プログラミング入門
第 3 回「条件式と述語」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

プログラムの構造と解釈（第 1 章のつづき）

1.1.6 条件式と述語

他の手続き型プログラミング言語と同様、条件式によって評価を場合分けする方法が Scheme には用意されています。cond と if という特殊形式の条件式です。

```
(define (myabs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

cond の一般的な形は次のようになります。

```
(cond (⟨述語 1⟩ ⟨式 1⟩)
      (⟨述語 2⟩ ⟨式 2⟩)
      …
      (⟨述語n⟩ ⟨式n⟩))
```

真(true, racket では #t)か偽(false, racket では #f)を返す関数は述語と呼ばれ、インタプリターは上から述語を順に評価していき、⟨述語i⟩ が真であったときに対応する⟨式i⟩を評価して cond の返値とします。abs は次のようにも書けます。

```
(define (myabs2 x)
  (cond ((< x 0) (- x))
        (else x)))
```

else は cond の最後に記述できる特殊記号で、else より上の述語が全て偽であったときに、else に対応する式が評価されます。

abs を書くのにもう一つ if を使う方法もあります。

```
(define (myabs3 x)
   (if (< x 0) (- x) x))
```

if の一般的な形は次のようになります。

(if ⟨述語⟩ ⟨式 1⟩ ⟨式 2⟩)

述語が真ならば式 1 が評価され、偽ならば式 2 が評価されます。

Scheme には次の述語が用意されています。

=, <, >, <=, >=	等号、不等号
and, or, not	AND, OR, 否定の論理演算

=は(= ⟨式 1⟩ ⟨式 2⟩)というふうに用いて、式 1 と式 2 の結果の数値が等しければ真(#t)を返し、そうでなければ偽(#f)を返します。

<と>は不等号で、(< ⟨式 1⟩ ⟨式 2⟩)というふうに用いて、式 1 の数値が式 2 の数値よりも小さい場合に真(#t)が返され、そうでなければ偽(#f)が返されます。>については(> ⟨式 1⟩ ⟨式 2⟩)というふうに用いて、式 1 の数値が式 2 の数値よりも大きい場合に真(#t)が返され、そうでなければ偽(#f)が返されます。

<=と>=は等号付き不等号で、(<= ⟨式 1⟩ ⟨式 2⟩)と用いた場合、式 1 の数値が式 2 の数値と等しいかそれよりも小さい場合に真(#t)が返され、そうでなければ偽(#f)が返されます。(>= ⟨式 1⟩ ⟨式 2⟩)と用いた場合、式 1 の数値が式 2 の数値と等しいかそれよりも大きい場合に真(#t)が返され、そうでなければ偽(#f)が返されます。

and, or, not は論理演算の述語で、and は(and ⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩)と用い、⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩のすべてが真であれば真を返し、そうでなければ偽を返します。or は(or ⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩)と用い、⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩のいずれかが真であれば真を返し、そうでなければ、つまり、すべて偽であれば、偽を返します。not は(not ⟨p⟩)と用い、引数pの真偽を反転した値を返します。

ただし、and と or は特殊形式であり、(and ⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩)や(or ⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩)を評価したとき、左から右に向かって順に評価されます(⟨$p1$⟩ ⟨$p2$⟩ … ⟨pn⟩の順に評価されます)。and の場合、途中で式が偽を返した場合には、右側の残りの式を評価せずに and の返値として偽を返します。or の場合は途中で式が真を返した場合には同様に残りの式を評価せずに or の返値として真を返します。

ここで、特殊形式とは何かについて説明します。前回までに式が与えられたときの一般的な評価の方法について勉強しましたが、その評価規則に従わない例外のことを「特殊形式」と呼びます。ここで、(define x 3)と(+ x 3)を見比べて見ましょう。(+ x 3)は x という変数に格納された値が返されて、3 という値と足された結果が返されます。例えば、x に 5 という値が入っていたならば、(+ 5 3)が評価され、8 という値が返ってきます。一方、(define x 3)はどうでしょう。x という値に 5 がはいっていると、(define 5 3)ということになりますが、これでは元々define で行おうとしていたことと違ったことを行うことになってしまいます。つまり、define は上の評価手順で評価されない特殊な式ということになります。こういう一般的な評価規則の例外を「特殊形式」と呼びます。define の他、cond や if も特殊形式です。Scheme の特徴として、この特殊形式が非常に少ない数で言語設計が行われていることがあります。つまり、プログラマは一般的な評価規則と、非常に少数の特殊形式さえ理解してしまえば、Scheme を理解することができるようになっています。

数値演算と述語のための組み込み関数

Racket には数値演算のために次の組み込み関数が用意されています。

+, -, *, /, remainder	和(+), 差(-), 積(*), 商(/), 余り(remainder)
-	符号の反転。(- x)とすると x の符号が反転する。(* -1 x)と同じ。
abs	(abs x)とすると、x の絶対値を返す。
exp, expt, log	指数や対数の計算をする関数。(exp n)は e の n 乗。(expt x n)は x の n 乗。(log n)は n の自然対数。
min, max	(min x1 ... xn)は x1,...,xn の中で一番小さな数値を返す。(max x1 ... xn)は x1,...,xn の中で一番大きな数値を返す。
gcd	(gcd x y)は x と y の最大公約数を計算する。

また、DrRacket では次の述語も組み込みで用意されています。

even?	受け取った数値が偶数なら真(#t)を返し、そうでなければ偽(#f)を返します。
odd?	受け取った数値が奇数なら真(#t)を返し、そうでなければ偽(#f)を返します。

even?や odd?は remainder を用いて次のように定義した場合と同じになります。

```
(define (even? x) (= (remainder x 2) 0))

(define (odd? x) (= (remainder x 2) 1))
```

内部定義とブロック構造

次のプログラムをみてみよう。これは平方根を求めるプログラムになっています。

```
(define (square x) (* x x))

(define (mysqrt x) (sqrt-iter 1.0 x))
```

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

このプログラムは(mysqrt x)を呼び出すことによって、x の平方根を求めることができます。しかし、このプログラムを他のユーザーが使うことを考えると、他のユーザーにとっては、必要な関数は mysqrt だけなのに、他の good-enough?や improve や sqrt-iter といった関数の名前が自分のプログラムとかぶらないように気をつける必要がでてきます。また、どれが本当に必要な関数なのか一目ではわからないようになっています。他のユーザーだけでなく自分自身にとっても整理がつきにくいプログラムになっています。このようなプログラムは大勢のプログラマで大きなプログラムを作っていくときにどんどんと大きな問題となっていきます。そこで、Scheme では関数の中に局所的な関数を定義できるようになっていて、mysqrt を実装するために必要な関数は mysqrt の中で定義できるようになっています。

```
(define (mysqrt x)
   (define (average x y) (/ (+ x y) 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))
```

このように定義が入れ子になっていることをブロック構造(block structure)といいます。さらに、x は mysqrt 内で束縛されているので、improve や good-enough?にわざわざ x を渡す必要がありません。よって x を引数から外すことができます。

```
(define (mysqrt x)
  (define (average x y) (/ (+ x y) 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))
```