アルゴリズム言語Scheme報告書改^7

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

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

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

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

John McCarthy および *Daniel Weinreb* の霊前に捧ぐ

March 26, 2017

要約

この報告書はプログラミング言語 Scheme の定義的記述を記載しています。Scheme は静的なスコープを持つ真正末尾再帰な Lisp プログラミング言語 [23] の方言で、Guy Lewis Steele Jr. および Gerald Jay Sussman によって発明されました。非常に明確でシンプルな意味論を持ち、わずかな構文だけで式を構成できるよう設計されています。手続き型、関数型、オブジェクト指向スタイルなど、幅広い様々なプログラミングパラダイムが Scheme で簡単に表現できます。

導入部では Scheme およびその報告書の歴史を簡単に述べます。

最初の 3 つの章では Scheme の基礎となる考え方を示し、その言語を説明するため、およびその言語でプログラムを書くために使う記法について述べます。

4 章および 5 章では式、定義、プログラム、ライブラリの構文および意味論について述べます。

6 章では Scheme の組み込み手続き、すなわち Scheme のデータ操作および入出力プリミティブのすべてについて述べます。

7 章では拡張 BNF 記法で記述された Scheme の形式構文を、その表示的意味論と共に掲載します。その後に Scheme の使用例も掲載します。

付録 A には標準ライブラリとそこからエクスポートされている識別子の一覧を掲載します。

付録 B には標準化されているけれどもオプショナルな処理系の機能名の一覧を掲載します。

最後に参考文献の一覧とアルファベット順の索引を掲載し、この報告書を締めくくります。

メモ: R^{5}RS および R^{6}RS から相当の部分がこの報告書に直接複写されており、そのため R^{5}RS および R^{6}RS の編集者をこの報告書の著者の一覧に掲載しています。これらの編集者が、個人的にあるいは共同で、この報告書を支持しているとかいないとかいった意味はありません。

CONTENTS

導入

プログラミング言語は機能の上に機能を積み重ねるのではなく、機能追加の要因となる弱点や制限を取り除くことによって設計するべきです。組み合わせ方に制限さえなければ、式を構成する非常に少数の規則だけで今日使われる主なプログラミングパラダイムのほとんどをサポートするのに十分柔軟で実用的かつ効率的なプログラミング言語を十分に形作れます。Scheme がそれを実証しています。

Scheme はラムダ計算で用いられるような第一級の手続きを取り入れた最初のプログラミング言語のひとつです。それにより動的な型を持つ言語において静的スコープ規則およびブロック構造が有用であることを示しました。Scheme はラムダ式とシンボルから手続きを独立させ、すべての変数に単一の字句環境を用い、手続き呼び出しにおける演算子の位置を被演算子の位置と同じように評価する最初の主要な Lisp 方言です。Scheme は繰り返しを表現するために全面的に手続き呼び出しに依存することにより、手続きの末尾呼び出しが本質的には引数を渡す GOTO であるという事実を強調しました。それにより一貫性と効率性を両立するプログラミングスタイルが可能となりました。Scheme は第一級の脱出手続きを採用した最初の広く使われたプログラミング言語です。これによりそれまで知られていた逐次実行の制御構造はすべて合成することができるようになりました。後のバージョンでは正確な数値および不正確な数値が導入されました。これは Common Lisp の汎用算術の拡張です。さらに近年では Scheme は衛生的なマクロをサポートした最初のプログラミング言語になりました。これにより一貫性があり信頼できる方法でブロック構造言語の構文を拡張できます。

背景

Scheme の最初の記述は 1975 年に書かれました [35]。報告書の改定版 [31] は 1978 年に出され、MIT による処理系が革新的なコンパイラをサポートするようアップグレードすると共に言語の進化が述べられました [32]。1981 年および 1982 年に MIT、イェール大学、およびインディアナ大学の課程で Scheme の亜種を用いるための 3 つの異なるプロジェクトが始められました [27, 24, 14]。1984 年には Scheme を用いたコンピュータサイエンスの入門用の教科書が出版されました [1]。

Scheme はさらに広まり、ローカルな方言が枝分かれし始め、学生と研究者がお互いに書いたコードを理解するのがしばしば難しくなってきました。そのため、より優れた、より広く受け入れられる Scheme の標準を作るために、1984 年の 10 月、15 人の主要な Scheme 処理系の代表者が集まりました。彼らの報告書 RRRS [8] は 1985 年の夏に MIT およびインディアナ大学で出版されました。1986 年の春にさらなる改定が加えられ、R^3RS [29] が作られました。1988 年の春の作業では R^4RS [10] が作られ、これは 1991 年の Scheme プログラミング言語の IEEE 標準 [18] の基礎となりました。1998 年には高水準の衛生的なマクロ、複数の戻り値、および eval などいくつかの追加要素が IEEE 標準に加えられ、R^5RS [20] としてまとめられました。

2006 年の秋、さらなる野心的な標準を作る作業が始められました。それには数多くの新たな改良と移植性の向上に焦点を置いたより厳密な要求事項が含まれていました。その成果である標準 R^6RS は 2007 年の 8 月に完成し [33]、言語のコアと必須の標準ライブラリの集合として編纂されました。これに準拠した新たな Scheme 処理系がいくつも作られました。しかしながら既存の R^5RS 処理系 (実質的にメンテナンスされていないものを除いても) はほとんど R^6RS を採用せず、またはその一部だけを選択的に採用したに過ぎませんでした。

その結果、2009 年の 8 月、Scheme 標準化委員会は標準を別々の、しかし互換性のある 2 つの言語—「小さな」言語と「大きな」言語に分割する決定を下しました。前者は教育者や研究者、埋め込み言語のユーザなどに適したもので、R^5RS との互換性に焦点を置いています。後者は主流のソフトウェア開発における実用的なニーズに焦点を置いたもので、R^6RS の置き換えとなることを意図しています。この報告書はそのうちの「小さな」言語について記述したものです。そのためこれを単独で R^6RS の後継と見なすことはできません。

謝辞

支援と助言を戴いた標準化委員会のメンバー William Clinger, Marc Feeley, Chris Hanson, Jonathan Rees, および Olin Shivers に感謝の意を表します。

この報告書はコミュニティの多大な努力の成果であり、以下の人々を含めコメントやフィードバックを戴いたすべての人に感謝の意を表します: David Adler, Eli Barzilay, Taylan Ulrich Bayırlı/Kammer, Marco Benelli, Pierpaolo Bernardi, Peter Bex, Per Bothner, John Boyle, Taylor Campbell, Raffael Cavallaro, Ray Dillinger, Biep Durieux, Sztefan Edwards, Helmut Eller, Justin Ethier, Jay Reynolds Freeman, Tony Garnock-Jones, Alan Manuel Gloria, Steve Hafner, Sven Hartrumpf, Brian Harvey, Moritz Heidkamp, Jean-Michel Hufflen, Aubrey Jaffer, Takashi Kato, Shiro Kawai, Richard Kelsey, Oleg Kiselyov, Pjotr Kourzanov, Jonathan Kraut, Daniel Krueger, Christian Stigen Larsen, Noah Lavine, Stephen Leach, Larry D. Lee, Kun Liang, Thomas Lord, Vincent Stewart Manis, Perry Metzger, Michael Montague, Mikael More, Vitaly Magerya, Vincent Manis, Vassil Nikolov, Joseph Wayne Norton, Yuki Okumura, Daichi Oohashi, Jeronimo Pellegrini, Jussi Piitulainen, Alex Queiroz, Jim Rees, Grant Rettke, Andrew Robbins, Devon Schudy, Bakul Shah, Robert Smith,

Arthur Smyles, Michael Sperber, John David Stone, Jay Sulzberger, Malcolm Tredinnick, Sam Tobin-Hochstadt, Andre van Tonder, Daniel Villeneuve, Denis Washington, Alan Watson, Mark H. Weaver, Göran Weinholt, David A. Wheeler, Andy Wingo, James Wise, Jörg F. Wittenberger, Kevin A. Wortman, Sascha Ziemann

さらに過去の編集者や彼らを手助けしたすべての人に感謝の意を表します: Hal Abelson, Norman Adams, David Bartley, Alan Bawden, Michael Blair, Gary Brooks, George Carrette, Andy Cromarty, Pavel Curtis, Jeff Dalton, Olivier Danvy, Ken Dickey, Bruce Duba, Robert Findler, Andy Freeman, Richard Gabriel, Yekta Gürsel, Ken Haase, Robert Halstead, Robert Hieb, Paul Hudak, Morry Katz, Eugene Kohlbecker, Chris Lindblad, Jacob Matthews, Mark Meyer, Jim Miller, Don Oxley, Jim Philbin, Kent Pitman, John Ramsdell, Guillermo Rozas, Mike Shaff, Jonathan Shapiro, Guy Steele, Julie Sussman, Perry Wagle, Mitchel Wand, Daniel Weise, Henry Wu, および Ozan Yigit. Scheme 311 バージョン 4 のリファレンスマニュアルから文章の拝借を許可いただいた Carol Fessenden, Daniel Friedman, および Christopher Haynes に感謝します。*TI Scheme Language Reference Manual* [37] の文章を許可いただいた Texas Instruments, Inc. に感謝します。影響を受けた MIT Scheme [24], T [28], Scheme 84 [15], Common Lisp [34], および Algol 60 [25] のマニュアルと共に、`http://srfi.schemers.org` で公開されている SRFI 0, 1, 4, 6, 9, 11, 13, 16, 30, 34, 39, 43, 46, 62, および 87 に感謝の意を表します。

言語の説明

1. Scheme の概要

1.1. 意味論

この節では Scheme の意味論の概要を述べます。非形式的な意味論は 3 章から 6 章で詳細に述べます。リファレンス目的のため 7.2 節に Scheme の形式意味論を掲載しています。

Scheme は静的なスコープを持つプログラミング言語です。変数の各使用はその変数の字句的に見えている束縛に紐付けられます。

Scheme は動的な型を持つ言語です。型は変数ではなく値 (オブジェクトと呼ばれることもあります) に紐付けられます。反対に静的な型を持つ言語では、型は値だけでなく変数や式にも紐付けられます。

Scheme の計算中に作成されるすべてのオブジェクトは無制限の生存期間を持ちます。手続きや継続もそれに含まれます。Scheme のオブジェクトは破棄されることはありません。Scheme の処理系が (通常は!) 記憶領域を使い切ることが無いのは、あるオブジェクトが将来のいかなる計算にも影響を与える可能性がないと保証できる場合にそのオブジェクトが占有している記憶領域を回収することができるためです。

Scheme 処理系は真正末尾再帰であることが要求されます。これにより繰り返し計算が構文的には再帰手続きとして記述されていても一定の空間内で実行することが可能になります。処理系が真正末尾再帰であることにより通常の手続き呼び出しを用いて繰り返しを表現することができます。そのため特殊な繰り返し構文は構文糖衣としての価値しかありません。3.5 節を参照してください。

Scheme の手続きは独立したオブジェクトです。手続きは動的に作成したり、データ構造に格納したり、手続きの結果として返したりすることができます。

Scheme 特有の機能のひとつに「第一級」の地位を持つ継続があります。これは他のほとんどの言語では水面下にしか存在しないものです。継続を利用することで非局所脱出、バックトラッキング、コルーチンなど、幅広い様々な高度な制御構造を実装できます。6.10 節を参照してください。

Scheme の手続きの引数は常に値渡しです。つまり実引数の式は手続きが制御を得る前に必要とされるか否かに関わらず評価されます。

Scheme の数値計算モデルはコンピュータ内における特定の数値表現方式になるべく依存しないよう設計されています。Scheme ではすべての整数は有理数であり、すべての有理数は実数であり、すべての実数は複素数であります。そのため多くのプログラミング言語では整数と実数の違いが非常に重要ですが、Scheme ではそうではありません。Scheme でそれに当たるものは正確な数値計算と不正確な数値計算の違いになります。正確な数値計算は数学における理想的な数値計算に対応するもので、不正確な数値計算は近似値によるものです。正確な数値計算は整数に限定されるものではありません。

1.2. 構文

Scheme は他の Lisp 方言と同様、完全に括弧で囲った前置記法を採用しており、これでプログラムやその他のデータを記述します。データに対しては Scheme の文法の部分言語が用いられます。Scheme のプログラムで他の Scheme のプログラムやデータを簡単に統一的に扱うことができる、というのがこのシンプルで統一された表現の重要なポイントです。例えば `eval` 手続きを用いて、データとして表現された Scheme のプログラムを評価できます。

`read` 手続きは読み込んだデータを字句的に分解すると共に構文的にも分解します。`read` 手続きは入力をプログラムではなくデータ (7.1.2 節) としてパースします。

Scheme の形式構文は 7.1 節に掲載されています。

1.3. 記法および用語

1.3.1. 基本的な機能と選択的な機能

この報告書で定義されているすべての識別子はひとつ以上の**ライブラリ**の中に現れます。*base* **ライブラリ**で定義されている識別子はこの報告書の本文では特別な印は付いていません。このライブラリには Scheme の中核となる構文とデータを操作する一般的に有用な手続きが含まれています。例えば変数 `abs` は引数をひとつ取り数値の絶対値を計算する手続きに束縛されており、変数 `+` は和を計算する手続きに束縛されています。すべての標準ライブラリとそこからエクスポートされている識別子の完全なリストは付録 A に掲載されています。

すべての Scheme 処理系は、

- base ライブラリとそこからエクスポートされているすべての識別子を提供しなければなりません。
- この報告書に記載されているそれ以外のライブラリは提供しても省いても構いません。ただしそれぞれのライブラリは完全に提供するか完全に省くかのいずれかでなければなりません。余計な識別子を追加することも認められません。
- この報告書に記載されていない他のライブラリを提供しても構いません。
- この報告書に記載されている任意の識別子の機能を拡張しても構いません。ただしその拡張はこの報告書の言語と矛盾していてはいけません。
- 移植性のあるコードをサポートするために、この報告書に記載されている字句構文と矛盾しない字句構文を使用するモードを提供しなければなりません。

1.3.2. エラー状態と未規定の動作

エラー状態について述べるとき、この報告書では「エラーが通知されます」という用語を用いて、処理系がエラーを検出し、報告しなければならないことを示します。エラーは 6.11 節に記載されている手続き `raise` を呼び出したかのように、継続不可能な例外が発生することにより通知されます。発生するオブジェクトは処理系依存です。以前に同じ用途で使われたオブジェクトと異なっている必要はありません。この報告書に記載されている場面でエラーが通知される以外にも、プログラマーは独自のエラーを通知させたり通知されたエラーを処理したりすることができます。

「〜を満たすエラーが通知されます」という用語も上で述べたようなエラーの通知を意味しますが、それに加え通知されたオブジェクトが指定された述語 (`file-error?` または `read-error?`) に渡されると、その述語が `#t` を返します。

エラーに関する議論でそのような語句が現れない場合、処理系がエラーを検出したり報告することは推奨されますが要求されません。そのような状況では常にではありませんが「エラーです」という用語が使われます。そのような状況では処理系はエラーを通知しても通知しなくても構いません。エラーを通知する場合、通知されるオブジェクトは述語 `error-object?`、`file-error?`、`read-error?` を満たしても満たさなくても構いません。それらの代わりに移植性の無い拡張を提供しても構いません。

例えば、扱えると明示的に規定されていない型の引数を手続きに渡すことはエラーです。たとえそのような定義域エラーがこの報告書であまり言及されていなくてもです。処理系はエラーを通知しても構いませんし、手続きの定義域を拡張してそのような引数を扱えるようにしても構いませんし、何らかの破滅的な結果を引き起こしても構いません。

処理系が課す何らかの制限のために、正しいプログラムの実行を続けることができない、という状況がありえます。そのような状況を示すときには「処理系の制限の違反を報告しても構いません」という用語が使われます。そのような状況では処理系はその旨を報告しても構いません。処理系の制限は無いことが望ましいですが、処理系は制限の違反を報告することが推奨されます。

例えばプログラムを実行するのに十分な記憶領域がない場合、あるいは数値計算の結果がその処理系では表現できないほど大きい正確な数値の場合、処理系は処理系の制限の違反を報告しても構いません。

式の値が「規定されていません」と述べられている場合、その式の結果はエラーを通知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りませんが、その値は処理系依存です。この報告書ではどのような値が返されるか明示的に述べません。

最後に、「しなければなりません」、「してはなりません」、「するべきです」、「するべきではありません」、「しても構いません」、「要求されます」、「推奨されます」、「オプショナルな」といった語句や用語は RFC 2119 [3] で述べられているように解釈されるものとします。これらはプログラマーやプログラムの動作についてのリファレンスではなく、処理系の作成者や処理系の動作のためのリファレンスとしてのみ用いられるものです。

1.3.3. 項目の書式

4 章および 6 章は項目に編成されています。それぞれの項目はひとつの言語機能または関連する機能のグループについて記述されています。それぞれの機能は構文または手続きのどちらかです。項目はひとつ以上の見出し行の形で始まります。

template *category*

この場合は base ライブラリの識別子です。

template *name*ライブラリの *category*

この *name* は付録 A で定義されているライブラリの短縮名です。

category が「構文」の場合、その項目は式型について記述しており、*template* はその式型の構文を表しています。式の各部分は構文変数で示されています。構文変数は ⟨expression⟩ や ⟨variable⟩ のように山括弧を用いて書かれています。構文変数の意図はプログラムテキストの断片を表すことです。例えば ⟨expression⟩ は構文的に有効な式となる任意の文字の列を表しています。

⟨thing$_1$⟩ ...

この記法はゼロ個以上の ⟨thing⟩ を表します。

⟨thing$_1$⟩ ⟨thing$_2$⟩ ...

この記法は 1 個以上の ⟨thing⟩ を表します。

category が「補助構文」の場合、その項目は特定の式の一部としてのみ現れる構文束縛について記述しています。独立した構文として使用したり変数として使用することはエラーです。

category が「手続き」の場合、その項目は手続きについて記述しており、見出し行はその手続きを呼び出すためのテンプレートを表しています。テンプレート内の引数名は斜体で示されています。

`(vector-ref` *vector k*`)` 手続き

この見出し行は `vector-ref` 変数に束縛されている手続きが 2 つの引数、ベクタ *vector* および正確な非負の整数 *k* を取ることを表しています (後述)。

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

この見出し行は `make-vector` 手続きが 1 つまたは 2 つの引数を取るように定義されていなければならないことを表しています。

扱えると規定されていない引数に手続きを適用することはエラーです。正確性を保つため以下の規約に従うものとします。引数の名前が 3.2 節の一覧にある型名の場合、その引数がその名前の型でなければエラーです。例えば前述の `vector-ref` の見出し行は `vector-ref` の第 1 引数がベクタ

でなければならないことを表しています。また以下の命名規約も型の制限を暗黙に示します。

alist	連想リスト (ペアのリスト)
boolean	ブーリアン値 (`#t` または `#f`)
byte	0 以上 256 未満の正確な整数
bytevector	バイトベクタ
char	文字
end	正確な非負の整数
k, k_1, ... k_j, ...	正確な非負の整数
letter	アルファベットの文字
list, $list_1$, ... $list_j$, ...	リスト (6.4 節を参照)
n, n_1, ... n_j, ...	整数
obj	任意のオブジェクト
pair	ペア
port	ポート
proc	手続き
q, q_1, ... q_j, ...	有理数
start	正確な非負の整数
string	文字列
symbol	シンボル
thunk	引数を取らない手続き
vector	ベクタ
x, x_1, ... x_j, ...	実数
y, y_1, ... y_j, ...	実数
z, z_1, ... z_j, ...	複素数

文字列、ベクタ、バイトベクタのインデックスとして *start* および *end* という名前が使われることがあります。これは以下の事柄を暗黙に示しています。

- *start*が *end*より大きい場合はエラーです。
- *end*がその文字列、ベクタ、バイトベクタの長さより大きい場合はエラーです。
- *start*が省略された場合、ゼロであるとみなされます。
- *end*が省略された場合、その文字列、ベクタ、バイトベクタの長さであるとみなされます。
- インデックス *start*は含まれます。インデックス *end*は含まれません。文字列を例に取ると、*start*と *end*が等しい場合は空文字列を表し、*start*がゼロで *end*が文字列の長さと等しい場合はその文字列全体を表します。

1.3.4. 評価の例

プログラムの例で使われている「⟹」は「〜に評価される」と解釈します。例えば

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

上記の例は式 `(* 5 8)` がオブジェクト `40` に評価されるという意味です。あるいはより正確に言うと文字の並び "`(* 5 8)`" で表される式は初期状態の環境では文字の並び "`40`" で表すことのできるオブジェクトに評価されるということです。オブジェクトの外部表現についての議論は 3.3 節を参照してください。

1.3.5. 命名規約

規約では、必ずブーリアン値を返す手続きの名前は最後の文字が `?` です。そういった手続きは**述語**と呼ばれます。一般的に述語には副作用が無いというのが暗黙の了解です。ただし間違った型の引数を渡したときに例外が発生することはあります。

同様に、すでに割り当て済みの場所 (3.4 節を参照) に値を代入する手続きの名前は最後の文字が `!` です。そういった手続きは**変更手続き**と呼ばれます。変更手続きの戻り値は規定されていません。

規約では、ある型のオブジェクトを受け取り別な型の似たオブジェクトを返す手続きは名前の間に "`->`" が入ります。例えば `list->vector` はリストを受け取り、そのリストと同じ要素を持つベクタを返します。

役に立つ値を返さない手続きは**命令**と呼ばれます。

引数を受け取らない手続きは**サンク**と呼ばれます。

2. 字句規約

この節では Scheme のプログラムを書くために用いる字句規約の一部について非形式的な説明を述べます。Scheme の形式構文は 7.1 節を参照してください。

2.1. 識別子

識別子は文字、数字、「拡張識別子文字」の任意の並びです。ただし有効な数値で始まっていてはいけません。またリスト構文で使う `.` (ピリオドひとつ) は識別子ではありません。

すべての Scheme 処理系は以下の拡張識別子文字をサポートしなければなりません。

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

代わりに垂直線 (|) で囲ったゼロ個以上の文字の並びで識別子を表すこともできます。これは文字列リテラルの記法のシンボル版です。この記法ではバックスラッシュと垂直線以外のホワイトスペースを含む任意の文字を直接書くことができます。さらに ⟨inline hex escape⟩ または文字列内で利用可能なものと同じエスケープシーケンスを使って文字を指定することもできます。

例えば識別子 `|H\x65;llo|` は `Hello` と同じ識別子です。また識別子 `|\x3BB;|` は識別子 λ と同じです (この Unicode 文字をサポートしている処理系の場合)。さらに言えば `|\t\t|` と `|\x9;\x9;|` も同じです。ちなみに `||` も有効な識別子で、これは他のいかなる識別子とも異なります。

いくつか識別子の例を挙げます。

```
...                                 +
+soup+                              <=?
->string                            a34kTMNs
lambda                              list->vector
q                                   V17a
|two words|                         |two\x20;words|
the-word-recursion-has-many-meanings
```

識別子の形式構文は 7.1.1 節を参照してください。

Scheme のプログラムでは識別子に 2 つの用途があります。

- あらゆる識別子は変数としてあるいは構文キーワードとして使うことができます (3.1 節および 4.3 節を参照)。
- 識別子がリテラル (4.1.2 節を参照) としてあるいはリテラル内に現れることで**シンボル** (6.5 節を参照) を表します。

以前のバージョンの報告書 [20] と異なり、識別子および文字の名前で使われる大文字小文字は区別されます。しかし識別子、文字、文字列の構文の中の ⟨inline hex escape⟩ や数値では大文字小文字は区別されません。この報告書で定義されている識別子に大文字を含むものはありません。

以下の指令で大文字小文字の区別を明示的に制御できます。

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

これらの指令はコメントを書ける場所 (2.2 節を参照) ならどこにでも書くことができます。指令の後にはデリミタを置かなければなりません。指令はコメントとして扱われます。ただし同じポートから読み込む後続のデータに影響を与えます。`#!fold-case` 指令は後続の識別子と文字名の大文字小文字を区別しないようにします。これは `string-foldcase`(6.7 節を参照) を適用したかのように扱われます。文字リテラルには影響しません。`#!no-fold-case` 指令は大文字小文字を区別するデフォルトの動作に戻します。

2.2. ホワイトスペースとコメント

ホワイトスペースには空白、タブ、改行といった文字が含まれます。(処理系は改ページのような追加のホワイトスペース文字を提供しても構いません。) ホワイトスペースは可読性を上げるため、および必要に応じてトークン同士を分離するために使われますが、それ以外には意味の無いものです (トークンとは識別子や数値のような分割できない字句単位のことです)。ホワイトスペースは 2 つのトークンの間に置くことはできますが、ひとつのトークンの途中に置くことはできません。文字列の中や垂直線で区切られたシンボルの中に現れるホワイトスペースは意味のあるものです。

字句構文にはコメントの形式がいくつかあります。コメントはホワイトスペースとまったく同様に扱われます。

セミコロン (`;`) は一行コメントの開始を表します。 このコメントはセミコロンが現れた行の終わりまで続きます。

コメントを表すもうひとつの方法は ⟨datum⟩ (7.1.2 節を参照) の前に `#;` を付けることです。オプショナルな ⟨whitespace⟩ を挟むこともできます。このコメントはコメント接頭辞 `#;`、空白、そして ⟨datum⟩ から成ります。この記法はコードの一部を「コメントアウト」するのに便利です。

ブロックコメントは `#|` および `|#` の組で表します。これはネストできます。

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

2.3. その他の表記

数値に使われる記法の説明は 6.2 節を参照してください。

`.` `+` `-` これらは数値に使われますが、識別子の中の任意の場所でも使うことができます。単独のプラス記号およびマイナス記号もまた識別子です。単独のピリオド (数値や識別子の中に現れたものでない) はペアの表記に使われます (6.4 節)。また仮引数リストの残余引数を表すためにも使われます (4.1.4 節)。ちなみに 2 個以上のピリオドの並びは識別子です。

`(` `)` 括弧はグループ化のためやリストを表すために使われます (6.4 節)。

`'` アポストロフィ (シングルクォート) 文字はリテラルデータを表すために使われます (4.1.2 節)。

`` ` `` グレーブアクセント (バッククォート) 文字は部分的に定数であるデータを表すために使われます (4.2.8 節)。

`,` `,@` コンマおよびコンマ・アットマークの並びは quasiquote の中で使われます (4.2.8 節)。

`"` 引用符は文字列を区切るために使われます (6.7 節)。

`\` バックスラッシュは文字定数 (6.6 節) の構文で使われます。また文字列定数 (6.7 節) および識別子 (7.1.1 節) の中でエスケープ文字としても使われます。

`[` `]` `{` `}` 角括弧および波括弧は将来の言語拡張の可能性のために予約されています。

`#` 井桁は直後の文字によって様々な目的に使われます。

`#t` `#f` これらはブーリアン定数です (6.3 節)。さらに `#true` および `#false` も同様です。

`#\` これは文字定数の始まりです (6.6 節)。

`#(` これはベクタ定数の始まりです (6.8 節)。ベクタ定数は `)` で終わります。

`#u8(` これはバイトベクタ定数の始まりです (6.9 節)。バイトベクタ定数は `)` で終わります。

`#e` `#i` `#b` `#o` `#d` `#x` これらは数値を表すために使われます (6.2.5 節)。

`#`⟨n⟩`=` `#`⟨n⟩`#` これらは他のリテラルデータの参照やラベル付けに使われます (2.4 節)。

2.4. データムラベル

#⟨n⟩=⟨datum⟩ 字句構文
#⟨n⟩# 字句構文

字句構文 #⟨n⟩=⟨datum⟩ は ⟨datum⟩ と同様に解釈されますが、その結果の ⟨datum⟩ に ⟨n⟩ でラベルを付けます。⟨n⟩ が数字の並びでない場合はエラーです。

字句構文 #⟨n⟩# は #⟨n⟩= でラベルを付けたオブジェクトへの参照として機能します。#⟨n⟩= と同じオブジェクトが結果となります (6.1 節を参照)。

これらの構文を使うと部分的な共有構造や循環構造を表すことができます。

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

データムラベルのスコープはそれが現れた最も外側のデータムのうちそのラベルより右側の部分です。従って参照 #⟨n⟩# はラベル #⟨n⟩= より後にだけ現れることができます。前方参照を試みることはエラーです。またラベルを付けたオブジェクトそれ自身として参照が現れた場合はエラーです。例えば #⟨n⟩= #⟨n⟩# のような場合です。この場合 #⟨n⟩= がラベルを付けているオブジェクトが何であるか不明なためです。

⟨program⟩ または ⟨library⟩ にリテラル以外の循環参照がある場合はエラーです。特に quasiquote(4.2.8 節) に循環参照がある場合はエラーです。

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

3. 基本概念

3.1. 変数、構文キーワード、有効範囲

識別子は構文の種類または値が格納される場所に名前を付けるために使われます。構文の種類に名前を付けている識別子は**構文キーワード**と呼ばれ、その構文の変換子に**束縛されている**と言います。場所に名前を付けている識別子は**変数**と呼ばれ、その場所に**束縛されている**と言います。プログラム内のある地点で見える有効な束縛すべての集合はその地点での有効な**環境**と呼ばれます。ある変数が束縛されている場所に格納されている値はその変数の値と呼ばれます。しばしば用語の誤った使い方により、変数が値に名前を付けているだとか、変数が値に束縛されているだとか言われることがあります。これらはあまり正確ではありませんが、この慣習が混乱を招くことはほとんどないでしょう。

新しい種類の構文を作成し、その構文に構文キーワードを束縛するには、ある特定の式型を使います。新しい場所を作成し、その場所に変数を束縛するには、また別の式型を使います。これらの式型は**束縛構文**と呼ばれます。構文キーワードを束縛する式型は 4.3 節に記載されています。最も基礎的な変数の束縛構文は lambda 式です。他の変数束縛構文はすべて lambda 式を使って説明できます。他の変数束縛構文には let、let*、letrec、letrec*、let-values、let*-values および do 式があります (4.1.4 節、4.2.2 節および 4.2.4 節を参照)。

Scheme はブロック構造を持つ言語です。プログラム内で束縛した各々の識別子にはその束縛が見えるプログラムテキストの**有効範囲**が対応しています。有効範囲はその束縛を確立した束縛構文の種類によって決まります。例えば lambda 式によって確立した束縛の場合、その有効範囲はその lambda 式全体です。識別子の使用はその識別子の束縛のうちその使用場所を含んでいる最も内側の有効範囲を確立したものを参照します。その使用場所を含んでいる有効範囲を持つその識別子の束縛がない場合、大域環境の変数に対する束縛があればそれを参照します。その識別子に対する束縛がなければ、その識別子は**束縛されていない**と言います。

3.2. 型の独立性

どのようなオブジェクトも以下の述語を 2 つ以上満たすことはありません。

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

define-record-type で作成されたすべての述語

これらの述語により型**ブーリアン**、**バイトベクタ**、**文字**、空リストオブジェクト、*EOF* **オブジェクト**、**数値**、**ペア**、**ポート**、**手続き**、**文字列**、**シンボル**、**ベクタ**、およびすべてのレコード型が定義されます。

独立したブーリアン型が存在してはいますが、条件判定目的においてはどのような Scheme の値もブーリアン値として使うことができます。6.3 で説明しているように条件判定では #f 以外のすべての値が真とみなされます。この報告書では「真」という言葉は #f 以外のすべての Scheme の値を表し、「偽」という言葉は #f を表します。

3.3. 外部表現

Scheme (および Lisp) における重要な概念にオブジェクトの**外部表現**があります。外部表現はそのオブジェクトを表す文字の並びです。例えば整数 28 の外部表現は “28” という文字の並びです。また整数 8 と整数 13 から成るリストの外部表現は “(8 13)” という文字の並びです。

オブジェクトの外部表現は唯一であるとは限りません。整数 28 は “#e28.000” や “#x1c” のようにも表せますし、前段落のリストは “(08 13)” や “(8 . (13 . ()))” のようにも表現できます (6.4 節を参照)。

多くのオブジェクトには標準の外部表現がありますが、手続きのように標準の外部表現がないオブジェクトもあります

(処理系はそういったオブジェクトの外部表現を定義しても構いません)。

外部表現はプログラム内で対応するオブジェクトを得るために使うことができます (4.1.2 節、quote を参照)。

外部表現は入出力のために使うこともできます。手続き read (6.13.2 節) は外部表現をパースし、手続き write (6.13.3 節) は外部表現を生成します。これらはエレガントでパワフルな入出力機能です。

"(+ 2 6)" という文字の並びはシンボル +、整数 2、整数 6 から成る 3 要素のリストの外部表現であることに注意してください。これは整数 8 に評価される式ではありますが整数 8 の外部表現ではありません。Scheme の構文には、式を表す文字の並びが何らかのオブジェクトの外部表現でもある、という特徴があります。これは混乱を招くこともあります。ある文字の並びがデータを表しているのかプログラムを表しているのか文脈なしには判断できない場合があるためです。しかしこれはパワーの源でもあります。インタプリタやコンパイラのようなプログラムをデータとして扱う (またはその逆の) プログラムを書くのが簡単になります。

様々な種類のオブジェクトの外部表現の構文は 6 章の各節にそのオブジェクトを扱うプリミティブの記述と一緒に記載されています。

3.4. 記憶領域のモデル

ペア、文字列、ベクタ、バイトベクタといったオブジェクトや変数は、場所または場所の並びを暗黙に指します。例えば文字列はその文字列の長さと同じだけの数の場所を指します。string-set! 手続きを使うとこれらの場所のひとつに新しい値を格納できます。しかしその文字列の指す場所が以前と異なる場所になるわけではありません。

変数参照や car、vector-ref、string-ref といった手続きによってある場所から取得したオブジェクトはその場所に最後に格納されたオブジェクトと eqv? の意味で同じです (6.1 節)。

すべての場所には使用中かどうかを示す印が付けられています。変数やオブジェクトが使用中でない場所を参照することはありません。

変数やオブジェクトに対して記憶領域が新たに割り当てられると言うとき、それは使用中でない場所から適切な数の場所を選び、使用中であることを示す印をその場所に付け、そしてその変数またはオブジェクトがその場所を指すようにする、ということを意味しています。これと異なり、空リストは新たに割り当てることはできないと考えられます。唯一のオブジェクトであるためです。場所を持たない空文字列、空ベクタ、空バイトベクタは新たに割り当てることができてもできなくても構いません。

場所を指すすべてのオブジェクトは可変または不変のいずれかです。リテラル定数および symbol->string から返される文字列は不変オブジェクトです。scheme-report-environment から返される環境は不変の場合があります。この報告書に掲載されているそれ以外の手続きから返されるオブジェクトはすべて可変です。不変オブジェクトの指す場所に新しい値を格納しようとすることはエラーです。

これらの場所は概念的なものであり、物理的なものではないと解釈されます。つまりメモリアドレスに対応付けられている必要はなく、対応付けられている場合でもそのメモリアドレスが固定されている必要はありません。

論拠: 多くのシステムにおいて、定数 (例えばリテラル式の値) は読み込み専用メモリに置くのが好ましいと考えられます。定数の変更をエラーとすることにより他のシステムに可変オブジェクトと不変オブジェクトの区別を要求することなくそういった実装戦略を取ることができるようになります。

3.5. 真正末尾再帰

Scheme の処理系は**真正末尾再帰**であることが要求されます。後述する特定の構文文脈で行われる手続き呼び出しを**末尾呼び出し**と呼びます。真正末尾再帰であるとは、アクティブな末尾呼び出しの数に制限が無いということです。呼び出しが**アクティブ**であるとは、その呼び出された手続きがまだ戻っていないということです。ちなみに後に呼び出される現在の継続または call-with-current-continuation によって予め捕捉した継続によって戻る呼び出しもこれに含まれます。もし継続が捕捉されていなければ、すべての呼び出しは多くとも 1 回だけ戻ることができ、アクティブな呼び出しとは、そのうちまだ戻っていない呼び出しのことになります。真正末尾再帰の形式的な定義は [6] にあります。

論拠:

直感的に言って、アクティブな末尾呼び出しは空間を消費しません。末尾呼び出しで使われる継続はその呼び出しを含む手続きに渡される継続と同じ意味論を持つためです。不正な処理系では、その呼び出しに新しい継続を使用し、この新しい継続へ戻った直後にその手続きに渡された継続へ戻るかもしれません。真正末尾再帰な処理系は、その継続に直接戻ります。

真正末尾再帰は Steele と Sussman によるオリジナルバージョンの Scheme の中核となるアイディアのひとつです。彼らの最初の Scheme インタプリタには関数とアクタの両方が実装されていました。制御の流れはアクタで表現されていました。関数は呼び出し元に結果を返しますが、アクタはそれと異なり結果を他のアクタに渡すものでした。この節の用語で言うとそれぞれのアクタが終了するときは他のアクタに末尾呼び出しをしていました。

Steele と Sussman は後に彼らのインタプリタ内のアクタを処理するコードと関数を処理するコードが同一のものであることに気付きました。つまり言語に両方を含める必要は無かったのです。

末尾呼び出しは**末尾文脈**で行われる手続き呼び出しです。末尾文脈は帰納的に定義されます。末尾文脈は常に特定のラムダ式について決定されることに注意してください。

- 以下の ⟨tail expression⟩ で示されるように、ラムダ式の本体の最後の式は末尾文脈です。case-lambda 式のすべての本体についても同様です。

(lambda ⟨formals⟩
  ⟨definition⟩* ⟨expression⟩* ⟨tail expression⟩)

(case-lambda (⟨formals⟩ ⟨tail body⟩)*)

