プログラミング言語論第1回
変数と関数の定義と評価のモデル

情報工学科 山本修身

この講義について

プログラミング言語のしくみについて学ぶ．実世界で使われているプログラミング言語には多様なものがあるが，色々なプログラミング言語に共通するしくみが存在する．それらのしくみをどのような言語で説明しても構わないが，多くの仕組みを包含するような言語の方が説明しやすい．C言語は計算機上で高速にプログラムを実行させるのには向いているが，このようなプログラム言語のしくみの多くを持たないという点において，このような作業に向いていない．

ここでは，Lisp言語の方言であるScheme言語を採用し，プログラミング言語のしくみについて詳しくみていく．

講義データ

講義名： プログラミング言語論
担当： 山本修身 (osami@meijo-u.ac.jp)
対象年次： 3年次
教室： N-206
日時： 金曜日3限 13:10〜14:40
ホームページ： `http://osami.s280.xrea.com/ProgLang2015/`
授業のスタイル： 講義（教室で説明をします．授業内容について講義のあと各自確認をすること．特に解説したプログラムについては各自確認すること．）

講義の目標： プログラミング言語の色々なしくみについて理解する．そのために，単純なしくみをもつプログラミング言語 Schemeを学び，そこから色々なしくみを探っていく．

プログラムの実行環境： Kawa (Scheme) (1)

この講義で扱う言語は，Schemeである．Scheme言語は色々な実現があるが，ここでは，Java言語で書かれたKawaを用いて説明する．Javaが動く環境であれば，動かすことができる．インタプリターなので，普通に立ち上げるとプロンプトがでてくる．

```
osami-2:yama502> java -jar kawa-1.7.jar
#|kawa:1|# (car '(a b c))
a
#|kawa:2|# (* 123123132 43234234243)
5323134329619809076
#|kawa:3|#
```

プロンプト

kawaのホームページ： http://www.gnu.org/software/kawa/

プロンプトを表示されて，それに対して人間が入力し，さらにそれに対してインタプリターが結果を返すというしくみを**REPL (read-eval-print loop)** と呼ぶ．

プログラムの実行環境： Kawa (Scheme) (2)

