [3, 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 flonum を使って表現された不正確数であり，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, および非 flonum 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に精度の高い表現の数を返す。*radix* は正確整数 2, 8, 10, または 16 でなければならない。*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。もし *string* が構文的に妥当な数の表記でなければ，string->number は #f を返す。

(string->number "100")	$\Longrightarrow$ 100
(string->number "100" 16)	$\Longrightarrow$ 256
(string->number "1e2")	$\Longrightarrow$ 100.0
(string->number "15##")	$\Longrightarrow$ 1500.0

注: `string->number` の定義域を実装は次のように制限してもよい。`string->number` は，*string* が明示的な基数接頭辞を含んでいた場合に対してつねに `#f` を返すことが許されている。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* が複素数に対する局座標記法または直交座標記法を使った場合に対してつねに `#f` を返すことが許されている。もしもすべての数が整数ならば，`string->number` は，分数記法が使われた場合に対してつねに `#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つねに，あるいは `#` が数字の位置に現れたならば，`#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つねに `#f` を返してよい。

6.3. 他のデータ型

この節は Scheme の非数値データ型のいくつか—ブーリアン，ペア，リスト，シンボル，文字，文字列およびベクタ—についての演算を記述する。

6.3.1. 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` と `#f` と書かれる。しかし，実際に問題になるのは，Scheme の条件式 (if, cond, and, or, 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。

すべての標準的な 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。`#f` を除いて，すべての標準的な Scheme 値は—`#t`，ペア，空リスト，シンボル，数，文字列，ベクタ，および手続きを含め—真と見なされる。

注: Lisp の他の方言に慣れているプログラマは，Scheme が `#f` と空リストをともにシンボル `nil` から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*) ライブラリ手続き

`not` 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*) ライブラリ手続き

`boolean?` 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
```

6.3.2. ペアとリスト

ペア *pair* (ときには点対 *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` によって造られる。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。より精確には，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，それ独自の型に属する一つの特殊なオブジェクトである (ペアではない)。空リストに要素はなく，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記法 (外部表現) は，"点対" 記法 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，より能率的な記法が使える。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記法である。

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。リスト記法と点対記法を組み合わせて，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，ということができる:

```
(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?` 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

一つの新しく割り付けられたペアを返す。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。`set-car!` が返す値は未規定である。

```
(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。`set-cdr!` が返す値は未規定である。

`(caar ` *pair*`)` ライブラリ手続き
`(cadr ` *pair*`)` ライブラリ手続き
⋮ ⋮
`(cdddar ` *pair*`)` ライブラリ手続き
`(cddddr ` *pair*`)` ライブラリ手続き

これらの手続きは `car` と `cdr` の合成である。ここで例えば `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。これらの手続きは全部で 28 個ある。

`(null? ` *obj*`)` ライブラリ手続き

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*`)` ライブラリ手続き

もし *obj* がリストならば `#t` を返し，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
```

`(list `*obj* `...)` ライブラリ手続き

その引数からなる，一つの新しく割り付けられたリストを返す。

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

`(length `*list*`)` ライブラリ手続き

list の長さを返す。

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

`(append `*list* `...)` ライブラリ手続き

最初の *list* の要素に，ほかの各 *list* の要素を続けたもので構成されたリストを返す。

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

結果のリストはつねに新しく割り付けられるが，ただし最後の *list* 引数とは構造を共有する。最後の引数は，実のところ，どんなオブジェクトでもよい。もしも最後の引数が真正リストでなければ，結果は非真正リストである。

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

`(reverse `*list*`)` ライブラリ手続き

逆順にした *list* の要素で構成された，一つの新しく割り付け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*`)` ライブラリ手続き

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。もしも *list* の要素数が *k* より少なければエラーである。`list-tail` 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*`)` ライブラリ手続き

list の第 *k* 要素を返す (これは `(list-tail `*list* *k*`)` の car と同じである)。もしも *list* の要素数が *k* より少なければエラーである。

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

`(memq `*obj* *list*`)` ライブラリ手続き
`(memv `*obj* *list*`)` ライブラリ手続き
`(member `*obj* *list*`)` ライブラリ手続き

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*`)` が返す非空リストである。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` 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 `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)
(memq 101 '(100 101 102))     ⟹  未規定
(memv 101 '(100 101 102))     ⟹  (101 102)
```

`(assq `*obj* *alist*`)` ライブラリ手続き
`(assv `*obj* *alist*`)` ライブラリ手続き
`(assoc `*obj* *alist*`)` ライブラリ手続き

alist (つまり，連想リスト “association list”) は，ペアからなるリストでなければならない。 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` 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 `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))
```

```
(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，通常は述語として使われるが，しかし，単なる #t や #f だけではない有用な値を返すから，名前に疑問符を付けない。

6.3.3. 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。これはまさに，プログラムで識別子を表現するために必要とされる性質であり，したがって Scheme の大多数の実装はシンボルを内部的にその目的のために利用している。シンボルは他の多くの応用にも有用である。たとえば，列挙値を Pascal 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。ただし，string->symbol 手続きは，名前に特殊文字や非標準ケースの英字が含まれているのでこの write/read 不変性が必ずしも成り立たない，というシンボルを造ることができる。

注: Scheme の実装によっては，すべてのシンボルに対する write/read 不変性を保証するため，“slashification” として知られる機能をもつものがあるが，歴史的にいえば，この機能の最も重要な用途は文字列データ型の欠如に対する埋め合わせであった。

実装によってはまた “未インターンのシンボル” をもつものがある。これは slashification 付きの実装においてさえも 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例外をもたらす。

(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->string *symbol*) 手続き

symbol の名前を一つの文字列として返す。もしもシンボルが，リテラル式 (4.1.2 節) の値として，あるいは read 手続きへの呼出しによって，返されたオブジェクトの一部だったならば，かつその名前にアルファベット文字が含まれているならば，そのとき返される文字列は，実装の選択した標準ケースの文字を含んでいるだろう—大文字を選択する実装もあれば，小文字を選択する実装もあるだろう。もしもシンボルが string->symbol によって返されたものだったならば，返される文字列の文字のケースは，string->symbol へ渡された文字列のケースと同じになるだろう。string-set! のような書換え手続きを，この手続きが返す文字列へ適用することはエラーである。

以下の例は，実装の標準ケースが小文字であることを仮定している:

```
(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，特殊文字や非標準ケースの英字を含んでいる名前をもったシンボルを造ることができるが，Scheme の実装によってはそのようなシンボルはそれ自身として read できないから，そういうシンボルを造ることは普通は悪いアイディアである。symbol->string を見よ。

以下の例は，実装の標準ケースが小文字であることを仮定している:

```
(eq? 'mISSISSIppi 'mississippi)
        ⟹ #t
(string->symbol "mISSISSIppi")
        ⟹ "mISSISSIppi" という名前のシンボル
(eq? 'bitBlt (string->symbol "bitBlt"))
        ⟹ #f
(eq? 'JollyWog
     (string->symbol
       (symbol->string 'JollyWog)))
        ⟹ #t
(string=? "K. Harper, M.D."
        (symbol->string
          (string->symbol "K. Harper, M.D.")))
        ⟹ #t
```

6.3.4. 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。文字は #\<文字> または #\<文字名> という記法を使って書かれる。たとえば:

```
#\a         ; 英小文字
#\A         ; 英大文字
#\(         ; 左丸カッコ
#\          ; スペース文字
#\space     ; スペースを書くための好ましい方法
#\newline   ; 改行文字
```

ケースは #\<文字> では意味をもつが，#\<文字名> では意味をもたない。もしも #\<文字> の <文字> がアルファベットならば，<文字> の後に続く文字は，スペースやカッコなどの区切り文字でなければならない。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。

