Racket

- http://mitpress.mit.edu/sicp/
- http://racket-lang.org/
- http://docs.racket-lang.org/

なぜ List

- Lisp は主流の言語ではないが
  - プログラムの構成やデータの構造を学び、言語の基礎となる言語学的昨日に関係づけるのに優れた媒体としての特徴を持つ
  - 主な特徴として、プロセスの手続き (procedures) という List による記述自体が Lisp データとして表現、処理できる
  - 「受動的」なデータと「能動的」なプロセスという区別をしないプログラム設計技法が使える
  - 手続きをデータとして表す能力は、実在する言語で一番便利

インストール

■ Ubuntu にインストール

```
$ apt-cache search racket
$ apt-cache show plt-scheme
$ apt-cache policy plt-scheme
plt-scheme:
  インストールされているバージョン：( なし )
  候補：                5.2.1+g6~92c8784+dfsg2-2+deb7u1
  バージョンテーブル：
     5.2.1+g6~92c8784+dfsg2-2+deb7u1 0
        500 http://jp.archive.ubuntu.com/ubuntu/ quantal/universe i386 Packages
$ sudo apt-get install plt-scheme
```

■ CentOS にインストール

- Racket を CentOS にインストール

プログラムの要素

- 言語はプロセスに関する考えをまとめる枠組み
- 単純な概念を統合して複雑な概念を構成するための手段を用意
- 強力な言語には 3 つの仕掛けがある

プログラムの要素	内容
基本式	言語が関わるもっとも単純なもの
組合せ法	より単純なものから合成物をつくる
抽象化法	合成物に名前をつけ、単一なものとして扱う

式 (expression)

- 式 (expression) を入力すると解釈系は応答してその式を評価した (evaluating) 結果を表示する

整数を与えると、応答を印字

```
> 486
486
```

手続き (+ や * など) と組み合わせて合成式とし、手続きの作用を表現

```
> (+ 137 349)
486
> (* 5 99)
495
```

■組合せ、演算子、被演算子、引数

・式の並びを括弧で囲んで手続きの作用を表現する式を組合せ (combinations) という
・左端の要素を演算子 (operator)、他の要素を被演算子 (operands) という
・組合せの値は、演算子が指定する手続きを、被演算子の値である引数 (arguments) に作用させて得る

■前置記法

・演算子を被演算子の左に置く書き方を前置記法 (prefix noations) という

利点

・任意の引数をとる手続きを許す

```
> (+ 21 35 12 7)
75
```

・組合せを入れ子にする (nested) ことを許す

```
> (+ (* 3 5) (- 10 6))
19
```

名前と環境

・名前を使って計算オブジェクトを指す手段を用意することが、プログラム言語の重要な点
・オブジェクトを値 (value) とする変数 (variable) を識別するのものが名前
・define で名前付け

```
> (define size 2)
> size
2
```

・値と記号を対応付け、後にそれを取り出せるようにするために記憶しておくことを環境 (environment) より正確には、大域環境 (global environment) という

組合せと評価

1. 組合せの部分式を評価する
2. 最左部分式の値である手続き (演算子) を、残りの部分式の値である引数 (被演算子) に作用させる

・評価の規則は本質的に再帰的 (recursive) である

・一般に再帰は階層的な木構造のオブジェクトを扱うのに強力な技法である

■特殊形式 (special forms)

・例えば、define は、2 つの値に作用させるのではなく、対応づけるだけのため、(define x 3) は組合せではない
・このような一般評価規則の例外を特殊形式 (special forms) という

・http://typea.info/blg/glob/2010/03/_3.html

合成手続き

■手続き定義 (procedure definetions)

・手続き定義 (procedure definetions) により、合成演算に名前を対応付け、一体として指すことができる

```
(define (< 名前 > < 仮パラメタ >) < 本体 >)
```

二乗を表す square という合成手続き (compund procedure) を作る

```
> (define (square x) (* x x))
> (square 2)
4
```

条件式と述語

■場合分け (case analysis)

・cond 場合分けを記述する特殊形式

```
(cond (<p1> <e1>)
      (<p2> <e2>)
       :
      (<pn> <en>))
```

```
> (define (abs2 x)
   (cond ((> x 0) x)
         ((= x 0) 0)
         ((< x 0) (- x))))
> (abs2 -19)
19
> (abs2 0)
0
```

・特殊記号 else を利用できる

```
> (define (abs3 x)
    (cond ((< x 0) (- x))
          (else x)))
```

・場合分けが 2 つの場合、特殊形式の if を利用できる

```
> (define (abs4 x)
    (if (< x 0)
        (- x)
```

```
x))
```

論理合成演算

・基本的述語以外に、以下のような論理合成演算が利用できる

論理合成演算
(and <e1> ... <en>)
(or <e1> ... <en>)
(not <e>)

手続きと変数

変数と有効範囲

■束縛変数 (bound variable)