プログラムファイルを実行させる場合には，以下のようにする．

プログラム： sample.scm

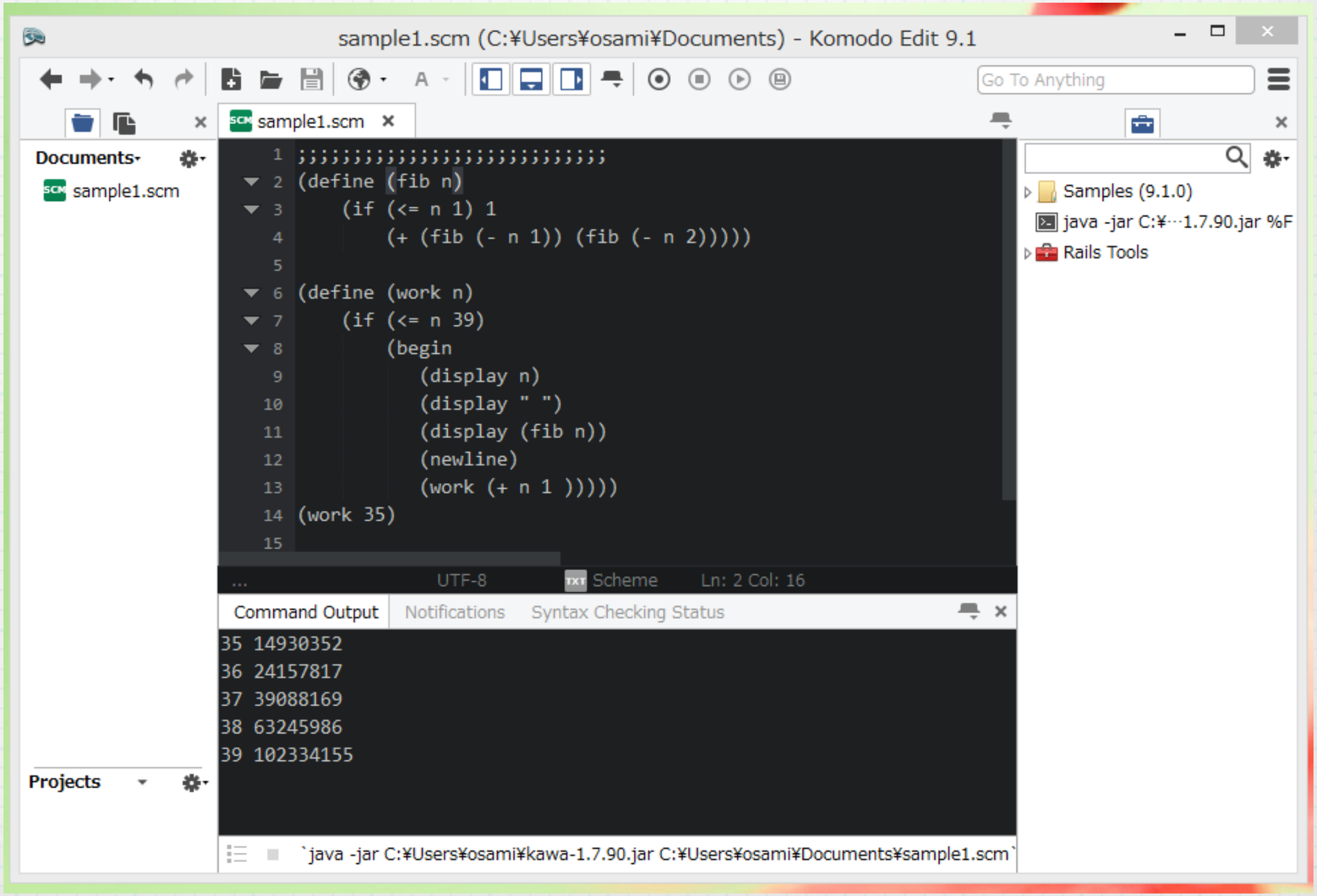
```
(define (exp a n)
  (if (zero? n) 1
      (let ((m2 (exp a (quotient n 2))))
        (if (even? n) (* m2 m2)
            (* m2 m2 a)))))

(display (exp 2 1000))
(newline)
```

```
osami-2:yama509> java -jar ./kawa-1.7.jar sample.scm
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
osami-2:yama510>
```

MS Windows 上での kawaの実行

KawaはjarファイルとしてOSなどによりファイルを変える必要はない．基本的には，前述の例と同様に，`java -jar ./kawa-1.7.jar sample.scm` のように実行する．コマンドプロンプトで実行しても良いが，たとえば，ActiveState社のkomodoエディタを用いれば以下のようになる．



色々なScheme言語処理系

Scheme言語の処理系にはコンパイラおよびインタプリタがある．本講義では主にインタプリタを用いて説明するが，効率が問題になる場合にはコンパイルして実行形式に変換して動かすことができる．

いくつかのインタプリタはweb上で動作する．たとえば，codepadではMzSchemeが動作するし，ideoneではguileが動作する． また，repl.it 上では biwascheme が利用できる．作ったプログラムの動作を確認するだけであれば，このようなweb上のシステムを利用することができる．

ある程度効率よく動かすには，schemeシステムをインストールする必要があり，前述のkawa以外にも mit-scheme，gambit scheme，gauche など色々な処理系が開発されている．

プログラムとデータの構成するS-式 (1)

SchemeのプログラムはS-式 (S-expression) と呼ばれるデータで構成されている S-式は以下のようなものである． Schemeで現れるデータはS-式以外にない.

1. アルファベット，数字，`_`，`-`，`?`，`<`，`>`，`+`，`*`，`/` などで構成されるアトムはS-式である.
2. 数はS-式である.
3. 文字列rはS-式である.
4. s_1, ..., s_nがS-式のとき，(s_1 s_2 … s_n)はS-式である.
5. s_1, ..., s_n, s_{n+1} が S-式のとき，(s_1 s_2 … s_n . s_{n+1}) はS-式である.

