アルゴリズムとデータ構造入門
2.3 記号データ(Symbolic Data)
例を用いた練習問題

奥 乃 博

The Fifth Commandment

When building a value with `+`, always use `0` for the value of the terminating line, for adding `0` does not change the value of an addition
When building a value with `*`, always use `1` for the value of the terminating line, for multiplying by `1` does not change the value of a multiplication.
When building a value with `cons`, always consider `()` for the value of terminating line.

The Sixth Commandment

Simplify only after the function is correct.

The Seventh Commandment

Recur on the *subparts* that are of the same nature
- On the sublists of a list.
- On the subexpressions of an arithmetic expression.

(Friedman, et al. "*The Little Schemer*", MIT Press)

1

12月6日・本日のメニュー

- 今日は復習:
- 練習問題中心
- 配布する用紙に名前を記入して下さい。
- 今週の宿題と共に提出して下さい。

1. 記号微分(symbolic differentiation)
2. 集合(set)

2

左上教科書表紙:http://mitpress.mit.edu/images/products/books/0262011530-f30.jpg

微分を定義を思い出そう

- $\frac{dc}{dx} =$ c が定数か x以外の変数
- $\frac{dx}{dx} =$
- $\frac{d(u+v)}{dx} =$
- $\frac{d(uv)}{dx} =$

3

代数式の表現を考えてください

■次の代数式の表現法

1.和 $x+y$

2.積 ax

■代数式のための構築子・選択子・述語の設計

1.構築子(*constructor*)

```
(make-sum x y)
(make-product x y)
```

2.選択子(*selector*)

```
(addend s)
(augend s)
(multiplicant p)
(multiplier p)
```

3.述語(*predicate*)

```
(variable? x)
(same-variable? x y)
(sum? x)
(product? x)
```

4

代数式の構文

:= は定義　　| は代替

<expression> := *<number>* | *<variable>* |
(*<unary operator>* *<expression>*) |
(*<binary operator>* *<expression>* *<expression>*) |
(*<expression>*)

<unary operator> := + | - | *<function>*

<binary operator> := + | - | * | / | ^ | < | > | = | <= | >=

<function> := sin | cos | tan | log | …

5

代数式の表現法とその実装

■次の代数式の表現法

1. 和	$x+y$	(+ x y)	(+ (x y))
2. 積	ax	(* x y)	(* (x y))

■代数式のための構築子・選択子の設計

1.構築子

```
(define (make-sum x y) (list '+ x y))
(define (make-product x y)
      (list '* x y) )
```

2.選択子

```
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (multiplicant p) (cadr p))
(define (multiplier p) (caddr p))
```

6

代数式表現の実装(続)

■代数式のための構築子・選択子・述語の設計

3.述語

```
(define (variable? x) (symbol? x))
(define (same-variable? x y)
      (and (variable? x) (eq? x y) )
(define (sum? x)
   (and (pair? x) (eq? (car x) '+)) )
(define (product? x)
   (and (pair? x) (eq? (car x) '*)) )
```

7

代数式の表現を考えてください

■次の代数式の表現法

1. 和 $x+y$ (+ x y)
2. 積 ax (* x y)

```
(+ (x y))
(* (x y))
```

■代数式のための構築子・選択子の設計

1.構築子

```
(define (make-sum x y)
   (list '+ (list x y)) )
(define (make-product x y)
   (list '* (list x y)) )
```

2.選択子

```
(define (addend s) (caadr s))
(define (augend s) (cadadr s))
(define (multiplicant p) (caadr p))
(define (multiplier p) (cadadr p))
```

8

では微分手続きを定義してください

```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var) 1 0) )
         ((sum? exp)
           (make-sum (deriv (addend exp) var)
                     (deriv (augend exp) var)) )
         ((product? exp)
          (make-sum
             (make-product (multiplier exp)
                           (deriv (multiplicand exp)
                                  var ))
             (make-product (deriv (multiplier exp)
                                  var )
                           (multiplicand exp) )))
         (else
           (error "unknown expression type –
                   DERIV" exp ))))
```

9

代数式の表現・実装の問題点

1. (deriv '(+ x y) 'x)
   (+ 1 0)　　1
2. (deriv '(* x y) 'x)
   (+ (* y 1) (* 0 x))　　y
3. (deriv '(+ (* x y) (* 3 x)) 'x)
   (+ (+ (* y 1) (* 0 x)) (+ (* x 0) (* 1 3)))
   y+3

何かおかしい

簡略化を忘れている。

10

微分結果の簡約化（その1）

- どの時点で簡略化をすればよいか。

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

11

微分結果の簡約化（その2）

```
(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)) ))
```

