アルゴリズム言語Scheme改^{7}の概要

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 25, 2017

SCHEME の概要

この文章は R7RS の小さな言語の概要を記載しています。この概要の目的は、リファレンスマニュアルとして編成されている R7RS の理解を手助けするために、Scheme の基本概念について十分な説明をすることです。そのため、この概要は Scheme の完全な入門書ではありませんし、すべての事柄や基準について、いかなる意味でも正確に述べているわけではありません。

Algol に倣い、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 は暗黙の型を持つ言語です。これは明示的な型と反対の概念です。型は変数ではなくオブジェクト (値とも呼ばれます) に紐付けられます。(暗黙の型を持つ言語のことを、型を持たない言語、弱い型を持つ言語、動的な型を持つ言語などと呼ぶ場合もあります。) 暗黙の型を持つ言語は他に Python、Ruby、Smalltalk などがあります。他の Lisp 方言もそうです。明示的な型を持つ言語 (強い型を持つ言語や静的な型を持つ言語と呼ばれることもあります) には Algol 60、C、C#、Java、Haskell、ML などがあります。

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 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。ほとんどのオブジェクトが無制限の生存期間を持つ言語には C#、Java、Haskell、ほとんどの Lisp 方言、ML、Python、Ruby、Smalltalk などがあります。

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。

Scheme はオブジェクトとしての手続きをサポートした最初の言語のひとつです。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。このような特徴を持つ言語には Common Lisp、Haskell、ML、Ruby、Smalltalk などがあります。

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。第一級の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。

Scheme では手続き呼び出しの引数の式は、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。C、C#、Common Lisp、Python、Ruby、Smalltalk などが手続き呼び出しの前に必ず引数式を評価する言語の例です。これは手続きが必要としない限り引数が評価されない Haskell の遅延評価の意味論や Algol 60 の名前渡しの意味論とはまったく異なるものです。

Scheme の数値計算モデルには豊富な数値型とその演算子が提供されています。さらに正確な数値と不正確な数値を区別しています。原則として、正確な数値オブジェクトは正確にあるひとつの数値に対応するもので、不正確な数値は丸めやその他の近似を伴った計算の結果です。

1. 基本的な型

Scheme のプログラムはオブジェクトを操作します (値と呼ばれることもあります)。Scheme のオブジェクトは型と呼ばれる値の集合に分類されます。この章では Scheme 言語の基礎となる重要な型の概要を述べます。

メモ: Scheme は暗黙の型を持つので、Scheme の文脈における型という用語の用途は他の言語、特に明示的な型を持つ言語における文脈での用途とは異なります。

数値 Scheme は幅広い豊富な数値データ型をサポートしています。任意精度の整数、有理数、複素数や様々な型の不正確な数値などがあります。

ブーリアン ブーリアンは真偽値です。真または偽のいずれかとなります。Scheme では「偽」のオブジェクトは `#f` と書きます。「真」のオブジェクトは `#t` と書きます。しかし真偽値を必要とする場所のほとんどでは、`#f` 以外のすべてのオブジェクトが真とみなされます。

ペアとリスト ペアは 2 つの部分を持つデータ構造です。ペアのほとんどの用途は (単方向連結) リストを表現することです。ひとつめの部分 (「car」) がリストの最初の要素を表し、ふたつめの部分 (「cdr」) がリストの残りを表します。Scheme にはさらに独立した空リストがあり、リストを形成するペアのチェーンの最後の cdr となります。

シンボル シンボルは文字列を表すオブジェクトです。シンボルが表す文字列はそのシンボルの名前と呼ばれます。文字列オブジェクトと異なり、同じ綴りの名前を持つ 2 つのシンボルは区別されません。シンボルには多くの活用方法があります。例えば他の言語では列挙型を使うような場面でシンボルを使うことができます。

R5RS と異なり、R7RS ではシンボルおよび識別子の大文字小文字を区別します。

文字 Scheme の文字はテキストの文字にほぼ対応します。より正確にいうと、Unicode 標準のスカラー値のサブセットと同じです。処理系によってはさらに拡張されている場合もあります。

文字列 文字列は有限個の文字の並びで、固定の長さを持ちます。すなわち任意の Unicode テキストを表します。

ベクタ ベクタはリストのように任意のオブジェクトの有限個の並びを表す線形のデータ構造です。リストの要素はそれを表しているペアのチェーンを辿って線形にアクセスされるのに対し、ベクタの要素は整数のインデックスでアクセスされます。そのため要素にランダムアクセスする場合はリストよりベクタの方が適しています。

