プログラミング言語論第5回

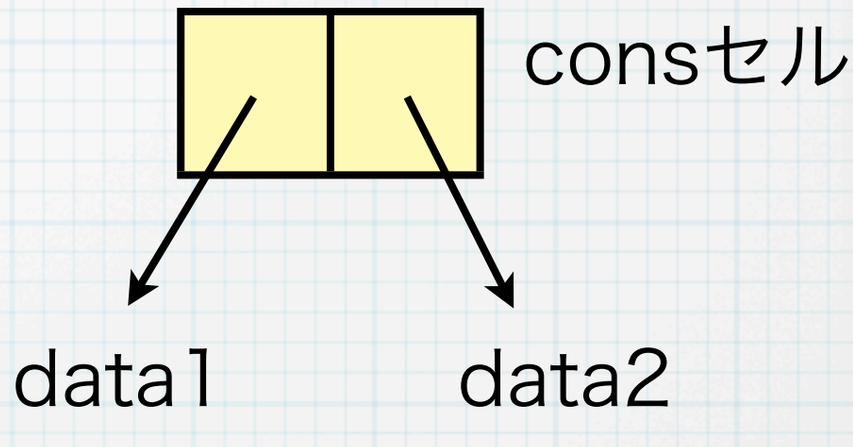
データによる抽象化（2）

情報工学科 山本修身

前回の復習 (1)

2つのデータを組み合せて1つのデータを作るためのデータとして consセルがある．consセルを作るための関数はconsである．

```
(cons data1 data2)
```



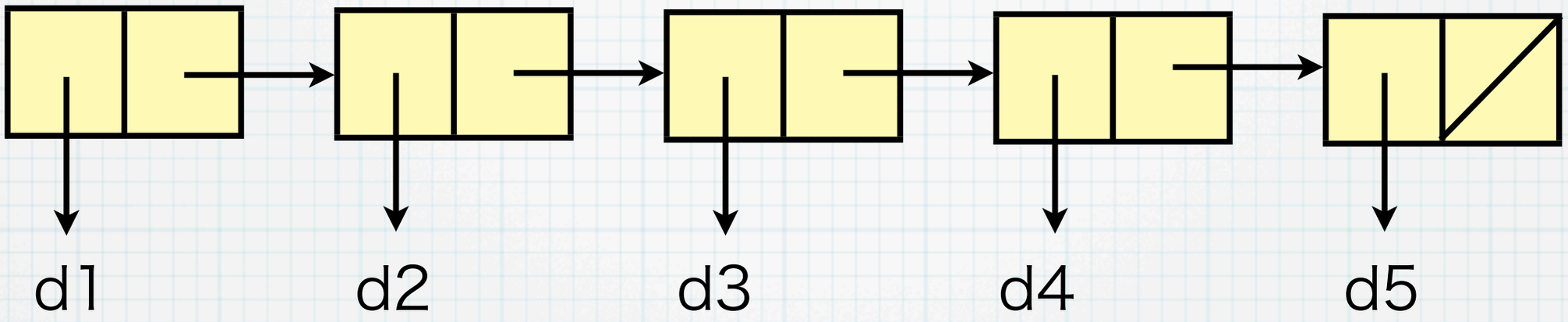
さらに，consセルのそれぞれ左側，右側のデータを取り出す関数として，carとcdrが用意されている．

```
(car pair)

(cdr pair)
```

```
p = (cons data1 data2)

    (car p) = data1
    (cdr p) = data2
```

前回の復習 (2)

consセルを連ねて，線形リストの構造を作ることができる.



線形リストを直接作る関数として list 関数がある.

```
(list d1 d2 d3 d4 d5)
        ||
(d1 d2 d3 d4 d5)
```

最後のセルの右側には何も入っていないが，これは通常( )と表現され，null値と呼ばれる．空のリストと考えても良い．nullであるか否かを判定する関数 null? が定義されている.

前回の復習(3)

与えられたS式を評価しないでそのまま値とするための特殊形式として，quoteが定義されている.

```
(quote S式)
```

通常，上記形式で書かなくても，'（クオート）をS式の先頭に置くことで同じ意味となる.

前回の復習 (4)

特殊形式 let によって，局所変数を定義し，letで定義された内部で式を評価することができる.

```
(let ((変数1 式1) ... (変数n 式n))
  本体の式1
  本体の式2
  ...
  本体の式m)
```

前回の復習 (5)

連想リストを定義し，それを用いることによって，色々な問題を可決することができる.

キー　値　キー　値

```
((key1 . val1) (key2 . val2) ...
               (keyn . valn))
```

集合の表現と集合演算 (1)

リストに含まれる要素を集合の要素と考えれば，リストによって集合を表現することができる．ここでは，集合の演算をSchemeで表現してみる．ここで集合の演算と言っているのは以下の演算である．

和集合： $A \cup B$

共通部分： $A \cap B$

差集合： $A \setminus B$ **Aに含まれてBに含まれない要素の集合**

ベキ集合： $\mathcal{P}(A)$ **Aのすべての部分集合の集合**

直積集合： $A \times B$

集合の表現と集合演算 (2)

まず，ある要素が集合に含まれるか否かを判定する関数を作る.

$x \in A$

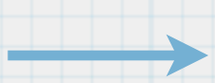
```
(define (element? x A)
  (cond ((null? A) #f)
        ((eq? x (car A)) #t)
        (else (element? x (cdr A)))))
```

element?を用いて共通部分を計算する関数 intersection を定義する.