#\ 記法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，大文字と小文字のケースの違いを無視する。ケース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=? $char_1$ $char_2$) 手続き
(char<? $char_1$ $char_2$) 手続き
(char>? $char_1$ $char_2$) 手続き
(char<=? $char_1$ $char_2$) 手続き
(char>=? $char_1$ $char_2$) 手続き

これらの手続きは文字の集合に全順序を課している。この順序のもとで保証されることは:

- 英大文字どうしはその順序どおりである。たとえば，(char<? #\A #\B) は #t を返す。
- 英小文字どうしはその順序どおりである。たとえば，(char<? #\a #\b) は #t を返す。
- 数字どうしはその順序どおりである。たとえば，(char<? #\0 #\9) は #t を返す。
- すべての数字は，すべての英大文字よりも前であるか，または後である。
- すべての数字は，すべての英小文字よりも前であるか，または後である。

実装は，これらの手続きを，それと対応する数値述語と同じく，2 個以上の引数をとるように一般化してもよい。

(char-ci=? $char_1$ $char_2$) ライブラリ手続き
(char-ci<? $char_1$ $char_2$) ライブラリ手続き
(char-ci>? $char_1$ $char_2$) ライブラリ手続き
(char-ci<=? $char_1$ $char_2$) ライブラリ手続き
(char-ci>=? $char_1$ $char_2$) 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。実装は，これらの手続きを，それと対応する数値述語と同じく，2 個以上の引数をとるように一般化してもよい。

(char-alphabetic? *char*) ライブラリ手続き
(char-numeric? *char*) ライブラリ手続き
(char-whitespace? *char*) ライブラリ手続き
(char-upper-case? *letter*) ライブラリ手続き
(char-lower-case? *letter*) 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。次の注釈は，ASCII 文字集合に固有であり，一つの指針としてだけ意図されている: アルファベット文字は大小あわせ 52 個の英字である。数値文字は 10 個の十進数字である。空白文字はスペース，タブ，改行，改頁，および復帰文字である。

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

文字が与えられたとき，char->integer はその文字の一つの正確整数表現を返す。ある文字の char->integer による像 (image) である正確整数が与えられたとき，integer->char はその文字を返す。これらの手続きは，char<=? 順序のもとにある文字の集合と，<= 順序のもとにある整数のある部分集合との間に，順序保存的な同形写像を実装している。すなわち，もしも

```
(char<=? a b) ⟹ #t   かつ   (<= x y) ⟹ #t
```

であって，かつ *x* と *y* が integer->char の定義域にあるならば，そのとき

```
(<= (char->integer a)
    (char->integer b))      ⟹  #t

(char<=? (integer->char x)
         (integer->char y)) ⟹  #t
```

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

これらの手続きは，(char-ci=? *char* $char_2$) を満たす文字 $char_2$ を返す。さらに，もしも *char* がアルファベットならば，char-upcase の結果は大文字であり，char-downcase の結果は小文字である。

6.3.5. 文字列

文字列 (string) とは，文字の列 (sequence of characters) である。文字列は，二重引用符 (") で囲まれた文字の列として書かれる。二重引用符は，それを逆スラッシュ (\) でエスケープすることによってのみ文字列の内部に書ける。たとえば

```
"The word \"recursion\" has many meanings."
```

逆スラッシュは，それをもう一つの逆スラッシュでエスケープすることによってのみ文字列の内部に書ける。文字列の内部の逆スラッシュであって，その後ろに二重引用符も逆スラッシュも続かないものの効果を Scheme は規定しない。

文字列は一つの行から次の行へと継続してもよいが，そのような文字列の正確な内容は未規定である。

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，文字列が造ら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。

“添字 *start* で始まり，添字 *end* で終わる *string* の文字” というような言い回しでは，添字 *start* はその文字に含まれるが，添字 *end* は含まれないと理解される。したがってもしも *start* と *end* が同じ添字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。

文字列について演算する手続きのいくつかは，大文字と小文字のケースの違いを無視する。ケースを無視するバージョンは，その名前に “-ci” (つまり “case insensitive”) を埋め込んでいる。

(**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，一つの新しく割り付けられた，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要素が *char* に初期化されるが，そうでなければ *string* の内容は未規定である。

(**string** *char* ...) ライブラリ手続き

引数から構成された，一つの新しく割り付けられた文字列を返す。

(**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 は *string* の妥当な添字でなければならない。`string-ref` は *string* の，ゼロから数えて第 *k* の文字を返す。

(**string-set!** *string* *k* *char*) 手続き

k は *string* の妥当な添字でなければならない。`string-set!` は *string* の要素 *k* に *char* を格納して，未規定の値を返す。

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

もしも二つ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` を返すが，そうでなければ `#f` を返す。`string-ci=?` は英大文字と英小文字をあたかも同じ文字であるかのように扱うが，`string=?` は大文字と小文字を異なった文字として扱う。

(**string<?** $string_1$ $string_2$) ライブラリ手続き
(**string>?** $string_1$ $string_2$) ライブラリ手続き
(**string<=?** $string_1$ $string_2$) ライブラリ手続き
(**string>=?** $string_1$ $string_2$) ライブラリ手続き
(**string-ci<?** $string_1$ $string_2$) ライブラリ手続き
(**string-ci>?** $string_1$ $string_2$) ライブラリ手続き
(**string-ci<=?** $string_1$ $string_2$) ライブラリ手続き
(**string-ci>=?** $string_1$ $string_2$) ライブラリ手続き

これらの手続きは，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である。たとえば，`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る。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。

実装は，これらおよび `string=?` と `string-ci=?` の手続きを，それと対応する数値述語と同じく，2 個以上の引数をとるように一般化してもよい。

(**substring** *string* *start* *end*) ライブラリ手続き

string は文字列でなければならず，*start* と *end* は下記を満たす正確整数でなければならない。

$$0 \le start \le end \le (\texttt{string-length}\ string).$$

`substring` は，*string* の，添字 *start* で始まり (境界を含む)，添字 *end* で終わる (境界を含まない) 文字から構成された，一つの新しく割り付けられた文字列を返す。

(**string-append** *string* ...) ライブラリ手続き

与えられた文字列の連結をその文字が構成している，一つの新しく割り付けられた文字列を返す。

(**string->list** *string*) ライブラリ手続き
(**list->string** *list*) ライブラリ手続き

`string->list` は，与えられた文字列を構成している文字からなる，一つの新しく割り付けられたリストを返す。`list->string` は，リスト *list*—これは文字からなるリストでなけ

ればならない—の文字から構成された，一つの新しく割り付けられた文字列を返す。`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*) ライブラリ手続き

与えられた *string* の，一つの新しく割り付けられたコピーを返す。

(`string-fill!` *string char*) ライブラリ手続き

与えられた *string* の各要素に *char* を格納して，未規定の値を返す。

6.3.6. 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。ベクタは典型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要される平均時間は典型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，ベクタが造ら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。

ベクタは記法 `#(`*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")
```

これはベクタの外部表現であるが，ベクタへと評価される式ではないことに注意せよ。リスト定数と同じく，ベクタ定数もクォートしなければならない:

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

(`vector?` *obj*) 手続き

もし *obj* がベクタならば `#t` を返し，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，一つの新しく割り付け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* ...) ライブラリ手続き

与えられた引数を要素として持つ，一つの新しく割り付けられたベクタを返す。`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 は *vector* の妥当な添字でなければならない。`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)
            (let ((i (round (* 2 (acos -1)))))
              (if (inexact? i)
                  (inexact->exact i)
                  i)))
          ⟹ 13
```

