Revised^5 Report on the Algorithmic Language Scheme

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

Dedicated to the Memory of Robert Hieb

日本語訳 (revision α1) 大岩 寛 (oiwa@is.s.u-tokyo.ac.jp)

1998/02/20

概要

このレポートはプログラム言語 Scheme の定義と解説を与える。Scheme は、静的なスコープと正確な末尾再帰を持つ Lisp の一種であり、Guy Lewes Steele Jr. と Gerald Jay Sussman によってつくり出された。類を見ないほど明解で単純でセマンティクスを持ち、式の書き方がほとんど一意になるように設計されている。命令型、関数型、Message Passing Style を含む広範囲なプログラムのパラダイムに Scheme は便利であろう。

序論では Scheme 言語とこのレポートについて簡単な歴史を示す。

最初の 3 章は、基本的な言語の考え方を与え、言語の説明とプログラムの記述に用いている記述の凡例を示す。

4 章および 5 章では、式、プログラム、定義についての構文とセマンティクスを示す。

6 章では、Scheme のデータ操作と入出力のプリミティブの全てを含む、組み込み手続きについて説明する。

7 章では拡張 BNF を用いて正式な Scheme の文法を、正式な表示的意味論とともにしめす。

最後には参考文献の一覧とアルファベット順の索引がある。

CONTENTS

序論

プログラム言語は機能を次から次へと積み重ねるのではなく、機能の追加が要求される原因である弱点と制限事項を取り除いていくことで設計されるべきである。Scheme は、今日用いられるプログラムパラダイムのほとんどと共に用いるのに十分な柔軟性をもち、実用的で効率の良いプログラム言語を構成するのには、組み合わせ方に制限のない非常に少ない式構成のルールで十分であることを示している。

Scheme は、ファースト・クラス・プロシージャをλ計算の形で取り込み、それによって動的型付け言語におけるブロック構造と静的スコープルールの有用性を証明した最初のプログラム言語の一つである。Scheme は、全ての変数に単一の構文的環境を用い、また手続き呼び出しのオペレータ部分を引数部分と同じように評価するために、プロシージャをシンボルやλ式と区別した最初の主要な Lisp 方言である。繰り返しの表現を完全に手続き呼び出しに頼ることから、Scheme は末尾再帰手続き呼び出しは意味的に引数を渡す goto であることを強調している。Scheme は広く使われている言語の中で、これまで知られる制御構造の全てを合成可能なファースト・クラス脱出手続きを取り込んだ最初の言語である。その後のバージョンの Scheme は Common Lisp の一般化演算の拡張である正確・非正確な数の概念を導入した。より最近では、Scheme はブロック構造言語の構文を一貫した信頼できる方法で拡張することのできる hygienic macro をサポートする最初のプログラム言語となった。

背景

最初の Scheme の記述は 1975 年 [28] に書かれた。最初の改訂レポート [25] は 1978 年に出され、MIT の実装が革新的なコンパイラをサポートするよう改良された時に、その言語の発展を記述した。3 つの独立なプロジェクトが、Scheme の変種を MIT、イェール、インディアナの各大学の授業で使うために 1981 年から 1982 年に始まった [21, 17, 10]。Scheme を使った入門的なコンピュータ科学の教科書が 1984 年に出版された [1]。

Scheme が広まっていくにつれ、ローカルな方言が広がり始め、時には学生や研究者が他のサイトで書かれたコードを理解するのを困難と感じるようになった。そのため、主要な Scheme の実装の代表 15 人が 1984 年 10 月に集まり、より良くより広く受け入れられる Scheme の実装を目指すことになった。彼らのレポート [4] は MIT とインディアナ大学で 1985 年夏に発表された。さらなる改良が 1986 年春 [23] と 1988 年春 [6] に行なわれた。この現在のレポートは 1992 年 6 月に Xerox PARC で行なわれた会合で合意されたさらなる改良に基づいている。

謝辞

以下の方々の援助に感謝する: 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, and Ozan Yigit. Carol Fessenden と Daniel Friedman と Christopher Haynes に、“Scheme 311 version 4 reference manual” からの引用許可に感謝する。Texas Instruments, Inc. に、*TI Scheme Language Reference Manual*[30] からの引用許可に感謝する。MIT Scheme[17], T[22], Scheme 84[11],Common Lisp[27], そして Algol 60[18] の各マニュアルに影響を受けたことに感謝する。

また、Betty Dexter のこのマニュアルを TEX で組版した多大な努力に、そして彼女の苦労の種となったプログラムをデザインした Donald Knuth に感謝する。

マサチューセッツ工科大学人工知能研究室、インディアナ大学のコンピュータ科学科、オレゴン大学のコンピュータ情報科学科、NEC 研究所がこのレポートの準備を支援した。MIT の作業の一部の支援は、Office of Naval Research contract N00014-80-C-0505 の元で、 Advanced Research Projects Agency of the Department of Defense によってなされた。インディアナ大学の作業の支援は NSF grants NCS 83-04567 and NCS 83-03325 によってなされた。

言語の解説

1. Scheme の概略

1.1. 意味論

この節では Scheme の意味論の概略を与える。非正式な意味論の概略は、3 章から 6 章で与える。参照の目的には、7.2 章で正式な Scheme の意味論を示す。

Algol 同様、Scheme は静的スコープを持つプログラム言語である。変数の使用はそれぞれがその変数の構文的に明らかな束縛に関連付けられる。

Scheme は型を決定することに関しては latent である。型は変数にではなく値（オブジェクトとも呼ばれる）に関連付けられる。（latent な型を持つ言語を弱く型付けされた、あるいは動的に型付けされた言語と呼ぶ著者もいる。）Latent な型を持つ言語には他に APL, Snobol, 他の Lisp 方言がある。決定された型を持つ言語（強く型付けされた、あるいは静的に型付けされた言語とも呼ばれる）は Algol 60、Pascal、C 言語などが含まれる。

関数や継続（continuation）を含む、Scheme の計算過程で作られたオブジェクトは無限の寿命を持つ。Scheme の実装が（大抵！）記憶を使い果たさないのは、あるオブジェクトがその後の計算に影響を及ぼす可能性がない時に、そのオブジェクトの占める記憶領域を回収することが許されているからである。オブジェクトが無限の寿命を持つような言語は他に、APL や他の Lisp の方言がある。

Scheme の実装は正確に末尾再帰することが可能であることが求められている。これにより、たとえ繰り返し計算が構文的に再帰手続きとして書かれている場合であっても一定の領域で計算することができる。よって、このような実装では繰り返しは通常の関数呼びだしの手順で表現でき、特別な繰り返しの構文はシンタックスシュガーとしてのみ有用である。3.5 節を参照されたい。

Scheme の関数はれっきとしたオブジェクトである。関数は動的に作ることができ、データ構造に格納でき、関数の帰り値にしたりもできる。このような特徴を持つ言語は他に Common Lisp や ML が含まれる。

Scheme の特徴的な機能の一つは、他の言語では実行の舞台裏でのみ扱われる継続も「ファーストクラス」であることである。継続は大域脱出、バックトラック、コルーチンといったものを含む、いろいろな複雑な制御構造に用いることができる。6.4 節を参照されたい。

Scheme の関数への引数は常に値で渡される。すなわち、実際の引数の式は関数がその値を必要とするかしないかに関わらず、関数が制御を得る前に評価される。ML、C 言語、APL などは同じく常に引数が値渡しされる言語である。これは Haskell の遅延評価方式や Algol 60 の名前渡し方式のように、関数がその値を必要とするまで式を評価しない方式とは異なる。

Scheme の算術演算のモデルは、コンピュータの中での値の表現方法の違いに依存しないように設計されている。Scheme においては、全ての整数は有理数であり、全ての有理数は実数であり、全ての実数は複素数である。よって、他の多くの言語で重要であるような、整数と実数の演算の区別は Scheme には存在しない。その代わりに存在するのは、数学的に一意な数に対応する正確な数の演算と、近似に対応する非正確な数の演算である。Common Lisp 同様、正確な数の演算は整数に限定されない。

1.2. 構文

ほとんどの Lisp の方言同様、Scheme は完全に括弧付けされた前置表記をプログラムと（その他の）データに用いる。したがって、Scheme の文法は、データに用いら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重要な帰着は、Scheme のプログラムとデータが同様に他の Scheme のプログラムから扱える感受性である。例えば、`eval` 関数はデータとして表現された Scheme のプログラムを評価する。

`read` 関数は構文的にも統語的にも読んだデータを分解する。`read` 関数はその入力をプログラムとしてではなくデータ（7.1.2 節）として分解する。

正式な Scheme の文法は 7.1 章で説明する。

1.3. 表記と用語

1.3.1. 基本、ライブラリ、任意の機能

Scheme の実装は任意と記されていない全ての機能をサポートする必要がある。実装は任意の機能を省くことができ、またこのレポートの言語と干渉しない範囲で拡張することができる。特に、実装者は移植可能なコードをサポートするため、このレポートの構文的な規約を邪魔することのない構文モードを提供しなければならない。

Scheme の実装や理解を助けるため、一部の機能はライブラリと記されている。これらは他の基本的な機能を用いて容易に実装することができる。それらは厳密な語法でいえば冗長であるが、良く使われる使用パターンを捉えており、よって便利な短縮語法として用意されている。

1.3.2. エラーと未定義動作

このレポートでエラーの状況について触れるとき、「エラーが報告される」というフレーズは実装がエラーを検出し報告しなければならないことを意味する。そのような単語がエラーの記述中にない場合、実装はエラーを検出し報告することが推奨されるが、必ずしもそれは必要ではない。実装が検出しなくてもよいエラーは大抵単純に「エラーである」とされる。

例えば、関数にその関数が扱うと明示的に記されていない引数を渡すことは、そのような値域エラーはこのレポート中で時折触れられているが、エラーである。実装は関数の値域

の定義に、そのような引数値を含むよう拡張することができる。

このレポートは「実装上の制約の違反を報告できる」というフレーズで、実装により導入された制限によって正しいプログラムの実行を続けられないことを報告することが許されることを示す。実装上の制約はもちろん推奨されないが、実装制約の違反を報告することは推奨される。

例えば、実装はプログラムの実行に十分な記憶容量を持たない時、実装上の制約の違反を報告することができる。

もし、式の値が「未定義である」とされているならば、式はエラーを報告することなく何らかのオブジェクトに評価されなければならないが、その値は実装に依存し、このレポートはその値について明示的に何も触れていないことを意味する。

1.3.3. 項目の書式

4 章と 6 章は項目の集まりになっている。それぞれの項目は言語の機能の一つか、関連する機能の集合かについて触れており、それぞれの機能はプログラムの構文の構造か組み込み関数である。それぞれの項目は 1 行あるいはそれ以上の見出しで始まっており、必須、基本の機能に関しては

テンプレート　　種類

のような、あるいはライブラリ、任意の機能に関しては修飾子をそれぞれ 1.3.1 節に示した「ライブラリ」・「任意」として

テンプレート　　修飾子 種類

のような形式で示す。

種類が “構文” であるとき、項目は式の種類を説明し、テンプレートがその式の構文を示す。式の構成要素は ⟨式⟩、⟨変数⟩ のように角括弧を用いた構文変数によって示される。構文変数はプログラムの一部を示すものである。例えば ⟨式⟩ は構文的に正しい式であるようなあらゆる文字列を示す。

⟨もの$_1$⟩ ...

は 0 個以上の ⟨もの⟩ を、

⟨もの$_1$⟩ ⟨もの$_2$⟩ ...

は 1 つ以上の ⟨もの⟩ を示す。

種類 が “関数” であるとき、項目は関数を説明しており、見出しの行は関数呼び出しのテンプレートを示す。テンプレート中の変数の名前は *italic* になっている。よって次のような見出し

`(vector-ref` *vector k*`)`　　関数

は、組み込み関数 `vector-ref` がベクトル *vector* と正確な非負整数 *k* （下を見よ）をとることを示している。次の見出し

`(make-vector` *k*`)`　　関数
`(make-vector` *k fill*`)`　　関数

は、関数 `make-vector` は 1 つまたは 2 つの引数をとるように定義されていることを示す。

関数に取り扱うとされていない引数を渡すことはエラーである。簡単のために、引数の名前が 3.2 節に示された型の名前でもある時には、その変数はその型でなければならないことを示す。例えば、`vector-ref` の見出しは上の規約によれば `vector-ref` の第 1 引数がベクトルでなければならないことを示す。また、次の表記規約も型制約を暗に示すこととする。

obj	任意のオブジェクト
list, $list_1$, ... $list_j$, ...	リスト (6.3.2 節を見よ)
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, ...	正確な非負整数

1.3.4. 評価例

プログラム例中の記号 “$\Longrightarrow$” は “…は…に評価される” と読まれる。例えば、

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

は式 `(* 5 8)` を評価するとオブジェクト `40` になることを示す。あるいは、もっと細かくいえば、「文字の列 “`(* 5 8)`” で与えられる式が初期環境で評価されると、“`40`” という文字の列で外部表現することのできるオブジェクトに評価される」ということである。3.3 節でオブジェクトの外部表現について論じる。

1.3.5. 名前の慣例

慣例として、常に論理値を返す関数の名前は大抵 “`?`” で終る。それらは述語と呼ばれる。

また、値を既に確保されたロケーション (3.4 節を見よ) に格納する関数の名前は大抵 “`!`” で終る。それらは mutation procedures と呼ばれる。慣例的には、mutation procedure からの返り値は未定義である。

また、ある型のオブジェクトを引数にとり別の型の類似のオブジェクトを返す関数では、名前の途中に “`->`” がある。例えば、`list->vector` はリストをとり、リストの要素と同じ要素を持つベクトルを返す。

2. 構文の規約

この節では非正式な形で Scheme のプログラムを書くのに使われる構文の規約の一部について触れる。正式な Scheme の構文については、7.1 節を見よ。

大文字と小文字は、文字定数と文字列定数の中を除いては一切区別されない。例えば、`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

他のプログラム言語で許されるほとんどの識別子はSchemeでも許される。正確な識別子を構成するルールはSchemeの実装によって異なるが、全ての実装においてアルファベット、数字、そして「拡張アルファベット文字」の列で、数を始めることのできない文字で始まるものは識別子である。加えて、+, -, ... は識別子である。識別子の例を挙げる。

```
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アルファベットと同様に用いることができる。次のものが拡張アルファベット文字である。

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

正式な識別子の構文は 7.1.1 を見よ。

識別子は Scheme プログラム中では次の 2 つの用法がある。

- あらゆる識別子は変数または構文キーワードとして用いることができる。(3.1 節と 4.3 節を見よ).
- 識別子がリテラルとして、あるいはリテラル中で現れた場合（4.1.2 節を見よ）、シンボル（6.3.3 節を見よ）を表すものとして用いられる。

2.2. 空白とコメント

空白文字はスペースと改行である。(実装は大抵タブや改ページといった空白文字を追加している。）空白は可読性の向上と、識別子や数といった分割不可能な構文単位であるトークン同士を分割する必要のために使われるが、それ以外には無意味である。空白は 2 つのトークンの間におくことができるが、トークン内には置くことができない。空白はまた文字列の中にも置くことができ、この場合は意味がある。

セミコロン (;) はコメントの開始を示す。 コメントはセミコロンが出現した行の行末まで続く。コメントは Scheme からは見えないが、改行は空白として見える。これはコメントが識別子や数の途中に現れることを防いでいる。

```
;;; The FACT procedure computes the factorial
;;; of a non-negative integer.
(define fact
  (lambda (n)
    (if (= n 0)
        1         ;Base case: return 1
        (* n (fact (- n 1))))))
```

2.3. その他の表記

数の表記法についての説明は、6.2 節を見よ。

`. + -` これらは数の中で使われ、また、識別子の最初の文字を除くあらゆる場所で用いることができる。単独の + や - も識別子である。(数や識別子の中でない）区切られたピリオドはペアの表記（6.3.2 節）の中で用いられ、また式の仮引数リスト（4.1.4 節）中で残りの引数を示すために用いられる。区切られた 3 つのピリオドの連続 “...” も識別子である。

`( )` 括弧はグルーピングとリストの表記（6.3.2 節）に用いられる。

`'` シングルクオート文字はリテラルデータの表記に用いられる（4.1.2 節)。

`` ` `` バッククオート文字はほとんど定数であるデータの表記に用いられる（4.2.6 節)。

`, ,@` コンマ文字とコンマ–アットマークはバッククオートと共に用いられる（4.2.6 節)。

`"` ダブルクオートは文字列を区切るのに使われる（6.3.5 節)。

`\` バックスラッシュは文字定数（6.3.4 節）の構文で用いられると同時に、文字列定数（6.3.5 節）中でのエスケープ文字として用いられる。

`[ ] { } |` 大括弧と中括弧と縦棒は将来の拡張のために予約されている。

`#` シャープは直後の文字に依存していろいろな目的に用いられる。

`#t #f` これらは論理定数である（6.3.1 節)。

`#\` これは文字定数を導入する（6.3.4 節)。

`#(` これはベクトル定数を導入する（6.3.6 節)。ベクトル定数は) で終了する。

`#e #i #b #o #d #x` これらは数の表記で用いられる（6.2.4 節)。

3. 基本的なコンセプト

3.1. 変数、構文キーワード、領域

識別子は構文の種類か値を格納できる場所を名付けることができる。構文の種類を名付けるような識別子は構文キーワードと呼ばれ、その構文に束縛されているという。ロケーションを名付ける識別子は変数と呼ばれ、そのロケーションに束縛されているという。プログラムのある場所で有効な全ての可視な束縛の集合は、その場所で有効な環境といわれる。ある変数の束縛されているロケーションに格納されている値はその変数の値と呼ばれる。言葉の濫用であるが、変数は時に

は値の名前とか値に束縛されるといわれることがある。これは適切でないが、これにより混乱を招くことは稀である。

ある種の式の種類は新たな構文を作り構文キーワードをその構文に束縛し、またある別の種類の式は新しいロケーションを作り変数をそれに束縛する。これらの式の種類は束縛を作る構文という。構文キーワードを束縛する構文は 4.3 節に列挙されている。最も基本的な変数を束縛する構文は `lambda` 式である。何故なら全ての他の変数を束縛する構文は `lambda` 式を用いて表現することができるからである。他の変数を束縛する構文は `let`, `let*`, `letrec`, `do` 式である。(4.1.4 節, 4.2.2 節, 4.2.4 節を見よ。)

Algol や Pascal のように、そして Common Lisp を除くほとんどの Lisp の方言と異なり、Scheme はブロック構造と静的スコープを持つ言語である。プログラム中の変数が束縛されている場所それぞれに、その束縛が有効であるプログラムの領域を関連付けることができる。領域は束縛を作るそれぞれの束縛構文で判別できる。例えば束縛が `lambda` 式で作られたなら、その領域はその `lambda` 式全体である。全ての識別子の使用は、その識別子の束縛のうちその使用を含む最も内側の領域を形成するものを参照する。識別子の束縛で領域が使用を含むものがない場合、もしトップレベルの環境にその識別子の束縛があるならそれを参照する (6 章及び 4 章)。もしその識別子の束縛がなければ、それは束縛されていないという。

3.2. 型の独立性

どのオブジェクトも次の述語のうち 2 つ以上を真にすることはない。