$A \cap B$

```
(define (intersection A B)
  (if (null? A) '()
      (if (element? (car A) B)
          (cons (car A) (intersection (cdr A) B))
          (intersection (cdr A) B))))
```

```
(display (intersection
          '(a b c d e f)
          '(d e f g h i)))
(newline)
```



```
(d e f)
```

集合の表現と集合演算 (3)

また差集合を計算する関数 setminus を以下のように定義する.

```
(define (setminus A B)
  (if (null? A) '()
      (if (element? (car A) B)
          (setminus (cdr A) B)
          (cons (car A) (setminus (cdr A) B)))))
```

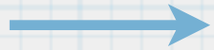
$A \setminus B$

これは，intersectionの定義の最後のifの中の2文が入れ替わっただけ. これを用いて，和集合を求める関数 union は以下のように定義される.

```
(define (union A B) (append (setminus A B) B))
```

$A \cup B$

```
(display (setminus
          '(a b c d e f)
          '(d e f g h i))) (newline)

(display (union
          '(a b c d e f)
          '(d e f g h i))) (newline)
```



```
(a b c)
(a b c d e f g h i)
```

集合の表現と集合演算 (4)

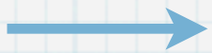
集合の直積はすべて組み合せを列挙することと同値である.

$$A \times B$$

```
(define (direct-product A B)
  (define (direct-product-one a B)
    (if (null? B) '()
        (cons (cons a (car B))
              (direct-product-one a (cdr B)))))
  (if (null? A) '()
      (append (direct-product-one (car A) B)
              (direct-product (cdr A) B))))
```

1つの要素と集合との直積

```
(display (direct-product
          '(a b c d e f)
          '(d e f g h i))) (newline)
```



```
((a . d) (a . e) (a . f) (a . g) (a . h) (a . i) (b . d) (b . e) (b . f)
 (b . g) (b . h) (b . i) (c . d) (c . e) (c . f) (c . g) (c . h) (c . i)
 (d . d) (d . e) (d . f) (d . g) (d . h) (d . i) (e . d) (e . e) (e . f)
 (e . g) (e . h) (e . i) (f . d) (f . e) (f . f) (f . g) (f . h) (f . i))
```

集合の表現と集合演算 (5)

最後にべき集合を計算する関数 powerset を以下のように定義する.

```
(define (power-set A)
  (define (union-one x A)
    (if (null? A) '()
        (cons (cons x (car A)) (union-one x (cdr A)))))
  (if (null? A) '(())
      (let ((pp (power-set (cdr A))))
        (append (union-one (car A) pp) pp))))
```

$\mathcal{P}(A)$

ベキ集合の計算の考え方

```
(let ((ans (power-set '(a b c d e f))))
  (display ans)(newline)
  (display (length ans)) (newline))
```

```
((a b c d e f) (a b c d e) (a b c d f) (a b c d) (a b c e f) (a b c e)
 (a b c f) (a b c) (a b d e f) (a b d e) (a b d f) (a b d) (a b e f) (a b e)
 (a b f) (a b) (a c d e f) (a c d e) (a c d f) (a c d) (a c e f) (a c e)
 (a c f) (a c) (a d e f) (a d e) (a d f) (a d) (a e f) (a e) (a f) (a)
 (b c d e f) (b c d e) (b c d f) (b c d) (b c e f) (b c e) (b c f) (b c)
 (b d e f) (b d e) (b d f) (b d) (b e f) (b e) (b f) (b) (c d e f) (c d e)
 (c d f) (c d) (c e f) (c e) (c f) (c) (d e f) (d e) (d f) (d) (e f) (e) (f) ())
64
```

ベキ集合の計算の考え方

集合Aのベキ集合 P(A) はAの部分集合をすべて集めたものである．直接部分集合をすべて列挙するのはやりづらい．そこで，以下のように考える．

1. Aに含まれるある要素 a に注目する．P(A)の要素でaを含むものと含まないものの2種類に分けることができる．

2. aを含まないものの全体は，P(A\{a}) である．

3. aを含むものの全体は P(A(\{a})の各要素にaを付け加えた集合である．

4. 以上2つの集合を合わせればAのベキ集合 P(A) ができる．

$$\mathcal{P}(A) = \mathcal{P}(A \setminus \{a\}) \cup \{s \cup \{a\} \mid s \in \mathcal{P}(A \setminus \{a\}\}$$

ランダムな集合による検算 (1)

ランダムな集合を作る関数 make-random-set を以下のように定義する．集合内に同一の値が2つ以上あると，集合の定義に反する．よって，その部分をチェックする必要がある．0〜M - 1の整数の要素n個の集合を返す．

inexact->exactは浮動小数点型数を整数型に変換する

```
(define (make-random-set n M)
  (define (rnum)
    (inexact->exact (floor (* (java.lang.Math:random) M))))
  (define (make-random-set-iter i res)
    (if (= i n) res
        (let ((m (rnum)))
          (if (element? m res)
              (make-random-set-iter i res)
              (make-random-set-iter (+ i 1) (cons m res))))))
  (make-random-set-iter 0 '()))
```

```
(display (make-random-set 20 100))(newline)
```

→

```
(27 70 19 97 45 47 43 55 44 17 11 73 14 16 67 88 8 68 38 20)
```

ランダムな集合による検算 (2)

実際に要素数1000の集合を２つ作って，共通部分と和集合を計算して，その大きさを測ってみる.

```
(let ((a (make-random-set 200 1000))
      (b (make-random-set 200 1000)))
  (let ((c (intersection a b))
        (d (union a b)))
    (display (length c)) (newline)
    (display (length d)) (newline)))
```

→

```
37
363
```

以下の包除定理が成り立っている.