(`vector-set!` *vector k obj*) 手続き

k は *vector* の妥当な添字でなければならない。`vector-set!` は *vector* の要素 *k* に *obj* を格納する。`vector-set!` の返す値は未規定である。

```
(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*) ライブラリ手続き
(`list->vector` *list*) ライブラリ手続き

`vector->list` は，*vector* の要素として含まれているオブジェクトからなる，一つの新しく割り付けられたリストを返す。`list->vector` は，リスト *list* の要素へと初期化された，一つの新しく割り付けられたベクタを返す。

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

(`vector-fill!` *vector fill*) ライブラリ手続き

vector の各要素に *fill* を格納する。`vector-fill!` の返す値は未規定である。

6.4. 制御機能

この章は，プログラム実行の流れを特殊な仕方で制御する様々なプリミティブ手続きを記述する。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 は手続きでなければならず，*args* はリストでなければならない。*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* $list_1$ $list_2$ ...) ライブラリ手続き

各 *list* はリストでなければならず，*proc* は *list* の個数と同じだけの個数の引数をとって単一の値を返す手続きでなければならない。もしも複数の *list* が与えられたならば，それらはすべて同じ長さでなければならない。map 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。*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。

```
(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))   ⟹  (5 7 9)

(let ((count 0))
  (map (lambda (ignored)
         (set! count (+ count 1))
         count)
       '(a b)))             ⟹  (1 2) または (2 1)
```

(for-each *proc* $list_1$ $list_2$ ...) ライブラリ手続き

for-each の引数は map の引数と同様だが，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。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。

```
(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)
```

(force *promise*) ライブラリ手続き

約束 *promise* の値を引き出させる (4.2.5 節 delay 参照)。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，それが二度目に force されたとき，以前に計算した値を返せるように，キャッシュされる (つまり "memoize" される)。

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

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

(head (tail (tail a-stream)))
                             ⟹  2
```

force と delay は，関数的スタイルで書かれたプログラムで用いられることを主に意図している。下記の例は良いプログラミング・スタイルを例示していると見なされるべきではないが，一つの約束に対しては，たとえそれが何度 force されようとも，たった一つの値だけが計算される，という性質を例示している。

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                            ⟹  約束
(force p)                    ⟹  6
p                            ⟹  約束 (のまま)
(begin (set! x 10)
       (force p))            ⟹  6
```

ここにあるのは delay と force の，一つの可能な実装である。約束はここでは無引数の手続きとして実装され，force は単にその引数を呼び出すだけである:

```
(define force
  (lambda (object)
    (object)))
```

我々は式

```
(delay <expression>)
```

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 (lambda () <expression>))
```

すなわち，次のように定義する:

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (make-promise (lambda () expression))))),
```

ここで make-promise は次のように定義される:

```
(define make-promise
  (lambda (proc)
    (let ((result-ready? #f)
          (result #f))
      (lambda ()
        (if result-ready?
            result
            (let ((x (proc)))
              (if result-ready?
                  result
                  (begin (set! result-ready? #t)
                         (set! result x)
                         result))))))))
```

根拠: 前述の最後の例のように，約束はそれ自身の値を参照するかもしれない。そのような約束を force すると，最初の force の値が計算され終わる前に，二度目の force がひき起こされるかもしれない。このことが make-promise の定義を複雑化している。

delay と force のこの意味論への様々な拡張が，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 force 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
```

- 実装によっては，cdr や + のようなプリミティブ手続きによって約束の値が force されるという ``implicit forcing'' を実装するかもしれない:

```
(+ (delay (* 3 7)) 13)      ⟹  34
```

(call-with-current-continuation *proc*) 手続き

proc は，1 引数の手続きでなければならない。手続き call-with-current-continuation は，現在の継続 (下記の根拠を見よ) を “脱出手続き” (escape procedure) としてパッケージ化し，それを *proc* へ引数として渡す。脱出手続きは一つの Scheme 手続きであり，もしもそれが後から呼び出されるならば，それは，その時たとえどんな継続が有効であっても捨ててしまい，そのかわりに脱出手続きが造られた時に有効だった継続を使うだろう。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* と *after* のサンクの起動をひき起こすかもしれない。

脱出手続きは，call-with-current-continuation への元々の呼出しに対する継続と同じだけ個数の引数を受理する。call-with-values 手続きが造った継続を除き，すべての継続は正確に 1 個の値を受け取る。call-with-values が造っ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てよい。

下記の例は call-with-current-continuation を使う最も平凡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 の平凡な用途は，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極めて有用である。

一つの Scheme 式が評価される時はいつでも，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，もしも式がトップ・レベル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，その答えをトップ・レベル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陽に取り扱う必要があり得る。call-with-current-continuation は，まさしく現在の継続のように振舞う手続きを造ることによって，そうすることを Scheme プログラマにとって可能にしている。

たいていのプログラミング言語は，exit や return，果ては goto のような名前をもつ専用の脱出コンストラクトを 1 個以上採り入れている，しかし，1965 年，Peter Landin [16] は，J 演算子と呼ばれる汎用の脱出演算子を発明した。John Reynolds [24] は，より単純だが同等に強力なコンストラクトを 1972 年に記述した。Sussman と Steele が 1975 年版 Scheme 報告書に記述した catch 特殊形式は，その名前こそ MacLisp にある汎用性の劣るコンストラクトに由来しているが，Reynolds のコンストラクトと正確に同じである。何人かの Scheme 実装者が，catch コンストラクトの完全な強力さを，特殊な構文のコンストラクトによってではなく手続きによって用意できることに気付き，そしてその名前 call-with-current-continuation が 1982 年につくり出された。この名前は説明的だが，このような長い名前の利点については意見の相違があり，一部の人々は call/cc という名前をかわりに使っている。

(values *obj* ...) 手続き

その継続に引数をすべて引き渡す。call-with-values 手続きが造った継続を除き，すべての継続は正確に 1 個の値を受け取る。values 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 引数を無引数で呼び出し，そして継続を呼び出す。この継続は，いくつかの値が渡されたとき，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 を無引数で呼び出し，その呼出しの結果を返す。*before* と *after* も，次の規則が要求するとおりに，無引数で呼び出される (call-with-current-continuation を使ってとらえら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，呼出しが開始される時から呼出しが戻る時までの期間である。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とらえられた継続が，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時，その動的寿命は終わる。
- 動的寿命の期間外にとらえられた継続が，実行がその動的寿命の中にあるときに起動されたならば，その時もまた，その動的寿命は終わる。

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。

とらえられた継続を使って，*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)
```

6.5. Eval

(eval *expression environment-specifier*) 手続き

expression を，指定された環境において評価し，その値を返す。*expression* は，データとして表現された一つの妥当な Scheme 式でなければならず，*environment-specifier* は，以下に記述する三つの手続きのうちの一つによって返された値でなければならない。実装は eval を，第 1 引数として式以外のプログラム (定義) を許したり，環境として他の値を許すように拡張してもよいが，次の制約が伴う: eval には，null-environment または scheme-report-environment と結合した環境において，新しい束縛を造ることは許されていない。