```
boolean?        pair?
symbol?         number?
char?           string?
vector?         port?
procedure?
```

これらの述語はそれぞれ論理値 (*boolean*)、ペア (*pair*)、シンボル (*symbol*)、数 (*number*)、文字 (*char* あるいは *character*)、文字列 (*string*)、ベクトル (*vector*)、ポート (*port*)、手続き (*procedure*、関数) を定義する。空リストはそれ独自の型を持つ特別なオブジェクトであり、上に挙げた述語のどれも真にしない。

独立した論理型というものが存在するにも関わらず、Scheme のあらゆる値は条件テストにおいては論理値として使うことができる。6.3.1 節に示すとおり、`#f` を除く全ての値は真と判断される。

このレポートにおいて、「真」という言葉は `#f` を除くあらゆる Scheme の値を、「偽」という値は `#f` を示す。

3.3. 外部表現

Scheme (と Lisp) の重要なコンセプトは文字の列としてのオブジェクトの外部表現である。例えば、整数 28 の外部表現は文字の列 "32" であり、2 つの整数 8, 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ただ 1 つである必要はない。整数 28 は "`#e28.000`" や "`#x1c`" といった表現を、また前のリストは "`( 08 13 )`" や "`(8 . (13 . ()))`" (6.3.2 節を見よ) といった表現を持つ。

多くのオブジェクトは標準の外部表現を持つが、手続きのように、標準の表現を持たないものもある (一部の実装はそれらの外部表現を持つかも知れない)。

外部表現は対応するオブジェクトを得るためにプログラム中に書くことができる (4.1.2 節の `quote` を見よ)。

外部表現はまた入出力にも使うことができる。手続き `read` (6.6.2 節) は外部表現を解読し、手続き `write` (6.6.3 節) は外部表現を生成する。両方とも簡潔で強力な入出力機能を提供する。

文字の列 "`(+ 2 6)`" は、整数 8 に評価される式であるにも関わらず、整数 8 の外部表現でないことに注意すべきである。その代わりに、これはシンボル `+` と 2 と 6 の 3 要素からなるリストの外部表現である。Scheme の構文は、式となる文字の列は必ずあるオブジェクトの外部表現になっているという特徴がある。これは、与えられた文字の列がデータとプログラムのどちらを表すか文脈から判読しにくいので混乱の種となるが、同時に、インタプリタやコンパイラのようにプログラムをデータとして (そしてその逆も) 扱うようなプログラムにとっては便利なものとなる。

いろいろな種類のオブジェクトの外部表現は、6 章の各節で、それらのオブジェクトを扱うプリミティブと共に述べる。

3.4. 記憶モデル

変数とペア、ベクトル、文字列といったオブジェクトは暗黙のうちにロケーションあるいはロケーションの列を表現する。例えば文字列は文字列中に存在する文字数だけのロケーションを持つ (ロケーションは必ずしもマシンワード全体に対応する必要はない)。新しい値はそれらのロケーションの一つに手続き `string-set!` を用いて格納できるが、文字列は前と同じロケーションを指し続ける。

変数参照か `car`, `vector-ref`, `string-ref` といった手続きでロケーションから取得したオブジェクトは、そのロケーションに取得以前の最後に格納したオブジェクトと `eqv?` の意味で等価である。

全てのロケーションは現在使用中であるかどうかを記録される。どの変数もどのオブジェクトもそのロケーションを参照していないならばそれは未使用である。このレポートで格納領域が変数かオブジェクトに確保されるというという表現が出てきた時、それは適切な数のロケーションが未使用のロケーションの集合の中から選ばれ、それらのロケーションは実際にオブジェクトや変数がそれらを指し示す前に使用中であると記録されることを意味している。

多くのシステムで定数 (リテラル式の値) は読みだし専用メモリにおかれることが望ましい。これを表現するためには、

全てのロケーションを示すオブジェクトに、オブジェクトが変更可能か変更不可能であるかを示すようなフラグがついていると想像すると便利である。そのようなシステムではリテラル定数と `symbol->string` で返される文字列は変更不可能なオブジェクトであり、その場合でもこのレポートの他の手続きで作られるオブジェクトは変更可能である。変更不可能なオブジェクトで示されるロケーション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. 正確な末尾再帰

Scheme の実装は正確に末尾再帰することが求められている。次に定義する構文の文脈で起こる手続き呼び出しは '末尾呼びだし' である。Scheme の実装において実行中の末尾呼び出しの個数に制限がない時、その実装は正確に末尾再帰している。呼び出しが実行中であるとは呼び出された手続きがまだ値を返す可能性があることである。これは現在の継続が値を返すことや前に `call-with-current-continuation` で捕捉された継続が後から呼び出されて値を返すことを含むことに注意しなければならない。捕捉された継続がなければ、呼び出しは多くても 1 回しか値を返さず、実行中の呼び出しはまだ値を返してない呼び出しに一致する。正確な "正確な末尾再帰" の定義は [8] にある。

Rationale:

直観的には、末尾呼び出しで使われる継続はそれを含む手続きに渡された継続と同じ意味を持つために、実行中の末尾呼び出しには容量を必要としない。不適切な実装はこの呼び出しに新しい継続を使うかも知れないが、この新しい継続へ戻ると、直後にこの呼出元手続きに渡された継続に戻ることになる。正確に末尾再帰する実装はその継続に直接戻る。

正確な末尾再帰は Steele と Sussman のオリジナルバージョンの Scheme の中心的アイディアの 1 つである。彼らの最初の Scheme インタプリタは関数と作用子を実装していた。制御の流れは作用子で表現され、これは値を呼出元に返す代わりに別の作用子に渡すという点で関数と異なっていた。この節の語法に従えば、それぞれの作用子は別の作用子への末尾呼び出しで終っていた。

Steele と Sussman は後に彼らのインタプリタの作用子を扱うコードは関数を扱うコードと同等であることを発見し、そのため両方を言語に含むことは必要なくなった。

末尾呼び出しは末尾文脈で起こる手続き呼び出しである。末尾文脈は導出的に定義されている。末尾文脈は常に特定の λ 式に着目して決定されることに注意されたい。

- 下で ⟨末尾式⟩ として示されている λ 式本体の最後の式は末尾文脈である。

  (`lambda` ⟨formals⟩
    ⟨定義⟩* ⟨式⟩* ⟨末尾式⟩)

- 末尾文脈に以下の式がある時、⟨末尾式⟩ で示される部分式は末尾文脈にある。これらは 7 にある文法から、⟨式⟩ の一部を ⟨末尾式⟩ で置き換えることで得られた。末尾文脈を含む規則のみ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` ⟨末尾式列⟩))

  (`case` ⟨式⟩
    ⟨case 節⟩$^+$)
  (`case` ⟨式⟩
    ⟨case 節⟩*
    (`else` ⟨末尾式列⟩))

  (`and` ⟨式⟩* ⟨末尾式⟩)
  (`or` ⟨式⟩* ⟨末尾式⟩)

  (`let` (⟨束縛記述⟩*) ⟨末尾本体⟩)
  (`let` ⟨変数⟩ (⟨束縛記述⟩*) ⟨末尾本体⟩)
  (`let*` (⟨束縛記述⟩*) ⟨末尾本体⟩)
  (`letrec` (⟨束縛記述⟩*) ⟨末尾本体⟩)

  (`let-syntax` (⟨文法記述⟩*) ⟨末尾本体⟩)
  (`letrec-syntax` (⟨文法記述⟩*) ⟨末尾本体⟩)

  (`begin` ⟨末尾式列⟩)

  (`do` (⟨繰り返し記述⟩*)
      (⟨テスト⟩ ⟨末尾式列⟩)
    ⟨式⟩*)

  ここで

  ⟨cond 節⟩ ⟶ (⟨テスト⟩ ⟨末尾式列⟩)
  ⟨case 節⟩ ⟶ ((⟨データ⟩*) ⟨末尾式列⟩)

  ⟨末尾本体⟩ ⟶ ⟨定義⟩* ⟨末尾式列⟩
  ⟨末尾式列⟩ ⟶ ⟨式⟩* ⟨末尾式⟩

- `cond` 式が末尾文脈にあり、(⟨式$_1$⟩ `=>` ⟨式$_2$⟩) の形の節を持つ時、⟨式$_2$⟩ の評価結果の手続きの暗示された呼び出しは末尾文脈にある。⟨式$_2$⟩ 自体は末尾文脈にはない。

一部の組み込み手続きは末尾呼び出しをするように要求されている。`apply` と `call-with-current-continuation` への第 1 引数、そして `call-with-values` の第 2 引数は末尾呼び出しとして呼ばれなければならない。また、`eval` はその引数を、eval 手続きの中で末尾の位置にあるように評価しなければならない。

次の例で、末尾呼び出しは `f` への呼び出しだけである。`g` や `h` の呼び出しは末尾呼び出しでない。`x` の参照は末尾文脈にあるが、これは呼び出しではないので末尾呼び出しでない。

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

```
    x)
(and (g) (f))))
```

Note: 実装は上 `h` の呼び出しのような、末尾呼び出しのようにして評価可能な末尾でない呼び出しを認識することを許されるが、要求はされない。上の例では、`let` 式は `h` への末尾呼び出しとしてコンパイルすることができる。(`h` が期待しない数の値を返す可能性は無視できる。何故ならそのような場合 `let` の効果は未定義で実装依存である。)

4. 式

式は基本と派生の 2 種類に分類される。基本式は変数と手続き呼び出しを含む。派生式は意味論的には基本的でなく、マクロとして定義することが可能である。マクロ定義が複雑な `quasiquote` は例外として、派生式はライブラリ機能に分類される。適切な定義は 7.3 節にある。

4.1. 基本式

4.1.1. 変数参照

⟨変数⟩ 構文

変数（3.1 節）からなる式は変数参照である。変数参照の値は変数が束縛されているロケーションに格納されている値である。束縛されていない変数を参照するのはエラーである。

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

4.1.2. 定数式

`(quote` ⟨データ⟩`)` 構文
`'`⟨データ⟩ 構文
⟨定数⟩ 構文

`(quote` ⟨データ⟩`)` は ⟨データ⟩ に評価される。⟨データ⟩ はあらゆる Scheme のオブジェクトの外部表現である（3.3 節を見よ）。この表記はリテラル定数を Scheme のコードの中に含めるのに使われる。

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

`(quote` ⟨データ⟩`)` は `'`⟨データ⟩ と短縮できる。この 2 つの表記は全ての面において等価である。

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

数定数、文字列定数、文字定数、論理定数は “自分自身に” 評価される。よってそれらを quote する必要はない。

```
'"abc"                         ⟹  "abc"
"abc"                          ⟹  "abc"
'145932                        ⟹  145932
145932                         ⟹  145932
'#t                            ⟹  #t
#t                             ⟹  #t
```

3.4 節で触れた通り、定数（リテラル式の値）を `set-car!` や `string-set!` といった mutation procedure で変更するのはエラーである。

4.1.3. 手続き呼びだし

(⟨オペレータ⟩ ⟨オペランド$_1$⟩ ...) 構文

手続き呼び出しは単純に呼び出される手続きの式とそれらに渡される引数を括弧で括ることで書かれる。オペレータとオペランドの式は（未定の順序で）評価され、評価結果の手続きに、評価結果の引数が渡される。

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

多数の手続きは変数の値として初期環境で利用可能である。例えば、上の例の加算手続きや乗算手続きは変数 `+` と変数 `*` の値である。新たな手続きは `lambda` 式 によって作ることができる（4.1.4 節を見よ）。

手続き呼び出しは任意の個数の値を返すことができる（6.4 節の `values` を見よ）。`values` を例外とすると、初期環境に存在する手続きは 1 つの値か、手続き `apply` のようにそれらの引数の 1 つである関数の返り値をそのまま返す。

手続き呼び出しは *combination* とも呼ばれる。

Note: 他の Lisp 方言と対照的に、評価の順序は未定義であり、またオペレータ式とオペランド式は常に同じルールの元で評価される。

Note: 評価の順序は他に何も示されていないが、オペレータ・オペランド式の並行する評価の結果はある順次実行の順序に一致している必要がある。その評価順は手続き呼び出しごとに毎回違うものが選ばれてもよい。

Note: 多くの Lisp の方言では、空の combination `()` は正当な式である。Scheme においては combination は少なくとも 1 つの部分式を持つ必要があり、`()` は従って文法的に正しくない式である。

4.1.4. 手続き

`(lambda` ⟨仮引数リスト⟩ ⟨本体⟩`)` 構文

構文*:* ⟨仮引数リスト⟩ は以下に述べる仮引数のリストでなければならず、⟨本体⟩ は 1 つ以上の式の列でなければならない。

意味*:* `lambda` 式は手続きに評価される。`lambda` 式が評価される時に有効な環境は手続きの一部として記憶される。後に手続きが実引数と共に呼び出された時、`lambda` 式が呼び出された時の環境は、引数リスト中の変数を新しいロケーションに束縛することで拡張され、対応する実引数の値はそれらのロケーションに格納され、そして `lambda` 式本体の式は拡張された環境で順に評価される。本体の最後の式の結果が手続き呼び出しの結果として返される。

```
(lambda (x) (+ x x))          ⟹  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
```

⟨仮引数リスト⟩ は次の形式の 1 つでなければならない。

- (⟨変数$_1$⟩ ...): この手続きは一定の数の引数をとる。手続きが呼び出されると、引数は対応する変数の束縛先に格納される。
- ⟨変数⟩: この手続きは任意の数の引数をとる。手続きが呼び出されると、実引数の列は新たに確保されたリストに変換され、そのリストが ⟨変数⟩ の束縛先に格納される。
- (⟨変数$_1$⟩ ... ⟨変数$_n$⟩ . ⟨変数$_{n+1}$⟩): 最後の変数の前にスペースで区切られたピリオドがある場合、手続きはピリオドの前の仮引数の数を n （少なくとも 1 つはなければならない）として、n 個以上の引数をとる。最後の変数の束縛先には、他の仮引数に対応する実引数の後に残った実引数の、新たに確保されたリストが格納される。

⟨仮引数リスト⟩ 中に ⟨変数⟩ が 2 度以上現れるのはエラーである。

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

eqv? や ev? が手続きについても働くよう（6.1 節を見よ）、lambda 式の評価結果の手続きは（概念的に）ロケーションに対応付けされる。

4.1.5. 条件分岐

(if ⟨テスト⟩ ⟨真の式⟩ ⟨偽の式⟩) 構文
(if ⟨テスト⟩ ⟨真の式⟩) 構文

構文: ⟨テスト⟩、⟨真の式⟩、⟨偽の式⟩ は任意の式である。

意味: if 式は次のように評価される。最初に ⟨テスト⟩ が評価される。もしそれが真（6.3.1 節）に評価されるなら、ついで ⟨真の式⟩ が評価されその値が返される。そうでなければ ⟨偽の式⟩ が評価されその値が返される。もし ⟨テスト⟩ が偽に評価され ⟨偽の式⟩ がなければ、式の返り値は未定義である。

```
(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、その値が ⟨変数⟩ の束縛されたロケーションに格納される。⟨変数⟩ は set! 式を含む何らかの領域かトップレベルで束縛されてなければならない。set! 式の返り値は未定義である。

```
(define x 2)
(+ x 1)                       ⟹  3
(set! x 4)                    ⟹  未定義
(+ x 1)                       ⟹  5
```

4.2. 派生式

4.3 節で述べられる通り、このセクションの構文は hygenic である。参照の目的に、7.3 節にはこの節で述べられているほとんどの構文を前の節で述べた基本構文に変換するマクロ定義が書かれている。

4.2.1. 条件分岐

(cond ⟨節$_1$⟩ ⟨節$_2$⟩ ...) ライブラリ構文

構文: それぞれの ⟨節⟩ は、⟨テスト⟩ を任意の式として、

(⟨テスト⟩ ⟨式$_1$⟩ ...)

の形である。あるいは、⟨節⟩ は

(⟨テスト⟩ => ⟨式⟩)

の形式でも良い。最後の ⟨節⟩ は 次の形式の “else 節” であっても良い。

(else ⟨式$_1$⟩ ⟨式$_2$⟩ ...).

意味: cond 式の評価は、連続する ⟨節⟩ の中の ⟨テスト⟩ を、どれか一つが真（6.3.1 節を見よ）に評価されるまで順に評価する。もしある ⟨テスト⟩ が真に評価されたなら、その ⟨節⟩ の残りの ⟨式⟩ は順に評価され、最後の ⟨式⟩ の値が cond 式の返り値となる。もし選ばれた節が ⟨テスト⟩ だけからなり ⟨式⟩ がないなら、⟨テスト⟩ の評価結果が返り値となる。もし選ばれた ⟨節⟩ が => の形式であるなら、まずその中の ⟨式⟩ が評価される。その値は 1 引数をとることのできる手続きでなければならない。その手続きが ⟨テスト⟩ の値を引数として呼び出され、その返り値が cond 式の返り値となる。全ての ⟨テスト⟩ が偽に評価されたとき、else 節がなければ、cond 式の返り値は未定義である。else 節がある時は、その節の ⟨式⟩ が順に評価され、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⟩ は任意の式である。それぞれの ⟨節⟩ は、

((⟨データ$_1$⟩ ...) ⟨式$_1$⟩ ⟨式$_2$⟩ ...)

の形式である。ここで ⟨データ⟩ はオブジェクトの外部表現である。全てのデータはことなる必要がある。最後の節は次の形式の “else 節” であってもよい。

(else ⟨式$_1$⟩ ⟨式$_2$⟩ ...).

