Overview of the Revised^7 Algorithmic Language Scheme

[アルゴリズム言語 Scheme 報告書 七訂版の概要]

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)

Dedicated to the memory of John McCarthy and Daniel Weinreb のみたまにささぐ

***** EDITOR'S DRAFT *** 2013年 4月 15日**

日本語訳 2014 年 1 月 22 日

SCHEME の概観

本稿では，R^{7}RS の小さな言語の概要を説明する。この概要の目的は，リファレンスマニュアルのように構成されたR^{7}RS 報告書の理解を容易にするために，言語の基本的な概念について十分に説明することである。したがって，この概要はいかなる意味においても，言語への完全な導入でなければ，すべての点や規範において正確でもない。

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

Scheme は，明示型とは対照的に，潜在型をもつ。型は，変数ではなくオブジェクト (値とも呼ばれる) と関連している。(一部の著者は，潜在型の言語を，型指定されていない，弱い型付けまたは動的型付け言語と呼ぶ。) 潜在型の他の言語は，Python や Ruby, Smalltalk, そして他の Lisp 方言である。明示型の言語 (時々強い型付け，または静的型付け言語と呼ばれる) には， Algol 60, C, C#, Java, Haskell, および ML が含まれている。

手続きや継続を含む， Scheme の計算過程で作成されたすべてのオブジェクトは，無期限の寿命をもっている。Scheme のオブジェクトは，破壊されることはない。Scheme の実装が (通常は!) 記憶領域を使い果たさない理由は，あるオブジェクトが将来のいかなる計算にも関係ないことを証明できれば，オブジェクトが占める記憶領域を再利用することを許可されているためである。ほとんどのオブジェクトが無制限の寿命をもっている他の言語は， C#, Java, Haskell, ほとんどの Lisp 方言, ML, Python, Ruby, and Smalltalk がある。

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

Scheme は，それ自体がオブジェクトとして手続きをサポートする最初の言語の一つであった。手続きを動的に作成すること，データ構造の中に格納すること，手続きの結果として返すことなどが可能である。これらの特性を有する他の言語は，Common Lisp, Haskell, ML, Ruby, そして Smalltalk がある。

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

Scheme では，手続き呼び出しの引数の式は，手続きが評価結果を必要とするか否かにかかわらず，手続きが制御を得る前に評価される。C, C#, Common Lisp, Python, Ruby, そして Smalltalk は，常に手続きを呼び出す前に引数の式を評価する他の言語である。これは，その値が手続きで必要とされない限り，引数の式が評価されない Haskell の遅延評価の意味や， Algol 60 の名前呼びの意味とは異なる。

Scheme の算術演算モデルは，数値型と演算の豊富な集合を提供する。これはさらに，正確と不正確数を区別する: 基本的に，正確数オブジェクトは正しく数値に相当しており，不正確数は丸めや他の近似に関連する計算結果である。

1. 基本型

Scheme のプログラムは，値とも呼ばれているオブジェクトを，操作する。Scheme のオブジェクトは 型と呼ばれる値の集合でまとめられている。本章では，Scheme 言語の基本的に重要な型の概要を説明する。

注: Scheme は潜在的に型指定されているので， Scheme の文脈における用語 型 の使用は，他の言語，特に明示型のもの，の文脈における用語の使用とは異なる。

数 Scheme は，任意の精度の整数，有理数，複素数，および様々な種類の不正確数を含むバラエティ豊かな数値データ型をサポートする。

ブーリアン ブーリアンは，真偽値であり，真または偽のいずれかである。Scheme では， “偽” のオブジェクトは， `#f` と書かれる。“真” のオブジェクトは， `#t` と書かれる。しかし，真偽値が期待されているほとんどの場所では，`#f` とは異なる任意のオブジェクトは，真と数えられる。

ペアとリスト ペアは，二つの構成要素を有するデータ構造である。ペアの最も一般的な用途は，第一構成要素 (“car”) はリストの最初の要素を表しており，第二構成要素 (“cdr”) リストの残りであるような (片方向) リストを表すことである。Scheme は，リストを形成するペアのチェーンの最後の cdr である，区別された空のリストも持っている。

シンボル シンボルは，文字列を表すオブジェクトであり，シンボルの 名前 である。文字列とは異なり，名前が同じように綴られている二つのシンボルが区別されることはない。シンボルは，多くの用途に有用である; 例えば，それらは，列挙された値が他の言語で使われている方法を用いることができる。