```
(eval '(* 7 3) (scheme-report-environment 5))
                                 ⟹  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                 ⟹  20
```

(scheme-report-environment *version*) 手続き
(null-environment *version*) 手続き

version は，正確整数 5 でなければならない。この数は，Scheme 報告書のこの版 (the Revised^5 Report on Scheme) に対応している。scheme-report-environment は，この報告書で定義されている すべての束縛—要求されているか，あるいは省略可能だが実装によってサポートされている—を除いて 空である環境の 指定子 (specifier) を返す。null-environment は，この報告書で定義されているすべての構文キーワード—要求されているか，あるいは省略可能だが実装によってサポートされている—を除いて空である環境の指定子を返す。

この報告書の過去の版にマッチする環境を指定するために他の値の *version* を使うこともできるが，そのサポートは要求されていない。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，実装はエラーを通知するだろう。

scheme-report-environment において束縛されている変数 (たとえば car) へ，(eval を使って) 代入をすることの効果は未規定である。したがって，scheme-report-environment によって指定される環境は，書換え不可能であってもよい。

(interaction-environment) 省略可能手続き

この手続きは，実装定義の束縛を含む環境—典型的にはこの報告書においてリストされた環境の，あるスーパーセット—の指定子を返す。その意図は，ユーザが動的に打鍵し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。

6.6. 入力と出力

6.6.1. 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，入力ポートとは文字をコマンドへ配送できる Scheme オブジェクトであり，出力ポートとは文字を受理できる Scheme オブジェクトである。

(call-with-input-file *string proc*) ライブラリ手続き
(call-with-output-file *string proc*) ライブラリ手続き

string は，一つのファイルを指名する文字列であること。そして，*proc* は，1 個の引数をとる手続きであること。call-with-input-file では，ファイルが既存であること。一方，call-with-output-file では，ファイルが既存だったときの効果は未規定である。これらの手続きは *proc* を 1 個の引数—指名されたファイルを入力または出力のために開くことによって得られたポート—とともに呼び出す。もしもポートを開くことができないならば，エラーが通知される。*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，再びその中に戻ることができる。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 を，call-with-input-file または call-with-output-file とともに使うポータブルなコードを書くことは，不可能になっただろう。

(input-port? *obj*) 手続き
(output-port? *obj*) 手続き

それぞれ，もし *obj* が入力ポートまたは出力ポートならば #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

現在のデフォルトの入力ポートまたは出力ポートを返す。

(`with-input-from-file` *string thunk*) 省略可能手続き
(`with-output-to-file` *string thunk*) 省略可能手続き

string は，一つのファイルを指名する文字列であること。そして，*thunk* は，引数をとらない手続きであること。`with-input-from-file` では，ファイルが既存であること。一方，`with-output-to-file` では，ファイルが既存だったときの効果は未規定である。入力または出力のためにファイルが開かれ，それに接続された入力ポートまたは出力ポートが，`current-input-port` または `current-output-port` によって返される (かつ (`read`)，(`write` *obj*) などで使われる) デフォルト値へと仕立てられ，そして *thunk* が無引数で呼び出される。*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。`with-input-from-file` と `with-output-to-file` は，*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ならば，これらの手続きの振舞は実装依存である。

(`open-input-file` *filename*) 手続き

既存のファイルを指名する文字列を取り，そして，そのファイルからの文字を配送できる入力ポートを返す。もしもファイルを開くことができないならば，エラーが通知される。

(`open-output-file` *filename*) 手続き

新造される出力ファイルを指名する文字列を取り，そして，その名前を付けられた一つの新しいファイルへと文字を書くことのできる出力ポートを返す。もしもファイルを開くことができないならば，エラーが通知される。もしも与えられた名前をしたファイルが既存ならば，その効果は未規定である。

(`close-input-port` *port*) 手続き
(`close-output-port` *port*) 手続き

port と結合したファイルを閉じて，*port* が文字を配送または受理できないようにする。 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。返される値は未規定である。

6.6.2. 入力

(`read`) ライブラリ手続き
(`read` *port*) ライブラリ手続き

`read` は Scheme オブジェクトの外部表現をオブジェクトそれ自身へと変換する。つまり，`read` は，非終端記号 <データ> に対するパーサである (7.1.2 節と 6.3.2 節を参照)。`read` は，与えられた入力 *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。

もしもオブジェクトの先頭になり得る文字が見つかる前に入力の中で end of file (ファイルの終わり) に出会ったならば，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 read しようと試みるとまた end of file オブジェクトが返されることになる。もしもオブジェクトの外部表現の先頭の後で end of file に出会ったが，その外部表現がまだ完了していないためパース可能でないならば，エラーが通知される。

port 引数は省略可であり，省略時値は `current-input-port` の返す値である。閉じたポートから read することはエラーである。

(`read-char`) 手続き
(`read-char` *port*) 手続き

入力 *port* から次に入手可能な文字を返し，そしてその次に来る文字を指すように *port* を更新する。もしも，もう文字が入手可能でないならば，end of file オブジェクトが返される。*port* は省略可であり，省略時値は `current-input-port` の返す値である。

(`peek-char`) 手続き
(`peek-char` *port*) 手続き

入力 *port* から次に入手可能な文字を返す。ただし，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。もしも，もう文字が入手可能でないならば，end of file オブジェクトが返される。*port* は省略可であり，省略時値は `current-input-port` の返す値である。

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，ということである。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いつでもハングするだろう。

(`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 オブジェクトの精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。

(`char-ready?`) 手続き
(`char-ready?` *port*) 手続き

もし入力 *port* で文字が準備できているならば `#t` を返し，そうでなければ `#f` を返す。もし `char-ready` が `#t` を返すならば，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。もし *port* が end of file にあるならば，そのとき `char-ready?` は `#t` を返す。*port* は省略可であり，省略時値は `current-input-port` の返す値である。

根拠*:* `char-ready?` 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編集により消去され得ないことを，保証しなければならない。もし仮に `char-ready?` が end of file で `#t` を返すとしたならば，end of file 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。

6.6.3. 出力

`(write` *obj*`)` ライブラリ手続き
`(write` *obj port*`)` ライブラリ手続き

obj の表記表現を，与えられた *port* へ書く。この表記表現の中に現れる文字列は二重引用符に囲まれ，そしてその文字列の内部では逆スラッシュと二重引用符は逆スラッシュでエスケープされる。文字オブジェクトは `#\` 記法を使って書かれる。`write` は未規定の値を返す。*port* 引数は省略可であり，省略時値は `current-output-port` の返す値である。

`(display` *obj*`)` ライブラリ手続き
`(display` *obj port*`)` ライブラリ手続き

obj の表現を，与えられた *port* へ書く。この表記表現の中に現れる文字列は二重引用符に囲まれず，そしてその文字列の内部ではどの文字も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。`display` は未規定の値を返す。*port* 引数は省略可であり，省略時値は `current-output-port` の返す値である。

根拠: `write` は機械可読な出力を生成するためにあり，`display` は人間可読な出力を生成するためにある。シンボルの中に "slashification" を許す実装は，シンボルの中にある奇妙な文字を slashify することを，おそらく `write` には望むが `display` には望まないだろう。

`(newline)` ライブラリ手続き
`(newline` *port*`)` ライブラリ手続き

end of line を *port* へ書く。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。*port* 引数は省略可であり，省略時値は `current-output-port` の返す値である。

`(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それ自体) を，与えられた *port* へ書き，未規定の値を返す。*port* 引数は省略可であり，省略時値は `current-output-port` の返す値である。

6.6.4. 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この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*`)` 省略可能手続き

filename は Scheme ソース・コードを内容とする既存のファイルを指名する文字列であること。`load` 手続きはそのファイルから式と定義を読んで逐次的に評価する。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。`load` は未規定の値を返す。

根拠: ポータビリティのため，`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。

`(transcript-on` *filename*`)` 省略可能手続き
`(transcript-off)` 省略可能手続き

filename は造られる出力ファイルを指名する文字列でなければならない。`transcript-on` の効果は，指名されたファイルを出力用に開くこと，そしてユーザと Scheme システムとで交わされるそれ以降の対話の写し (transcript) がそのファイルへ書かれるようにすることである。写しは `transcript-off` を呼び出すことによって終わる。これは写しのファイルを閉じる。いつでもたった一つの写しだけが進行できるが，実装はこの制限を緩和してよい。これらの手続きの返す値は未規定である。

7. 形式的な構文と意味論

この章はこの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

この節は拡張 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区別されない。たとえば，`#x1A` と `#X1a` は等価である。<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

この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。

暗黙の終止を要求するトークン (識別子，数，文字，およびドット) は任意の <区切り文字> によって終止できるが，それ以外のものでは必ずしも終止できるとは限らない。

次に挙げる五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: `[ ] { } |`

<トークン> ⟶ <識別子> | <ブーリアン> | <数>
  | <文字> | <文字列>
  | (|) | #(| ' | ` | , | ,@ | .
<区切り文字> ⟶ <空白> | (|) | " | ;
<空白> ⟶ <スペースまたは改行>
<注釈> ⟶ ; ⟨ 行末までのすべての後続する文字 ⟩
<アトモスフィア> ⟶ <空白> | <注釈>
<トークン間スペース> ⟶ <アトモスフィア>*

<識別子> ⟶ <頭文字> <後続文字>*
  | <独特な識別子>
<頭文字> ⟶ <英字> | <特殊頭文字>
<英字> ⟶ 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
<特殊後続文字> ⟶ + | - | . | @
<独特な識別子> ⟶ + | - | ...
<構文キーワード> ⟶ <式キーワード>
  | else | => | define
  | unquote | unquote-splicing

<式キーワード> ⟶ quote | lambda | if
  | set! | begin | cond | and | or | case
  | let | let* | letrec | do | delay
  | quasiquote

<変数> ⟶ ⟨<構文キーワード>でない任意の<識別子>⟩

<ブーリアン> ⟶ #t | #f
<文字> ⟶ #\ <任意の文字>
  | #\ <文字名>
<文字名> ⟶ space | newline

<文字列> ⟶ " <文字列要素>* "
<文字列要素> ⟶ <" と \を除く任意の文字>
  | \" | \\

<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 進無符号実数>，<R 進無符号整数>，および <R 進接頭辞> に対する下記の規則は $R = 2, 8, 10, 16$ に対してそれぞれ繰り返されるものとする。<2 進小数>，<8 進小数>，および <16 進小数> に対する規則はないが，これは小数点や指数部をもつ数は十進 (decimal radix) でなければならないことを意味する。

<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 進無符号実数> i
  | <R 進実数> - <R 進無符号実数> i
  | <R 進実数> + i | <R 進実数> - i
  | + <R 進無符号実数> i | - <R 進無符号実数> i
  | + i | - i
<R 進実数> ⟶ <符号> <R 進無符号実数>
<R 進無符号実数> ⟶ <R 進無符号整数>
  | <R 進無符号整数> / <R 進無符号整数>
  | <R 進小数>
<10 進小数> ⟶ <10 進無符号整数> <接尾辞>
  | . <10 進数字>$^+$ #* <接尾辞>
  | <10 進数字>$^+$. <10 進数字>* #* <接尾辞>
  | <10 進数字>$^+$ #$^+$. #* <接尾辞>
<R 進無符号整数> ⟶ <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>

<接尾辞> ⟶ <空>
  | <指数部マーカ> <符号> <10 進数字>$^+$
<指数部マーカ> ⟶ e | s | f | d | l
<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 (6.6.2 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>
<単純データ> ⟶ <ブーリアン> | <数>
  | <文字> | <文字列> | <シンボル>
<シンボル> ⟶ <識別子>
<複合データ> ⟶ <リスト> | <ベクタ>
<リスト> ⟶ (<データ>*) | (<データ>$^+$. <データ>)
  | <略記形>
<略記形> ⟶ <略記接頭辞> <データ>
<略記接頭辞> ⟶ ' | ` | , | ,@
<ベクタ> ⟶ #(<データ>*)

7.1.3. 式

<式> ⟶ <変数>
  | <リテラル>
  | <手続き呼出し>
  | <λ式>
  | <条件式>
  | <代入>
  | <派生式>
  | <マクロ使用>
  | <マクロ・ブロック>

<リテラル> ⟶ <引用> | <自己評価的データ>
<自己評価的データ> ⟶ <ブーリアン> | <数>
  | <文字> | <文字列>
<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>

<λ式> ⟶ (lambda <仮引数部> <本体>)
<仮引数部> ⟶ (<変数>*) | <変数>
  | (<変数>$^+$. <変数>)
<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! <変数> <式>)

<派生式> ⟶
  (cond <cond 節>$^+$)
  | (cond <cond 節>* (else <列>))
  | (case <式>
    <case 節>$^+$)
  | (case <式>
    <case 節>*
    (else <列>))
  | (and <テスト>*)
  | (or <テスト>*)
  | (let (<束縛仕様>*) <本体>)
  | (let <変数> (<束縛仕様>*) <本体>)
  | (let* (<束縛仕様>*) <本体>)
  | (letrec (<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>)
  | <準引用>

<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>)
<束縛仕様> ⟶ (<変数> <式>)
<繰返し仕様> ⟶ (<変数> <初期値> <ステップ>)
  | (<変数> <初期値>)
<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>)

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。D は入れ子の深さ (depth) を保持する。

<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> $\longrightarrow$
(<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> $\longrightarrow$
#(<qq テンプレートまたは継ぎ合わせ D>*)
<脱引用 D> $\longrightarrow$,<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> $\longrightarrow$
<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> $\longrightarrow$,@<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> $\longrightarrow$
(syntax-rules (<識別子>*) <構文規則>*)
<構文規則> $\longrightarrow$ (<パターン> <テンプレート>)
<パターン> $\longrightarrow$ <パターン識別子>
| (<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・データ>
<パターン・データ> $\longrightarrow$ <文字列>
| <文字>
| <ブーリアン>
| <数>
<テンプレート> $\longrightarrow$ <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> $\longrightarrow$ <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> $\longrightarrow$ <パターン・データ>
<パターン識別子> $\longrightarrow$ <... を除く任意の識別子>
<省略符号> $\longrightarrow$ <識別子 ...>

7.1.6. プログラムと定義

<プログラム> $\longrightarrow$ <コマンドまたは定義>*
<コマンドまたは定義> $\longrightarrow$ <コマンド>
| <定義>
| <構文定義>
| (begin <コマンドまたは定義>$^+$)
<定義> $\longrightarrow$ (define <変数> <式>)
| (define (<変数> <def 仮引数部>) <本体>)
| (begin <定義>*)
<def 仮引数部> $\longrightarrow$ <変数>*
| <変数>* . <変数>
<構文定義> $\longrightarrow$
(define-syntax <キーワード> <変換子仕様>)

7.2. 形式的意味論

この節は 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。ここで使う概念と記法は [29] に記述されている。記法を下記に要約する:

$\langle\ldots\rangle$	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
$t \rightarrow a, b$	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	ドメイン D の中への x の単射
$x \mid$ D	ドメイン D への x の射影

式の継続が単一の値の代わりに値の列をとる理由は，手続き呼出しと複数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，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。ここでは私たちはそれを，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模倣する。これは必ずしも適切ではない。なぜなら (どんな所与の個数の引数についても) プログラムのいたるところで評価の順序が一定であると，間違って，思わせるからである。しかしこれは，左から右への評価に比べれば，意図した意味論へのよりよい近似である。

記憶領域を割り付ける関数 *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 $new\,\sigma \in$ L ならば $\sigma\,(new\,\sigma \mid \mathrm{L}) \downarrow 2 = \mathit{false}$.

$\mathcal{K}$ 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

$$\mathcal{E}[\![\texttt{((lambda (I*) P')}\ \langle\text{undefined}\rangle\ \ldots\texttt{)}]\!]$$

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，<undefined> は *undefined* 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である。

7.2.1. 抽象構文

$\mathrm{K} \in \mathrm{Con}$	定数 (constants), 引用を含む
$\mathrm{I} \in \mathrm{Ide}$	識別子 (identifiers) つまり変数
$\mathrm{E} \in \mathrm{Exp}$	式 (expressions)
$\Gamma \in \mathrm{Com} = \mathrm{Exp}$	コマンド (commands)