(a b c d) のように()で構成されたものを **リスト (list)** と呼ぶ.

Schemeの文法にはこの他に ’ (クオート）や , （コンマ）などの マクロの記法があるがここでは説明に含めない.

プログラムとデータの構成するS-式 (2)

S-式の例

```
(abc def 123 443 8273)

(+ 232 23312)

(find-current-list this-list foo bar)

((2 3 4) (4 3 2) (6 5 4) (1 1 1))

(((((34 45)))))

(a b c d e f g)

(2343 “abcde” abcde 3we . 2343)

((abc . def) (mmm . 332) (mmm . 232))
```

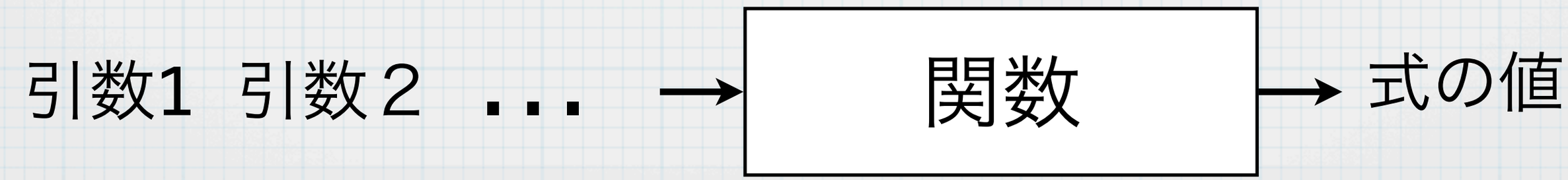
評価できる S-式

S-式によって色々な構造をもつデータを表現することができる.
SchemeではプログラムもS-式によって表現する.

さらに，Schemeでは表現されたプログラムを**評価 (evaluate)** することにより結果が得られる．評価することが計算そのものである．評価できる式は以下のような一定の構造をもつ必要がある.

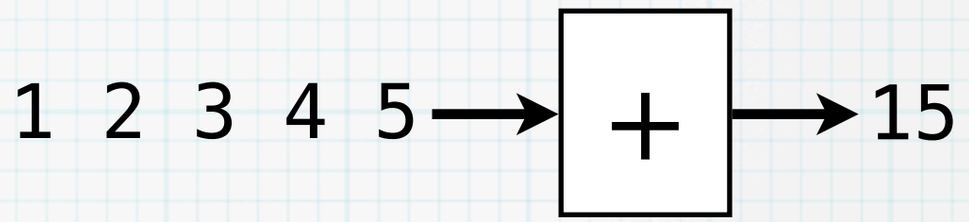
```
(関数　引数1　引数２　...　)
```

ここで，「関数」は入力を出力に変える能力をもつものである．「引数1」，「引数２」，... はこの関数に与えられるデータである．この関数の出力はこの式の評価値となる.

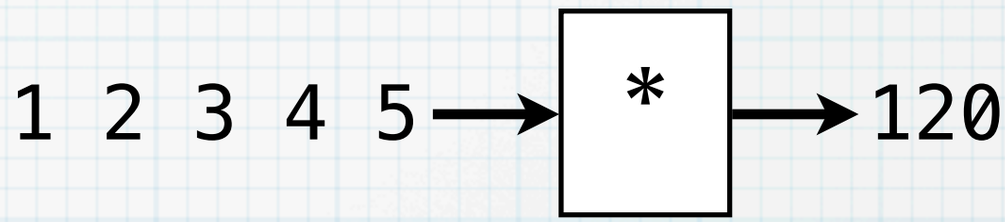


S-式の評価の例 (1)