意味: case 式は次のように評価される。まず ⟨キー⟩ が評価され、その値がそれぞれの ⟨データ⟩ と比較される。もし ⟨キー⟩ の評価結果がある ⟨データ⟩ と等価（eqv? の意味で。6.1 節を見よ。）であるなら、対応する ⟨節⟩ の ⟨式⟩ が左から右へ順に評価され、最後の ⟨式⟩ の結果が返される。⟨キー⟩ の評価結果がどの ⟨データ⟩ とも一致しないならば、else 節がある時はその中の ⟨式⟩ が順に評価され、最後の ⟨式⟩ の結果が返される。else 節がない時の case 式の結果は未定義である。

```
(case (* 2 3)
  ((2 3 5 7) 'prime)
  ((1 4 6 8 9) 'composite)) ⟹  composite
(case (car '(c d))
  ((a) 'a)
  ((b) 'b))                 ⟹  未定義
(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

⟨テスト⟩ の式が左から右へ評価され、最初に偽に評価された式の値（6.3.1 節を見よ）が返される。残りの式は評価されない。全ての式が真に評価された時は、最後の式の値が返される。式がない時は #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

⟨テスト⟩ の式が左から右へ評価され、最初に真に評価された式の値（6.3.1 節を見よ）が返される。残りの式は評価されない。全ての式が偽に評価された時は、最後の式の値が返される。式がない時は #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. 束縛構文

3 つの束縛構文 let, let*, letrec は Scheme に Algol 60 のようなブロック構造をもたらす。3 つの構文の文法は同じであるが、それぞれの作る変数の束縛の領域が異なっている。let 式では初期値はどの変数も新しく束縛されないうちに計算される。let* 式では束縛と評価は順に実行される。そして letrec 式では、束縛は他の初期値を計算する時点で有効であり、したがって相互再帰の定義をすることができる。

(let ⟨束縛リスト⟩ ⟨本体⟩) ライブラリ構文

構文: ⟨束縛リスト⟩ は次の形式を持つ。

((⟨変数$_1$⟩ ⟨初期化子$_1$⟩) ...)

ここで ⟨初期化子⟩ は式であり、⟨本体⟩ は 1 つ以上の式の列である。束縛される変数のリスト中に ⟨変数⟩ が 2 度以上現れるのはエラーである。

意味: ⟨初期化子⟩ は現在の環境で（ある未定義順序で）評価され、⟨変数⟩ はその結果を持つ新しいロケーションに束縛される。⟨本体⟩ は拡張された環境で評価され、最後の式の結果が返される。それぞれの ⟨変数⟩ の束縛の持つ領域は ⟨body⟩ である。

```
(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 節の名前つき let も見よ。

(let* ⟨束縛リスト⟩ ⟨本体⟩) ライブラリ構文

構文: ⟨束縛リスト⟩ は次の形式

((⟨変数$_1$⟩ ⟨初期化子$_1$⟩) ...)

を持ち、⟨本体⟩ は 1 つ以上の式の列である。

意味: let* は let に似ているが、束縛は左から右に順に行なわれ、(⟨変数⟩ ⟨初期化子⟩) で与えられる束縛の領域はその束縛の直後からの let* 式の部分である。したがって 2 番目の束縛は 1 つ目の束縛の有効な環境で行なわれ、3 番目以降も同様である。

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

(letrec ⟨束縛リスト⟩ ⟨本体⟩) ライブラリ構文

構文: ⟨束縛リスト⟩ は次の形式を持ち。

((⟨変数$_1$⟩ ⟨初期化子$_1$⟩) ...)

⟨本体⟩ は 1 つ以上の式の列である。束縛される変数のリスト中に ⟨変数⟩ が 2 度以上現れるのはエラーである。

意味: ⟨変数⟩ は最初に未定義値を持つ新しいロケーションに束縛され、⟨初期化子⟩ はその束縛後の環境で（未定義の順序で）評価される。それぞれの ⟨変数⟩ に ⟨初期化子⟩ の評価結果が代入され、⟨本体⟩ がその環境で評価される。⟨本体⟩ の最後の式の評価結果が返される。それぞれの ⟨変数⟩ の束縛は letrec 式全体を領域として持つので、相互再帰手続きを定義することができる。

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

次の letrec についての制限事項は重要である。それぞれの ⟨初期化子⟩ の値はどの ⟨変数⟩ の値も参照したりそれに代入することなく評価できなければならない。もしこの制限に違反するなら、それはエラーである。この制限は Scheme が引数を名前でなく値で渡すため必要である。通常の letrec の用法では、⟨初期化子⟩ は全て lambda 式であり、この制限は自動的に満たされる。

4.2.3. 順次実行

(begin ⟨式$_1$⟩ ⟨式$_2$⟩ ...)　　ライブラリ構文

⟨式⟩ は左から右に順に評価され、最後の ⟨式⟩ の評価結果が返される。この構文は入出力のような副作用を順序付けるのに使われる。

```
(define x 0)

(begin (set! x 5)
       (+ x 1))              ⟹  6

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))    ⟹  未定義
           そして  4 plus 1 equals 5  を出力する
```

4.2.4. 繰り返し

(do ((⟨変数$_1$⟩ ⟨初期化子$_1$⟩ ⟨更新値$_1$⟩)　　ライブラリ構文
    ...)
   (⟨テスト⟩ ⟨式⟩ ...)
 ⟨コマンド⟩ ...)

do は繰り返しの構文である。これは束縛される変数の集合と、どのように最初に初期化されるか、そして毎回の繰り返しでどのように更新されるかを指定する。終了条件が満たされた時、⟨式⟩ を評価してループから抜ける。

do 式は次のように評価される。まず ⟨初期化子⟩ が（未定義順序で）評価される。⟨変数⟩ が新しいロケーションに束縛され、⟨初期化子⟩ の評価結果が代入され、繰り返しが始まる。

毎回の繰り返しの最初にまず ⟨テスト⟩ が評価される。もしこの結果が偽である（6.3.1 節を見よ）なら、⟨コマンド⟩ の式が順番に実行され、⟨変分⟩ 式が未定義順序で評価され、⟨変数⟩ が新しいロケーションに束縛されて、⟨更新値⟩ の評価結果が代入されて、次の繰り返しにはいる。

⟨テスト⟩ が真に評価されたときは、⟨式⟩ が左から右に評価され、最後の式の結果が返される。⟨式⟩ が存在しない時は、do 式の返り値は未定義である。

⟨変数⟩ の束縛の領域は、⟨初期化子⟩ を除く do 式全体である。⟨変数⟩ が do の変数に 2 度以上現れるのはエラーである。

⟨更新値⟩ は省略可能で、その場合は (⟨変数⟩⟨初期化子⟩) の代わりに (⟨変数⟩ ⟨初期化子⟩ ⟨変数⟩) と書くのと同等である。

```
(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⟩ ⟨束縛リスト⟩ ⟨本体⟩)　　ライブラリ構文

“名前つき let” は do よりもより一般的なループ構文をもたらし、再帰の表現もできる let の構文の変種である。⟨変数⟩ が ⟨本体⟩ 内で、仮引数が束縛された変数で本体が ⟨本体⟩ であるような手続きに束縛される他は、通常の let と同じ意味・構文を持つ。よって ⟨本体⟩ の実行は ⟨変数⟩ で名付けられた手続きの呼び出しで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 構文は手続き force と共に、*lazy evaluation* や *call by need* を実装するのに用いられる。(delay ⟨式⟩) は将来

のある点で（手続き `force` によって）評価を要求でき、その結果をもたらすような *promise* と呼ばれるオブジェクトを返す。⟨式⟩ が複数の値を返す時の効果は未定義である。

より完全な `delay` の解説は、6.4 節の `force` の解説を見よ。

4.2.6. Quasi クオート

`(quasiquote` ⟨qq テンプレート⟩`)` 構文
`` `⟨qq テンプレート⟩ `` 構文

“バッククオート” あるいは “quasi クオート” 式は、全てではないがほとんどの要素が既知のリストやベクトルの構造を作るのに有用である。コンマが ⟨qq テンプレート⟩ の中に現れない時は、`` `⟨qq テンプレート⟩ `` の評価結果は `'⟨qq テンプレート⟩` の評価結果と同じである。しかし、コンマ が ⟨qq テンプレート⟩ に現れた時は、コンマに続く式は評価（アンクオート）され、その結果はコンマとその式の代わりに元の構造に挿入される。コンマが直後にアットマーク（`@`）を伴って現れた時は、それに続く式はリストに評価されなければならない。リストの開き括弧と閉じ括弧は “引き剥され” リストの要素はコンマ–アットマーク–式の列の代わりに挿入される。コンマ–アットマークはリストやベクトルの ⟨qq テンプレート⟩ の中でのみ許される。

```
`(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)
```

quasi クオートはネストできる。置き換えは最も外側のバッククオートと同じネストレベルのアンクオート要素でのみ起こる。ネストレベルはそれぞれの quasi クオートの内側で 1 つずつ増え、それぞれのアンクオートの中で 1 つずつ減る。

```
`(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⟩`)` という 2 つの表記は全ての面で同一である。また、`,`⟨式⟩ は `(unquote` ⟨式⟩`)` と同等で、`,@`⟨式⟩ は `(unquote-splicing` ⟨式⟩`)` と同等である。リストの car 要素がこれらのシンボルであるリストから `write` で生成される外部表現は実装によって異なる。

```
(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))
```

`quasiquote`, `unquote`, `unquote-splicing` が上で述べた場所以外の ⟨qq テンプレート⟩ 内で現れた場合の動作は予測不可能である。

4.3. マクロ

Scheme のプログラムはマクロと呼ばれる新たな式の種類を定義して使うことができる。プログラムで定義された式の形は

`(`⟨キーワード⟩ ⟨データ⟩ `...)`

という構文を持つ。ここで ⟨キーワード⟩ はユニークに式の種類を決定する識別子である。この識別子をマクロの 構文キーワードあるいは単にキーワードと呼ぶ。⟨データ⟩ の個数とその構文は式の形による。

それぞれのマクロの出現をマクロの使用と呼ぶ。マクロがより基本的な構文にどのように変換されるかを示す規則の集合をマクロの変換子と呼ぶ。

マクロ定義の機能は 2 つの部分からなる。

- 特定の識別子をマクロキーワードとし、それらをマクロの変換子と対応づけ、そのマクロの定義のスコープを制御する式
- マクロ変換子を定義するパターン言語

マクロの構文キーワードは変数の束縛を隠すことができ、またローカル変数の束縛はキーワードの束縛を隠すことができる。パターン言語を使ったマクロの定義は “hygienic” で “参照等価性を持つ” ので、Scheme の構文スコープを保つことができる [14, 15, 2, 7, 9]。

- マクロ変換子が識別子（変数またはキーワード）の束縛を挿入する時は、その識別子は機能的にはそのスコープ中ずっとリネームされ、他の識別子との衝突を防ぐ。トップレベルの `define` は束縛を導入することもあるししないこともあることに注意せよ。5.2 節を見よ。

  マクロ変換子が識別子への自由な参照を導入する時は、その参照はマクロの使用の周囲のどんなローカル束縛にもよらず、変換子が定義された場所での束縛を参照する。

4.3.1. 構文キーワードの束縛構文

`let-syntax` と `letrec-syntax` は `let` と `letrec` と類似であるが、変数を値を持つロケーションに束縛する代わりに、キーワードをマクロ変換子に束縛する。構文キーワードはまたトップレベルにも束縛できる。5.3 節を見よ。

`(let-syntax` ⟨束縛リスト⟩ ⟨本体⟩`)` 構文

構文: ⟨束縛リスト⟩は次の形式を持つ。

`((`⟨キーワード⟩ ⟨変換子表現⟩`) ...)`

それぞれの ⟨キーワード⟩ は識別子であり、⟨変換子表現⟩ は syntax-rules のインスタンスであり、⟨本体⟩ は 1 つ以上の式の列である。⟨キーワード⟩ が束縛されるキーワードのリスト中に 2 回以上出現するのはエラーである。

意味: ⟨本体⟩ は、let-syntax 式の構文環境を、キーワードが ⟨キーワード⟩ であり、それぞれが指定された変換子に束縛されているようなマクロで拡張して得られる構文環境で展開される。それぞれの ⟨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 ⟨束縛リスト⟩ ⟨本体⟩)　　構文

構文: let-syntax と同じである。

意味: ⟨本体⟩ は、letrec-syntax 式の構文環境を、キーワードが ⟨キーワード⟩ であり、それぞれが指定された変換子に束縛されているようなマクロで拡張して得られる構文環境で展開される。それぞれの ⟨キーワード⟩ の束縛は ⟨本体⟩ だけでなく⟨束縛リスト⟩ をもその領域に持つので、変換子は式を、letrec-syntax 式自身によって導入されるマクロの使用に変換することができる。

```
(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

⟨変換子指定⟩ は次の形式を持つ。

(syntax-rules ⟨リテラル集合⟩ ⟨構文規則⟩ ...)

構文: ⟨リテラル集合⟩ は識別子のリストであり、それぞれの ⟨構文規則⟩は次のような形式である。

(⟨パターン⟩ ⟨テンプレート⟩)

⟨構文規則⟩ のなかの ⟨パターン⟩ はマクロのキーワードから始まる ⟨パターン⟩ のリストである。

⟨パターン⟩ は識別子、定数、あるいは次のうちのいずれかである。

(⟨パターン⟩ ...)
(⟨パターン⟩ ⟨パターン⟩ ... . ⟨パターン⟩)
(⟨パターン⟩ ... ⟨パターン⟩ ⟨省略記号⟩)
#(⟨パターン⟩ ...)
#(⟨パターン⟩ ... ⟨パターン⟩ ⟨省略記号⟩)

テンプレートは識別子か定数かあるいは次のうちのいずれかである。

(⟨要素⟩ ...)
(⟨要素⟩ ⟨要素⟩ ... . ⟨テンプレート⟩)
#(⟨要素⟩ ...)

⟨要素⟩ は ⟨テンプレート⟩ あるいはそれに ⟨省略記号⟩ が続いたものであり、⟨省略記号⟩ は識別子 "..." である。("..." はテンプレートと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 hygienic な書き換え規則の列を与えて新しいマクロ変換子を作る。キーワードが syntax-rules によって変換子と関連付けられているマクロの使用は、⟨構文規則⟩ に含まれるパターンと最も左の ⟨構文規則⟩ から順に照合される。マッチが見つかると、マクロの使用は hygenical にテンプレートにしたがって書き換えられる。

⟨構文規則⟩ のパターンに出現する識別子は、パターンの始めのキーワードである場合、⟨リテラル集合⟩ に含まれる場合、"..." である場合を除いてはパターン変数である。パターン変数は特定の入力の要素とマッチし、テンプレート中で入力の要素を参照するのに用いられる。⟨パターン⟩ の中に同じパターン変数が 2 度以上現れるのはエラーである。

⟨構文規則⟩ 中でパターンを開始するキーワードはマッチングでは考慮されず、パターン変数でもリテラル識別子でもない。

Rationale: キーワードのスコープはそれをマクロ変換子に束縛する式または構文定義によって決定される。キーワードがパターン変数かリテラル変数であったならば、そのパターンに続くテンプレートはキーワードが let-syntax と letrec-syntax のどちらで束縛されるかに関わらずそのスコープに入るであろう。

⟨リテラル集合⟩ に現れる識別子は対応する入力の一部分と対応付けされるリテラル識別子として解釈される。入力の部分式がリテラル識別子と対応するのは、それが識別子であり、そのマクロ式中とマクロ定義中での出現が同じ構文的束縛を持つか、同じ識別子で両方とも構文的束縛を持たない場合のみである。

パターンの一部で ... が後に続くものは、0 個以上の入力の要素と対応する。⟨リテラル集合⟩ 中に ... が出現すれば、

それはエラーである。パターンの中では、識別子 ... は部分パターンの空でない列の最後に続かなければならない。

より正式には、入力の形式 F はパターン P と次の場合のみ対応する。

- P がリテラル識別子でない識別子である。
- P がリテラル識別子で F が同じ束縛を持つ識別子である。
- P がリスト $(P_1\ \ldots\ P_n)$ であり F が n 要素のリストで、各要素が P_1 から P_n とそれぞれ対応する。
- P が不完全なリスト $(P_1\ P_2\ \ldots\ P_n\ .\ P_{n+1})$ であり、F が n 要素以上の不完全なリストであって、最初の n 要素が P_1 から P_n とそれぞれ対応し、n 番めの "cdr" が P_{n+1} と対応する
- P が $(P_1\ \ldots\ P_n\ P_{n+1}$ ⟨省略記号⟩$)$ の形式（ただし ⟨省略記号⟩ は識別子 ...）であり、F が少なくとも n 要素ある完全なリストであって、最初の n 要素が P_1 から P_n とそれぞれ対応し、残りの要素がすべて P_{n+1} と対応する。
- P が #$(P_1\ \ldots\ P_n)$ の形をしたベクトルであり、F が n 要素のベクトルで各要素が P_1 から P_n に対応する。
- P が #$(P_1\ \ldots\ P_n\ P_{n+1}$ ⟨省略記号⟩$)$ の形式（ただし ⟨省略記号⟩ は識別子 ...）であり、F が n 要素以上のベクトルであって最初の n 要素が P_1 から P_n とそれぞれ対応し、残りの各要素が P_{n+1} と対応する。
- P がデータであり、F が P と手続き equal? 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内で、どのパターンともマッチしない式で使用するのはエラーである。

マクロの使用が対応する ⟨構文規則⟩ のテンプレートにしたがって変換される時、テンプレート中のパターン変数は入力中の対応した部分で置き換えられる。部分パターン中で 1 つ以上の識別子 ... を後に伴って現れたパターン変数は、同じ数の ... が後に続く部分テンプレートでのみ許される。それらは出力で、入力のマッチする部分全てに、示された通りに分配されて置き換えられる。示された通りに出力を構成することができないならばそれはエラーである。

テンプレートに現れるもののパターン変数でも ... でもない識別子は、出力にリテラル識別子として出力される。識別子が出力に自由変数として出力される時は、それはその識別子の、syntax-rules が現れた場所での束縛先を参照する。リテラル識別子が束縛変数として挿入された時は、それは不用意な自由変数の捕捉を避けるため、改名されたのと同様の効果を持つ。

例えば、let と cond が 7.3 節の様に定義されたならば、それらは（要求される通り）hygenic であり、次のはエラーとならない。

```
(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、したがって式として認識し、マクロ変換子が構文キーワードと認識するトップレベルの識別子 => とは認識しない。よってこの例は

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

と展開され、不正な手続き呼び出しとなる

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

とはならない。

5. プログラムの構造

5.1. プログラム

Scheme のプログラムは式、定義、構文定義の列からなる。式は 4 章で解説した。定義と構文定義がこの章の対象である。

プログラムは典型的にはファイルに格納されるかあるいは対話的に実行中の Scheme にシステムに入力されるかであるが、他の方法も考えられる。ユーザーインタフェースの問題はこのレポートの対象外である。（実際、Scheme は実際の実装無しであっても、計算方法の表現のための記述法としても有用である。）

プログラムのトップレベルでの定義と構文定義の出現は宣言的に解釈できる。それらはトップレベルの環境に束縛を作るか、あるいは既に存在するトップレベルの束縛の値を変更する。トップレベルに現れる式は命令的に解釈される。それらはプログラムが起動された時あるいは読み込まれた時に順次実行され、たいていはある種の初期化を行なう。

プログラムのトップレベルでは (begin ⟨文$_1$⟩ ...) は begin の本体を構成する式、定義、構文定義の列と等価である。

5.2. 定義

定義は式が許される場所の一部で許されるが全ての場所で許されるわけではない。それらは ⟨プログラム⟩ のトップレベルか ⟨本体⟩ の先頭でのみ許される。

定義は次の形式の 1 つをとる。

- (define ⟨変数⟩ ⟨式⟩)
- (define (⟨変数⟩ ⟨仮引数リスト⟩) ⟨本体⟩)

  ⟨仮引数リスト⟩ は 0 個以上の変数の列か、1 つ以上の変数にスペースで区切られたピリオドと 1 つの変数である（lambda 式と同じ）。この形式は

  (define ⟨変数⟩
  (lambda (⟨仮引数リスト⟩) ⟨本体⟩)).

