IS^2 —Interpreter of Scheme on Scheme—

Yoshihiko Kakutani

はじめに

Scheme の処理系を Scheme で実装することを考えます。実装に使用するプログラミング言語が Scheme である必要はないのですが、Scheme にはリストデータと式を区別しないという特徴があるので、Scheme の処理系を実装するのに向いています。

もちろん、これから定義する処理系は厳密に Scheme の仕様に従うものではなく、サポートされない構文も多くあります。必要があれば、自分でコードを改良してよりよい処理系を作成してください。

本稿では、入門編の内容は既知のこととして扱われますので、基礎的な部分で疑問がある場合は入門編を参照してください。

1　環境モデル

Scheme の処理系を実装するには、処理系の挙動を知る必要があります。Scheme の式を評価する際、変数の現在の値が何かという情報が必要になるのは明らかです。この情報を環境（environment）と呼びます。Scheme の処理系は、常に環境を保持しており、プログラムの実行に合わせて環境の参照や更新を行います。環境は、変数と値の対応表のようなものと考えられます。例えば、次の式を評価するには、変数 x の値が必要になります。

```
(+ x 1)
```

変数 x が未定義のうちに上の式を実行するとエラーになります。環境は define 式を評価する度に更新されます。例えば、次の式を評価すると、x が 1 であるという情報が環境に追加されます。

```
(define x 1)
```

define 式以外で環境を変化させる可能性があるのは、let 式です。let 式は局所変数を定義する構文でした。ただし、局所変数は let 式の内部でしか有効ではないので、let 式を評価し終わった後の環境は以前と変化がないはずです。このような局所変数のスコープを実現するために、環境を階層的な構造で捉えることにします。先の式に続けて次のようなプログラムを考えます。

```
(define y (+ x 1))
(let ((x 4) (z (+ x 2)))
  (+ x y z))
```

define 式の評価が終わった時点で、環境はこうなっています。

x	↦	1
y	↦	2

let 式の内部 (+ x y z) を評価するには環境の変更が必要なのですが、let 式を評価し終わった時点でまたこの環境に戻らなければならないので、環境を直接変化させるのではなく、階層を 1 つ追加する方法を取ります。局所変数 z の値は、(+ x 2) を拡張前の環境で評価したものになります。

x	↦	1	x	↦	4
y	↦	2			
			z	↦	3

このように let 式の評価で追加される階層の 1 つ 1 つをフレーム（frame）と呼ぶことにします。フレームは let が入れ子になる度に追加され、その環境における変数の値は新しく追加されたフレームから順に探されます。上の環境では、x の値は 4、y の値は 2、z の値は 3、ということになります。フレームの階層のうち、古い側を外側、新しい側を内側と呼ぶことにします。（図で言えば、左が外側、右が内側です。）let 式の評価が終わると、追加したフレームが廃棄され、元の環境に戻ります。[*1]

フレームが追加されるのは、let 式の場合だけではありません。関数適用の評価時にもフレームの追加が起こります。次の例を見てみましょう。

```
(define square (lambda (x) (* x x)))
(square 3)
```

(square 3) が評価されると、square の値である λ 閉包から情報が取り出され、(* x x) の評価が行われます。このときの環境は以下のようになっています。ここで、⟨closure⟩ は λ 閉包を表しています。

x	↦	1	x	↦	3
y	↦	2			
square	↦	⟨closure⟩			

関数の定義に使われているパラメータが局所変数のように働いているのが分かると思います。実際、パラメータと局所変数には大きな違いはなく、let 式は常に lambda 式と関数適用を使った等価な式に書き換えることが可能です。上にあった例だと、次のようになります。

```
((lambda (x z) (+ x y z)) 4 (+ x 2))
```

関数適用式の最初の要素は lambda 式とは限らないので、任意の関数適用を等価な let 式で置き換えるということはできません。この意味で、関数適用は let よりも一般的だと言うことができます。

ここまでの説明で、処理系の環境に関する部分は全て説明が付きそうですが、まだ大きな問題が残されています。一つは再帰的に定義された関数の扱いで、もう一つは λ 閉包の持つ情報についてです。再帰のことは一先ず置いておいて、次の章では、λ 閉包と環境の関係について考察します。

[*1] 必ずしもフレームを使って環境の変化を実現する必要はないのですが、理論的にこのように理解することが可能だという意味です。

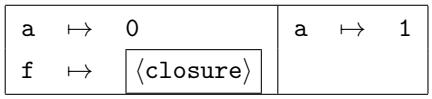
2　λ閉包

自由変数（パラメータでない変数のこと）を含まない関数を定義して使っている分には、これまでの解釈に紛れはなかったのですが、関数の定義に自由変数が現れると疑問が生じます。次のようなプログラムを考えることにします。

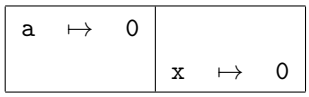
```
(define a 0)
(define (f x) (+ x a))
(let ((a 1)) (f 0))
```

この最後の let 式の評価結果は、0 になります。つまり、関数定義に現れる変数 a の値は、関数適用実行時には 0 になっているということです。言い換えれば、関数の挙動は関数が定義されたときに定まっているということになります。

環境の変化を追いながら、このプログラムの評価を考察してみましょう。let 式の内部の (f 0) を評価する際の環境は、以下のようになっているはずです。（前の章の変数定義は無視しています。）