バイトベクタ バイトベクタはベクタに似ていますが、要素がバイトである点が異なります。つまり 0〜255 の範囲の正確な整数しか格納できません。

手続き Scheme では手続きは値です。

レコード レコードは構造化された値であり、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にひとつずつ場所があります。レコードはレコード型に分類されます。述語、コンストラクタ、フィールドアクセサ、フィールドミューテータがそれぞれのレコード型に対して定義されます。

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。

2. 式

Scheme コードの最も重要な要素が式です。式は、評価して値を生成することができます。最も基礎的な式はリテラル式です。

```
#t                              ⟹ #t
23                              ⟹ 23
```

この表記は式 `#t` が `#t`、つまり「真」を表す値に評価され、式 23 が 23 という数を表す数値に評価されることを意味しています。

複合式は部分式のまわりに括弧を置くことで作られます。最初の部分式は演算を識別し、残りの部分式はその演算の被演算子です。

```
(+ 23 42)                       ⟹ 65
(+ 14 (* 23 42))                ⟹ 980
```

最初の例では、`+` は組み込みの加算演算子の名前で、23 と 42 がその被演算子です。式 `(+ 23 42)` は「23 と 42 の和」と解釈します。複合式は入れ子にできます — ふたつめの例は「23 と 42 の積と 14 の和」と解釈します。

これらの例が示すように、Scheme の複合式は常に同じ前置表記を使って書きます。必然的に構造を示すために括弧が必要となります。そのため数学の表記や多くのプログラミングで許容されている「不要な」括弧は、Scheme では許容されません。

多くのプログラミング言語と同様に、それが式の部分式を分割している場合、ホワイトスペース (改行も含みます) は重要ではなく、構造を表すために使うことができます。

3. 変数と束縛

Scheme では識別子を用いて値を保持する場所を表すことができます。これらの識別子は変数と呼ばれます。多くの場合において、特にその場所の値が作成後に変更されることがない場合、変数はその値を直接表すものとして考えるのが便利でしょう。

```
(let ((x 23)
      (y 42))
  (+ x y))                      ⟹ 65
```

この場合、`let` で始まる式は束縛構文です。`let` の次の括弧で囲まれた部分は変数と式のリストです。変数 `x` と 23、そして変数 `y` と 42 です。`let` 式は `x` を 23 に束縛し、`y` を 42 に束縛します。これらの束縛は `let` 式の本体の中、つまり `(+ x y)` でのみ有効です。

4. 定義

`let` 式で束縛された変数は局所的です。それらの束縛は `let` の本体からのみ見えます。以下のようにして識別子に対する最上位の束縛を作ることもできます。

```
(define x 23)
(define y 42)
(+ x y)                         ⟹ 65
```

(これらは最上位のプログラムまたはライブラリの本体内で実際に「最上位」です。)

最初の 2 つの括弧で囲まれた部分は定義です。これらは `x` を 23 に、`y` を 42 に束縛する最上位の束縛を作成します。定義は式ではありません。式を書けるすべての場所に定義が書けるわけではありません。また定義は値を持ちません。

束縛はプログラムの字句構造に従います。同じ名前の束縛がいくつか存在する場合、変数は最も近い束縛を参照します。プログラム中のその変数が現れた場所から始まり、内側から外側へと進み、その途中で局所的な束縛が見付からなければ最も外側の束縛を参照します。

```
(define x 23)
(define y 42)
(let ((y 43))
  (+ x y))                      ⟹ 66

(let ((y 43))
  (let ((y 44))
    (+ x y)))                   ⟹ 67
```

5. 手続き

定義を使って手続きを定義することもできます。

```
(define (f x)
  (+ x 42))

(f 23)                          ⟹ 65
```

手続きは、いくらか単純化された、オブジェクトに対する式の抽象化です。この例では、ひとつめの定義は f という名前の手続きを定義しています。f x のまわりの括弧は、これが手続き定義であることを表しています。式 (f 23) は手続きの呼び出しであり、おおむね「x を 23 に束縛して (+ x 42) (手続きの本体です) を評価する」というような意味合いです。

手続きはオブジェクトなので他の手続きに渡すことができます。

```
(define (f x)
  (+ x 42))

(define (g p x)
  (p x))

(g f 23)                        ⟹ 65
```

この例では、p を f に束縛し、x を 23 に束縛して、g の本体が評価されます。これは (f 23) と同等であり、すなわち 65 に評価されます。

実のところ、Scheme の定義済み手続きの多くは構文としてではなく、値が手続きである変数として提供されています。例えば + 演算子は他の多くの言語では特別な構文として扱われていますが、Scheme では普通の識別子です。単に数値を加算する手続きに束縛されているだけです。* や他の多くの演算子も同様です。

```
(define (h op x y)
  (op x y))

(h + 23 42)                     ⟹ 65
(h * 23 42)                     ⟹ 966
```