と同等である。

- (define (⟨変数⟩ . ⟨仮引数⟩) ⟨本体⟩)
  ⟨仮引数⟩ は 1 つの変数である。この形式は

  ```
  (define ⟨変数⟩
  (lambda ⟨仮引数⟩ ⟨本体⟩)).
  ```

  と同等である。

5.2.1. トップレベルの定義

プログラムのトップレベルでは、定義

(define ⟨変数⟩ ⟨式⟩)

は、⟨変数⟩ が束縛されているなら効果的には代入文

(set! ⟨変数⟩ ⟨式⟩)

と同じ効果を持つ。しかしながら ⟨変数⟩ が束縛されていない時は、この定義は代入する前に、⟨変数⟩ を新しいロケーションに束縛する。これは束縛されていない変数に set! で代入するとエラーとなるのとはことなる。

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

初期の環境に、ほとんどの変数を未定義値としたまま、全ての使われる可能性のある変数をロケーションに束縛してある Scheme の実装もある。このような実装ではトップレベルの定義は完全に代入と同等である。

5.2.2. 内部定義

定義は ⟨本体⟩（lambda, let, let*, letrec, let-syntax, letrec-syntax および適当な形式の定義の本体）の先頭でも行なうことができる。そのような定義は前に述べたトップレベル定義と対照して内部定義と呼ばれる。内部定義で定義された変数は ⟨本体⟩ にローカルである。すなわち、⟨変数⟩ は代入ではなく束縛され、その束縛の範囲は ⟨本体⟩ 全体である。

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

内部定義を含む ⟨本体⟩ は、常に完全に等価な letrec 式に変換できる。例えば、上の let 式の例は

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

に等価である。

等価な letrec 式と同様に、⟨本体⟩ 中でのそれぞれの内部定義の ⟨式⟩ は、定義される ⟨変数⟩ のどの値も参照せず、またどれにも代入することなく評価できなければならない。

内部定義が行なわれうる場所では (begin ⟨定義$_1$⟩ ...) は begin の本体を構成する定義の列に等価である。

5.3. 構文定義

構文定義は ⟨プログラム⟩ のトップレベルでのみ許される。それらは次の形式を持つ。

(define-syntax ⟨キーワード⟩ ⟨変換子指定⟩)

⟨キーワード⟩ は識別子で、⟨変換子指定⟩ は syntax-rules のインスタンスでなければならない。トップレベルの構文環境は ⟨キーワード⟩ を与えられた変換子に束縛することで拡張される。

define-syntax 類似の内部定義の構文はない。

マクロは文脈的に許される限りは定義や構文定義に展開できるが、しかし構文キーワードのうちで、隠す定義中の文の列中のある文が実際に定義や構文定義であるかであるかを決定するのに必要であったり、グループとその後に続く式の境目を定めるのに必要であるようなものを、隠すような形で定義や構文定義をすることはエラーである。例えば、次のものはどれもエラーである。

```
(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 の組み込み手続きについて述べる。Scheme の初期環境（あるいはトップレベルの環境）には最初に、主としてデータを扱う組み込み手続きなどの有意義な値を持ったロケーションに束縛されたいくつもの変数が存在する。例えば、変数 abs は 1 引数をとり数の絶対値を計算する手続き（が最初に格納されているロケーション）に束縛されており、また変数 + は和を計算する手続きに束縛されている。組み込み手続きのうちで他の組み込み手続きを用いて用意に書くことのできるものは「ライブラリ手続き」としてある。

プログラムはトップレベル宣言を用いてあらゆる変数を束縛できる。さらに、そのような束縛を代入（4.1.6 節を見よ）

を用いて変更できる。これらの操作は Scheme の組み込み手続きの動作に影響しない。定義で導入されていないトップレベル束縛の変更は組み込み手続きの動作に未定義の効果を与える。

6.1. 等価述語

述語は常に論理値（#t と #f）を返す手続きである。等価述語は数学の同値関係（対称的で反射的で推移的なもの）の計算上の類似物である。この節で述べる等価述語の中では、eq? が最も細かく識別性があり、equal? はもっとも粗い。eqv? はわずかに eq? より粗い。

(eqv? *obj*$_1$ *obj*$_2$) 手続き

eqv? はオブジェクトの有用な同値関係を定義する。簡単には、*obj*$_1$ と *obj*$_2$ が通常同じオブジェクト考えられる時に #t を返す。この関係はわずかに解釈の余地があるが、次の部分的な定義は全ての Scheme 実装の eqv? が満たす。

手続き eqv? は次の場合 #t を返す。

- *obj*$_1$ と *obj*$_2$ が共に #t あるいは共に #f である。
- *obj*$_1$ と *obj*$_2$ は共にシンボルで

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

  Note: これは *obj*$_1$ と *obj*$_2$ がどちらも 6.3.3 節で触れた “uninterned symbol” でないことを仮定している。このレポートは実装依存の拡張における eqv? の動作を推測はしない。

- *obj*$_1$ と *obj*$_2$ は共に数であり、数として等しく（6.2 節の = を見よ）、共に正確な数であるかあるいは共に非正確な数である。
- *obj*$_1$ と *obj*$_2$ は共に文字であり、手続き char=? の意味で同じ文字である。
- *obj*$_1$ と *obj*$_2$ はともに空リストである。
- *obj*$_1$ と *obj*$_2$ は記憶中（3.4 節）で同じロケーションを持つペア・ベクトル・文字である。
- *obj*$_1$ と *obj*$_2$ はロケーションタグが等しい（4.1.4 節）手続きである。

eqv? 手続きは次の場合 #f を返す。

- *obj*$_1$ と *obj*$_2$ は違う型である（3.2 節）。
  obj$_1$ と *obj*$_2$ の一方は #t であるが他方は #f である。
- *obj*$_1$ と *obj*$_2$ はシンボルであるが

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

- *obj*$_1$ と *obj*$_2$ の一方は正確な数であり他方は非正確な数である。
- *obj*$_1$ と *obj*$_2$ は手続き = が #f を返す数である。
- *obj*$_1$ と *obj*$_2$ は手続き char=? が #f を返す文字である。
- *obj*$_1$ と *obj*$_2$ は一方が空リストであるが他方は異なる。
- *obj*$_1$ と *obj*$_2$ は異なるロケーションを指し示すペア、ベクトル、文字列である。
- *obj*$_1$ と *obj*$_2$ は手続きであり、ある引数に関して（返り値が異なるかあるいは副作用が異なるという）異なった振舞いをする。

```
(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
```

次の例は上の規則が eqv? の動作を完全には定義しない例である。このような場合、唯一確実なのは eqv? は必ず論理値を返すということである。

```
(eqv? "" "")                    ⟹  未定義
(eqv? '#() '#())                ⟹  未定義
(eqv? (lambda (x) x)
      (lambda (x) x))           ⟹  未定義
(eqv? (lambda (x) x)
      (lambda (y) y))           ⟹  未定義
```

次の例はローカルな状態を持つ手続きに対する eqv? の使用である。gen-counter は内部に固有のカウンタを持つので、毎回区別可能な手続きを返す必要がある。しかしながら gen-loser は内部状態が値にも副作用にも影響しないので、毎回等価な手続きを返す。

```
(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))
                               ⟹  未定義

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                               ⟹  未定義

(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（リテラル式の結果）を変更することはエラーであるので、実装は可能な場合定数間で構造を共有することが許されるが、強制はしない。よって定数に対する eqv? の値は時に実装依存となる。

```
(eqv? '(a) '(a))               ⟹  未定義
(eqv? "a" "a")                 ⟹  未定義
(eqv? '(b) (cdr '(a b)))       ⟹  未定義
(let ((x '(a)))
  (eqv? x x))                  ⟹  #t
```

Rationale: 上の eqv? の定義は手続きやリテラルの扱いに関して実装に自由度を与えている。実装は 2 つのリテラルや手続きが等価であることを検出してもしなくてもよいし、また等価なオブジェクトに同じポインタやビットパターンを用いて表現を統合してもしなくてもよい。

(eq? obj_1 obj_2) 手続き

eq? はいくつかの場合で eqv? よりも細かく区別をできる可能性がある他は eqv? に類似している。

eq? と eqv? はシンボル、論理値、空リスト、ペア、手続き、空でない文字列とベクトルに関しては同じ振舞いをする。eq? の数と文字に対する振舞いは実装依存であるが、必ず真偽値を返し、真を返すのは eqv? も真を返す場合に限られる。eq? はまた空ベクトルや空文字列に対し eqv? と違う動作をしても良い。

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

Rationale: 大抵、eq? は eqv? よりもより効率的に実装することができる。例えば単純なポインタ比較でよく、複雑な動作をしなくても済む。一つの理由は 2 数の比較は eqv? では定数時間でできないかも知れないが、ポインタ比較で実装された eq? は定数時間で終る。eq? は、手続きに関して eqv? と同じ規則に従うので、手続きを状態を持つオブジェクトの実装に使うようなアプリケーションでは eqv? と同様に用いることができる。

(equal? obj_1 obj_2) ライブラリ手続き

equal? は数やシンボルに対して eqv? を適用しながら、ペア、ベクトル、文字列に対し再帰的に比較をする。一つの簡単なルールは同じように出力されるものならば大抵 equal? である。equal? は引数が循環するデータ構造の時は終了しないことがある。

```
(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))        ⟹  未定義
```

6.2. 数

数値計算は Lisp の世界では歴史的に軽視されてきた。Common Lisp が登場するまで、MacLisp システム [20] がわずかに数値計算を効率的に行なう努力をしてきた他は、数値計算を構成する方法についての深い考察はなかった。このレポートは Common Lisp の委員会のすぐれた作業を認識し、彼らの推奨の多くを受け入れる。いくつかの場面ではこのレポートは Scheme の目的に沿うように彼らの推奨を単純化あるいは一般化している。

Scheme の数がモデル化しようとしている数学的な数、Scheme の数を実装するのに用いられる数の計算機表現、そして数を書くのに用いられる表記法の 3 者を区別するのは重要である。このレポートは 数、複素数、実数、有理数、整数といった型を数学的な数と Scheme の数の双方に対して用いる。固定小数点や浮動小数点といった計算機表現は浮動小数点数や浮動小数点数として参照する。

6.2.1. 数の型

数学的には数はあるレベルがその上のレベルの部分集合となるような部分型のタワーに分類することができる。

数
複素数
実数
有理数
整数

例えば 3 は整数である。よって 3 は有理数であり、実数であり、複素数である。これは 3 をモデルする Scheme の数にも当てはまる。Scheme の数においては、これらの型は `number?`, `complex?`, `real?`, `rational?`, `integer?` といった述語で定義される。

数の型と計算機の中での表現の間には単純な関係はない。ほとんどの Scheme の実装者は少なくとも 2 つの異なった表現を 3 に対して提供するが、その異なる表現は同じ整数を指し示す。

Scheme の算術操作は数をできる限り実装表現とは独立に、抽象的なデータとして取り扱う。Scheme の実装は浮動小数点数や固定小数点数、そしてまた違った数の表現を使うかも知れないが、それらはシンプル名なプログラムを書くプログラマには見えてはならない。

しかしながら、正確に表現された数とそうでない数とのあいだの区別は重要である。例えば、データ構造のインデックスや数式処理システムの係数などは正確でなければならないが、一方でたとえば計測結果などはもともと非正確だし、無理数は有理数で近似されうるのでその場合非正確な数での近似となる。非正確な数の使用を正確な数を必要とする場所での使用で検出するため、Scheme は明示的に正確な数を非正確な数から区別する。この区別は型の次元とは直交している。

6.2.2. 正確さ

Scheme の数は正確であるか非正確である。もし数が、正確な定数で書かれるか、あるいは正確な数から正確な演算のみで得られたならば、その数は正確である。数は、非正確な定数で書かれるか、非正確な要素を含む演算から得られたか、非正確な演算から得られたならば非正確である。つまり非正確な性質は数の伝染性の性質である。

もし 2 つの実装が正確な結果を、非正確な中間結果を用いずに得たのであれば、その最終結果は数学的に等価であろう。一方で不正確な数の演算では、浮動小数点演算といった近似法が使われ得るので、必ずしも当てはまらない。しかし数学的に正しい結果にできるだけ正しい結果を得ることは実装の責任である。

`+` のような有理演算は正確な引数を渡された時は常に正確な結果を返すべきである。もし演算が正確な結果をつくり出すことができないのであれば、実装の制約の違反を報告するか、あるいは暗黙のうちに結果を非正確な値に変換することができる。6.2.3 節を見よ。

`inexact->exact` を例外として、この節で述べられる演算は一つでも非正確な数を渡された時には非正確な数で結果を返すべきである。しかし演算は、結果が引数の非正確さに影響されないことを示すことができる時は、正確な結果を返すことができる。例えば、正確な 0 とあらゆる数との乗算では、他方の引数が非正確であっても正確な 0 を返すことができる。

6.2.3. 実装の制約

Scheme の実装は必ずしも 6.2.1 の節に挙げた部分型の集合の全てを実装する必要はない。しかし、Scheme 言語の思想と実装の目的の双方に適合する一貫した部分集合を実装しなければならない。例えば、全ての数は実数であるような実装も十分有用である。

実装は全ての型について、この節の要求を満たす範囲内でサポートする数の範囲を限定することが許される。どんな型の正確な数の範囲も同じ型の非正確な数の範囲と異なって良い。例えば、全ての非正確な実数を浮動小数点数で表現する実装が、非正確な実数の範囲（よって非正確な整数や有理数の範囲）を浮動小数点数の表現範囲に限定した場合であっても、実質的に無限精度の整数や有理数を実装して良い。さらに、そのような実装ではこの範囲の限界に近付いた時に、表現可能な不正確な整数や有理数の間隔は非常に大きくなることが多い。

Scheme の実装は、リスト・ベクトル・文字列のインデックスや、それらの長さの計算から返される値の範囲全体で正確な整数をサポートしなければならない。手続き `length`, `vector-length`, `string-length` は必ず正確な整数を返さなければならず、正確な整数以外をインデックスに使うことはエラーである。さらに、インデックスの範囲の任意の整数定数は、正確な整数定数の形式で表現されている時には、この範囲外でのどんな実装上の制約にもよらず、正確な整数として読み込まれなければならない。最後に、次に挙げた手続きに正確な整数を引数として渡した場合で、数学的な答が実装の正確な整数の範囲で表現可能である時には、必ず正確な数を答として返さなければならない。

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

実装は必ずしも要求はされないが、実質的に無限のサイズと精度を持った正確な整数及び有理数をサポートし、上記の手続きと手続き `/` に正確な引数を渡した時には常に正確な数を返すことが推奨される。これらの手続きが正確な引数を渡されて正確な結果を返すことができない時は、実装の制約の違反を報告するか、あるいは暗黙のうちに結果を非正確な数に変換することができる。そのような変換はのちにエラーを引き起こす可能性もある。

実装は非正確な数の実装に、浮動小数点や他の近似表現を用いて良い。このレポートは IEEE の 32 ビット及び 64 ビットの浮動小数点数の表現を、浮動小数点数表現を用いる実装が用いることを推奨するが強制はしない。他の表現を使う実装はこれらの標準表現を用いた場合と同等かあるいはそれ以上の精度を得るべきである [12]。

特に、浮動小数点数表現を用いる実装は次の規則に従わなければならない: 浮動小数点数の結果は少なくとも引数のどの非正確な数の表現に使われた精度よりも同等以上の精度で表

現されなければならない。sqrt のように非正確な可能性のある演算が正確な数に適用された場合、可能な限り正確な結果を返すことが望ましい（が要求しない）。（例えば正確な 4 の平方根は正確な 2 であるべきである。）しかしながら、正確な数に対する演算が非正確な結果を生み出す場合（例えば sqrt）、結果が浮動小数点数で表現されるならば、存在するもっとも精度の高い浮動小数点数の形式を使わなければならない。但し最高精度の浮動小数点数の形式と同等あるいはそれ以上の精度の他の表現ができる場合はその限りでない。

Scheme は多くの種類の数の表記を許すが、実装はその一部の表記しかサポートしなくても良い。例えば、全ての数が実数であるような実装においては、直交形式及び極形式の複素数の表現はサポートする必要がない。実装が、読み込んだ正確な数定数を正確な数で表現できない時は、実装の制約の違反を報告しても良いし、暗黙のうちに定数を非正確な数で表現しても良い。

6.2.4. 数定数の書式

数の表記表現の書式は正式には 7.1.1 節で触れる。大文字小文字も区別は数定数の中ではなされない。

数は基数接頭語を使うことで 2 進、8 進、10 進、あるいは 16 進数で表記することができる。基数接頭語は #b (2 進)、#o (8 進)、#d (10 進)、#x (16 進) である。基数接頭語がない場合、数は 10 進数で表現されているとみなされる。

数定数は接頭語を使うことで正確あるいは非正確であると指定できる。#e が正確な数の、#i が非正確な数の接頭語である。正確性の接頭語はどの基数接頭語の前後にもつけることができる。もし数の表記表現に正確性接頭語がついていない場合は、定数の正確性は正確あるいは非正確どちらもありえる。表現が小数点を含む場合、あるいは “#” が数字の場所にある場合は非正確であり、そうでなければ正確である。

非正確な数に複数の精度があるシステムでは定数の精度を指示するのは有用である。そのために、数定数は非正確な数表現の希望する精度を示す指数記号を用いて書くことができる。文字 s, f, d, l がそれぞれ *short*, *single*, *double*, *long* の各精度の使用を示す。（4 種類の精度が存在しない場合、4 種の精度指示はそれらの使用可能な精度の上に対応付けられる。例えば、2 種類の内部表現がある実装では例えば short と single を同じものに、long と double を同じものにそれぞれ対応付けることができる。）それに加えて、指数記号 e は実装の標準の精度を指示する。標準の精度は少なくとも *double* と同等の精度を持つが、実装はその標準をユーザが選択することを認めても良い。

```
3.14159265358979F0
        single に丸める — 3.141593
0.6L0
        long に拡張する — .600000000000000
```

6.2.5. 数値演算

読者は以下の数値演算の引数の型制約について、 1.3.3 節の規約を参照されたい。また、この節で用いた例では、正確な数表記で書かれた数定数は実際に正確な数で表現されていると仮定している。また、一部の例は非正確な数表記で書かれた数定数が値の正確さを失うことなく表現できると仮定している。非正確な数の表現に浮動小数点を用いる処理系で、できるだけ結果が一致するように例中の非正確な定数は選ばれている。

(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

これら数の型の述語は数でないものを含むあらゆる引数に適用できる。オブジェクトがそれぞれの名前の型である時に #t を返し、そうでなければ #f を返す。一般的に、ある数に型述語が真を返すならば、それより上位の型の述語は全て真を返す。同様に、ある数に型述語が偽を返すならば、それより下位の型の述語は全て偽を返す。

もし z が非正確な複素数であるなら、(real? z) は (zero? (imag-part z)) が真であるときその時のみ真になる。x が非正確な実数であるなら、(integer? x) は (= x (round x)) が真である時その時のみ真となる。

```
(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
```