この環境を拡張して (+ x a) を評価すると、a の値は 1 なので、評価結果は 1 ということになってしまいます。実は、Scheme では、(+ x a) が評価されるときの環境は次のようになっています。[*2]



このことは、自分が作られたときの環境の情報をλ閉包が含んでいるとすることで説明できます。λ閉包が作られたとき、つまり、変数 f が定義されたとき、a が 0 であるという環境はλ閉包の中に格納されます。関数適用による関数内部の評価時には、この環境がλ閉包から取り出されて、拡張されて用いられます。関数の定義に未定義の変数が含まれる場合や、関数が再帰的に定義される場合には、もう少し細かい説明が必要になるのですが、それについては後述とします。

ちなみに、ここで挙げた例と同等のプログラムが 1 を返すようなプログラミング言語を、動的スコープ（dynamic scope）を持つ言語と呼びます。Scheme のように 0 を返す言語は、静的スコープ（static scope）を持つ言語と呼ばれます。静的は構文的（lexical）と呼ばれることもあります。C では以下のようなコードになるので、C は静的スコープを持つということが分かります。

```
#include <stdio.h>
int a = 0;
int f(int x) { return a + x; }
int main() { int a = 1; printf("%d\n",f(0)); return 0; }
```

Emacs の中で動く Emacs-Lisp という言語は、Scheme と似た言語ですが、以下のコードが 1 を返すので、動的スコープを持ちます。（ただし、バージョン 24 以降の Emacs は静的スコープもサポートしているので、

[*2] 正確には、f がλ閉包であるという情報も外側のフレームに含まれます。

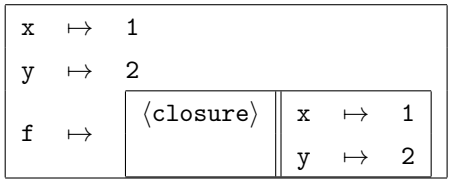
全てのプログラムが動的に振る舞うというわけではありません。)

```
(defvar a 0)
(defun f (x) (+ x a))
(let ((a 1)) (f 0))
```

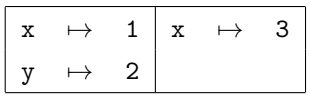
これまでの例では、let 式や関数適用の評価が終わると、局所変数用に拡張された部分のフレームはすぐに捨てられていましたが、返り値が関数の場合には、拡張されたフレームもλ閉包の中に残り続けることになります。次の例を考えてみましょう。

```
(define x 1)
(define y 2)
(define f (lambda (x) (lambda (y) (+ x y))))
((f (+ x y)) (* 2 y))
```

最初の 3 行を評価すると環境がこうなるのはよいと思います。

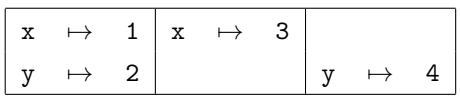


この環境で (f (+ x y)) の部分を評価します。すると、f の評価結果がλ閉包、(+ x y) の評価結果が 3 だと分かります。次に、関数適用が起こり、λ閉包の中から環境が取り出されて、パラメータのための拡張が行われます。



この環境で、(lambda (y) (+ x y)) が評価されます。ここで、また新しくλ閉包が作られるので、この環境がλ閉包に含まれることになります。この時点で、(f (+ x y)) の部分の評価が完了しています。

では、残りの評価に入ります。[*3](* 2 y) は外の環境で評価されるので、4 になります。その後、関数適用が起こるので、先程のλ閉包から環境が取り出されて使われます。(+ x y) がこの環境で評価されて、全体の評価結果は 7 ということになります。



このように、λ閉包が作られるときに環境が格納され、関数適用で関数内部の評価をするときに取り出されるという仕組みで、複雑な場合も説明することができます。作られたλ閉包を変数に割り当てておくと、仕組みがより明確になります。

```
(define g
  (let ((a 1) (b 2))
```

[*3] (f (+ x y)) と (* 2 y) のどちらが先に評価されるかは、仕様では決まっていません。

```
  (lambda (x) (* (+ x a) b))))
```

このように g を定義した場合、g の値はλ閉包になります。このλ閉包が持つ環境は、lambda 式が評価されたときの環境なので、次のようになります。

a ↦ 0	a ↦ 1 b ↦ 2

内側のフレームは、let 式を評価するに当たって拡張されたフレームです。let 式の評価が終わると、拡張されたフレームは捨てられ、元のトップレベルのみの環境に戻るのですが、拡張部分はλ閉包の中に残り続けます。続けて、次のような式を実行してみます。

```
(g a)
```

この式の評価時の環境は以下なので、a の値は 0 です。

a ↦ 0			
g ↦	⟨closure⟩	a ↦ 0	a ↦ 1 b ↦ 2

関数内部の実行に入ると、λ閉包に格納されていた環境が取り出されます。つまり、次の環境の下で (* (+ x a) b) が評価されることになり、最終的な評価結果は結局 2 になります。