- 以下の式のいずれにおいても、それが末尾文脈にあれば ⟨tail expression⟩ で示される部分式は末尾文脈です。これらは 7 章に記載されている文法規則から派生したもので、⟨body⟩ のいくつかを ⟨tail body⟩ に、⟨expression⟩ のいくつかを ⟨tail expression⟩ に、⟨sequence⟩ のいくつかを ⟨tail sequence⟩ に置き換えています。ここに示されているのはそれらの規則のうち末尾文脈を含むもののみです。

(if ⟨expression⟩ ⟨tail expression⟩ ⟨tail expression⟩)
(if ⟨expression⟩ ⟨tail expression⟩)

(cond ⟨cond clause⟩+)
(cond ⟨cond clause⟩* (else ⟨tail sequence⟩))

(case ⟨expression⟩
  ⟨case clause⟩+)
(case ⟨expression⟩
  ⟨case clause⟩*
  (else ⟨tail sequence⟩))

(and ⟨expression⟩* ⟨tail expression⟩)
(or ⟨expression⟩* ⟨tail expression⟩)

(when ⟨test⟩ ⟨tail sequence⟩)
(unless ⟨test⟩ ⟨tail sequence⟩)

(let (⟨binding spec⟩*) ⟨tail body⟩)
(let ⟨variable⟩ (⟨binding spec⟩*) ⟨tail body⟩)
(let* (⟨binding spec⟩*) ⟨tail body⟩)
(letrec (⟨binding spec⟩*) ⟨tail body⟩)
(letrec* (⟨binding spec⟩*) ⟨tail body⟩)
(let-values (⟨mv binding spec⟩*) ⟨tail body⟩)
(let*-values (⟨mv binding spec⟩*) ⟨tail body⟩)

(let-syntax (⟨syntax spec⟩*) ⟨tail body⟩)
(letrec-syntax (⟨syntax spec⟩*) ⟨tail body⟩)

(begin ⟨tail sequence⟩)

(do (⟨iteration spec⟩*)
    (⟨test⟩ ⟨tail sequence⟩)
  ⟨expression⟩*)

ただし

⟨cond clause⟩ ⟶ (⟨test⟩ ⟨tail sequence⟩)
⟨case clause⟩ ⟶ ((⟨datum⟩*) ⟨tail sequence⟩)

⟨tail body⟩ ⟶ ⟨definition⟩* ⟨tail sequence⟩
⟨tail sequence⟩ ⟶ ⟨expression⟩* ⟨tail expression⟩

- cond 式または case 式が末尾文脈であり、それが (⟨expression$_1$⟩ => ⟨expression$_2$⟩) 形式の節を持っている場合、⟨expression$_2$⟩ の評価結果である手続きの (暗黙の) 呼び出しは末尾文脈です。⟨expression$_2$⟩ 自身は末尾文脈ではありません。

この報告書で定義されている一部の手続きも末尾呼び出しを行うことが要求されます。apply の第 1 引数、call-with-current-continuation の第 1 引数および call-with-values の第 2 引数は末尾呼び出しで呼び出されなければなりません。同様に eval の第 1 引数は eval 内の末尾位置で呼び出されたかように評価されなければなりません。

以下の例では f の呼び出しはすべて末尾呼び出しです。g および h の呼び出しは末尾呼び出しではありません。x の参照は末尾文脈ですが、呼び出しではないので末尾呼び出しではありません。

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

メモ: 処理系は上記の例の h のような末尾呼び出しでない呼び出しの一部を末尾呼び出しとして評価可能であると認識しても構いません。上記の例では let 式を h の末尾呼び出しとしてコンパイルすることができます。(h が多値を返す可能性は無視できます。その場合 let の効果は明示的に規定されていないので処理系依存です。)

4. 式

式型はプリミティブまたは派生に分類されます。プリミティブ式型は変数や手続き呼び出しなどです。派生式型は意味論上プリミティブでなく、マクロで定義できるものを言います。7.3 節にいくつかの派生式型の適切な構文定義が掲載されています。

delay、delay-force および parameterize 式型と密接に関連している手続き force、promise?、make-promise および make-parameter もこの章で説明しています。

4.1. プリミティブ式型

4.1.1. 変数参照

⟨variable⟩ 構文

変数 (3.1 節) から成る式は変数参照です。変数参照の値はその変数が束縛されている場所に格納されている値です。束縛されていない変数を参照することはエラーです。

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

4.1.2. リテラル式

(quote ⟨datum⟩) 構文
’⟨datum⟩ 構文
⟨constant⟩ 構文

(quote ⟨datum⟩) は ⟨datum⟩ に評価されます。⟨datum⟩ には任意の Scheme のオブジェクトの外部表現 (3.3 節を参照) を指定できます。Scheme のコードにリテラル定数を含めるためにこの記法を使います。

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

(quote ⟨datum⟩) は省略して ’⟨datum⟩ と書くことができます。この 2 種類の記法はあらゆる点で同等です。

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

数値定数、文字列定数、文字定数、ベクタ定数、バイトベクタ定数、ブーリアン定数はそれ自身に評価されます。quote する必要はありません。

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

3.4 節で述べたように set-car! や string-set! のような変更手続きを使用して定数 (すなわちリテラル式の値) の変更を試みることはエラーです。

4.1.3. 手続き呼び出し

(⟨operator⟩ ⟨operand$_1$⟩ ...) 構文

手続き呼び出しは、呼び出す手続きの式の後に渡す引数の式を並べ、括弧で囲って書きます。演算子と被演算子の式が評価され (順番は規定されていません)、結果の手続きに結果の引数が渡されます。

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

この文書に掲載されている手続きは標準ライブラリからエクスポートされている変数の値として利用可能です。例えば上記の例の加算手続きと乗算手続きは base ライブラリの変数 + および * の値です。lambda 式 (4.1.4 節を参照) を評価することで新しい手続きを作ることができます。

手続き呼び出しは任意の個数の値を返すことができます (6.10 節の values を参照)。この報告書で定義されている手続きのほとんどは単一の値を返します。apply のような手続きの場合、引数に渡した手続きを呼び出した戻り値がそのまま返されます。例外はそれぞれ個別の説明に記載されています。

メモ: 他の Lisp 方言と異なり評価順は規定されていません。また演算子の式と被演算子の式は常に同じ評価規則で評価されます。

メモ: 評価順は規定されていませんが、演算子と被演算子の式を並列的に評価する場合は何らかの逐次的な評価順と一貫性を持たなければならないという制約があります。評価順は手続き呼び出しのたびに異なっていても構いません。

メモ: 多くの Lisp 方言では空リスト () は自分自身に評価される正当な式です。Scheme ではエラーです。

4.1.4. 手続き

(lambda ⟨formals⟩ ⟨body⟩) 構文

構文: ⟨formals⟩ は後述の仮引数リストです。⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。

意味論: lambda 式は手続きに評価されます。lambda 式が評価されたときの有効な環境が手続きの一部として記憶されます。後ほど手続きがいくつかの実引数を伴って呼び出されると、lambda 式が評価されたときの環境が仮引数リストの変数に新しい場所を束縛することによって拡張され、対応する実引数の値がそれらの場所に格納されます。(**新しい場所**とはそれまで存在していたどの場所とも異なる場所のことです。) 次にラムダ式の本体の式 (定義があれば letrec* の形で表されます — 4.2.2 節を参照) がその拡張された環境で逐次的に評価されます。その本体の最後の式の結果がその手続き呼び出しの結果として返されます。

```
(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
```

⟨formals⟩ は以下のいずれかの形です。

- (⟨variable$_1$⟩ ...): 手続きは固定の個数の引数を取ります。手続きが呼ばれると引数が対応する変数に束縛されている新しい場所に格納されます。
- ⟨variable⟩: 手続きは任意の個数の引数を取ります。手続きが呼ばれると、実引数の並びが新しく割り当てられたリストに変換され、そのリストが ⟨variable⟩ に束縛されている新しい場所に格納されます。

- (⟨variable$_1$⟩ ... ⟨variable$_n$⟩ . ⟨variable$_{n+1}$⟩): 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、その手続きは n 個以上の引数を取ります。ただし n はピリオドの前の仮引数の数です (最低ひとつ以上なければエラーです)。最後の変数の束縛に格納される値は他の仮引数に対して実引数を一致させた後に残った実引数の新たに割り当てられたリストになります。

⟨variable⟩ が ⟨formals⟩ に 2 回以上現れる場合はエラーです。

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

lambda 式 を評価した結果作成された手続きはそれぞれ (概念的に) ある記憶領域の位置に紐付けられます。それにより手続きに対して eqv? および eq? を適用することができます (6.1 節を参照)。

4.1.5. 条件判定

(if ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩) 構文
(if ⟨test⟩ ⟨consequent⟩) 構文

構文: ⟨test⟩、⟨consequent⟩ および ⟨alternate⟩ は式です。

意味論: if 式は以下のように評価されます。まず ⟨test⟩ が評価されます。その結果が真の値 (6.3 節を参照) であった場合、⟨consequent⟩ が評価され、その値が返されます。そうでなければ ⟨alternate⟩ が評価され、その値が返されます。⟨test⟩ の結果が偽の値であり、⟨alternate⟩ が指定されていない場合、式の結果は規定されていません。

```
(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! ⟨variable⟩ ⟨expression⟩) 構文

意味論: ⟨expression⟩ が評価され、その結果の値が ⟨variable⟩ の束縛されている場所に格納されます。⟨variable⟩ が set! 式を囲むいずれの有効範囲にも大域的にも束縛されていなければエラーです。set! 式の結果は規定されていません。

```
(define x 2)
(+ x 1)                     ⟹  3
(set! x 4)                  ⟹  規定されていない
(+ x 1)                     ⟹  5
```

4.1.7. インクルード

(include ⟨string$_1$⟩ ⟨string$_2$⟩ ...) 構文
(include-ci ⟨string$_1$⟩ ⟨string$_2$⟩ ...) 構文

意味論: include および include-ci は両方とも文字列定数として表されたひとつ以上のファイル名を取り、処理系固有のアルゴリズムを適用して対応するファイルを検索し、read を繰り返し適用したかのように順番にそのファイルの内容を読み込み、そのファイルから読み込んだ内容を含む begin 式でその include 式または include-ci 式を実質的に置換します。この 2 つの違いは以下のようなものです。include-ci は各ファイルの先頭に #!fold-case 指令があるかのように読み込みます。include はそのようなことをしません。

メモ: 処理系はインクルードする側のファイルがあるディレクトリでファイルを検索することが推奨されます。またユーザが他の検索ディレクトリを指定するための方法を提供することが推奨されます。

4.2. 派生式型

この節の構文は 4.3 節で述べられてるように衛生的です。リファレンス目的のためこの節で説明している構文のほとんどを前の節で説明したプリミティブ構文に変換する構文定義が 7.3 節に掲載されています。

4.2.1. 条件判定

(cond ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...) 構文
else 補助構文
=> 補助構文

構文: ⟨clause⟩ は以下のいずれかの形をひとつ以上取ります。

(⟨test⟩ ⟨expression$_1$⟩ ...)
(⟨test⟩ => ⟨expression⟩)

ただし ⟨test⟩ は任意の式です。

最後の ⟨clause⟩ は以下のような「else 節」にすることもできます。

(else ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...)

意味論: cond 式は以下のように評価されます。まず真の値 (6.3 節を参照) に評価されるまで一連の ⟨clause⟩ の ⟨test⟩ 式が順番に評価されます。⟨test⟩ が真の値に評価されると、その ⟨clause⟩ の中の残りの ⟨expression⟩ が順に評価され、その ⟨clause⟩ の中の最後の ⟨expression⟩ の結果が cond 式全体の結果として返されます。

選択された ⟨clause⟩ が ⟨test⟩ だけで ⟨expression⟩ を持たない場合、⟨test⟩ の値が結果として返されます。選択された ⟨clause⟩ が => 代理形を使っている場合、まず ⟨expression⟩ が評価されます。その値が引数をひとつ受け取る手続きでなければエラーです。⟨test⟩ の値に対してその手続きが呼び出され、その手続きが返した値が cond 式から返されます。

すべての ⟨test⟩ が `#f` に評価され、else 節がない場合、`cond` 式の結果は規定されていません。else 節があれば、その ⟨expression⟩ が順に評価され、その最後の値が返されます。

```
(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` ⟨key⟩ ⟨clause$_1$⟩ ⟨clause$_2$⟩ ...`)` 構文

構文: ⟨key⟩ は任意の式を指定できます。それぞれの ⟨clause⟩ は以下の形です。

`((`⟨datum$_1$⟩ ...`)` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)`

ただしそれぞれの ⟨datum⟩ は何らかのオブジェクトの外部表現です。式の中に同じ ⟨datum⟩ がふたつ以上ある場合はエラーです。また ⟨clause⟩ には以下の形も指定できます。

`((`⟨datum$_1$⟩ ...`) =>` ⟨expression⟩`)`

最後の ⟨clause⟩ は以下のいずれかの形を持つ「else 節」にできます。

`(else` ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)`
`(else =>` ⟨expression⟩`)`

意味論: `case` 式は以下のように評価されます。⟨key⟩ が評価され、その結果が各々の ⟨datum⟩ と比較されます。⟨key⟩ の評価結果が ⟨datum⟩ と等しい (`eqv?` の意味で; 6.1 節を参照) 場合、対応する ⟨clause⟩ 内の式が順番に評価され、その ⟨clause⟩ の最後の式の結果が `case` 式の結果として返されます。

⟨key⟩ の評価結果がどの ⟨datum⟩ とも異なる場合、else 節があればその式が評価され、その最後の結果が `case` 式の結果になります。そうでなければ `case` 式の結果は規定されていません。

選択された ⟨clause⟩ または else 節が `=>` 代理形を使っている場合、まず ⟨expression⟩ が評価されます。その値がひとつの引数を受け取る手続きでなければエラーです。⟨key⟩ の値に対してその手続きが呼び出され、その手続きが返した値が `case` 式から返されます。

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

`(and` ⟨test$_1$⟩ ...`)` 構文

意味論: ⟨test⟩ 式が左から右に評価され、いずれかの式が `#f` に評価されるとそこで `#f` が返されます。残りの式は評価されません。すべての式が真の値に評価された場合、その最後の式の値が返されます。式がひとつも無ければ `#t` を返します。

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

`(or` ⟨test$_1$⟩ ...`)` 構文

意味論: ⟨test⟩ 式が左から右に評価され、真の値 (6.3 節を参照) に評価された最初の式の値が返されます。残りの式は評価されません。すべての式が `#f` に評価された場合または式がひとつもない場合は `#f` を返します。

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

`(when` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` 構文

構文: ⟨test⟩ は式です。

意味論: ⟨test⟩ が評価され、それが真の値に評価された場合、⟨expression⟩ が順番に評価されます。`when` 式の戻り値は規定されていません。

```
(when (= 1 1.0)
  (display "1")
  (display "2"))            ⟹  規定されていない
          and prints  12
```

`(unless` ⟨test⟩ ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...`)` 構文

構文: ⟨test⟩ は式です。

意味論: ⟨test⟩ が評価され、それが `#f` に評価された場合、⟨expression⟩ が順番に評価されます。`unless` 式の戻り値は規定されていません。

```
(unless (= 1 1.0)
  (display "1")
  (display "2"))            ⟹  規定されていない
          そして何も表示されない
```

`(cond-expand` ⟨ce-clause$_1$⟩ ⟨ce-clause$_2$⟩ ...`)` 構文

構文: `cond-expand` 式型は処理系に依存して異なる式に静的に展開される方法を提供します。⟨ce-clause⟩ 節は以下の形を取ります。

`(`⟨feature requirement⟩ ⟨expression⟩ ...`)`

最後の節は以下の形の「else 節」にできます。

`(else` ⟨expression⟩ ...`)`

⟨feature requirement⟩ は以下の形のいずれかひとつを取ります。

- ⟨feature identifier⟩
- (library ⟨library name⟩)
- (and ⟨feature requirement⟩ ...)
- (or ⟨feature requirement⟩ ...)
- (not ⟨feature requirement⟩)

意味論: 各処理系はインポート可能なライブラリのリストおよび存在する機能識別子のリストを管理しています。それぞれの ⟨feature identifier⟩ および (library ⟨library name⟩) を、処理系の持つリストにある場合は #t、ない場合は #f に置き換え、その結果の式を and、or、not の通常の解釈の下で Scheme のブーリアン式として評価することによって、⟨feature requirement⟩ の値が決定されます。

次に一連の ⟨ce-clause⟩ の ⟨feature requirement⟩ が #t を返すまで順番に評価されます。真の節が見つかれば、対応する ⟨expression⟩ が begin に展開されます。残りの節は無視されます。どの ⟨feature requirement⟩ も #t に評価されない場合、else 節があればその ⟨expression⟩ が含まれます。そうでなければ cond-expand の動作は規定されていません。cond と異なり cond-expand は何の変数の値にも依存しません。

提供されている正確な機能は処理系定義ですが、移植性のために機能の中核的なセットが付録 B に掲載されています。

4.2.2. 束縛構文

束縛構文 let、let*、letrec、letrec*、let-values、let*-values は Algol 60 のようなブロック構造を Scheme に持ち込みます。最初の 4 つは同じ構文ですが、それらが確立する変数束縛の有効範囲が異なっています。let 式ではどの変数が束縛されるよりも前に初期値が計算されます。let* 式では束縛と評価が逐次的に行われます。letrec 式および letrec* 式では初期値が計算されている間もすべての束縛が有効であり、これによって相互再帰を定義することができます。let-values 構文および let*-values 構文はそれぞれ let および let* に似ていますが、多値の式を処理できるよう設計されており、返された多値をそれぞれ異なる識別子に束縛できます。

(let ⟨bindings⟩ ⟨body⟩) 構文

構文: ⟨bindings⟩ は以下の形を取ります。

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...)

それぞれの ⟨init⟩ は式で、⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。4.1.4 節も参照してください。束縛される変数のリストに同じ ⟨variable⟩ がふたつ以上現れた場合はエラーです。

意味論: ⟨init⟩ が現在の環境で (何らかの規定されていない順番で) 評価され、その結果を格納した新しい場所に ⟨variable⟩ が束縛され、その拡張された環境で ⟨body⟩ が評価され、⟨body⟩ の最後の式の値が返されます。各 ⟨variable⟩ の束縛は ⟨body⟩ をその有効範囲として持ちます。

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

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

4.2.4 節の「名前付き let」も参照してください。

(let* ⟨bindings⟩ ⟨body⟩) 構文

構文: ⟨bindings⟩ は以下の形を取ります。

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...)

⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。4.1.4 節も参照してください。

意味論: let* 束縛構文は let と同様ですが、束縛が左から右に逐次的に行われます。(⟨variable⟩ ⟨init⟩) によって示される束縛の有効範囲は let* 式のその束縛より右側の部分になります。従って 2 番目の束縛は最初の束縛が見える環境で行われ、以下同様です。⟨variable⟩ がそれぞれ異なっている必要はありません。

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

(letrec ⟨bindings⟩ ⟨body⟩) 構文

構文: ⟨bindings⟩ は以下の形を取ります。

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...)

⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。4.1.4 節も参照してください。束縛される変数のリストに同じ ⟨variable⟩ がふたつ以上現れた場合はエラーです。

意味論: 規定されていない値を格納した新しい場所に ⟨variable⟩ が束縛され、その結果の環境で ⟨init⟩ が (何らかの規定されていない順番で) 評価され、⟨init⟩ の結果がそれぞれ対応する ⟨varible⟩ に代入され、その結果の環境で ⟨body⟩ が評価され、⟨body⟩ の最後の式の値が返されます。各 ⟨variable⟩ の束縛は letrec 式全体をその有効範囲として持ちます。それにより相互再帰手続きを定義することが可能です。

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

letrec には非常に重要な制限がひとつあります。各 ⟨init⟩ はどの ⟨variable⟩ にも代入も参照もせずに評価できなければなりません。さもなければエラーです。この制限は必要なものです。lambda 式が ⟨variable⟩ を ⟨init⟩ の値に束縛する手続き呼び出しによって letrec が定義されているためです。letrec のほとんどの用途では、⟨init⟩ は lambda 式であり、この制限は自動的に満たされます。

(letrec* ⟨bindings⟩ ⟨body⟩) 構文

構文: ⟨bindings⟩ は以下の形を取ります。

((⟨variable$_1$⟩ ⟨init$_1$⟩) ...)

⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。4.1.4 節も参照してください。束縛される変数のリストに同じ ⟨variable⟩ がふたつ以上現れた場合はエラーです。

意味論: ⟨variable⟩ が新しい場所に束縛され、左から右の順番で ⟨init⟩ が評価された結果がそれぞれ対応する ⟨variable⟩ に代入され、その結果の環境で ⟨body⟩ が評価され、⟨body⟩ の最後の式が返されます。左から右の評価代入の順番にもかかわらず、各 ⟨variable⟩ の束縛は letrec* 式全体をその有効範囲として持ちます。それにより相互再帰手続きを定義することが可能です。

各 ⟨init⟩ は対応する ⟨variable⟩ の値およびそれより右側のどの ⟨bindings⟩ の ⟨variable⟩ にも参照も代入もせず評価できなければなりません。さもなければエラーです。もうひとつ制限があります。⟨init⟩ の継続を 2 回以上呼び出すことはエラーです。

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

(let-values ⟨mv binding spec⟩ ⟨body⟩) 構文

構文: ⟨mv binding spec⟩ は以下の形を取ります。

((⟨formals$_1$⟩ ⟨init$_1$⟩) ...)

それぞれの ⟨init⟩ は式で、⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。4.1.4 も参照してください。⟨formals⟩ の集合内に同じ変数がふたつ以上現れた場合はエラーです。

意味論: ⟨init⟩ が現在の環境で (何らかの規定されていない順番で) call-with-values によって呼び出されたかのように評価され、⟨init⟩ の戻り値を格納した新しい場所に ⟨formals⟩ 内の変数が束縛されます。ただし ⟨formals⟩ は lambda 式が手続き呼び出しの際に引数を一致させるのと同じ方法でその戻り値を一致させます。その後その拡張された環境で ⟨body⟩ が評価され、⟨body⟩ の最後の式の値が返されます。それぞれの ⟨variable⟩ の束縛は ⟨body⟩ をその有効範囲とします。

⟨init⟩ の返した値の数が対応する ⟨formals⟩ に一致しない場合はエラーです。

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

(let*-values ⟨mv binding spec⟩ ⟨body⟩) 構文

構文: ⟨mv binding spec⟩ は以下の形を取ります。

((⟨formals⟩ ⟨init⟩) ...)

⟨body⟩ はゼロ個以上の定義に続くひとつ以上の式です。4.1.4 も参照してください。それぞれの ⟨formals⟩ において同じ変数がふたつ以上現れた場合はエラーです。

意味論: let*-values 構文は let-values に似ていますが、左から右に逐次的に、⟨init⟩ が評価され、束縛が作成されます。それぞれの ⟨formals⟩ の束縛の有効範囲は ⟨body⟩ だけでなくその ⟨init⟩ の右側も含まれます。従って 2 番目の ⟨init⟩ は最初の束縛の集合が見えて初期化された環境で評価され、以下同様です。

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

4.2.3. 逐次実行

Scheme の逐次実行構文は両方とも begin という名前ですが、それぞれ微妙に異なった形と用途があります。

(begin ⟨expression or definition⟩ ...) 構文

この形の begin は ⟨body⟩ の一部、⟨program⟩ の最上位、REPL、またはこの形の begin に直接ネストしている場合に使うことができます。囲んでいる begin 構文が存在しない場合とまったく同様に、内部の式および定義が評価されます。

論拠: この形は一般的に、複数の定義を生成しそれらを展開先の文脈に繋ぎ合わせる必要のあるマクロの出力で使われます (4.3 節を参照)。

(begin ⟨expression$_1$⟩ ⟨expression$_2$⟩ ...) 構文

この形の begin は普通の式として使うことができます。⟨expression⟩ は左から右に逐次的に評価され、最後の ⟨expression⟩ の値が返されます。この式型は代入や入出力のような副作用を逐次実行するために使われます。

```
(define x 0)

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

(begin (display "4 plus 1 equals ")
       (display (+ 4 1)))   ⟹  規定されていない
                               そして 4 plus 1 equals 5  を表示します
```

ちなみにライブラリ宣言として使われる 3 つめの形式の begin があります。5.6.1 節を参照してください。

4.2.4. 繰り返し

(do ((⟨variable$_1$⟩ ⟨init$_1$⟩ ⟨step$_1$⟩) ...) (⟨test⟩ ⟨expression⟩ ...) ⟨command⟩ ...) 構文

構文: ⟨init⟩、⟨step⟩、⟨test⟩、⟨command⟩ はすべて式です。

意味論: do 式は繰り返し構文です。束縛する変数の集合、それらがどのように初期化されるか、また繰り返しごとにどのように更新されるかを指定します。終了条件を満たすと ⟨expression⟩ を評価したあとループを抜けます。

do 式は以下のように評価されます。⟨init⟩ 式が (何らかの規定されていない順番で) 評価され、⟨variable⟩ が新しい場所に束縛され、⟨init⟩ 式の結果がその ⟨variable⟩ の束縛に格納され、そして繰り返しフェーズが始まります。

それぞれの繰り返しの始めに ⟨test⟩ が評価されます。その結果が偽 (6.3 節を参照) であれば ⟨command⟩ 式が順番に評価され、⟨step⟩ 式が何らかの規定されていない順番で評価され、⟨variable⟩ が新しい場所に束縛され、⟨step⟩ の結果が ⟨variable⟩ の束縛に格納され、そして次の繰り返しが始まります。

⟨test⟩ が真に評価された場合は、⟨expression⟩ が左から右に評価され、最後の ⟨expression⟩ の値が返されます。⟨expression⟩ が存在しなければ do 式の値は規定されていません。

⟨variable⟩ の束縛の有効範囲は ⟨init⟩ を除く do 式全体から成ります。do の変数リストに同じ ⟨variable⟩ がふたつ以上現れた場合はエラーです。

⟨step⟩ は省略できます。その場合は (⟨variable⟩ ⟨init⟩) の代わりに (⟨variable⟩ ⟨init⟩ ⟨variable⟩) と書かれた場合と同じ効果を持ちます。

```
(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 ⟨variable⟩ ⟨bindings⟩ ⟨body⟩) 構文

意味論: 「名前付き let」は let 構文の亜種で、do よりも一般的なループ構文です。再帰を表現するために使うこともできます。普通の let と同じ構文と意味論を持ちますが、⟨variable⟩ が ⟨body⟩ 内で仮引数がその束縛された変数で本体が ⟨body⟩ である手続きに束縛される点が異なります。そのため ⟨variable⟩ という名前の手続きを呼ぶことにより ⟨body⟩ を繰り返すことができます。

```
(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 ⟨expression⟩) lazy ライブラリの構文

意味論: delay 構文は手続き force と共に**遅延評価**または**必要渡し**を実装するために使われます。(delay ⟨expression⟩) は**プロミス**と呼ばれるオブジェクトを返します。これは ⟨expression⟩ を評価してその戻り値を取得するために将来のある時点で (force 手続きによって) 問い合わせることができるオブジェクトです。⟨expression⟩ が多値を返した場合の効果は規定されていません。

(delay-force ⟨expression⟩) lazy ライブラリの構文

意味論: (delay-force *expression*) 式は概念的には (delay (force *expression*)) と同様です。ただし delay-force の結果を force すると (force *expression*) に末尾呼び出しする点が異なります。それに対して (delay (force *expression*)) はそうでない可能性があります。そのため delay と force の長いチェーンとなる可能性のある遅延の繰り返しアルゴリズムは、delay-force を使って書き直すことで、評価中に空間を無制限に消費するのを防ぐことができます。

(force *promise*) lazy ライブラリの手続き

force 手続きは delay、delay-force または make-promise によって作成された *promise*の値を要求します。そのプロミスに対する値が計算されていなければ値が計算されて返されます。プロミスの値は二回目要求されたとき以前計算した値を返すためにキャッシュ(または「メモ化」) されます。そのため遅延式は最初に値を要求した force 呼び出しのパラメータおよび例外ハンドラを用いて評価されます。*promise* がプロミスでなければ、その値が変更されずに返されます。

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

(define integers
  (letrec ((next
            (lambda (n)
```

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

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

以下の例は遅延ストリームフィルタリングアルゴリズムを機械的に Scheme に変換したものです。構築手続きの呼び出しはそれぞれ delay に包まれ、参照手続きの引数はそれぞれ force に包まれています。手続き本体のまわりでは (delay (force ...)) の代わりに (delay-force ...) を使用することで、増加してゆく一連の保留中のプロミスが利用可能な記憶領域を使い切らないようにしています。これは force がそのような一連のプロミスを実質的に繰り返し force してくれるためです。

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

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

delay、force および delay-force は主に関数スタイルで書かれるプログラムでの使用が意図されており、以下の例は良いプログラミングスタイルを示しているわけではありませんが、何度要求されたかに関わらずひとつのプロミスに対してはひとつの値しか計算されないということを示す例になっています。

```
(define count 0)
(define p
  (delay (begin (set! count (+ count 1))
                (if (> count x)
                    count
                    (force p)))))
(define x 5)
p                             ⟹  プロミス
(force p)                     ⟹  6
p                             ⟹  プロミス (未だに)
(begin (set! x 10)
       (force p))             ⟹  6
```

処理系によっては delay、force および delay-force のこの意味論に様々な拡張が行われています。

- プロミスでないオブジェクトに対して force が呼ばれた場合は単にそのオブジェクトを返しても構いません。
- プロミスがその force された値といかなる意味でも操作的に区別できなくても構いません。つまり処理系は以下のような式を #t または #f のいずれに評価しても構いません。

```
(eqv? (delay 1) 1)            ⟹  規定されていない
(pair? (delay (cons 1 2)))    ⟹  規定されていない
```

- 処理系は「暗黙の force」を実装しても構いません。これは cdr や * のような特定の型の引数にのみ作用する手続きがプロミスの値を force するという機能です。ただし list のように引数を一様に処理する手続きはそれらを force してはなりません。

```
(+ (delay (* 3 7)) 13)        ⟹  規定されていない
(car
  (list (delay (* 3 7)) 13))  ⟹  プロミス
```

(promise? *obj*)　　lazy ライブラリの手続き

promise? 手続きはその引数がプロミスであれば #t を返し、そうでなければ #f を返します。プロミスが他の Scheme の型 (手続きなど) から独立している必要はないことに注意してください。

(make-promise *obj*)　　lazy ライブラリの手続き

make-promise 手続きは force されたときに *obj* を返すプロミスを返します。delay に似ていますが引数は遅延されません。これは構文ではなく手続きです。*obj* がすでにプロミスであればそれが返されます。

4.2.6. 動的束縛

手続き呼び出しが開始されてからそれが戻るまでの間の時間を手続き呼び出しの**動的生存期間**と呼びます。Scheme では call-with-current-continuation (6.10 節) によって、手続き呼び出しが戻った後もその動的生存期間に入り直すことができます。そのため呼び出しの動的生存期間は連続した単一の時間でない場合があります。

この節では**パラメータオブジェクト**が導入されます。これは動的生存期間に対して新しい値を束縛できるオブジェクトです。ある時点でのすべてのパラメータの束縛の集合は**動的環境**と呼ばれます。

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

新しく割り当てられたパラメータオブジェクトを返します。パラメータオブジェクトは引数を取らない手続きで、そのパラメータオブジェクトに紐付けられた値を返します。初期状態ではこの値は (*converter init*) の値、変換手続き *converter* が指定されていない場合は *init* の値です。紐付けられた値は以下で述べる parameterize を使って一時的に変更することができます。

パラメータオブジェクトに引数を渡したときの効果は処理系依存です。

`(parameterize ((⟨param$_1$⟩ ⟨value$_1$⟩) ...) ⟨body⟩)` 構文

構文: ⟨param$_1$⟩ および ⟨value$_1$⟩ は両方とも式です。

⟨param⟩ 式の値のいずれかがパラメータオブジェクトでない場合はエラーです。

意味論: `parameterize` 式は、本体を評価する間、指定されたパラメータオブジェクトの返す値を変更するために使用します。

⟨param⟩ および ⟨value⟩ 式の評価順序は規定されていません。⟨body⟩ は新しい動的環境で評価されます。その動的環境では、指定されたパラメータが呼ばれると対応する値を変換手続きに渡した結果を返します。変換手続きはパラメータオブジェクト作成時に指定したものです。その後パラメータの以前の値が変換手続きに渡されずに復元されます。⟨body⟩ 内の最後の式の結果が `parameterize` 式全体の結果として返されます。

メモ: 変換手続きが冪等でない場合、`(parameterize ((x (x))) ...)` はパラメータ xを現在の値に束縛しているように見えるものの、ユーザが期待する結果とは異なるかもしれません。

処理系がマルチスレッド実行をサポートしている場合、`parameterize` は現在のスレッドおよび ⟨body⟩ の中で生成されたスレッド以外のいかなるスレッドのいかなるパラメータに紐付けられた値も変更してはなりません。

パラメータオブジェクトは呼び出しチェーンのすべての手続きに値を明示的に渡す必要なく計算に対する変更可能な設定を指定するために使うことができます。

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

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

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

(radix 16)                      ⟹ 規定されていない

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

4.2.7. 例外処理

`(guard (⟨variable⟩ ⟨cond clause$_1$⟩ ⟨cond clause$_2$⟩ ...) ⟨body⟩)` 構文

構文: ⟨cond clause⟩ は `cond` の仕様に記載されているものと同じです。

意味論: ⟨body⟩ は例外ハンドラを持った状態で評価されます。例外ハンドラでは例外オブジェクト (6.11 節の `raise` を参照) が ⟨variable⟩ に束縛され、その束縛のスコープ内でそれぞれの節が `cond` 式の節であるかのように評価されます。暗黙の `cond` 式は `guard` 式の継続と動的環境で評価されます。すべての ⟨cond clause⟩ の ⟨test⟩ が `#f` に評価され、かつ else 節が無い場合、現在の例外ハンドラが `guard` 式のものであること以外 `raise` または `raise-continuable` の呼び出し元と同じ動的環境で、その例外オブジェクトに対して `raise-continuable` が呼び出されます。

例外のより完全な議論は 6.11 節を参照してください。

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

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

4.2.8. quasiquote

`(quasiquote ⟨qq template⟩)` 構文
`` `⟨qq template⟩ `` 構文
`unquote` 補助構文
`,` 補助構文
`unquote-splicing` 補助構文
`,@` 補助構文

「quasiquote」式は全部ではないけれど部分的にあらかじめ判っているようなリストやベクタ構造を構築する場合に便利です。⟨qq template⟩ 内にコンマが無ければ `` `⟨qq template⟩ `` を評価した結果は `'⟨qq template⟩` を評価した結果と同等です。しかし ⟨qq template⟩ 内にコンマがある場合、コンマに続く式は評価 (「unquote」) され、その結果がそのコンマと式の代わりにその構造内に挿入されます。コンマにホワイトスペースを挟まずアットマーク (`@`) が続いた場合、後続の式はリストに評価できなければエラーであり、そのリストの開き括弧および閉じ括弧は「剥ぎ取られ」、そのリストの要素がコンマ・アットマーク・式の並びのあった場所に挿入されます。通常コンマ・アットマークはリストおよびベクタの ⟨qq template⟩ 内にのみ現れます。

メモ: `@` で始まる識別子を unquote したい場合は明示的に `unquote` を使うか、コンマ・アットマークの並びと一致するのを避けるためコンマの後にホワイトスペースを置く必要があります。

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

quasiquote 式はネストできます。置換は最も外側の quasiquote と同じネストレベルに現れた unquote 部分にのみ行われます。ネストレベルは quasiquote の内側に入るごとにひとつ上がり、unquote の内側に入るごとにひとつ下がります。

```
`(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)
```

quasiquote 式は式の評価中に実行時に構築された任意の構造について、新しく割り当てられた可変オブジェクトを返してもリテラル構造を返しても構いません。再構築する必要のない部分は常にリテラルです。つまり

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

これは以下のいずれかの式と同等なものとして扱われる可能性があります。

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

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

しかし以下の式と同等ではありません。

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

2 種類の記法 `⟨qq template⟩ および (quasiquote ⟨qq template⟩) はあらゆる意味において同等です。,⟨expression⟩ は (unquote ⟨expression⟩) と同等であり,@⟨expression⟩ は (unquote-splicing ⟨expression⟩) と同等です。write 手続きはどちらの書式で出力しても構いません。

