工学部専門科目
「プログラミング言語」SICP
第4章 ～超言語的抽象～
その1

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科
通信情報システム専攻

June 12, 2013

今日のメニュー

4.1 The Metacircular Evaluator
4.1.1 The Core of the Evaluator
4.1.2 Representing Expressions
4.1.3 Evaluator Data Structures
4.1.4 Running the Evaluator as a Program

4.1.5 Data as Programs
4.1.6 Internal Definitions
4.1.7 Separating Syntactic Analysis from Execution

Metalinguistic Abstraction

問題領域に応じた新しい言語を作って抽象化

- **どんなプログラミング言語も「万能」ではない**
- **問題領域に応じた抽象化機構が必要**

本章の内容

core Scheme (**とその変種**) **の** evaluator (**評価器**) **を作る**

> *"The evaluator, which determines the meaning of expressions in a programming language, is just another program."*

$\Rightarrow$ Programs as Data!

4.1 The Metacircular Evaluator

- metacircular: **定義する・される言語が等しい**
- **基本アルゴリズムは** 3.2 **節の式の評価モデル**
- **核となる (相互再帰) 関数**: `eval` **と** `apply`
- **式のデータ表現** (4.1.2 **節**)
  - **式の種類を調べる関数**: `assignment?`, `if?`, . . .
  - **部分式を取り出す関数**: `assignment-variable`, `assignment-value`, `if-predicate`, `if-consequent`, . . .
  - **式を作る関数**: `make-procedure`, `make-if`
    - ★ **背景黄色で書くので，具体的な定義はひとまず気にしないでよい**

eval と apply の定義

- eval
  - 引数: 評価対象の式 exp と環境 (変数とその値の情報) env
  - 返値: exp を env の下で評価した結果 (の値)
  - 式の形で場合分け、関数呼出式の場合は関数部・引数部を評価して apply を呼び出す

    ⇒ 式の種類が増えると eval の再定義が必要 (cf. Ex.4.3)
- apply
  - 引数: 手続きとその引数 (の値)
  - 返値: 手続きを引数に適用した結果 (の値)
  - 仮引数と実引数の関連情報を環境に追加して関数本体を評価する (eval を呼び出す)

```
(define (eval exp env)
  (cond  ;; exp の種類で場合分け
   ((self-evaluating? exp) ...)
   ((variable? exp) ...)
   ((quoted? exp) ...)
   ((assignment? exp) ...)
   ((definition? exp) ...)
   ((if? exp) ...)
   ((lambda? exp) ...)
   ((begin? exp) ...)
   ((cond? exp) ...)
   ((application? exp) ...)
   (else (error ...))))
```

```
(define (eval exp env)
  (cond ...
   ((application? exp) ;; 関数適用の形ならば
    ;; 関数部(operator)と引数(operands)の
    ;; 評価結果を apply に渡す
    (apply (eval (operator exp) env)
           (list-of-values (operands exp) env))
   ...))
(define (list-of-values exps env)
  (if (no-operands? exps) '()
      (cons (eval (first-operand exps) env)
            (list-of-values
              (rest-operands exps) env))))
```

関数適用式の評価(3.2節より)

環境 A のもとでの関数適用式（e_0 e_1 . . . e_n）の値:

- A のもとでの e_i の値 を v_i とする
- 関数閉包 v_0 の中身の環境を B，パラメータを $x_1, \ldots, x_n$ とする
  - v_0 が関数閉包でなかったり，パラメータの数が n でない場合はエラー
- 束縛 $x_i : v_i$ から成る新しいフレームを作り，B を指すようにする．この環境を C とする
- C のもとで関数閉包の本体式 e の値を求めたものが，関数適用式全体の値

```
(define (apply procedure arguments)
  (cond  ;; procedure の種類で場合分け
    ((primitive-procedure? procedure)
     ;; プリミティブ手続き
     ...)
    ((compound-procedure? procedure)
     ;; ユーザ定義手続き
     ...)
    (else
     (error ...))))
```

```
(define (apply procedure arguments)
  (cond ...
    ((compound-procedure? procedure)
     ;; ユーザ定義手続き
     (eval-sequence ;; 関数本体の式を評価
       (procedure-body procedure)
       ;; 仮・実引数情報で拡張した環境下で
       (extend-environment
         (procedure-parameters procedure)
         arguments
         (procedure-environment procedure))))
    (else
     (error ...))))
```

```
(define (eval-sequence exps env)
  ;; 式のリストを前から評価して最後の式の値を返す
  (cond ((last-exp? exps)
         (eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence
                (rest-exps exps) env))))
```