```
Exp  ⟶  K | I | (E0 E*)
        | (lambda (I*) Γ* E0)
        | (lambda (I* . I) Γ* E0)
        | (lambda I Γ* E0)
        | (if E0 E1 E2) | (if E0 E1)
        | (set! I E)
```

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
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	ペア
$\mathtt{E_v}$	$= \mathtt{L}^* \times \mathtt{T}$	ベクタ
$\mathtt{E_s}$	$= \mathtt{L}^* \times \mathtt{T}$	文字列
$\mathtt{M}$	$= \{\mathit{false},\ \mathit{true},\ \mathit{null},\ \mathit{undefined},\ \mathit{unspecified}\}$	雑値 (miscellaneous)
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	手続き値
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	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
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	コマンドの継続
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	式の継続
$\mathtt{A}$		答え (answers)
$\mathtt{X}$		エラー

7.2.3. 意味関数

$$\begin{aligned}
\mathcal{K} &: \mathrm{Con} \to \mathtt{E}\\
\mathcal{E} &: \mathrm{Exp} \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}\\
\mathcal{E}^* &: \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{K} \to \mathtt{C}\\
\mathcal{C} &: \mathrm{Com}^* \to \mathtt{U} \to \mathtt{C} \to \mathtt{C}
\end{aligned}$$

ここで $\mathcal{K}$ の定義は故意に省略する。

$$\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa$$

$$\begin{aligned}
\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,&\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
&\quad(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \to\\
&\qquad\qquad \mathit{wrong}\ \text{“未定義変数”},\\
&\qquad\qquad \mathit{send}\,\epsilon\,\kappa))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(}\mathrm{E_0}\ \ \mathrm{E^*}\mathtt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}^*(\mathit{permute}(\langle\mathrm{E_0}\rangle\ \S\ \mathrm{E^*}))\\
&\qquad\quad \rho\\
&\qquad\quad (\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,\mathit{applicate}\,(\epsilon^*\downarrow 1)\,(\epsilon^*\dagger 1)\,\kappa)\\
&\qquad\qquad (\mathit{unpermute}\,\epsilon^*)))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(lambda\ (}\mathrm{I^*}\mathtt{)}\ \Gamma^*\ \mathrm{E_0}\mathtt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\\
&\qquad \mathit{new}\,\sigma \in \mathtt{L} \to\\
&\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma\,|\,\mathtt{L},\\
&\qquad\qquad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to\\
&\qquad\qquad\quad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\kappa'))\\
&\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\\
&\qquad\qquad\qquad \epsilon^*,\\
&\qquad\qquad\quad \mathit{wrong}\ \text{“引数の個数違い”}\rangle\\
&\qquad\qquad \mathbf{in}\ \mathtt{E})\\
&\qquad\quad \kappa\\
&\qquad\quad (\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathtt{L})\ \mathit{unspecified}\,\sigma),\\
&\qquad\quad \mathit{wrong}\ \text{“メモリ不足”}\ \sigma
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(lambda\ (}\mathrm{I^*}\ \mathtt{.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E_0}\mathtt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\lambda\sigma\,.\\
&\qquad \mathit{new}\,\sigma \in \mathtt{L} \to\\
&\qquad\quad \mathit{send}\,(\langle \mathit{new}\,\sigma\,|\,\mathtt{L},\\
&\qquad\qquad \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to\\
&\qquad\qquad\quad \mathit{tievalsrest}\\
&\qquad\qquad\qquad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\kappa'))\\
&\qquad\qquad\qquad\quad (\mathit{extends}\,\rho\,(\mathrm{I}^*\ \S\ \langle\mathrm{I}\rangle)\,\alpha^*))\\
&\qquad\qquad\qquad \epsilon^*\\
&\qquad\qquad\qquad (\#\mathrm{I}^*),\\
&\qquad\qquad\quad \mathit{wrong}\ \text{“引数が少な過ぎる”}\rangle\ \mathbf{in}\ \mathtt{E})\\
&\qquad\quad \kappa\\
&\qquad\quad (\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathtt{L})\ \mathit{unspecified}\,\sigma),\\
&\qquad\quad \mathit{wrong}\ \text{“メモリ不足”}\ \sigma
\end{aligned}$$

$$\mathcal{E}[\![\mathtt{(lambda}\ \mathrm{I}\ \Gamma^*\ \mathrm{E_0}\mathtt{)}]\!] = \mathcal{E}[\![\mathtt{(lambda\ (.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E_0}\mathtt{)}]\!]$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(if}\ \mathrm{E_0}\ \ \mathrm{E_1}\ \ \mathrm{E_2}\mathtt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E_0}]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E_1}]\!]\rho\kappa,\\
&\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E_2}]\!]\rho\kappa))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(if}\ \mathrm{E_0}\ \ \mathrm{E_1}\mathtt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E_0}]\!]\,\rho\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \to \mathcal{E}[\![\mathrm{E_1}]\!]\rho\kappa,\\
&\qquad\qquad\qquad \mathit{send}\ \mathit{unspecified}\,\kappa))
\end{aligned}$$

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) の代わりに使ってよい。

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(set!}\ \mathrm{I}\ \mathrm{E}\mathtt{)}]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
&\qquad\qquad\qquad \epsilon\\
&\qquad\qquad\qquad (\mathit{send}\ \mathit{unspecified}\,\kappa)))
\end{aligned}$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\kappa\,.\,\kappa\langle\,\rangle$$

$$\begin{aligned}
&\mathcal{E}^*[\![\mathrm{E_0}\ \mathrm{E}^*]\!] =\\
&\quad \lambda\rho\kappa\,.\,\mathcal{E}[\![\mathrm{E_0}]\!]\,\rho\,(\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle\ \S\ \epsilon^*))))
\end{aligned}$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0\ \Gamma^*]\!] = \lambda\rho\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\theta)$$

7.2.4. 補助関数

$$\mathit{lookup} : \mathtt{U} \to \mathrm{Ide} \to \mathtt{L}$$
$$\mathit{lookup} = \lambda\rho\mathrm{I} \,.\, \rho\mathrm{I}$$

$$\mathit{extends} : \mathtt{U} \to \mathrm{Ide}^* \to \mathtt{L}^* \to \mathtt{U}$$
$$\begin{array}{l} \mathit{extends} = \\ \quad \lambda\rho\mathrm{I}^*\alpha^* \,.\, \#\mathrm{I}^* = 0 \to \rho, \\ \qquad\qquad \mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1) \end{array}$$

$\mathit{wrong} : \mathtt{X} \to \mathtt{C}$　　[実装依存]

$$\mathit{send} : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{send} = \lambda\epsilon\kappa \,.\, \kappa\langle\epsilon\rangle$$

$$\mathit{single} : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$$
$$\begin{array}{l} \mathit{single} = \\ \quad \lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1), \\ \qquad\qquad \mathit{wrong}\ \text{“戻り値の個数違い”} \end{array}$$

$\mathit{new} : \mathtt{S} \to (\mathtt{L} + \{\mathit{error}\})$　　[実装依存]

$$\mathit{hold} : \mathtt{L} \to \mathtt{K} \to \mathtt{C}$$
$$\mathit{hold} = \lambda\alpha\kappa\sigma \,.\, \mathit{send}\,(\sigma\alpha \downarrow 1)\kappa\sigma$$

$$\mathit{assign} : \mathtt{L} \to \mathtt{E} \to \mathtt{C} \to \mathtt{C}$$
$$\mathit{assign} = \lambda\alpha\epsilon\theta\sigma \,.\, \theta(\mathit{update}\,\alpha\epsilon\sigma)$$

$$\mathit{update} : \mathtt{L} \to \mathtt{E} \to \mathtt{S} \to \mathtt{S}$$
$$\mathit{update} = \lambda\alpha\epsilon\sigma \,.\, \sigma[\langle\epsilon, \mathit{true}\rangle/\alpha]$$

$$\mathit{tievals} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$$
$$\begin{array}{l} \mathit{tievals} = \\ \quad \lambda\psi\epsilon^*\sigma \,.\, \#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma, \\ \qquad\qquad \mathit{new}\,\sigma \in \mathtt{L} \to \mathit{tievals}\,(\lambda\alpha^* \,.\, \psi(\langle\mathit{new}\,\sigma \mid \mathtt{L}\rangle \mathbin{\S} \alpha^*)) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\epsilon^* \dagger 1) \\ \qquad\qquad\qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma), \\ \qquad\qquad \mathit{wrong}\ \text{“メモリ不足”}\sigma \end{array}$$

$$\mathit{tievalsrest} : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{N} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{tievalsrest} = \\ \quad \lambda\psi\epsilon^*\nu \,.\, \mathit{list}\,(\mathit{dropfirst}\,\epsilon^*\nu) \\ \qquad\qquad (\mathit{single}(\lambda\epsilon \,.\, \mathit{tievals}\,\psi\,((\mathit{takefirst}\,\epsilon^*\nu) \mathbin{\S} \langle\epsilon\rangle))) \end{array}$$

$$\mathit{dropfirst} = \lambda ln \,.\, n = 0 \to l, \mathit{dropfirst}\,(l \dagger 1)(n-1)$$