Note: 非正確な数に対するこれらの型述語の振舞いは、不正確さが結果に影響を及ぼす可能性があるので信頼できない。

Note: 多くの実装では、手続き rational? は real? と同値であり、また complex? は number? と同じであるが、普通でない実装では一部の無理数を正確に表すことができるかも知れないし、数体系を複素数でない数をサポートするように拡張しているかも知れない。

(exact? z) 手続き
(inexact? z) 手続き

これらの数値述語は値の正確性を調べる方法を与える。どんな Scheme の数についても、この 2 つのどちらか 1 つが真となる。

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

(<= x_1 x_2 x_3 ...) 手続き
(>= x_1 x_2 x_3 ...) 手続き

これらの手続きはそれぞれ引数が等しい、単調増加である、単調減少である、広義単調増加である、広義単調減少であるときに真となる。

これらの手続きは推移的でなければならない。

Note: これらの述語の Lisp 風の言語での伝統的な実装は推移的ではない。

Note: 非正確な数をこれらの述語で比較することはエラーではないが、小さな不正確さが結果に影響するので結果は信用できない。特に = と zero? についてはそれが言える。疑問があれば、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値のある性質について調べ、#t か #f を返す。上の Note を参照されたい。

(max x_1 x_2 ...) ライブラリ手続き
(min x_1 x_2 ...) ライブラリ手続き

これらの手続きは引数の最大および最少を返す。

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

Note: もし引数の 1 つでも非正確であるならば、結果は非正確数となる（手続きがそれらの非正確さが結果に影響しないことを示すことができる場合はその限りでないが、それは特殊な実装でのみ可能である）。min や max が正確・非正確の混ざった数値を比較する場合で、答が非正確数として情報の損失なしに表現できない場合、手続きは実装の制約の違反を報告して良い。

(+ z_1 ...) 手続き
(* z_1 ...) 手続き

これらの手続きは引数の和及び積を返す。

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

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

2 引数以上で呼ばれた場合、これらの手続きは引数の左結合での差および商を返す。1 引数の場合は、引数の加算的あるいは乗算的な逆元を返す。

```
(- 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 は引数の絶対値を返す。

```
(abs -7)                 ⟹  7
```

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

これらの手続きは数値理論的な（整数）除算を実装する。n_2 は 0 であってはならない。この 3 つの手続きは全て整数を返す。もし n_1/n_2 が整数であるなら、

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

である。もし n_1/n_2 が整数でなければ、

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

である。ここでn_q は n_1/n_2 を 0 に向かって丸めたもので、$0 < |n_r| < |n_2|, 0 < |n_m| < |n_2|$ であり、n_r と n_m は n_1 と n_2 の整数倍だけ異なり、n_r は n_1 と同じ符号を、n_m は n_2 と同じ符号をとる。

よってこれより、整数 n_1 と 0 でない整数 n_2 について、全ての関係する数値は正確な数であるとして、

```
(= n1 (+ (* n2 (quotient n1 n2))
         (remainder n1 n2)))
                         ⟹  #t
```

が成り立つ。

```
(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  ; 非正確
```

(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および最小公倍数を返す。返り値は常に非負である。

```
(gcd 32 -36)                ⟹  4
(gcd)                       ⟹  0
(lcm 32 -36)                ⟹  288
(lcm 32.0 -36)              ⟹  288.0   ; 非正確
(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と分母を返す。結果は引数が最も簡単な分数で表現されたかのように計算される。分母は常に正であり、0 の分母は 1 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 を越えない最大の整数を返す。ceiling は x を下回らない最小の整数を返す。truncate は絶対値が x の絶対値を越えない最も x に近い整数を返す。round は x に最も近い整数を返し、ちょうど 2 整数の中間に x が存在する時には偶数に丸める。

Rationale: IEEE 浮動小数点標準の標準の丸めモードと一貫して、round は偶数に丸める。

Note: これらの手続きへの引数が非正確な数であるなら、結果も非正確な数となる。正確な数が必要ならば、結果を手続き inexact->exact に渡す必要がある。

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

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

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

(rationalize x y) ライブラリ手続き

rationalize は x から y 以上離れていない最も単純な有理数を返す。2 つの有理数を $r_1 = p_1/q_1$、$r_2 = p_2/q_2$（既約分数とする）としたとき、$|p_1| \le |p_2|$ かつ $|q_1| \le |q_2|$ であるならば、有理数 r_1 は有理数 r_2 より単純であるという。よって 3/5 は 4/7 より単純である。

この順序づけで全ての有理数が比較可能ではないが、(2/7 と 3/5 を比較せよ）全ての区間にはその区間内の他の有理数と比べても単純であるような有理数が存在する（より単純な 2/5 が 2/7 と 3/5 の間にある)。$0 = 0/1$ は最も単純な有理数である。

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

(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全ての実装に含まれ、これらは常用される超越関数を計算する。log は z の自然対数（常用対数ではない）を、asin, acos, atan はそれぞれ逆正弦 ($\sin^{-1}$)、逆余弦 ($\cos^{-1}$)、逆正接 ($\tan^{-1}$) を計算する。2 引数の atan は、複素数をサポートしない実装でも (angle (make-rectangular x y)) (後述）を計算する。

一般には、数学的な対数・逆三角関数は多価関数である。$\log z$ の値は虚数部の値が $-\pi$ を越え π 以下の範囲にあると定義する。$\log 0$ は定義されない。log をこのように定義すると、逆三角関数の値は次の式で定義できる。

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

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

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

以上の定義は [27] にしたがっており、これは [19] を引用している。枝切り、境界条件、これらの関数の実装についてはこれらの文献を参照されたい。可能な限りこれらの手続きは実数引数からは実数の結果を返す。

(sqrt z) 手続き

z の平方根の主値を返す。結果は正の実数部を持つか、実数部が 0 である場合は非負の虚数部を持つ。

(expt z_1 z_2) 手続き

z_1 を z_2 乗した結果を返す。$z_1 \neq 0$ であるなら

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

である。0^z は $z = 0$ のとき 1 でありそれ以外は 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の複素数をサポートする全ての実装に含まれる。x_1, x_2, x_3, and x_4 を実数とし、z を

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

を満たす複素数とすると、

`(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`)` $\Longrightarrow$ $|x_3|$
`(angle` z`)` $\Longrightarrow$ x_{angle}

である。ただし x_{angle} は$-\pi < x_{angle} \leq \pi$ でかつある整数 n について $x_{angle} = x_4 + 2\pi n$ となる実数である。

Rationale: `Magnitude` は実引数については `abs` と等価であるが、`abs` は全ての実装に必要なのに対し、`Magnitude` は一般の複素数をサポートする実装でのみ必要とされる。

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

`exact->inexact` は z の非正確な数の表現を返す。返り値は数値的に引数にもっとも近い非正確な数である。正確な数の引数が適切に近い非正確な数を持たない場合は、実装の制約の違反を報告して良い。

`inexact->exact` は z の正確な数の表現を返す。返り値は数値的に引数にもっとも近い正確な数である。非正確な数の引数が適切に近い正確な数を持たない場合は、実装の制約の違反を報告して良い。

これらの手続きは実装依存の範囲の整数について、正確な数と非正確な数の間の自明な 1 対 1 の対応を作る。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 は正確な整数 2, 8, 10, 16 のどれかでなければならない。省略された場合は 10 とみなされる。手続き `number->string` は数値 z と基数 *radix* をとり、与えられた数の与えられた基数での外部表現の文字列を、

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

を満たすように返す。この式を満たすような値が存在しない場合はエラーである。

もし z が非正確で、基数が 10 であり、小数点を含む表現で上の式を満たすことができるならば、返り値は小数点を含み上の式を満たすことのできる最小の桁数で（桁合わせ及び末尾の 0 を除いて）表現される。そうでなければ返り値の形式は未定義である。

`number->string` で返される値は明示的な基数接頭語を含まない。

Note: エラーは唯一、 z が複素数でないか、無理数の実部あるいは虚部を持つ複素数であるばあいのみ起こる。

Rationale: z が浮動小数点で表現された非正確な数で、基数が 10 であるなら、上の式は小数点を含む結果で通常満たされる。未定義のケースは無限大、NaN、浮動小数点でない表現などを許す。

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

string で表現されるもっとも精度の高い表現の数値を返す。*radix* は正確な整数 2, 8, 10, 16 でなければならない。指定された時は、*radix* は標準の基数となるが、*string* 中の明示的な基数接頭語が優先する（例えば `"#o177"`）。もし *radix* が指定されなければ、標準の基数は 10 である。*string* が構文的に正しい数値の表記でなければ、`string->integer` は `#f` を返す。

```
(string->number "100")      ⟹  100
(string->number "100" 16)   ⟹  256
(string->number "1e2")      ⟹  100.0
(string->number "15##")     ⟹  1500.0
```

Note: `string->number` の定義域は実装によって次のように制限できる。`string->number` は *string* が明示的な基数接頭語を持っている場合は `#f` を返しても良い。もし実装のサポートする数の範囲が実数であるならば、`string->number` は直交座標および極座標を用いた複素数の表記に対して `#f` を返しても良い。もし全ての数が整数であるならば、`string->number` は分数表記が使われている場合に `#f` を返しても良い。また、全ての数が正確な数であるならば、`string->number` は指数記号か明示的な正確性接頭語が用いられている場合、または `#` が数字の位置に現れている場合に `#f` を返しても良い。もし全ての非正確な数が整数であるならば、`string->number` は小数点が使われている場合に `#f` を返して良い。

6.3. その他のデータ型

この節では数値以外の Scheme のデータ型、すなわち論理型、ペア、リスト、シンボル、文字、文字列、ベクトルについての演算を説明する。

6.3.1. 論理値

標準の論理値のオブジェクトは真偽それぞれ `#t` と `#f` とかかれる。しかし、実際に問題となるのは Scheme の条件式（`if`, `cond`, `and`, `or`, `do`）がオブジェクトを真偽どちらとし

て扱うかである。“真の値” （あるいは単純に “真”) といった場合は条件式で真とみなされるあらゆるオブジェクトを、“偽の値”（あるいは “偽”）といった場合は条件式が偽とみなすあらゆるオブジェクトを指す。

全ての標準の Scheme の値の中で、`#f` だけが条件式で偽と扱われる。`#t`, ペア、空リスト、シンボル、数、文字列、ベクトル、手続きなど、`#f` 以外の全ての値が真と扱われる。

Note: 他の Lisp の方言に慣れたプログラマは、Scheme は空リストと `#f` を、どちらもシンボル `nil` と区別することに注意すべきである。

論理定数は自分自身に評価されるので、プログラム中でクオ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* が偽の時に `#t` を、それ以外は`#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` の時に `#t` を、それ以外は `#f` を返す。

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

6.3.2. ペアとリスト

ペア（ドット対とも呼ばれる）は car および cdr と（歴史的事情により）呼ばれる 2 つのフィールドからなるレコード構造である。ペアは手続き `cons` によって作られる。car, cdr の各フィールドは手続き `car` および `cdr` でアクセスできる。それぞれのフィールドは手続き `set-car!` と `set-cdr!` を用いて代入できる。

ペアは主にリストを表現するために用いられる。リストは再帰的に、空リストか cdr がリストであるペアとして定義される。より厳密には、リストの集合は次の条件を満たす最小の集合 X として定義される。

- 空リストは X に含まれる。
- *list* が X に含まれるならば、cdr フィールドに *list* を持つあらゆるペアも X に含まれる。

リストの連続するペアの car フィールドに含まれるオブジェクトがリストの要素である。例えば、2 要素のリストは car がリストの第 1 要素で、cdr が「car がリストの第 2 要素で cdr が空リストであるペア」であるペアである。リストの長さは要素の数であり、ペアの数に一致する。

空リストはそれ独自の型を持つ特別なオブジェクトである（ペアではない）。空リストは要素を持たず長さは 0 である。

Note: 以上の定義は全てのリストは有限の長さを持ち空リストで終ることを暗に示している。

Scheme のペアのもっとも一般的な表記（外部表現）は「ドット対」表現（c_1 `.` c_2）であり、ここで c_1 は car フィールドの値、c_2 は cdr フィールドの値である。例えば `(4 . 5)` は car が 4 で cdr が 5 であるペアである。`(4 . 5)` はペアの外部表現であり、ペアに評価される式ではないことに注意せよ。

リストはより直線的な表記をすることもできる。リストの要素を単純に括弧で括り空白で区切れば良い。空リストは `()` と書かれる。例えば、

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

と

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

はシンボルのリストの等価な表記である。

空リストで終らないペアの連続は不完全リストと呼ばれる。不完全リストはリストではない。リストとドット対の表記は不完全リストを表現するために併用することができる。

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

と

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

は等価である。

あるペアがリストであるかどうかは cdr に何が格納されているかによる。`set-cdr!` を用いると、ある時にはリストであったオブジェクトが次の時にはリストでなくなっていることもあり得る。

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

リテラル式と手続き `read` で読み込まれるオブジェクトの表現においては、`'`⟨データ⟩, `` ` ``⟨データ⟩, `,`⟨データ⟩, `,@`⟨データ⟩ はそれぞれ、第 1 要素がシンボル `quote`, `quasiquote`, `unquote`, `unquote-splicing` であるような 2 要素のリス

トを意味する。この場合の第 2 要素は ⟨データ⟩ である。この規約はあらゆる Scheme のプログラムがリストとして表現できるようにサポートされている。つまり、Scheme の文法に従えば、全ての ⟨式⟩ は ⟨データ⟩ である（7.1.2 節を見よ）。他の機能と並び、この機能は手続き `read` を用いて Scheme のプログラムを解釈することを可能とする。3.3 節を参照せよ。

`(pair?` *obj*`)` 手続き

`pair?` は *obj* がペアであるなら `#t` を、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

car が obj_1 で cdr が obj_2 であるような新しく確保されたペアを返す。ペアは既存のどのオブジェクトとも（`eqv?` の意味で）異なることが保証される。

```
(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 フィールドの内容を返す。空リストの car をとることはエラーである。

```
(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 フィールドの内容を返す。空リストの cdr をとることはエラーである。

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

`(set-car!` *pair obj*`)` 手続き

obj を *pair* の car フィールドに格納する。`set-car!` の返り値は未定義である。

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

`(set-cdr!` *pair obj*`)` 手続き

obj を *pair* の cdr フィールドに格納する。`set-car!` の返り値は未定義である。

`(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` の合成であり、例えば `caddr` は次のように定義できる。

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

4 段までの全ての合成が定義されている。全部でこれらの手続きは 28 個になる。

`(null?` *obj*`)` ライブラリ手続き

obj が空リストであるときに `#t` を、それ以外は `#f` を返す。

`(list?` *obj*`)` ライブラリ手続き

obj がリストである時に `#t` を、そうでない時に `#f` を返す。定義により、全てのリストは有限の長さを持ち空リストで終る。

```
(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

与えられた引数を要素とする新たに確保されたリストを返す。

```
(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* の要素の後に 残りの *list* の要素を順に繋げたリストを返す。

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

返り値のリストは最後の *list* の構造を共有する他は新しく確保される。最後の引数は実際は何のオブジェクトでも良い。最後の要素がリストでない時は不完全リストが返る。

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

(reverse *list*) ライブラリ手続き

list の要素を逆順に並べた新しく確保されたリストを返す。

```
(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* 要素を取り除いた部分リストを返す。*list* が *k* 未満しか要素を持たない場合はエラーである。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 *k* 番めの要素を返す（これは (list-tail *list k*) の car に等しい)。*list* が *k* 未満しか要素を持たない場合はエラーである。

```
(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* の長さより小さいある整数 *k* を用いて (list-tail *list k*) で得られるような空でないリストのうちで、car フィールドが *obj* に等しい最初のものを返す。*obj* が *list* 中にない時は #f（空リストではない）が返される。memq uses eq? を *obj* と *list* の要素との比較に用い、同様に memv は eqv? を、member は equal?を用いる。

```
(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))   ⟹  未定義
(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（関係リスト association list）はペアのリストでなければならない。これらの手続きは *alist* のなかのペアで、car フィールドが *obj* であるような最初のペアを返す。*alist* の中に *obj* を car として持つペアがなければ、#f（空リストではない）が返される。assq は eq? を *obj* とペアの car との比較に用い、同様に assv は eqv? を、assoc は equal? を用いる。

```
(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)))
                            ⟹  未定義
(assv 5 '((2 3) (5 7) (11 13)))
                            ⟹  (5 7)
```

Rationale: これらの手続きは大抵は述語として用いられるにも関わらず、memq, memv, member, assq, assv, assoc は名前の最後に ? を持たない。これはこれらの手続きが #t や #f だけでなく有用な値を返すからである。

6.3.3. シンボル

シンボルは同じ綴をもつ時その時のみ等しい（eqv? の意味で）という性質から有用なオブジェクトである。これはプログラム中で識別子が持つ必要のある性質と同一であるので、ほとんどの Scheme の実装は内部的にこの目的でシンボルを使っている。シンボルは他のアプリケーションにも有用である。例えば、 Pascal での列挙値のような方法で使うことができる。

シンボルを書く規則は完全に識別子の規則と同一である。2.1 節および 7.1.1 節を見よ。

リテラル式や手続き read での読み込みで得られたシンボルを、write で書き出した時は、再び（eqv? の意味で）同じシンボルとして読み込まれることが保証される。しかしながら、手続き string->symbol は読み書きの不変性が保証されないシンボルを作ることができる。これは作られたシンボルの名前が大文字小文字の通常使われない方のアルファベットや特殊な文字を含むことができるからである。

Note: 一部の実装は読み書きの不変性を全てのシンボルに保証するために、“slashification” と呼ばれる機能をサポートしている。しかしこの機能は歴史的には文字列型がなかったことを補うための目的が主であった。

一部の実装はまた、slashification をサポートする処理系でも読み書きの不変性を壊す “intern されないシンボル” を持つ。これはまた名前が同じシンボルが等しいという規則にも例外を作る。

(symbol? *obj*) 手続き

obj がシンボルである時に #t を、そうでない時に #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 の名前を文字列として返す。シンボルがリテラル式（4.1.2 節）の結果の一部であったとき、あるいは手続き read の結果として得られた時に、名前がアルファベットを含んでいたならば、返される文字列は実装の優先する文字種で返される。大文字を返す実装も小文字を返す実装もある。シンボルが string->symbol の結果として得られたものならば、返される文字列の文字種はもともと string->symbol に渡された文字列と同じである。この手続きに返り値の文字列に string-set! などの mutation procedure を適用するのはエラーである。

次の例は実装の標準文字種が小文字であると仮定している。

```
(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* であるようなシンボルを返す。この手続きは特殊文字や非標準の文字種を含むシンボルを作ることができるが、そのようなシンボルを作ることは、実装によってはそれらをそのまま読むことができないので良い考えでない。symbol->string を参照せよ。

次の例は実装の標準文字種が小文字であると仮定している。

```
(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

文字はアルファベットや数字といった印刷される文字を表現するものである。文字は #\⟨文字⟩ あるいは #\⟨文字名⟩ という表記法で書かれる。