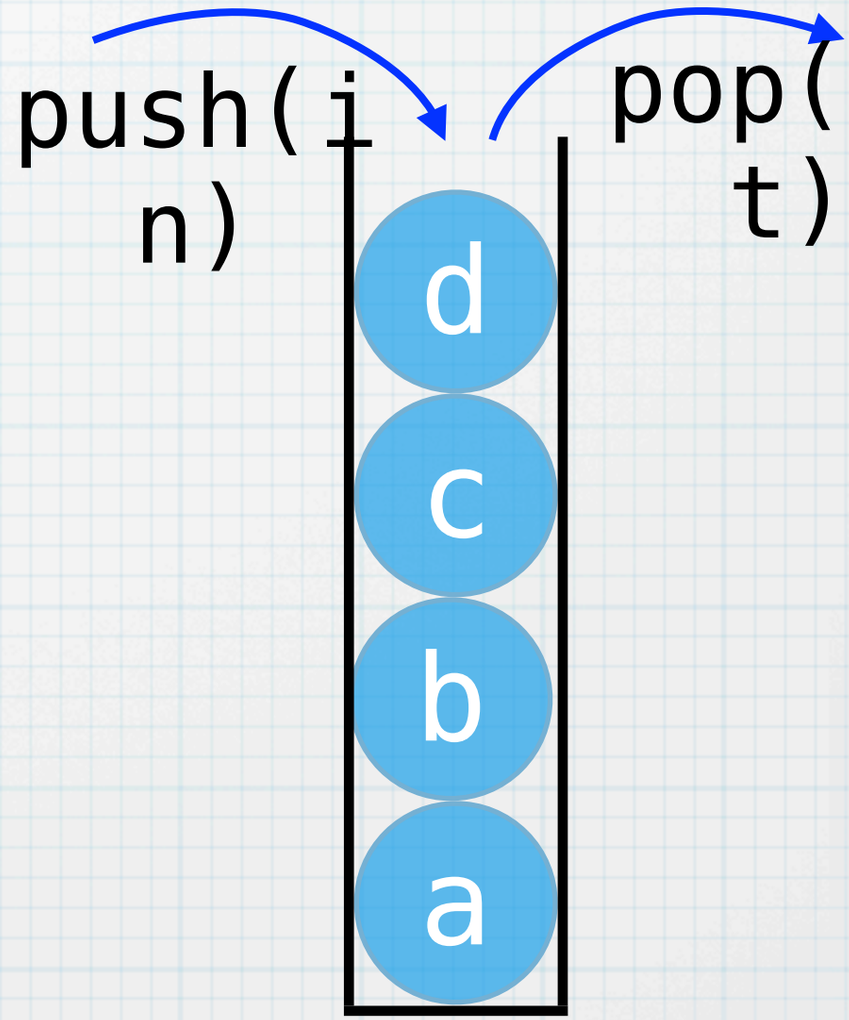
$$n(A \cup B) = n(A) + n(B) - n(A \cap B)$$

スタックの表現 (1)

スタックとは，LIFO (Last In First Out) とも呼ばれ，データを格納するための構造で，最後に入れたデータが最初に取り出されるものである．ちょうど．干し草などの山にたとえてスタックと呼ぶ．

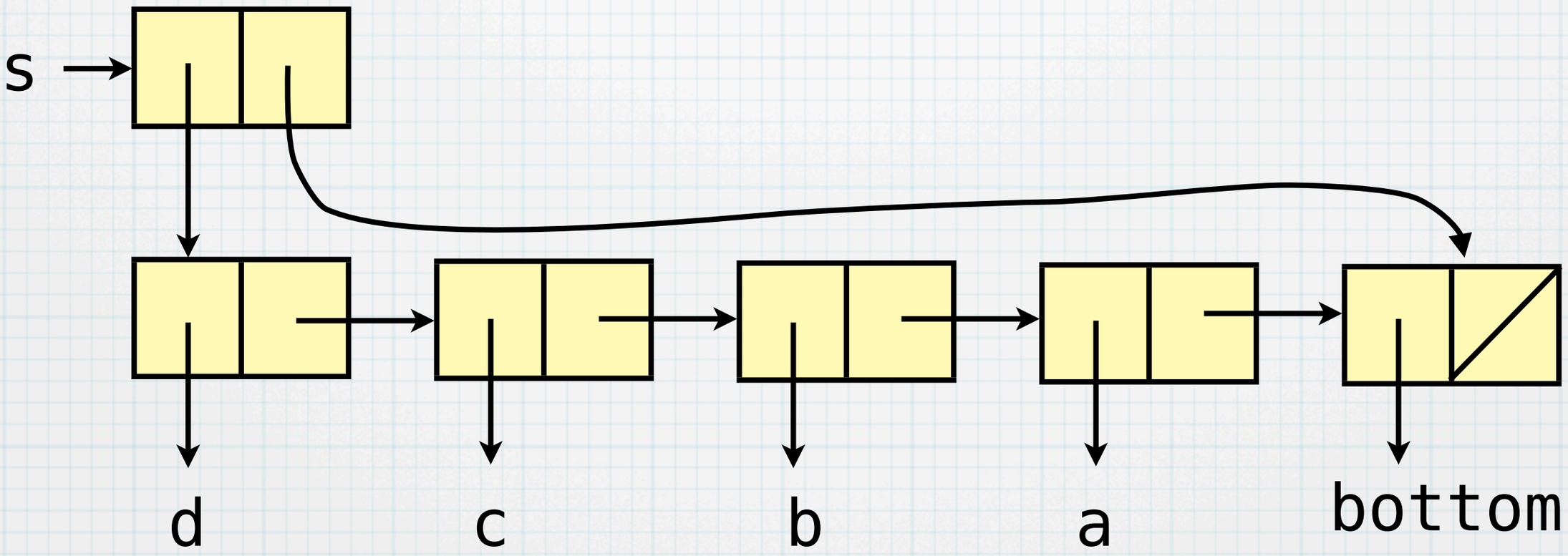
このしくみは計算機ではしばしば必要となるデータ構造で，たとえば，機械語におけるサブルーチン呼び出しのときの返り番地の処理や局所変数の実現などに普通に用いられている．