```
(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))
```

識別子 quasiquote、unquote、unquote-splicing のいずれかが上で説明した以外の ⟨qq template⟩ 内の位置に現れた場合はエラーです。

4.2.9. case-lambda

(case-lambda ⟨clause⟩ ...) case-lambda ライブラリ構文

構文: それぞれの ⟨clause⟩ は (⟨formals⟩ ⟨body⟩) の形を取ります。ただし ⟨formals⟩ および ⟨body⟩ は lambda 式の場合と同じ構文です。

意味論: case-lambda 式は可変個の引数を取る手続きに評価され、lambda 式から返される手続きと同様の字句的スコープを持ちます。この手続きが呼ばれると、その引数が ⟨formals⟩ とマッチする最初の ⟨clause⟩ が選択されます。ただしマッチは lambda 式の ⟨formals⟩ に対するものと同様に指定されます。⟨formals⟩ の変数が新しい場所に束縛され、その場所に引数の値が格納され、その拡張された環境で ⟨body⟩ が評価され、⟨body⟩ の結果がその手続き呼び出しの結果として返されます。

引数がどの ⟨clause⟩ の ⟨formals⟩ ともマッチしなかった場合はエラーです。

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

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

4.3. マクロ

Scheme のプログラムではマクロと呼ばれる新しい派生式型を定義して使うことができます。プログラムによって定義された式型は以下の構文を持ちます。

(⟨keyword⟩ ⟨datum⟩ ...)

ただし ⟨keyword⟩ はその式型を一意に決定する識別子です。この識別子はそのマクロの構文キーワードまたは単にキーワードと呼ばれます。⟨datum⟩ の数およびその構文はその式型に依存します。

マクロの実体化はそのマクロの使用と呼ばれます。マクロの使用をよりプリミティブな式にどのように変換するかを指定する規則の集合はそのマクロの変換子と呼ばれます。

マクロ定義機能はふたつの部分から成ります。

- 特定の識別子をマクロキーワードとして確立し、それをマクロ変換子に紐付け、マクロが定義されるスコープを制御するために使う式の集合。
- マクロ変換子を指定するためのパターン言語。

マクロの構文キーワードは変数束縛を覆い隠し、局所変数束縛は構文束縛を覆い隠します。意図しない衝突を防ぐためにふたつの仕組みがあります。

- マクロ変換子が識別子 (変数またはキーワード) に対する束縛を挿入した場合、その識別子は他の識別子との衝突を避けるためにそのスコープにおいて実質的に改名されます。ちなみに大域変数の定義は束縛を導入してもしなくても構いません。5.3 節を参照してください。
- マクロ変換子が識別子の自由参照を挿入した場合、その参照は変換子が指定された場所から見えていた束縛を参照します。マクロの使用場所のまわりにあるいかなる局所束縛も関係しません。

これによりパターン言語を用いて定義されたマクロはすべて「衛生的」かつ「参照透明」であり、Scheme の字句的スコープが保たれます。[21, 22, 2, 9, 12]

処理系は他の種類のマクロ機能を提供しても構いません。

4.3.1. 構文キーワードの束縛構文

`let-syntax` および `letrec-syntax` 束縛構文は `let` および `letrec` に似ていますが、値を持つ場所に変数を束縛する代わりに構文キーワードをマクロ変換子に束縛します。構文キーワードは `define-syntax` を使って大域的または局所的に束縛することもできます。5.4 節を参照してください。

`(let-syntax` ⟨bindings⟩ ⟨body⟩`)` 構文

構文: ⟨bindings⟩ は以下の形を取ります。

`((`⟨keyword⟩ ⟨transformer spec⟩`) ...)`

⟨keyword⟩ は識別子、⟨transformer spec⟩ は `syntax-rules` のインスタンス、⟨body⟩ はひとつ以上の定義にひとつ以上の式が続いたものです。束縛されるキーワードのリストに同じ ⟨keyword⟩ が 2 回以上現れた場合はエラーです。

意味論: `let-syntax` 式の構文環境を指定した変換子にキーワード ⟨keyword⟩ を束縛するマクロで拡張することによって得られた構文環境で ⟨body⟩ が展開されます。それぞれの ⟨keyword⟩ の束縛は ⟨body⟩ をその有効範囲とします。

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

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

`(letrec-syntax` ⟨bindings⟩ ⟨body⟩`)` 構文

構文: `let-syntax` と同じです。

意味論: `letrec-syntax` 式の構文環境を指定した変換子にキーワード ⟨keyword⟩ を束縛するマクロで拡張することによって得られた構文環境で ⟨body⟩ が展開されます。それぞれの ⟨keyword⟩ の束縛は ⟨body⟩ と同様に ⟨transformer spec⟩ もその有効範囲として持ち、そのため変換子はその `letrec-syntax` 式によって導入されたマクロを使用して式を書き換えることができます。

```
(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

⟨transformer spec⟩ は以下のいずれかの形を取ります。

`(syntax-rules (`⟨literal⟩ `...)` 構文
  ⟨syntax rule⟩ `...)`
`(syntax-rules` ⟨ellipsis⟩ `(`⟨literal⟩ `...)` 構文
  ⟨syntax rule⟩ `...)`
_ 補助構文
... 補助構文

構文: ⟨literal⟩ または 2 番目の形の ⟨ellipsis⟩ のいずれかが識別子でなければエラーです。⟨syntax rule⟩ が以下の形でない場合もエラーです。

`(`⟨pattern⟩ ⟨template⟩`)`

⟨syntax rule⟩ 内の ⟨pattern⟩ は最初の要素が識別子であるリストです。

⟨pattern⟩ は識別子、定数、または以下のいずれかです。

`(`⟨pattern⟩ `...)`
`(`⟨pattern⟩ ⟨pattern⟩ `...` `.` ⟨pattern⟩`)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...)`
`(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...`
  `.` ⟨pattern⟩`)`
`#(`⟨pattern⟩ `...)`
`#(`⟨pattern⟩ `...` ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩ `...)`

⟨template⟩ は識別子、定数、または以下のいずれかです。

`(`⟨element⟩ `...)`
`(`⟨element⟩ ⟨element⟩ `...` `.` ⟨template⟩`)`
`(`⟨ellipsis⟩ ⟨template⟩`)`
`#(`⟨element⟩ `...)`

ただし ⟨element⟩ は ⟨template⟩ であるか、⟨template⟩ に ⟨ellipsis⟩ が続いたものです。⟨ellipsis⟩ は `syntax-rules` の 2 番目の形で指定された識別子であるか、そうでなければデフォルトの識別子 `...` (連続した 3 つのピリオド) です。

意味論: `syntax-rules` のインスタンスは衛生的な書き換えルールの並びを指定することによって新しいマクロ変換子を生成します。`syntax-rules` で指定された変換子に紐付けられたキーワードのマクロを使用すると ⟨syntax rule⟩ 内のパターンに対してマッチされます。マッチは最も左の ⟨syntax rule⟩ から行われます。マッチが見つかるとそのマクロ使用はテンプレートに従って衛生的に書き換えられます。

⟨pattern⟩ 内に現れる識別子はアンダースコア (_)、⟨literal⟩ の一覧にあるリテラル識別子、または ⟨ellipsis⟩ で、⟨pattern⟩ に現れるそれ以外のすべての識別子は**パターン変数**です。

⟨syntax rule⟩ のパターンの最初の位置にあるキーワードはマッチングに影響せず、パターン変数ともリテラル識別子とも見なされません。

パターン変数は任意の入力要素とマッチし、テンプレート内でその入力要素を参照するために使われます。ひとつの ⟨pattern⟩ 内に同じパターン変数が 2 回以上現れた場合はエラーです。

アンダースコアは任意の入力要素とマッチしますがパターン変数ではなく、その要素を参照するために使うことはできません。アンダースコアが ⟨literal⟩ のリストに現れた場合はそれが優先され、⟨pattern⟩ 内のアンダースコアはリテラルとしてマッチされます。アンダースコアは ⟨pattern⟩ 内に複数回現れることができます。

(⟨literal⟩ ...) 内に現れた識別子は対応する入力要素に対してマッチさせるためのリテラル識別子として解釈されます。入力要素がリテラル識別子にマッチするには、その入力要素が識別子であり、マクロ式内のそれとマクロ定義内のそれが両方とも同じ字句的束縛を持っているか、2 つの識別子が同じであり両方とも字句的束縛を持っていない場合に限ります。

部分パターンに ⟨ellipsis⟩ が続く場合はゼロ個以上のその入力要素にマッチされます。ただし ⟨ellipsis⟩ が ⟨literal⟩ 内に現れた場合を除きます。その場合はリテラルとしてマッチされます。

より形式的に言うと以下の場合に限り入力要素 E はパターン P にマッチします。

- P がアンダースコア (_) である。
- P がリテラルでない識別子である。または
- P がリテラル識別子であり、 E が同じ束縛を持つ識別子である。または
- P がリスト (P_1 ... P_n) であり、 E が n 個の要素を持つリストであり、その要素が P_1〜P_n にそれぞれマッチする。または
- P が非真正リスト (P_1 P_2 ... P_n . P_{n+1}) であり、E が n 個以上の要素を持つリストであるか非真正リストであり、その要素が P_1〜P_n にそれぞれマッチし、その n 番目の末尾が P_{n+1} にマッチする。または
- P が (P_1 ... P_k P_e ⟨ellipsis⟩ P_{m+1} ... P_n) の形であり、E が n 個の要素を持つ真正リストであり、その最初の k 個がそれぞれ P_1〜P_k にマッチし、続く $m-k$ 個の要素それぞれが P_e にマッチし、残りの $n-m$ 個の要素が P_{m+1}〜P_n にマッチする。または
- P が (P_1 ... P_k P_e ⟨ellipsis⟩ P_{m+1} ... P_n . P_x) の形であり、E が n 個の要素を持つリストまたは非真正リストであり、その最初の k 個の要素が P_1〜P_k にマッチし、続く $m-k$ 個の要素それぞれが P_e にマッチし、残りの $n-m$ 個の要素が P_{m+1}〜P_n にマッチし、n 番目の要素である最後の cdr が P_x にマッチする。または
- P が `#(`P_1 ... P_n) の形のベクタであり、E が n 個の要素を持つベクタであり、その要素が P_1〜P_n にマッチする。または
- P が`#(`P_1 ... P_k P_e ⟨ellipsis⟩ P_{m+1} ...P_n) の形であり、E が n 個の要素を持つベクタであり、その最初の k 個の要素が P_1〜P_k にマッチし、続く $m-k$ 個の要素それぞれが P_e にマッチし、残りの $n-m$ 個の要素が P_{m+1}〜P_n にマッチする。または
- P が定数であり、 E が P と `equal?` 手続きの意味において等しい。

マクロキーワードをその束縛のスコープ内においてそのパターンのいずれにもマッチしない式で使用することはエラーです。

マクロの使用がマッチした ⟨syntax rule⟩ のテンプレートに従って書き換えられる際、テンプレート内のパターン変数が入力にマッチした要素に置き換えられます。識別子 ⟨ellipsis⟩ がひとつ以上後続する部分パターン内に現れたパターン変数は、同じ数の ⟨ellipsis⟩ が後続する部分テンプレート内でだけ使うことができます。入力中にマッチしたすべての要素により、指定された通りの現れ方で出力中のそれらが置き換えられます。指定された通りに出力を組み立てられない場合はエラーです。

テンプレート内に現れた、パターン変数でも識別子 ⟨ellipsis⟩ でもない識別子は、リテラル識別子として出力中に挿入されます。リテラル識別子が自由識別子として挿入された場合、それは `syntax-rules` のインスタンスが現れたスコープ内のその識別子の束縛を参照します。リテラル識別子が束縛識別子として挿入された場合、それは自由識別子を意図せず捕捉しないよう実質的に改名されます。

(⟨ellipsis⟩ ⟨template⟩) の形のテンプレートは ⟨template⟩ と同一です。ただしこのテンプレート内では省略記号は特別な意味を持ちません。つまり ⟨template⟩ 内に含まれる省略記号はすべて通常の識別子として扱われます。特にテンプレート (⟨ellipsis⟩ ⟨ellipsis⟩) は単一の ⟨ellipsis⟩ を生成します。

これにより省略記号を含むコードに展開する構文抽象が可能になります。

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

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

例えば let および cond が 7.3 節のように定義されている場合、それらは衛生的であり (要求通りです)、以下はエラーとはなりません。

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

cond のマクロ変換子は => が局所変数であり、従ってそれは式であり、マクロ変換子が構文キーワードとして扱うべき base ライブラリの識別子 => とは異なるということを認識します。そのため上記の例は以下のように展開されます。

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

以下のようには展開されません。

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

もしこのように展開された場合、無効な手続き呼び出しが行われてしまうでしょう。

4.3.3. マクロ変換子のエラー通知

(syntax-error ⟨message⟩ ⟨args⟩ ...) 構文

syntax-error は error (6.11) と同様に動作しますが、展開のパスと評価のパスが分かれている処理系では syntax-error が展開されたら直ちにエラーを通知するべきです。これはマクロの無効な使用方法である ⟨pattern⟩ に対する syntax-rules の ⟨template⟩ として使うことができ、より説明的なエラーメッセージを提供することができます。⟨message⟩ は文字列リテラルで、⟨args⟩ は追加の情報を提供する任意の式です。アプリケーションは例外ハンドラやガードで構文エラーを捕捉できると考えてはいけません。

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

5. プログラムの構造

5.1. プログラム

Scheme のプログラムはひとつ以上のインポート宣言に続く式および定義の並びで構成されます。インポート宣言はプログラムまたはライブラリが依存するライブラリを指定します。そのライブラリからエクスポートされている識別子のサブセットがプログラムで利用可能となります。式は 4 章で説明しています。定義は変数定義、構文定義、レコード型定義のいずれかで、これらはすべてこの章で説明します。これらは式を書ける場所のうちいくつかの部分 (すべてではない) に、特に ⟨program⟩ の最も外側のレベルおよび ⟨body⟩ の最初の部分に書くことができます。

プログラムの最も外側のレベルでは、(begin ⟨expression or definition$_1$⟩ ...) はその begin の中の式および定義の並びと同等です。同様に ⟨body⟩ では、(begin ⟨definition$_1$⟩ ...) は ⟨definition$_1$⟩ ... の並びと同等です。マクロはそのような begin の形に展開されることがあります。形式的な定義は 4.2.3 を参照してください。

インポート宣言および定義は大域環境に束縛を作成し、または既存の大域束縛の値を変更します。プログラムの初期状態の環境は空です。そのため最初の束縛を導入するために少なくともひとつのインポート宣言が必要です。

プログラムの最も外側のレベルに現れる式は束縛を作成しません。これらはプログラムが呼び出されたときまたはロードされたときに順番に実行され、通常、何らかの種類の初期化を行います。

プログラムおよびライブラリは通常、ファイルに格納されています。処理系によっては実行中の Scheme システムに対話的に入力できます。他の方式も有り得ます。ライブラリをファイルに格納している処理系は、ライブラリの名前からファイルシステム中の場所への対応付けを文章化するべきです。

5.2. インポート宣言

インポート宣言は以下の形を取ります。

(import ⟨import-set⟩ ...)

インポート宣言はライブラリからエクスポートされている識別子をインポートする方法を提供します。それぞれの ⟨import set⟩ はライブラリからインポートされる束縛の集合の名前を指定し、またそのインポートした束縛に局所的な名前を指定することもできます。以下の形のいずれかを取ります。

- ⟨library name⟩
- (only ⟨import set⟩ ⟨identifier⟩ ...)
- (except ⟨import set⟩ ⟨identifier⟩ ...)

- (`prefix` ⟨import set⟩ ⟨identifier⟩)
- (`rename` ⟨import set⟩
  (⟨identifier$_1$⟩ ⟨identifier$_2$⟩) ...)

最初の形式では、その名前のライブラリの export 節に記載されているすべての識別子を同じ名前で (`rename` でエクスポートされている場合はそのエクスポート名で) インポートします。他の形式の ⟨import set⟩ はこの集合を以下のように修正します。

- `only` は指定された ⟨import set⟩ のうち、指定された識別子 (もしあれば名前変更後の) のみを含む部分集合を生成します。指定された識別子のいずれかが元の集合に見つからない場合はエラーです。
- `except` は指定された ⟨import set⟩ から、指定された識別子 (もしあれば名前変更後の) を除いた部分集合を生成します。指定された識別子のいずれかが元の集合に見つからない場合はエラーです。
- `rename` は指定された ⟨import set⟩ を変更し、⟨identifier$_1$⟩ を ⟨identifier$_2$⟩ に置換します。指定された ⟨identifier$_1$⟩ のいずれかが元の集合に見つからない場合はエラーです。
- `prefix` は指定された ⟨import set⟩ のすべての識別子を改名し、指定された ⟨identifier⟩ をその先頭に付けます。

プログラムおよびライブラリ宣言では、同じ識別子を異なる束縛でインポートしたり、インポートした束縛を定義で再定義したり、`set!` で変更したり、識別子をインポートする前にそれを参照することはエラーです。しかし REPL ではそのような動作は許容されるべきです。

5.3. 変数定義

変数定義はひとつ以上の識別子を束縛し、その初期値を指定します。最も単純な種類の変数定義は以下のいずれかの形を取ります。

- (`define` ⟨variable⟩ ⟨expression⟩)
- (`define` (⟨variable⟩ ⟨formals⟩) ⟨body⟩)
  ⟨formals⟩ はゼロ個以上の変数の並びであるか、ひとつ以上の変数にスペースで区切られたピリオドともうひとつ別の変数が続いたものです (lambda 式と同様です)。この形は以下と同等です。

```
(define ⟨variable⟩
  (lambda (⟨formals⟩) ⟨body⟩))
```

- (`define` (⟨variable⟩ . ⟨formal⟩) ⟨body⟩)
  ⟨formal⟩ は単一の変数です。この形は以下と同等です。

```
(define ⟨variable⟩
  (lambda ⟨formal⟩ ⟨body⟩))
```

5.3.1. 最上位の定義

プログラムの最も外側のレベルでは、以下のような定義

```
(define ⟨variable⟩ ⟨expression⟩)
```

は ⟨variable⟩ がすでに構文以外の値に束縛されている場合、実質的に以下の代入式と同じ効果を持ちます。

```
(set! ⟨variable⟩ ⟨expression⟩)
```

ただし ⟨variable⟩ が束縛されていないか構文キーワードである場合、上記の定義は代入を行う前に ⟨variable⟩ を新しい場所に束縛します。それに対し `set!` は束縛されていない変数に対して行うとエラーです。

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

5.3.2. 内部定義

定義は ⟨body⟩ の最初に書くこともできます。⟨body⟩ は `lambda`、`let`、`let*`、`letrec`、`letrec*`、`let-values`、`let*-values`、`let-syntax`、`letrec-syntax`、`parameterize`、`guard` または `case-lambda` の本体です。このような本体は他の構文の展開後まで現れない場合があることに注意してください。このような定義は前述の大域定義に対して**内部定義**と呼ばれます。内部定義で定義された変数はその ⟨body⟩ に局所的です。つまり ⟨variable⟩ は代入されるのではなく束縛され、⟨body⟩ 全体をその束縛の有効範囲とします。以下に例を示します。

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

内部定義を持つ ⟨body⟩ の展開形は完全に同等な `letrec*` 式に常に変換できます。例えば上の例の `let` 式は以下と同等です。

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

この同等な `letrec*` 式と同様に、⟨body⟩ で定義されたそれぞれの内部定義の ⟨expression⟩ は対応する ⟨variable⟩ および ⟨body⟩ 内の後続する ⟨variable⟩ に代入も参照もせず評価できなければならず、そうでなければエラーです。

同じ ⟨body⟩ で同じ識別子を 2 回以上定義した場合はエラーです。

内部定義を書ける場所では常に (`begin` ⟨definition$_1$⟩ ...) はその `begin` の本体を形成する定義の並びと同等です。

5.3.3. 多値の定義

もうひとつの種類の定義は `define-values` です。多値を返す式から複数の定義を作成します。`define` が書ける場所ならどこでも書くことができます。

`(define-values` ⟨formals⟩ ⟨expression⟩`)` 構文

⟨formals⟩ の集合に同じ変数が 2 回以上現れた場合はエラーです。

意味論: ⟨expression⟩ が評価され、`lambda` 式が手続き呼び出しで引数を ⟨formals⟩ にマッチさせるのと同じ方法で、その値が ⟨formals⟩ に束縛されます。

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

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

5.4. 構文定義

構文定義は以下の形を取ります。

`(define-syntax` ⟨keyword⟩ ⟨transformer spec⟩`)`

⟨keyword⟩ は識別子で、⟨transformer spec⟩ は `syntax-rules` のインスタンスです。変数定義と同様に構文定義は最も外側のレベルか本体内にネストして書くことができます。

`define-syntax` が最も外側のレベルに現れた場合、その ⟨keyword⟩ に指定された変換子を束縛することによって大域構文環境が拡張されますが、⟨keyword⟩ の大域束縛を用いたすでに展開済みのものはそのまま残ります。そうでない場合それは**内部構文定義**であり、それが定義された ⟨body⟩ に局所的なものとなります。対応する定義の前に構文キーワードが使用された場合はエラーです。特に内部定義に先行する使用に外側の定義を適用することはできません。

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

マクロは定義が許される文脈では定義に展開することができます。しかしその定義自身や同じグループの内部定義に属する先行する定義の意味を決めるために束縛が既知である必要のある識別子を定義する定義はエラーです。同様にそれが属する本体中の内部定義と式の境界線を決めるために既知である必要のある識別子を定義する内部定義もエラーです。例えば以下の例はエラーです。

```
(define define 3)

(begin (define begin list))

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

5.5. レコード型定義

レコード型定義は**レコード型**と呼ばれる新しいデータ型を定義するために使われます。他の定義と同様に最も外側のレベルか本体内で使うことができます。レコード型の値は**レコード**と呼ばれます。レコードはゼロ個以上の**フィールド**の集合体です。それぞれのフィールドはひとつずつ場所を持ちます。それぞれのレコード型に対して述語、コンストラクタ、フィールドアクセサおよびミューテータが定義されます。

`(define-record-type` ⟨name⟩ 構文
⟨constructor⟩ ⟨pred⟩ ⟨field⟩ ...`)`

構文: ⟨name⟩ および ⟨pred⟩ は識別子です。⟨constructor⟩ は以下の形を取ります。

(⟨constructor name⟩ ⟨field name⟩ ...)

それぞれの ⟨field⟩ は以下の形のいずれかを取ります。

(⟨field name⟩ ⟨accessor name⟩)
(⟨field name⟩ ⟨accessor name⟩ ⟨modifier name⟩)

フィールド名として同じ識別子が 2 回以上現れた場合はエラーです。アクセサまたはミューテータの名前として同じ識別子が 2 回以上現れた場合もエラーです。

`define-record-type` 構文は生成的です。Scheme の定義済みの型や他のレコード型、同じ名前や同じ構造を持つレコード型も含め、すでに存在するどんな型とも異なる独立した新しいレコード型が使用のたびに作られます。

`define-record-type` の使用は以下の定義と同等です。

- ⟨name⟩ はレコード型それ自身の表現に束縛されます。これは実行時オブジェクトでも純粋に構文的な表現でも構いません。この表現はこの報告書では使用されませんが、今後言語を拡張する際に使うためにそのレコード型を識別するものとして提供されます。
- ⟨constructor name⟩ は、(⟨constructor name⟩ ...) 部分式内の ⟨field name⟩ と同じ個数だけの引数を取り、型 ⟨name⟩ の新しいレコードを返す手続きに束縛されます。⟨constructor name⟩ と共に名前が指定されたフィールドは、対応する引数をその初期値として持ちます。それ

以外のすべてのフィールドの初期値は規定されていません。⟨field name⟩ に無いフィールド名が ⟨constructor⟩ 内に現れた場合はエラーです。

- ⟨pred⟩ は、⟨constructor name⟩ に束縛された手続きが返した値を指定すると `#t` を返し、それ以外のすべてに対して `#f` を返す述語に束縛されます。
- それぞれの ⟨accessor name⟩ は、型 ⟨name⟩ のレコードを取り対応するフィールドの現在の値を返す手続きに束縛されます。適切な型のレコード以外の値をアクセサに渡した場合はエラーです。
- それぞれの ⟨modifier name⟩ は、型 ⟨name⟩ のレコードと対応するフィールドの新しい値を取る手続きに束縛されます。戻り値は規定されていません。適切な型のレコード以外の値をモディファイアの第 1 引数に渡した場合はエラーです。

例えば、以下のレコード型定義は

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

`<pare>` のインスタンスに対して `kons` をコンストラクタ、`kar` および `kdr` をアクセサ、`set-kar!` をモディファイア、`pare?` を述語として定義します。

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

5.6. ライブラリ

ライブラリは Scheme のプログラムを、プログラムの他の部分への明示的に定義されたインタフェースを持った再利用可能な部品に編成する手段です。この節ではライブラリの記法と意味論を定義します。

5.6.1. ライブラリの構文

ライブラリ定義は以下の形を取ります。

`(define-library` ⟨library name⟩
⟨library declaration⟩ ...`)`

⟨library name⟩ は識別子および正確な非負の整数を内容とするリストです。これは他のプログラムやライブラリからインポートする際にそのライブラリを一意に識別するために使われます。最初の識別子が `scheme` であるライブラリは、この報告書および報告書の将来のバージョンで使用するために予約されています。最初の識別子が `srfi` であるライブラリは Scheme Requests for Implementation を実装するライブラリ用に予約されています。文字 `| \ ? * < " : > + [ ] /` および制御文字 (エスケープ展開後) を含むような識別子をライブラリ名に使用することは推奨されませんが、エラーではありません。

⟨library declaration⟩ は以下のいずれかの形を取ります。

- `(export` ⟨export spec⟩ ...`)`
- `(import` ⟨import set⟩ ...`)`
- `(begin` ⟨command or definition⟩ ...`)`
- `(include` ⟨filename$_1$⟩ ⟨filename$_2$⟩ ...`)`
- `(include-ci` ⟨filename$_1$⟩ ⟨filename$_2$⟩ ...`)`
- `(include-library-declarations` ⟨filename$_1$⟩ ⟨filename$_2$⟩ ...`)`
- `(cond-expand` ⟨ce-clause$_1$⟩ ⟨ce-clause$_2$⟩ ...`)`

`export` 宣言は他のライブラリまたはプログラムに見せる識別子のリストを指定します。⟨export spec⟩ は以下の形のいずれかを取ります。

- ⟨identifier⟩
- `(rename` ⟨identifier$_1$⟩ ⟨identifier$_2$⟩`)`

⟨export spec⟩ では ⟨identifier⟩ がそのライブラリで定義されたかそのライブラリにインポートされた束縛のひとつに名前を付けます。ただしそのエクスポートする外部名はそのライブラリ内の束縛の名前と同じです。`rename` 指定はそれぞれの (⟨identifier$_1$⟩ ⟨identifier$_2$⟩) ペアについて、そのライブラリ内で定義されたかそのライブラリにインポートされた ⟨identifier$_1$⟩ という名前の束縛を ⟨identifier$_2$⟩ という外部名でエクスポートします。

`import` 宣言は他のライブラリからエクスポートされた識別子をインポートする手段です。これはプログラムや REPL で使われるインポート宣言と同じ構文と意味論を持ちます (5.2 節を参照)。

`begin`、`include` および `include-ci` 宣言はライブラリの本体を指定するために使われます。これらは対応する式型と同じ構文と意味論を持ちます。`begin` は 4.2.3 節で定義されている 2 種類の `begin` に似ていますが、同じではありません。

`include-library-declarations` 宣言は `include` と似ていますが、ファイルの内容を現在のライブラリ定義内に直接継ぎ合わせる点が異なります。これは例えば、同じ形のライブ

ラリインタフェースを持つ複数のライブラリで同じ export 宣言を共有するために使うことができます。

cond-expand 宣言は cond-expand 式型と同じ構文と意味論を持ちますが、式が begin で囲まれるのではなくライブラリ宣言に継ぎ合わされる点が異なります、

ライブラリの実装方法のひとつとして以下のような方式が考えられるでしょう。まずすべての cond-expand ライブラリ宣言を展開します。そしてすべてのインポートした束縛から成るそのライブラリ用の新しい環境を構築します。それから begin、include および include-ci ライブラリ宣言によるすべての式をその環境でそのライブラリ内に現れた順番で展開します。あるいは、それぞれの import 宣言によってインポートされた束縛を追加するたびに環境を成長させながら左から右の順番で他の宣言の処理と一緒に cond-expand および import 宣言を処理していっても構いません。

ライブラリがロードされるとその式が書かれた順番で実行されます。ライブラリの定義がプログラムまたはライブラリ本体の展開済みの形から参照される場合、そのライブラリは展開されたプログラムまたはライブラリ本体が評価される前にロードされなければなりません。このルールは推移的です。あるライブラリが 2 つ以上のプログラムまたはライブラリからインポートされている場合、そのライブラリが何度かロードされる可能性があっても構いません。

同様に、あるライブラリ (foo) の展開中にそのライブラリを展開するために他のライブラリ (bar) からインポートした構文キーワードが必要な場合、(foo) の展開前にライブラリ (bar) が展開されその構文定義が評価されなければなりません。

ライブラリがロードされる回数に関わらず、インポート宣言が現れた数に関わらず、ライブラリから束縛をインポートするそれぞれのプログラムまたはライブラリはそのライブラリの単一のロードからそれらの束縛をインポートしなければなりません。つまり (import (only (foo) a)) に (import (only (foo) b)) が続いたものは (import (only (foo) a b)) と同じ効果を持ちます。

5.6.2. ライブラリの例

プログラムをどのようにライブラリと比較的小さなメインプログラムに分割するかを以下の例に示します [16]。メインプログラムを REPL に入力する場合は base ライブラリをインポートする必要はありません。

```
(define-library (example grid)
  (export make rows cols ref each
          (rename put! set!))
  (import (scheme base))
  (begin
    ;; Create an NxM grid.
    (define (make n m)
      (let ((grid (make-vector n)))
        (do ((i 0 (+ i 1)))
            ((= i n) grid)
          (let ((v (make-vector m #false)))
            (vector-set! grid i v)))))
    (define (rows grid)
      (vector-length grid))
    (define (cols grid)
      (vector-length (vector-ref grid 0)))
    ;; Return #false if out of range.
    (define (ref grid n m)
      (and (< -1 n (rows grid))
           (< -1 m (cols grid))
           (vector-ref (vector-ref grid n) m)))
    (define (put! grid n m v)
      (vector-set! (vector-ref grid n) m v))
    (define (each grid proc)
      (do ((j 0 (+ j 1)))
          ((= j (rows grid)))
        (do ((k 0 (+ k 1)))
            ((= k (cols grid)))
          (proc j k (ref grid j k)))))))

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

```
;; Main program.
(import (scheme base)
        (only (example life) life)
        (rename (prefix (example grid) grid-)
                (grid-make make-grid)))

;; Initialize a grid with a glider.
(define grid (make-grid 24 24))
(grid-set! grid 1 1 #true)
(grid-set! grid 2 2 #true)
(grid-set! grid 3 0 #true)
(grid-set! grid 3 1 #true)
(grid-set! grid 3 2 #true)

;; Run for 80 iterations.
(life grid 80)
```

5.7. REPL

処理系は *REPL* (Read-Eval-Print Loop) と呼ばれる対話環境を提供していても構いません。これはインポート宣言、式、定義を一度にひとつずつ入力し評価することができる環境です。利便性と使用の容易さのため REPL における Scheme の大域環境は空であってはならず、少なくとも base ライブラリで提供されている束縛を持った状態で開始しなければなりません。このライブラリは Scheme の中核となる構文とデータを操作する一般的に有用な手続きが含まれています。例えば変数 `abs` は引数をひとつ取り数値の絶対値を計算する手続きに束縛されており、変数 `+` は和を計算する手続きに束縛されています。(scheme base) の束縛の完全なリストは付録 A に掲載されています。

処理系はすべての有り得る変数がすでに場所に束縛されているかのように動作する初期状態の REPL 環境を提供していても構いません。その場合ほとんどの場所は初期値が規定されていません。そのような処理系では最上位の REPL の定義は完全に代入と同等です。ただしその識別子が構文キーワードとして定義されている場合を除きます。

処理系はファイルから入力を読み込む REPL の動作モードを提供していても構いません。そういったファイルは開始以外の場所にインポート宣言を持つことができるので、一般的にプログラムと同じではありません。

6. 標準手続き

この章では Scheme の組み込み手続きを説明します。

手続き `force`、`promise?` および `make-promise` は式型 `delay` および `delay-force` と密接に関連しているため、4.2.5 節で説明しています。同様に手続き `make-parameter` は式型 `parameterize` と密接に関連しているため、4.2.6 節で説明しています。

プログラムは大域変数定義を用いて任意の変数を束縛できます。それらの束縛は後に代入によって変更される可能性があります (4.1.6 節を参照)。これらの操作がこの報告書で定義されている手続きの動作を変更することはありません。またライブラリ (5.6 節を参照) からインポートされた手続きの動作を変更することもありません。定義によって導入されたものでない大域変数を変更した場合、この章で定義されている手続きの動作に与える効果は規定されていません。

手続きが**新しく割り当てられた**オブジェクトを返すと言うとき、それはそのオブジェクトの場所が新しいという意味です。

6.1. 等値述語

常にブーリアン値 (`#t` または `#f`) を返す手続きを**述語**と呼びます。**等値述語**は数学の等値関係をコンピュータ的に真似たものです。つまり対称性を持ち、反射性を持ち、推移的です。この節で説明している等値述語は `eq?` が最も細かく (最も識別性が高く)、`equal?` が最も粗く、`eqv?` がその中間です。

(`eqv?` obj_1 obj_2) 手続き

`eqv?` 手続きはオブジェクトに対する有用な等値関係を定義します。大雑把に言うと `eqv?` は obj_1 と obj_2 が普通に考えて同じオブジェクトである場合に `#t` を返します。この関係は若干解釈の余地が残されていますが、以下に述べる `eqv?` の部分的仕様はすべての Scheme 処理系において維持されています。

`eqv?` は以下の場合に `#t` を返します。

- obj_1 と obj_2 が共に `#t` であるか、共に `#f` である。
- obj_1 と obj_2 が共にシンボルであり、`symbol=?` 手続きによれば等しい (6.5 節)。
- obj_1 と obj_2 が共に正確な数値であり、(`=` の意味で) 数値的に等しい。
- obj_1 と obj_2 が共に不正確な数値であり、(`=` の意味で) 数値的に等しく、そして Scheme 標準の数値計算手続き (ただし NaN 値を返さない場合に限る) の有限個の合成関数として定義することのできる任意の他の手続きに引数として渡したときに (`eqv?` の意味で) 同じ結果を生成する。
- obj_1 と obj_2 が共に文字であり、`char=?` 手続きによれば同じ文字である (6.6 節)。
- obj_1 と obj_2 が共に空リストである。
- obj_1 と obj_2 がペア、ベクタ、バイトベクタ、レコードまたは文字列であり、同じ格納場所を指し示す (3.4 節)。
- obj_1 と obj_2 が手続きであり、同じ場所に紐付けられている (4.1.4 節)。

`eqv?` 手続きは以下の場合に `#f` を返します。

- obj_1 と obj_2 が異なる型である (3.2 節)。
- obj_1 または obj_2 の一方が `#t` であり、他方が `#f` である。
- obj_1 および obj_2 がシンボルであるが、`symbol=?` 手続きによれば同じシンボルでない (6.5 節)。
- obj_1 または obj_2 の一方が正確な数値であり、他方が不正確な数値である。
- obj_1 と obj_2 が共に正確な数値であるが、(`=` の意味で) 数値的に等しくない。
- obj_1 と obj_2 が共に不正確な数値であり、(`=` の意味で) 数値的に等しくないか、または Scheme 標準の数値計算手続き (ただし NaN 値を返さない場合に限る) の有限個の合成関数として定義することのできる任意の他の手続きに引数として渡したときに (`eqv?` の意味で) 同じ結果を生成しない。例外として、obj_1 と obj_2 が共に NaN であるとき、`eqv?` の動作は規定されていません。
- obj_1 および obj_2 が文字であり、`char=?` 手続きが `#f` を返す。
- obj_1 または obj_2 の一方が空リストであり、他方がそうでない。
- obj_1 および obj_2 がペア、ベクタ、バイトベクタ、レコードまたは文字列であり、その指し示す場所が異なる。
- obj_1 および obj_2 が手続きであり、何らかの引数に対して異なる動作をする (異なる値を返すか、異なる副作用を持つ)。

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

前述のルールで `eqv?` の動作が完全には規定されていない状況を以下の例に示します。そのような場合において言えることは、`eqv?` の返す値がブーリアンでなければならない、ということだけです。

```
(eqv? "" "")                    ⟹  規定されていない
(eqv? '#() '#())                ⟹  規定されていない
(eqv? (lambda (x) x)
      (lambda (x) x))           ⟹  規定されていない
(eqv? (lambda (x) x)
      (lambda (y) y))           ⟹  規定されていない
(eqv? 1.0e0 1.0f0)              ⟹  規定されていない
(eqv? +nan.0 +nan.0)            ⟹  規定されていない
```

負のゼロが区別されている場合 (`eqv? 0.0 -0.0`) は `#f` を返し、負のゼロが区別されていない場合は `#t` を返します。

次の一連の例は局所的な状態を持つ手続きに対する `eqv?` の使用を示します。`gen-counter` 手続きは毎回別々の手続きを返さなければなりません。なぜならそれぞれ別個の内部カウンタが必要であるためです。しかし `gen-loser` 手続きは毎回操作的に同等な手続きを返します。なぜなら局所状態がその手続きの値にも副作用にも影響しないためです。しかし `eqv?` はこの等しさを検出してもしなくても構いません。

```
(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))
                                ⟹  規定されていない

(letrec ((f (lambda () (if (eqv? f g) 'both 'f)))
         (g (lambda () (if (eqv? f g) 'both 'g))))
  (eqv? f g))
                                ⟹  規定されていない

(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 (リテラル式によって返される) を変更することはエラーであるので、処理系は適切な状況においては定数間で構造を共有しても構いません。そのため定数に対する `eqv?` の値は処理系依存になることがあります。