$$\mathit{takefirst} = \lambda ln \,.\, n = 0 \to \langle\,\rangle, \langle l \downarrow 1\rangle \mathbin{\S} (\mathit{takefirst}\,(l \dagger 1)(n-1))$$

$$\mathit{truish} : \mathtt{E} \to \mathtt{T}$$
$$\mathit{truish} = \lambda\epsilon \,.\, \epsilon = \mathit{false} \to \mathit{false}, \mathit{true}$$

$\mathit{permute} : \mathrm{Exp}^* \to \mathrm{Exp}^*$　　[実装依存]

$\mathit{unpermute} : \mathtt{E}^* \to \mathtt{E}^*$　　[$\mathit{permute}$ の逆関数]

$$\mathit{applicate} : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{applicate} = \\ \quad \lambda\epsilon\epsilon^*\kappa \,.\, \epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa, \mathit{wrong}\ \text{“無効手続き”} \end{array}$$

$$\mathit{onearg} : (\mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$$
$$\begin{array}{l} \mathit{onearg} = \\ \quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa, \\ \qquad\qquad \mathit{wrong}\ \text{“引数の個数違い”} \end{array}$$

$$\mathit{twoarg} : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$$
$$\begin{array}{l} \mathit{twoarg} = \\ \quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa, \\ \qquad\qquad \mathit{wrong}\ \text{“引数の個数違い”} \end{array}$$

$$\mathit{list} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{list} = \\ \quad \lambda\epsilon^*\kappa \,.\, \#\epsilon^* = 0 \to \mathit{send}\,\mathit{null}\,\kappa, \\ \qquad\qquad \mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon \,.\, \mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa)) \end{array}$$

$$\mathit{cons} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{cons} = \\ \quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \to \\ \qquad\qquad (\lambda\sigma' \,.\, \mathit{new}\,\sigma' \in \mathtt{L} \to \\ \qquad\qquad\qquad \mathit{send}\,(\langle\mathit{new}\,\sigma \mid \mathtt{L}, \mathit{new}\,\sigma' \mid \mathtt{L}, \mathit{true}\rangle \\ \qquad\qquad\qquad\qquad \textbf{in}\ \mathtt{E}) \\ \qquad\qquad\qquad\qquad \kappa \\ \qquad\qquad\qquad\qquad (\mathit{update}(\mathit{new}\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'), \\ \qquad\qquad\qquad \mathit{wrong}\ \text{“メモリ不足”}\sigma') \\ \qquad\qquad (\mathit{update}(\mathit{new}\,\sigma \mid \mathtt{L})\epsilon_1\sigma), \\ \qquad\qquad \mathit{wrong}\ \text{“メモリ不足”}\sigma) \end{array}$$

$$\mathit{less} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{less} = \\ \quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\ \qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\ \qquad\qquad \mathit{wrong}\ \text{“< に非数値引数”}) \end{array}$$

$$\mathit{add} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{add} = \\ \quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\ \qquad\qquad \mathit{send}\,((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R})\ \textbf{in}\ \mathtt{E})\kappa, \\ \qquad\qquad \mathit{wrong}\ \text{“+ に非数値引数”}) \end{array}$$

$$\mathit{car} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{car} = \\ \quad \mathit{onearg}\,(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E_p} \to \mathit{hold}\,(\epsilon \mid \mathtt{E_p} \downarrow 1)\kappa, \\ \qquad\qquad \mathit{wrong}\ \text{“}\texttt{car}\text{ に非ペア引数”}) \end{array}$$

$\mathit{cdr} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$　　[car と同様]

$$\mathit{setcar} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{setcar} = \\ \quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{E_p} \to \\ \qquad\qquad (\epsilon_1 \mid \mathtt{E_p} \downarrow 3) \to \mathit{assign}\,(\epsilon_1 \mid \mathtt{E_p} \downarrow 1) \\ \qquad\qquad\qquad\qquad \epsilon_2 \\ \qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa), \\ \qquad\qquad \mathit{wrong}\ \text{“}\texttt{set-car!}\text{ に書換え不可能引数”}, \\ \qquad\qquad \mathit{wrong}\ \text{“}\texttt{set-car!}\text{ に非ペア引数”}) \end{array}$$

$$\mathit{eqv} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$$
$$\begin{array}{l} \mathit{eqv} = \\ \quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to \\ \qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{M} = \epsilon_2 \mid \mathtt{M} \to \mathit{true}, \mathit{false})\kappa, \\ \qquad\quad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to \\ \qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{Q} = \epsilon_2 \mid \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa, \\ \qquad\quad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to \\ \qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{H} = \epsilon_2 \mid \mathtt{H} \to \mathit{true}, \mathit{false})\kappa, \\ \qquad\quad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to \\ \qquad\qquad \mathit{send}\,(\epsilon_1 \mid \mathtt{R} = \epsilon_2 \mid \mathtt{R} \to \mathit{true}, \mathit{false})\kappa, \\ \qquad\quad (\epsilon_1 \in \mathtt{E_p} \wedge \epsilon_2 \in \mathtt{E_p}) \to \\ \qquad\qquad \mathit{send}\,((\lambda p_1 p_2 \,.\, ((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge \\ \qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true}, \\ \qquad\qquad\qquad\qquad \mathit{false}) \\ \qquad\qquad\qquad (\epsilon_1 \mid \mathtt{E_p}) \\ \qquad\qquad\qquad (\epsilon_2 \mid \mathtt{E_p})) \\ \qquad\qquad\quad \kappa, \end{array}$$

$$\begin{array}{l}
(\epsilon_1 \in \mathtt{E_v} \wedge \epsilon_2 \in \mathtt{E_v}) \rightarrow \ldots, \\
(\epsilon_1 \in \mathtt{E_s} \wedge \epsilon_2 \in \mathtt{E_s}) \rightarrow \ldots, \\
(\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \rightarrow \\
\quad \mathit{send}\,((\epsilon_1 \mid \mathtt{F} \downarrow 1) = (\epsilon_2 \mid \mathtt{F} \downarrow 1) \rightarrow \mathit{true}, \mathit{false}) \\
\qquad \kappa, \\
\quad \mathit{send}\ \mathit{false}\ \kappa)
\end{array}$$

$\mathit{apply} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}$

$$\begin{array}{l}
\mathit{apply} = \\
\quad \mathit{twoarg}\,(\lambda \epsilon_1 \epsilon_2 \kappa \,.\, \epsilon_1 \in \mathtt{F} \rightarrow \mathit{valueslist}\,\langle \epsilon_2 \rangle (\lambda \epsilon^* \,.\, \mathit{applicate}\, \epsilon_1 \epsilon^* \kappa), \\
\qquad \mathit{wrong}\ \text{“}\mathtt{apply}\ \text{に無効手続き引数”})
\end{array}$$

$\mathit{valueslist} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}$

$$\begin{array}{l}
\mathit{valueslist} = \\
\quad \mathit{onearg}\,(\lambda \epsilon \kappa \,.\, \epsilon \in \mathtt{E_p} \rightarrow \\
\qquad \mathit{cdr}\langle \epsilon \rangle \\
\qquad\quad (\lambda \epsilon^* \,.\, \mathit{valueslist} \\
\qquad\qquad \epsilon^* \\
\qquad\qquad (\lambda \epsilon^* \,.\, \mathit{car}\langle \epsilon \rangle (\mathit{single}(\lambda \epsilon \,.\, \kappa(\langle \epsilon \rangle \mathbin{\S} \epsilon^*))))), \\
\qquad \epsilon = \mathit{null} \rightarrow \kappa \langle\, \rangle, \\
\qquad \mathit{wrong}\ \text{“}\mathtt{values\text{-}list}\ \text{に非リスト引数”})
\end{array}$$

$\mathit{cwcc} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}$ $\quad$ [`call-with-current-continuation`]

$$\begin{array}{l}
\mathit{cwcc} = \\
\quad \mathit{onearg}\,(\lambda \epsilon \kappa \,.\, \epsilon \in \mathtt{F} \rightarrow \\
\qquad (\lambda \sigma \,.\, \mathit{new}\,\sigma \in \mathtt{L} \rightarrow \\
\qquad\quad \mathit{applicate}\,\epsilon \\
\qquad\qquad \langle \langle \mathit{new}\,\sigma \mid \mathtt{L}, \lambda \epsilon^* \kappa' \,.\, \kappa \epsilon^* \rangle \ \mathtt{in}\ \mathtt{E} \rangle \\
\qquad\qquad \kappa \\
\qquad\qquad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L}) \\
\qquad\qquad\quad \mathit{unspecified} \\
\qquad\qquad\quad \sigma), \\
\qquad\quad \mathit{wrong}\ \text{“メモリ不足”}\ \sigma), \\
\qquad \mathit{wrong}\ \text{“無効手続き引数”})
\end{array}$$

$\mathit{values} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}$

$\mathit{values} = \lambda \epsilon^* \kappa \,.\, \kappa \epsilon^*$

$\mathit{cwv} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C}$ $\quad$ [`call-with-values`]

$$\begin{array}{l}
\mathit{cwv} = \\
\quad \mathit{twoarg}\,(\lambda \epsilon_1 \epsilon_2 \kappa \,.\, \mathit{applicate}\, \epsilon_1 \langle\, \rangle (\lambda \epsilon^* \,.\, \mathit{applicate}\, \epsilon_2\ \epsilon^*))
\end{array}$$

7.3. 派生式型

この節は原始式型 (リテラル，変数，呼出し，`lambda`，`if`，`set!`) による派生式型のマクロ定義を与える。`delay` の一つの可能な定義については 6.4 節を見よ。