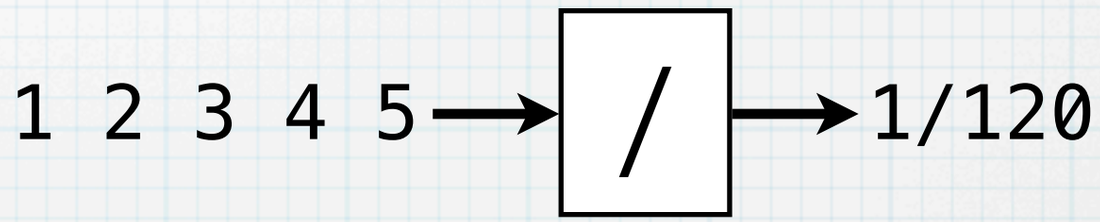
```
(+ 1 2 3 4 5) => 15
```




```
(* 1 2 3 4 5) => 120
```




```
(/ 1 2 3 4 5) => 1/120
```




```
#|kawa:5|# (+ 1 2 3 4 5)
15
#|kawa:6|# (* 1 2 3 4 5)
120
#|kawa:7|# (/ 1 2 3 4 5)
1/120
#|kawa:8|#
```

S-式の評価の例 (2)

もっと複雑な式を作ることもできる.

$$\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+\frac{1}{3\cdot 4}+\frac{1}{4\cdot 5}+\frac{1}{5\cdot 6}$$

```
(+ (/ 1 1 2) (/ 1 2 3) (/ 1 3 4) (/ 1 4 5) (/ 1 5 6))
=> 5/6
```

この場合，まず内部のS-式を評価して値を確定させてからそれを用いて外側のS-式を評価する.

```
#|kawa:10|# (+ (/ 1 1 2) (/ 1 2 3) (/ 1 3 4) (/ 1 4 5) (/ 1 5 6))
5/6
```

S-式の評価の例 (3)

文字列の連結： “abc” + “def” + “xyz”

```
#|kawa:14|# (string-append "abc" "def" "xyz")
abcdefxyz
```

文字列の長さ： |”meijo-university-information-engineering”|

```
#|kawa:16|# (string-length "meijo-university-information-engineering")
40
```

文字列を大文字にする：”hello” => “HELLO”

```
#|kawa:17|# (string-upcase "hello")
HELLO

#|kawa:18|# (string-upcase "Meijo University")
MEIJO UNIVERSITY
```

変数の導入

Schemeでは変数を定義して利用することができる．変数を定義する方法はいくつかあるが，一番単純な方法は `define` を用いることである．Schemeでは変数は型をもたない（データは型を持つ）．したがって，任意の変数に任意のデータ（S-式）を結ぶことができる．変数を定義する場合には以下のようなS-式を評価する．

```
(define 変数名 値)
```

```
#|kawa:19|# (define r 2.0)
#|kawa:20|# (define pi 3.1415926)
#|kawa:21|# (define s (* pi r r))
#|kawa:22|# s
12.5663704
```

S-式の表示

S式を画面に表示するには display 関数を用いる．displayは特殊な出力（(void) を返す．画面に表示されることが，この関数については重要である．関数 display に渡される引数はもちろん評価されてから表示される．

```
(display S−式)
```

display関数は改行を出力しない．そのため改行を出力するにはnewline関数を用いる．newline関数は引数を持たない．

```
(newline)
```

簡単なプログラム例 (1)

プログラムの中で漢字を使う場合には，UTF-8で記述すること．

円の面積を計算するプログラムは以下のように書ける．プログラムは評価すべき式を順にソースファイルに並べていけば良い．改行は関係ないが，行の中に；があるとそれ以降，改行までがコメント文となる．

simpleProg.scm

```
;; プログラム例

(define r 2.0)          ; 半径
(define pi 3.1415926) ; 円周率
(define s (* pi r r)) ; 面積の計算
(define el (* 2 pi r)); 円周長の計算

;; 計算結果表示

(display "r = ")(display r)(newline)
(display "s = ")(display s)(newline)
(display "el = ")(display el)(newline)
```

```
OMacBook:yama507> java -jar kawa-1.7.jar simpleProg.scm
r = 2.0
s = 12.5663704
el = 12.5663704
OMacBook:yama508>
```