```
(eqv? '(a) '(a))                ⟹  規定されていない
(eqv? "a" "a")                  ⟹  規定されていない
(eqv? '(b) (cdr '(a b)))        ⟹  規定されていない
(let ((x '(a)))
  (eqv? x x))                   ⟹  #t
```

上で述べた `eqv?` の定義は手続きおよびリテラルの扱いについて処理系に自由を与えます。処理系は 2 つの手続きや 2 つのリテラルがお互いに同等であるか検出できてもできなくてもよく、2 つの同等なオブジェクトの表現を同じビットパターンやポインタを用いてマージするかしないかを選ぶことができます。

メモ: 不正確な数値が IEEE 二進浮動小数点数値で表現している場合、同じサイズの不正確な数値を単純にビットごとの比較で行う `eqv?` の実装は上記の定義によれば正しいものです。

(eq? obj_1 obj_2) 手続き

eq? 手続きは eqv? 手続きに似ています。ただしいくつかの場合において eqv? で検出可能なよりも細かい差異を識別する能力があります。eqv? が #f を返す状況では同様に #f を返さなければなりませんが、eqv? が #t を返す状況でも #f を返す場合があります。

シンボル、ブーリアン、空リスト、ペア、レコード、および空でない文字列、ベクタ、バイトベクタにおいて eq? と eqv? は同じ動作をすることが保証されています。手続きにおいては引数の場所の紐付けが同じ場合 #t を返さなければなりません。数値および文字においては eq? の動作は処理系依存です。ただし必ず真または偽のどちらかを返します。空文字列、空ベクタ、空バイトベクタにおいても eq? と eqv? は異なる動作をして構いません。

```
(eq? 'a 'a)                  ⟹  #t
(eq? '(a) '(a))              ⟹  規定されていない
(eq? (list 'a) (list 'a))    ⟹  #f
(eq? "a" "a")                ⟹  規定されていない
(eq? "" "")                  ⟹  規定されていない
(eq? '() '())                ⟹  #t
(eq? 2 2)                    ⟹  規定されていない
(eq? #\A #\A)                ⟹  規定されていない
(eq? car car)                ⟹  #t
(let ((n (+ 2 3)))
  (eq? n n))                 ⟹  規定されていない
(let ((x '(a)))
  (eq? x x))                 ⟹  #t
(let ((x '#()))
  (eq? x x))                 ⟹  #t
(let ((p (lambda (x) x)))
  (eq? p p))                 ⟹  #t
```

論拠: 通常 eq? は eqv? よりも非常に効率良く実装することができます。例えば eqv? がいくらか複雑な操作をする代わりに eq? は単なるポインタの比較で済みます。その理由のひとつは 2 つの数値の eqv? を計算することが常に定数時間でできるわけではないということです。それに対してポインタ比較で実装された eq? は必ず定数時間で処理できます。

(equal? obj_1 obj_2) 手続き

equal? 手続きはペア、ベクタ、文字列、バイトベクタに適用された場合、それらを再帰的に比較します。それらを (無限長の可能性もある) 木構造に展開したとき順序付きの木構造として (equal? の意味で) 等しければ #t を返し、そうでなければ #f を返します。ブーリアン、シンボル、数値、文字、ポート、手続きおよび空リストに適用されたときは eqv? と同じです。2 つのオブジェクトが eqv? であれば同様に equal? でもなければなりません。それ以外のすべての状況では equal? は #t を返しても #f を返しても構いません。

引数が循環データ構造であっても equal? は必ず終了しなければなりません。

```
(equal? 'a 'a)               ⟹  #t
(equal? '(a) '(a))           ⟹  #t
(equal? '(a (b) c)
        '(a (b) c))          ⟹  #t
(equal? "abc" "abc")         ⟹  #t
(equal? 2 2)                 ⟹  #t
(equal? (make-vector 5 'a)
        (make-vector 5 'a))  ⟹  #t
(equal? '#1=(a b . #1#)
        '#2=(a b a b . #2#)) ⟹  #t
(equal? (lambda (x) x)
        (lambda (y) y))      ⟹  規定されていない
```

メモ: 大雑把に言うと、2 つのオブジェクトが同じようにプリントされる場合、一般的にそれらは equal? です。

6.2. 数値

数学の数値と、それをモデル化した Scheme の数値と、それを表現するために使われる機械表現と、数値を書くために使われる記法を区別することは重要です。この報告書では数値、複素数、実数、有理数、整数といった型を数学の数値と Scheme の数値の両方を示すために用います。

6.2.1. 数値の型

数学的には、各階がそれより上の階の部分型であるような塔に、数値を編成することができます。

数値
複素数
実数
有理数
整数

例えば 3 は整数です。またそれゆえに 3 は有理数でもあり、実数でもあり、複素数でもあります。これは 3 をモデル化した Scheme の数値においても同様です。Scheme の数値では述語 number?、complex?、real?、rational? および integer? によりこれらの型が定義されます。

数値の型とそのコンピュータにおける内部表現の間には単純な関係はありません。ほとんどの Scheme 処理系では 3 の表現が少なくとも 2 種類は提供されていますが、これらの異なる表現は同じ整数を表しています。

Scheme の数値計算では数値を可能な限りその表現から独立した抽象的なデータとして扱います。Scheme 処理系は数値の内部表現を複数用いても構いませんが、単純なプログラムを書くカジュアルプログラマーには判らないようにするべきです。

6.2.2. 正確性

正確に表現された数値とそうでない可能性のある数値の区別は有用です。例えばデータ構造へのインデックスは、記号代数系における多項式の係数と同様に、正確に判明している必要があります。一方、計測の結果などは本質的に不正確であったり、無理数などは有理数によって近似された不正確な近似値であったりします。正確な数値が必要なところでの不

正確な数値の使用を捕捉するために Scheme では正確な数値と不正確な数値を区別しています。この区別は型の次元とは直交しています。

Scheme では正確な定数として書かれたか、正確な演算のみを用いて正確な数値から得られた場合、その数値は正確です。不正確な数値として書かれたか、不正確な発生源から得られたか、不正確な演算を用いて得られた場合、その数値は不正確です。従って数値の不正確性は伝染する性質を持ちます。特に**正確な複素数**は正確な実部と正確な虚部を持ち、そうでないすべての複素数は**不正確な複素数**です。

2 つの処理系がある計算に対して不正確な中間結果を生じずに正確な結果を生成する場合、その 2 つの最終結果は数学的に等しくなります。これは不正確な数値を生成する計算では一般的に成立しません。浮動小数点計算のような近似的な手法が使われる場合があるためです。しかし各々の処理系は数学上の理想的な結果に実用上十分近い結果を生成する義務があります。

+ のような有理数の演算は正確な引数が与えられると必ず正確な結果を生成します。演算が正確な結果を生成することができない場合は処理系の制限の違反を報告するか、その結果を不正確な値に黙って変換しても構いません。ただし (/ 3 4) が 0 を返すようなことは許容されません。それは数学的に正しくありません。6.2.3 節を参照してください。

exact を除き、この節で説明されている演算は一般に、不正確な引数が与えられた場合は不正確な結果を返さなければなりません。ただしその結果の値が引数の不正確性に影響を受けないことが保証できる場合は正確な結果を返しても構いません。例えば正確なゼロにはどんな数値を掛けても正確なゼロの結果を生成することができます。たとえそれが不正確な値であってもです。

具体的な例として式 (* 0 +inf.0) は 0 を返しても構いませんし、+nan.0 を返しても構いませんし、不正確な数値をサポートしていない旨を報告しても構いませんし、有理数でない実数をサポートしていない旨を報告しても構いませんし、黙って死んでも構いませんし、あるいは処理系固有の方法で騒がしくエラーを知らせても構いません。

6.2.3. 処理系の制限

Scheme 処理系は 6.2.1 節で述べた部分型の塔全体を実装することは要求されていません。しかし処理系の目的と Scheme 言語の精神の両方を満たす一貫性のある部分集合を実装しなければなりません。例えばすべての数値が実数である処理系、実数以外の数値が常に不正確である処理系、正確な数値が常に整数である処理系などは依然として非常に有用でしょう。

処理系はこの節の要求を満たす限り、任意の型の数値のある特定の範囲のみをサポートしても構いません。任意の型の正確な数値のサポートされている範囲がその型の不正確な数値のサポートされている範囲と異なっていても構いません。例えばすべての不正確な実数を表現するために IEEE 二進倍精度浮動小数点数値を採用している処理系では、不正確な実数の範囲 (ゆえに不正確な整数および有理数の範囲も) が IEEE 二進倍精度形式のダイナミックレンジに制限される一方、正確な整数と有理数を事実上無制限の範囲でサポートしていて構いません。さらに言えばそのような処理系では、この範囲制限に近づくにつれ表現可能な不正確な整数および有理数の隙間が非常に大きくなる可能性があります。

Scheme 処理系はリスト、ベクタ、バイトベクタ、文字列のインデックスのために、およびそれらの長さを計算した結果のために必要な数値の範囲全体に対して、正確な整数をサポートしなければなりません。length、vector-length、bytevector-length および string-length 手続きは正確な整数を返さなければなりません。またインデックスとして正確な整数以外のものを使用することはエラーです。さらにインデックス範囲内のあらゆる整数定数は、この範囲外で適用されるいかなる処理系の制限にも関わらず、正確な整数の構文で表現されていれば正確な整数として読めなければなりません。最後に、以下の一覧に記載されている手続きはすべての引数が正確な整数でかつ数学的に期待される結果が処理系の範囲内の正確な整数で表現可能ならば必ず正確な整数の結果を返さなければなりません。

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

処理系は事実上無制限の大きさと精度を持つ正確な整数と正確な有理数とサポートし、上記の手続きと / 手続きを正確な引数に対して必ず正確な結果を返すよう実装することが推奨されますが要求されません。これらの手続きはいずれも正確な引数を与えられたとき正確な結果を返すことができなければ、処理系の制限の違反を報告しても構いませんし、黙ってその結果を不正確な数値に変換しても構いません。そのような変換は後のエラーの原因となる可能性があります。とはいえ正確な有理数を提供していない処理系は処理系の制限を報告するよりも不正確な有理数を返す方が良いでしょう。

処理系は不正確な数値に対して浮動小数点や他の近似表現戦略を用いても構いません。この報告書では IEEE 754 標準に従った浮動小数点表現を用いることが推奨されますが要求されません。他の表現方法を用いる処理系ではこの浮動小数点標準を用いて達成可能な精度と同等かそれを超えることが推奨されますが要求されません。特にそのような処理系は IEEE 754-2008 の超越関数の記述に、とりわけ無限大と NaN に関して、従うべきです。

Scheme は数値に対する様々な書き方を規定していますが、処理系はそれらの一部しかサポートしなくても構いません。

例えば数値がすべて実数である処理系は、複素数の直交座標表示や極座標表示をサポートする必要はありません。処理系が正確な数値として表現できない正確な数値定数に出会った場合、処理系の制限の違反を報告しても構いませんし、黙って不正確な数値で表現しても構いません。

6.2.4. 処理系の拡張

処理系は 2 つ以上の異なる精度の浮動小数点数値表現を提供していても構いません。そのような処理系では不正確な結果は少なくともその演算に使われたどの不正確引数も表現できるだけの精度を持っていなければなりません。`sqrt` のような潜在的に不正確な演算は正確な引数を適用したときは正確な結果を生成することが望ましいものの、もし正確な数値を演算して不正確な結果を生成する場合は利用可能な中で最も精度の高い表現を用いなければなりません。例えば `(sqrt 4)` の値は `2` となるべきですが、単精度と倍精度の浮動小数点数値を両方提供している処理系では、後者を用いても構いませんが、前者を用いてはなりません。

その処理系で表現するには大きすぎる絶対値や仮数部を持つ不正確な数値オブジェクトの使用を避けるのはプログラマーの責任です。

さらに処理系は正の無限大、負の無限大、NaN、および負のゼロといった特別な数値を区別しても構いません。

正の無限大は有理数で表現可能ないかなる数値よりも大きな不定の値を表現する不正確な実数 (しかし有理数ではない) と見なされます。負の無限大は有理数で表現可能ないかなる数値よりも小さな不定の値を表現する不正確な実数 (しかし有理数ではない) と見なされます。

無限大の値にいかなる有限の実数を加算および乗算してもその結果は (適切な符号の) 無限大となります。しかし正の無限大と負の無限大の和は NaN です。正の無限大はゼロの逆数で、負の無限大は負のゼロの逆数です。IEEE 754 に従った超越関数の動作は無限大に対して非常に複雑です。

NaN は任意の実数を表し得る不定の不正確な実数 (しかし有理数ではない) と見なされます。これには正負の無限大や、正の無限大より大きな値、負の無限大より小さな値も含まれます。実数以外の数値をサポートしない処理系では `(sqrt -1.0)` や `(asin 2.0)` のような実数でない値を表現するために NaN を用いても構いません。

NaN はどのような数値と比較しても必ず偽になります。NaN 自身と比較しても同様です。数値演算は引数のいずれかが NaN であれば NaN を返します。ただしその NaN をどのような有理数と置き換えても結果は同じであると処理系が保証できる場合を除きます。ゼロをゼロで除算すると、両方のゼロが正確でなければ、結果は NaN になります。

負のゼロは不正確な実数であり、`-0.0` と書かれ、(`eqv?` の意味で) `0.0` と区別されます。Scheme 処理系は負のゼロを区別することは要求されません。しかし区別する場合は超越関数の動作は IEEE 754 に従った複雑なものになります。特に複素数と負のゼロを両方サポートする Scheme 処理系は、複素対数関数の分岐を `(imag-part (log -1.0-0.0i))` が π でなく $-\pi$ となるように処理しなければなりません。

さらに言えば負のゼロの反数は通常のゼロであり、逆も同様です。これは 2 つ以上の負のゼロの和が負であり、負のゼロから (正の) ゼロを引いた結果も同様に負であることを暗黙に示しています。しかし数値的な比較においては、負のゼロとゼロは等しいものとして扱われます。

ちなみに複素数の実部と虚部はいずれも無限大、NaN または負のゼロを取ることができます。

6.2.5. 数値定数の構文

数値の表現を書くための構文は 7.1.1 節で形式的に記述されています。ちなみに数値定数では大文字小文字は区別されません。

数値は基数接頭辞を使うことで 2 進数、8 進数、10 進数または 16 進数で書くことができます。基数接頭辞は `#b` (2 進数)、`#o` (8 進数)、`#d` (10 進数)、および `#x` (16 進数) です。基数接頭辞が無ければ数値は 10 進数で表現されているとみなされます。

数値定数は接頭辞によって正確または不正確のいずれかを指定できます。正確の接頭辞は `#e` で不正確の接頭辞は `#i` です。正確性接頭辞は基数接頭辞の前でも後でも構いません。正確性接頭辞を付けずに数値の表現を書いた場合、小数点または指数があればその定数は不正確であり、そうでなければ正確です。

様々な精度の不正確な数値を持つシステムでは定数の精度を指定できると有用です。このため処理系は不正確な表現の希望精度を指定する指数マーカーが書かれた数値定数を受け付けても構いません。その場合、文字 `e` の場所に、その代わりに `s`、`f`、`d` または `l` を使うことができ、それぞれ短精度、単精度、倍精度、長精度を意味しています。デフォルトの精度は少なくとも倍精度以上でなければなりませんが、処理系はこのデフォルトをユーザーの設定によって変更できても構いません。

```
3.14159265358979F0
        単精度に丸められる — 3.141593
0.6L0
        長精度に拡張される — .600000000000000
```

正の無限大、負の無限大、NaN はそれぞれ `+inf.0`、`-inf.0`、`+nan.0` と書かれます。NaN は `-nan.0` と書かれることもあります。書かれた表現の符号の使用は、もし NaN 値の内部表現に符号があっても、それを反映する必要はありません。処理系はこれらの数値をサポートすることは要求されていませんが、サポートする場合は全般的に IEEE 754 に準拠しなければなりません。ただし処理系は Signaling NaN をサポートしたり異なる NaN を区別する方法を提供することは要求されません。

実数でない複素数を表記するための記法が 2 つあります。ひとつは**直交座標表示**で a`+`b`i` のように表記します。ただし a は実部で bは虚部です。もうひとつは**極座標表示**で r`@`θ のように表記します。ただし rは動径で θ はラジアンで表した位相 (偏角) です。これらは $a + b\mathrm{i} = r\cos\theta + (r\sin\theta)\mathrm{i}$ の関係があります。a、b、rおよび θ はすべて実数です。

6.2.6. 数値演算

数値ルーチンの引数の型の制限を指定するために使われる命名規約の要約については 1.3.3 節を参照してください。この節の例では正確な表記で書かれた数値定数はいずれも実際に正確な数値を表しているものとみなしています。また不正確な表記で書かれた数値定数は精度を失うことなく表現されているものとみなしています。そういった不正確な定数は、不正確な数値の表現に IEEE 二進倍精度を採用している処理系でその仮定が成立するように選ばれています。

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

これらの数値型の述語は数値でないものを含むいかなる型の引数にも適用できます。オブジェクトがその名前の型であれば #t を返し、そうでなければ #f を返します。一般的に、ある型の述語がある数値に対して真であれば、より上位の型の述語もすべてその数値に対して真となります。従って、ある型の述語がある数値に対して偽であれば、より下位の型の述語もすべてその数値に対して偽となります。

z が複素数の場合 (real? z) は (zero? (imag-part z)) が真のときに限り真となります。x が不正確な実数である場合 (integer? x) は (= x (round x)) が真のときに限り真となります。

+inf.0、-inf.0 および +nan.0 は実数ですが有理数ではありません。

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

メモ: 不正確な数値に対するこれらの型の述語の動作は信頼できません。不正確さが結果に影響する場合があるためです。

メモ: 多くの処理系では complex? 手続きは number? と同じですが、ある種の無理数を正確に表現できたり、数値系を拡張して何らかの複素数でない数値をサポートしたりする、普通でない処理系があるかもしれません。

(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判定します。どのようなScheme の数値もこれらの述語のいずれか片方だけが真になります。

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

(exact-integer? z) 手続き

z が正確かつ整数であれば #t を返し、そうでなければ #f を返します。

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

(finite? z) inexact ライブラリの手続き

finite? 手続きは +inf.0、-inf.0、+nan.0 以外のすべての実数および実部と虚部が共に有限である複素数に対して #t を返します。そうでなければ #f を返します。

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

(infinite? z) inexact ライブラリの手続き

infinite? 手続きは +inf.0、-inf.0 および実部または虚部または両方が無限大である複素数に対して #t を返します。そうでなければ #f を返します。

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

(nan? z) inexact ライブラリの手続き

nan? 手続きは +nan.0 および実部または虚部または両方が +nan.0 である複素数に対して #t を返します。そうでなければ #f を返します。

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

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

これらの手続きは引数がそれぞれ等しい、単調に増加している、単調に減少している、単調に減少していない、単調に増加していない場合に #t を返し、そうでなければ #f を返します。引数のいずれかが +nan.0 の場合はどの手続きも #f

を返します。これらは不正確なゼロと不正確な負のゼロを区別しません。

これらの手続きは推移的であることが要求されます。

メモ: いずれかの引数が不正確であればすべての引数を不正確な数値に変換する、というような実装手法は推移的ではありません。例えば、big を (expt 2 1000) として、その big が正確であり、不正確な数値は 64 ビットの IEEE 二進浮動小数点数で表現されているとしましょう。その場合、この実装手法では大きな整数の IEEE 表現の制限のため (= (- big 1) (inexact big)) と (= (inexact big) (+ big 1)) が共に真でありながら (= (- big 1) (+ big 1)) は偽となってしまうでしょう。不正確な数値をそれと (= の意味で) 同じ正確な数値に変換すればこの問題を回避できますが、無限大に対して特別な配慮が必要となります。

メモ: これらの述語を用いて不正確な数値を比較することはエラーではありませんが、わずかな不正確さが結果に影響を及ぼす可能性があるためその結果は信頼できません。= や zero? の場合、特にそうです。疑わしい場合は数値解析の専門家に相談してください。

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

これらの数値述語は特定の性質を判定し、#t または #f を返します。上記の注意点も参照してください。

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

これらの手続きは引数の最大値または最小値を返します。

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

メモ: いずれかの引数が不正確であれば結果も不正確になります (その不正確さが結果に影響しないほど大きくないことが保証できる場合は除きますが、そのようなことは普通でない処理系にのみ可能なことです)。min または max を使って正確性が混在した数値を比較し、その結果の数値を正確さを犠牲にすることなく不正確な数値で表現することができない場合、これらの手続きは処理系の制限の違反を報告しても構いません。

(+ z_1 ...) 手続き
(* z_1 ...) 手続き

これらの手続きは引数の和または積を返します。

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

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

引数がふたつ以上の場合、これらの手続きは左結合で引数の差または商を返します。しかし引数がひとつの場合は、その引数の反数または逆数を返します。

/ の第 2 引数以降のいずれかが正確なゼロの場合はエラーです。第 1 引数が正確なゼロであり他の引数がいずれも NaN でなければ、正確なゼロを返しても構いません。

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

(abs x) 手続き

abs 手続きは引数の絶対値を返します。

```
(abs -7)                 ⟹  7
```

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

これらの手続きは数論的な (整数の) 除算を実装します。n_2 がゼロの場合はエラーです。/ で終わる手続きはふたつの整数を返し、それ以外の手続きはひとつの整数を返します。どの手続きも $n_1 = n_2 n_q + n_r$ が成り立つような商 n_q と剰余 n_r を計算します。それぞれの除算演算子に対して 3 つの手続きが以下のように定義されます。

(⟨operator⟩/ n_1 n_2) ⟹ n_q n_r
(⟨operator⟩-quotient n_1 n_2) ⟹ n_q
(⟨operator⟩-remainder n_1 n_2) ⟹ n_r

剰余 n_r は整数 n_q が決まると自動的に $n_r = n_1 - n_2 n_q$ のように決定されます。n_q の決め方は各演算子によって異なります。

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

いずれの演算子も、またいずれの整数 n_1 および n_2(ただし n_2 がゼロでない場合) においても、以下が成り立ちます。

(= n_1 (+ (* n_2 (⟨operator⟩-quotient n_1 n_2))
(⟨operator⟩-remainder n_1 n_2)))
⟹ #t

ただしすべての数値が正確な計算によって得られる場合に限ります。

例:

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

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

quotient および remainder 手続きはそれぞれ truncate-quotient および truncate-remainder と同等であり、modulo は floor-remainder と同等です。

メモ: これらの手続きは以前のバージョンの報告書との後方互換性のために提供されています。

(gcd n_1 ...) 手続き
(lcm n_1 ...) 手続き

これらの手続きは引数の最大公約数または最小公倍数を返します。結果は必ず非負です。

```
(gcd 32 -36)              ⟹ 4
(gcd)                     ⟹ 0
(lcm 32 -36)              ⟹ 288
(lcm 32.0 -36)            ⟹ 288.0  ; 不正確
(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または分母を返します。結果は引数が既約分数として表現されているかのように計算されます。分母は必ず正です。ゼロの分母は 1 であると定義されます。

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

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

これらの手続きは整数を返します。floor 手続きは x より大きくない最も大きな整数を返します。ceiling 手続きは x より小さくない最も小さな整数を返します。truncate 手続きは絶対値が x の絶対値より大きくない x に最も近い整数を返します。round 手続きは x に最も近い整数を返しますが、x がふたつの整数の中央のときは偶数側に丸めます。

論拠: round 手続きの偶数丸めは IEEE 754 IEEE 浮動小数点標準で規定されているデフォルトの丸めモードとの一貫性のためです。

メモ: これらの手続きの引数が不正確な場合、結果も不正確になります。正確な値が必要であれば結果を exact 手続きに渡しても構いません。引数が無限大または NaN の場合はそのまま返されます。

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

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

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

(rationalize x y) 手続き

rationalize 手続きは x から距離 y 以内の**最も簡単**な有理数を返します。ある有理数 r_1 が別の有理数 r_2 より**簡単**であるとは、$r_1 = p_1/q_1$ および $r_2 = p_2/q_2$ (いずれも既約) としたとき、$|p_1| \leq |p_2|$ かつ $|q_1| \leq |q_2|$ である場合のことを言います。つまり 3/5 は 4/7 より簡単です。すべての有理数がこの順序付けで比較できるわけではありませんが (2/7 と 3/5 を考えてみてください)、どのような区間においても他のすべての有理数より簡単な有理数というものがひとつ存在しています (2/7 と 3/5 の間にはより簡単な 2/5 があります)。ちなみにすべての有理数のうち最も簡単なものは $0 = 0/1$ です。

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

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

これらの手続きは通常の超越関数を計算します。log 手続きは引数がひとつの場合は z の自然対数を計算し (10 を底とする対数ではありません)、引数がふたつの場合は z_2 を底とする z_1 の対数を計算します。asin、acos、atan 手続きはそれぞれ逆正弦 ($\sin^{-1}$)、逆余弦 ($\cos^{-1}$)、逆正接 ($\tan^{-1}$) を計算します。atan の 2 引数バージョンは (angle

(`make-rectangular` x y)) (後述) を計算します (処理系が複素数をサポートしていない場合でも)。

一般的に対数、逆正弦、逆余弦、逆正接といった数学関数は多値関数として定義されます。$\log z$ の値はその虚部が $-\pi$ (`-0.0` が区別されている場合は含まれず、そうでなければ含まれる) から π (常に含まれる) の範囲にある場合は 1 に定義されます。$\log 0$ の値は数学的に未定義です。log をこのように定義すると $\sin^{-1} z$、$\cos^{-1} z$、$\tan^{-1} z$ は以下の式に従います。

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

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

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

しかし処理系が無限大 (および `-0.0`) をサポートしていれば、(`log 0.0`) は `-inf.0` を返します (そして (`log -0.0`) は `-inf.0+`π`i` を返します)。

(`atan` y x) の範囲は以下の表のようになります。星印 (*) は負のゼロを区別する処理系に適用される項目であることを示しています。

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

上記の仕様は [34] に従ったもので、それは [26] から引用されたものです。分岐条件や境界条件およびこれらの関数の実装についてのより詳細な議論はこれらの情報源を参照してください。可能であればこれらの手続きは実数の引数から実数の結果を生成します。

(`square` z) 手続き

z の平方を返します。これは (`*` z z) と同等です。

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

(`sqrt` z) inexact ライブラリの手続き

z の正の平方根を返します。結果は正の実部を持つか、ゼロの実部と非負の虚部を持つかのいずれかです。

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

(`exact-integer-sqrt` k) 手続き

$k = s^2 + r$ および $k < (s+1)^2$ が成り立つふたつの非負の正確な整数 s および r を返します。

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

(`expt` z_1 z_2) 手続き

z_1 の z_2 乗を返します。z_1 がゼロでなければ、これは

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

です。0^z の値は (`zero? z`) の場合 1、(`real-part z`) が正の場合 0 で、それ以外はエラーです。0.0^z の場合も同様ですが不正確な結果になります。

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

実数 x_1、x_2、x_3、x_4 および複素数 z について

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

が成り立つ場合、以下がすべて成り立ちます。

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

ただし $-\pi \leq x_{angle} \leq \pi$ かつ $x_{angle} = x_4 + 2\pi n$ (n は整数) とします。

`make-polar` は引数が正確であっても不正確な複素数を返して構いません。`real-part` および `imag-part` 手続きは不正確な複素数に適用した場合でも、`make-rectangular` に渡された対応する引数が正確であったならば、正確な実数を返して構いません。

論拠: `magnitude` 手続きは実数の引数に対しては `abs` と同じですが、`abs` は base ライブラリの手続きであるのに対し、`magnitude` はオプショナルな complex ライブラリの手続きとなっています。

(inexact *z*) 手続き
(exact *z*) 手続き

手続き inexact は *z* の不正確な表現を返します。戻り値は数値的に引数に最も近い不正確な数値です。不正確な引数に対してはその引数と同じ値を返します。正確な複素数に対しては引数の実部と虚部をそれぞれ inexact に適用した結果を実部と虚部に持つ複素数を返します。正確な引数に十分近い (= の意味で) 同等な不正確な値がない場合は、処理系の制限の違反を報告しても構いません。

手続き exact は *z* の正確な表現を返します。戻り値は数値的に引数に最も近い正確な値です。正確な引数に対してはその引数と同じ値を返します。整数でない不正確な実数の引数に対しては有理数による近似値を返しても構いませんし、処理系の制限の違反を報告しても構いません。不正確な複素数の引数に対しては引数の実部と虚部をそれぞれ exact に適用した結果を実部と虚部に持つ複素数を返します。不正確な引数に十分近い (= の意味で) 同等な正確な値がない場合は、処理系の制限の違反を報告しても構いません。

これらの手続きは処理系依存の範囲内の正確な整数と不正確な整数に対して自然な 1 対 1 の対応関係を実装しています。6.2.3 節を参照してください。

メモ: これらの手続きは R^{5}RS ではそれぞれ exact->inexact および inexact->exact として知られていました。しかしこれらは常にどちらの正確性の引数も受け付けていました。新しい名前は R^{6}RS と互換性があると同時により明確で短いものです。

6.2.7. 数値の入出力

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

radix が 2、8、10、16 のいずれでもなければエラーです。

手続き number->string は数値と基数を取り、以下の式を満たすような指定した基数における指定した数値の外部表現を文字列として返します。

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

この式を満たせる結果が存在しない場合はエラーです。*radix* を省略した場合のデフォルト値は 10 です。

z が不正確で、基数が 10 で、かつ小数点を含む結果によって上記の式が満たせる場合、その結果は小数点を含む上記の式を満たすために必要な最小限の桁数 (指数と末尾のゼロを除く) で表現されます [4, 5]。そうでない場合、結果の書式は規定されていません。

number->string の戻り値が明示的な基数接頭辞を持つことはありません。

メモ: エラーの状況は *z* が複素数でないか、実部または虚部が有理数でない複素数の場合にのみ発生する可能性があります。

論拠: *z* が不正確な数値かつ基数が 10 の場合、上記の式は通常、小数点を含む結果によって満たすことができます。規定されていない結果は無限大、NaN、あるいは普通でない表現の場合に許容されます。

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

string によって表された数値の最大限に正確な表現を返します。*radix* が 2、8、10、16 のいずれでもなければエラーです。

radix が指定された場合はそれがデフォルトの基数となります。これは *string* に明示的な基数接頭辞があればオーバーライドされます (例えば "#o177")。*radix* が指定されなかった場合デフォルトの基数は 10 です。*string* が構文的に有効な数値の表記でない場合、または結果の数値を処理系が表現できない場合、string->number は #f を返します。*string* の中身を理由にエラーが通知されることはありません。

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

メモ: 処理系は string->number の定義域を以下のように制限しても構いません。処理系のサポートする数値が実数のみの場合、*string* が極座標表示または直交座標表示の複素数であれば string->number は #f を返すことが許容されます。数値が整数のみの場合、分数表記が使われたら #f を返しても構いません。正確な数値のみの場合、指数マーカーや明示的な正確性接頭辞が使われたら #f を返しても構いません。不正確な数値が整数のみの場合、小数点が使われたら #f を返しても構いません。

内部の数値処理、I/O、プログラム処理の間で一貫性を維持するため、ある特定の処理系が string->number に対して用いるルールは、read やプログラムの読み込みルーチンにも適用されなければなりません。従って *string* が明示的な基数接頭辞を持っている場合は #f を返しても良いという R^{5}RS の記述は廃止されました。

6.3. ブーリアン

真および偽に対する標準のブーリアンオブジェクトは #t および #f のように書きます。 代わりに、それぞれ #true および #false と書くこともできます。しかし本当に重要な点は、Scheme の条件式 (if、cond、and、or、when、unless、do) が真または偽として扱うオブジェクトです。用語「真の値」(または単に「真」) は条件式において真として扱われるあらゆるオブジェクトを意味し、用語「偽の値」(または「偽」) は条件式において偽として扱われるあらゆるオブジェクトを意味します。

すべての Scheme の値のうち #f のみが条件式において偽とみなされます。#t を含めそれ以外のすべての Scheme の値は真とみなされます。

メモ: 他の Lisp 方言と異なり、Scheme では #f と空リストはお互いに区別され、またシンボル nil とも区別されます。

ブーリアン定数はそれ自身に評価されるためプログラム中で quote する必要はありません。

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

(not *obj*) 手続き

not 手続きは *obj*が偽の場合 #t を返し、そうでなければ #f を返します。

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

(boolean? *obj*) 手続き

boolean? 述語は *obj*が #t または #f の場合 #t を返し、そうでなければ #f を返します。

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

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

引数がすべてブーリアンで、すべて #t であるかすべて #f であれば、#t を返します。

6.4. ペアとリスト

ペア (**ドット対**と呼ばれることもあります) は (歴史的理由により) car および cdr という名前で呼ばれるふたつのフィールドを持つレコード構造です。ペアは手続き cons で作ることができます。手続き car および cdr で car および cdr フィールドにアクセスできます。手続き set-car! および set-cdr! で car および cdr フィールドに代入できます。

ペアは主にリストを表現するために使われます。**リスト**は空リストまたは cdr がリストであるペアとして再帰的に定義できます。より正確に言うとリストの集合は以下を満たす最小の集合 X として定義されます。

- 空リストは X の要素です。
- リストが X の要素であれば、cdr フィールドにリストを持つペアもすべて X の要素です。

リストを構成するペアの car フィールドのオブジェクトはそのリストの要素です。例えば 2 要素のリストとは、ペアであって、そのペアの car が最初の要素、そのペアの cdr がまたペアであって、そのペアの car が 2 番目の要素、そのペアの cdr が空リストであるようなものを言います。リストの長さは要素の数であり、ペアの数と同じです。

空リストは独立した型を持つ特殊なオブジェクトです。それはペアではなく、要素を持たず、その長さはゼロです。

メモ: 上記の定義は、すべてのリストが有限の長さを持ち、空リストで終端することを暗黙に示しています。

Scheme のペアの最も汎用的な表記 (外部表現) は (*c*1 . *c*2) のような「ドット」記法です。ただし c_1 は car フィールドの値で、c_2 は cdr フィールドの値です。例えば (4 . 5) は car が 4 であり cdr が 5 であるペアです。(4 . 5) はペアの外部表現であって、ペアに評価される式ではないことに注意してください。

リストに対してはより流線的な記法が使われます。単純にリストの要素をスペースで区切って括弧で囲みます。空リストは () と書きます。例を挙げます。

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

これはシンボルのリストの表記で、以下と同等です。

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

空リストで終端しないペアのチェーンは**非真正リスト**と呼ばれます。非真正リストはリストではないことに注意してください。リストとドット記法を組み合わせて非真正リストを表現することができます。

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

これは以下と同等です。

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

与えられたペアがリストであるかどうかは cdr フィールドに何が格納されているかに依ります。set-cdr! 手続きを使うと、ある瞬間にはリストであったオブジェクトが次の瞬間にはそうでなくなる場合があります。

```
(define x (list 'a 'b 'c))
(define y x)
y                               ⟹  (a b c)
(list? y)                       ⟹  #t
(set-cdr! x 4)                  ⟹  規定されていない
x                               ⟹  (a . 4)
(eqv? x y)                      ⟹  #t
y                               ⟹  (a . 4)
(list? y)                       ⟹  #f
(set-cdr! x x)                  ⟹  規定されていない
(list? x)                       ⟹  #f
```

リテラル式や read 手続きによって読み込んだオブジェクトの表現の中では '⟨datum⟩、`⟨datum⟩、,⟨datum⟩、,@⟨datum⟩ といった形は最初の要素がそれぞれシンボル quote、quasiquote、unquote、unquote-splicing である 2 要素のリストを表します。それぞれの 2 番目の要素は ⟨datum⟩ です。この規約により任意の Scheme プログラムをリストとして表現できます。つまり Scheme の文法によれば、すべての ⟨expression⟩ は ⟨datum⟩ でもある、ということです (7.1.2 節を参照)。特に、これにより read 手続きで Scheme のプログラムをパースできるようになっています。3.3 節も参照してください。

(pair? *obj*) 手続き

pair? 述語は *obj*がペアであれば #t を返し、そうでなければ #f を返します。

```
(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 であり cdr が obj_2 である新しく割り当てられたペアを返します。このペアは既存のいかなるオブジェクトとも (eqv? の意味で) 異なることが保証されています。

```
(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します。空リストの car を取ることはエラーであることに注意してください。

```
(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します。空リストの cdr を取ることはエラーであることに注意してください。

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

(set-car! *pair obj*) 手続き

*pair*の car フィールドに *obj*を格納します。