12

微分結果の簡略化の実験

1. (deriv '(+ x y) 'x)
   1
2. (deriv '(* x y) 'x)
   y
3. (deriv '(+ (* x y) (* 3 x)) 'x)
   (+ y 3)

y+3

今度は

簡略化成功

13

和と積に対する微分を拡張

1. 差、商に拡張
   (deriv '(- x y) 'x)
   (deriv '(/ 3 x) 'x)
2. 冪乗に拡張
   (deriv '(** x 3) 'x)
3. 2項演算子を多項演算子に拡張
   (deriv '(+ (* 3 x) y (* x y)) 'x)
   (deriv '(* x y (+ x 3)) 'x)
4. 任意の関数が自由に付加できる微分システム

14

記号微分の拡張法

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0) )
        ((sum? exp)
          (make-sum (deriv (addend exp) var)
                    (deriv (augend exp) var)) )
        ((product? exp)
         (make-sum
            (make-product (multiplier exp)
                          (deriv (multiplicand exp)
                                 var ))
            (make-product (deriv (multiplier exp)
                                 var )
                          (multiplicand exp) )))
        (ここに微分ルールを追加)
        (else
          (error "unknown expression type - DERIV"
                 exp ))))
```

15

差・商に対する微分ルールを追加

1.差は (* -1 <exp>) で表現

```
(- x y)    (+ (* -1 x) y)
```

2.商は

```
(define (make-division d1 d2)
  (cond ((=number? d1 0) 0)
        ((=number? d1 1) d2)
        ((and (number? d1) (number? d2))
         (/ d1 d2))
        (else (list '/ d1 d2)) ))

(define (devident d) (cadr d))
(define (divisor d) (caddr d))
(define (division? x)
   (and (pair? x) (eq? (car x) '/)) )
```

16

商に対する微分ルールを追加

$$\frac{d}{dx}\frac{u}{v} = -\frac{u}{v^2}\frac{dv}{dx} + \frac{1}{v}\frac{du}{dx}$$

```
((division? exp)
 (make-sum
   (make-product
      (make-division
         (make-product
            (make-product -1
               (devident exp) )
            (deriv (divisor exp) var) )
          (make-product
             (divisor exp)
             (divisor exp)  )))
   (make-product
      (make-division 1 (divisor exp))
      (deriv (devident exp) x) )))
```

17

冪乗に対する表現法と基本手続き

```
(define (make-exponentiation b e)
   (cond ((=number? e 0) 1)
         ((=number? e 1) b)
         ((=number? b 1) 1)
         ((and (number? b) (number? e))
          (** b e) )
         (else (list '** b e)) ))

(define (base x) (cadr x))
(define (exponent x) (caddr x))
(define (exponentiation? x)
   (and (pair? x) (eq? (car x) '**)) )
```

18

冪乗に対する微分ルールを追加

$$\frac{d}{dx}b^e = eb^{e-1}\frac{db}{dx}$$

```
((exponentiation? exp)
 (make-product
   (make-product
      (exponent exp)
      (deriv (base exp) var) )
   (make-exponentiation
      (base exp)
      (make-sum (exponent exp) -1) )))
```

19

三角関数に対する微分ルールを追加

$$\frac{d}{dx}\sin(u) = \cos(u)\frac{du}{dx}$$

```
((sin? exp)
 (make-product
    (make-function
       'cos
       (argument exp) )
    (deriv (argument exp) var) ))