ここでは，consセルを用いてスタックを実現してみる．



スタックの表現 (2)

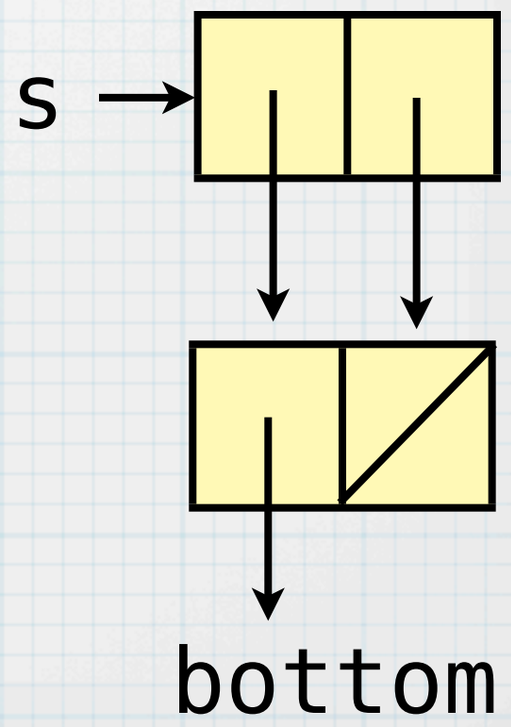
スタックは以下のように，線形リストとその先頭と最後の要素を指しているconsセルによって構成される．bottomを指しているセルは末端を表し，スタックに入っているデータは a, b, c, d であると考える．



スタックの表現 (3)

スタックを操作するプログラムは以下のとおり．スタックは初期状態で右下のようになっている．make-an-empty-stack によって空のスタックをつくり，それにデータを入れる時はpush, データを取り出すときはpopを用いる．stackが空であるか否かはstack-empty?で調べることができる．

```
(define (make-an-empty-stack)
  (let ((a (cons 'bottom '())))
    (cons a a)))

(define (push s a) (set-car! s (cons a (car s))))

(define (stack-empty? s) (eq? (car s) (cdr s)))

(define (pop s)
  (if (stack-empty? s) '()
      (let ((a (car (car s))))
        (set-car! s (cdr (car s)))
        a)))
```



スタックの表現 (4)

実際に動作するか否かを調べてみる.

```
(let ((s (make-an-empty-stack)))
  (display (stack-empty? s))(newline)
  (push s 1)
  (push s 23)
  (push s 'the-third-item)
  (display (pop s))(newline)
  (display (stack-empty? s))(newline)
  (display (pop s))(newline)
  (display (pop s))(newline)
  (display (pop s))(newline)
  (display (stack-empty? s))(newline))
```

```
osami-2:yama507> kawa stack.scm
#t
the-third-item
#f
23
1
()
#t
```

逆ポーランド式の電卓の作成 (1)

我々が普通に書く数式は演算子を中間に入れた 2 + 3, 4 × 5 のような形式のものである（この方式を中置記法 infix notation と呼ぶ）. この書き方は我々にとっては馴染みの深いもので直感的であるが, 2 + 3 × 4 のような式の場合, どちらの演算を先に行うかによって意味が変わってしまう. そのため, 演算子に優先順位 (precedence) を付ける必要がある. また, その場だけで優先順位を変更するために括弧を使う必要がある.

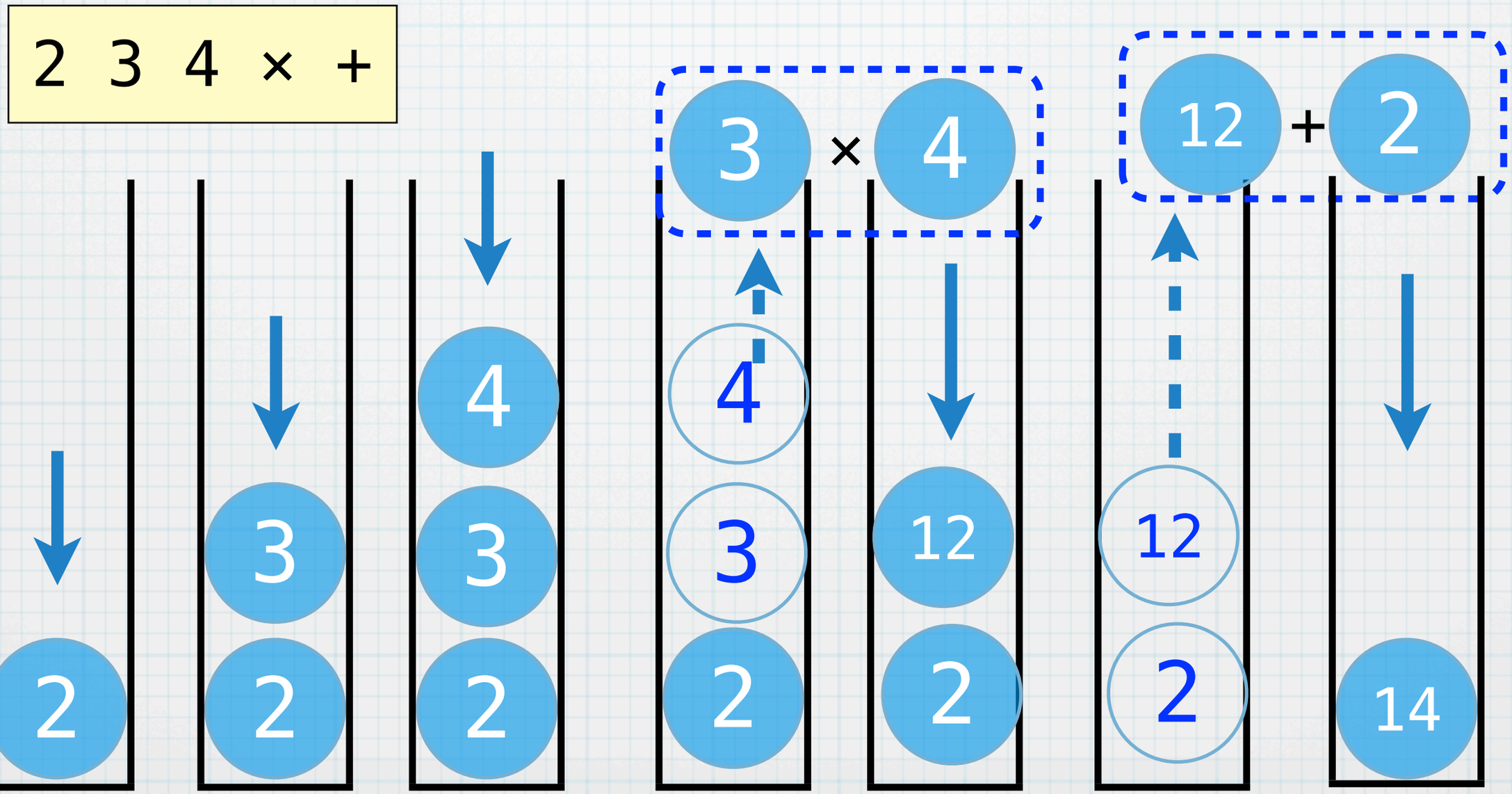
2 + 3 × 4　　**(2 + 3) × 4**　　**2 + (3 × 4)**