a ↦ 0	a ↦ 1 b ↦ 2	x ↦ 0

ここまで理解できれば、処理系を自分で実装するまであと一歩ですが、少し寄り道をすることにします。処理系の実装を早く知りたい人は、次の章まで読み飛ばしてください。

上の例では、局所変数だった b には、g を通じてしかアクセスできなくなってしまっています。このような変数の秘匿の仕組みを利用すると、オブジェクト指向風のプログラミングが可能となります。ここでは、局所変数を含む環境で作られたλ閉包をオブジェクト、オブジェクトを生成する関数をクラスとみなします。オブジェクトの持つフィールドとメソッドは秘匿された局所変数で、λ閉包の含む環境でその値が定められます。オブジェクト指向プログラミングでフィールドの値を書き換える場合には、メソッド内で set! を使用します。

```
(define (make-account balance)
  (define (withdraw amount)
    (if (<= amount balance)
        (begin (set! balance (- balance amount))
               balance)
        (begin (display "Insufficient funds")
               (newline))))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
```

```
  (define (show)
    (display "Balance: ")
    (write balance)
    (newline))
  (define (unknown-method m)
    (display "Unknown request: ")
    (write m)
    (newline))
  (define (dispatch m)
    (cond ((equal? m 'withdraw) withdraw)
          ((equal? m 'deposit) deposit)
          ((equal? m 'show) show)
          (else (unknown-method m))))
  dispatch)
```

この例は、銀行口座のようなものを実装しています。make-account が定義された時点では、中に書かれている define 式はまだ評価されていないことに注意してください。make-account を使って関数適用を行った結果は関数となり、口座を意味するオブジェクトになります。メソッドの指定は、名前を引数として渡すことで実現されます。例えば、次のような使い方をします。

```
(define rich-acc (make-account 1000000))
(define poor-acc (make-account 1000))
((rich-acc 'withdraw) 500000)
((rich-acc 'deposit) 1000000)
((rich-acc 'show))
((poor-acc 'show))
```

rich-acc と poor-acc は、それぞれ固有の balance フィールドを持っていて、お互いに干渉することはありません。実際に実行してみれば、rich-acc をいくら操作しても、poor-acc に変化がないことが確認できます。

3　処理系

閑話休題、処理系の作成の続きです。Scheme には、eval という組込み関数があります。この関数は、Scheme の式と環境を引数として受け取って、その式の評価結果を返します。環境をデータとして扱っている点が異なりますが、eval は処理系の行なっている内部処理そのものを意味していると言うことができます。そこで、我々はこの eval に相当する関数 base-eval を定義することで、処理系を作成します。

混乱を避けるため、今後、これから作る処理系を単に「処理系」、元々ある本物の処理系を「メタ処理系」と呼ぶことにします。また、処理系の対象とメタ処理系の対象を区別する必要がある場合には、単語の頭にメタと付けることでメタ処理系の対象であることを表すことにします。

3.1 Read-Eval-Print ループ

base-eval を定義することが処理系を作成することの大部分を占めるのですが、先にそれ以外の部分について議論しておくことにします。以下の関数定義を見てください。

```
(define (scheme)
  (let ((top-env (make-top-env)))
    (define (rep-loop env)
      (display "sister> ")
      (let* ((res (base-eval env (read)))
             (env (car res))
             (val (cdr res)))
        (print-data val)
        (newline)
        (if (equal? val '*exit*)
            #t
            (rep-loop env))))
    (rep-loop top-env)))
```

base-eval と make-top-env が適切に定義されていれば、これが処理系の定義となります。つまり、以下の式をメタ処理系で実行すれば、処理系が動きます。

```
(scheme)
```

上の式が実行されると、最初に (make-top-env) が評価されます。make-top-env は引数を取らない関数で、大域変数のための初期環境を返すとします。read は標準入力から式を 1 つ読み込む関数です。(read) の評価結果は Scheme の式になるので、自分で構文解析する必要がありません。read を使うと、quote の略記は自動で解消され、コメントも読み飛ばされます。この (read) の評価結果は base-eval に渡されます。base-eval は環境と式を受け取って評価を実行する関数ですが、組み込みの eval と違って、環境と値のペアを返します。SICP では、組み込みの eval と同じく、値のみを返す関数を使っていますが、それは副作用を利用しているからです。副作用については、再帰的関数定義のところでもう少し説明します。print-data は、処理系が扱っているデータをユーザに表示するための関数です。内部データをそのまま表示すると読みにくいことがあるので、利用します。

まとめると、関数 scheme は、「標準入力から式を読み込み、評価し、その結果を表示する」という動作を繰り返すことになります。処理系とは正にそのようなものだったので、この関数は処理系の実装であるということができます。処理系が実行されている間、env という名前の変数によって、何らかの環境が常に使用されているのが分かると思います。

3.2 環境

ここまでで処理系の枠組みが定義できたので、より細かい部分の定義を見ていきます。まず、環境のデータ構造を定義しましょう。フレームを、変数と値の組に関する連想リストとして定義します。フレームを操作する関数は以下のように定義できます。

```
(define (empty-frame)
  (list))
(define (update frame var val)
  (cons (cons var val) frame))
(define (lookup var frame)
  (assoc var frame))
```

コードを見れば推測できると思いますが、empty-frame は空のフレームを返す関数、update はフレームと変数、値を受け取って、情報を加えた新しいフレームを返す関数、lookup は変数の値が何かフレーム内から見つける関数です。値が見つからない場合、lookup は #f を返します。フレームを連想リストで実装する理由は特にありません。二分探索木など他のデータ構造を採用しても、上の関数さえ適切に定義されていれば、以後の議論に変更はありません。

