プログラミング入門

第 8 回「記号」

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

データによる抽象の構築（第 2 章の続き）

2.3 記号データ

これまで作ってきた合成データオブジェクトは最終的に数値で構成されていました。ここでは記号(symbol)を導入し、表現の幅を増やしていきます。

2.3.1 クォート

Scheme では記号(symbol)のデータオブジェクトを表現するためにシングルクォートを使います。例えば、a という記号を表現するには'a とします。変数との違いをみてみましょう。

```
> (define a 3)
> (define b 2)
> (list a b)
(3 2)
> (list 'a 'b)
(a b)
```

```
> (list 'a b)
(a 2)
```

となります。

リストの印字表現に対してクォートをつけることによって、そのリストをデータオブジェクトとして得ることができます。ただし、その場合、クォートをつけられたリストの中に出現する文字列は、変数ではなく、記号として扱われます。数は数のままとなります。

```
> (list a b)
(3 2)
> '(a b)
(a b)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

'(a b (a b))は(list 'a 'b (list 'a 'b))と等価であることに注意してください(ただし、代入操作を使うようになると違いが生じてくるので注意してください)。シングルクォートはリストを表現するのに大変便利ですが、リストの中に変数の値を代入したい場合は list や cons などの関数でデータをつくらないといけません。また、これにならうと、空リストが'()となることがよくわかると思います。

次に、数値の等価性は=で評価できましたが、記号の等価性については eq?で評価することができます。eq?を使うことによって memq という非常によく使われる関数が定義できます。(racket の R5RS に組み込みで定義されています)

```
(define (memq item x)
  (cond ((null? x) #f)
```

```
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
> (memq 'apple '(pear banana prune))
#f
> (memq 'apple '(x (apple sauce) y apple pear))
(apple pear)
```

item と eq?でマッチした要素を先頭とした部分リストが返ってきます。マッチする要素がない場合は false(#f)が返ってきます。ただし、木構造に対しては適用できない(再帰的に呼び出すようにはなっていない)ことに注意してください。(後者の例では、(apple sauce)ではなく、二つ目の apple にマッチして、(apple pear)を返しています)

記号(symbol)と似たデータ構造として文字列(string)というのがあります。文字列は普通のテキスト(e-mail の文書やテキストエディターで編集する文書など)など文字通り文字の列である文字列を扱うのに適しています。racket では”(ダブルクォーテーション)で囲むことで文字列を表現できます。

```
> (display "Hello, world!")
Hello, world!
```

記号にはスペースや括弧など Scheme の構文と相容れないキャラクターが使えない、記号を書き換えることはできない(例えば、'apple の 3 文字目を a にして、'apale にする、といったことができない)、という制約があって、一般の文字列を扱うには向いていません。いろんな文書や文字列を扱うときには文字列(string)を使いましょう。一方、記号は内部で独自の表現で管理されているため、文字列に比べ等価性の判定処理が非常に速い、という利点があります。一般の文字列を扱いたいときは文字列(string)を使って、効率的に文字列の等価性の判定処理を行いたい場合は記号を使う、というふうに使い分けると良いと思います。

2.3.2 例: 記号微分

記号とリストを用いることにより、数式などの抽象化されたデータを表現することが出来るようになります。

抽象データによる微分プログラム

（簡単のため）二つの引数を持った加算と乗算の演算だけからなる式を扱うことにします。そうすると、次の式があれば記号微分プログラムを実現することができます。

$\frac{dc}{dx} = 0$　（cは定数かxと異なる変数）

$$\frac{dx}{dx} = 1$$

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d(uv)}{dx} = u\left(\frac{dv}{dx}\right) + \left(\frac{du}{dx}\right)v$$

ここでは、数式を表現するのに、Scheme での数式表現と同様前置記法(prefix notation)で表現することとします。例えば、目標の記号微分プログラムを deriv として、$d(3xy+x)/dx$を求めたい場合、(deriv '(+ (* y (* x 3)) x) 'x) を実行したら、'(+ (* y 3) 1)が返ってくればよい、ということになります。

まず、次のような基本的なインターフェースがすでにあるとします。

(variable? e)　　e は変数か

(same-variable? v1 v2)　　v1 と v2 は同じ変数か

(make-sum a1 a2)　　a1 と a2 の和を構成

(make-product m1 m2)　　m1 と m2 の積を構成

(sum? e)　　e は和か

(product? e)　　e は積か

(arg1 e)　　e が和か積の式として、その第一引数を返す

(arg2 e)　　e が和か積の式として、その第二引数を返す

これらと数を見分ける基本述語 number?を使うと次のように記号微分プログラムを実現できる。

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (arg1 exp) var)
                   (deriv (arg2 exp) var)))
        ((product? exp)
         (make-sum
          (make-product (arg1 exp)
                        (deriv (arg2 exp) var))
          (make-product (deriv (arg1 exp) var)
                        (arg2 exp))))
        (else
         (error "unknown expression type --DERIV" exp))))
```

次に基本的なインターフェースも実装してみます。記号を識別するためには symbol?という基本関数を用いれば ok です。

```
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2) (and (variable? v1) (variable? v2) (eq? v1 v2)))
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
```

```
(define (sum? x) (and (pair? x) (eq? (car x) '+)))

(define (product? x) (and (pair? x) (eq? (car x) '*)))

(define (arg1 s) (car (cdr s)))

(define (arg2 s) (car (cdr (cdr s))))
```

これで、簡単な記号微分のプログラムを実現することができます。

```
> (deriv '(+ (* y (* x 3)) x) 'x)

(+ (+ (* y (+ (* x 0) (* 1 3))) (* 0 (* x 3))) 1)
```

冗長ではありますが、とりあえず、求めていた導関数が得られました。

さらに、(* x 0)は0、(* x 1)はx、(+ x 0)はxであることを用いれば表現を簡約することができます。また、(+ a b)や(* a b)のaとbがどちらも数値であった場合には実際に(+ a b)や(* a b)を計算してやればさらに簡約化することができます(上の例だと(* 1 3)の部分は実際にこれを計算して3と置き換えるようにすればよい)。make-sumとmake-productを次のように変更すればokです。

```
(define (=number? exp num) (and (number? exp) (= exp num)))

(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
 (define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
```

```
        (else (list '* m1 m2))))
```

すると、次のような結果が得られます。

```
> (deriv '(+ (* y (* x 3)) x) 'x)
(+ (* y 3) 1)
```