これに対して, 後置記法 (postfix notation; 逆ポーランド記法）がある. この場合, 演算子がいくつの数を引数としてとるかが分かっていれば優先順位を考える必要がない. 括弧も必要ない.

2 3 4 × +　　**2 3 + 4 ×**　　**2 3 4 × +**

逆ポーランド式の電卓の作成 (2)

逆ポーランド記法の式が与えられたとき，それを評価するのにスタックを用いることは自然である．計算される対象が出現したあと，最後に演算子が出てくるので，その時点で計算されている結果がスタックに乗っていれば，それを読み込んで計算することができる．



逆ポーランド式の電卓の作成 (3)

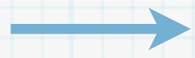
前述のスタックを用いて，逆ポーランド記法（後置記法）で書かれた式を計算するプログラムを作る.

```
(load "stack.scm")

(define the-stack (make-an-empty-stack))
(define (operate-a-list lst)
  (if (null? lst) '()
      (let ((ele (car lst)))
        (cond ((number? ele) (push the-stack ele))
              ((or (eq? ele '+) (eq? ele '-)
                   (eq? ele '*) (eq? ele '/))
               (let ((a (pop the-stack))
                     (b (pop the-stack)))
                 (push the-stack
                       ((cond ((eq? ele '+) +)
                              ((eq? ele '-) -)
                              ((eq? ele '*) *)
                              ((eq? ele '/) /)) a b))))
              ((eq? ele '=) (display (pop the-stack)) (newline)))
        (operate-a-list (cdr lst)))))
```

逆ポーランド式の電卓の作成 (4)

実際に計算させてみる.

```
(operate-a-list '(2 3 4 * + =))
(operate-a-list '(3 4 / 5 6 / + =))
```



```
OMacBook:yama510> kawa dentaku.scm
14
38/15
```

```
(define (operate-lists)
  (let ((lst (read)))
    (if (eof-object? lst)
        (begin
          (display "Finish.")(newline))
        (begin
          (operate-a-list lst)
          (operate-lists)))))
```

さらに連続的にS式を入力して評価するには左のプログラムのようにする.

```
(operate-lists)
```

readはS式を1つ読み込むための関数である.

読み込まれたオブジェクトがファイル末端か否かはeof-objet?でチェックする.

```
OMacBook:yama514> kawa dentaku2.scm
(3 4 5 * +)
(=)
23
(1 2 3 4 5 6 7 8 9 10 + + + + + + + + + + )
(=)
55
Finish.
```

練習問題

ここで解説した逆ポーランド形式の電卓は，最低限の四則演算を行う機能しか実装していない．以下の機能を実装してみよ．

(1) 変数が利用できるようにする．たとえば，以下のようにdefで変数定義をして，refで参照するようにする．

```
(abc 23 def)
(abc ref abc ref * =)
--> 469
```