・手続き定義のなかで、仮パラメータはどんな名前でもかまわないし、名前をすべて変更しても手続きの意味は変わらないという意味で、手続き定義は、仮パラメータを束縛している (bind)。そういう名前を束縛変数 (bound variable) という。変数が束縛されていなければ、自由である (free)。

■有効範囲 (Scope)

・名前が束縛されている式の範囲

■ブロック構造 (block structure)

・定義の入れ子。単純な名前保護の機構。
・手続きの定義の中で別の定義を行うことができる。

```
> (define (sqrt x)
    (define (square a)(* a a))
    (define (average a b)(/ (+ a b) 2))
    (define (good-enough? guess x)
      (< (abs (- (square guess) x )) 0.001))
    (define (improve guess x)
      (average guess (/ x guess)))
    (define (sqrt-iter guess x)
      (if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x) x)))
    (sqrt-iter 1.0 x))


> (sqrt 10)
3.162277665175675
```

利用者にとって、重要な処理は、sqrt であり、good-enough?、improve、sqrt-iter は重要ではないので、sqrt の中に隠す

■静的有効範囲 (lexicacl scoping)

・上記例で、x は、sqrt の定義に束縛されている。なので、その内側の関数は、x の有効範囲内にあるため明示的に渡さなくてもよいので、自由変数にできる。こうしたやり方を静的有効範囲 (lexical scoping) という。

```
(define (sqrt x)
    (define (square y) (* y y))
    (define (average a b) (/ (+ a b) 2))
    (define (good-enough? guess)
      (< (abs (- (square guess) x )) 0.001))
    (define (improve guess)
      (average guess (/ x guess)))
    (define (sqrt-iter guess)
      (if (good-enough? guess)
          guess
          (sqrt-iter (improve guess))))
    (sqrt-iter 1.0))
```

ブロック内に定義した関数の仮引数から、x を取り除き、本体の中で利用することができる

手続きとその生成するプロセス

・熟練者になるには、いろいろな手続きの生成するプロセスを視覚化することを学ばなければならない。
・そうしてから、期待した動きをするプログラムを作ることを学べる。

線形再帰と反復

■再帰的 (recursive process) プロセス

階乗の計算

n! = n・(n-1)・(n-2) ・・・3・2・1

```
(define (factorial n)
  (if (= n 1)
      1
  (* n (factorial (- n 1)))))
```

結果

```
> (factorial 4)
24
```

プロセスの視覚化

```
(factorial 4)
(* 4 (factorial 3))
(* 4  (* 3 (factorial 2)))
(* 4  (* 3  (* 2 (factorial 1))))
(* 4  (* 3  (* 2 1)))
(* 4  (* 3  2))
(* 4  6)
24
```

・プロセスの視覚化したものを見ると、膨張と収縮の形をとる
・膨張はプロセスが、遅延演算 (deferred operations) を作るときに起きる
・収縮は演算が実際に実行されるときに起こる
・n! の計算では、遅延乗算のれつの長さを覚えておくのに必要な量は、n に線形 (比例して) に成長する。こういうプロセスを線形再帰的プロセス (linear recursive process) という

■反復的プロセス (iterative process)

・階乗の手続きを書き直す

(define (factorial n)

```
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                 (+ counter 1)
                 max-count)))
(fact-iter 1 1 n))
```

プロセスを視覚化

```
(factorial 4)
(fact-iter 1 1 4)
(fact-iter 1 2 4)
(fact-iter 2 3 4)
(fact-iter 6 4 4)
(fact-iter 24 5 4)
24
```

・一般に反復的プロセスの状態は、状態が一定個数の状態変数 (state variables) と、状態が移ったときに状態変数をどうこうしんするのかの一定した規則とプロセスを停止させる条件を規定する終了テストで総括できる。
・n! の計算に必要なステップ数は、n に線形に成長する。そういうプロセスを線形反復的プロセス (linear iterative process) という。

■線形再帰と反復

＊反復ではどの時点においても、プログラム変数がプロセスの状態の完全な記述を持っている (計算の一時停止 / 再開に必要なのは、3 つのプログラム変数だけ) が、再帰プロセスではそうはならない (プログラム変数以外に、プロセスのいる場所という隠れた情報が必要)。

・再帰的プロセス (process) と再帰的手続き (procedure) を混同しないようにしなければならない (上記 facto-iter は記述構文としては再帰的だが、プロセスは反復的である)
・紛らわしいのは、通常の言語の実装で消費する記憶量がプロセスが原理的に反復的であっても、手続き呼び出し数とともに増加する (stack につまれる) ように設計してあるからである。
  ・これらの言語で反復プロセスは、do,repeat,for,while のような特殊目的のループ構造でしか記述できないが、Scheme(Racket) にはこの欠点がなく、固定スペースで実行できる。
  ・この性質の実装を末尾再帰的 (tail recursive) という。

木構造再帰

TODO 12/22