```
(define (f) (list 'not-a-constant-list))
(define (g) '(constant-list))
(set-car! (f) 3)            ⟹  規定されていない
(set-car! (g) 3)            ⟹  エラー
```

(set-cdr! *pair obj*) 手続き

*pair*の cdr フィールドに *obj*を格納します。

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

これらの手続きは以下のような car および cdr の合成関数です。

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

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

これら 24 個の手続きは同じ考え方に基づいた car および cdr のさらなる合成関数です。例えば caddr は以下のように定義できます。

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

深さ 4 までのすべての組み合わせが提供されています。

(null? *obj*) 手続き

*obj*が空リストであれば #t を返し、そうでなければ #f を返します。

(list? *obj*) 手続き

*obj*がリストであれば #t を返し、そうでなければ #f を返します。定義により、リストはすべて有限の長さを持ち、空リストで終端します。

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

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

*k*個の要素を持つ新しく割り当てられたリストを返します。第 2 引数が指定された場合は各要素が *fill*に初期化されます。そうでなければ各要素の初期内容は規定されていません。

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

(list *obj* ...) 手続き

その引数から成る新しく割り当てられたリストを返します。

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

(length *list*) 手続き

*list*の長さを返します。

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

(append *list* ...) 手続き

最後の引数 (もしあれば) は任意の型を指定できます。

最初の *list*の要素に他の *list*の要素を続けたものを要素とするリストを返します。引数が無ければ空リストが返されます。引数がひとつだけの場合はそれが返されます。それ以外の場合は、結果のリストは常に新しく割り当てられますが、最後の引数の構造は共有します。最後の引数が真正リストでなければ結果は非真正リストです。

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

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

(reverse *list*) 手続き

*list*の要素から成る逆順の新しく割り当てられたリストを返します。

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

(list-tail *list k*) 手続き

*list*の要素が *k* 個より少ない場合はエラーです。

最初の *k* 個の要素を除いて得られる *list*の部分リストを返します。list-tail 手続きは以下のように定義できます。

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

(list-ref *list k*) 手続き

*list*引数は循環構造でも構いません。*list*の要素が *k* 個より少ない場合はエラーです。

*list*の *k* 番目の要素を返します。(これは (list-tail *list k*) の car と同じです。)

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

(list-set! *list k obj*) 手続き

k が *list*の有効なインデックスでない場合はエラーです。

list-set! 手続きは *list*の *k* 番目の要素に *obj*を格納します。

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

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

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

これらの手続きは car が *obj*である最初の *list*の部分リストを返します。*list*の部分リストは (list-tail *list k*) によって返される空でないリストです (*k*は *list*の長さより小さいものとします)。*list*内に *obj*が現れない場合は #f が返されます (空リストではありません)。memq 手続きは *obj*と *list*の要素との比較に eq? を用いるのに対して、memv は eqv? を用い、member は *compare*が指定された場合はそれを用い、そうでなければ equal? を用います。

```
(memq 'a '(a b c))          ⟹  (a b c)
(memq 'b '(a b c))          ⟹  (b c)
(memq 'a '(b c d))          ⟹  #f
(memq (list 'a) '(b (a) c)) ⟹  #f
(member (list 'a)
        '(b (a) c))         ⟹  ((a) c)
(member "B"
        '("a" "b" "c")
        string-ci=?)        ⟹  ("b" "c")
(memq 101 '(100 101 102))   ⟹  規定されていない
(memv 101 '(100 101 102))   ⟹  (101 102)
```

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

alist(“association list”(連想リスト) の略) がペアのリストでなければエラーです。

これらの手続きは car フィールドが *obj*である *alist*内の最初のペアを探し、そのペアを返します。car に *obj*を持つペアが *alist*内に無ければ #f が返されます (空リストではありません)。assq 手続きは *obj*と *alist*内のペアの car フィールドとの比較に eq? を用いるのに対して、assv は eqv? を用い、assoc は *compare*が指定された場合はそれを用い、そうでなければ equal? を用います。

```
(define e '((a 1) (b 2) (c 3)))
(assq 'a e)                 ⟹  (a 1)
(assq 'b e)                 ⟹  (b 2)
(assq 'd e)                 ⟹  #f
(assq (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  #f
(assoc (list 'a) '(((a)) ((b)) ((c))))
                            ⟹  ((a))
(assoc 2.0 '((1 1) (2 4) (3 9)) =)
```

```
                              ⟹ (2 4)
(assq 5 '((2 3) (5 7) (11 13)))
                              ⟹ 規定されていない
(assv 5 '((2 3) (5 7) (11 13)))
                              ⟹ (5 7)
```

論拠: memq、memv、member、assq、assv および assoc はよく述語として用いられはしますが、名前に疑問符は付いていません。これは単なる #t または #f でなく、場合によっては有用な値を返すためです。

(list-copy *obj*) 手続き

*obj*がリストの場合、その新しく割り当てられたコピーを返します。コピーされるのはペア自身だけです。結果の car は *list*の car と (eqv? の意味で) 同じになります。*obj*が非真正リストの場合、結果も非真正リストとなり、最後の cdr は eqv? の意味で同じになります。*obj*がリストでない場合はそのまま返されます。*obj*が循環リストの場合はエラーです。

```
(define a '(1 8 2 8)) ; a は不変かもしれない
(define b (list-copy a))
(set-car! b 3)        ; b は可変
b                             ⟹ (3 8 2 8)
a                             ⟹ (1 8 2 8)
```

6.5. シンボル

シンボルはその名前が同じ綴りである場合に限り (eqv? の意味で) 等しいという事実に有用性があるオブジェクトです。例えば他の言語では列挙型を使うような場面で使うことができます。

シンボルの書き方の規則は識別子の書き方の規則とまったく同じです。2.1 節および 7.1.1 節を参照してください。

リテラル式の一部として返され、または read 手続きを用いて読み込まれ、その後 write 手続きを用いて書き出されたシンボルはすべて、(eqv? の意味で) 同一のシンボルとして読み戻されます。

メモ: 「インターン化されていないシンボル」として知られている、write/read 不変原則を破る値を持つ処理系もあります。またこれは 2 つのシンボルの名前が同じ綴りの場合に限り同じであるという規則にも違反するものです。この報告書ではそのような処理系依存の拡張の動作は規定されていません。

(symbol? *obj*) 手続き

*obj*がシンボルであれば #t を返し、そうでなければ #f を返します。

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

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

引数がすべてシンボルであり、その名前がすべて (string=? の意味で) 同じであれば #t を返します。

メモ: 上記の定義はどの引数もインターン化されていないシンボルではないという想定に基づいています。

(symbol->string *symbol*) 手続き

*symbol*の名前を文字列として返します。ただしエスケープは行われていません。この手続きが返した文字列に string-set! のような変更手続きを適用するのはエラーです。

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

(string->symbol *string*) 手続き

名前が *string*であるシンボルを返します。この手続きにより、書き出すときにエスケープが必要な特殊な文字を含む名前のシンボルを作ることができますが、入力中のエスケープは解釈されません。

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

6.6. 文字

文字はアルファベットや数字のような印刷文字を表現するオブジェクトです。Scheme の処理系はすべて少なくとも ASCII 文字のレパートリーをサポートしていなければなりません。つまり Unicode 文字の U+0000〜U+007F です。処理系は他の好きな Unicode 文字をサポートしていても構いませんし、非 Unicode 文字をサポートしていても構いません。特に規定のない限り、以下の手続きに非 Unicode 文字を適用した結果は処理系依存です。

文字は #\⟨character⟩ または #\⟨character name⟩ または #\x⟨hex scalar value⟩ の記法を用いて書かれます。

以下の文字名は記載された値と共にすべての処理系でサポートされていなければなりません。処理系は他の名前を追加し

ても構いませんが、x を前置した hex scalar value として解釈可能でないものに限ります。

```
#\alarm      ; U+0007
#\backspace  ; U+0008
#\delete     ; U+007F
#\escape     ; U+001B
#\newline    ; 改行文字, U+000A
#\null       ; 空文字, U+0000
#\return     ; 復帰文字, U+000D
#\space      ; 空白を書く望ましい方法
#\tab        ; タブ文字, U+0009
```

以下に追加の例を示します。

```
#\a      ; 小文字
#\A      ; 大文字
#\(      ; 開き括弧
#\       ; 空白文字
#\x03BB  ; λ (その文字がサポートされている場合)
#\iota   ; ι (その文字と名前がサポートされている場合)
```

#\⟨character⟩ および #\⟨character name⟩ では大文字小文字は区別されますが、#\x⟨hex scalar value⟩ では区別されません。#\⟨character⟩ 内の ⟨character⟩ がアルファベットの場合、⟨character⟩ の直後の文字が識別子に使われるものであってはなりません。この規則は曖昧なケースを解決するためのものです。曖昧なケースというのは、例えば "#\space" という文字の並びは、空白文字の表現とも、文字表現 "#\s" にシンボルの表現 "pace" が続いたものとも取れます。

#\ 記法で書かれた文字はそれ自身に評価されます。つまりプログラム中で quote する必要はありません。

文字を操作する手続きには大文字小文字の違いを無視するものがあります。大文字小文字を無視する手続きは名前に "-ci" ("case insensitive"(大文字小文字を区別しない) の略) が入っています。

(char? *obj*) 手続き

*obj*が文字であれば #t を返し、そうでなければ #f を返します。

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

これらの手続きは引数を char->integer に渡した結果がそれぞれ等しい、単調に増加している、単調に減少している、単調に減少していない、単調に増加していない場合に #t を返します。

これらの述語は推移的であることが要求されます。

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

これらの手続きは char=? 等に似ていますが、大文字小文字を同一視する点が異なります。例えば (char-ci=? #\A #\a) は #t を返します。

具体的にはこれらの手続きは比較前に引数に char-foldcase を適用したかのように動作します。

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

これらの手続きは引数がそれぞれアルファベットである、数字である、ホワイトスペースである、大文字である、小文字である場合に #t を返し、そうでなければ #f を返します。

具体的にはこれらはそれぞれ Unicode のプロパティである Alphabetic、Numeric_Digit、White_Space、Uppercase、Lowercase を持つ文字に適用すると #t を返し、そうでない Unicode 文字に適用すると #f を返します。アルファベットでありながら大文字でも小文字でもない文字が Unicode にはたくさんあることに注意してください。

(digit-value *char*) char ライブラリの手続き

この手続きは引数が数字 (つまり char-numeric? が #t を返す) の場合、その引数の数値 (0〜9) を返し、それ以外の文字に対しては #f を返します。

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

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

char->integer に Unicode 文字を与えると、その文字の Unicode スカラー値と等しい 0〜#xD7FF または #xE000〜#x10FFFF の正確な整数を返します。非 Unicode 文字を与えると、#x10FFFF より大きい正確な整数を返します。処理系が内部的に Unicode 表現を用いているか否かに関係なくこのように動作します。

ある文字に char->integer を適用して返された正確な整数を integer->char に与えるとその文字を返します。

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

char-upcase 手続きは Unicode 大文字小文字ペアの小文字を引数に与えるとその大文字を返します。ただしその Scheme 処理系がその両方の文字をサポートしている場合に限ります。言語固有の大文字小文字ペアは用いられないことに注意してください。引数がそのようなペアの小文字でなかった場合は引数がそのまま返されます。

char-downcase 手続きは Unicode 大文字小文字ペアの大文字を引数に与えるとその小文字を返します。ただしその Scheme 処理系がその両方の文字をサポートしている場合に限ります。言語固有の大文字小文字ペアは用いられないことに注意してください。引数がそのようなペアの大文字でなかった場合は引数がそのまま返されます。

char-foldcase 手続きは引数に対して Unicode の単純な大文字小文字畳み込みアルゴリズムを適用し、その結果を返します。言語固有の畳み込みは用いられないことに注意してください。引数が大文字の場合、結果は小文字であるか、その小文字が存在しないまたは処理系がサポートしていない場合は引数がそのまま返されます。詳細は UAX #29 [11] (Unicode 標準の一部) を参照してください。

対応する大文字がない小文字が Unicode にはたくさんあることに注意してください。

6.7. 文字列

文字列は文字の並びです。文字列はダブルクォート (") で囲まれた文字の並びとして書かれます。文字列リテラル内では、様々なエスケープシーケンスが特別に解釈されます。エスケープシーケンスは必ずバックスラッシュ(\) で始まります。

- \a : アラーム, U+0007
- \b : バックペース, U+0008
- \t : タブ, U+0009
- \n : 改行, U+000A
- \r : 復帰, U+000D
- \" : ダブルクォート, U+0022
- \\ : バックスラッシュ, U+005C
- \| : 垂直線, U+007C
- \⟨intraline whitespace⟩*⟨line ending⟩ ⟨intraline whitespace⟩* : 無
- \x⟨hex scalar value⟩; : 指定された文字 (最後にセミコロンが付いていることに注意)

文字列中でバックスラッシュの後にこれ以外の文字が続いた場合の結果は規定されていません。

文字列リテラル中のエスケープシーケンス外の任意の文字は、改行を除き、それ自身を表します。\⟨intraline whitespace⟩ に改行が続いたものは (後続の intraline whitespace も含めて) 無に展開されるので、可読性向上のため文字列をインデントするのに使うことができます。それ以外の改行はすべて文字列中に \n 文字を入れるのと同じ効果を持ちます。

例を挙げます。

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

文字列に含まれる文字の数をその文字列の**長さ**と言います。この数値は正確な非負の整数で文字列の作成時に固定されます。文字列の長さ未満の正確な非負の整数をその文字列の**有効なインデックス**と言います。文字列の最初の文字のインデックスは 0 で、二番目の文字のインデックスは 1 で、以下同様です。

文字列を操作する手続きには大文字小文字の違いを無視するものがあります。大文字小文字を無視するバージョンは名前の最後に "-ci" ("case insensitive"(大文字小文字を区別しない) の略) が付いています。

処理系は文字列中に特定の文字が現れることを禁止しても構いません。ただし #\null 以外の ASCII 文字を禁止してはなりません。例えば、Unicode のレパートリー全体をサポートするけれども文字列中では U+0001〜U+00FF (#\null を除く Latin-1 のレパートリー) しか使えない処理系などが有り得ます。

そのような禁止文字を make-string、string、string-set!、string-fill! に渡したり、リストの一部として list->string に渡したり、ベクタの一部として vector->string (6.8 節を参照) に渡したり、バイトベクタ内に UTF-8 エンコードされた形で utf8->string (6.9 節を参照) に渡すことはエラーです。また string-map (6.10 節を参照) に渡した手続きが禁止文字を返したり、read-string (6.13.2 節を参照) に禁止文字を読ませようと試みることもエラーです。

(string? *obj*) 手続き

*obj*が文字列であれば #t を返し、そうでなければ #f を返します。

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

make-string 手続きは新しく割り当てられた長さ *k* の文字列を返します。*char*が与えられた場合はその文字列のすべての文字が *char*で初期化されます。そうでなければ文字列の内容は規定されていません。

(string *char* ...) 手続き

引数から成る新しく割り当てられた文字列を返します。これは list の文字列版です。

(string-length *string*) 手続き

与えられた *string*の文字数を返します。

(string-ref *string k*) 手続き

k が *string*の有効なインデックスでなければエラーです。

string-ref 手続きは *string*の *k* 番目の文字を返します。インデックスはゼロから始まります。 この手続きを定数時間で実行することは要求されていません。

(string-set! *string k char*) 手続き

k が *string*の有効なインデックスでなければエラーです。

string-set! 手続きは *string*の *k* 番目の要素に *char*を格納します。この手続きを定数時間で実行することは要求されていません。

```
(define (f) (make-string 3 #\*))
(define (g) "***")
(string-set! (f) 0 #\?)      ⟹  規定されていない
(string-set! (g) 0 #\?)      ⟹  エラー
(string-set! (symbol->string 'immutable)
             0
             #\?)            ⟹  エラー
```

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

すべての文字列が同じ長さで正確に同じ文字を同じ位置に持つ場合 #t を返し、そうでなければ #f を返します。

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

大文字小文字畳み込みの後、すべての文字列が同じ長さで同じ位置に同じ文字を持つ場合 #t を返し、そうでなければ #f を返します。具体的にはこれらの手続きは比較前に引数に string-foldcase を適用したかのように動作します。

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

これらの手続きは引数がそれぞれ単調に増加している、単調に減少している、単調に増加していない、単調に減少していない場合に #t を返します。

これらの述語は推移的であることが要求されます。

これらの手続きは処理系定義の方法で文字列を比較します。ひとつの方法として文字に対する順序付けを文字列に辞書的に拡張することが考えられます。その場合 string<? は文字に対する char<? の順序付けによって文字列に対する辞書的な順序付けを行うことになるでしょう。また 2 つの文字列の長さが異なるものの短い方の文字列の長さまでは同じ内容の場合、短い方の文字列は長い方の文字列よりも辞書的に小さいと考えられます。しかし処理系の文字列の内部表現による自然な順序付けや、より複雑なロケール固有の順序付けを用いることも許容されます。

いずれの場合でも、一組の文字列に対して string<?、string=?、string>? のうちひとつだけが満たされなければならず、string>? が満たされない場合に限り string<=? が満たされなければならず、string<? が満たされない場合に限り string>=? が満たされなければなりません。

"-ci" 付き手続きは、対応する "-ci" 無しの手続きを呼ぶ前に引数に string-foldcase を適用したかのように動作します。

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

これらの手続きは引数に対して Unicode の完全な大文字小文字変換アルゴリズムを適用し、その結果を返します。場合によっては結果の長さが引数と異なることもあります。結果が引数と string=? の意味で同じ場合は引数をそのまま返しても構いません。ちなみに言語固有のマッピングおよび畳み込みは用いられません。

Unicode 標準ではギリシア文字の Σ に特別な扱いが規定されており、通常の小文字形は σ ですが、単語の終わりに来た場合は ς になります。詳細は UAX #29 [11] (Unicode 標準の一部) を参照してください。しかし string-downcase の実装にこの動作を行うことは要求されません。すべての場合において Σ を σ に変換しても構いません。

(substring *string start end*) 手続き

substring 手続きは *string*内のインデックス *start*で始まりインデックス *end*で終わる文字から成る新しく割り当てられた文字列を返します。これは同じ引数で string-copy を呼ぶのと同等ですが、後方互換性およびスタイル上の柔軟性のために提供されています。

(string-append *string* ...) 手続き

与えられた文字列の文字を連結した文字を持つ新しく割り当てられた文字列を返します。

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

*list*の要素のいずれかが文字でなければエラーです。

string->list 手続きは *string*の *start*〜*end*の文字の新しく割り当てられたリストを返します。list->string はリスト *list*内の要素から成る新しく割り当てられた文字列を返します。どちらの手続きでも順番は維持されます。equal? の意味において string->list および list->string はお互い逆関数です。

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

与えられた *string*の *start*〜*end*の部分の新しく割り当てられたコピーを返します。

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

*at*がゼロより小さいか *to*の長さより大きい場合はエラーです。(- (string-length *to*) *at*) が (- *end start*) より小さい場合もエラーです。

文字列 *from*の *start*〜*end*の文字を、文字列 *to*の *at*から始まる位置にコピーします。文字がコピーされる順番は規定されていません。ただしコピー元とコピー先が重なっている場合は、コピー元がいったん一時的な文字列にコピーされ、それからコピー先にコピーされたかのように動作します。正しい方向でコピーを行うように気を付ければそのような状況でも領域を割り当てることなくこれを行うことができます。

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

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

*fill*が文字でなければエラーです。

string-fill! 手続きは *string*の *start*〜*end*の要素に *fill*を格納します。

6.8. ベクタ

ベクタは要素を整数でインデックスする異種混合の構造体です。ベクタは一般的に同じ長さのリストよりも小さな空間しか使用せず、一般的にランダムに選んだ要素のアクセスに必要な平均時間がリストよりもベクタの方が小さくて済みます。

ベクタが持つ要素の数をそのベクタの**長さ**と言います。この数は非負の整数でベクタの作成時に固定されます。ベクタの長さよりも小さい正確な非負の整数をそのベクタの**有効なインデックス**と言います。ベクタの最初の要素のインデックスは 0 で、最後の要素のインデックスはベクタの長さよりも 1 小さい値です。

ベクタは #(*obj* ...) という表記を用いて書きます。例えば 0 番目の要素に数値のゼロ、1 番目の要素にリスト (2 2 2 2)、2 番目の要素に文字列 "Anna" を持つ長さ 3 のベクタは以下のように書くことができます。

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

ベクタ定数はそれ自身に評価されます。そのためプログラム中で quote する必要はありません。

(vector? *obj*) 手続き

*obj*がベクタであれば #t を返し、そうでなければ #f を返します。

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

*k*個の要素を持つ新しく割り当てられたベクタを返します。第 2 引数が与えられた場合、各要素は *fill*に初期化されます。そうでなければ各要素の初期内容は規定されていません。

(vector *obj* ...) 手続き

与えられた引数を要素に持つ新しく割り当てられたベクタを返します。これは list のベクタ版です。

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

(vector-length *vector*) 手続き

*vector*の要素の数を正確な整数として返します。

(vector-ref *vector k*) 手続き

k が *vector*の有効なインデックスでなければエラーです。

vector-ref 手続きは *vector*の *k* 番目の要素の内容を返します。

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

(vector-set! *vector k obj*) 手続き

k が *vector*の有効なインデックスでなければエラーです。

vector-set! 手続きは *vector*の *k* 番目の要素に *obj*を格納します。

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

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

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

vector->list 手続きは *vector*の *start*〜*end*の要素に格納されているオブジェクトから成る新しく割り当てられたリストを返します。list->vector 手続きはリスト *list*の要素に初期化されている新しく割り当てられたベクタを返します。

どちらの手続きでも順番は維持されます。

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

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

*vector*の *start*〜*end*の要素のいずれかが文字でなければエラーです。

vector->string 手続きは *vector*の *start*〜*end*の要素に格納されているオブジェクトから成る新しく割り当てられた文字列を返します。string->vector 手続きは文字列 *string* の *start*〜*end*の要素に初期化されている新しく割り当てられたベクタを返します。

どちらの手続きでも順番は維持されます。

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

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

与えられた *vector*の *start*〜*end*の要素の新しく割り当てられたコピーを返します。新しいベクタの要素は古いベクタの要素と (eqv? の意味で) 同じです。

```
(define a #(1 8 2 8)) ; a may be immutable
(define b (vector-copy a))
(vector-set! b 0 3)   ; b is mutable
b                           ⟹ #(3 8 2 8)
(define c (vector-copy b 1 3))
c                           ⟹ #(8 2)
```

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

*at*がゼロより小さいか *to*の長さよりも大きい場合はエラーです。(- (vector-length *to*) *at*) が (- *end start*) より小さい場合もエラーです。

ベクタ *from*の *start*〜*end*の要素をベクタ *to*の *at*から始まる位置にコピーします。要素がコピーされる順番は規定されていません。ただしコピー元とコピー先が重なっている場合は、コピー元がいったん一時的なベクタにコピーされ、それからコピー先にコピーされたかのように動作します。正しい方向でコピーを行うように気を付ければそのような状況でも領域を割り当てることなくこれを行うことができます。

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

(vector-append *vector* ...) 手続き

与えられたベクタの要素を連結した要素を持つ新しく割り当てられたベクタを返します。

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

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

vector-fill! 手続きは *vector*の *start*〜*end*の要素に *fill*を格納します。

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

6.9. バイトベクタ

バイトベクタはバイナリデータの塊を表します。これは固定サイズのバイトの並びです。**バイト**とは、0〜255(両端を含む) の範囲の正確な整数です。バイトベクタは一般的に同じ値を持つベクタよりも高い空間効率を持ちます。

バイトベクタが持つ要素の数をそのバイトベクタの**長さ**と言います。この数は非負の整数で、バイトベクタ作成時に固定されます。バイトベクタの長さより小さい正確な非負の整

数を、そのバイトベクタの**有効なインデックス**と言います。ベクタ同様にインデックスはゼロから始まります。

バイトベクタは `#u8(`*byte* ...`)` という表記を用いて書きます。例えば 0 番目の要素にバイト 0、1 番目の要素にバイト 10、2 番目の要素にバイト 5 を持つ長さ 3 のバイトベクタは以下のように書くことができます。

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

バイトベクタ定数はそれ自身に評価されます。そのためプログラム中で quote する必要はありません。

`(bytevector?` *obj*`)` 手続き

*obj*がバイトベクタであれば `#t` を返します。そうでなければ `#f` を返します。

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

`make-bytevector` 手続きは長さ *k* の新しく割り当てられたバイトベクタを返します。*byte*が与えられた場合は、そのバイトベクタのすべての要素が *byte*に初期化されます。そうでなければ各要素の内容は規定されていません。

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

`(bytevector` *byte* ...`)` 手続き

引数を持つ新しく割り当てられたバイトベクタを返します。

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

`(bytevector-length` *bytevector*`)` 手続き

*bytevector*のバイト単位の長さを正確な整数として返します。

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

k が *bytevector*の有効なインデックスでなければエラーです。

*bytevector*の *k*番目のバイトを返します。

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

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

k が *bytevector*の有効なインデックスでなければエラーです。

*bytevector*の *k*番目のバイトとして *byte*を格納します。

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

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

*bytevector*の *start*〜*end*のバイトを持つ新しく割り当てられたバイトベクタを返します。

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

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

*at*がゼロより小さいか *to*の長さよりも大きい場合はエラーです。`(-` `(bytevector-length` *to*`)` *at*`)` が `(-` *end start*`)` より小さい場合もエラーです。

バイトベクタ *from*の *start*〜*end*のバイトをバイトベクタ *to* の *at*から始まる位置にコピーします。バイトがコピーされる順番は規定されていません。ただしコピー元とコピー先が重なっている場合は、コピー元がいったん一時的なバイトベクタにコピーされ、それからコピー先にコピーされたかのように動作します。正しい方向でコピーを行うように気を付ければそのような状況でも領域を割り当てることなくこれを行うことができます。

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

メモ: この手続きは R^{6}RS にもありましたが、Scheme の他の同様の手続きとは逆に、コピー元をコピー先より先に指定するようになっていました。

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

与えられたバイトベクタの要素を連結した要素を持つ新しく割り当てられたバイトベクタを返します。

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

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

*bytevector*が無効な UTF-8 バイトシーケンスを含んでいる場合はエラーです。

これらの手続きは文字列とその文字列を UTF-8 エンコーディングでエンコードしたバイトベクタとの間で変換を行います。`utf8->string` 手続きはバイトベクタの *start*〜*end*のバイトをデコードし、対応する文字列を返します。`string->utf8` 手続きは文字列の *start*〜*end*の文字をエンコードし、対応するバイトベクタを返します。

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

6.10. 制御機能

この節ではプログラム実行の流れを特殊な方法で制御する様々なプリミティブ手続きについて述べます。手続き引数を呼び出すこの節の手続きは必ず元の手続き呼び出しと同じ動的環境でそれを呼び出します。procedure? 述語もここで述べます。

(procedure? *obj*) 手続き

*obj*が手続きであれば #t を返し、そうでなければ #f を返します。

```
(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

apply 手続きはリスト (append (list arg_1 ...) *args*) の要素を実引数として *proc*を呼び出します。

```
(apply + (list 3 4))        ⟹  7

(define compose
  (lambda (f g)
    (lambda args
      (f (apply g args)))))

((compose sqrt *) 12 75)    ⟹  30
```

(map *proc* $list_1$ $list_2$ ...) 手続き

*proc*が *list* の数と同じ数の引数を取らない場合、および単一の値を返さない場合はエラーです。

map 手続きは *list*の要素ごとに *proc*を適用し、その結果の順番通りのリストを返します。*list*が 2 つ以上与えられ、長さの異なるリストがある場合、map は最も短いリストを使い切った時点で終了します。*list*には循環リストも使えますが、すべてのリストが循環リストであった場合はエラーです。*proc*がリストのいずれかを変更した場合はエラーです。*list*の要素に *proc*が適用される動的な順番は規定されていません。map からの戻りが複数回発生した場合、先に返された値が変更されることはありません。

```
(map cadr '((a b) (d e) (g h)))
          ⟹  (b e h)

(map (lambda (n) (expt n n))
     '(1 2 3 4 5))
          ⟹  (1 4 27 256 3125)

(map + '(1 2 3) '(4 5 6 7)) ⟹  (5 7 9)

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

(string-map *proc* $string_1$ $string_2$ ...) 手続き

*proc*が *string* の数と同じ数の引数を取らない場合、および単一の値を返さない場合はエラーです。

string-map 手続きは *string*の要素ごとに *proc*を適用し、その結果の順番通りの文字列を返します。*string*が 2 つ以上与えられ、長さの異なる文字列がある場合、string-map は最も短い文字列を使い切った時点で終了します。*string*の要素に *proc*が適用される動的な順番は規定されていません。string-map からの戻りが複数回発生した場合、先に返された値が変更されることはありません。

```
(string-map char-foldcase "AbdEgH")
          ⟹  "abdegh"

(string-map
 (lambda (c)
   (integer->char (+ 1 (char->integer c))))
 "HAL")
          ⟹  "IBM"

(string-map
 (lambda (c k)
   ((if (eqv? k #\u) char-upcase char-downcase)
    c))
 "studlycaps xxx"
 "ululululul")
          ⟹   "StUdLyCaPs"
```

(vector-map *proc* $vector_1$ $vector_2$ ...) 手続き

*proc*が *vector* の数と同じ数の引数を取らない場合、および単一の値を返さない場合はエラーです。

vector-map 手続きは *vector*の要素ごとに *proc*を適用し、その結果の順番通りのベクタを返します。*vector*が 2 つ以上与えられ、長さの異なるベクタがある場合、vector-map は最も短いリストを使い切った時点で終了します。*vector*の要素に *proc*が適用される動的な順番は規定されていません。vector-map からの戻りが複数回発生した場合、先に返された値が変更されることはありません。

```
(vector-map cadr '#((a b) (d e) (g h)))
          ⟹  #(b e h)

(vector-map (lambda (n) (expt n n))
            '#(1 2 3 4 5))
          ⟹  #(1 4 27 256 3125)

(vector-map + '#(1 2 3) '#(4 5 6 7))
          ⟹  #(5 7 9)
```

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

(for-each *proc* $list_1$ $list_2$ ...) 手続き

*proc*が *list* の数と同じ数の引数を取らない場合はエラーです。

for-each の引数は map の引数と同様ですが、for-each では値のためではなく副作用のために *proc*が呼ばれます。map と異なり、for-each では最初の要素から最後の要素まで順番通りに *list*の要素に対して *proc*が呼ばれることが保証されています。for-each の戻り値は規定されていません。*list*が 2 つ以上与えられ、長さの異なるリストがある場合、for-each は最も短いリストを使い切った時点で終了します。*list*には循環リストも使えますが、すべてのリストが循環リストであった場合はエラーです。

*proc*がリストのいずれかを変更した場合はエラーです。

```
(let ((v (make-vector 5)))
  (for-each (lambda (i)
              (vector-set! v i (* i i)))
            '(0 1 2 3 4))
  v)                        ⟹  #(0 1 4 9 16)
```

(string-for-each *proc* $string_1$ $string_2$ ...) 手続き

*proc*が *string* の数と同じ数の引数を取らない場合はエラーです。

string-for-each の引数は string-map の引数と同様ですが、string-for-each では値のためではなく副作用のために *proc*が呼ばれます。string-map と異なり、string-for-each では最初の要素から最後の要素まで順番通りに *list* の要素に対して *proc*が呼ばれることが保証されています。string-for-each の戻り値は規定されていません。*string*が 2 つ以上与えられ、長さの異なる文字列がある場合、string-for-each は最も短いリストを使い切った時点で終了します。*proc*が文字列のいずれかを変更した場合はエラーです。

```
(let ((v '()))
  (string-for-each
   (lambda (c) (set! v (cons (char->integer c) v)))
   "abcde")
  v)                        ⟹  (101 100 99 98 97)
```

(vector-for-each *proc* $vector_1$ $vector_2$ ...) 手続き

*proc*が *vector* の数と同じ数の引数を取らない場合はエラーです。

vector-for-each の引数は vector-map の引数と同様ですが、vector-for-each では値のためではなく副作用のために *proc*が呼ばれます。vector-map と異なり、vector-for-each では最初の要素から最後の要素まで順番通りに *vector* の要素に対して *proc*が呼ばれることが保証されています。vector-for-each の戻り値は規定されていません。*vector*が 2 つ以上与えられ、長さの異なるベクタがある場合、vector-for-each は最も短いベクタを使い切った時点で終了します。*proc*がベクタのいずれかを変更した場合はエラーです。

```
(let ((v (make-list 5)))
  (vector-for-each
   (lambda (i) (list-set! v i (* i i)))
   '#(0 1 2 3 4))
  v)                        ⟹  (0 1 4 9 16)
```

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

*proc*が引数をひとつ取らない場合はエラーです。

手続き call-with-current-continuation (または同等の省略形 call/cc) は現在の継続 (後述の論拠を参照) を「脱出手続き」としてパッケージ化し、それを引数として *proc* に渡します。脱出手続きは Scheme の手続きで、後ほど呼ばれるとその時点での有効な継続を放棄し、代わりに脱出手続き作成時点で有効であった継続を使用します。脱出手続きを呼び出すと dynamic-wind を用いてインストールされた *before*サンクおよび *after*サンクが呼び出されます。

脱出手続きは call-with-current-continuation の呼び出し元の継続と同じ数の引数を取ります。ほとんどの継続は値をひとつだけ取ります。call-with-values 手続き (define-values、let-values および let*-values 式の初期化式も含む) によって作成された継続は、その消費者が期待している数の値を取ります。lambda、case-lambda、begin、let、let*、letrec、letrec*、let-values、let*-values、let-syntax、letrec-syntax、parameterize、guard、case、cond、when および unless 式などにおける、式の並びの中にある最後でない式の継続はすべて、任意の数の値を取ります。渡された値が何であれ破棄するだけだからです。これらのいずれかの方法によって作成されたものでない継続にゼロ個の値を渡したり 2 つ以上の値を渡した場合の効果は規定されていません。

*proc*に渡される脱出手続きは Scheme の他のどんな手続きとも同様に無制限の生存期間を持ちます。変数やデータ構造に格納することができ、好きな回数だけ呼ぶことができます。しかし raise や error 手続き同様、呼び出し元に返ることはありません。

以下の例は call-with-current-continuation の最も単純な用途のみを示しています。実際の用途がすべてこの例のように単純であれば call-with-current-continuation のような強力な手続きは必要ないでしょう。

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                    ⟹  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
```

```
(letrec ((r
          (lambda (obj)
            (cond ((null? obj) 0)
                  ((pair? obj)
                   (+ (r (cdr obj)) 1))
                  (else (return #f))))))
  (r obj))))))

(list-length '(1 2 3 4))    ⟹  4

(list-length '(a b . c))    ⟹  #f
```

論拠:

call-with-current-continuation の良く有る使い方は、ループや手続き本体からの構造化された非局所的な脱出です。しかし call-with-current-continuation は幅広い様々な高度な制御構造を実装する役に立ちます。実際 raise および guard は非局所的脱出のためのより構造化された仕組みです。

Scheme の式が評価されるときは必ずその結果を欲している**継続**が存在しています。継続はその計算の (デフォルトの) 未来全体を表しています。例えば式を REPL で評価する場合、その継続は、結果を受け取り、それを画面に出力し、次の入力のためのプロンプトを表示し、それを評価し、以下同様に永遠に繰り返すというものです。ほとんどの場合、継続はユーザーコードによって指定されたアクションを含みます。結果を受け取り、ある局所変数に格納された値をそれに掛け、7 を加え、そしてその答えを出力するために REPL の継続に引き渡す、のような感じです。通常これらの普遍的な継続は水面下に隠されており、プログラマーはこれについて深く考えたりしません。しかしプログラマーが明示的に継続を扱わなければならない状況も稀にあります。call-with-current-continuation 手続きが現在の継続として動作する手続きを作成することにより、Scheme のプログラマーにはそれが可能となります。

(values *obj* ...) 手続き

引数をすべて継続に渡します。values 手続きは以下のように定義することができます。

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

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

*producer*が引数無しで呼ばれ、その *producer*の呼び出し元の継続に値が渡されると、その値を引数として *consumer* 手続きが呼ばれます。*consumer*の呼び出し元の継続は call-with-values の呼び出し元の継続です。

```
(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*が引数無しで呼ばれ、その呼び出しの結果を返します。*before*および *after*も以下の規則に従って引数無しで呼ばれます。ちなみに call-with-current-continuation を用いて捕捉された継続への呼び出しが無ければ、3 つの引数は順番にそれぞれ一度だけ呼ばれます。*thunk*の呼び出しの動的生存期間に入るときは必ず *before*が呼ばれ、その動的生存期間を出るときは必ず *after*が呼ばれます。手続き呼び出しの動的生存期間とは、その呼び出しが開始されてから戻るまでの間の期間です。*before*サンクおよび *after*サンクは dynamic-wind の呼び出し元と同じ動的環境で呼ばれます。Scheme には call-with-current-continuation が存在するため、呼び出しの動的生存期間は連続した単一の時間でない場合があります。これは以下のように定義されます。

- 呼ばれた手続きの本体の実行が開始されるとき、その動的生存期間に入ります。
- 動的生存期間中に (call-with-current-continuation を用いて) 捕捉された継続をその動的生存期間外で呼んだときもその動的生存期間に入ります。
- 呼ばれた手続きから戻るとき、その動的生存期間から抜けます。
- 動的生存期間外で捕捉された継続をその動的生存期間中に呼んだときもその動的生存期間から抜けます。

*thunk*の呼び出しの動的生存期間中に 2 回目の dynamic-wind の呼び出しが発生し、何らかの継続が呼び出されてこれら 2 回の dynamic-wind 呼び出しの *after*を両方とも呼ぶべき状況になった場合は、2 回目の (内側の)dynamic-wind 呼び出しに関連付けられている *after*が先に呼ばれます。

*thunk*の呼び出しの動的生存期間中に 2 回目の dynamic-wind の呼び出しが発生し、何らかの継続が呼び出されてこれら 2 回の dynamic-wind 呼び出しの *before*を両方とも呼ぶべき状況になった場合は、1 回目の (外側の)dynamic-wind 呼び出しに関連付けられている *before*が先に呼ばれます。

継続が呼び出されたことにより、ある dynamic-wind 呼び出しの *before*と別の dynamic-wind 呼び出しの *after*を呼ぶ必要が生じた場合は、*after*の方が先に呼ばれます。

捕捉した継続を用いて *before*や *after*の呼び出しの動的生存期間を出入りした場合の効果は規定されていません。

```
(let ((path '())
      (c #f))
  (let ((add (lambda (s)
               (set! path (cons s path)))))
    (dynamic-wind
      (lambda () (add 'connect))
      (lambda ()
        (add (call-with-current-continuation
               (lambda (c0)
                 (set! c c0)
                 'talk1))))
      (lambda () (add 'disconnect)))
    (if (< (length path) 4)
        (c 'talk2)
        (reverse path))))