変数への値の代入

set! を用いて，すでに定義されている変数の値を変更することができる．（変数が定義されていないと，エラーになるのが本来だが，kawaの場合には適当に変数が定義されるようである．）

(set! 変数名 値)

```
#|kawa:8|# (define a 2)
#|kawa:9|# (set! a (* a a a a))  ; 2^4
#|kawa:10|# (set! a (* a a a a)) ; 2^16
#|kawa:11|# (set! a (* a a a a)) ; 2^64
#|kawa:12|# (set! a (* a a a a)) ; 2^256
#|kawa:13|# a
115792089237316195423570985008687907853269984665640564039457584007913129639936
```

簡単なプログラム例 (2)

指数関数の底 e を計算する．e^xは以下のように近似することができる．

$$e^x \fallingdotseq 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

$$= \left(\left(\cdots \left(\left(\frac{x}{n} + 1 \right) \frac{x}{n-1} + 1 \right) \cdots \right) \frac{x}{2} + 1 \right) x + 1$$

n = 20とすれば，以下のようなプログラムが得られる．

```
(define p (+ 1.0 (/ 1.0 20)))
(set! p (+ (/ p 19) 1))
(set! p (+ (/ p 18) 1))
(set! p (+ (/ p 17) 1))
(set! p (+ (/ p 16) 1))
(set! p (+ (/ p 15) 1))
(set! p (+ (/ p 14) 1))
(set! p (+ (/ p 13) 1))
(set! p (+ (/ p 12) 1))
(set! p (+ (/ p 11) 1))
```

```
(set! p (+ (/ p 10) 1))
(set! p (+ (/ p 9) 1))
(set! p (+ (/ p 8) 1))
(set! p (+ (/ p 7) 1))
(set! p (+ (/ p 6) 1))
(set! p (+ (/ p 5) 1))
(set! p (+ (/ p 4) 1))
(set! p (+ (/ p 3) 1))
(set! p (+ (/ p 2) 1))
(set! p (+ (/ p 1) 1))
(display p)
(newline)
```

```
#|kawa:27|# (load "exp.scm")
2.718281828459045
```

関数を定義する

いくつかのまとまった動きを一つをまとめて一つの命令にすることができる．まとまった動きは関数として呼び出して実行させることができる．関数定義は変数定義の同じように以下のように行う．

```
(define (関数名 引数1 ... 引数n) 式1 式2 ... 式m)
```

関数本体

関数を呼び出すと，式1 式2, ... 式mが順に評価されて，式の評価値は式mの評価値となる．

```
#|kawa:28|# (define (dd x) (+ x x))
#|kawa:29|# (dd 23)
46
#|kawa:30|# (dd 3847)
7694
```

簡単なプログラム例 (2)：改良

前述のプログラム例は関数を用いることでより見通しが良くなる.

```
(define p (+ 1.0 (/ 1.0 20)))
(define (iter n) (set! p (+ (/ p n) 1)))
(iter 19)(iter 18)(iter 17)(iter 16)(iter 15)
(iter 14)(iter 13)(iter 12)(iter 11)(iter 10)
(iter 9)(iter 8)(iter 7)(iter 6)(iter 5)
(iter 4)(iter 3)(iter 2)(iter 1)
(display p)
(newline)
```

関数iterの内容はいちいち書かなくてもiterを用いて表現することができる. これにより, プログラムの行数は飛躍的に少なくなる. しかし, 「繰り返し」ができないので, ひとつずつ命令を書かなければならないの不自由である.

条件分岐の導入 (1)

ある条件が成り立つか否かによって，評価値を変えるような動作を導入することにより，プログラムをより柔軟に動かすことが可能になる．そのための命令として以下のような if を導入する．

```
(if 条件式 式1 式2)
```

```
(if 条件式 式1)
```

ifには2つの形式がある．最初の形式は条件式が真であれば式1を評価，偽であれば，式2を評価して式の値とする．後者は条件式が真であれば，式1を評価して値とし，偽であれば値が無いことを意味するvoid値を返す．

```
#|kawa:1|# (define n 23)
#|kawa:2|# (if (< n 30) (display "yes"))
yes
#|kawa:3|# (if (>= n 30) (display "yes"))
#|kawa:4|# (if (odd? n) (display "yes"))
yes
#|kawa:5|#
```

