2.3.2 Symbolic Differentiation

```
(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)) ))
```

代数式の表現

```
    (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 (addend s) (cadr s))
    (define (augend s) (caddr s))
    (define (product? x) (and (pair? x) (eq? (car x) '*)))
    (define (multiplier p) (cadr p))
    (define (multiplicand p) (caddr p))
```

```
(define (make-sum a1 a2)  ; 簡略化
  (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)))
(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)) ))
```

2.3.3 Representing Sets

```
(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 (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1) (intersection-set (cdr set1) set2)) )
        (else (intersection-set (cdr set1) set2)) ))
```

順序木

```
(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))) ))
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2) (cons x1 (intersection-set (cdr set1) (cdr set2))))
              ((< x1 x2) (intersection-set (cdr set1) set2))
              ((< x2 x1) (intersection-set set1 (cdr set2))) ))))
```

二分木

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right) (list entry left right))
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (entry set)) true)
        ((< x (entry set)) (element-of-set? x (left-branch set)))
        ((> x (entry set)) (element-of-set? x (right-branch set))) ))
(define (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set) (adjoin-set x (left-branch set)) (right-branch set)) )
        ((> x (entry set))
         (make-tree (entry set) (left-branch set) (adjoin-set x (right-branch set)) ) )))
```

Converting a binary tree to a list

```
(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree) (tree->list-1 (right-branch tree))) )))
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list (right-branch tree) result-list) ))))
  (copy-to-list tree '()))
```

Converting an ordered list to a balanced binary tree

```
(define (list->tree elements)
  (car (partial-tree elements (length elements))) )
```

```
(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result (partial-tree (cdr non-left-elts) right-size)) )
              (let ((right-tree (car right-result))
                    (remaining-elts (cdr right-result)))
                (cons (make-tree this-entry left-tree right-tree) remaining-elts) )))))))
```

Sets and Information Retrieval

```
(define (lookup given-key set-of-records)
  (cond ((null? set-of-records) false)
        ((equal? given-key (key (car set-of-records)))
         (car set-of-records))
        (else (lookup given-key (cdr set-of-records)))))
```

2.3.4 Huffman Encodeing Trees

```
(define (make-leaf symbol weight) (list 'leaf symbol weight))
(define (leaf? object) (eq? (car object) 'leaf))
(define (symbol-leaf x) (cadr x))
(define (weight-leaf x) (caddr x))
(define (make-code-tree left right)
  (list left
        right
        (append (symbols left) (symbols right))
        (+ (weight left) (weight right)) ))
(define (left-branch tree) (car tree))
(define (right-branch tree) (cadr tree))
(define (symbols tree)
  (if (leaf? tree)
      (list (symbol-leaf tree))
      (caddr tree)) )
(define (weight tree)
  (if (leaf? tree)
      (weight-leaf tree)
       (cadddr tree)) )
;; decoding
(define (decode bits tree)
  (define (decode-1 bits current-branch)
    (if (null? bits)
        '()
        (let ((next-branch (choose-branch (car bits) current-branch)))
          (if (leaf? next-branch)
              (cons (symbol-leaf next-branch) (decode-1 (cdr bits) tree))
              (decode-1 (cdr bits) next-branch) ))))
  (decode-1 bits tree) )
```

```
(define (choose-branch bit branch)
  (cond ((= bit 0) (left-branch branch))
        ((= bit 1) (right-branch branch))
        (else (error "bad bit -- CHOOSE-BRANCH" bit)) ))
;; sets
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((< (weight x) (weight (car set))) (cons x set))
        (else (cons (car set) (adjoin-set x (cdr set)))) ))
(define (make-leaf-set pairs)
  (if (null? pairs)
      '()
      (let ((pair (car pairs)))
        (adjoin-set (make-leaf (car pair) (cadr pair))
                    (make-leaf-set (cdr pairs)) ))))
```

3　宿題 – メ切は従来通り: １２月２０日 (月) 午後５時 事務室レポート箱

1. Exercise 2.53 〜 2.58
2. (optional) 覆面算 SEND + MORE = MONEY を解け.
3. プログラムは動くことを確認すること.

4　補足

A	α	alpha
B	β	beta
Γ	γ	gamma
Δ	δ	delta
E	ϵ	epsilon
Z	ζ	zeta
Y	η	eta
Θ	θ	theta
I	ι	iota
K	κ	kappa
Λ	λ	lambda
M	μ	mu
N	ν	nu
Ξ	ξ	xi
O	o	omicron
Π	π	pi
R	ρ	rho
Σ	σ	sigma
T	τ	tau
Υ	υ	upsilon
Φ	ϕ	phi
X	χ	chi
Ψ	ψ	psi
Ω	ω	omega

塵劫記 (寛永８年版)

88plex	muryoutaisuu	無量大数
80plex	fukashigi	不可思議
72plex	nayuta	那由他
64plex	asougi	阿僧祇
56plex	gougasha	恒河砂
48plex	goku	極
44plex	sai	載
40plex	sei	正
36plex	kani	澗
32plex	kou	溝
28plex	jou	穣
24plex	jo	杼 (禾へん)
20plex	gai	垓
16plex	kei	京
12plex	chou	兆
8plex	oku	億
4plex	man	萬 (万)
3plex	sen	千
2plex	hyaku	百
1plex	juu	十
0plex	ichi	一

1minex	bu	分
2minex	rin	厘
3minex	mou	毫 (毛)
4minex	shi	絲 (糸)
5minex	kotsu	忽
6minex	bi	微
7minex	sen	纖 (繊)
8minex	sha	沙
9minex	jin	塵
10minex	ai	埃
11minex	byou	渺
12minex	baku	漠
13minex	moko	模糊
14minex	shunjun	逡巡
15minex	shuyu	須臾
16minex	shunsoku	瞬息
17minex	danshi	弾指
18minex	setsuna	殺那
19minex	rittoku	六徳
20minex	kyo	虚
21minex	kuu	空
22minex	sei	清
23minex	jou	浄