R^{7}RS では， R^{5}RS とは異なり，シンボルと識別子は大文字と小文字が区別される。

文字 Scheme の文字は，ほとんどテキスト文字に対応している。より正確には，可能な実装依存の拡張でそれらは Unicode 標準のスカラ値の部分集合と同型である。

文字列 文字列は固定長の文字の有限列であり，したがって，任意の Unicode テキストを表す。

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列を表す線形データ構造である。リストの要素は，それを表すペアの連鎖を通じて順にアクセスされるのに対し，ベクタの要素は，整数の添字によってアドレス指定される。このように，ベクタは，リストよりも要素へのランダムアクセスに適している。

バイトベクタ バイトベクタは，その内容が 0 から 255 までの範囲の正確整数であるバイトということを除いて，ベクタに似ている。

手続き 手続きは，Scheme 内の値である。

レコード型 レコードは，構造化された値であり，各々が単一の場所を保持しているゼロ個以上のフィールドの集計である。レコードは，レコード型にまとめられている。述語，コンストラクタ，およびフィールドアクセサとミューテータは，各レコード型のために定義することができる。

ポート ポートは，入出力デバイスを表す。入力ポートは，コマンドに応じたデータを Scheme に配送することができる Scheme オブジェクトであると同時に，出力ポートは，データを受理することができる Scheme オブジェクトである。

2. 式

Scheme コードの最も重要な要素は，式である。式は評価され，値を生成することができる (実際には，任意個の値。) 最も基本的な式はリテラル式である:

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

この表記は，式 `#t` が `#t` ，すなわち，“真” の値に評価され，式 23 が数値 23 を表す数に評価されることを意味する。

複合式は，その部分式の前後に括弧を置くことによって形成される。最初の部分式は，演算子を識別する; 残りの部分式は，演算子のオペランドである:

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

これらの例の最初では， `+` は加算のための組み込み演算子の名前で，23 と 42 がオペランドである。式 `(+ 23 42)` は，“23 と 42 の和” として読み出される。複合式は入れ子にすることができる—二番目の例では，“14 と，23 と 42 の積との和” として読み出される。

これらの例が示すように， Scheme 中の複合式は，常に同じ接頭辞表記を使用して書かれている。結果として，括弧は，構造を示すために必要とされる。したがって，数学的表記や多くのプログラミング言語でも，多くの場合許容される “余分な” 括弧は， Scheme では許可されない。

(行末を含む) 空白は，式の部分式を分離し，構造を示すために使用することができる場合に，他の多くの言語同様，重要ではない。

3. 変数と束縛

Scheme は，識別子が値を含んでいる場所を表すことを可能にする。これらの識別子は，変数と呼ばれる。多くの場合，特に，場所の値が，作成後に変更されることがないとき，値を直接表している変数と考えると便利である。

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

この場合は， `let` で始まる式は束縛コンストラクトである。`let` に続く括弧で囲まれた構造は，変数の横に並んだ式を一覧にしている: 変数 `x` に 23 が並び，変数 `y` に 42 が並んでいる。`let` 式が `x` に 23 を， `y` に 42 を束縛する。これらの束縛は， `let` 式の本体 `(+ x y)` ，およびそこだけで利用可能である。

4. 定義

`let` 式によって束縛された変数は，その束縛が `let` の本体でのみ可視なので， 局所的 である。Scheme は，次のように識別子のトップレベルの束縛の作成を可能にする:

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

(これらは実際に，トップレベルのプログラムやライブラリの本体の “トップレベル” にある。)

最初の二つの括弧で囲まれた構造が定義である; それらはトップレベルの束縛を作成し，`x` に 23 を， `y` に 42 を束縛する。定義は式ではなく，式が発生することができるすべての場所に現れることはできない。さらに，定義は値をもたない。

束縛は，プログラムの字句構造に従う: 同じ名前の束縛が複数存在する場合，変数は，プログラムでの発生から始めて内側から外側に向かって，最も近い束縛を参照し，局所的な束縛が途中で見つからない場合は最も外側の束縛を参照する:

```
(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

定義は，手続きを定義するためにも使用することができる:

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

(f 23)                          ⟹ 65
```