条件分岐の導入 (2)

さらに別のタイプの条件分岐を導入する．ある条件が成り立つとき，一連の式を評価する命令である．

```
(when 条件式 式1 式2 ... 式n)
```

条件式が真のとき，式1, 式2, ... , 式n を順に評価し，式の値を式nの値とする．

簡単なプログラム例 (2)：改良の改良

関数 work を以下のように定義する．(work n) は最初に (iter n)を呼び出して，その後，(work (- n 1)) を呼び出して一つ小さなnについて実行する．ただし，nが0になったらやらない．これで繰返しが表現できる．

```
(define p (+ 1.0 (/ 1.0 20)))
(define (iter n) (set! p (+ (/ p n) 1)))
(define (work n)
  (when (> n 0) (iter n) (work (- n 1))))
(work 19)
(display p)
(newline)
```

このプログラムの問題点はpという変数にいちいち代入し直している点で，繰返しをいちいち手で表現しなくて済んでいるので，そこは良いが，今ひとつ美しくない．

簡単なプログラム例 (2)：改良の改良の改良

そこで，全体的にpの内容を関数の引数として持ち回るプログラムを作る．この場合，workの引数pに値がどんどん更新されて入って行く．最初の値は計算して与える．

```
(define (work n p)
  (if  (= n 0)
       p
       (work (- n 1) (+ (/ p n) 1))))
(display (work 19 (+ 1.0 (/ 1.0 20))))
(newline)
```

以前のプログラムに比べるときれいな形になったが，割り算をしてから1を足すという部分が2カ所出て来たり，まだ今ひとつすっきりしない．

簡単なプログラム例 (2)：改良の改良の改良の改良

そこで，n = 20の部分も分離しないで，同じプログラムで表現することにして以下のようなプログラムが得られる．このプログラムは簡潔で美しい形になっている．

```
(define (work n p)
  (if  (= n 0)
       p
       (work (- n 1) (+ (/ p n) 1))))
(display (work 20 1.0))
(newline)
```