環境はフレームを階層的に積み上げたものだったので、フレームのリストとして定義することにします。環境を操作する以下の 4 つの関数を定義しておけば、処理系の実装には十分です。

```
(define (make-env)
  (list (empty-frame)))
(define (extend-env env)
  (cons (empty-frame) env))
(define (define-var env var val)
  (if (null? env)
      env
      (cons (update (car env) var val) (cdr env))))
(define (lookup-var var env)
  (if (null? env)
      #f
      (let ((found (lookup var (car env))))
        (if (pair? found)
            found
            (lookup-var var (cdr env))))))
```

ここでは、リストの先頭が内側で、トップレベルのフレームが最後尾となっています。make-env は、空のフレームを 1 つだけ含むような環境を返す関数です。extend-env は、環境を受け取ってそれを空のフレームで拡張したものを返します。define-var は、最内のフレームに変数と値の情報を追加する関数です。define-var の返り値は受け取った環境とは別の新しい環境になっています。lookup-var は、環境から変数

の割り当て情報を取り出す関数です。

3.3 λ閉包と組み込み関数

次に、処理系が内部で扱うデータの構造を決めなければなりません。定数データに関しては、処理系内部でもメタデータと同じものを使えばよいでしょう。問題はλ閉包です。処理系では、λ閉包を使って関数適用を実行する必要があるので、メタ処理系のλ閉包をそのまま流用するわけにはいきません。λ閉包が含むべき情報は、「パラメータ」と「関数本体のコード」、それから「環境」でした。単純に考えれば、それらをリストにしたものをλ閉包とみなすことができます。ただし、Scheme には通常のリストデータが別にあるので、処理系内部でデータとしてのリストとλ閉包を区別する必要があります。そこで、*lambda* という特別なシンボルから始まるリストをλ閉包と定義し、処理系内部のリストデータやペアデータの方は、そのままリストやペアとして扱うことにします。こうしてしまうと、*lambda* から始まるリストを意味するリストデータとλ閉包の区別が付かなくなりますが、その点は妥協することにします。[*4]いつも通り、データ構造を決めたら、それを操作する関数を用意します。

```
(define (make-closure env params body)
  (cons '*lambda* (cons env (cons params body))))
(define (data-closure? data)
  (and (pair? data) (equal? (car data) '*lambda*)))
(define closure-env cadr)
(define closure-params caddr)
(define closure-body cdddr)
```

make-closure がλ閉包を返す関数です。data-closure? でデータがλ閉包かどうかを判定します。closure-env、closure-params、closure-body は、それぞれ、λ閉包から環境、パラメータ、本体部分を取り出す関数です。本体部分は単一の式とは限らないので、cadddr ではなく cdddr になっています。（つまり、closure-body の返り値は、式ではなく、式のリストが想定されています。）

組み込み関数は Scheme 内で定義することはできないので、λ閉包とは別にデータ構造を用意する必要があります。[*5]関数 make-primitive で、組み込み関数の本体を意味するデータを作ります。data-primitive? は、data-closure? と同じく、データが組み込み関数かどうかを判定する述語です。primitive-arity で、組み込み関数データから引数の数を、primitive-fun で、組み込み関数データから必要な関数を取り出します。primitive-fun の返す値は、メタ処理系の作る本物のλ閉包です。具体的な使い方は、base-eval の定義のところで述べることにします。

```
(define (make-primitive arity fun)
  (list '*primitive* arity fun))
(define (data-primitive? data)
  (and (pair? data) (equal? (car data) '*primitive*)))
(define primitive-arity cadr)
```

[*4] リストデータにも先頭に特殊な識別子を入れることで、λ閉包と区別が可能になりますが、組み込み関数の定義が面倒になります。

[*5] 定義可能な組込み関数も存在します。そのような関数は組込み関数として扱わない方が便利です。

```
(define primitive-fun caddr)
```

また、データ構造に合わせて、print-data を定義しておきます。*lambda* の他にも特別な用途で使うシンボルがあるので、それに関する表示も定義されています。デバッグ時には、関数定義を適当に変更して、もう少し多くの情報を表示するようにしておくとよいでしょう。

```
(define (print-data data)
  (cond ((data-closure? data) (display "#<closure>"))
        ((data-primitive? data) (display "#<primitive>"))
        ((equal? data '*unspecified*) (display "#<unspecified>"))
        ((equal? data '*error*) (display "#<error>"))
        ((equal? data '*exit*))
        (else (write data))))
```

3.4 評価関数

後は、base-eval と make-top-env を定義すれば、処理系は完成します。最初に、base-eval の骨組みを考えてみましょう。base-eval は、環境と式を受け取って、環境とデータのペアを返す関数だということを思い出してください。[*6]base-eval の動作は、引数として受け取った式の形によって異なるので、以下のように書けるはずです。Scheme では長い式はリストデータでもあったので、引数として受け取った式に対して car を適用することで、式の先頭のシンボルを取り出せることに注意してください。

```
(define (base-eval env exp)
  (cond ((eof-object? exp) (cons env '*exit*))
        ((constant? exp) (cons env exp))
        ((symbol? exp) (var-eval env exp))
        ((not (pair? exp)) (eval-error env 'unknown-data exp))
        ((equal? (car exp) 'exit) (cons env '*exit*))
        ((equal? (car exp) 'define) (def-eval env exp))
        ((equal? (car exp) 'let) (let-eval env exp))
        ((equal? (car exp) 'letrec) (letrec-eval env exp))
        ((equal? (car exp) 'lambda) (lambda-eval env exp))
        ((equal? (car exp) 'if) (if-eval env exp))
        ((equal? (car exp) 'begin) (begin-eval env exp))
        ((equal? (car exp) 'quote) (quote-eval env exp))
        (else (app-eval env exp))))
```