やや単純化するが，手続きは，オブジェクトに対する式を抽象化したものである。この例では，最初の定義は， f と呼ばれる手続きを定義している。(これは手続きの定義であることを示す f x を括弧で囲むことに注意せよ。) 式 (f 23) は手続き呼び出しの意味で，大体， “(+ x 42) (手続き本体) を， 23 に束縛された x で評価する”。

手続きはオブジェクトなので，それらは他の手続きに渡すことができる:

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

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

(g f 23)                          ⟹ 65
```

この例では， g の本体は f で束縛された p と， 23 で束縛された x で評価され，それは (f 23) と等価であり， 65 と評価される。

実際には， Scheme のあらかじめ定義された多くの演算は，構文によってではなく，その値が手続きである変数によって提供されている。たとえば， + 演算は，他の多くの言語で特別な構文上の扱いを受けて受け取るが，Scheme においては数を加算する手続きに束縛されているただの通常の識別子である。同じことが， * や他の多くで成り立つ:

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

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

手続き定義は，手続きを作成するための唯一の方法ではない。lambda 式は，名を指定しなくても，オブジェクトとして新しい手続きを作成する:

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

この例の式全体が手続き呼び出しである; (lambda (x) (+ x 42)) は，一つの数をとり，それに 42 を加算する手続きに評価される。

6. 手続き呼び出しと構文キーワード

(+ 23 42)，(f 23)，および ((lambda (x) (+ x 42)) 23) がすべて手続き呼び出しの例であるのに対し，lambda および let 式はそうではない。これは， let は識別子であったとしても変数でなく，その代わりに 構文キーワードだからである。その最初の部分式として構文キーワードをもっているリストは，キーワードによって決定される特別な規則に従う。定義中の define 識別子もまた構文キーワードである。そのため，定義もまた手続き呼び出しではない。

lambda キーワードのルールは，最初の部分リストはパラメータのリストであり，残りの部分リストは手続きの本体であることを指定する。let 式では，最初の部分リストは束縛仕様のリストであり，残りの部分リストは式の本体を構成している。

手続き呼び出しは，リストの最初の位置で構文キーワードを探すことによって，これらの式型と区別することができる: 最初の位置に構文キーワードが含まれていない場合，式は手続き呼び出しである。Scheme の構文キーワードの集合は通常，この作業をかなり簡単にし，かなり小さい。しかしこれは，構文キーワードの新しい束縛を作成することが可能である。

7. 代入

定義または let または lambda 式によって束縛される Scheme の変数は，実際には直接それぞれの束縛に指定されたオブジェクトにではなく，これらのオブジェクトを格納している場所へ束縛される。これらの場所の内容は，続く代入を通じて破壊的に変更することができる:

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

この場合，let 式の本体は，最終的な式の値が let 式全体の値になるという，順次評価される二つの式で構成されている。式 (set! x 42) は代入であり， “x で参照される場所のオブジェクトを 42 に置換する” と言う。こうして， x の以前の値 23 は， 42 に置き換えられる。

8. 派生構文とマクロ

R^{7}RS の小さな言語の一部として指定された式の型の多くは，より基本的な式型に変換することができる。たとえば，let 式は，手続き呼び出しと lambda 式に変換することができる。次の二つの式は等価である:

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

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

let 式のような構文式は，その意味が，構文的変換による式の他の種類のものから派生することができるので，派生と呼ばれている。いくつかの手続きの定義もまた派生式である。次の二つの定義は等価である:

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

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

Scheme では，マクロに構文キーワードを束縛することによって自身の派生式を作成するためのプログラムが可能である。

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

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

define-syntax コンストラクトは，パターン（def f（p ...） body）に一致する括弧で囲まれた構造を指定し，それは式に変換される。ここで f, p, および body はパターン変数である。こうして，例で示す def 式は次に変換される:

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

新しい構文キーワードを作成する機能は，他の言語に組み込まれている機能の多くが Scheme で直接実装されることを可能にし， Scheme を非常に柔軟で表現力豊かにする:任意の Scheme プログラマは，新しい式型を追加することができる。

9. 構文上のデータとデータ値

データ値は Scheme オブジェクトのサブセットを構成する。これらには，ブーリアン，数字，文字，シンボル，文字列だけでなく，その要素がデータ値であるリスト，ベクタ，およびバイトベクタが含まれている。各データ値は，情報を失うことなく背後で読み書きができる構文的データとしてテキストで表すことができる。各データ値に対応する一つ以上の構文データが，一般的に存在する。さらに，各データ値は，対応する構文データの先頭に ’ を付加することで，プログラム内のリテラル式に自明に変換することができる:

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

前の例で示される ’ は，シンボルとリスト以外のリテラル定数の表現のために必要とされていない。構文データ foo は “foo” という名前をもつシンボルを表し， ’foo はその値としてのシンボルをもつリテラル式である。(1 2 3) は要素 1, 2, 3 をもつリストを表す構文データであり， ’(1 2 3) はその値としてこのリストをもつリテラル式である。同様に， #(1 2 3) は要素 1, 2, 3 をもつベクタを表す構文データであり， ’#(1 2 3) は対応するリテラルである。

構文データは， Scheme 式のスーパーセットである。したがって，データは，データオブジェクトとして Scheme 式を表すために使用することができる。特に，シンボルは識別子を表すために使用することができる。

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

これは， Scheme のソースコード上で演算するプログラム，特に，インタプリタやプログラム変換器を書くことを容易にする。

10. 継続

一つの Scheme 式が評価される時は常に，その式の結果を欲している一つの継続が存在する。継続は，計算に対する (デフォルトの) 未来全体を表現する。たとえば，非公式に式の 3 の継続