```
#\a         ; 小文字
#\A         ; 大文字
#\(         ; 左括弧
#\          ; スペース
#\space     ; スペースの望ましい書き方
#\newline   ; 改行
```

大文字小文字は #\⟨文字⟩ では区別されるが、#\⟨文字名⟩ では区別されない。#\⟨文字⟩ 中の ⟨文字⟩ がアルファベットであるならば、⟨文字⟩ の直後の文字はスペースか括弧などの区切り文字でなければならない。この規則は、例えば文字列 "#\space" がスペース文字の表現かあるいは文字の表現 "#\s" にシンボルの表現 "pace" が続いたものかというような曖昧さを解決する。

#\ 表記で書かれた文字は自分自身に評価される。すなわちプログラム中でクオートする必要はない。

一部の文字を扱う手続きは大文字と小文字を区別しない。そのような手続きは "-ci" を名前に持つ。

(char? *obj*) 手続き

obj が文字でなければ #t を、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に全順序を導入する。この順序関係では次のことが保証される。

- 大文字は順番にならんでいる。例えば、(char<? #\A #\B) は #t を返す。
- 小文字は順番にならんでいる。例えば、(char<? #\a #\b) は #t を返す。
- 数字は順番にならんでいる。例えば、(char<? #\0 #\9) は #t を返す。
- 全ての数字はあらゆる大文字の前にあるか、あるいは逆である。
- 全ての数字はあらゆる小文字の前にあるか、あるいは逆である。

一部の実装はこれらの手続きを、対応する数述語のように 3 つ以上の引数をとるように拡張している。

(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=? などに類似しているが、大文字と小文字を同一とみなす。例えば、(char-ci=? #\A #\a) は #t を返す。一部の実装はこれらの手続きを、対応する数述語のように 3 つ以上の引数をとるように拡張している。

(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、それぞれ引数がアルファベットである、数字である、空白である、大文字である、小文字である時に #t を、そうでない時に #f を返す。次の ASCII 文字セットにおける注釈は参考のためだけである。アルファベットは 52 文字の大文字・小文字である。数字は 10 個の 10 進数字である。空白はスペース、タブ、改行、改ページ、復帰文字である。

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

char->integer は、文字を受けとりその文字の正確な整数での表現を返す。integer->char は、char->integer でのある文字の対応先となる正確な整数を受けとり、その文字を返す。これらの手続きは、char<=? で順序付けされた文字の集合と、<= で順序付けされた整数の部分集合との間に、順序同型の関係を作る。つまり、

(char<=? a b) $\Longrightarrow$ #t かつ (<= x y) $\Longrightarrow$ #t

であり、x と y が integer->char の定義域内であるなら、次が成り立つ。

```
(<= (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* $char_2$) であるような文字 $char_2$ を返す。さらに、*char* がアルファベットであるなら、char-upcase の返り値は大文字であり、char-downcase の返り値は小文字である。

6.3.5. 文字列

文字列は文字の並びである。文字はダブルクオート（"）で囲われた文字の並びとして書かれる。ダブルクオートは文字列の中では、バックスラッシュ（\）でエスケープすることでのみ次のように書くことができる。

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

バックスラッシュは文字列中ではもう一つのバックスラッシュでエスケープして書くことができる。文字列中の、ダブルクオートやバックスラッシュの続かないバックスラッシュの効果は Scheme は定義しない。

文字列定数はある行から次の行へと続くことができるが、そのような文字列の実際の内容は未定義である。

文字列の長さはそれの含む文字の数である。この数は正確な非負整数で、文字列が作られた時に固定される。文字列の正しいインデックスとは文字列の長さより小さい正確な非負整数である。文字列の最初の文字はインデックス 0 を持ち、2 文字目はインデックス 1 を持ち、以下同様に続く。

“文字列のインデックス *start* からインデックス *end* までの文字” というような文脈では、*start* の文字を含み *end* の文字を含まないという意味である。よって *start* と *end* が同じ時は空な部分文字列を参照し、*start* が 0 で *end* が文字列の長さと等しい時は、文字列の全体が参照される。

文字列を扱う手続きには大文字小文字の差を無視するものがある。それらの手続きは名前に “-ci” を持つ。

(string? *obj*) 手続き

obj が文字列である時 #t を、そうでない時 #f を返す。

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

make-string は長さ k の新しく確保された文字列を返す。*char* が与えられた時は、文字列の全ての要素は *char* に初期化され、そうでない場合の *string* の内容は未定義である。

(string *char* ...) ライブラリ手続き

引数の連接したものを内容とする新しく確保された文字列を返す。

(string-length *string*) 手続き

string の文字数を返す。

(string-ref *string* k) 手続き

k は *string* の正しいインデックスでなければならない。string-ref は *string* の、0 から始まるインデックスで k 番目の文字を返す。

(string-set! *string* k *char*) 手続き

k は *string* の正しいインデックスでなければならない。string-set! は *string* のインデックス k の要素に *char* を格納し未定義値を返す。

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)      ⟹  未定義
(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

2 つの文字列が同じ長さで、同じ位置に同じ文字がある時に #t を、そうでない時には #f を返す。string-ci=? は大文字と小文字を区別しないが、string=? は区別する。

(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の順序の、文字列への辞書順拡張である。例えば、string<? は char<? の順番による辞書順の順序づけである。2 つの文字列の長さが異なり、短い方の長さまでの部分が一致するならば、短い文字列は辞書順で長い文字列より前にあると判断される。

実装は、対応する数述語のように、これらの手続きを 3 引数以上受けとることができるように拡張することができる。

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

string は文字列で、*start* と *end* は

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

を満たす正確な整数でなければならない。substring は、*string* のインデックス *start*（その文字を含む）からインデックス *end*（その文字は含まない）までの部分文字列を新たに確保して返す。

(string-append *string* ...)　ライブラリ手続き

与えられた文字列の文字を繋げて得られる文字列を新たに確保して返す。

(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する文字のリストを新たに確保して返す。list->string は、文字のリスト *list* に含まれる文字からなる文字列を新たに確保して返す。string->list と list->string は equal? の意味で互いに逆関数になっている。

(string-copy *string*)　ライブラリ手続き

新たに確保された *string* のコピーを返す。

(string-fill! *string char*)　ライブラリ手続き

string の全ての要素に *char* を格納し、未定義値を返す。

6.3.6. ベクトル

ベクトルは整数でインデックス付けされた異成分からなる構造体である。ベクトルは普通同じ長さのリストより少ない領域しか必要とせず、ランダムに選ばれた要素をアクセスするのに必要な時間は普通リストよりベクトルの方が短い。

ベクトルの長さはベクトルの含む要素の数である。この数は非負整数で、ベクトルが作られる時に固定される。ベクトルの正しいインデックスはベクトルの長さより小さい正確な非負整数である。ベクトルの最初の要素のインデックスは 0 であり、最後の要素はベクトルの長さから 1 を引いたものである。

ベクトルは #(*obj* ...) のような表記で書かれる。例えば、長さ 3 のベクトルで要素 0 に数 0 を、要素 1 にリスト（2 2 2 2）を、要素 2 に文字列"Anna"を持つものは

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

のように書かれる。これはベクトルの外部表現であって、ベクトルに評価される式ではない。リスト定数と同様、ベクトル定数はクオートされなければならない。

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

(vector? *obj*)　手続き

obj がベクトルである時 #t を、そうでない時 #f を返す。

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

k 要素からなる新しく確保されたベクトルを返す。第 2 引数が渡された時は、それぞれの要素は *fill* で初期化される。そうでない時は各要素の初期値は不定である。

(vector *obj* ...)　ライブラリ手続き

与えられた引数を要素とする新しく確保されたベクトルを返す。list と類似している。

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

(vector-length *vector*)　手続き

ベクトル *vector* の要素数を正確な整数として返す。

(vector-ref *vector k*)　手続き

k はベクトル *vector* の正しいインデックスでなければならない。vector-ref は *vector* の要素 *k* 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 はベクトル *vector* の正しいインデックスでなければならない。vector-set! は *obj* を *vector* の要素 *k* に格納する。vector-set! の返り値は定義されない。

```
(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 ; 定数ベクトル
```

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

vector->list はベクトル *vector* に要素として含まれていたオブジェクトからなる新しく確保されたリストを返す。list->vector はリスト *list* の要素で初期化された新しく確保されたベクトルを返す。

```
(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

ベクトル *vector* の全要素に *fill* を格納する。vector-fill! の返り値は未定義である。

6.4. 制御機能

この章では、プログラムの実行の流れを特別な方法で制御する組み込み手続きについて述べる。述語 procedure? についてもここで述べる。

(procedure? *obj*) 手続き

obj が手続きである時 #t を、そうでない時 #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、*args* はリストでなければならない。リスト (append (list arg_1 ...) *args*) の要素を実際の引数として、 *proc* を呼び出す。

```
(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

list は全てリストであり、*proc* はそのリストの数と同じ数の引数をとることができ、1 つの値を返す手続きでなければならない。2 つ以上のリストが渡された時には、それらはすべて同じ長さでなければならない。map は *list* の要素に 1 要素ずつ *proc* を適用し、結果のリストを返す。*list* の要素に *proc* を適用する順序は未定義である。

```
(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) か (2 1)
```

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

for-each の引数は map と同様であるが、for-each は *proc* を、値ではなく副作用を目的として呼び出す。map と異なり、for-each は *proc* を *list* の各要素に先頭要素から最後の要素へと順番に適用することが保証される。for-each の返り値は未定義である。

```
(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

引数 *promise* の値を force する（4.2.5 節の delay を見よ）。 promise に対して値が計算されていない場合は値を計算してそれを返す。promise の値はキャッシュされ（記憶され）それ以降に再び force された場合は前に計算された値が返される。

```
(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` は主に関数型スタイルで書かれたプログラムを意図している。次の例は良いプログラミングスタイルを示していると考えてはならないが、何回 force されても 1 つの値しか計算されないという性質を表している。

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                           ⟹  a promise
(force p)                   ⟹  6
p                           ⟹  相変わらず promise
(begin (set! x 10)
       (force p))           ⟹  6
```

ここには `delay` と `force` の実装の例を示す。promise はここでは引数を持たない手続きとして実装され、`force` は単純にその引数を呼び出している。

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

式

```
(delay ⟨式⟩)
```

は次の手続き呼びだし

```
(make-promise (lambda () ⟨式⟩))
```

と同じ意味を持つように、次のように実装される。

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

ここで `make-promise` は次のように定義する。

```
(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))))))))
```

Rationale: promise は、先の例の一番最後のように、自分自身の値を参照することができる。そのような promise を force すると、promise の 1 回目の値が計算されるより先に 2 度目の force が起こる。このことが `make-promise` の実装を複雑にしている。

`delay` と `force` の方法に対する次のような拡張が一部の実装でサポートされている。

- `force` を promise でないオブジェクトを引数にして呼び出すと単純にそのオブジェクトを返す。
- promise とその force された値を操作的に区別する必要がない場合がある。つまり、次のような式は実装に依存して `#t` と `#f` のどちらを返す場合もありえる。

  ```
  (eqv? (delay 1) 1)          ⟹  未定義
  (pair? (delay (cons 1 2)))  ⟹  未定義
  ```

- `cdr` や `+` といった組み込み手続きが引数の promise を force する “暗黙の force” を持つ実装もある。

  ```
  (+ (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。`call-with-current-continuation` は現在の継続（後述の rationale を見よ）を “脱出手続き” としてパッケージして、*proc* に引数として渡す。脱出手続きは、後に呼び出された時に、その時点での有効な継続を全て破棄して、脱出手続きが作られた時の継続をその代わりに使うという効果を持つ Scheme の手続きである。脱出手続きの呼び出しは `dynamic-wind` を用いて導入された thunk *before* および *after* の実行を引き起こすことがある。

脱出手続きは `call-with-current-continuation` のもともとの呼び出しの継続と同じ数の引数を受け付ける。`call-with-values` によって作られた継続を除いては、全ての継続はちょうど 1 つの値をとる。`call-with-values` によって作られたのでない継続に 0 あるいは 2 以上の値を渡した時の効果は未定義である。

proc に渡された脱出手続きは Scheme の他の手続きと同様、無限の寿命を持つ。それを変数やデータ構造に格納することもでき、また必要なだけ何度でも呼び出すことができる。

次の例は call-with-current-continuation の最も良く使われる用法のみを示す。もし全ての使用目的が次の例のように単純な場合だけであれば、call-with-current-continuation のような強力な手続きは必要ない。

```
(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
```

Rationale:

call-with-current-continuation の最も一般的な用法はループや手続き本体からの構造化された非ローカルな脱出であるが、実際には call-with-current-continuation は広範な高度な制御構造の実装に非常に有用である。

Scheme の式が評価される時にはいつでも、その結果を要求する継続 (continuation) が存在する。継続は計算の（なにも特別なことがない場合の）将来のゆくえ全体を示す。例えば、式がトップレベルで評価される時には、継続は結果を受けとり、画面にそれを表示し、次の入力を待ち、それを評価し……と永久に続く。例えば、「結果を受けとってそれにローカル変数に格納された値をかけ算し、7 を足して、結果を表示するトップレベルの継続にそれを渡す」という継続のように、ほとんどの場合において継続はユーザのプログラムで示された動作を含む。大抵の場合はこれらの遍在する継続は舞台裏に隠されていてプログラマはそれらをあまり意識することがない。しかし、たまにプログラマはそれらの継続を明示的に取り扱う必要がある。call-with-current-continuation により、 Scheme のプログラマはそれを、ちょうど現在の継続と同じような動作をする手続きを作ることで実現することができる。

ほとんどのプログラム言語は、exit、return、あるいは goto とさえいうような名前を持つ、1 つかそれ以上の特殊目的の脱出構文を持つ。しかし、1965 年に Peter Landin [16] は J オペレータという汎用の脱出オペレータを考案した。John Reynolds [24] は 1972 年に、同様の能力を持つより簡潔な構造について述べている。1975 年の Scheme のレポートで Sussman と Steele が述べた catch 特殊構文は、名前こそ汎用性の低い MacLisp の構文からとっているものの、Reynolds の構文と全く同じであった。複数の Scheme の実装者が、catch の構文の全ての能力は特殊な構文要素でなくとも関数として実現可能であることに気付き、call-with-current-continuation という名前は 1982 年に作られた。この名前は意味を良く表しているが、このような長い名前の善し悪しは意見がわかれるところであり、かわりに call/cc という名前を使う人もいる。

(values *obj* ...) 手続き

引数を全て自身の継続に渡す。手続き call-with-values で作られた 継続を除き、全ての継続は 1 つの値しかとらない。values は次のように定義できる。

```
(define (values . things)
  (call-with-current-continuation
    (lambda (cont) (apply cont things))))
```

(call-with-values *producer consumer*) 手続き

producer を引数なしで、渡された値を引数として *consumer* を呼び出す継続と共に呼び出す。*consumer* の呼び出しの継続は call-with-values の呼び出しの継続である。

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
                                ⟹  5

(call-with-values * -)        ⟹  -1
```

(dynamic-wind *before thunk after*) 手続き

thunk を引数無しで呼び出し、その返り値を返す。*before* と *after* は、次の規則で必要な場合に引数無しで呼ばれる（call-with-current-continuation で捕捉された継続の呼び出しがない場合には、この 3 つの引数はそれぞれ 1 回ずつ順に呼び出される）。*before* は *thunk* の呼び出しの動的有効範囲に実行が入る時は常に、また *after* は動的有効範囲から実行が出る時に常に呼び出される。手続き呼び出しの動的有効範囲とは、呼び出しが始まってから値が戻されるまでの間である。Scheme においては、call-with-current-continuation が存在するため、呼び出しの動的有効範囲は 1 つの繋がった時間の範囲とはならない。動的有効範囲は次のように定義される。

- 呼び出された手続きの本体の実行が始まる時は、実行が動的有効範囲に入る。
- 実行が動的有効範囲に入っておらず、前に動的実行範囲の中で（call-with-current-continuation によって）捕捉された継続が呼び出された時にも、動的有効範囲に実行が入る。
- 呼び出された手続きが値を返す時には、実行が動的有効範囲から出る。
- 実行が動的有効範囲に入っていて、動的有効範囲の外で捕捉された 継続が呼び出された時にも実行が動的有効範囲から出る。

thunk の実行中に別の dynamic-wind の呼び出しが起こり、その後、その 2 回の dynamic-wind の after が両方とも呼び出されるような形で継続の呼び出しが起こった場合、2 回目の（内側の）dynamic-wind に関係した *after* の方が先に呼び出される。

thunk の実行中に別の dynamic-wind の呼び出しが起こり、その後、その 2 回の dynamic-wind の before が両方とも呼び出されるような形で継続の呼び出しが起こった場合、1 回目の（外側の）dynamic-wind に関係した *before* の方が先に呼び出される。

継続の呼び出しが、ある dynamic-wind の呼び出しの際の *before* と別の際の *after* の両方の呼び出しを必要とする場合、*after* の方が先に呼び出される。

捕捉された継続を使って *before* や *after* の呼び出しの動的有効範囲に出入りした場合の効果は定義されない。