ここで、constant? は次のように定義される述語です。つまり、定数データとして扱うべきかどうかを判別します。

[*6] SICP では、受け取る引数の順番が逆になっているので、注意してください。

```
(define (constant? exp)
  (or (boolean? exp) (number? exp) (string? exp)))
```

定数データは評価してもそのままなので、base-eval を適用すると、定義の 3 行目にあるように分岐して、(cons env exp) が評価されます。

その他の場合は、基本的に別の関数を呼び出すことになります。2 行目の EOF オブジェクトの場合と、5 行目のエラーの場合は本質的ではないので、無視してください。エラー処理は他の部分にも出てきますが、真面目にやると実装が大変なので、今回は、エラーが発生したことを表示するだけにしておきます。このコードでは、一度エラーが発生した後の処理系の挙動は保証されません。

```
(define (eval-error env type exp)
  (display "ERROR: ")
  (write type)
  (display ": ")
  (print-data exp)
  (newline)
  (cons env '*error*))
```

base-eval の定義の 6 行目は、exit 関数に関する処理です。メタ処理系では exit は組み込み関数ですが、ここでは、特殊フォームと同じように式の形に関する条件分岐で処理しています。[*7]本来なら、どのような文脈でも (exit) が評価されると処理系が終了するべきですが、エラー処理が難しいのと同じ理由で、この処理系は (exit) がトップレベルで評価される場合しか想定していないので、注意してください。

一見すると、base-eval の定義は再帰的ではないようですが、場合分け後に使われる eval 系関数のほとんどは内部で base-eval を呼び出すので、本質的には再帰を使っています。つまり、これらの定義は、一種の相互再帰になっていると言うことができます。

それでは、場合分けされた後の処理を更に詳しく見ていくことにします。実は、受け取る環境と返す環境が異なるのは、define 式を評価する def-eval だけです。let の場合も環境が変化しそうですが、let-eval では、内部のプログラムを評価する際に元の環境を拡張するだけで、返す環境は受け取った環境と同じです。ちなみに、let 式は関数適用と等価だったので、let-eval はその変形を行うだけでも十分です。

```
(define (let-eval env exp)
  (if (correct-syntax? 'let exp)
      (base-eval env (let->app exp))
      (eval-error env 'syntax-error exp)))
(define (let->app exp)
  (let ((decl (cadr exp))
        (body (cddr exp)))
    (cons (cons 'lambda (cons (map car decl) body))
          (map cadr decl))))
```

[*7] 今回の枠組みでも、exit を組み込み関数として処理することは可能なので、特殊フォーム扱いすることに深い意味はありません。

この let-eval の中身は本質的に (base-eval env (let->app exp)) で、前後の部分は構文チェックをしているに過ぎません。構文エラーは正しくない式を受け取ったときにしか起こらないので、正しい入力しか想定していない本稿の方針では、極端な話、correct-syntax? を以下のように定義してしまっても問題はありません。

```
(define (correct-syntax? type exp) #t)
```

全体のプログラムが有限であれば、ユーザは define 式の代わりに let 式を使ってプログラムを書けばよいのですが、利便性を考えて define 式への対応も実装しておきます。

```
(define (def-eval env exp)
  (if (correct-syntax? 'define exp)
      (let* ((var (cadr exp))
             (res (base-eval env (caddr exp)))
             (env (car res))
             (val (cdr res)))
        (cons (define-var env var val) var))
      (eval-error env 'syntax-error exp)))
```

let-eval の場合と同じく、最初の条件分岐は構文チェックです。def-eval は、define-var によって更新された環境を返します。define-var が評価されるより先に、変数に割り当てられる値が計算されていることに注意してください。例えば、メタ処理系で (define x (+ 1 2)) という式を実行した場合、(+ 1 2) が計算されてから、変数 x にその値が割り当てられるのと同様です。以下の式をメタ処理系で評価すると何が返ってくるのか、確認しておいてください。

```
(let ((env (make-env))
      (exp1 '(define x 1))
      (exp2 '(define y x)))
  (car (def-eval (car (def-eval env exp1)) exp2)))
```

ちなみに、この def-eval の実装は、関数定義のための略記をサポートしていないので、練習問題として、関数定義の略記が使えるように改良してみるとよいでしょう。

環境の操作で define-var の対となる lookup-var を使うのは、var-eval です。lookup-var で環境から変数の値を取り出して、それを返すだけです。簡単なので、この定義は練習問題としておきます。

次に、lambda-eval を考えます。lambda 式の評価結果はλ閉包なので、make-closure を使ってλ閉包データを作ります。その際、現在の環境がλ閉包に含まれることになります。

```
(define (lambda-eval env exp)
  (if (correct-syntax? 'lambda exp)
      (cons env (make-closure env (cadr exp) (cddr exp)))
      (eval-error env 'syntax-error exp)))
```

最も重要なのが、app-eval です。入門編で見たように Scheme は値呼びプログラミング言語なので、関数適用式の評価に当たって、先にリストの各要素を評価する必要があります。そのための補助関数 map-base-eval