        ⟹ (connect talk1 disconnect
            connect talk2 disconnect)
```

6.11. 例外

この節では Scheme の例外処理および例外発生手続きについて述べます。Scheme の例外の概念については 1.3.2 節を参照してください。guard 構文については 4.2.7 も参照してください。

例外的な状況が通知されたときにプログラムが取るアクションを決める手続きを**例外ハンドラ**と呼びます。例外ハンドラは引数をひとつ取ります。システムは現在の例外ハンドラを動的環境で暗黙的に管理しています。

例外が発生すると現在の例外ハンドラが呼ばれ、その例外に関する情報をカプセル化したオブジェクトが渡されます。引数をひとつ取る手続きなら何でも例外ハンドラとして用いることができます。また、どんなオブジェクトでも例外を表すために用いることができます。

(with-exception-handler *handler thunk*) 手続き

*handler*が引数をひとつ取らない場合はエラーです。また *thunk*がゼロ個の引数を取らない場合もエラーです。

with-exception-handler 手続きは *thunk*を呼び、その結果を返します。*thunk*の呼び出しに対して使われる動的環境に現在の例外ハンドラとして *handler*をインストールします。

```
(call-with-current-continuation
 (lambda (k)
  (with-exception-handler
   (lambda (x)
    (display "condition: ")
    (write x)
    (newline)
    (k 'exception))
   (lambda ()
    (+ 1 (raise 'an-error))))))
                          ⟹ exception
        そして condition: an-error  が出力される

(with-exception-handler
 (lambda (x)
  (display "something went wrong\n"))
 (lambda ()
  (+ 1 (raise 'an-error))))
           something went wrong  が出力される
```

2 番目の例では、出力の後、別の例外が発生します。

(raise *obj*) 手続き

例外を発生させ *obj*に対して現在の例外ハンドラを呼び出します。ハンドラは raise の呼び出し元と同じ動的環境で呼ばれます。ただし現在の例外ハンドラは呼ばれるハンドラがインストールされたときのものになります。ハンドラから戻った場合、そのハンドラと同じ動的環境で第二の例外が発生します。*obj*とその第二の例外の関係は規定されていません。

(raise-continuable *obj*) 手続き

例外を発生させ *obj*に対して現在の例外ハンドラを呼び出します。ハンドラは raise-continuable の呼び出し元と同じ動的環境で呼ばれます。ただし (1) 現在の例外ハンドラは呼ばれるハンドラがインストールされたときのものになり、(2) 呼ばれたハンドラから戻った場合それが再び現在の例外ハンドラになります。ハンドラから戻った場合、その戻り値が raise-continuable の戻り値になります。

```
(with-exception-handler
  (lambda (con)
    (cond
      ((string? con)
       (display con))
      (else
       (display "a warning has been issued")))
    42)
  (lambda ()
    (+ (raise-continuable "should be a number")
       23)))
    prints: should be a number
                              ⟹ 65
```

(error *message obj ...*) 手続き

*message*は文字列であるべきです。

*message*および**イリタント**として知られる *obj* によって与えられた情報をカプセル化した処理系定義の新しく割り当てられたオブジェクトに対して raise を呼んだかのように例外を発生させます。そのオブジェクトに対して手続き error-object? を呼ぶと #t を返さなければなりません。

```
(define (null-list? l)
  (cond ((pair? l) #f)
        ((null? l) #t)
        (else
          (error
            "null-list?: argument out of domain"
            l))))
```

(error-object? *obj*) 手続き

*obj*が error によって作成されたオブジェクトであれば #t を返します。何らかの処理系定義のオブジェクトに対して #t を返す場合もあります。そうでなければ #f を返します。エラーを通知するオブジェクトは、述語 file-error? や read-error? を満たすものも含め、error-object? を満たしても満たさなくても構いません。

(error-object-message *error-object*) 手続き

*error-object*にカプセル化されているメッセージを返します。

(error-object-irritants *error-object*) 手続き

*error-object*にカプセル化されているイリタントのリストを返します。

(read-error? *obj*) 手続き
(file-error? *obj*) 手続き

エラー型の述語です。それぞれ *obj* が read 手続きによって発生した場合、またはファイルの入出力ポートを開けなかったことによって発生した場合に #t を返します。そうでなければ #f を返します。

6.12. 環境と評価

(environment $list_1$ ...) eval ライブラリの手続き

この手続きは、空の環境を用意し、そこに各 *list* をインポートセットとみなしてインポートし、その結果の環境の指定子を返します。(インポートセットの説明は 5.6 節を参照。) この指定子によって表された環境の束縛は環境自身と同様に不変です。

(scheme-report-environment *version*)
r5rs ライブラリの手続き

version が 5 (R^{5}RS に対応する) と等しければ、R^{5}RS ライブラリで定義されている束縛のみを持つ環境の指定子を返します。処理系はこの値の *version* をサポートしなければなりません。

処理系は他の値の *version* をサポートしても構いません。その場合は指定されたバージョンの報告書に対応する束縛を持つ環境の指定子を返します。*version* が 5 でなく、処理系がサポートしている他の値でもない場合はエラーが通知されます。

scheme-report-environment 内で束縛されている識別子 (例えば car) を (eval の使用を通して) 定義または代入した場合の効果は規定されていません。すなわち環境とそこに含まれる束縛は両方とも不変であって構いません。

(null-environment *version*) r5rs ライブラリの手続き

version が 5 (R^{5}RS に対応する) と等しければ、R^{5}RS ライブラリで定義されているすべての構文キーワードの束縛のみを持つ環境の指定子を返します。処理系はこの値の *version* をサポートしなければなりません。

処理系は他の値の *version* をサポートしていても構いません。その場合は指定されたバージョンの報告書に対応する適切な束縛を持つ環境の指定子を返します。*version* が 5 でなく、処理系がサポートしている他の値でもない場合は、エラーが通知されます。

scheme-report-environment 内で束縛されている識別子 (例えば car) を (eval の使用を通して) 定義または代入した場合の効果は規定されていません。すなわち環境とそこに含まれる束縛は、両方とも不変であっても構いません。

(interaction-environment) repl ライブラリの手続き

この手続きは処理系定義の束縛の集合、一般的には (scheme base) からエクスポートされているもののスーパーセットを持つ変更可能な環境の指定子を返します。この手続きはユーザーが REPL に入力した式を評価する環境を返すことを意図しています。

(eval *expr-or-def environment-specifier*)
eval ライブラリの手続き

expr-or-def が式の場合、指定された環境でそれが評価され、その値が返されます。定義の場合、指定された識別子が指定された環境で定義されます。ただしその環境が不変でない場合に限ります。処理系は eval を拡張して他のオブジェクトを受け付けても構いません。

```
(eval '(* 7 3) (environment '(scheme base)))
                                ⟹  21

(let ((f (eval '(lambda (f x) (f x x))
               (null-environment 5))))
  (f + 10))
                                ⟹  20
(eval '(define foo 32)
      (environment '(scheme base)))
                                ⟹  エラーが通知される
```

6.13. 入出力

6.13.1. ポート

ポートは入出力機器を表します。Scheme では入力ポートは要求に応じてデータを供給する Scheme オブジェクトで、出力ポートはデータを消費する Scheme オブジェクトです。入力ポート型と出力ポート型が独立しているかどうかは処理系依存です。

ポート型によって操作するデータは異なります。Scheme の処理系は**テキストポート**と**バイナリポート**をサポートすることが要求されますが、他のポート型を提供していても構いません。

テキストポートは後述する read-char および write-char を用いた文字ベースのバッキングストアに対する個々の文字の読み書きをサポートします。また read や write といった文字の観点で定義される操作もサポートします。

バイナリポートは後述する read-u8 および write-u8 を用いたバイトベースのバッキングストアに対する個々のバイトの読み書きをサポートします。またバイトの観点で定義される操作も同様にサポートします。テキストポート型とバイナリポート型が独立しているかどうかは処理系依存です。

ポートは Scheme のプログラムを実行しているホストシステム上のファイルやデバイスなどにアクセスするために使うことができます。

(call-with-port *port proc*) 手続き

proc が引数をひとつ取らない場合はエラーです。

`call-with-port` 手続きは *port*を引数として *proc*を呼びます。*proc*から戻ると、そのポートは自動的に閉じられ、*proc*の生成した値が返されます。*proc*から戻らない場合、そのポートが今後読み書き操作に使われることがないと保証できない限り、自動的に閉じられてはなりません。

論拠: Scheme の脱出手続きは無制限の生存期間を持つため、現在の継続から脱出して後に再開することが可能です。現在の継続から脱出した際にポートを閉じることを認めた場合、`call-with-current-continuation` と `call-with-port` を両方用いた移植性のあるコードを書くことが不可能になってしまいます。

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

*proc*が引数をひとつ取らない場合はエラーです。

これらの手続きは `open-input-file` または `open-output-file` を用いたかのように指定された名前のファイルを入力用または出力用に開き、テキストポートを取得します。そしてそのポートと *proc*が `call-with-port` と同等の手続きに渡されます。

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

これらの手続きは *obj*がそれぞれ入力ポートである、出力ポートである、テキストポートである、バイナリポートである、任意の種類のポートである場合に `#t` を返します。そうでなければ `#f` を返します。

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

*port*がまだ開かれていて入出力が可能であれば `#t` を返します。 そうでなければ `#f` を返します。

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

それぞれ現在のデフォルトの入力ポート、出力ポート、エラーポート (出力ポートの一種) を返します。これらの手続きはパラメータオブジェクトであり、`parameterize` (4.2.6 節を参照) でオーバーライドできます。これらに対する束縛の初期値は処理系定義のテキストポートです。

(`with-input-from-file` *string thunk*) file ライブラリの手続き
(`with-output-to-file` *string thunk*) file ライブラリの手続き

`open-input-file` または `open-output-file` を用いたかのように入力用または出力用にファイルが開かれ、その新しいポートが `current-input-port` または `current-output-port` (`(read)`、`(write` *obj*`)` などでも使われます) から返されるようにします。その後 *thunk*が引数無しで呼ばれます。*thunk*が戻るとそのポートは閉じられ、以前のデフォルト値が復元されます。*thunk*がゼロ個の引数を取らない場合はエラーです。どちらの手続きでも *thunk*の生成した値が返されます。脱出手続きを用いてこれらの手続きから脱出した場合、現在の入出力ポートは `parameterize` で動的束縛されているかのように動作します。

(`open-input-file` *string*) file ライブラリの手続き
(`open-binary-input-file` *string*) file ライブラリの手続き

既存のファイルを表す *string*を取り、そのファイルからデータを供給することができるテキスト入力ポートまたはバイナリ入力ポートを返します。そのファイルが存在しない、または開くことができなかった場合は、`file-error?` を満たすエラーが通知されます。

(`open-output-file` *string*) file ライブラリの手続き
(`open-binary-output-file` *string*) file ライブラリの手続き

作成される出力ファイルの名前となる *string*を取り、その名前の新しいファイルにデータを書き出すことができるテキスト出力ポートまたはバイナリ出力ポートを返します。 与えられた名前のファイルがすでに存在していた場合の効果は規定されていません。ファイルを開くことができなかった場合は `file-error?` を満たすエラーが通知されます。

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

*port*に関連付けられているリソースを閉じ、*port*を読み書きできないようにします。最後の 2 つの手続きをそれぞれ入力ポートでないポート、出力ポートでないポートに適用した場合はエラーです。処理系はソケットのように入力ポートであると同時に出力ポートでもあるようなポートを提供していても構いません。`close-input-port` および `close-output-port` 手続きを使うとそのようなポートの入力側と出力側を個別に閉じることができます。

ポートがすでに閉じられている場合、これらのルーチンは何の効果もありません。

(`open-input-string` *string*) 手続き

文字列を取り、その文字列から文字を供給するテキスト入力ポートを返します。その文字列が変更された場合の効果は規定されていません。

(`open-output-string`) 手続き

`get-output-string` によって取得するために文字を蓄積するテキスト出力ポートを返します。

(get-output-string *port*) 手続き

*port*が open-output-string を用いて作成したものでない場合はエラーです。

それまでにポートに出力された文字から成る文字列を返します。文字はそれらが出力された順番に格納されます。結果の文字列が変更された場合の効果は規定されていません。

```
(parameterize
    ((current-output-port
      (open-output-string)))
    (display "piece")
    (display " by piece ")
    (display "by piece.")
    (newline)
    (get-output-string (current-output-port)))

        ⟹ "piece by piece by piece.\n"
```

(open-input-bytevector *bytevector*) 手続き

バイトベクタを取り、そのバイトベクタからバイトを供給するバイナリ入力ポートを返します。

(open-output-bytevector) 手続き

get-output-bytevector で取得するためにバイトを蓄積するバイナリ出力ポートを返します。

(get-output-bytevector *port*) 手続き

*port*が open-output-bytevector を用いて作成したものでない場合はエラーです。

それまでにポートに出力されたバイトから成るバイトベクタを返します。バイトはそれらが出力された順番に格納されます。

6.13.2. 入力

入力手続きで *port*が省略された場合は (current-input-port) の戻り値がデフォルト値となります。閉じられたポートに対して入力操作を試みることはエラーです。

(read) read ライブラリの手続き
(read *port*) read ライブラリの手続き

read 手続きは Scheme のオブジェクトの外部表現をそのオブジェクト自身に変換します。つまりこれは非終端記号 ⟨datum⟩ (7.1.2 節および 6.4 節を参照) のパーサーです。与えられたテキスト入力ポート *port*からパース可能な次のオブジェクトを返し、そのオブジェクトの外部表現が終わった次の文字を指すよう *port*を更新します。

処理系はデータム表現を持たないレコード型や他の型を表現するために構文を拡張しても構いません。

オブジェクトの始まりとなる文字が見つかる前に入力がファイルの終端に達した場合は end-of-file オブジェクトが返されます。ポートは開いたまま維持され、さらに読み取りを試みると再び end-of-file オブジェクトが返されます。オブジェクトの外部表現が始まった後で、しかし外部表現が不完全なためパース不可能な状態でファイルの終端に達した場合は、read-error? を満たすエラーが通知されます。

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

テキスト入力ポート *port*から利用可能な次の文字を返し、その次の文字を指すよう *port*を更新します。それ以上文字が無い場合は end-of-file オブジェクトが返されます。

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

テキスト入力ポート *port*から利用可能な次の文字を返しますが、その次の文字を指すよう *port*を更新しません。それ以上文字が無い場合は end-of-file オブジェクトが返されます。

メモ: peek-char の呼び出しから返される値は同じ *port*に対して read-char を呼んだときに返される値と同じです。ただしその次その *port*に対して read-char または peek-char を呼んだとき先に呼んだ peek-char から返された値を返す点だけが異なります。ちなみに対話的なポートに対する read-char の呼び出しが入力待ちのために停止するような状況では peek-char の呼び出しも同様に停止します。

(read-line) 手続き
(read-line *port*) 手続き

テキスト入力ポート *port*から利用可能な次の行のテキストを返し、その後の文字を指すよう *port*を更新します。行末を読み取った場合は、その行末までのテキストをすべて含む (ただし行末自体は含まない) 文字列が返され、その行末の直後を指すようポートを更新します。行末を読み取る前にファイルの終端に達したが、文字をいくつか読み取った場合は、その文字を含む文字列が返されます。いかなる文字も読み取ることなくファイルの終端に達した場合は、end-of-file オブジェクトが返されます。この手続きにおいて行末とは改行文字、復帰文字、または復帰文字に続く改行文字の並びのいずれかです。処理系は他の行末文字や行末文字の並びを認識しても構いません。

(eof-object? *obj*) 手続き

*obj*が end-of-file オブジェクトであれば #t を返し、そうでなければ #f を返します。end-of-file オブジェクトの正確な集合は処理系によって異なりますが、いかなる場合でも read によって読み取られる可能性のあるオブジェクトが end-of-file オブジェクトとなることはありません。

(eof-object) 手続き

end-of-file オブジェクトを返します。end-of-file オブジェクトは唯一であるとは限りません。

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

テキスト入力ポート *port*において文字の準備ができた状態であれば #t を返し、そうでなければ #f を返します。char-ready が #t を返した場合はその *port*に対する次の read-char が停止しないことが保証されています。*port*がファイルの終端に達している場合は #t を返します。

論拠: char-ready? 手続きは入力待ちによって停止してしまうことなく対話的なポートから文字を受け取れるようにするために存在しています。そういったポートに IME が紐付いている場合は char-ready? によって存在が明らかになった文字が入力から削除されることがないよう保証しなければなりません。仮に char-ready? がファイルの終端で #f を返した場合、ファイルの終端に達したポートとまだ文字が準備できていない対話的なポートを区別することはできないでしょう。

(read-string *k*) 手続き
(read-string *k port*) 手続き

テキスト入力ポート *port*から次の *k*個の文字またはファイルの終端までに有る限りの文字を読み取り、新しく割り当てられた文字列に左から右の順で格納し、その文字列を返します。ファイルの終端までに利用可能な文字が無い場合は end-of-file オブジェクトが返されます。

(read-u8) 手続き
(read-u8 *port*) 手続き

バイナリ入力ポート *port*から利用可能な次のバイトを返し、その次のバイトを指すよう *port*を更新します。それ以上バイトが無い場合は end-of-file オブジェクトが返されます。

(peek-u8) 手続き
(peek-u8 *port*) 手続き

バイナリ入力ポート *port*から利用可能な次のバイトを返しますが、その次のバイトを指すよう *port*を更新**しません**。それ以上バイトが無い場合は end-of-file オブジェクトが返されます。

(u8-ready?) 手続き
(u8-ready? *port*) 手続き

バイナリ入力ポート *port*においてバイトの準備ができた状態であれば #t を返し、そうでなければ #f を返します。u8-ready が #t を返した場合はその *port*に対する次の read-u8 が停止しないことが保証されています。*port*がファイルの終端に達している場合は #t を返します。

(read-bytevector *k*) 手続き
(read-bytevector *k port*) 手続き

バイナリ入力ポート *port*から次の *k*個のバイトまたはファイルの終端までに有る限りのバイトを読み取り、新しく割り当てられたバイトベクタに左から右の順で格納し、そのバイトベクタを返します。ファイルの終端までに利用可能なバイトが無い場合は end-of-file オブジェクトが返されます。

(read-bytevector! *bytevector*) 手続き
(read-bytevector! *bytevector port*) 手続き
(read-bytevector! *bytevector port start*) 手続き
(read-bytevector! *bytevector port start end*) 手続き

バイナリ入力ポート *port*から次の $end - start$ 個のバイト、またはファイルの終端までに有る限りのバイトを読み取り、*bytevector*の *start*から始まる位置に左から右の順で格納します。*end*が指定されなかった場合は *bytevector*の終わりに達するまで読み取られます。*start*が指定されなかった場合は 0 から始まる位置に読み取られます。読み取ったバイト数を返します。バイトが無い場合は end-of-file オブジェクトが返されます。

6.13.3. 出力

出力手続きで *port*が省略された場合は (current-output-port) の戻り値がデフォルト値となります。閉じられたポートに対して出力操作を試みることはエラーです。

(write *obj*) write ライブラリの手続き
(write *obj port*) write ライブラリの手続き

与えられたテキスト出力ポート *port*に *obj*の表現を書き出します。書き出される表現内では文字列は引用符で囲まれ、文字列中のバックスラッシュおよび引用符はバックスラッシュでエスケープされます。非 ASCII 文字を含むシンボルは垂直線でエスケープされます。文字オブジェクトは #\ 記法で書き出されます。

*obj*が循環構造を持ち、通常の書き出される表現では無限ループが発生する場合は、少なくともその循環部分を形成するオブジェクトは 2.4 節で説明されているデータムラベルを用いて表現されなければなりません。循環構造が無い場合はデータムラベルを用いてはなりません。

処理系はデータム表現を持たないレコード型や他の型を表現するために構文を拡張しても構いません。

write 手続きの戻り値は規定されていません。

(write-shared *obj*) write ライブラリの手続き
(write-shared *obj port*) write ライブラリの手続き

write-shared 手続きは write と同じですが、出力中に 2 回以上現れるすべてのペアおよびベクタに対してその共有構造をデータムラベルで表現しなければなりません。

(write-simple *obj*) write ライブラリの手続き
(write-simple *obj port*) write ライブラリの手続き

write-simple 手続きは write と同じですが、データムラベルで共有構造を表現することはありません。このため、*obj*が循環構造を持っていると write-simple は終了しません。

(display *obj*) write ライブラリの手続き
(display *obj port*) write ライブラリの手続き

与えられたテキスト出力ポート *port*に *obj*の表現を書き出します。書き出される表現内に現れる文字列は write の代わりに write-string を用いたかのように出力されます。シンボルはエスケープされません。表現内に現れる文字は write の代わりに write-char を用いたかのように出力されます。

それ以外のオブジェクトに対する display の表現は規定されていません。しかし自己参照ペア、ベクタ、レコードに対して display が無限ループしてはなりません。そのため、通常の write の表現が使われる場合、write 同様に循環を表現するためデータムラベルが必要です。

処理系はデータム表現を持たないレコード型や他の型を表現するために構文を拡張しても構いません。

display 手続きの戻り値は規定されていません。

論拠: write は機械処理用の出力を生成し、display は人間が読める出力を生成する意図があります。

(newline) 手続き
(newline *port*) 手続き

テキスト出力ポート *port*に行末を書き出します。これがどのように為されるかはオペレーティングシステムによって異なります。戻り値は規定されていません。

(write-char *char*) 手続き
(write-char *char port*) 手続き

与えられたテキスト出力ポート *port*に文字 *char*を書き出します (その文字の外部表現ではありません)。戻り値は規定されていません。

(write-string *string*) 手続き
(write-string *string port*) 手続き
(write-string *string port start*) 手続き
(write-string *string port start end*) 手続き

テキスト出力ポート *port*に *string*の *start*〜*end*の文字を左から右の順で書き出します。

(write-u8 *byte*) 手続き
(write-u8 *byte port*) 手続き

与えられたバイナリ出力ポート *port*に *byte*を書き出します。戻り値は規定されていません。

(write-bytevector *bytevector*) 手続き
(write-bytevector *bytevector port*) 手続き
(write-bytevector *bytevector port start*) 手続き
(write-bytevector *bytevector port start end*) 手続き

バイナリ出力ポート *port*に *bytevector*の *start*〜*end*のバイトを左から右の順で書き出します。

(flush-output-port) 手続き
(flush-output-port *port*) 手続き

バッファリングされているすべての出力を出力ポートのバッファから基礎となるファイルまたはデバイスにフラッシュします。戻り値は規定されていません。

6.14. システムインタフェース

一般的に言って、システムインタフェースの問題はこの報告書の対称範囲外です。しかし以下の操作は重要であり、ここで述べるだけの価値があります。

(load *filename*) load ライブラリの手続き
(load *filename environment-specifier*) load ライブラリの手続き

*filename*が文字列でなければエラーです。

*filename*は処理系依存の方法により Scheme のソースコードを持つ既存のファイルの名前に変換されます。load 手続きはそのファイルから式と定義を読み取り、*environment-specifier*で指定された環境でそれらを逐次的に評価します。*environment-specifier*が省略された場合は (interaction-environment) が想定されます。

式の結果がプリントされるか否かは規定されていません。load 手続きは current-input-port や current-output-port の戻り値に影響を与えません。load 手続きの戻り値は規定されていません。

論拠: 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りません。他の種類のファイルに対する動作は処理系によって様々です。

(file-exists? *filename*) file ライブラリの手続き

*filename*が文字列でなければエラーです。

file-exists? 手続きはその名前のファイルが手続きが呼ばれた時点で存在していれば #t を返し、そうでなければ #f を返します。

(delete-file *filename*) file ライブラリの手続き

*filename*が文字列でなければエラーです。

delete-file 手続きは、その名前のファイルが存在していて削除可能であれば、それを削除します。戻り値は規定されていません。そのファイルが存在しないか、削除可能でなければ、file-error? を満たすエラーが通知されます。

(command-line) process-context ライブラリの手続き

プロセスに渡されたコマンドラインを文字列のリストとして返します。最初の文字列はコマンド名に対応していますが、具体的な内容は処理系依存です。これらの文字列のいずれかを変更することはエラーです。

`(exit)` process-context ライブラリの手続き
`(exit` *obj*`)` process-context ライブラリの手続き

保留中の dynamic-wind の *after*手続きをすべて実行し、実行中のプログラムを終了し、オペレーテイングシステムに終了値を伝えます。引数が指定されていない場合、または *obj* が `#t` の場合はプログラムが正常終了した旨がオペレーティングシステムに伝えられます。*obj*が `#f` の場合はプログラムが異常終了した旨がオペレーティングシステムに伝えられます。それ以外の場合は *obj*がそのオペレーティングシステム用の適切な終了値に変換されます (可能であれば)。

`exit` 手続きは例外を通知したりその継続に戻ってはなりません。

メモ: ハンドラを実行するという要求のため、この手続きは単なるオペレーティングシステムの終了手続きではありません。

`(emergency-exit)` process-context ライブラリの手続き
`(emergency-exit` *obj*`)` process-context ライブラリの手続き

保留中の dynamic-wind の *after*手続きを実行せずにプログラムを終了し、`exit` と同様の方法でオペレーティングシステムに終了値を伝えます。

メモ: `emergency-exit` 手続きは Windows や Posix における `_exit` 手続きに対応しています。

`(get-environment-variable` *name*`)` process-context ライブラリの手続き

多くのオペレーティングシステムでは実行中の各プロセスに**環境変数**から成る**環境**があります。(この環境を `eval` に渡せる Scheme の環境と混同しないようにしてください。6.12 節を参照。) 環境変数の名前と値は両方とも文字列です。手続き `get-environment-variable` は環境変数 *name* の値を返します。その名前の環境変数が無ければ `#f` を返します。環境変数の名前をエンコードしたり値をデコードするためにロケール情報が用いられる場合があります。`get-environment-variable` が値をデコードできない場合はエラーです。結果の文字列を変更することもエラーです。

```
(get-environment-variable "PATH")
          ⟹ "/usr/local/bin:/usr/bin:/bin"
```

`(get-environment-variables)` process-context ライブラリの手続き

すべての環境変数の名前と値を連想リストとして返します。各エントリの car が名前、cdr が値で、両方とも文字列です。リストの順番は規定されていません。これらの文字列や連想リスト自体を変更することはエラーです。

```
(get-environment-variables)
          ⟹ (("USER" . "root") ("HOME" . "/"))
```

`(current-second)` time ライブラリの手続き

国際原子時 (TAI) を基準とした現在の時刻を表す不正確な数値を返します。値 0.0 が TAI における 1970 年 1 月 1 日の真夜中 (世界時の真夜中の 10 秒前と同等) を表し、値 1.0 がその 1 TAI 秒後を表します。正確さや精度の高さは要求されません。協定世界時に適切な定数を加えて返す程度しかできなくても構いません。

`(current-jiffy)` time ライブラリの手続き

処理系定義の適切な基点から経過した *jiffy* の数を正確な整数として返します。jiffy は 1 秒を処理系定義の数で割った時間の単位で、`jiffies-per-second` 手続きの戻り値によって定義されます。始まりの基点はプログラムを実行している間は一定であることが保証されていますが、実行のたびに変わっても構いません。

論拠: `current-jiffy` を最小のオーバーヘッドで実行できるようにするため、jiffy は処理系依存であることが認められています。コンパクトな表現の整数で十分に戻り値を表せるようなものであるべきです。固定の jiffy ではどのように選んでも処理系によっては不適切なサイズとなるでしょう。非常に速いマシンにおいてはマイクロ秒は長すぎますし、一方で非常に小さな単位を用いると処理系によってはほとんどの場合に記憶領域の割り当てが必要となり、精密な時間計測の手段としては有用性が低下してしまいます。

`(jiffies-per-second)` time ライブラリの手続き

1 SI 秒あたりの jiffy の数を表す正確な整数を返します。この値は処理系固有の定数です。

```
(define (time-length)
  (let ((list (make-list 100000))
        (start (current-jiffy)))
    (length list)
    (/ (- (current-jiffy) start)
       (jiffies-per-second))))
```

`(features)` 手続き

`cond-expand` が真とみなす機能識別子のリストを返します。このリストを変更することはエラーです。`features` が返すリストの例を以下に挙げます。