手続き定義は手続きを作る唯一の方法ではありません。lambda 式を使うと名前を指定する必要なしにオブジェクトとしての新しい手続きを作ることができます。

```
((lambda (x) (+ x 42)) 23)  ⟹ 65
```

この例の式全体は手続き呼び出しです。(lambda (x) (+ x 42)) が数値をひとつ取りそれに 42 を加算する手続きに評価されます。

6. 手続き呼び出しと構文キーワード

(+ 23 42) や (f 23)、((lambda (x) (+ x 42)) などがすべて手続き呼び出しの例であるのに対し、lambda 式や let 式はそうではありません。これは let が、識別子ではあるものの変数ではなく、構文キーワードであるためです。最初の部分式に構文キーワードを持つリストは、そのキーワードによって決まる特別な規則に従います。定義で使われる識別子 define も構文キーワードです。従って定義は手続き呼び出しではありません。

lambda キーワードの規則では、最初の部分リストは引数リストで、残りの部分リストは手続きの本体になります。let 式の場合は、最初の部分リストは束縛指定のリストで、残りの部分リストは式の本体を構成します。

こういった式型と手続き呼び出しとは、リストの最初の位置に構文キーワードを見つけることで区別されます。もし最初の位置に構文キーワードがなければ、その式は手続き呼び出しです。Scheme の構文キーワードはとても少なく、この作業は実に簡単になっています。しかし新しい構文キーワードに対する束縛を作ることもできます。

7. 代入

定義や let、lambda で束縛した Scheme の変数は、実際にはそれぞれの束縛で指定されたオブジェクトに直接束縛されるわけではありません。そうではなく、それらのオブジェクトを保持している場所に束縛されます。これらの場所の内容は後に代入によって破壊的に変更できます。

```
(let ((x 23))
  (set! x 42)
  x)                            ⟹ 42
```

この例では let 式の本体には 2 つの式があります。これらは順番に評価され、最後の式の値が let 式全体の値となります。式 (set! x 42) は代入です。「x が参照している場所のオブジェクトを 42 に置き換えなさい」という意味です。つまり x の以前の値 23 が 42 に置き換えられます。

8. 派生構文とマクロ

R[7]RS の小さな言語の一部として規定されている式型の多くは、より基本的な式型に変換できます。例えば let 式は手続き呼び出しと lambda 式に変換できます。以下のふたつの式は同等です。

```
(let ((x 23)
      (y 42))
  (+ x y))                      ⟹ 65

((lambda (x y) (+ x y)) 23 42)
          ⟹ 65
```

let 式のような構文は派生構文と呼ばれます。その意味論が構文変換により他の形の式から派生しているためです。手続きにも派生式として定義されているものがあります。以下のふたつの式は同等です:

```
(define (f x)
  (+ x 42))

(define f
  (lambda (x)
    (+ x 42)))
```

Scheme では構文キーワードをマクロに束縛することでプログラム自身により独自の派生式を作ることができます。

```
(define-syntax def
  (syntax-rules ()
    ((def f (p ...) body)
```

```
    (define (f p ...)
      body))))

(def f (x)
  (+ x 42))
```

この define-syntax 式は、パターン (def f (p ...) body) に一致する括弧で囲まれた構文を (define (f p ...) body) に変換する、という指定です。f、p、body はパターン変数です。従ってこの例の def 式は以下のように変換されます:

```
(define (f x)
  (+ x 42))
```

新しい構文キーワードを作る能力により、Scheme は非常に柔軟で表現力が高い言語となっています。他の言語で組み込まれている多くの機能は Scheme で直接実装することができます。Scheme プログラマーなら誰でも新しい式型を追加できるのです。

9. 構文データムとデータム値

データム値は Scheme のオブジェクトのサブセットです。ブーリアン、数値、文字、シンボル、文字列、および要素がデータム値であるリスト、ベクタ、バイトベクタがこれに含まれます。それぞれのデータム値は構文データムとしてテキストで表現することができ、情報の損失なしに書き出し、読み戻すことができます。それぞれのデータム値に対応する構文データムは一般的にひとつ以上あります。さらに、それぞれのデータム値は対応する構文データムに ’ を前置することでプログラム内のリテラル式に変換することができます。

```
’23                    ⟹ 23
’#t                    ⟹ #t
’foo                   ⟹ foo
’(1 2 3)               ⟹ (1 2 3)
’#(1 2 3)              ⟹ #(1 2 3)
```