を定義します。map-base-eval は式のリストと環境を受け取り、リストの各要素に base-eval を適用し、評価結果をリストにして環境と共に返す関数です。[*8]define 式を評価するときにしか環境は変化しないので、引数の中の評価されるような場所に define 式が現れないという仮定をおけば、map-base-eval は以下のように定義することができます。[*9]

```
(define (map-base-eval env el)
  (cons env
        (map (lambda (exp) (cdr (base-eval env exp))) el)))
```

define 式に対応した map-base-eval を定義するのは、練習問題としておきます。[*10]

app-eval は map-base-eval を使って、以下のように定義されます。

```
(define (app-eval env exp)
  (if (correct-syntax? 'app exp)
      (let* ((l (map-base-eval env exp))
             (env (car l))
             (fun (cadr l))
             (args (cddr l)))
        (base-apply env fun args))
      (eval-error env 'synatx-error exp)))
```

app-eval の中では、base-apply が呼ばれます。base-apply は、組み込みの apply と似た関数で、λ閉包か組み込み関数データとそれに与える引数、そして環境を受け取って、関数適用を実行する関数です。base-apply の定義は、以下のように、最初の引数がλ閉包か組み込み関数かで場合分けされます。

```
(define (base-apply env fun args)
  (cond ((data-closure? fun)
         ;; ここは自分で埋める
         )
        ((data-primitive? fun)
         (if (or (not (number? (primitive-arity fun)))
                 (= (primitive-arity fun) (length args)))
             ((primitive-fun fun) env args)
             (eval-error env 'wrong-number-of-args fun)))
        (else
         (eval-error env 'non-function fun))))
```

最初の引数がλ閉包の場合には、extend-env を使って環境を拡張し、その環境の下で本体部分を base-eval で評価します。環境モデルとλ閉包の章の説明をよく思い出してください。元になる環境は、関数適用の外から渡される環境ではなく、λ閉包に含まれているものを使います。この部分のコードを例示してしまうと自分

[*8] 仕様では、関数適用の引数部分をどの順に評価するかまでは決まっていません。
[*9] この演習では、できるだけ副作用のないプログラミングを目指しているので、この仮定に反するプログラムは登場しません。
[*10] 実は、後で紹介する方法で再帰をサポートすると、このままでも define 式に対応することになります。

で書く所がなくなってしまうので、詳しい実装は練習問題としておきます。面倒なら、関数の本体部分には 1 つの式しか書けないという制約を設けてもよいでしょう。また、base-apply の返り値は環境と評価結果のペアですが、これがどの環境なのかも注意してください。

組み込み関数は、base-apply の定義に合うように構成します。make-primitive の第 1 引数は、関数データが受け付ける引数の数でした。環境と関数データに渡す引数のリストを受け取って、環境と値のペアを返すメタλ閉包が、第 2 引数です。例えば、= に相当する組み込み関数データは、以下の式を評価すると得られます。パラメータ args は、引数を 1 つのリストにまとめたものを受け取ります。

```
(make-primitive 2 (lambda (env args) (cons env (= (car args) (cadr args)))))
```

make-top-env は、このような組み込み関数データを関数名に割り当てる関数です。通常の組み込み関数は環境を変化させないので、ほとんどが = と同じように書けます。(make-top-env) の評価結果は、空の環境に組み込み関数の情報を加えた環境になります。ここでは、+ のような可変個の引数を受け取る関数でも、簡単のために引数の数を固定しています。

```
(define (make-top-env)
  (let* ((env (make-env))
         (env
          (define-var env '=
            (make-primitive 2 (lambda (env args)
                                (cons env (= (car args) (cadr args)))))))
         (env
          (define-var env '+
            (make-primitive 2 (lambda (env args)
                                (cons env (+ (car args) (cadr args)))))))
         (env
          (define-var env '*
            (make-primitive 2 (lambda (env args)
                                (cons env (* (car args) (cadr args)))))))
         (env
          (define-var env 'list
            (make-primitive #f (lambda (env args) (cons env args)))))
         (env
          (define-var env 'null?
            (make-primitive 1 (lambda (env args)
                                (cons env (null? (car args)))))))
         (env
          (define-var env 'equal?
            (make-primitive 2 (lambda (env args)
                                (cons env (equal? (car args) (cadr args)))))))
         (env
```

```
        (define-var env 'display
          (make-primitive
           1
           (lambda (env args)
             (display (car args))
             (cons env '*unspecified*)))))
       (env
        (define-var env 'load ; load に関しては理解できなくもよい
          (make-primitive
           1
           (lambda (env args)
             (with-input-from-file (car args)
               (lambda ()
                 (define (re-loop env)
                   (let* ((res (base-eval env (read)))
                          (env (car res))
                          (val (cdr res)))
                     (if (equal? val '*exit*)
                         (cons env '*unspecified*)
                         (re-loop env))))
                 (re-loop env)))))))
  env))
```

もちろん、上のコードは Scheme の組み込み関数全てには対応していないので、必要な組み込み関数があれば、例に倣って自分で付け加えてください。処理系の動作テストをする際に便利なので、load も定義してありますが、その部分は必ずしも理解する必要はありません。load は環境を変化させるための関数なので、他の組み込み関数の定義とは形が異なっています。

また、別の関数から定義可能な組み込み関数は、普通の Scheme のコードとして関数定義を準備しておき、後で処理系に読み込ませるのがよいでしょう。関数を make-primitive で準備するには、自分で書いている処理系の処理を考えながらコードを書かなければいけませんが、後で処理系に読み込ませるスタイルなら、メタ処理系で実行するのと同じ普通の Scheme のコードを書くだけなので、間違いが起こりにくくなり、処理系内部のデータ構造を変更した場合にも簡単に対応できます。例えば、not はメタ処理系では組み込み関数ですが、以下のように普通の関数として定義することができます。

```
(define (not b) (if b #f #t))
```