```
(features)                    ⟹
  (r7rs ratios exact-complex full-unicode
   gnu-linux little-endian
   fantastic-scheme
   fantastic-scheme-1.0
   space-ship-control-system)
```

7. 形式構文と形式意味論

この章ではこの報告書のここまでの章で非形式的に述べたことについて形式的な記述を掲載します。

7.1. 形式構文

この節では拡張 BNF 記法で書かれた Scheme の形式構文を掲載します。

文法中の空白はすべて可読性のためです。⟨letter⟩、⟨character name⟩ および ⟨mnemonic escape⟩ の定義におけるものを除いて、大文字小文字は区別されません。例えば `#x1A` と `#X1a` は同等ですが、`foo` と `Foo` や `#\space` と `#\Space` は別物です。⟨empty⟩ は空文字列を表します。

記述をより簡潔化するために BNF を以下のように拡張しています。⟨thing⟩*はゼロ回以上の ⟨thing⟩ の出現を意味します。⟨thing⟩$^+$ は少なくとも 1 回以上の ⟨thing⟩ の出現を意味します。

7.1.1. 字句構造

この節では個々のトークン (識別子や数値など) が文字の並びからどのように形成されるのかを記載しています。後続の節ではトークンの並びからどのように式とプログラムが形成されるのかを記載しています。

⟨intertoken space⟩ はいかなるトークンの隣にも現れることができますが、トークンの中に現れることはできません。

垂直線で始まっていない識別子は ⟨delimiter⟩ または入力の終端で終わります。ドット、数値、文字、ブーリアンも同様です。垂直線で始まる識別子は別の垂直線で終わります。

ASCII レパートリー中の 4 つの文字 `[ ] { }` は言語の将来の拡張のために予約されています。

Scheme 処理系は下記で規定されている ASCII レパートリーの識別子文字に加え、Unicode 文字から追加のレパートリーを識別子に用いることができても構いません。ただし Unicode 一般カテゴリが Lu、 Ll、 Lt、 Lm、 Lo、 Mn、 Mc、 Me、 Nd、 Nl、 No、 Pd、 Pc、 Po、 Sc、 Sm、 Sk、 So、 Co のいずれかの文字であるか、U+200C (ゼロ幅非接合子) または U+200D (ゼロ幅接合子) である場合に限られます (これらの接合子はペルシア語、ヒンディー語およびその他の言語を正しく書くために必要なものです)。ただし最初の文字が一般カテゴリ Nd、Mc、Me のいずれかの場合はエラーです。シンボルや識別子に非 Unicode 文字を用いることもエラーです。

すべての Scheme 処理系は垂直線で囲まれた Scheme の識別子内に現れるエスケープシーケンス `\x<hexdigits>;` をサポートしなければなりません。処理系がその Unicode スカラー値を持つ文字をサポートしていれば、そのようなシーケンスを含む識別子は対応する文字を含む識別子と同等です。

⟨token⟩ ⟶ ⟨identifier⟩ | ⟨boolean⟩ | ⟨number⟩
    | ⟨character⟩ | ⟨string⟩
    | `(` | `)` | `#(` | `#u8(` | `'` | `` ` `` | `,` | `,@` | `.`
⟨delimiter⟩ ⟶ ⟨whitespace⟩ | ⟨vertical line⟩
    | `(` | `)` | `"` | `;`
⟨intraline whitespace⟩ ⟶ ⟨space or tab⟩
⟨whitespace⟩ ⟶ ⟨intraline whitespace⟩ | ⟨line ending⟩
⟨vertical line⟩ ⟶ `|`
⟨line ending⟩ ⟶ ⟨newline⟩ | ⟨return⟩ ⟨newline⟩
    | ⟨return⟩
⟨comment⟩ ⟶ `;` ⟨all subsequent characters up to a line ending⟩
    | ⟨nested comment⟩
    | `#;` ⟨intertoken space⟩ ⟨datum⟩
⟨nested comment⟩ ⟶ `#|` ⟨comment text⟩ ⟨comment cont⟩* `|#`
⟨comment text⟩ ⟶ ⟨character sequence not containing `#|` or `|#`⟩
⟨comment cont⟩ ⟶ ⟨nested comment⟩ ⟨comment text⟩
⟨directive⟩ ⟶ `#!fold-case` | `#!no-fold-case`

⟨directive⟩ に ⟨delimiter⟩ またはファイルの終端以外のものが続いた場合は文法違反であることに注意してください。

⟨atmosphere⟩ ⟶ ⟨whitespace⟩ | ⟨comment⟩ | ⟨directive⟩
⟨intertoken space⟩ ⟶ ⟨atmosphere⟩*

下記の `+i`、`-i` および ⟨infnan⟩ は ⟨peculiar identifier⟩ 規則の例外であることに注意してください。これらは識別子ではなく数値としてパースされます。

⟨identifier⟩ ⟶ ⟨initial⟩ ⟨subsequent⟩*
    | ⟨vertical line⟩ ⟨symbol element⟩* ⟨vertical line⟩
    | ⟨peculiar identifier⟩
⟨initial⟩ ⟶ ⟨letter⟩ | ⟨special initial⟩
⟨letter⟩ ⟶ `a` | `b` | `c` | `...` | `z`
    | `A` | `B` | `C` | `...` | `Z`
⟨special initial⟩ ⟶ `!` | `$` | `%` | `&` | `*` | `/` | `:` | `<` | `=`
    | `>` | `?` | `^` | `_` | `~`
⟨subsequent⟩ ⟶ ⟨initial⟩ | ⟨digit⟩
    | ⟨special subsequent⟩
⟨digit⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨hex digit⟩ ⟶ ⟨digit⟩ | `a` | `b` | `c` | `d` | `e` | `f`
⟨explicit sign⟩ ⟶ `+` | `-`
⟨special subsequent⟩ ⟶ ⟨explicit sign⟩ | `.` | `@`
⟨inline hex escape⟩ ⟶ `\x`⟨hex scalar value⟩`;`
⟨hex scalar value⟩ ⟶ ⟨hex digit⟩$^+$
⟨mnemonic escape⟩ ⟶ `\a` | `\b` | `\t` | `\n` | `\r`
⟨peculiar identifier⟩ ⟶ ⟨explicit sign⟩
    | ⟨explicit sign⟩ ⟨sign subsequent⟩ ⟨subsequent⟩*
    | ⟨explicit sign⟩ `.` ⟨dot subsequent⟩ ⟨subsequent⟩*
    | `.` ⟨dot subsequent⟩ ⟨subsequent⟩*
⟨dot subsequent⟩ ⟶ ⟨sign subsequent⟩ | `.`
⟨sign subsequent⟩ ⟶ ⟨initial⟩ | ⟨explicit sign⟩ | `@`
⟨symbol element⟩ ⟶
    ⟨any character other than ⟨vertical line⟩ or `\`⟩
    | ⟨inline hex escape⟩ | ⟨mnemonic escape⟩ | `\|`

⟨boolean⟩ ⟶ `#t` | `#f` | `#true` | `#false`

⟨character⟩ ⟶ `#\` ⟨any character⟩
| `#\` ⟨character name⟩
| `#\x`⟨hex scalar value⟩
⟨character name⟩ ⟶ `alarm` | `backspace` | `delete`
| `escape` | `newline` | `null` | `return` | `space` | `tab`

⟨string⟩ ⟶ `"` ⟨string element⟩* `"`
⟨string element⟩ ⟶ ⟨any character other than " or \⟩
| ⟨mnemonic escape⟩ | `\"` | `\\`
| `\`⟨intraline whitespace⟩*⟨line ending⟩
⟨intraline whitespace⟩*
| ⟨inline hex escape⟩
⟨bytevector⟩ ⟶ `#u8(`⟨byte⟩*`)`
⟨byte⟩ ⟶ ⟨any exact integer between 0 and 255⟩

⟨number⟩ ⟶ ⟨num 2⟩ | ⟨num 8⟩
| ⟨num 10⟩ | ⟨num 16⟩

下記の規則 ⟨num R⟩、⟨complex R⟩、⟨real R⟩、⟨ureal R⟩、⟨uinteger R⟩ および ⟨prefix R⟩ は、$R = 2, 8, 10, 16$ について暗黙に複製されます。⟨decimal 2⟩、⟨decimal 8⟩ および ⟨decimal 16⟩ の規則は存在しません。つまり小数点や指数を含む数値は必ず 10 進数だという意味です。以下には示していませんが、数値の文法で使われるアルファベットの文字はすべて、大文字でも小文字でも構いません。

⟨num R⟩ ⟶ ⟨prefix R⟩ ⟨complex R⟩
⟨complex R⟩ ⟶ ⟨real R⟩ | ⟨real R⟩ `@` ⟨real R⟩
| ⟨real R⟩ `+` ⟨ureal R⟩ `i` | ⟨real R⟩ `-` ⟨ureal R⟩ `i`
| ⟨real R⟩ `+ i` | ⟨real R⟩ `- i` | ⟨real R⟩ ⟨infnan⟩ `i`
| `+` ⟨ureal R⟩ `i` | `-` ⟨ureal R⟩ `i`
| ⟨infnan⟩ `i` | `+ i` | `- i`
⟨real R⟩ ⟶ ⟨sign⟩ ⟨ureal R⟩
| ⟨infnan⟩
⟨ureal R⟩ ⟶ ⟨uinteger R⟩
| ⟨uinteger R⟩ `/` ⟨uinteger R⟩
| ⟨decimal R⟩
⟨decimal 10⟩ ⟶ ⟨uinteger 10⟩ ⟨suffix⟩
| `.` ⟨digit 10⟩$^+$ ⟨suffix⟩
| ⟨digit 10⟩$^+$ `.` ⟨digit 10⟩* ⟨suffix⟩
⟨uinteger R⟩ ⟶ ⟨digit R⟩$^+$
⟨prefix R⟩ ⟶ ⟨radix R⟩ ⟨exactness⟩
| ⟨exactness⟩ ⟨radix R⟩
⟨infnan⟩ ⟶ `+inf.0` | `-inf.0` | `+nan.0` | `-nan.0`

⟨suffix⟩ ⟶ ⟨empty⟩
| ⟨exponent marker⟩ ⟨sign⟩ ⟨digit 10⟩$^+$
⟨exponent marker⟩ ⟶ `e`
⟨sign⟩ ⟶ ⟨empty⟩ | `+` | `-`
⟨exactness⟩ ⟶ ⟨empty⟩ | `#i` | `#e`
⟨radix 2⟩ ⟶ `#b`
⟨radix 8⟩ ⟶ `#o`
⟨radix 10⟩ ⟶ ⟨empty⟩ | `#d`
⟨radix 16⟩ ⟶ `#x`
⟨digit 2⟩ ⟶ `0` | `1`
⟨digit 8⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
⟨digit 10⟩ ⟶ ⟨digit⟩
⟨digit 16⟩ ⟶ ⟨digit 10⟩ | `a` | `b` | `c` | `d` | `e` | `f`

7.1.2. 外部表現

⟨datum⟩ は `read` 手続き (6.13.2 節を参照) が正常にパースできるものです。⟨expression⟩ としてパースできる文字列はすべて ⟨datum⟩ としてもパースできます。

⟨datum⟩ ⟶ ⟨simple datum⟩ | ⟨compound datum⟩
| ⟨label⟩ `=` ⟨datum⟩ | ⟨label⟩ `#`
⟨simple datum⟩ ⟶ ⟨boolean⟩ | ⟨number⟩
| ⟨character⟩ | ⟨string⟩ | ⟨symbol⟩ | ⟨bytevector⟩
⟨symbol⟩ ⟶ ⟨identifier⟩
⟨compound datum⟩ ⟶ ⟨list⟩ | ⟨vector⟩ | ⟨abbreviation⟩
⟨list⟩ ⟶ `(`⟨datum⟩*`)` | `(`⟨datum⟩$^+$ `.` ⟨datum⟩`)`
⟨abbreviation⟩ ⟶ ⟨abbrev prefix⟩ ⟨datum⟩
⟨abbrev prefix⟩ ⟶ `'` | `` ` `` | `,` | `,@`
⟨vector⟩ ⟶ `#(`⟨datum⟩*`)`
⟨label⟩ ⟶ `#` ⟨uinteger 10⟩

7.1.3. 式

この節および後続の節にある定義はこの報告書で定義されているすべての構文キーワードがそのライブラリから適切にインポートされており、いずれも再定義または隠蔽されていないものと仮定しています。

⟨expression⟩ ⟶ ⟨identifier⟩
| ⟨literal⟩
| ⟨procedure call⟩
| ⟨lambda expression⟩
| ⟨conditional⟩
| ⟨assignment⟩
| ⟨derived expression⟩
| ⟨macro use⟩
| ⟨macro block⟩
| ⟨includer⟩

⟨literal⟩ ⟶ ⟨quotation⟩ | ⟨self-evaluating⟩
⟨self-evaluating⟩ ⟶ ⟨boolean⟩ | ⟨number⟩ | ⟨vector⟩
| ⟨character⟩ | ⟨string⟩ | ⟨bytevector⟩
⟨quotation⟩ ⟶ `'`⟨datum⟩ | `(quote` ⟨datum⟩`)`
⟨procedure call⟩ ⟶ `(`⟨operator⟩ ⟨operand⟩*`)`
⟨operator⟩ ⟶ ⟨expression⟩
⟨operand⟩ ⟶ ⟨expression⟩

⟨lambda expression⟩ ⟶ `(lambda` ⟨formals⟩ ⟨body⟩`)`
⟨formals⟩ ⟶ `(`⟨identifier⟩*`)` | ⟨identifier⟩
| `(`⟨identifier⟩$^+$ `.` ⟨identifier⟩`)`
⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩
⟨command⟩ ⟶ ⟨expression⟩

⟨conditional⟩ ⟶ `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`
⟨test⟩ ⟶ ⟨expression⟩
⟨consequent⟩ ⟶ ⟨expression⟩
⟨alternate⟩ ⟶ ⟨expression⟩ | ⟨empty⟩

⟨assignment⟩ ⟶ `(set!` ⟨identifier⟩ ⟨expression⟩`)`

⟨derived expression⟩ ⟶
`(cond` ⟨cond clause⟩$^+$`)`
| `(cond` ⟨cond clause⟩* `(else` ⟨sequence⟩`))`
| `(case` ⟨expression⟩
⟨case clause⟩$^+$`)`
| `(case` ⟨expression⟩
⟨case clause⟩*
`(else` ⟨sequence⟩`)))`
| `(case` ⟨expression⟩
⟨case clause⟩*
`(else =>` ⟨recipient⟩`)))`
| `(and` ⟨test⟩*`)`
| `(or` ⟨test⟩*`)`
| `(when` ⟨test⟩ ⟨sequence⟩`)`
| `(unless` ⟨test⟩ ⟨sequence⟩`)`
| `(let (`⟨binding spec⟩*`)` ⟨body⟩`)`
| `(let` ⟨identifier⟩ `(`⟨binding spec⟩*`)` ⟨body⟩`)`
| `(let* (`⟨binding spec⟩*`)` ⟨body⟩`)`
| `(letrec (`⟨binding spec⟩*`)` ⟨body⟩`)`
| `(letrec* (`⟨binding spec⟩*`)` ⟨body⟩`)`
| `(let-values (`⟨mv binding spec⟩*`)` ⟨body⟩`)`
| `(let*-values (`⟨mv binding spec⟩*`)` ⟨body⟩`)`
| `(begin` ⟨sequence⟩`)`
| `(do (`⟨iteration spec⟩*`)`
`(`⟨test⟩ ⟨do result⟩`)`
⟨command⟩*`)`
| `(delay` ⟨expression⟩`)`
| `(delay-force` ⟨expression⟩`)`
| `(parameterize ((`⟨expression⟩ ⟨expression⟩`)`*`)`
⟨body⟩`)`
| `(guard (`⟨identifier⟩ ⟨cond clause⟩*`)` ⟨body⟩`)`
| ⟨quasiquotation⟩
| `(case-lambda` ⟨case-lambda clause⟩*`)`

⟨cond clause⟩ ⟶ `(`⟨test⟩ ⟨sequence⟩`)`
| `(`⟨test⟩`)`
| `(`⟨test⟩ `=>` ⟨recipient⟩`)`
⟨recipient⟩ ⟶ ⟨expression⟩
⟨case clause⟩ ⟶ `((`⟨datum⟩*`)` ⟨sequence⟩`)`
| `((`⟨datum⟩*`) =>` ⟨recipient⟩`)`
⟨binding spec⟩ ⟶ `(`⟨identifier⟩ ⟨expression⟩`)`
⟨mv binding spec⟩ ⟶ `(`⟨formals⟩ ⟨expression⟩`)`
⟨iteration spec⟩ ⟶ `(`⟨identifier⟩ ⟨init⟩ ⟨step⟩`)`
| `(`⟨identifier⟩ ⟨init⟩`)`
⟨case-lambda clause⟩ ⟶ `(`⟨formals⟩ ⟨body⟩`)`
⟨init⟩ ⟶ ⟨expression⟩
⟨step⟩ ⟶ ⟨expression⟩

⟨do result⟩ ⟶ ⟨sequence⟩ | ⟨empty⟩

⟨macro use⟩ ⟶ `(`⟨keyword⟩ ⟨datum⟩*`)`
⟨keyword⟩ ⟶ ⟨identifier⟩

⟨macro block⟩ ⟶
`(let-syntax (`⟨syntax spec⟩*`)` ⟨body⟩`)`
| `(letrec-syntax (`⟨syntax spec⟩*`)` ⟨body⟩`)`
⟨syntax spec⟩ ⟶ `(`⟨keyword⟩ ⟨transformer spec⟩`)`

⟨includer⟩ ⟶
| `(include` ⟨string⟩$^+$`)`
| `(include-ci` ⟨string⟩$^+$`)`

7.1.4. quasiquote

quasiquote 式に対する以下の文法は文脈自由文法ではありません。これは生成規則を無限に生成するレシピとして表現されています。$D = 1, 2, 3, \ldots$ に対して以下のルールが複製されると考えてください。ちなみに D は入れ子の深さです。

⟨quasiquotation⟩ ⟶ ⟨quasiquotation 1⟩
⟨qq template 0⟩ ⟶ ⟨expression⟩
⟨quasiquotation D⟩ ⟶ `` ` ``⟨qq template D⟩
| `(quasiquote` ⟨qq template D⟩`)`
⟨qq template D⟩ ⟶ ⟨simple datum⟩
| ⟨list qq template D⟩
| ⟨vector qq template D⟩
| ⟨unquotation D⟩
⟨list qq template D⟩ ⟶ `(`⟨qq template or splice D⟩*`)`
| `(`⟨qq template or splice D⟩$^+$ `.` ⟨qq template D⟩`)`
| `'`⟨qq template D⟩
| ⟨quasiquotation $D+1$⟩
⟨vector qq template D⟩ ⟶ `#(`⟨qq template or splice D⟩*`)`
⟨unquotation D⟩ ⟶ `,`⟨qq template $D-1$⟩
| `(unquote` ⟨qq template $D-1$⟩`)`
⟨qq template or splice D⟩ ⟶ ⟨qq template D⟩
| ⟨splicing unquotation D⟩
⟨splicing unquotation D⟩ ⟶ `,@`⟨qq template $D-1$⟩
| `(unquote-splicing` ⟨qq template $D-1$⟩`)`

⟨quasiquotation⟩ 内において ⟨list qq template D⟩ が ⟨unquotation D⟩ または ⟨splicing unquotation D⟩ のどちらかと曖昧になることがあります。この場合 ⟨unquotation⟩ または ⟨splicing unquotation D⟩ として解釈する方を優先します。

7.1.5. 変換子

⟨transformer spec⟩ ⟶
`(syntax-rules (`⟨identifier⟩*`)` ⟨syntax rule⟩*`)`
| `(syntax-rules` ⟨identifier⟩ `(`⟨identifier⟩*`)`
⟨syntax rule⟩*`)`
⟨syntax rule⟩ ⟶ `(`⟨pattern⟩ ⟨template⟩`)`

⟨pattern⟩ ⟶ ⟨pattern identifier⟩
| ⟨underscore⟩
| (⟨pattern⟩*)
| (⟨pattern⟩+ . ⟨pattern⟩)
| (⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩*)
| (⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩* . ⟨pattern⟩)
| #(⟨pattern⟩*)
| #(⟨pattern⟩* ⟨pattern⟩ ⟨ellipsis⟩ ⟨pattern⟩*)
| ⟨pattern datum⟩
⟨pattern datum⟩ ⟶ ⟨string⟩
| ⟨character⟩
| ⟨boolean⟩
| ⟨number⟩
⟨template⟩ ⟶ ⟨pattern identifier⟩
| (⟨template element⟩*)
| (⟨template element⟩+ . ⟨template⟩)
| #(⟨template element⟩*)
| ⟨template datum⟩
⟨template element⟩ ⟶ ⟨template⟩
| ⟨template⟩ ⟨ellipsis⟩
⟨template datum⟩ ⟶ ⟨pattern datum⟩
⟨pattern identifier⟩ ⟶ ⟨any identifier except ...⟩
⟨ellipsis⟩ ⟶ ⟨an identifier defaulting to ...⟩
⟨underscore⟩ ⟶ ⟨the identifier _⟩

7.1.6. プログラムおよび定義

⟨program⟩ ⟶
⟨import declaration⟩+
⟨command or definition⟩+
⟨command or definition⟩ ⟶ ⟨command⟩
| ⟨definition⟩
| (begin ⟨command or definition⟩+)
⟨definition⟩ ⟶ (define ⟨identifier⟩ ⟨expression⟩)
| (define (⟨identifier⟩ ⟨def formals⟩) ⟨body⟩)
| ⟨syntax definition⟩
| (define-values ⟨formals⟩ ⟨body⟩)
| (define-record-type ⟨identifier⟩ ⟨constructor⟩ ⟨identifier⟩ ⟨field spec⟩*)
| (begin ⟨definition⟩*)
⟨def formals⟩ ⟶ ⟨identifier⟩*
| ⟨identifier⟩* . ⟨identifier⟩
⟨constructor⟩ ⟶ (⟨identifier⟩ ⟨field name⟩*)
⟨field spec⟩ ⟶ (⟨field name⟩ ⟨accessor⟩)
| (⟨field name⟩ ⟨accessor⟩ ⟨mutator⟩)
⟨field name⟩ ⟶ ⟨identifier⟩
⟨accessor⟩ ⟶ ⟨identifier⟩
⟨mutator⟩ ⟶ ⟨identifier⟩
⟨syntax definition⟩ ⟶
(define-syntax ⟨keyword⟩ ⟨transformer spec⟩)

7.1.7. ライブラリ

⟨library⟩ ⟶
(define-library ⟨library name⟩
⟨library declaration⟩*)
⟨library name⟩ ⟶ (⟨library name part⟩+)
⟨library name part⟩ ⟶ ⟨identifier⟩ | ⟨uinteger 10⟩
⟨library declaration⟩ ⟶ (export ⟨export spec⟩*)
| ⟨import declaration⟩
| (begin ⟨command or definition⟩*)
| ⟨includer⟩
| (include-library-declarations ⟨string⟩+)
| (cond-expand ⟨cond-expand clause⟩+)
| (cond-expand ⟨cond-expand clause⟩+ (else ⟨library declaration⟩*))
⟨import declaration⟩ ⟶ (import ⟨import set⟩+)
⟨export spec⟩ ⟶ ⟨identifier⟩
| (rename ⟨identifier⟩ ⟨identifier⟩)
⟨import set⟩ ⟶ ⟨library name⟩
| (only ⟨import set⟩ ⟨identifier⟩+)
| (except ⟨import set⟩ ⟨identifier⟩+)
| (prefix ⟨import set⟩ ⟨identifier⟩)
| (rename ⟨import set⟩ (⟨identifier⟩ ⟨identifier⟩)+)
⟨cond-expand clause⟩ ⟶
(⟨feature requirement⟩ ⟨library declaration⟩*)
⟨feature requirement⟩ ⟶ ⟨identifier⟩
| ⟨library name⟩
| (and ⟨feature requirement⟩*)
| (or ⟨feature requirement⟩*)
| (not ⟨feature requirement⟩)

7.2. 形式意味論

この節では Scheme のプリミティブ式およびいくつかの組み込み手続きに対する形式的な表示的意味論を掲載します。ここで使用する概念および記法は [36] で説明されています。`dynamic-wind` の定義は [39] から取っています。以下に記法を要約します。

$\langle \ldots \rangle$	並びの形成
$s \downarrow k$	並び s の k 番目の要素 (1 ベース)
$\#s$	並び s の長さ
$s \S t$	並び s および t の連結
$s \dagger k$	並び s の最初の k 個の要素を捨てる
$t \to a, b$	マッカーシーの条件式 「もし t ならば a、さもなくば b」
$\rho[x/i]$	ρ の i に対応する値を x に置換
x in D	x の領域 D への注入
$x \,\vert\, \mathtt{D}$	x の領域 D への写像

式の継続が単一の値ではなく値の並びを取る理由は、手続き呼び出しと複数の戻り値の形式的な扱いをシンプルにするためです。

ペア、ベクタおよび文字列に紐づけられたブーリアンのフラ

グは、可変オブジェクトの場合は真、不変オブジェクトの場合は偽になります。

手続き呼び出し式内の評価順序は規定されていません。ここでは、適当な置換関数 *permute* および *unpermute* を評価の前後で呼び出しの引数に適用することで、それを表しています。これらは逆関数でなければなりません。これは、プログラム全体で (同じ個数の引数に対しては) 評価順序が固定されることを暗に意味するわけで、あまり正しくはありません。しかし左から右へ評価するよりは、意図している意味論に近い近似です。

記憶領域の割り当て関数 *new* は実装依存ですが、$new\,\sigma \in \mathtt{L}$ ならば $\sigma\,(new\,\sigma\,|\,\mathtt{L}) \downarrow 2 = false$ でなければなりません。

$\mathcal{K}$ の定義は、それほど興味深くないにもかかわらず、正確に定義すると意味論が複雑化するため、省略しています。

あらゆる変数が参照または代入される前に定義済みであるとする場合、プログラム P の意味は以下のようになります。

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

ただし、I*は P で定義される変数の並び、P′ は P の全ての定義を代入で置換して得られる式の並び、⟨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す。

7.2.1. 抽象構文

K ∈ Con	定数 (quote を含む)
I ∈ Ide	識別子 (変数)
E ∈ Exp	式
Γ ∈ Com = Exp	命令

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. 領域方程式

$\alpha \in \mathtt{L}$		位置
$\nu \in \mathtt{N}$		自然数
$\mathtt{T}$	$= \{false, true\}$	ブーリアン
$\mathtt{Q}$		シンボル
$\mathtt{H}$		文字
$\mathtt{R}$		数値
$\mathtt{E_p}$	$= \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	ペア
$\mathtt{E_v}$	$= \mathtt{L}^* \times \mathtt{T}$	ベクタ
$\mathtt{E_s}$	$= \mathtt{L}^* \times \mathtt{T}$	文字列
$\mathtt{M}$	$= \{false, true, null, undefined, unspecified\}$	その他いろいろ
$\phi \in \mathtt{F}$	$= \mathtt{L} \times (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	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
$\sigma \in \mathtt{S}$	$= \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	記憶装置
$\rho \in \mathtt{U}$	$= \mathrm{Ide} \to \mathtt{L}$	環境
$\theta \in \mathtt{C}$	$= \mathtt{S} \to \mathtt{A}$	命令の継続
$\kappa \in \mathtt{K}$	$= \mathtt{E}^* \to \mathtt{C}$	式の継続
$\mathtt{A}$		答え
$\mathtt{X}$		エラー
$\omega \in \mathtt{P}$	$= (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{root\}$	動的点

7.2.3. 意味関数

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

$\mathcal{K}$ の定義は意図的に省略しています。

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

$$\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\omega\kappa\,.\,hold\,(lookup\,\rho\,\mathrm{I})\,(single(\lambda\epsilon\,.\,\epsilon = undefined \to wrong\ \text{“undefined variable”},\ send\,\epsilon\,\kappa))$$

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

$$\mathcal{E}[\![\texttt{(lambda (I*) }\Gamma^*\ \mathrm{E_0)}]\!] = \lambda\rho\omega\kappa\,.\,\lambda\sigma\,.\,new\,\sigma \in \mathtt{L} \to send\,(\langle new\,\sigma\,|\,\mathtt{L},\ \lambda\epsilon^*\omega'\kappa'\,.\,\#\epsilon^* = \#\mathrm{I^*} \to tievals(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathrm{E_0}]\!]\rho'\omega'\kappa'))\ (extends\,\rho\,\mathrm{I^*}\,\alpha^*))\ \epsilon^*,\ wrong\ \text{“wrong number of arguments”}\rangle\ \text{in E})\ \kappa\ (update\,(new\,\sigma\,|\,\mathtt{L})\,unspecified\,\sigma),\ wrong\ \text{“out of memory”}\ \sigma$$

$$\mathcal{E}[\![\texttt{(lambda (I* . I) }\Gamma^*\texttt{ E}_0\texttt{)}]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\lambda\sigma\,.$$
$$new\,\sigma \in \mathtt{L} \to$$
$$send\,(\langle new\,\sigma \mid \mathtt{L},$$
$$\lambda\epsilon^*\omega'\kappa'\,.\,\#\epsilon^* \geq \#\mathrm{I}^* \to$$
$$tievalsrest$$
$$(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![\mathrm{E}_0]\!]\rho'\omega'\kappa'))$$
$$(extends\,\rho\,(\mathrm{I}^* \S \langle \mathrm{I} \rangle)\,\alpha^*))$$
$$\epsilon^*$$
$$(\#\mathrm{I}^*),$$
$$wrong\ \text{“too few arguments”}\rangle\ \text{in}\ \mathtt{E})$$
$$\kappa$$
$$(update\,(new\,\sigma \mid \mathtt{L})\,unspecified\,\sigma),$$
$$wrong\ \text{“out of memory”}\ \sigma$$

$$\mathcal{E}[\![\texttt{(lambda I }\Gamma^*\texttt{ E}_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (. I) }\Gamma^*\texttt{ E}_0\texttt{)}]\!]$$

$$\mathcal{E}[\![\texttt{(if E}_0\texttt{ E}_1\texttt{ E}_2\texttt{)}]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,$$
$$\mathcal{E}[\![\mathrm{E}_2]\!]\rho\omega\kappa))$$

$$\mathcal{E}[\![\texttt{(if E}_0\texttt{ E}_1\texttt{)}]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single\,(\lambda\epsilon\,.\,truish\,\epsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,$$
$$send\,unspecified\,\kappa))$$

ここでは (他の場所でも)、*unspecified* の代わりに任意の式の値 (*undefined* を除く) を使用しても構いません。

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

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

$$\mathcal{E}^*[\![\mathrm{E}_0\ \mathrm{E}^*]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(single(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\omega\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle \S \epsilon^*))))$$

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

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

7.2.4. 補助関数

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

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

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

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

$$single : (\mathtt{E} \to \mathtt{C}) \to \mathtt{K}$$
$$single =$$
$$\lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \to \psi(\epsilon^* \downarrow 1),$$
$$wrong\ \text{“wrong number of return values”}$$

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

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

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

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

$$tievals : (\mathtt{L}^* \to \mathtt{C}) \to \mathtt{E}^* \to \mathtt{C}$$
$$tievals =$$
$$\lambda\psi\epsilon^*\sigma\,.\,\#\epsilon^* = 0 \to \psi\langle\,\rangle\sigma,$$
$$new\,\sigma \in \mathtt{L} \to tievals\,(\lambda\alpha^*\,.\,\psi(\langle new\,\sigma \mid \mathtt{L}\rangle \S \alpha^*))$$
$$(\epsilon^* \dagger 1)$$
$$(update(new\,\sigma \mid \mathtt{L})(\epsilon^* \downarrow 1)\sigma),$$
$$wrong\ \text{“out of memory”}\sigma$$

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

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

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

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

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

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

$$applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$applicate =$$
$$\lambda\epsilon\epsilon^*\omega\kappa\,.\,\epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F} \downarrow 2)\epsilon^*\omega\kappa, wrong\ \text{“bad procedure”}$$

$$onearg : (\mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$$
$$onearg =$$
$$\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 1 \to \zeta(\epsilon^* \downarrow 1)\omega\kappa,$$
$$wrong\ \text{“wrong number of arguments”}$$

$$twoarg : (\mathtt{E} \to \mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$$
$$twoarg =$$
$$\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 2 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\omega\kappa,$$
$$wrong\ \text{“wrong number of arguments”}$$

$$threearg : (\mathtt{E} \to \mathtt{E} \to \mathtt{E} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}) \to (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$$
$$threearg =$$
$$\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 3 \to \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)(\epsilon^* \downarrow 3)\omega\kappa,$$
$$wrong\ \text{“wrong number of arguments”}$$

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

$$cons : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$cons =$$
$$twoarg\,(\lambda\epsilon_1\epsilon_2\kappa\omega\sigma\,.\,new\,\sigma \in \mathtt{L} \to$$
$$(\lambda\sigma'\,.\,new\,\sigma' \in \mathtt{L} \to$$
$$send\,(\langle new\,\sigma \mid \mathtt{L}, new\,\sigma' \mid \mathtt{L}, true\rangle$$
$$\text{in}\ \mathtt{E})$$
$$\kappa$$
$$(update(new\,\sigma' \mid \mathtt{L})\epsilon_2\sigma'),$$
$$wrong\ \text{“out of memory”}\sigma')$$
$$(update(new\,\sigma \mid \mathtt{L})\epsilon_1\sigma),$$
$$wrong\ \text{“out of memory”}\sigma)$$

```
less : E* → P → K → C
less =
  twoarg (λε1ε2ωκ . (ε1 ∈ R ∧ ε2 ∈ R) →
                     send (ε1 | R < ε2 | R → true, false)κ,
                     wrong "non-numeric argument to <")

add : E* → P → K → C
add =
  twoarg (λε1ε2ωκ . (ε1 ∈ R ∧ ε2 ∈ R) →
                     send ((ε1 | R + ε2 | R) in E)κ,
                     wrong "non-numeric argument to +")

car : E* → P → K → C
car =
  onearg (λεωκ . ε ∈ Ep → car-internal εκ,
                 wrong "non-pair argument to car")

car-internal : E → K → C
car-internal =  λεωκ . hold (ε | Ep ↓ 1)κ

cdr : E* → P → K → C      [car と同様]

cdr-internal : E → K → C      [car-internal と同様]

setcar : E* → P → K → C
setcar =
  twoarg (λε1ε2ωκ . ε1 ∈ Ep →
                   (ε1 | Ep ↓ 3) → assign (ε1 | Ep ↓ 1)
                                          ε2
                                          (send unspecified κ),
                   wrong "immutable argument to set-car!",
                   wrong "non-pair argument to set-car!")

eqv : E* → P → K → C
eqv =
  twoarg (λε1ε2ωκ . (ε1 ∈ M ∧ ε2 ∈ M) →
                     send (ε1 | M = ε2 | M → true, false)κ,
                   (ε1 ∈ Q ∧ ε2 ∈ Q) →
                     send (ε1 | Q = ε2 | Q → true, false)κ,
                   (ε1 ∈ H ∧ ε2 ∈ H) →
                     send (ε1 | H = ε2 | H → true, false)κ,
                   (ε1 ∈ R ∧ ε2 ∈ R) →
                     send (ε1 | R = ε2 | R → true, false)κ,
                   (ε1 ∈ Ep ∧ ε2 ∈ Ep) →
                     send ((λp1p2 . ((p1 ↓ 1) = (p2 ↓ 1)∧
                                     (p1 ↓ 2) = (p2 ↓ 2)) → true,
                                       false)
                          (ε1 | Ep)
                          (ε2 | Ep))
                         κ,
                   (ε1 ∈ Ev ∧ ε2 ∈ Ev) → ...,
                   (ε1 ∈ Es ∧ ε2 ∈ Es) → ...,
                   (ε1 ∈ F ∧ ε2 ∈ F) →
                     send ((ε1 | F ↓ 1) = (ε2 | F ↓ 1) → true, false)
                          κ,
                     send false κ)

apply : E* → P → K → C
apply =
  twoarg (λε1ε2ωκ . ε1 ∈ F → valueslist ε2(λε* . applicate ε1ε*ωκ),
                     wrong "bad procedure argument to apply")

valueslist : E → K → C
valueslist =
  λεκ . ε ∈ Ep →
          cdr-internal ε
                       (λε* . valueslist
                              ε*
                              (λε* . car-internal
                                     ε
                                     (single(λε . κ(⟨ε⟩ § ε*))))),
        ε = null → κ⟨ ⟩,
          wrong "non-list argument to values-list"

cwcc: E* → P → K → C
          [call-with-current-continuation]
cwcc =
  onearg (λεωκ . ε ∈ F →
                 (λσ . new σ ∈ L →
                       applicate ε
                                 ⟨⟨new σ | L,
                                   λε*ω′κ′ . travel ω′ω(κε*)⟩
                                  in E⟩
                                 ω
                                 κ
                                 (update (new σ | L)
                                         unspecified
                                         σ),
                       wrong "out of memory" σ),
                 wrong "bad procedure argument")

travel : P → P → C → C
travel =
  λω1ω2 . travelpath ((pathup ω1(commonancest ω1ω2)) §
                      (pathdown (commonancest ω1ω2)ω2))

pointdepth : P → N
pointdepth =
  λω . ω = root → 0, 1 + (pointdepth (ω | (F × F × P) ↓ 3))

ancestors : P → PP
ancestors =
  λω . ω = root → {ω}, {ω} ∪ (ancestors (ω | (F × F × P) ↓ 3))

commonancest : P → P → P
commonancest =
  λω1ω2 . the only element of
          {ω′ | ω′ ∈ (ancestors ω1) ∩ (ancestors ω2),
                pointdepth ω′ ≥ pointdepth ω″
                ∀ω″ ∈ (ancestors ω1) ∩ (ancestors ω2)}

pathup : P → P → (P × F)*
pathup =
  λω1ω2 . ω1 = ω2 → ⟨⟩,
          ⟨(ω1, ω1 | (F × F × P) ↓ 2)⟩ §
          (pathup (ω1 | (F × F × P) ↓ 3)ω2)

pathdown : P → P → (P × F)*
pathdown =
  λω1ω2 . ω1 = ω2 → ⟨⟩,
          (pathdown ω1(ω2 | (F × F × P) ↓ 3)) §
          ⟨(ω2, ω2 | (F × F × P) ↓ 1)⟩

travelpath : (P × F)* → C → C
travelpath =
```

$\lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta,$
$((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1)$
$(\lambda\epsilon^*\,.\,travelpath\,(\pi^* \dagger 1)\theta)$

$dynamicwind : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$dynamicwind =$
$threearg\,(\lambda\epsilon_1\epsilon_2\epsilon_3\omega\kappa\,.\,(\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F} \wedge \epsilon_3 \in \mathtt{F}) \to$
$applicate\,\epsilon_1\langle\rangle\omega(\lambda\zeta^*\,.$
$applicate\,\epsilon_2\langle\rangle((\epsilon_1 \mid \mathtt{F}, \epsilon_3 \mid \mathtt{F}, \omega)\ \mathrm{in}\ \mathtt{P})$
$(\lambda\epsilon^*\,.\,applicate\,\epsilon_3\langle\rangle\omega(\lambda\zeta^*\,.\,\kappa\epsilon^*))),$
$wrong$ “bad procedure argument”)

$values : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$
$values = \lambda\epsilon^*\omega\kappa\,.\,\kappa\epsilon^*$

$cwv : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$ [call-with-values]
$cwv =$
$twoarg\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,applicate\,\epsilon_1\langle\,\rangle\omega(\lambda\epsilon^*\,.\,applicate\,\epsilon_2\,\epsilon^*\omega))$

7.3. 派生式型

この節ではプリミティブ式型 (リテラル、変数、呼び出し、lambda、if および set!) を用いた quasiquote 以外の派生式型の構文定義を掲載しています。

条件分岐の派生式型:

```
(define-syntax cond
  (syntax-rules (else =>)
    ((cond (else result1 result2 ...))
     (begin result1 result2 ...))
    ((cond (test => result))
     (let ((temp test))
       (if temp (result temp))))
    ((cond (test => result) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (cond clause1 clause2 ...))))
    ((cond (test)) test)
    ((cond (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (cond clause1 clause2 ...))))
    ((cond (test result1 result2 ...))
     (if test (begin result1 result2 ...)))
    ((cond (test result1 result2 ...)
           clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (cond clause1 clause2 ...)))))


(define-syntax case
  (syntax-rules (else =>)
    ((case (key ...)
       clauses ...)
     (let ((atom-key (key ...)))
       (case atom-key clauses ...)))
    ((case key
       (else => result))
     (result key))
    ((case key
       (else result1 result2 ...))
     (begin result1 result2 ...))
    ((case key
       ((atoms ...) result1 result2 ...))
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)))
    ((case key
       ((atoms ...) => result))
     (if (memv key '(atoms ...))
         (result key)))
    ((case key
       ((atoms ...) => result)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (result key)
         (case key clause clauses ...)))
    ((case key
       ((atoms ...) result1 result2 ...)
       clause clauses ...)
     (if (memv key '(atoms ...))
         (begin result1 result2 ...)
         (case key clause clauses ...)))))


(define-syntax and
  (syntax-rules ()
    ((and) #t)
    ((and test) test)
    ((and test1 test2 ...)
     (if test1 (and test2 ...) #f))))


(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or test) test)
    ((or test1 test2 ...)
     (let ((x test1))
       (if x x (or test2 ...))))))


(define-syntax when
  (syntax-rules ()
    ((when test result1 result2 ...)
     (if test
         (begin result1 result2 ...)))))


(define-syntax unless
  (syntax-rules ()
    ((unless test result1 result2 ...)
     (if (not test)
         (begin result1 result2 ...)))))
```

束縛構文:

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

以下の letrec マクロはシンボル <undefined> を使っています。これは格納した場所から値を取り出そうとするとエラーになる何らかのものを返す式です。(Scheme ではそのような式は定義されていません。) 値を評価する順序を規定することを避けるために必要な一時的な名前を生成するためにトリックを用いています。これは補助マクロを使うことによっても成し遂げられます。

```
(define-syntax letrec
  (syntax-rules ()
    ((letrec ((var1 init1) ...) body ...)
     (letrec "generate_temp_names"
       (var1 ...)
       ()
       ((var1 init1) ...)
       body ...))
    ((letrec "generate_temp_names"
       ()
       (temp1 ...)
       ((var1 init1) ...)
       body ...)
     (let ((var1 <undefined>) ...)
       (let ((temp1 init1) ...)
         (set! var1 temp1)
         ...
         body ...)))
    ((letrec "generate_temp_names"
       (x y ...)
       (temp ...)
       ((var1 init1) ...)
       body ...)
     (letrec "generate_temp_names"
       (y ...)
       (newtemp temp ...)
       ((var1 init1) ...)
       body ...))))


(define-syntax letrec*
  (syntax-rules ()
    ((letrec* ((var1 init1) ...) body1 body2 ...)
     (let ((var1 <undefined>) ...)
       (set! var1 init1)
       ...
       (let () body1 body2 ...))))))

(define-syntax let-values
  (syntax-rules ()
    ((let-values (binding ...) body0 body1 ...)
     (let-values "bind"
         (binding ...) () (begin body0 body1 ...)))

    ((let-values "bind" () tmps body)
     (let tmps body))

    ((let-values "bind" ((b0 e0)
         binding ...) tmps body)
     (let-values "mktmp" b0 e0 ()
         (binding ...) tmps body))

    ((let-values "mktmp" () e0 args
         bindings tmps body)
     (call-with-values
       (lambda () e0)
       (lambda args
         (let-values "bind"
             bindings tmps body))))

    ((let-values "mktmp" (a . b) e0 (arg ...)
         bindings (tmp ...) body)
     (let-values "mktmp" b e0 (arg ... x)
         bindings (tmp ... (a x)) body))

    ((let-values "mktmp" a e0 (arg ...)
        bindings (tmp ...) body)
     (call-with-values
       (lambda () e0)
       (lambda (arg ... . x)
         (let-values "bind"
             bindings (tmp ... (a x)) body))))))


(define-syntax let*-values
  (syntax-rules ()
    ((let*-values () body0 body1 ...)
     (let () body0 body1 ...))

    ((let*-values (binding0 binding1 ...)
         body0 body1 ...)
     (let-values (binding0)
       (let*-values (binding1 ...)
         body0 body1 ...)))))