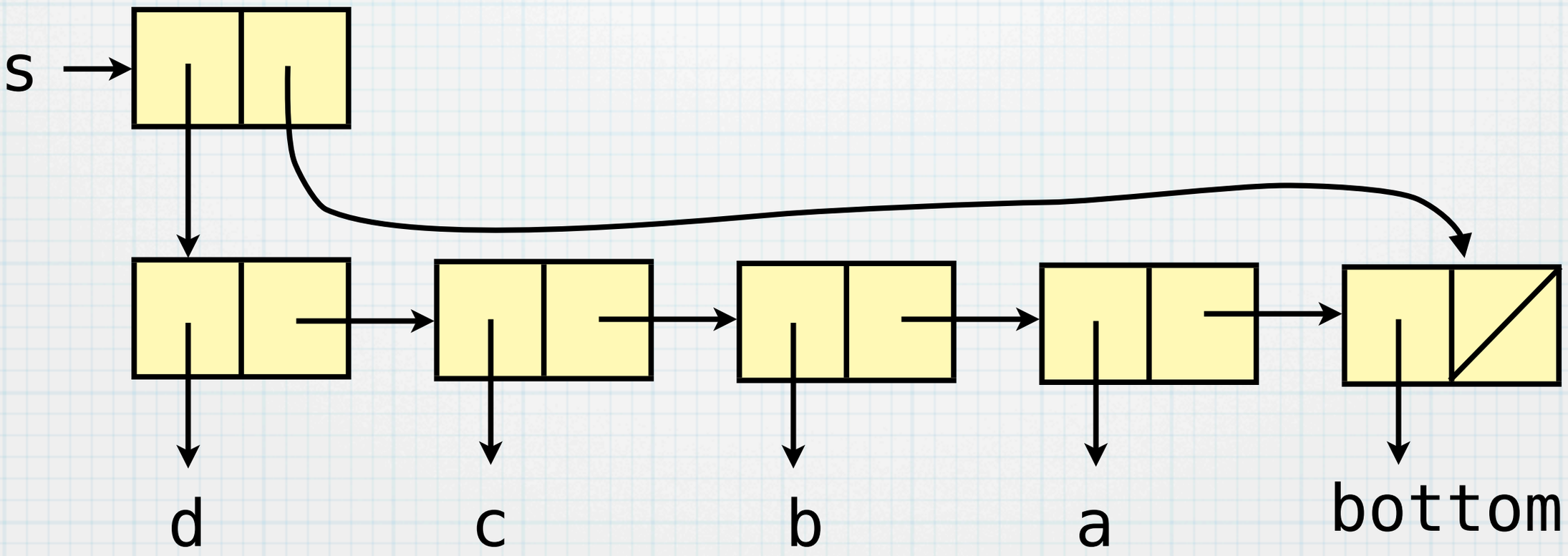
(2) 三角関数や指数関数などが利用できるようにする．たとえば，

```
(2.0 sin =)
--> 0.9092974268256817
(1.0 exp =)
--> 2.718281828459045
```

のように出力するようにする．

キューの表現 (1)

ここでは，スタックで用いた内部表現と同様のものを用いる．キューはFIFO (First In First Out) なので，一方の側からデータを入れて逆側からデータを取り出す．この場合は，最後尾にデータを入れて，先頭から取り出すことにする．

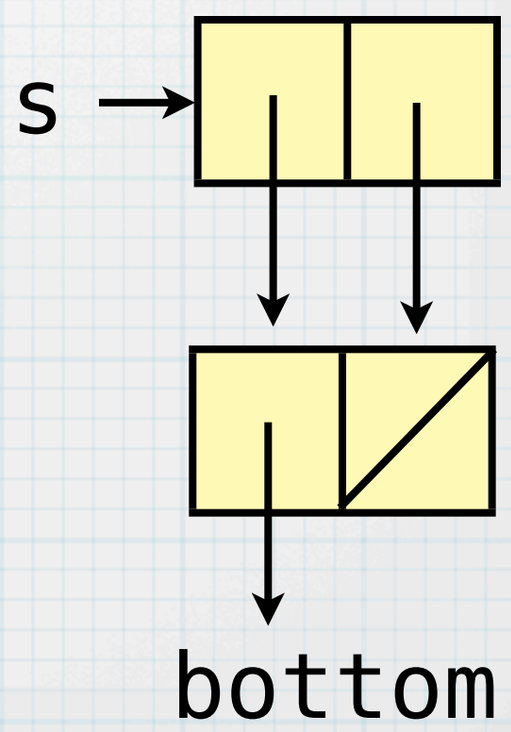


キューの表現 (2)

スタックと同様に初期状態では右下のようになる．データを入れる関数をenqueue, データを取り出す関数を dequeueとする.

```
(define (make-an-empty-queue)
  (let ((a (cons 'bottom '())))
    (cons a a)))

(define (enqueue s a)
  (let ((the-last (cdr s))
        (new-bottom (cons 'bottom '())))
    (set-car! the-last a)
    (set-cdr! the-last new-bottom)
    (set-cdr! s new-bottom)))

(define (queue-empty? s) (eq? (car s) (cdr s)))

(define (dequeue s)
  (if (queue-empty? s) '()
      (let ((a (car (car s))))
        (set-car! s (cdr (car s)))
        a)))
```

dequeueはstackにおけるpopと全く同じ



キューの表現 (3)

動きを確認してみる.