特に、この手法は、map のような内部で関数適用を行う高階関数に有効です。普通の Scheme のコードでは map は以下のように定義することができますが、これを make-primitive によって組み込み関数データとして実装するには、内部で base-apply を使うなどの工夫が必要になります。

```
(define (map f l)
  (if (null? l)
```

```
      l
      (cons (f (car l)) (map f (cdr l)))))
```

base-eval が完成しているなら、処理系に入力する代わりに、make-top-env 内に次のように書いてもかまいません。

```
(define (make-top-env)
  (let* ((env (make-env))
         ...
         (env
          (car
           (base-eval env '
                      (define not (lambda (b) (if b #f #t))))))
         (env
          (car
           (base-eval env '
                      (define map
                        (lambda (f l)
                          (if (null? l) l (cons (f (car l)) (map f (cdr l)))))))))
         ...)
    env))
```

もっとよいのは、「preamble.scm」などというファイルを用意しておいて、(load "preamble.scm") を処理系に実行させる方式でしょう。例えば、処理系の定義を以下のようにしておくと、処理系が起動するときに preamble.scm が読み込まれるようになります。

```
(define (scheme)
  (let ((top-env (make-top-env)))
    (define (rep-loop env)
      (display "sister> ")
      (let* ((res (base-eval env (read)))
             (env (car res))
             (val (cdr res)))
        (print-data val)
        (newline)
        (if (equal? val '*exit*)
            #t
            (rep-loop env))))
    (let ((top-env (car (base-eval top-env '(load "preamble.scm")))))
      (rep-loop top-env))))
```

残りの if-eval と quote-eval の実装は特に工夫を必要としないので、練習問題とします。

ここまで書けば、ある程度動く Scheme 処理系が完成しているはずです。関数の再帰的定義が使えませんが、それについては次の章で説明します。とりあえず、ここまでの動作を確認して、処理系が動くことを実感しておきましょう。

4 再帰と環境

これまで作ってきた処理系では、再帰的な関数定義や中に未定義の変数が現れる関数定義は許されていませんでした。このような関数定義に対応するには、環境の操作に副作用を取り入れる必要があります。

4.1 リストの破壊的操作

副作用のある関数の使い方を知るためには、リストについての理解を深めなければなりません。Scheme では、新しい変数が定義されるときや、関数適用でパラメータに値が割り当てられるときには、データそのものがコピーされていると考えて問題ありませんでした。しかしながら、実際には、リストデータ（ペアデータ）はポインタによって実現されています。次のプログラムを見てください。もちろん、メタ処理系で実行します。

```
(define a (list 1 2))
(define b (list 1 2))
(define c a)
(equal? a b)
(eq? a b)
(eq? a c)
```

結果は予想通りだったでしょうか？ このことは、リストデータはポインタとしてしか扱えないと解釈すれば、理解することができます。cons や list でリストデータを作った場合、そのデータそのものはメモリ上に 1 つ作られ、それを割り当てられた変数などはそのポインタを値として持つことになります。関数適用でリストデータが渡される場合なども、データそのものではなくポインタが渡されます。したがって、上の例で変数 c が定義されたとき、c の値もまたポインタとなり、a の値が指すデータと同じものを指すことになります。今はリストデータについて説明しましたが、ペアデータの扱いも全く同様です。

ペアやリストがポインタとして扱われるのは、何も変数への受け渡しのときだけではありません。ペアが入れ子になっている場合、データの内部でもポインタが使われています。外側のペアが含む情報は、内側のペアへのポインタになります。これは、C の構造体で二分木を扱う場合とよく似ています。

Scheme には C のようにポインタを明示的に扱う手段がないので、eq? のような特殊な関数を使わない限り、ペアデータを直接扱っていても、そのポインタを扱っていても、プログラマにとって違いはありません。しかしながら、ペアがポインタで実装されていることを利用して、ペアを操作する関数も存在します。それが、set-car! と set-cdr! です。set-car! は、ペアと何らかのデータを受け取り、第 2 引数のデータをペアの第 1 要素に割り当てます。返り値は不定です。先に述べたように、第 1 引数は実際にはペアそのものではなく、ペアへのポインタなので、C におけるポインタを受け取る関数のように、ポインタの指す先のメモリを書き換えることになります。

```
(define d (cons 0 c))
```

```
(set-car! a 3)
a
d
```

このプログラムを実行してみると、set-car! によって、変数 a の値が指すデータの内容が書き換わっているのが確認できたはずです。また、 d の値が指すリストの内容も変更されています。これは、ペアデータの内部でもポインタが使われている証拠です。

set-cdr! は set-car! と同じような関数ですが、ペアの第 2 要素を書き換えます。ちなみに、set-cdr! を使うと、循環リスト（これはリストではありません）を作ることも可能です。次の式を評価した後の b の値が指すペアデータは、第 2 要素を無限に辿ることができます。

```
(set-cdr! (cdr b) b)
(car (cdr (cdr (cdr (cdr (cdr (cdr b)))))))
```