```
(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

eval は、*expression* を *environment-specifier* で指定された環境で評価し、その値を返す。引数のうち *expression* はデータとして表現された Scheme の正しい式でなければならず、*environment-specifier* はこの後に述べる 3 つの手続きの返り値のどれかでなければならない。実装は eval を、式でないプログラム（定義）を第 1 引数にとり、あるいはここで指定された以外の値を第 2 引数にとるように拡張することができるが、eval は null-environment や scheme-report-environment と関係付けられた環境に新しい束縛を作ってはならないという制約がある。

```
(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 は正確な整数 5 でなければならず、この Scheme レポートのリビジョン（the Revised^5 Report on Scheme）を表す。scheme-report-environment はこのレポートで定義された束縛で、必須とされるものと、任意であるとされるうちでその実装でサポートされるものを含みそれ以外は空であるような環境を表す指示子を返す。null-environment はこのレポートで定義され、必須であるか、あるいは任意でありその実装でサポートされているような構文キーワードの（構文）束縛を除き空であるような環境を表す指示子を返す。

version のその他の値は、このレポートの前のリビジョンに対応する環境を示すために用いることができるが、そのサポートは要求されない。実装は *version* が 5 でなく、実装がサポートする他の値でもない場合、エラーを報告する。

scheme-report-environment の中で束縛されている変数（例えば car）に（eval を使って）値を代入した場合の効果は定義されない。したがって、scheme-report-environment で指示される環境は変更不可能であっても良い。

(interaction-environment) 任意手続き

この手続きは実装で定義された束縛、一般的にはこのレポートでリストされたものの上位集合、からなる環境の指示子を返す。この手続きはユーザが動的に入力した式を実装が評価できるような環境を返すことを意図している。

6.6. 入出力

6.6.1. ポート

ポートは入出力装置を表現する。Scheme にとって、入力ポートはコマンドを通じて文字を届けることのできる Scheme のオブジェクトであり、出力ポートは文字を受けとることのできる Scheme の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 はファイル名の文字列であり、*proc* は 1 引数をとる手続きでなければならない。call-with-input-file では、引数に指定されたファイルは既に存在しなければならず、call-with-output-file では、そのファイルが既に存在する時の効果は未定義である。これらの手続きは *proc* を 1 引数で呼び出す。引数は指定されたファイルを入力あるいは出力用に開いて得られたポートである。ファイルが開けなかった場合、エラーが報告される。*proc* から戻ると、ポートは

自動的に閉じられ *proc* の返り値が返される。*proc* から戻らない場合、ポートが今後の読み書きで使われることがないことが証明できる場合を除き、ポートが自動的に閉じられることはない。

Rationale: Scheme の脱出手続きが無限の寿命を持つため、現在の継続から一旦脱出し、再び中に戻ることが可能である。実装が現在の実装からのあらゆる脱出でポートを閉じることを許されているならば、`call-with-current-continuation` と `call-with-input-file` あるいは `call-with-output-file` を同時に使って移植可能なプログラムを書くことはできないであろう。

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

obj がそれぞれ入力ポートあるいは出力ポートであるなら `#t` を、そうでなければ `#f` を返す。

(`current-input-port`) 手続き
(`current-output-port`) 手続き

現在の標準の入力ポートあるいは出力ポートを返す。

(`with-input-from-file` *string thunk*) 任意手続き
(`with-output-to-file` *string thunk*) 任意手続き

string はファイル名の文字列であり、*thunk* は引数のない手続きでなければならない。`with-input-from-file` ではファイルは既に存在しなければならず、`with-output-to-file` ではファイルが既に存在する場合の効果は未定義である。ファイルは入力あるいは出力用に開かれ、それらに接続するポートを `current-input-port` や `current-output-port` で返される（そして (`read`)、(`write` *obj*) などで用いられる）標準のポートとしてから、*thunk* を引数なしで呼び出す。*thunk* から戻ると、ポートは閉じられ、標準ポートは元に戻される。`with-input-from-file` と `with-output-to-file` は、*thunk* の返り値を返す。これらの手続きの継続から脱出手続きで脱出した場合、動作は実装依存である。

(`open-input-file` *filename*) 手続き

既に存在するファイル名の文字列を引数にとり、それらの文字列から文字を読みとることのできる入力ポートを返す。ファイルが開けない時は、エラーが報告される。

(`open-output-file` *filename*) 手続き

新たに作成される出力ファイル名の文字列を引数にとり、そのファイルに書き出すことのできる出力ポートを返す。ファイルが開けない時は、エラーが報告される。指定された名前のファイルが既に存在する時の効果は未定義である。

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

port と関連付けられたファイルを閉じ、それらのポートが文字を書き出したり読み込んだりできないことを示す。これらの手続きは、ファイルが既に閉じられている時にはなにも動作をしない。返り値は未定義である。

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 のオブジェクトの外部表現をそのオブジェクトに変換する。つまり、これは端末以外の ⟨データ⟩（7.1.2 節および 6.3.2 節を見よ）のパーサである。`read` は与えられた入力ポート *port* から解析可能な次のオブジェクトを返し、*port* の状態をオブジェクトの外部表現の終端の次の文字を指すようにする。

オブジェクトの表現を始めることのできる文字を見つけるより先にファイルの終端に到達した場合、ファイル終端オブジェクトが返される。ポートは引続き開かれたままであり、その後の読みだしはまたファイル終端オブジェクトを返す。もしオブジェクトの外部表現が始まってからファイル終端に到達し、その表現が不完全で解析不可能であるならば、エラーが報告される。

port は省略可能であり、その場合は `current-input-port` から返される標準のポートが用いられる。既に閉じられたポートからの読み出しはエラーである。

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

入力ポート *port* から読み出すことのできる次の文字を返し、*port* の状態をその文字の次を指し示すように更新する。もう文字が得られない場合は、ファイル終端オブジェクトが返される。*port* は省略可能であり、その場合は `current-input-port` から返される標準のポートが用いられる。

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

入力ポート *port* から読み出すことのできる次の文字を返すが、*port* の状態はその文字の次を指し示すように更新しない。もう文字が得られない場合は、ファイル終端オブジェクトが返される。*port* は省略可能であり、その場合は `current-input-port` から返される標準のポートが用いられる。

Note: `peek-char` への呼び出しから返される値は同じ *port* への `read-char` から返されるであろう値と同じである。唯一の違いは直後の同じ *port* への `read-char` あるいは `peek-char` への呼び出しがその前の `peek-char` への呼び出しで返される値と同じであることである。具体的には、対話環境へ繋がるポートへの `read-char` の呼び出しが、入力を待ってブロックするような状況では、`peek-char` の呼び出しも同じようにブロックする。

(`eof-object?` *obj*) 手続き

obj がファイル終端オブジェクトである時は `#t` を、そうでない時は `#f` を返す。正確なファイル終端オブジェクトの集合は実装によって異なるが、どのような場合でもファイル終

端オブジェクトは、read によって読みとることのできるオブジェクトではない。

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

文字が入力ポート *port* に準備できている時は #t を、そうでない時は #f を返す。char-ready? が #t を返す時は、その *port* への次の read-char の呼び出しはブロックしないことが保証される。*port* がファイルの終端を指している場合、char-ready? は #t を返す。*port* は省略可能であり、その場合は current-input-port から返される標準のポートが用いられる。

Rationale: char-ready? はプログラムが対話ポートから入力待ちで詰まることなくデータを受けとることができるようにするために存在する。そのようなポートに繋がる入力編集機構は、 char-ready? で存在が確認された文字が削除できないことを保証しなければならない。char-ready? がファイルの終端で #f を返していたならば、ファイル終端のポートはまだ準備できた文字のない対話ポートと区別することができないであろう。

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 の出力表現を *port* に与えられたポートに出力する。出力表記に現れる文字列はダブルクオートで括られ、その文字列中のバックスラッシュとダブルクオートはバックスラッシュ文字でエスケープされる。文字オブジェクトは #\ 表記で書き出される。write の返り値は未定義である。*port* は省略可能であり、その場合は current-output-port から返される標準のポートが用いられる。

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

obj の出力表現を *port* に与えられたポートに出力する。出力表記に現れる文字列はダブルクオートで括られることはなく、文字列中のどの文字もエスケープされない。文字オブジェクトは write でなく write-char で書き出したのと同様の表現で書き出される。display の返り値は未定義である。*port* は省略可能であり、その場合は current-output-port から返される標準のポートが用いられる。

Rationale: write は機械が読みとることのできるな出力を、display は人間が読みとることのできる出力を生成することを意図している。シンボルの “slashification” を認めるような実装では write はシンボル中の普通でない文字を slashify するよう、display はしないようにするであろう。

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

行末を *port* に書き出す。実際に何が行なわれるかはオペレーティングシステムによってことなる。返り値は未定義である。*port* は省略可能であり、その場合は current-output-port から返される標準のポートが用いられる。

(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い）を *port* に書き出す。返り値は未定義である。*port* は省略可能であり、その場合は current-output-port から返される標準のポートが用いられる。

6.6.4. システムとのインタフェース

システムとのインタフェースの問題はこのレポートの対象範囲外である。しかし、次の操作はここで述べるに十分な重要性を持つ。

(load *filename*) 任意手続き

filename は Scheme のソースコードを内容とする既存のファイル名の文字列である。手続き load は式と定義をファイルから読み込み順に評価する。式の値が表示されるかされないかは未定義である。手続き load は current-input-port および current-output-port からの返り値には影響しない。laod の返り値は未定義である。*Rationale:* 移植性のため、load はソースファイルに対して動作しなければならない。その他のファイルに対しての動作は必然的に実装によって異なる。

(transcript-on *filename*) 任意手続き
(transcript-off) 任意手続き

filename は作成される出力ファイルの名前の文字列でなければならない。transcript-on は指示された名前のファイルを出力用に開き、それに続くユーザと Scheme システムの対話の写しをファイルに書き込む効果を持つ。transcript-off はその記録ファイルを閉じ、写しの記録を終了させる。同時には 1 つの記録しかとることができないが、実装によってはこの制限を緩めることができる。これらの手続きの返り値は未定義である。

7. 形式的な文法と意味論

この章ではこのレポートでこれまでに非形式的に述べられた事項の形式的な解説を与える。

7.1. 形式的文法

この節では Scheme の形式的文法をを拡張 BNF で与える。

文法中の全ての空白は読みやすさのためである。大文字小文字は区別されない。例えば `#x1A` と `#X1a` は同等である。⟨空⟩ は空文字列を意味する。

次の BNF の拡張は記述をより簡潔にするものである。⟨thing⟩* は 0 個以上の ⟨thing⟩ の出現を表す。⟨thing⟩$^+$ は 1 つ以上の ⟨thing⟩ を表す。

7.1.1. 構文的構造

この節では、個々のトークン（識別子、数など）を文字の並びから構成する方法について述べる。次の節では式とプログラムをトークンの列から構成する方法について述べる。

⟨トークン間スペース⟩ はあらゆるトークンに隣接するどちらの側にも現れることができるが、トークン内には現れることができない。

トークンのうち暗黙の終了を要求するもの（識別子、数字、文字、ドット）は ⟨区切り文字⟩ で終了できるが、その他のものは必ずしもそうしなくてよい。

5 つの文字 `[ ] { } |` は将来の言語の拡張のために予約されている。

⟨トークン⟩ ⟶ ⟨識別子⟩ | ⟨論理値⟩ | ⟨数字⟩
　| ⟨文字⟩ | ⟨文字列⟩
　| `(` | `)` | `#(` | `'` | `` ` `` | `,` | `,@` | `.`
⟨区切り文字⟩ ⟶ ⟨空白⟩ | `(` | `)` | `"` | `;`
⟨空白⟩ ⟶ ⟨空白文字 または 改行⟩
⟨コメント⟩ ⟶ `;` ⟨ 改行までの
　　あらゆる文字の連続 ⟩
⟨空間⟩ ⟶ ⟨空白⟩ | ⟨コメント⟩
⟨トークン間スペース⟩ ⟶ ⟨空間⟩*

⟨変数⟩ ⟶ ⟨開始文字⟩ ⟨残りの文字⟩*
　| ⟨特別な識別子⟩
⟨開始文字⟩ ⟶ ⟨英文字⟩ | ⟨特別開始文字⟩
⟨英文字⟩ ⟶ `a` | `b` | `c` | ... | `z`

⟨特別開始文字⟩ ⟶ `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=`
　| `>` | `?` | `^` | `_` | `~`
⟨残りの文字⟩ ⟶ ⟨開始文字⟩ | ⟨数字⟩
　| ⟨特別な残りの文字⟩
⟨数字⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨特別な残りの文字⟩ ⟶ `+` | `-` | `.` | `@`
⟨特別な識別子⟩ ⟶ `+` | `-` | ...
⟨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
　　あらゆる ⟨識別子⟩ ⟩

⟨論理値⟩ ⟶ `#t` | `#f`
⟨文字⟩ ⟶ `#\` ⟨任意の文字⟩
　| `#\` ⟨文字名⟩
⟨文字名⟩ ⟶ `space` | `newline`

⟨文字列⟩ ⟶ `"` ⟨文字列の要素⟩* `"`
⟨文字列の要素⟩ ⟶ ⟨", \ 以外の文字⟩
　| `\"` | `\\`

⟨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進符号無し実数⟩、⟨R 進符号無し整数⟩、⟨R 進接頭語⟩は $R = 2, 8, 10, 16$ についてそれぞれ写される。⟨2 進小数⟩、⟨8 進小数⟩、⟨16 進小数⟩は存在しないが、これは小数点あるいは指数表記を含む数は 10 進で書かれなければならないことを示す。

⟨R 進数⟩ ⟶ ⟨R 進接頭語⟩ ⟨R 進複素数⟩
⟨R 進複素数⟩ ⟶ ⟨R 進実数⟩
　| ⟨R 進実数⟩ `@` ⟨R 進実数⟩
　| ⟨R 進実数⟩ `+` ⟨R 進符号無し実数⟩ `i`
　| ⟨R 進実数⟩ `-` ⟨R 進符号無し実数⟩ `i`
　| ⟨R 進実数⟩ `+ i` | ⟨R 進実数⟩ `- i`
　| `+` ⟨R 進符号無し実数⟩ `i`
　| `-` ⟨R 進符号無し実数⟩ `i`
　| `+ i` | `- i`
⟨R 進実数⟩ ⟶ ⟨符号⟩ ⟨R 進符号無し実数⟩
⟨R 進符号無し実数⟩ ⟶ ⟨R 進符号無し整数⟩
　| ⟨R 進符号無し整数⟩ `/` ⟨R 進符号無し整数⟩
　| ⟨R 進小数⟩
⟨10 進小数⟩ ⟶ ⟨10 進符号無し整数⟩ ⟨接尾語⟩
　| `.` ⟨10 進数字⟩$^+$ `#`* ⟨接尾語⟩
　| ⟨10 進数字⟩$^+$ `.` ⟨10 進数字⟩* `#`* ⟨接尾語⟩
　| ⟨10 進数字⟩$^+$ `#`$^+$ `.` `#`* ⟨接尾語⟩
⟨R 進符号無し整数⟩ ⟶ ⟨R 進数字⟩$^+$ `#`*
⟨R 進接頭語⟩ ⟶ ⟨R 進基数記号⟩ ⟨正確さ記号⟩
　| ⟨正確さ記号⟩ ⟨R 進基数記号⟩

⟨接尾語⟩ ⟶ ⟨空⟩
　| ⟨指数記号⟩ ⟨符号⟩ ⟨10 進数字⟩$^+$
⟨指数記号⟩ ⟶ `e` | `s` | `f` | `d` | `l`
⟨符号⟩ ⟶ ⟨空⟩ | `+` | `-`
⟨正確さ記号⟩ ⟶ ⟨空⟩ | `#i` | `#e`
⟨2 進基数記号⟩ ⟶ `#b`
⟨8 進基数記号⟩ ⟶ `#o`

⟨10 進基数記号⟩ ⟶ ⟨空⟩ | `#d`
⟨16 進基数記号⟩ ⟶ `#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⟩ は関数 `read`（6.6.2 節）が正常に解析できるものである。⟨式⟩ として解析できる文字列は必ず ⟨データ⟩ として解析できることに注意せよ。

⟨データ⟩ ⟶ ⟨単純データ⟩ | ⟨複合データ⟩
⟨単純データ⟩ ⟶ ⟨論理値⟩ | ⟨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⟩ | ⟨ベクトル⟩
⟨リスト⟩ ⟶ `(`⟨データ⟩`*)` | `(`⟨データ⟩$^+$ `.` ⟨データ⟩`)`
　| ⟨省略記法⟩
⟨省略記法⟩ ⟶ ⟨省略接頭語⟩ ⟨データ⟩
⟨省略接頭語⟩ ⟶ `'` | `` ` `` | `,` | `,@`
⟨ベクトル⟩ ⟶ `#(`⟨データ⟩`*)`

7.1.3. Expressions

⟨式⟩ ⟶ ⟨変数⟩
　| ⟨リテラル⟩
　| ⟨関数呼びだし⟩
　| ⟨λ 式⟩
　| ⟨条件分岐⟩
　| ⟨代入⟩
　| ⟨派生式⟩
　| ⟨マクロ使用⟩
　| ⟨マクロブロック⟩

⟨リテラル⟩ ⟶ ⟨クオート⟩ | ⟨自己評価⟩
⟨自己評価⟩ ⟶ ⟨論理値⟩ | ⟨数⟩
　| ⟨文字⟩ | ⟨文字列⟩
⟨クオート⟩ ⟶ `'`⟨データ⟩ | `(quote` ⟨データ⟩`)`
⟨関数呼び出し⟩ ⟶ `(`⟨オペレータ⟩ ⟨オペランド⟩`*)`
⟨オペレータ⟩ ⟶ ⟨式⟩
⟨オペランド⟩ ⟶ ⟨式⟩

⟨λ 式⟩ ⟶ `(lambda` ⟨仮引数リスト⟩ ⟨本体⟩`)`
⟨仮引数リスト⟩ ⟶ `(`⟨変数⟩`*)` | ⟨変数⟩
　| `(`⟨変数⟩$^+$ `.` ⟨変数⟩`)`
⟨本体⟩ ⟶ ⟨定義⟩* ⟨式列⟩
⟨式列⟩ ⟶ ⟨コマンド⟩* ⟨式⟩
⟨コマンド⟩ ⟶ ⟨式⟩

⟨条件分岐⟩ ⟶ `(if` ⟨テスト⟩ ⟨真の式⟩ ⟨偽の式⟩`)`
⟨テスト⟩ ⟶ ⟨式⟩
⟨真の式⟩ ⟶ ⟨式⟩
⟨偽の式⟩ ⟶ ⟨式⟩ | ⟨空⟩

⟨代入⟩ ⟶ `(set!` ⟨変数⟩ ⟨式⟩`)`

⟨派生式⟩ ⟶
　　`(cond` ⟨cond 節⟩$^+$`)`
　| `(cond` ⟨cond 節⟩`*` `(else` ⟨式列⟩`))`
　| `(case` ⟨式⟩
　　⟨case 節⟩$^+$`)`
　| `(case` ⟨式⟩
　　⟨case 節⟩*
　　`(else` ⟨式列⟩`)))`
　| `(and` ⟨テスト⟩`*)`
　| `(or` ⟨テスト⟩`*)`
　| `(let (`⟨束縛リスト⟩`*)` ⟨本体⟩`)`
　| `(let` ⟨変数⟩ `(`⟨束縛リスト⟩`*)` ⟨本体⟩`)`
　| `(let*` `(`⟨束縛リスト⟩`*)` ⟨本体⟩`)`
　| `(letrec` `(`⟨束縛リスト⟩`*)` ⟨本体⟩`)`
　| `(begin` ⟨式列⟩`)`
　| `(do (`⟨繰り返し指定⟩`*)`
　　`(`⟨テスト⟩ ⟨do の結果⟩`)`
　　⟨コマンド⟩`*)`
　| `(delay` ⟨式⟩`)`
　| ⟨quasi クオート⟩

⟨cond 節⟩ ⟶ `(`⟨テスト⟩ ⟨式列⟩`)`
　| `(`⟨テスト⟩`)`
　| `(`⟨テスト⟩ `=>` ⟨受け取り先⟩`)`
⟨受け取り先⟩ ⟶ ⟨式⟩
⟨case 節⟩ ⟶ `((`⟨データ⟩`*)` ⟨式列⟩`)`
⟨束縛リスト⟩ ⟶ `(`⟨変数⟩ ⟨式⟩`)`
⟨繰り返し指定⟩ ⟶ `(`⟨変数⟩ ⟨初期化子⟩ ⟨更新値⟩`)`
　| `(`⟨変数⟩ ⟨初期化子⟩`)`
⟨初期化子⟩ ⟶ ⟨式⟩
⟨更新値⟩ ⟶ ⟨式⟩
⟨do の結果⟩ ⟶ ⟨式列⟩ | ⟨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 (`⟨構文指定⟩`*)` ⟨本体⟩`)`
　| `(letrec-syntax (`⟨構文指定⟩`*)` ⟨本体⟩`)`
⟨構文指定⟩ ⟶ `(`⟨キーワード⟩ ⟨変換子表現⟩`)`

7.1.4. quasi クオート

次の quasi クオート式に対する文法は文脈自由でない。これは無限の生成規則を生成する処方として書かれている。次のルールを $D = 1, 2, 3, \ldots$ について展開したものを考えられたい。D はネストレベルを記憶する。

⟨quasi クオート⟩ ⟶ ⟨quasi クオート 1⟩
⟨qq テンプレート 0⟩ ⟶ ⟨式⟩

⟨quasi クオート 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⟩
　| ⟨ベクトル qq テンプレート D⟩
　| ⟨アンクオート D⟩
⟨リスト qq テンプレート D⟩ ⟶
　(⟨qq テンプレートまたは接合 D⟩*)
　| (⟨qq テンプレートまたは接合 D⟩$^+$.
　　⟨qq テンプレート D⟩)
　| '⟨qq テンプレート D⟩
　| ⟨quasi クオート $D+1$⟩
⟨ベクトル qq テンプレート D⟩ ⟶
　#(⟨qq テンプレートまたは接合 D⟩*)
⟨アンクオート D⟩ ⟶ ,⟨qq テンプレート $D-1$⟩
　| (unquote ⟨qq テンプレート $D-1$⟩)
⟨qq テンプレートまたは接合 D⟩ ⟶
　⟨qq テンプレート D⟩
　| ⟨接合アンクオート D⟩
⟨接合アンクオート D⟩ ⟶ ,@⟨qq テンプレート $D-1$⟩
　| (unquote-splicing ⟨qq テンプレート $D-1$⟩)

⟨quasi クオート⟩ の中では、⟨リスト qq テンプレート D⟩ は ⟨アンクオート D⟩ および ⟨接合アンクオート D⟩ と混同されることがある。この場合、⟨アンクオート D⟩ および ⟨接合アンクオート D⟩ を優先して解析される。

7.1.5. 変換指定子

⟨変換子表現⟩ ⟶
　(syntax-rules (⟨識別子⟩*) ⟨構文規則⟩*)
⟨構文規則⟩ ⟶ (⟨パターン⟩ ⟨テンプレート⟩)
⟨パターン⟩ ⟶ ⟨パターン識別子⟩
　| (⟨パターン⟩*)
　| (⟨パターン⟩$^+$. ⟨パターン⟩)
　| (⟨パターン⟩* ⟨パターン⟩ ⟨省略記号⟩)
　| #(⟨パターン⟩*)
　| #(⟨パターン⟩* ⟨パターン⟩ ⟨省略記号⟩)
　| ⟨パターンデータ⟩
⟨パターンデータ⟩ ⟶ ⟨文字列⟩
　| ⟨文字⟩
　| ⟨論理値⟩
　| ⟨数⟩
⟨テンプレート⟩ ⟶ ⟨パターン識別子⟩
　| (⟨テンプレート要素⟩*)
　| (⟨テンプレート要素⟩$^+$. ⟨テンプレート⟩)
　| #(⟨テンプレート要素⟩*)
　| ⟨テンプレートデータ⟩
⟨テンプレート要素⟩ ⟶ ⟨テンプレート⟩
　| ⟨テンプレート⟩ ⟨省略記号⟩
⟨テンプレートデータ⟩ ⟶ ⟨パターンデータ⟩
⟨パターン識別子⟩ ⟶ ⟨... 以外の識別子⟩
⟨省略記号⟩ ⟶ ⟨識別子 ...⟩

7.1.6. プログラムと定義

⟨プログラム⟩ ⟶ ⟨コマンドまたは定義⟩*
⟨コマンドまたは定義⟩ ⟶ ⟨コマンド⟩
　| ⟨定義⟩
　| ⟨構文定義⟩
　| (begin ⟨コマンドまたは定義⟩$^+$)
⟨定義⟩ ⟶ (define ⟨変数⟩ ⟨式⟩)
　| (define (⟨変数⟩ ⟨def 仮引数⟩) ⟨本体⟩)
　| (begin ⟨定義⟩*)
⟨def 仮引数⟩ ⟶ ⟨変数⟩*
　| ⟨変数⟩* . ⟨変数⟩
⟨構文定義⟩ ⟶
　(define-syntax ⟨キーワード⟩ ⟨変換子表現⟩)

7.2. 正式な意味論

この節では Scheme の基本式といくつかの組み込み手続きに関しての正式な表示的意味論を示す。ここで用いられる概念と表記については [29] に述べられている。表記を次にまとめる。

$\langle \ldots \rangle$	列の構成
$s \downarrow k$	列 s の k 番目の要素（1 から始まる）
$\#s$	列 s の長さ
$s \S t$	列 s と t の連接
$s \dagger k$	s から最初の k 要素を落す
$t \to a, b$	McCarthy の条件分岐 “if t then a else b”
$\rho[x/i]$	置換 “ρ with x for i”
x in D	injection of x into domain D
$x \mid$ D	x の領域 D への射影

式の継続が 1 つの値でなく値の列をとるのは、手続き呼び出しと複数の返り値の正式な扱いを単純化するためである。

ペア・ベクトル・文字列に付けられている論理値のフラグは変更可能な時に真、変更不可能な時に偽となる。

呼び出しの時の評価順は未定義である。これを模擬するために、互いに逆関数となる任意の並び変え *permute* と *unpermute* を引数とそれの評価結果にそれぞれ適用している。これは正確には正しくない。なぜなら、これは評価順がどんな引数の数にたいしてもプログラム中で固定されているということを誤って意味してしまうが、これは意図する意味論を左から右へと評価されるものよりは正確に表現している。

記憶割当子 *new* は実装依存であるが、つぎの公理を満たさなければならない。$new\,\sigma \in \mathrm{L}$, であるなら $\sigma(new\sigma \mid \mathrm{L}) \downarrow 2 = \mathit{false}$.

$\mathcal{K}$ の定義は省略した。それは適切な $\mathcal{K}$ の定義は意味なく意味論を複雑にするからである。

P が全ての変数を参照あるいは代入の前に定義するようなプログラムであるならば、P の意味は

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

となる。ここで I* は P で定義された変数の列、P′ は全ての P 中の定義を代入で置き換えて得られる式の列、⟨undefined⟩ は *undefined* に評価される式、$\mathcal{E}$ は式に意味を割り当てる意味関数である。

7.2.1. 抽象的な構文

K ∈ Con	クオートを含む定数
I ∈ Ide	識別子（変数）
E ∈ Exp	式
Γ ∈ Com = Exp	コマンド

Exp ⟶ K | I | (E$_0$ E*)
| (lambda (I*) Γ* E$_0$)
| (lambda (I* . I) Γ* E$_0$)
| (lambda I Γ* E$_0$)
| (if E$_0$ E$_1$ E$_2$) | (if E$_0$ E$_1$)
| (set! I E)

7.2.2. 領域の一覧

$\alpha \in \mathtt{L}$		ロケーション
$\nu \in \mathtt{N}$		自然数
$\mathtt{T}$	$= \{\mathit{false}, \mathit{true}\}$	論理値
$\mathtt{Q}$		シンボル
$\mathtt{H}$		文字
$\mathtt{R}$		数
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	ペア
$\mathtt{E_v}$	$= \mathtt{L}^* \times \mathtt{T}$	ベクトル
$\mathtt{E_s}$	$= \mathtt{L}^* \times \mathtt{T}$	文字列
$\mathtt{M}$	$= \{\mathit{false}, \mathit{true}, \mathit{null}, \mathit{undefined}, \mathit{unspecified}\}$	雑
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$	手続きの値
$\epsilon \in \mathtt{E}$	$= \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	式の値
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	格納
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	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
$\mathtt{X}$		エラー

7.2.3. 意味論関数

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

$\mathcal{K}$ の定義は敢えて省いている。

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

$$\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\;(\mathit{single}(\lambda\epsilon\,.\,\epsilon = \mathit{undefined} \to \mathit{wrong}\ \text{“未定義の変数”},\ \mathit{send}\,\epsilon\,\kappa))$$

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

$$\mathcal{E}[\![\mathtt{(lambda\ (}\mathrm{I}^*\mathtt{)}\ \Gamma^*\ \mathrm{E_0}\mathtt{)}]\!] = \lambda\rho\kappa\,.\,\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \to \mathit{send}\,(\langle \mathit{new}\,\sigma\,|\,\mathtt{L},\ \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* = \#\mathrm{I}^* \to \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\kappa'))\;(\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\;\epsilon^*,\ \mathit{wrong}\ \text{“引数の数が不正”}\rangle\ \text{in E})\;\kappa\;(\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathtt{L})\,\mathit{unspecified}\,\sigma),\ \mathit{wrong}\ \text{“メモリ不足”}\,\sigma$$

$$\mathcal{E}[\![\mathtt{(lambda\ (}\mathrm{I}^*\ \mathtt{.}\ \mathrm{I}\mathtt{)}\ \Gamma^*\ \mathrm{E_0}\mathtt{)}]\!] = \lambda\rho\kappa\,.\,\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \to \mathit{send}\,(\langle \mathit{new}\,\sigma\,|\,\mathtt{L},\ \lambda\epsilon^*\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to \mathit{tievalsrest}\;(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\kappa'))\;(\mathit{extends}\,\rho\,(\mathrm{I}^* \S \langle \mathrm{I}\rangle)\,\alpha^*))\;\epsilon^*\;(\#\mathrm{I}^*),\ \mathit{wrong}\ \text{“引数が不足”}\rangle\ \text{in E})\;\kappa\;(\mathit{update}\,(\mathit{new}\,\sigma\,|\,\mathtt{L})\,\mathit{unspecified}\,\sigma),\ \mathit{wrong}\ \text{“メモリ不足”}\,\sigma$$

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

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

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

ここおよび他の場所で、 *undefined* を除くあらゆる式の値を *unspecified* の位置で用いることができる。

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

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

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

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

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

7.2.4. 補助関数

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

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

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

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

$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$
$single =$
$\quad \lambda\psi\epsilon^* \,.\, \#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$
$\qquad\qquad wrong$ “返り値の数が不正”

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

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

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

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

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

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

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

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

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

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

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

$applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$applicate =$
$\quad \lambda\epsilon\epsilon^*\kappa \,.\, \epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\kappa, wrong$ “不適当な手続き”

$onearg : (\mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$
$onearg =$
$\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\kappa,$
$\qquad\qquad wrong$ “引数の数が不正”

$twoarg : (\mathtt{E} \to \mathtt{E} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{K} \to \mathtt{C})$
$twoarg =$
$\quad \lambda\zeta\epsilon^*\kappa \,.\, \#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\kappa,$
$\qquad\qquad wrong$ “引数の数が不正”

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

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

$less : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$less =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to$
$\qquad\qquad send\,(\epsilon_1 \mid \mathtt{R} < \epsilon_2 \mid \mathtt{R} \to true, false)\kappa,$
$\qquad\qquad wrong$ “< への数でない引数”)

$add : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$add =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to$
$\qquad\qquad send\,((\epsilon_1 \mid \mathtt{R} + \epsilon_2 \mid \mathtt{R}) \text{ in } \mathtt{E})\kappa,$
$\qquad\qquad wrong$ “+ への数でない引数”)

$car : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$car =$
$\quad onearg\,(\lambda\epsilon\kappa \,.\, \epsilon \in \mathtt{E_p} \to hold\,(\epsilon \mid \mathtt{E_p} \downarrow 1)\kappa,$
$\qquad\qquad wrong$ “car へのペアでない引数”)

$cdr : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$ [similar to car] [car に類似]

$setcar : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C}$
$setcar =$
$\quad twoarg\,(\lambda\epsilon_1\epsilon_2\kappa \,.\, \epsilon_1 \in \mathtt{E_p} \to$
$\qquad\qquad (\epsilon_1 \mid \mathtt{E_p} \downarrow 3) \to assign\,(\epsilon_1 \mid \mathtt{E_p} \downarrow 1)$
$\qquad\qquad\qquad\qquad\qquad\quad \epsilon_2$
$\qquad\qquad\qquad\qquad\qquad\quad (send\,unspecified\,\kappa),$
$\qquad\qquad wrong$ “set-car! への変更不可能な引数”,
$\qquad\qquad wrong$ “set-car! へのペアでない引数”)

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

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

$$
\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\
\mathit{apply} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\epsilon_1 \in \mathtt{F} \rightarrow \mathit{valueslist}\,\langle\epsilon_2\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_1\epsilon^*\kappa), \\
\qquad\qquad \mathit{wrong}\ \text{“}\mathtt{apply}\ \text{への手続き引数が不適切”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{valueslist} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\
\mathit{valueslist} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \rightarrow \\
\qquad\qquad \mathit{cdr}\langle\epsilon\rangle \\
\qquad\qquad\quad (\lambda\epsilon^*\,.\,\mathit{valueslist} \\
\qquad\qquad\qquad\quad \epsilon^* \\
\qquad\qquad\qquad\quad (\lambda\epsilon^*\,.\,\mathit{car}\langle\epsilon\rangle(\mathit{single}(\lambda\epsilon\,.\,\kappa(\langle\epsilon\rangle \S \epsilon^*))))), \\
\qquad\qquad \epsilon = \mathit{null} \rightarrow \kappa\langle\,\rangle, \\
\qquad\qquad \mathit{wrong}\ \text{“}\mathtt{values\text{-}list}\ \text{へのリストでない引数”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \quad [\mathtt{call\text{-}with\text{-}current\text{-}continuation}] \\
\mathit{cwcc} = \\
\quad \mathit{onearg}\,(\lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{F} \rightarrow \\
\qquad\qquad (\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \rightarrow \\
\qquad\qquad\qquad\quad \mathit{applicate}\,\epsilon \\
\qquad\qquad\qquad\qquad\quad \langle\langle \mathit{new}\,\sigma \mid \mathtt{L}, \lambda\epsilon^*\kappa'\,.\,\kappa\epsilon^*\rangle\ \mathrm{in}\ \mathtt{E}\rangle \\
\qquad\qquad\qquad\qquad\quad \kappa \\
\qquad\qquad\qquad\qquad\quad (\mathit{update}\,(\mathit{new}\,\sigma \mid \mathtt{L}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{unspecified} \\
\qquad\qquad\qquad\qquad\qquad\qquad \sigma), \\
\qquad\qquad\qquad\quad \mathit{wrong}\ \text{“メモリ不足”}\ \sigma), \\
\qquad\qquad \mathit{wrong}\ \text{“手続き引数が不適当”})
\end{array}
$$

$$
\begin{array}{l}
\mathit{values} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \\
\mathit{values} = \lambda\epsilon^*\kappa\,.\,\kappa\epsilon^*
\end{array}
$$

$$
\begin{array}{l}
\mathit{cwv} : \mathtt{E}^* \rightarrow \mathtt{K} \rightarrow \mathtt{C} \quad [\mathtt{call\text{-}with\text{-}values}] \\
\mathit{cwv} = \\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\,.\,\mathit{applicate}\,\epsilon_1\langle\,\rangle(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_2\,\epsilon^*))
\end{array}
$$

7.3. 派生式

この節では派生式の基本式（リテラル、変数、手続き呼び出し、lambda、if、set!）を用いたマクロ定義を示す。delay の定義の例は 6.4 節を参照せよ。