```
(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 result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
       ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((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 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ると，その場所に格納された値を得る試みをエラーにしてしまうようなあるものを返す式 (このような式は Scheme では定義されない) の代わりとして，シンボル <undefined> を使っている。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 begin
  (syntax-rules ()
    ((begin exp ...)
     ((lambda () exp ...)))))
```

下記の代替的な begin の展開は，λ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 ...)
     (let ((x exp1))
       (begin exp2 ...)))))
```

下記の do の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)))
```

注

言語変化

この節は “Revised^4 report” [6] が出版されて以来 Scheme になされて来た変更点を列挙する。

- 本報告書は今や Scheme に対する IEEE 規格 [13] のスーパーセットである: 本報告書に合致する実装はその規格にもまた合致することになる。このことは下記の変更を要求した。
  - 空リストは今や真とみなされることが要求される。
  - 機能が本質的 (essential) か非本質的 (inessential) かという分類はなくなった。今や組込み手続きの種類は原始的，ライブラリ，および省略可能の三種類である。省略可能手続きは `load`，`with-input-from-file`，`with-output-to-file`，`transcript-on`，`transcript-off`，および `interaction-environment`，そして三引数以上の `-` と `/` である。これらのどれも IEEE 規格にはない。
  - プログラムが組込み手続きを再定義してもよい。そうすることによって他の組込み手続きの振舞が変わることはない。
- ポートが，分離的な型のリストに加えられた。
- マクロの付録はなくなった。高水準マクロは今や報告書の本体の一部である。派生式に対する書換え規則はマクロ定義で置き換えられた。予約識別子は存在しない。
- `syntax-rules` は今やベクタ・パターンを許す。
- 複数個の値の戻し (multiple-value return)，`eval`，および `dynamic-wind` が加えられた。
- 真正に末尾再帰的な方式で実装されることが要求される呼出しが，明示的に定義された。
- ‘`@`’ を識別子の中に使ってよい。‘`|`’ は将来あり得る拡張のために予約された。

追加資料

The Internet Scheme Repository at

`http://www.cs.indiana.edu/scheme-repository/`

は広範な Scheme 文献目録に加え，論文，プログラム，実装，およびその他の Scheme 関連資料を含んでいる。

例

`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
  (lambda (system-derivative initial-state h)
    (let ((next (runge-kutta-4 system-derivative h)))
      (letrec ((states
                (cons initial-state
                      (delay (map-streams next
                                          states)))))
        states))))
```