4.2 副作用による環境操作

set-car! と set-cdr! を使って、環境の操作を実装することで再帰的な関数定義が可能になります。環境に情報を付け加える関数 define-var を、以下の define-var! に置き換えてみましょう。

```
(define (define-var! env var val)
  (if (null? env)
      #f
      (set-car! env (update (car env) var val)))
  env)
```

define-var! を使うことで、何が変わるのでしょうか。define-var! は引数として受け取った環境をそのまま返しています。環境はリストだったので、実質的にはポインタが渡されて、そのポインタをそのまま返していることになります。ちなみに、define-var の返すポインタは、define-var! の場合と違って、引数として与えられたものとは異なっています。define-var! の処理系内での使われ方を考えると、define-var! の関数適用式が評価される前と後で、処理系内の環境を意味する変数の値は同じメモリを指し続けていることになります。ただし、途中で set-car! が呼ばれるので、ポインタの指すリストデータの内容は書き換わっています。

set-car! を使った式を評価すると、それよりも前に定義された変数の値が指すデータを書き換えることができます。処理系の実装で重要なのは、λ閉包の含む環境です。λ閉包も環境もリストなので、実際にはポインタが使われています。λ閉包は作られたときの環境を含んでいるのですが、set-car! を使うことで、後からλ閉包の中の環境を書き換えることができます。次の式を順に処理系で実行することを考えてみましょう。

```
(define former (lambda (x) latter))
(define latter 10)
(former 0)
```

変数 former が定義されると、その値はλ閉包になります。λ閉包は環境を含んでいますが、この時点では、その環境に変数 latter の情報はありません。次に latter がトップレベル環境に定義されると、λ閉包の中の環境にもその情報が反映されます。この挙動を確認するには、関数 scheme の定義中にある (print-data val) という 1 文を以下の式に書き換えてから (scheme) を実行し、処理系に上の例を入力してみるとよいでしょう。

```
(let ((info (lookup-var 'former env)))
  (if (pair? info)
      (print-data (lookup-var 'latter (closure-env (cdr info))))
      (display "undefined")))
```

これで、関数定義の段階で未定義の変数が含まれている場合にも対応できるようになりましたが、実は、同じ原理で既に再帰も実現できています。本当にこれで再帰が可能になっているのかは自明ではありませんが、実際に処理系で再帰的な関数を定義してみることである程度確認することができます。

最後に、処理系内での letrec 式の評価方法について触れておきます。letrec と let の違いは再帰を許すかどうかでしたが、これは、局所変数に割り当てる値を求めるための評価と環境の拡張の順番の違いだと捉えることができます。次の式を処理系（またはメタ処理系）で実行するとエラーになります。トップレベルに変数 fact は定義されていないと仮定してください。

```
(let ((fact (lambda (n)
              (if (< n 1)
                  1
                  (* n (fact (- n 1)))))))
  (fact 10))
```

これは、局所変数 fact に割り当てられているλ閉包の含む環境が fact の情報を持っていないことが原因です。λ閉包が作られるときの環境は、拡張される前のトップレベル環境です。変数 fact の情報は、その後に拡張されたフレームに含まれるので、λ閉包には反映されません。自分で作った処理系の場合、λ閉包の環境に後から情報を加えることができると述べましたが、λ閉包の外で用意された新たなフレームの情報まで反映されることはありません。extend-env が、引数として受け取った元の環境を書き換えることなく、新たなフレームへのポインタを要素に持つ新たなペアデータを作っているからです。

これに対して、let を letrec に書き換えた以下の式がメタ処理系でうまく処理できるのは、先に環境を拡張してから、その環境を使ってメタλ閉包を作っていることによります。つまり、メタλ閉包の含むメタ環境が (fact 10) を評価するときと同じメタ環境になっているということになります。自分の処理系でも、この順序で環境の拡張と lambda 式の評価を行えば、(fact 10) を処理するときに使われる環境データへのポインタが、λ閉包に含まれるポインタと一致するようになります。

```
(letrec ((fact (lambda (n)
                 (if (< n 1)
                     1
                     (* n (fact (- n 1)))))))
  (fact 10))
```

ところで、この評価処理は以下の式に対する評価処理とよく似ています。これまでの規則に則って以下の式を評価すると、まず let 式による環境の拡張が行われますが、新しくできたフレームに局所変数は定義されていません。次に define 式が評価され、最内のフレームに fact という変数が定義されます。このとき、fact の値はλ閉包になっていますが、そのλ閉包は現在の環境を含んでいるため、再帰的な関数定義が有効になっています。最後に同じ環境で、(fact 10) が評価されますが、λ閉包から取り出された環境には fact の情報があるので、問題なく評価することができます。

```
(let ()
  (define fact (lambda (n)
                 (if (< n 1)
                     1
                     (* n (fact (- n 1))))))
  (fact 10))
```

したがって、letrec-eval を普通に実装する代わりに、受け取った式を上の例に倣って let と define を使った式に変形して base-eval に渡すというコードを書いてもよいということになります。[*11]ただし、その場合、let 式の本体部分に複数の式を書けるようになっている必要があるので、注意してください。

以上で、再帰をサポートする処理系が完成したことになります。これまで処理系を書くのに利用した構文や組み込み関数を全てサポートするように処理系を改良すれば、処理系の中で処理系を実行することも可能になります。時間があれば、ぜひ挑戦してみてください。

[*11] R5RS には、むしろ、このような局所的な define 式があった場合には、letrec 式に変換されるとあります。