```
(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 ...)))))
```

次の letrec マクロはシンボル <undefined> を、ある場所にその値を格納するとその場所に格納された値を読みだすことをエラーとするような値（そのような値は Scheme では定義されていない）の場所で用いている。値を評価する順番を指示することを防ぐために必要な仮の名前を作るのにトリックを用いている。これは補助マクロを用いることでも実現できる。

```
(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 ...)))))
```

begin に対する次の定義はλ式の本体に複数の式を書くことのできるという機能を用いていない。どちらにしても、これらの規則は begin の本体に定義を含まない時にのみ適用できることに注意せよ。

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp)
     exp)
    ((begin exp1 exp2 ...)
     (let ((x exp1))
       (begin exp2 ...)))))
```

次の do の定義は変数の節を展開するのにトリックを用いている。上の letrec 同様、補助マクロを用いても良い。式 (if #f #f) を未定義値を得るのに用いている。

```
(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)))
```

注釈

言語の変更点

この節では “Revised^4 report” [6] が発行されて以降の変更点を挙げる。

- このレポートは今では Scheme に関する IEEE 標準 [13] の上位セットとなっている。すなわちこのレポートに合致する実装は IEEE 標準にも合致する。そのために次の変更を必要とした。
  - 空リストは「真」の論理値として扱われることが要求されるようになった。
  - 機能の essential, inessential の分類は撤廃された。これらは組み込み手続きの 3 つの分類、すなわち必須 primitive、ライブラリ library、任意 optional の区別となった。任意に分類される手続きは `load`, `with-input-from-file`, `with-output-to-file`, `transcript-on`, `transcript-off`, そして `interaction-environment` と、3 引数以上の `-` および `/` である。これらはいずれも IEEE 標準にはない。
  - プログラムは組み込み手続きを再定義しても良くなった。これをしても他の組み込み手続きの動作には影響を与えない。
- ポートは独立な型のリストに追加された。
- マクロに関する付録は削除された。高レベルマクロはレポートの本体の一部となった。派生式の書き換えルールはマクロ定義で置き換えられた。予約された識別子は存在しない。
- `syntax-rules` はベクトルパターンを認めるようになった。
- 複数の返り値、`eval`、`dynamic-wind` が追加された。
- 正確な末尾再帰の方法で実装されることが必要とされる呼び出しを明確に定義した。
- ‘`@`’ は識別子中で用いることができる。‘`|`’ は将来の拡張のために予約されている。

追加資料

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

の Internet Scheme Repository には、論文、プログラム、実装、その他の Scheme 関連の資料といった広範な Scheme の参考資料がある。

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 法を用いて積分する。

引数 system-derivative は系の状態（状態変数 $y_1, \ldots, y_n$ の値のベクトル）をとって系の微分係数（$y'_1, \ldots, y'_n$の値）を返す関数である。引数 `initial-state` は系の初期状態を、`h` は積分間隔の初期推測値を与える。

`integrate-system` の返り値は系の状態の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状態の微分係数をつくり出す関数 `f` を引数にとる。`runge-kutta-4` は系の状態をとり新しい系の状態を返す関数を返す。

```
(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 に類似している。第 1 引数（手続き）を第 2 引数（ストリーム）の全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、car にストリームの第 1 要素を、そして第 2 引数に残りのストリームの promise を持つペアとして実装されている。

```
(define head car)
(define tail
  (lambda (stream) (force (cdr stream))))
```

減衰オシレータの系

$$C\frac{dv_C}{dt} = -i_L - \frac{v_C}{R}$$

$$L\frac{di_L}{dt} = v_C$$

の積分に integrate-system を用いた例を次に示す。

```
(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、手続きの索引

それぞれの単語、関数、キーワードの最も主となる項目は各項の最初に、別の項目とはセミコロンで区切って示されている。