`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。`runge-kutta-4` 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
  (lambda (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
  (lambda (f)
    (lambda vectors
      (generate-vector
        (vector-length (car vectors))
        (lambda (i)
          (apply f
                 (map (lambda (v) (vector-ref  v i))
                      vectors)))))))

(define generate-vector
  (lambda (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
  (lambda (s)
    (elementwise (lambda (x) (* x s)))))
```

map-streams 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
  (lambda (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
  (lambda (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
  (lambda (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] 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.

[4] 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.

[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 and Jonathan Rees, editors. The revised^4 report on the algorithmic language Scheme. In *ACM Lisp Pointers* 4(3), pages 1–55, 1991.

[7] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 1991 ACM Conference on Principles of Programming Languages*, pages 155–162.

[8] William Clinger. Proper Tail Recursion and Space Efficiency. To appear in *Proceedings of the 1998 ACM Conference on Programming Language Design and Implementation*, June 1998.

[9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation* 5(4):295–326, 1993.

[10] 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 [11].

[11] D. Friedman, C. Haynes, E. Kohlbecker, and M. Wand. Scheme 84 interim reference manual. Indiana University Computer Science Technical Report 153, January 1985.

[12] *IEEE Standard 754-1985. IEEE Standard for Binary Floating-Point Arithmetic.* IEEE, New York, 1985.

[13] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[14] Eugene E. Kohlbecker Jr. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[15] 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.

[16] Peter Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM* 8(2):89–101, February 1965.

[17] MIT Department of Electrical Engineering and Computer Science. Scheme manual, seventh edition. September 1984.

[18] Peter Naur et al. Revised report on the algorithmic language Algol 60. *Communications of the ACM* 6(1):1–17, January 1963.

[19] 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.

[20] Kent M. Pitman. The revised MacLisp manual (Saturday evening edition). MIT Laboratory for Computer Science Technical Report 295, May 1983.

[21] 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.

[22] Jonathan A. Rees, Norman I. Adams IV, and James R. Meehan. The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[23] 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.

[24] John Reynolds. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740. ACM, 1972.

[25] Guy Lewis Steele Jr. and Gerald Jay Sussman. The revised report on Scheme, a dialect of Lisp. MIT Artificial Intelligence Memo 452, January 1978.

[26] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[27] Guy Lewis Steele Jr. *Common Lisp: The Language, second edition.* Digital Press, Burlington MA, 1990.

[28] Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: an interpreter for extended lambda calculus. MIT Artificial Intelligence Memo 349, December 1975.

[29] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, 1977.

[30] Texas Instruments, Inc. TI Scheme Language Reference Manual. Preliminary version 1.0, November 1985.

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。