4.1.2 Representing Expressions

定義される言語の構文 (BNF **定義**)

$$
\begin{array}{lcl}
\langle \textbf{式} \rangle & ::= & \langle \textbf{定数} \rangle \mid \langle \textbf{変数} \rangle \mid \langle \textbf{引用} \rangle \\
& \mid & \langle \textbf{代入} \rangle \mid \langle \textbf{定義} \rangle \mid \langle \textbf{ラムダ} \rangle \\
& \mid & \langle \textbf{条件式} \rangle \mid \langle \textbf{逐次実行} \rangle \\
& \mid & \langle \textbf{関数適用} \rangle \mid \langle \text{cond } \textbf{式} \rangle \\
\langle \textbf{定数} \rangle & ::= & \langle \textbf{数値定数} \rangle \mid \langle \textbf{文字列定数} \rangle \\
\langle \textbf{変数} \rangle & ::= & \langle \textbf{シンボル} \rangle \\
\langle \textbf{引用} \rangle & ::= & \texttt{(quote } \langle \textbf{式} \rangle \texttt{)} \\
\langle \textbf{代入} \rangle & ::= & \texttt{(set! } \langle \textbf{変数} \rangle \ \langle \textbf{式} \rangle \texttt{)} \\
\langle \textbf{定義} \rangle & ::= & \texttt{(define } \langle \textbf{変数} \rangle \ \langle \textbf{式} \rangle \texttt{)} \\
& \mid & \texttt{(define (} \langle \textbf{変数} \rangle \ \langle \textbf{変数} \rangle \cdots \langle \textbf{変数} \rangle \texttt{)} \\
& & \quad \langle \textbf{式} \rangle \cdots \langle \textbf{式} \rangle \texttt{)}
\end{array}
$$

$$
\begin{array}{lcl}
\langle \text{ラムダ} \rangle & ::= & \texttt{(lambda (}\langle \text{変数} \rangle \cdots \langle \text{変数} \rangle\texttt{)} \\
 & & \quad \langle \text{式} \rangle \cdots \langle \text{式} \rangle\texttt{)} \\
\langle \text{条件式} \rangle & ::= & \texttt{(if } \langle \text{式} \rangle\ \langle \text{式} \rangle\ \langle \text{式} \rangle\texttt{)} \\
\langle \text{逐次実行} \rangle & ::= & \texttt{(begin } \langle \text{式} \rangle \cdots \langle \text{式} \rangle\texttt{)} \\
\langle \text{関数適用} \rangle & ::= & \texttt{(}\langle \text{式} \rangle\ \langle \text{式} \rangle \cdots \langle \text{式} \rangle\texttt{)} \\
\langle \text{cond 式} \rangle & ::= & \texttt{(cond } \langle \text{節} \rangle \cdots \langle \text{節} \rangle\texttt{)} \\
\langle \text{節} \rangle & ::= & \texttt{(}\langle \text{条件} \rangle\ \langle \text{式} \rangle \cdots \langle \text{式} \rangle\texttt{)} \\
 & | & \texttt{(else } \langle \text{式} \rangle \cdots \langle \text{式} \rangle\texttt{)} \\
\langle \text{条件} \rangle & ::= & \langle \text{式} \rangle
\end{array}
$$

(cond で else 節は最後に来なければならない，という条件は無視している．)

式の構造を表現するための関数

- 式を作る関数: `make-procedure`, `make-if`
- 式の種類を調べる関数: `assignment?`, `if?`, ...
- 部分式を取り出す関数: `assignment-variable`, `assignment-value`, `if-predicate`, `if-consequent`, `if-alternative`, ...
  - 定義 (define) のための関数は読み替えを行う

(define (⟨変数⟩ ⟨変数⟩ ··· ⟨変数⟩)
        ⟨式⟩ ··· ⟨式⟩)

⇓ 読み替え

(define ⟨変数⟩
  (lambda (⟨変数⟩ ··· ⟨変数⟩) ⟨式⟩ ··· ⟨式⟩))

Derived expressions

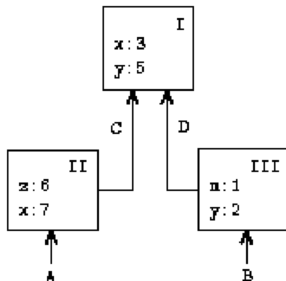
より単純な機能の組み合わせで表された複合的機能を持つ式:

- **より単純な機能の組み合わせに変換し** `eval` **に渡す**
- **つまり** evaluator **によるマクロ展開**
- **先程の** `define` **の読み替えも変換の一種**
- `cond->if` **関数**: `cond` **式を** `if` **の繰り返しに変換**

```
(define (eval exp env)
  (cond
   ...
   ((cond? exp) (eval (cond->if exp) env))
   ...))
```

4.1.3 評価器が内部で使うデータ構造

- 真偽値: シンボル `false` が偽，それ以外は真
- 手続き値:
  - `(procedure` 〈仮引数リスト〉〈本体〉〈環境〉`)`
  - プリミティブ関数は後述
- 環境: 変数とその値の関係を保持したフレームの列



環境操作のインターフェース

- `the-empty-environment`
  ··· 空の環境 (フレーム|の指す「親」環境)
- `(lookup-variable-value` ⟨変数⟩ ⟨環境⟩`)`
  ···⟨環境⟩から⟨変数⟩の値を取ってくる
  - `(lookup-variable-value 'y A)` $\Rightarrow$ 5
- `(extend-environment` ⟨変数列⟩⟨値列⟩⟨環境⟩`)`
  $\Rightarrow$ ⟨環境⟩に⟨変数列⟩と⟨値列⟩からなる新しいフレームを追加した新しい環境を返す
  - `(extend-environment '(z y) '(2 4) A)` $\Rightarrow$

- (define-variable! ⟨変数⟩ ⟨値⟩ ⟨環境⟩)
  …⟨環境⟩中の最初のフレームに変数束縛を追加
  - (define-variable! 'y 4 A) ⇒

- (set-variable-value! ⟨変数⟩ ⟨値⟩ ⟨環境⟩)
  …⟨環境⟩中の⟨変数⟩の値を⟨値⟩に更新
  - (set-variable-value! 'y 8 A) ⇒

環境のデータ構造

ここでの(単純だがあまり効率のよくない)実装方法:

- 環境 = フレームのリスト
- フレーム = (同じ長さの)変数リストと値リストのペア

```
(define (make-frame variables values)
  (cons variables values))
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      ... ;; エラー処理
      ))
```

4.1.4 評価器を走らせる

- **初期環境の定義**
- **プリミティブ手続きの実現**
- Read-Eval-Print Loop (REPL)

初期環境の定義

プリミティブ手続き・真偽値を空の環境に追加

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment
           (primitive-procedure-names)
           (primitive-procedure-objects)
          the-empty-environment)))
    (define-variable! 'true true initial-env)
    (define-variable! 'false false initial-env)
    initial-env))
```

プリミティブ手続きの実現(1/2)

手続き値の表現:

- `(procedure` ⟨**仮引数リスト**⟩ ⟨**本体**⟩ ⟨**環境**⟩`)`

プリミティブ手続きの実現(1/2)

手続き値の表現:

- (procedure 〈**仮引数リスト**〉〈**本体**〉〈**環境**〉)
- (primitive 〈Scheme **プリミティブ**〉)
  - initial-env **は** primitive-procedures (**名前とそれが指す手続きのペアのリスト**) **から作る**

```
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        ...
        ))
```

プリミティブ手続きの実現(2/2)

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    arguments))
        ...))
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme
   (primitive-implementation proc) args))
```

プリミティブ手続きの実現(2/2)

```
(define apply-in-underlying-scheme apply)
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    arguments))
        ...))
(define (apply-primitive-procedure proc args)
  (apply-in-underlying-scheme ;; 元々の apply!
   (primitive-implementation proc) args))
```

定義の順番に注意!!

プリミティブ手続きの実現(3/2)

apply の再定義を避ける別のやり方:

```
(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure
                                    arguments))
        ...))
(define (apply-primitive-procedure proc args)
  (apply ;; 元々の apply!
   (primitive-implementation proc) args))
```

Read-Eval-Print Loop

関数 `driver-loop`:

1. 入力プロンプトを出力
2. `read` でキーボードから式を読み込む
3. `eval` で大域環境のもとで評価
   - 環境は評価中に書き換わる可能性あり
4. 結果を出力
5. 最初に戻る

宿題： 6/19 午前8時 締切

- Ex. 4.4, Ex. 4.11
  - 講義の web page に教科書のコードあり
- レポートには
  - 考え方の説明
  - プログラムリストと考え方の対応
  - 実行例

  を示すこと
- レポート (pdf) とプログラムファイルを提出システムを通じて提出
- 友達に教えてもらったら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)