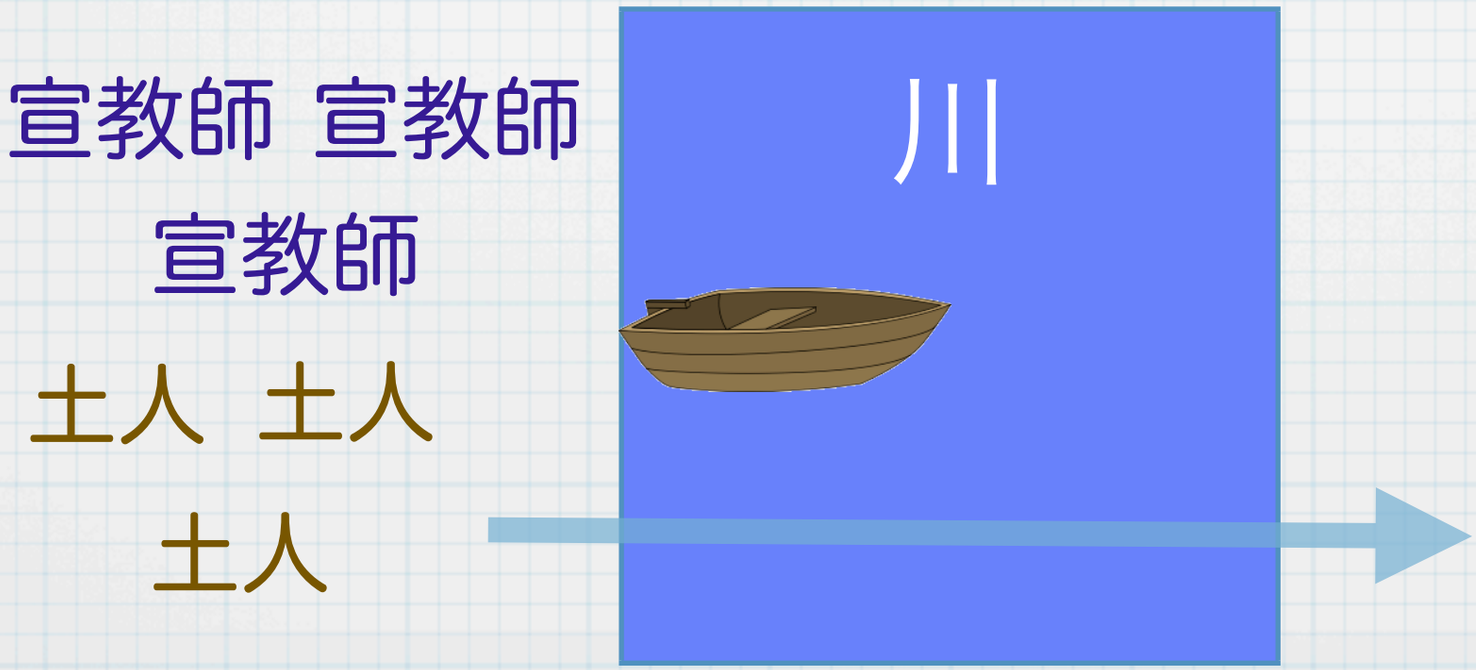
```
(define myqueue (make-an-empty-queue))

(define (enqueue-all lst)
  (if (null? lst) '()
      (begin
        (enqueue myqueue (car lst))
        (enqueue-all (cdr lst)))))

(enqueue-all '(1 2 3 4 5 6 7 8))

(define (dequeue-all queue)
  (if (queue-empty? queue) '()
      (begin
        (display (dequeue queue))
        (dequeue-all queue))))

(dequeue-all myqueue)
(newline)
```

```
OMacBook:yama529> kawa queue.scm
1 2 3 4 5 6 7 8
```

3人の宣教師と3人の土人 (1)

アルゴリズム・データ構造１で扱ったパズルをSchemeで解いてみる．パズルは以下のとおり．

> 川があり，川の片側に３人の宣教師と３人の土人がいる．全員で向こう岸へ渡りたい．２人乗りのボートが１隻ある．ボートを何回か往復させて渡るが，その際，どちらの岸についても土人の人数の方が宣教師の人数を上回ると，宣教師は食べられてしまう．宣教師が食べられないように渡るにはどうすればよい



3人の宣教師と3人の土人 (2)

○ 状態の表現を以下のように定義する

(左岸の土人の人数　左岸の宣教師の人数　ボートの位置　親状態)

```
(define (make-state c m pos parent) (list c m pos parent))
```

○ボートの乗り方の可能性は，ボートに乗る土人と宣教師の人数をconsセルで繋いで表現すれば，以下のようになる.

```
(define moves '((2 . 0) (1 . 1) (0 . 2) (1 . 0) (0 . 1)))
```

○ある状態 state で土人に宣教師が食べられてしまうか否かを判定する関数 check を以下のように定義する.

```
(define (check state)
  (let ((c (car state))
        (m (cadr state)))
    (if (or (and (> m 0) (< m c))
            (and (< m 3) (< (- 3 m) (- 3 c))))
        #f #t)))
```

3人の宣教師と3人の土人 (3)

ある状態で，対岸にボートで移動させたあとの状態を作る関数 move は以下のように定義する．stateは状態で，mpatは移動の仕方を表すcosセルである．移動できない場合には null = () を返す.

```
(define (move state mpat)
  (let ((pos (caddr state)))
    (let ((dc (car mpat))
          (dm (cdr mpat))
          (c (if (= pos 0) (car state) (- 3 (car state))))
          (m (if (= pos 0) (cadr state) (- 3 (cadr state)))))
      (if (or (> dc c) (> dm m)) '()
          (if (= pos 0)
              (make-state (- c dc) (- m dm) (- 1 pos) state)
              (make-state (+ (- 3 c) dc) (+ (- 3 m) dm)
                          (- 1 pos) state))))))
```

3人の宣教師と3人の土人 (4)

幅優先探索のための関数 search は以下のとおり.

```
(define (search the-queue)
  (define (enqueue-children state moves)
    (if (null? moves) '()
        (let ((next-state (move state (car moves))))
          (if (and (not (null? next-state))
                   (check next-state)
                   (check-eqstate state next-state))
              (enqueue the-queue next-state))
          (enqueue-children state (cdr moves)))))
  (let ((state (dequeue the-queue)))
    (if (null? state)
        (begin (display "No more Solutions.") (newline))
        (let ((c (car state))
              (m (cadr state))
              (pos (caddr state)))
          (if (and (= c 0) (= m 0))
              (begin
                (print-result state)
                (display "-----")
                (newline))
              (enqueue-children state moves))
          (search the-queue)))))
```

子供のノードで有効なものをすべてキューに入れるための関数.

キューが空ならば探索をやめる

解がひとつ見つかれば表示する

3人の宣教師と3人の土人 (5)

ルートからここまで探索の経路上に次の状態がすでに出現していないかどうかを調べる関数 check-eqstate を以下のように定義する.