この例のうち、シンボルとリスト以外のリテラル定数表現には ’ は不要です。構文データム foo は名前が “foo” であるシンボルを表し、’foo はそのシンボルを値として持つリテラル式です。(1 2 3) は要素が 1、2、3 であるリストを表す構文データムであり、’(1 2 3) はそのリストを値として持つリテラル式です。同様に、#(1 2 3) は要素が 1、2、3 であるベクタを表す構文データムであり、’#(1 2 3) は対応するリテラルです。

構文データムは Scheme の式のスーパーセットです。すなわちデータムを使って Scheme の式をデータオブジェクトとして表現することができます。特にシンボルを使って識別子を表現できます。

```
’(+ 23 42)                    ⟹ (+ 23 42)
’(define (f x) (+ x 42))
          ⟹ (define (f x) (+ x 42))
```

これは Scheme のソースコードを操作するプログラム、特にインタプリタやプログラム変換器を書く基礎となります。

10. 継続

Scheme の式を評価するときは常に、その式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) 未来全体を表現します。例えば以下の式の 3 の継続は、

```
(+ 1 3)
```

それに 1 を加算することです。通常、これらの普遍的に存在している継続は水面下に隠されており、プログラマーはそれらについて考えません。しかしプログラマーが明示的に継続を扱う必要のある状況が稀にあります。現在の継続を復元する手続きを作成する call-with-current-continuation 手続きにより、Scheme プログラマーは継続を扱うことができます。

以下の例では、引数に 1 を加算する継続を表す脱出手続きを escape に束縛し、引数として 3 を与えてそれを呼び出しています。escape への呼び出しの継続は放棄され、代わりに 1 を加算する継続に 3 が渡されます。

```
(+ 1 (call-with-current-continuation
       (lambda (escape)
         (+ 2 (escape 3)))))
          ⟹ 4
```

脱出手続きは無制限の生存期間を持ちます。その継続を補足した呼び出しが戻った後に呼び出すことができ、複数回呼び出すこともできます。これにより call-with-current-continuation は他の言語の例外処理のような典型的な非局所制御構文に比べて非常に強力なものになっています。

11. ライブラリ

Scheme のコードはライブラリと呼ばれる部品にまとめることができます。ライブラリは定義と式を持ち、他のライブラリから定義をインポートしたり、他のライブラリに定義をエクスポートしたりできます。

以下に示す (hello) という名前のライブラリは、hello-world という名前の定義をエクスポートし、base ライブラリと display ライブラリをインポートしています。エクスポートしている hello-world は Hello World を表示して改行する手続きです。

```
(define-library (hello)
  (export hello-world)
  (import (scheme base)
          (scheme display))
  (begin
    (define (hello-world)
      (display "Hello World")
      (newline))))
```

12. プログラム

ライブラリは他のライブラリから、最終的には Scheme のプログラムから呼び出されます。ライブラリ同様に、プログラムはインポート、定義、式を持つことができ、実行の開始点を規定します。そのためプログラムはライブラリインポートの推移閉包を通して Scheme のプログラムを定義します。

以下のプログラムは process-context ライブラリの `command-line` 手続きを使ってコマンドラインから最初の引数を取得します。それから `with-input-from-file` を使ってファイルを開きます。この手続きはファイルを現在の入力ポートとし、最後に閉じるよう手配します。次に `read-line` 手続きを呼んでファイルからテキストを 1 行読み込み、そして `write-string` および `newline` でその行を出力し、それをファイルの終端まで繰り返します。

```
(import (scheme base)
        (scheme file)
        (scheme process-context))
(with-input-from-file
  (cadr (command-line))
  (lambda ()
    (let loop ((line (read-line)))
      (unless (eof-object? line)
        (write-string line)
        (newline)
        (loop (read-line))))))
```

13. 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、評価できる環境です。REPL は base ライブラリをインポートした状態で開始されます。他のライブラリをインポートしていても構いません。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。

以下に短い REPL の対話の様子を示します。文字 `>` は REPL の入力プロンプトを表しています。

```
> ; A few simple things
> (+ 2 2)
4
> (sin 4)
Undefined variable: sin
> (import (scheme inexact))
> (sin 4)
-0.756802495307928
> (define sine sin)
> (sine 4)
-0.756802495307928
> ; Guy Steele's three-part test
> ; True is true ...
> #t
#t
> ; 100!/99! = 100 ...
> (define (fact n)
    (if (= n 0) 1 (* n (fact (- n 1)))))
> (/ (fact 100) (fact 99))
100
> ; If it returns the *right* complex number,
> ; so much the better ...
> (define (atanh x)
    (/ (- (log (+ 1 x))
          (log (- 1 x)))
       2))
> (atanh -2)
-0.549306144334055+1.5707963267949i
```