(define-syntax define-values
  (syntax-rules ()
    ((define-values () expr)
     (define dummy
       (call-with-values (lambda () expr)
                         (lambda args #f))))
    ((define-values (var) expr)
```

```
 (define var expr))
((define-values (var0 var1 ... varn) expr)
 (begin
   (define var0
     (call-with-values (lambda () expr)
                       list))
   (define var1
     (let ((v (cadr var0)))
       (set-cdr! var0 (cddr var0))
       v)) ...
   (define varn
     (let ((v (cadr var0)))
       (set! var0 (car var0))
       v))))
((define-values (var0 var1 ... . varn) expr)
 (begin
   (define var0
     (call-with-values (lambda () expr)
                       list))
   (define var1
     (let ((v (cadr var0)))
       (set-cdr! var0 (cddr var0))
       v)) ...
   (define varn
     (let ((v (cdr var0)))
       (set! var0 (car var0))
       v))))
((define-values var expr)
 (define var
   (call-with-values (lambda () expr)
                     list)))))
```

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp ...)
     ((lambda () exp ...)))))
```

begin に対する以下の代替展開形はラムダ式の本体に 2 つ以上の式を書くことができる能力を用いていません。いずれにせよ、これらのルールは begin の本体に定義が含まれていない場合にのみ適用することに注意してください。

```
(define-syntax begin
  (syntax-rules ()
    ((begin exp)
     exp)
    ((begin exp1 exp2 ...)
     (call-with-values
         (lambda () exp1)
       (lambda args
         (begin exp2 ...))))))
```

以下の do の構文定義は変数の節を展開するためにトリックを用いています。前述の letrec と同様に、補助マクロを用いることもできます。規定されていない値を取得するために式 (if #f #f) を用いています。

```
(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)))
```

以下に delay、force および delay-force の実装例を示します。以下の式

```
(delay-force ⟨expression⟩)
```

が以下の手続き呼び出し

```
(make-promise #f (lambda () ⟨expression⟩))
```

と同じ意味を持つようにするため、以下のように定義します。

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

また、以下の式

```
(delay ⟨expression⟩)
```

が以下の式

```
(delay-force (make-promise #t ⟨expression⟩))
```

と同じ意味を持つようにするため、以下のように定義します。

```
(define-syntax delay
  (syntax-rules ()
    ((delay expression)
     (delay-force (make-promise #t expression)))))
```

ただし make-promise は以下のように定義されます。

```
(define make-promise
  (lambda (done? proc)
    (list (cons done? proc))))
```

最後に、[38] に倣ってトランポリンテクニックを用い、遅延されていない結果 (つまり delay-force でなく delay で作成された値) が返されるまでプロミス内の手続き式を繰り返し呼ぶよう force を定義します。以下のようになります。

```
(define (force promise)
  (if (promise-done? promise)
      (promise-value promise)
      (let ((promise* ((promise-value promise))))
        (unless (promise-done? promise)
          (promise-update! promise* promise))
        (force promise))))
```

プロミスの各アクセサは以下のようになります。

```
(define promise-done?
  (lambda (x) (car (car x))))
(define promise-value
  (lambda (x) (cdr (car x))))
(define promise-update!
  (lambda (new old)
    (set-car! (car old) (promise-done? new))
    (set-cdr! (car old) (promise-value new))
    (set-car! new (car old))))
```

以下の make-parameter および parameterize の実装はスレッドをサポートしていない処理系用です。ここでは適当な 2 つの唯一オブジェクト <param-set!> および <param-convert> を用い、手続きとしてパラメータオブジェクトを実装しています。

```
(define (make-parameter init . o)
  (let* ((converter
          (if (pair? o) (car o) (lambda (x) x)))
         (value (converter init)))
    (lambda args
      (cond
       ((null? args)
        value)
       ((eq? (car args) <param-set!>)
        (set! value (cadr args)))
       ((eq? (car args) <param-convert>)
        converter)
       (else
        (error "bad parameter syntax"))))))
```

parameterize は dynamic-wind を用いて紐付けられた値を再束縛しています。

```
(define-syntax parameterize
  (syntax-rules ()
    ((parameterize ("step")
                   ((param value p old new) ...)
                   ()
                   body)
     (let ((p param) ...)
       (let ((old (p)) ...
             (new ((p <param-convert>) value)) ...)
         (dynamic-wind
          (lambda () (p <param-set!> new) ...)
          (lambda () . body)
          (lambda () (p <param-set!> old) ...)))))
    ((parameterize ("step")
                   args
                   ((param value) . rest)
                   body)
     (parameterize ("step")
                   args
                   rest
                   (let ((param value))
                     body)))
    ((parameterize ("step")
                   ((param value p old new) . args)
                   rest
                   body))
    ((parameterize ((param value) ...) . body)
     (parameterize ("step")
                   ()
                   ((param value) ...)
                   body))))
```

以下の guard の実装は guard-aux と呼ばれる補助マクロに依存しています。

```
(define-syntax guard
  (syntax-rules ()
    ((guard (var clause ...) e1 e2 ...)
     ((call/cc
       (lambda (guard-k)
         (with-exception-handler
          (lambda (condition)
            ((call/cc
              (lambda (handler-k)
                (guard-k
                 (lambda ()
                   (let ((var condition))
                     (guard-aux
                       (handler-k
                         (lambda ()
                           (raise-continuable condition)))
                       clause ...))))))))
          (lambda ()
            (call-with-values
             (lambda () e1 e2 ...)
             (lambda args
               (guard-k
                 (lambda ()
                   (apply values args)))))))))))))

(define-syntax guard-aux
  (syntax-rules (else =>)
    ((guard-aux reraise (else result1 result2 ...))
     (begin result1 result2 ...))
    ((guard-aux reraise (test => result))
     (let ((temp test))
       (if temp
           (result temp)
           reraise)))
    ((guard-aux reraise (test => result)
                clause1 clause2 ...)
     (let ((temp test))
       (if temp
           (result temp)
           (guard-aux reraise clause1 clause2 ...))))
    ((guard-aux reraise (test))
     (or test reraise))
    ((guard-aux reraise (test) clause1 clause2 ...)
     (let ((temp test))
       (if temp
           temp
           (guard-aux reraise clause1 clause2 ...))))
    ((guard-aux reraise (test result1 result2 ...))
```

```
     (if test
         (begin result1 result2 ...)
         reraise))
    ((guard-aux reraise
                (test result1 result2 ...)
                clause1 clause2 ...)
     (if test
         (begin result1 result2 ...)
         (guard-aux reraise clause1 clause2 ...)))))


(define-syntax case-lambda
  (syntax-rules ()
    ((case-lambda (params body0 ...) ...)
     (lambda args
       (let ((len (length args)))
         (let-syntax
             ((cl (syntax-rules ::: ()
                    ((cl)
                     (error "no matching clause"))
                    ((cl ((p :::) . body) . rest)
                     (if (= len (length '(p :::)))
                         (apply (lambda (p :::)
                                  . body)
                                args)
                         (cl . rest)))
                    ((cl ((p ::: . tail) . body)
                         . rest)
                     (if (>= len (length '(p :::)))
                         (apply
                          (lambda (p ::: . tail)
                            . body)
                          args)
                         (cl . rest))))))
               (cl (params body0 ...) ...)))))))
```

この cond-expand の定義は features 手続きと協調していません。処理系が提供している各々の機能識別子を明示的に記述する必要があります。

```
(define-syntax cond-expand
  ;; Extend this to mention all feature ids and libraries
  (syntax-rules (and or not else r7rs library scheme base)
    ((cond-expand)
     (syntax-error "Unfulfilled cond-expand"))
    ((cond-expand (else body ...))
     (begin body ...))
    ((cond-expand ((and) body ...) more-clauses ...)
     (begin body ...))
    ((cond-expand ((and req1 req2 ...) body ...)
                  more-clauses ...)
     (cond-expand
       (req1
         (cond-expand
           ((and req2 ...) body ...)
           more-clauses ...))
       more-clauses ...))
    ((cond-expand ((or) body ...) more-clauses ...)
     (cond-expand more-clauses ...))
    ((cond-expand ((or req1 req2 ...) body ...)
                  more-clauses ...)
     (cond-expand
       (req1
        (begin body ...))
       (else
        (cond-expand
           ((or req2 ...) body ...)
           more-clauses ...))))
    ((cond-expand ((not req) body ...)
                  more-clauses ...)
     (cond-expand
       (req
         (cond-expand more-clauses ...))
       (else body ...)))
    ((cond-expand (r7rs body ...)
                  more-clauses ...)
       (begin body ...))
    ;; Add clauses here for each
    ;; supported feature identifier.
    ;; Samples:
    ;; ((cond-expand (exact-closed body ...)
    ;;               more-clauses ...)
    ;;   (begin body ...))
    ;; ((cond-expand (ieee-float body ...)
    ;;               more-clauses ...)
    ;;   (begin body ...))
    ((cond-expand ((library (scheme base))
                   body ...)
                  more-clauses ...)
      (begin body ...))
    ;; Add clauses here for each library
    ((cond-expand (feature-id body ...)
                  more-clauses ...)
       (cond-expand more-clauses ...))
    ((cond-expand ((library (name ...))
                   body ...)
                  more-clauses ...)
       (cond-expand more-clauses ...))))
```

付録 A. 標準ライブラリ

この節では標準ライブラリによって提供されるエクスポートの一覧を掲載しています。ライブラリは処理系によってはサポートされない可能性のある機能やロードに時間がかかる機能を分離するように編成されています。

すべての標準ライブラリに対して scheme ライブラリ接頭辞が使われています。またこれは将来の標準で使用するために予約されています。

base ライブラリ

(scheme base) ライブラリは伝統的に Scheme と紐付けられている多数の手続きと構文の束縛をエクスポートしています。base ライブラリと他の標準ライブラリは構造ではなく用途によって分けられています。特に他の標準手続きや構文に基づいた機能ではなく一般的にコンパイラやランタイムシステムによりプリミティブとして実装されている機能には base ライブラリの一部ではなく別のライブラリで定義されているものもあります。同様に base ライブラリのエクスポートには他のエクスポートに基づいて実装できるものもあります。これらは厳密に言うと冗長ではあるものの共通の使用パターンを捕えており、そのため便利な短縮形として提供されています。

```
*                                 +
-                                 ...
/                                 <
<=                                =
=>                                >
>=                                _
abs                               and
append                            apply
assoc                             assq
assv                              begin
binary-port?                      boolean=?
boolean?                          bytevector
bytevector-append                 bytevector-copy
bytevector-copy!                  bytevector-length
bytevector-u8-ref                 bytevector-u8-set!
bytevector?                       caar
cadr
call-with-current-continuation
call-with-port                    call-with-values
call/cc                           car
case                              cdar
cddr                              cdr
ceiling                           char->integer
char-ready?                       char<=?
char<?                            char=?
char>=?                           char>?
char?                             close-input-port
close-output-port                 close-port
complex?                          cond
cond-expand                       cons
current-error-port                current-input-port
current-output-port               define
define-record-type                define-syntax
define-values                     denominator
do                                dynamic-wind
else                              eof-object
eof-object?                       eq?
equal?                            eqv?
error                             error-object-irritants
error-object-message              error-object?
even?                             exact
exact-integer-sqrt                exact-integer?
exact?                            expt
features                          file-error?
floor                             floor-quotient
floor-remainder                   floor/
flush-output-port                 for-each
gcd                               get-output-bytevector
get-output-string                 guard
if                                include
include-ci                        inexact
inexact?                          input-port-open?
input-port?                       integer->char
integer?                          lambda
lcm                               length
let                               let*
let*-values                       let-syntax
let-values                        letrec
letrec*                           letrec-syntax
list                              list->string
list->vector                      list-copy
list-ref                          list-set!
list-tail                         list?
make-bytevector                   make-list
make-parameter                    make-string
make-vector                       map
max                               member
memq                              memv
min                               modulo
negative?                         newline
not                               null?
number->string                    number?
numerator                         odd?
open-input-bytevector             open-input-string
open-output-bytevector            open-output-string
or                                output-port-open?
output-port?                      pair?
parameterize                      peek-char
peek-u8                           port?
positive?                         procedure?
quasiquote                        quote
quotient                          raise
raise-continuable                 rational?
rationalize                       read-bytevector
read-bytevector!                  read-char
read-error?                       read-line
read-string                       read-u8
real?                             remainder
reverse                           round
set!                              set-car!
set-cdr!                          square
string                            string->list
string->number                    string->symbol
string->utf8                      string->vector
string-append                     string-copy
```

```
string-copy!              string-fill!
string-for-each           string-length
string-map                string-ref
string-set!               string<=?
string<?                  string=?
string>=?                 string>?
string?                   substring
symbol->string            symbol=?
symbol?                   syntax-error
syntax-rules              textual-port?
truncate                  truncate-quotient
truncate-remainder        truncate/
u8-ready?                 unless
unquote                   unquote-splicing
utf8->string              values
vector                    vector->list
vector->string            vector-append
vector-copy               vector-copy!
vector-fill!              vector-for-each
vector-length             vector-map
vector-ref                vector-set!
vector?                   when
with-exception-handler    write-bytevector
write-char                write-string
write-u8                  zero?
```

case-lambda ライブラリ

(scheme case-lambda) ライブラリは case-lambda 構文をエクスポートしています。

```
case-lambda
```

char ライブラリ

(scheme char) ライブラリには Unicode 全体をサポートすると巨大なテーブルが必要になる可能性がある文字を扱う手続きが提供されています。

```
char-alphabetic?          char-ci<=?
char-ci<?                 char-ci=?
char-ci>=?                char-ci>?
char-downcase             char-foldcase
char-lower-case?          char-numeric?
char-upcase               char-upper-case?
char-whitespace?          digit-value
string-ci<=?              string-ci<?
string-ci=?               string-ci>=?
string-ci>?               string-downcase
string-foldcase           string-upcase
```

complex ライブラリ

(scheme complex) ライブラリは一般的に実数でない数値でのみ役に立つ手続きをエクスポートしています。

```
angle                     imag-part
magnitude                 make-polar
make-rectangular          real-part
```

cxr ライブラリ

(scheme cxr) ライブラリは car および cdr を 3 つまたは 4 つ合成した 24 個の手続きをエクスポートしています。例えば caddar は以下のように定義できます。

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

手続き car および cdr 自身、およびそれらの 2 段合成である 4 個の手続きは base ライブラリに含まれています。6.4 節を参照してください。

```
caaaar                    caaadr
caaar                     caadar
caaddr                    caadr
cadaar                    cadadr
cadar                     caddar
cadddr                    caddr
cdaaar                    cdaadr
cdaar                     cdadar
cdaddr                    cdadr
cddaar                    cddadr
cddar                     cdddar
cddddr                    cdddr
```

eval ライブラリ

(scheme eval) ライブラリは Scheme のデータをプログラムとして評価するための手続きをエクスポートしています。

```
environment               eval
```

file ライブラリ

(scheme file) ライブラリにはファイルにアクセスするための手続きが提供されています。

```
call-with-input-file      call-with-output-file
delete-file               file-exists?
open-binary-input-file    open-binary-output-file
open-input-file           open-output-file
with-input-from-file      with-output-to-file
```

inexact ライブラリ

(scheme inexact) ライブラリは一般的に不正確な値を扱う場合にのみ役に立つ手続きをエクスポートしています。

```
acos                      asin
atan                      cos
exp                       finite?
infinite?                 log
nan?                      sin
sqrt                      tan
```

lazy ライブラリ

(scheme lazy) ライブラリは遅延評価のための手続きと構文キーワードをエクスポートしています。

```
delay                    delay-force
force                    make-promise
promise?
```

load ライブラリ

(scheme load) ライブラリはファイルから Scheme の式をロードする手続きをエクスポートしています。

```
load
```

process-context ライブラリ

(scheme process-context) ライブラリはプログラムを呼び出している文脈にアクセスするための手続きをエクスポートしています。

```
command-line             emergency-exit
exit
get-environment-variable
get-environment-variables
```

read ライブラリ

(scheme read) ライブラリには Scheme のオブジェクトを読み取る手続きが提供されています。

```
read
```

repl ライブラリ

(scheme repl) ライブラリは interaction-environment 手続きをエクスポートしています。

```
interaction-environment
```

time ライブラリ

(scheme time) ライブラリには時間に関係する値へのアクセスが提供されています。

```
current-jiffy            current-second
jiffies-per-second
```

write ライブラリ

(scheme write) ライブラリには Scheme のオブジェクトを書き出すための手続きが提供されています。

```
display                  write
write-shared             write-simple
```

r5rs ライブラリ

(scheme r5rs) ライブラリには R^5RS で定義されている識別子が提供されています。ただし transcript-on および transcript-off はありません。exact および inexact 手続きはそれぞれその R^5RS での名前である inexact->exact および exact->inexact として現れることに注意してください。ただし処理系が特定のライブラリ (例えば complex ライブラリ) を提供していない場合、それに対応する識別子はこのライブラリにも現れません。

```
*                        +
-                        /
<                        <=
=                        >
>=                       abs
acos                     and
angle                    append
apply                    asin
assoc                    assq
assv                     atan
begin                    boolean?
caaaar                   caaadr
caaar                    caadar
caaddr                   caadr
caar                     cadaar
cadadr                   cadar
caddar                   cadddr
caddr                    cadr
call-with-current-continuation
call-with-input-file     call-with-output-file
call-with-values         car
case                     cdaaar
cdaadr                   cdaar
cdadar                   cdaddr
cdadr                    cdar
cddaar                   cddadr
cddar                    cdddar
cddddr                   cdddr
cddr                     cdr
ceiling                  char->integer
char-alphabetic?         char-ci<=?
char-ci<?                char-ci=?
char-ci>=?               char-ci>?
char-downcase            char-lower-case?
char-numeric?            char-ready?
char-upcase              char-upper-case?
char-whitespace?         char<=?
char<?                   char=?
char>=?                  char>?
char?                    close-input-port
close-output-port        complex?
cond                     cons
cos                      current-input-port
current-output-port      define
define-syntax            delay
denominator              display
do                       dynamic-wind
eof-object?              eq?
equal?                   eqv?
eval                     even?
exact->inexact           exact?
exp                      expt
floor                    for-each
force                    gcd
if                       imag-part
inexact->exact           inexact?
input-port?              integer->char
integer?                 interaction-environment
lambda                   lcm
length                   let
```

```
let*                        let-syntax
letrec                      letrec-syntax
list                        list->string
list->vector                list-ref
list-tail                   list?
load                        log
magnitude                   make-polar
make-rectangular            make-string
make-vector                 map
max                         member
memq                        memv
min                         modulo
negative?                   newline
not                         null-environment
null?                       number->string
number?                     numerator
odd?                        open-input-file
open-output-file            or
output-port?                pair?
peek-char                   positive?
procedure?                  quasiquote
quote                       quotient
rational?                   rationalize
read                        read-char
real-part                   real?
remainder                   reverse
round
scheme-report-environment
set!                        set-car!
set-cdr!                    sin
sqrt                        string
string->list                string->number
string->symbol              string-append
string-ci<=?                string-ci<?
string-ci=?                 string-ci>=?
string-ci>?                 string-copy
string-fill!                string-length
string-ref                  string-set!
string<=?                   string<?
string=?                    string>=?
string>?                    string?
substring                   symbol->string
symbol?                     tan
truncate                    values
vector                      vector->list
vector-fill!                vector-length
vector-ref                  vector-set!
vector?                     with-input-from-file
with-output-to-file         write
write-char                  zero?
```

付録 B. 標準の機能識別子

処理系は `cond-expand` および `features` で用いるために以下の一覧にある機能識別子の一部またはすべてを提供しても構いません。ただし機能が提供されていない場合は、それに対応する機能識別子も提供してはなりません。

`r7rs`

すべての R^{7}RS Scheme 処理系はこの機能を持つ。

`exact-closed`

`/` を除くすべての代数演算は正確な入力が与えられると正確な値を生成する。

`exact-complex`

正確な複素数が提供されている。

`ieee-float`

不正確な数値は IEEE 754 二進浮動小数点の値である。

`full-unicode`

Unicode バージョン 6.0 に存在するすべての文字が Scheme の文字としてサポートされている。

`ratios`

正確な引数で `/` を呼ぶと除数がゼロでなければ正確な結果を生成する。

`posix`

この処理系は POSIX システム上で実行されている。

`windows`

この処理系は Windows 上で実行されている。

`unix, darwin, gnu-linux, bsd, freebsd, solaris, ...`

オペレーティングシステムのフラグ (おそらくひとつ以上)。

`i386, x86-64, ppc, sparc, jvm, clr, llvm, ...`

CPU アーキテクチャのフラグ。

`ilp32, lp64, ilp64, ...`

C のメモリモデルのフラグ。

`big-endian, little-endian`

バイトオーダーのフラグ。

⟨name⟩

処理系の名前。

⟨name-version⟩

処理系の名前およびバージョン。

言語の変更点

R5RS との非互換点

この節ではこの報告書と「報告書改 5」 [20] の間の非互換点を列挙します。

この一覧には権威がありませんが、正確かつ完全であると信じられています。

- シンボルおよび文字名の大文字小文字の区別はデフォルトになりました。つまりシンボルをある文脈では FOO または Foo と書き別の文脈では foo と書くことができるという仮定の元で書かれたコードは変更する必要があるか、新しい #!fold-case 指令で印をつけるか、include-ci ライブラリ宣言を使ってライブラリにインクルードする必要があるということです。標準の識別子はすべて完全に小文字です。
- syntax-rules 構文は _ (アンダースコア) をワイルドカードとして認識するようになりました。つまりこれを構文変数として使うことはできなくなったということです。リテラルとしては未だ使うことができます。
- R5RS の手続き exact->inexact および inexact->exact はそれぞれ R6RS での名前 inexact および exact に改名されました。これらの名前はより短くより正確です。前者の名前は R5RS ライブラリで未だ利用可能です。
- 文字列の比較 (string<? およびそれ関連の述語による) は文字の比較 (char<? およびそれ関連の述語による) の辞書的な拡張であるという保証は無くなりました。
- 数値リテラル中の # 文字のサポートはもはや要求されなくなりました。
- 指数マーカーとしての文字 s、f、d、l のサポートはもはや要求されなくなりました。
- string->number の実装は引数が明示的な基数接頭辞を持っているときに #f を返すことがもはや認められなくなりました。read およびプログラム中の数値構文と互換でなければなりません。
- 手続き transcript-on および transcript-off は削除されました。

それ以外の R5RS 以降の言語の変更点

この節ではこの報告書と「報告書改 5」 [20] の間のさらなる違いを列挙しています。

この一覧には権威がありませんが、正確かつ完全であると信じられています。

- R5RS におけるマイナーな曖昧な点や不明確な点が色々整理されました。
- コードのカプセル化と共有を向上させるための新しいプログラム構造としてライブラリが追加されました。既存の識別子および新規の識別子は分離されたいくつかのライブラリに編成されました。ライブラリは識別子のエクスポートや名前変更を制御しつつ他のライブラリやメインプログラムにインポートされます。ライブラリの内容は使われる処理系の機能に応じた条件分岐ができるようになりました。R5RS 互換ライブラリがあります。
- 式型 include、include-ci および cond-expand が base ライブラリに追加されました。これらは対応するライブラリ宣言と同じ意味論を持っています。
- raise、raise-continuable あるいは error を用いて明示的に例外を通知できるようになり、with-exception-handler および guard 構文を用いて例外を処理できるようになりました。エラーコンディションとして任意のオブジェクトを指定できます。error によって通知される処理系定義のコンディションにはそれを検出するための述語および error に渡された引数を取得するためのアクセサ関数があります。read およびファイル関連の手続きによって通知されるコンディションにもそれらを検出するための述語があります。
- SRFI 9 [19] の define-record-type を用いて複数のフィールドへのアクセスをサポートする新しい独立した型を生成できるようになりました。
- make-parameter を用いてパラメータオブジェクトを作成し、parameterize を用いてそれを動的に再束縛できます。手続き current-input-port および current-output-port はパラメータオブジェクトになりました。新しく導入された current-error-port も同様です。
- プロミスに対するサポートが SRFI 45 [38] に基づいて強化されました。
- **バイトベクタ**と呼ばれる 0〜255 の正確整数のベクタが新しい独立した型として追加されました。ベクタの手続きのサブセットが提供されています。UTF-8 文字エンコーディングに従ってバイトベクタと文字列を相互に変換できます。バイトベクタにはデータム表現があり、それ自身に評価されます。
- ベクタ定数はそれ自身に評価されます。
- 手続き read-line が提供され、行指向のテキスト入力が簡単になりました。
- 手続き flush-output-port が提供され、出力ポートのバッファリングに対する最低限の制御ができるようになりました。
- **ポート**は**テキスト**ポートまたは**バイナリ**ポートに指定されるようになり、バイナリデータを読み書きするための新しい手続きが提供されます。ポートが開いているか閉じられたかを返す新しい述語 input-port-open? および output-port-open? が追加されました。ポート

を閉じる新しい手続き close-port が追加されました。ポートが入力と出力を両方兼ねている場合はその両方が閉じられます。

- 文字列に対して文字の読み書きを行う**文字列ポート**とバイトベクタに対してバイトの読み書きを行う**バイトベクタポート**が追加されました。
- 文字列およびバイトベクタに固有の入出力手続きがあります。
- write 手続きを循環オブジェクトに適用するとデータムラベルを生成するようになりました。ラベルを生成しない新しい手続き write-simple とすべての共有構造および循環構造に対してラベルを生成する新しい手続き write-shared が追加されました。display 手続きは循環オブジェクトに対してループしてはならなくなりました。
- R^{6}RS の手続き eof-object が追加されました。EOF オブジェクトは独立した型であることが要求されるようになりました。
- 変数定義ができる場所ならどこでも構文定義ができるようになりました。
- syntax-rules 構文は省略記号のシンボルをデフォルトの ... の代わりに明示的に指定できるようになり、省略記号を接頭したリストによりテンプレートをエスケープできるようになり、また省略記号のパターンに続く末尾のパターンを指定できるようになりました。
- マクロ展開時に即座により詳細なエラーを通知する手段として syntax-error 構文が追加されました。
- letrec* 束縛構文が追加され、内部 define がそれを基に規定されるようになりました。
- define-values、let-values および let*-values によって多値の捕捉に対するサポートが強化されました。式の並びを持つ標準の式型はその並びの最後以外のすべての式の継続にゼロ個または 2 個以上の値を渡すことができるようになりました。
- case 条件分岐で => 構文がサポートされるようになりました。これは cond の真似ですが、通常の節だけでなく else 節でも同様に使用できます。
- 手続きに渡された引数の個数に基づいて分岐する case-lambda がそれ専用のライブラリに追加されました。
- 便利な条件分岐 when および unless が追加されました。
- 不正確な数値に対する eqv? の動作が R^{6}RS の定義に沿ったものになりました。
- eq? および eqv? を手続きに適用した場合に異なる答えを返すことが許されるようになりました。
- R^{6}RS の手続き boolean=? および symbol=? が追加されました。
- 正の無限大、負の無限大、NaN および負の不正確なゼロが不正確な値として数塔に追加されました。それぞれ +inf.0、-inf.0、+nan.0、-0.0 と書き表されます。これらのサポートは要求されません。表現 -nan.0 は +nan.0 と同義です。
- log 手続きは対数の底を指定する第 2 引数を取るようになりました。
- 手続き map および for-each は最も短いリストで停止することが要求されるようになりました。
- 手続き member および assoc は使用するべき等値述語を指定するオプショナルな第 3 引数を取るようになりました。
- 数値手続き finite?、infinite?、nan?、exact-integer?、square および exact-integer-sqrt が追加されました。
- -、/ 手続きおよび文字比較、文字列比較の述語は 3 つ以上の引数のサポートが要求されるようになりました。
- 形式 #true および #false が #t および #f と同様にサポートされるようになりました。
- 手続き make-list、list-copy、list-set!、string-map、string-for-each、string->vector、vector-append、vector-copy、vector-map、vector-for-each、vector->string、vector-copy!、string-copy! がシーケンス操作の完全化のために追加されました。
- 文字列およびベクタの手続きのいくつかはオプショナルな *start*および *end*引数を用いた文字列またはベクタの部分的な処理がサポートされます。
- リストの手続きのいくつかは循環リストに対して定義されるようになりました。
- 処理系は ASCII を含む完全な Unicode レパートリーの任意のサブセットを提供しても構いませんが、そういったサブセットは Unicode と矛盾しないようにサポートしなければなりません。様々な文字および文字列の手続きがそれに従って拡張され、文字列用の大文字小文字変換手続きが追加されました。文字列の比較と文字の比較の一貫性を保つことはもはや要求されなくなりました。文字の比較は Unicode スカラー値にのみ基づいて行われます。数値的な文字の数の値を取得する新しい digit-value 手続きが追加されました。
- 2 つのコメント構文、次のデータムまでスキップするための #; およびネスト可能なブロックコメントのための #| ... |# が追加されました。
- データムラベル #<n>= を接頭したデータを #<n># で参照することができ、共有構造を持つデータを読み書きできます。

- 文字列およびシンボルでニーモニックおよび数値によるエスケープシーケンスが使えるようになり、名前付き文字の一覧が拡張されました。
- 手続き `file-exists?` および `delete-file` が `(scheme file)` ライブラリで利用可能です。
- システム環境、コマンドライン、およびプロセスの終了状態へのインタフェースが `(scheme process-context)` ライブラリで利用可能です。
- 時間関係の値にアクセスする手続きが `(scheme time)` ライブラリで利用可能です。
- 整数除算演算の変種の小さなセットが新しいより明確な名前で提供されています。
- `load` 手続きはロード先の環境を指定する第 2 引数を取るようになりました。
- `call-with-current-continuation` 手続きに同義の `call/cc` が追加されました。
- read-eval-print loop の意味論が部分的に定められ、手続きの再定義が遡って効果を持つよう要求されるようになりましたが、構文キーワードはそうではありません。
- 形式意味論で `dynamic-wind` を処理するようになりました。

R^{6}RS との非互換点

この節では R^{7}RS と「報告書改 ^{6}」 [33] およびそれに付属している標準ライブラリドキュメントの間の非互換点を列挙します。

この一覧には権威がなく、不完全な可能性があります。

- R^{7}RS のライブラリは `library` ではなく `define-library` キーワードで始まります。これは R^{6}RS のライブラリと構文的に区別できるようにするためです。R^{7}RS の用語で言うと、R^{6}RS のライブラリは単一のエクスポート宣言に続く単一のインポート宣言に続く命令と定義から成ります。R^{7}RS では命令と定義は本体に直接書くことはできません。それらは `begin` ライブラリ宣言にラップする必要があります。
- `include`、`include-ci`、`include-library-declarations` および `cond-expand` ライブラリ宣言に直接対応するものは R^{6}RS にありません。一方で R^{7}RS のライブラリはフェーズやバージョン仕様をサポートしていません。
- 標準化された識別子をライブラリにグループ化する方針が R^{6}RS と異なっています。特に明示的にせよ暗黙的にせよ R^{5}RS でオプショナルとなっている手続きは base ライブラリから削除されました。絶対的な要求事項は base ライブラリのみです。
- 識別子構文の形式は提供されません。
- 内部構文定義は使用可能ですが、定義されるよりも前にその構文形式の使用が現れることはできません。R^{6}RS にある `even`/`odd` の例は許容されません。
- R^{6}RS の例外システムはそのまま導入されましたが、コンディション型は規定されていないままにしています。特に R^{6}RS では特定の型のコンディションの通知が要求されるところで R^{7}RS では「それはエラーです」とだけ述べ、通知の問題はオープンなままになっています。
- 完全な Unicode のサポートは要求されません。正規化は提供されません。文字の比較は Unicode によって定義されますが、文字列の比較は処理系依存です。非 Unicode 文字が許容されています。
- 完全な数塔は R^{5}RS と同様にオプショナルですが、IEEE の無限大、NaN、-0.0 のオプショナルなサポートが R^{6}RS から採用されました。数値の結果の明確化もほとんど採用されましたが、R^{6}RS の手続き `real-valued?`、`rational-valued?` および `integer-valued?` は採用されていません。R^{6}RS の除算演算子 `div`、`mod`、`div-and-mod`、`div0`、`mod0` および `div0-and-mod0` は提供されません。
- 結果が規定されていないときでも、それが単一の値であることが要求されます。しかし本体の最後でない式は任意の数の値を返せます。
- `map` および `for-each` の意味論が変更され、SRFI 1 [30] の早期終了動作を用いるようになりました。同様に、`assoc` および `member` が変更され、R^{6}RS にある別個の `assp` および `memp` 手続きの代わりに、SRFI 1 のようにオプショナルな `equal?` 引数を取るようになりました。
- R^{6}RS の `quasiquote` の明確化が採用されました。ただし `unquote` および `unquote-splicing` の多引数化は除かれています。
- R^{6}RS の仮数部の幅を指定する方法は採用されていません。
- 文字列ポートは R^{6}RS の代わりに SRFI 6 [7] と互換です。
- R^{6}RS 形式のバイトベクタが導入されていますが、符号無しバイト (`u8`) の手続きのみ提供されています。字句的構文は R^{6}RS の`#vu8` 形式ではなく SRFI 4 [13] と互換性のある`#u8` を用います。
- 便利なマクロ `when` および `unless` が提供されていますが、結果は規定されていないままになっています。
- 標準ライブラリドキュメントの残りの機能は将来の標準化の努力に残されています。

追加資料

http://schemers.org にある Scheme コミュニティのウェブサイトには学習、プログラミング、仕事、イベントに対する追加のリソースおよび Scheme のユーザーグループの情報が掲載されています。

http://library.readscheme.org にある Scheme に関する研究の参考文献は技術論文や Scheme 言語に関するそれらへリンクされています。古典的な論文や最近の研究も含まれています。

オンラインの Scheme の議論は irc.freenode.net の #scheme チャンネルで IRC を用いて行われています。また Usenet のディスカッショングループ comp.lang.scheme でも行われています。

例

手続き integrate-system は微分方程式系

$$y_k' = f_k(y_1, y_2, \ldots, y_n),\ k = 1, \ldots, n$$

をルンゲ=クッタ法で積分します。

パラメータ system-derivative は系の状態 (状態変数 $y_1, \ldots, y_n$ に対する値のベクタ) を取り系の微分係数 (値 $y_1', \ldots, y_n'$) を生成する関数です。パラメータ initial-state は系の初期状態を与え、h は積分ステップの長さの初期推定値です。

integrate-system の戻り値は系の状態の無限ストリームです。

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

手続き runge-kutta-4 は系の状態から系の微分係数を生成する関数 f を取ります。これは系の状態を取り新しい系の状態を生成する関数を生成します。

```
(define (runge-kutta-4 f h)
  (let ((*h (scale-vector h))
        (*2 (scale-vector 2))
        (*1/2 (scale-vector (/ 1 2)))
        (*1/6 (scale-vector (/ 1 6))))
    (lambda (y)
      ;; y is a system state
      (let* ((k0 (*h (f y)))
             (k1 (*h (f (add-vectors y (*1/2 k0)))))
             (k2 (*h (f (add-vectors y (*1/2 k1)))))
             (k3 (*h (f (add-vectors y k2)))))
        (add-vectors y
          (*1/6 (add-vectors k0
                             (*2 k1)
                             (*2 k2)
                             k3)))))))

(define (elementwise f)
  (lambda vectors
    (generate-vector
     (vector-length (car vectors))
     (lambda (i)
       (apply f
              (map (lambda (v) (vector-ref  v i))
                   vectors))))))

(define (generate-vector size proc)
  (let ((ans (make-vector size)))
    (letrec ((loop
              (lambda (i)
                (cond ((= i size) ans)
```

```
                    (else
                     (vector-set! ans i (proc i))
                     (loop (+ i 1)))))))
      (loop 0))))

(define add-vectors (elementwise +))

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

map-streams 手続きは map のストリーム版です。第 1 引数 (手続き) を第 2 引数 (ストリーム) のすべての要素に適用します。

```
(define (map-streams f s)
  (cons (f (head s))
        (delay (map-streams f (tail s)))))
```

無限ストリームは car がストリームの最初の要素を持ち cdr がストリームの残りを供給するプロミスを持つペアとして実装されています。

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

integrate-system の使い方として減衰振動をモデル化した系

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

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

を積分する例を以下に示します。

```
(define (damped-oscillator R L C)
  (lambda (state)
    (let ((Vc (vector-ref state 0))
          (Il (vector-ref state 1)))
      (vector (- 0 (+ (/ Vc (* R C)) (/ Il C)))
              (/ Vc L)))))

(define the-states
  (integrate-system
     (damped-oscillator 10000 1000 .001)
     '#(1 0)
     .01))
```

参考文献

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs, second edition.* MIT Press, Cambridge, 1996.

[2] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, pages 86–95.

[3] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. http://www.ietf.org/rfc/rfc2119.txt, 1997.

[4] Robert G. Burger and R. Kent Dybvig. Printing floating-point numbers quickly and accurately. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116.

[5] William Clinger. How to read floating point numbers accurately. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 92–101. Proceedings published as *SIGPLAN Notices* 25(6), June 1990.

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

[7] William Clinger. SRFI 6: Basic String Ports. http://srfi.schemers.org/srfi-6/, 1999.

[8] William Clinger, editor. The revised revised report on Scheme, or an uncommon Lisp. MIT Artificial Intelligence Memo 848, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

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

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

[11] Mark Davis. Unicode Standard Annex #29, Unicode Text Segmentation. http://unicode.org/reports/tr29/, 2010.

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

[13] Marc Feeley. SRFI 4: Homogeneous Numeric Vector Datatypes. http://srfi.schemers.org/srfi-45/, 1999.

[14] Carol Fessenden, William Clinger, Daniel P. Friedman, and Christopher Haynes. Scheme 311 version 4 reference manual. Indiana University Computer Science Technical Report 137, February 1983. Superseded by [15].

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

[16] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life." In *Scientific American*, 223:120–123, October 1970.

[17] *IEEE Standard 754-2008. IEEE Standard for Floating-Point Arithmetic.* IEEE, New York, 2008.

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

[19] Richard Kelsey. SRFI 9: Defining Record Types. `http://srfi.schemers.org/srfi-9/`, 1999.

[20] Richard Kelsey, William Clinger, and Jonathan Rees, editors. The revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7-105, 1998.

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

[22] Eugene E. Kohlbecker Jr., Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161.

[23] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM* 3(4):184–195, April 1960.

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

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

[26] Paul Penfield, Jr. Principal values and branch cuts in complex APL. In *APL '81 Conference Proceedings,* pages 248–256. ACM SIGAPL, San Francisco, September 1981. Proceedings published as *APL Quote Quad* 12(1), ACM, September 1981.

[27] Jonathan A. Rees and Norman I. Adams IV. T: A dialect of Lisp or, lambda: The ultimate software tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

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

[29] Jonathan Rees and William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *ACM SIGPLAN Notices* 21(12), pages 37–79, December 1986.

[30] Olin Shivers. SRFI 1: List Library. `http://srfi.schemers.org/srfi-1/`, 1999.

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

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

[33] Michael Sperber, R. Kent Dybvig, Mathew Flatt, and Anton van Straaten, editors. *The revised^6 report on the algorithmic language Scheme.* Cambridge University Press, 2010.

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

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

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

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

[38] Andre van Tonder. SRFI 45: Primitives for Expressing Iterative Lazy Algorithms. `http://srfi.schemers.org/srfi-45/`, 2002.

[39] Martin Gasbichler, Eric Knauel, Michael Sperber and Richard Kelsey. How to Add Threads to a Sequential Language Without Getting Tangled Up. *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, November 2003.

概念、キーワード、手続き定義の アルファベット順の索引

各項目、手続き、キーワードの主なエントリは他のエントリからセミコロンで区切られて最初に示されています。