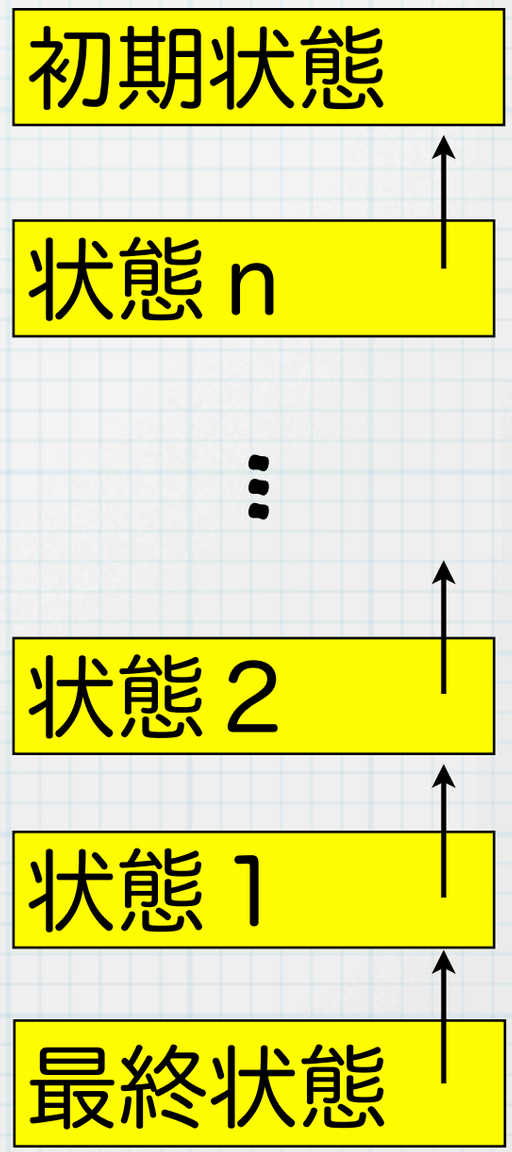
```
(define (check-eqstate st1 st2)
  (if (null? st1) #t
      (let ((c1 (car st1))
            (c2 (car st2))
            (m1 (cadr st1))
            (m2 (cadr st2))
            (pos1 (caddr st1))
            (pos2 (caddr st2))
            (parent (cadddr st1)))
        (if (and (= c1 c2) (= m1 m2) (= pos1 pos2)) #f
            (check-eqstate parent st2)))))
```

rootとst1 を結ぶ経路上にst2と同じ状態が見つかったら #t を返す．見つからなければ#tを返す.

3人の宣教師と3人の土人 (6)

ルートから現在の状態までの経路を表示するための関数 print-result を以下のように定義する.

```
(define (print-result state)
  (if (null? state) '()
      (let ((c (car state))
            (m (cadr state))
            (pos (caddr state))
            (parent (cadddr state)))
        (print-result parent)
        (display (list c m "|  |" (- 3 c) (- 3 m)
                       (if (= pos 0) "=>" "<=")))
        (newline))))
```



3人の宣教師と3人の土人 (7)

探索を実行して，すべての可能解を表示するには以下のようにする.

```
(define the-queue (make-an-empty-queue))
(enqueue the-queue (make-state 3 3 0 '()))
(search the-queue)
```

実行結果は以下のようになる.

```
OMacBook:yama576>
kawa cannibals.scm
(3 3 |   | 0 0 =>)
(1 3 |   | 2 0 <=)
(2 3 |   | 1 0 =>)
(0 3 |   | 3 0 <=)
(1 3 |   | 2 0 =>)
(1 1 |   | 2 2 <=)
(2 2 |   | 1 1 =>)
(2 0 |   | 1 3 <=)
(3 0 |   | 0 3 =>)
(1 0 |   | 2 3 <=)
(2 0 |   | 1 3 =>)
(0 0 |   | 3 3 <=)
-----
(3 3 |   | 0 0 =>)
(1 3 |   | 2 0 <=)
(2 3 |   | 1 0 =>)
(0 3 |   | 3 0 <=)
(1 3 |   | 2 0 =>)
(1 1 |   | 2 2 <=)
(2 2 |   | 1 1 =>)
(2 0 |   | 1 3 <=)
(3 0 |   | 0 3 =>)
(1 0 |   | 2 3 <=)
(1 1 |   | 2 2 =>)
(0 0 |   | 3 3 <=)
-----
(3 3 |   | 0 0 =>)
(2 2 |   | 1 1 <=)
(2 3 |   | 1 0 =>)
(0 3 |   | 3 0 <=)
(1 3 |   | 2 0 =>)
(1 1 |   | 2 2 <=)
(2 2 |   | 1 1 =>)
(2 0 |   | 1 3 <=)
(3 0 |   | 0 3 =>)
(1 0 |   | 2 3 <=)
(2 0 |   | 1 3 =>)
(0 0 |   | 3 3 <=)
-----
(3 3 |   | 0 0 =>)
(2 2 |   | 1 1 <=)
(2 3 |   | 1 0 =>)
(0 3 |   | 3 0 <=)
(1 3 |   | 2 0 =>)
(1 1 |   | 2 2 <=)
(2 2 |   | 1 1 =>)
(2 0 |   | 1 3 <=)
(3 0 |   | 0 3 =>)
(1 0 |   | 2 3 <=)
(1 1 |   | 2 2 =>)
(0 0 |   | 3 3 <=)
-----
```

本日のまとめ

リスト構造を用いて，集合，スタック，キューを実現した．すべては，consセルを元にして，構成されており，それ以上のものを使っているわけではない．