(define (make-function func . args)
   (cons func args) )
```

20

記号微分の拡張

1. 2項演算子を多項演算子に拡張
   augend, multiplier の定義を変更するだけで
   (deriv '(+ x (* x y) (** x 3)) 'x)
   に対応できる。
2. 多項式の整理
   - 多項式を降冪あるいは昇冪の順に整列
   - 多項式を簡略化により整理

   2.5.3 記号代数(Symbolic Algebra)
4. 任意の関数が自由に付加できる微分システム

   2.5.3 Data-Directed Programming and Additivity

23

写真:http://upload.wikimedia.org/wikipedia/en/0/0c/GraceHopper.jpg

Bugを最初に見つけたのは

- Grace Murray Hopper (Dec. 9, 1906 – Jan. 1, 1992)
- 1st women to receive a Ph.D in mathematics.
- Sep. 9, 1945. Mark II の Relay #70, Panel F で発見。
- bug はそれ以前から使用されていた。
- コンパイラ開発にも従事。
- NavyでCOBOL言語の検証ソフト開発

26

A Bug (moth) in the Mark-II

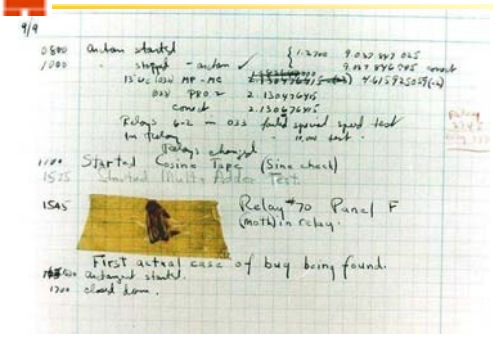

Photo #: NH 96566-KN (Color)

U.S. Naval Historical Center Photograph.

実験ノートをつけること。ルーズリーフはだめ

上写真:http://www.eingang.org/Lecture/graphics/slides/hmark1.gif
左下写真:http://www.ieee-virtual-museum.org/media/at3xJkunrmeT.jpg
右下写真:http://www-06.ibm.com/jp/event/museum/rekishi/i/mark1.gif

12月6日・本日のメニュー

- 今日は復習:
- 練習問題中心
- 配布する用紙に名前を記入して下さい。
- 今週の宿題と共に提出して下さい。

1. 記号微分
2. 集合

31

集合(set)の表現

- 自然数の集合を定義してみよう

1. {0, 1, 2, 3, …}
   外延的記法(extensional notation)
2. *S* = {*n*|0, *n*+1 *if n* ∈ *S*}
   内延的記法(intensional notation)

- 外延的記法での課題
  次の定義のどちらがよいか?

1. {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, … }
2. {0, 10, 20, 30, 2, 12, 22, 24, 4, 14, 24, … }

32

集合(set)の手続きと表現法

- 集合の手続き
  1. union-set S∪T
  2. intersection-set S∩T
  3. element-of-set? e∈T
  4. adjoin-set {e}∪S
- 集合の表現法
  1. 順序なし表現(*unordered list*)
     {30, 0, 20, 10, 22, 2, 12, 24, 34, … }
     (30 0 20 10 22 2 12 24 34 …)
  2. 順序付き表現(*ordered list*)
     {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, … }
     (0 2 4 6 8 10 12 14 16 18 …)

33

集合(set)のUnordered List表現

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)) ))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))

(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) ))))
```

35

union-set の両者の違いは

```
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) )
        )))
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (union-set (cdr s1)
                         (cons (car s1) s2)) ))))
```

36

集合(set)のunordered list表現(続)

```
(define (intersection-set s1 s2)
  (cond ((or (null? s1) (null? s2)) ())
        ((element-of-set? (car s1) s2)
         (cons (car s1)
           (intersection-set
               (cdr s1) s2 )))
        (else (intersection-set
                  (cdr s1) s2 ))))
```

37

集合手続きの計算量(#set=nとする)

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)) )) )

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))

(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) ))))
```

Θ(n)

Θ(n)

#s1=n
#s2=m
Θ(mn)

38

intersection-setの計算量

```
(define (intersection-set s1 s2)
  (cond ((or (null? s1) (null? s2)) ())
        ((element-of-set? (car s1) s2)
         (cons (car s1)
           (intersection-set
               (cdr s1) s2 )))
        (else (intersection-set
                 (cdr s1) s2 ))))
```

計算のオーダは #s1=m1, #s2=m2 とすると、

$\Theta(n^2)$ n=max{m1,m2} (m1*m2 のオーダ)

39

集合(set)のOrdered List表現

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)) )))
```

Θ(n) 平均的には n/2

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((= x (car set)) set)
        ((< x (car set)) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))
              ))))
```

Θ(n) 平均的にはn/2

40

集合(set)のOrdered List表現

```
(define (union-set s1 s2)
  (if (null? s1)
      s2
      (let ((x1 (car s1)) (x2 (car s2)))
         (cond ((= x1 x2)
                (cons x1
                      (union-set (cdr s1) (cdr s2))) )
               ((< x1 x2)
                (cons x1 (union-set (cdr s1) s2)) )
               (else
                 (cons x2
                       (union-set s1 (cdr s2))
                 )))))))
```

計算のオーダは　#s1=m1, #s2=m2　とすると、

Θ(n)　n=max{m1,m2}(m1+m2 のオーダ)

41

集合(set)のunordered list表現(続)

```
(define (intersection-set s1 s2)
  (if (or (null? s1) (null? s2))
      ()
      (let ((x1 (car s1)) (x2 (car s2)))
        (cond ((= x1 x2)
               (cons x1
                 (intersection-set (cdr s1) (cdr s2)) )
              ((< x1 x2)
               (intersection-set (cdr s1) s2) )
              (else
               (intersection-set s1 (cdr s2)) )))))
```

計算のオーダは　#s1=m', #s2=m とすると、

Θ(n)　n=max{m1,m2} (m+nのオーダ)

42

宿題:配布用紙と共に提出すること

- 宿題は、次の計7問:
- Ex.2.54, 2.56, 2.57, 2.58, 2.59,
- Ex.2.61, 2.62

12月11日午後5時締切

56