```
(+ 1 3)
```

は，それに 1 を加算する。通常，これらの遍在する継続は舞台裏に隠されており，プログラマはそれらについてあまり考えない。しかし，まれに，プログラマが継続を明示的に扱う必要がある。call-with-current-continuation 手続きは，Scheme プログラマが現在の継続を回復する手続きを作成することを可能にする。call-with-current-continuation 手続きは，脱出手続きを引数にした手続きを受けとり，すぐにそれを呼び出す。この脱出手続きはこのとき，call-with-current-continuation の呼び出しの結果になる引数で呼び出すことができる。つまり，エスケープ手続きは自身の継続を放棄し， call-with-current-continuation の呼び出しの継続を回復する。

次の例では，その引数に 1 を加算する，継続を表す脱出手続きは escape に束縛され，その後 3 を引数にして呼ばれる。escape の呼び出しの継続は放棄され，代わりに， 3 が 1 を加算する継続に渡される。

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

脱出手続きは，無制限の寿命を有する: それは取り込まれた継続が呼び出された後に呼び出すことができ，複数回呼び出すことができる。これは，call-with-current-continuation を，他の言語における例外のような一般的な非局所制御構文よりもはるかに強力にする。

11. ライブラリ

Scheme のコードは，ライブラリと呼ばれるコンポーネントにまとめることができる。各ライブラリには，定義および式が含まれている。それは他のライブラリとエクスポート定義から，他のライブラリに定義をインポートすることができる。

(hello) と呼ばれる次のライブラリは， hello-world と呼ばれる定義をエクスポートし，基本ライブラリと表示ライブラリをインポートする。hello-world エクスポートは，別の行に Hello World と表示する手続きである:

```
(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によって呼び出されるが，最終的には Scheme プログラムによって呼び出される。ライブ

ラリのように，プログラムはインポート，定義および式が含まれており，実行のためのエントリポイントを指定する。したがってプログラムは，それをインポートするライブラリの推移閉包を経由して Scheme プログラムを定義している。

次のプログラムは，プロセスコンテキストライブラリから `command-line` 手続きを経由してコマンドラインからの最初の引数を取得する。その後，ファイルが現在の入力ポートを発生させる `with-input-from-file` を使用してファイルを開き，それが最後に閉じられるために手配する。次に，ファイルからテキスト行を読み取るために `read-line` 手続きを，その後，行を出力するために `write-string` と `newline` を呼び出し，ファイルの最後までループしている:

```
(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

実装は，インポート宣言，式および定義が，一度に入力および評価できる *REPL* (read-eval-print ループ) と呼ばれる対話型セッションを提供してもよい。REPL は，基本ライブラリおよび，おそらく他のライブラリがインポートされて始まる。実装は， REPL が入力をファイルから読み取る操作モードを提供してもよい。このようなファイルは，先頭以外の場所にインポート宣言を含めることができるので，一般的には，プログラムと同じではない。

ここに短い REPL セッションがある。> 文字は入力のための REPL のプロンプトを表す。

```
> ; 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
```