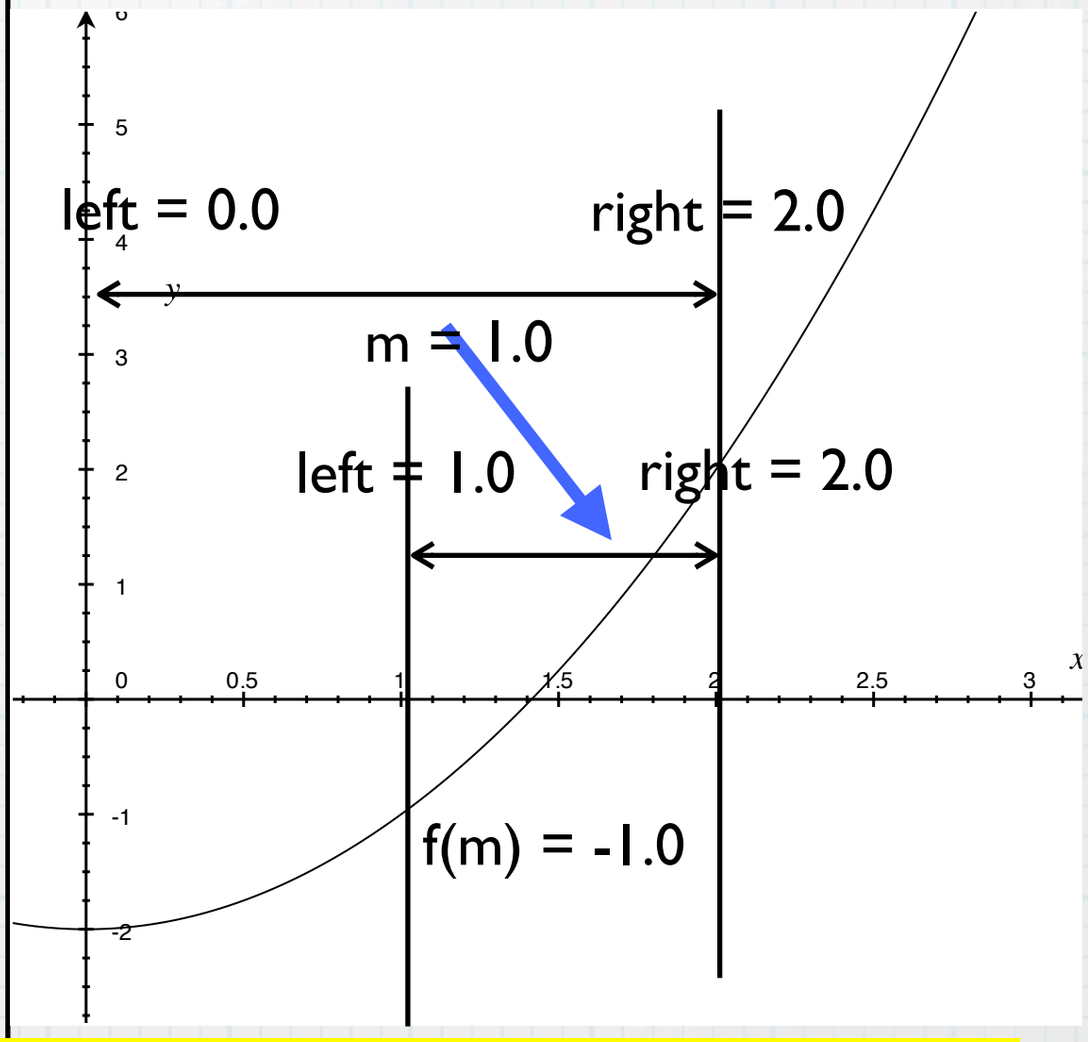
Schemeではfor文やwhile文がなくてもプログラムの繰返しを書くことができる（C言語でも同様の呼び出しによって書くことはできるが，スタックを消費するので現実的ではない）．

2分法で平方根を求める (1)

2分法とは，区間 [l，r] を半分に狭めることによって，方程式の根を求める方法である．JavaScriptでのプログラムは以下のようなになる．ここではwhile文を用いて，繰り返し計算が行われている．

```
function binary(){
    var EPS = 1.0e-6
    var left = 0.0
    var right = 2.0
    while (right - left > EPS){
        var m = (left + right) / 2
        var v = m * m - 2
        if (v < 0) left = m
        else right = m
    }
    puts(left + ", " + right)
}

binary()
```

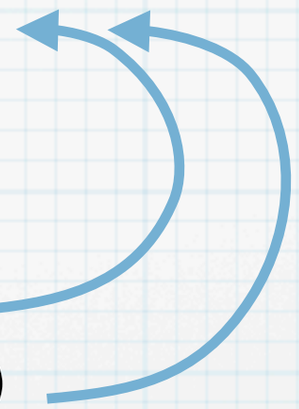



```
1.4142131805419922, 1.4142141342163086
```

2分法で平方根を求める (2)

繰り返しを行うのではなくて，binaryの中からbinaryを呼び出すことで繰り返しの動作を表現する.

```
(define (binary left right f eps)
  (define m (/ (+ left right) 2))
  (if (< (- right left) eps) left
      (if (< (f m) 0)
          (binary m right f eps)
          (binary left m f eps))))
```



```
(define (f x) (- (* x x) 2))
(display (binary 0.0 2.0 f 1.0e-6))
(newline)
```

```
OMacBook:yama997> ../kawa prog1.scm
1.4142131805419922
```

```
function binary(){
   var EPS = 1.0e-6
   var left = 0.0
   var right = 2.0
   while (right - left > EPS){
      var m = (left + right) / 2
      var v = m * m - 2
      if (v < 0) left = m
      else right = m
   }
   puts(left + ", " + right)
}

binary()
```

関数評価に関するルール

関数を評価する場合のルールについて以下にまとめる.

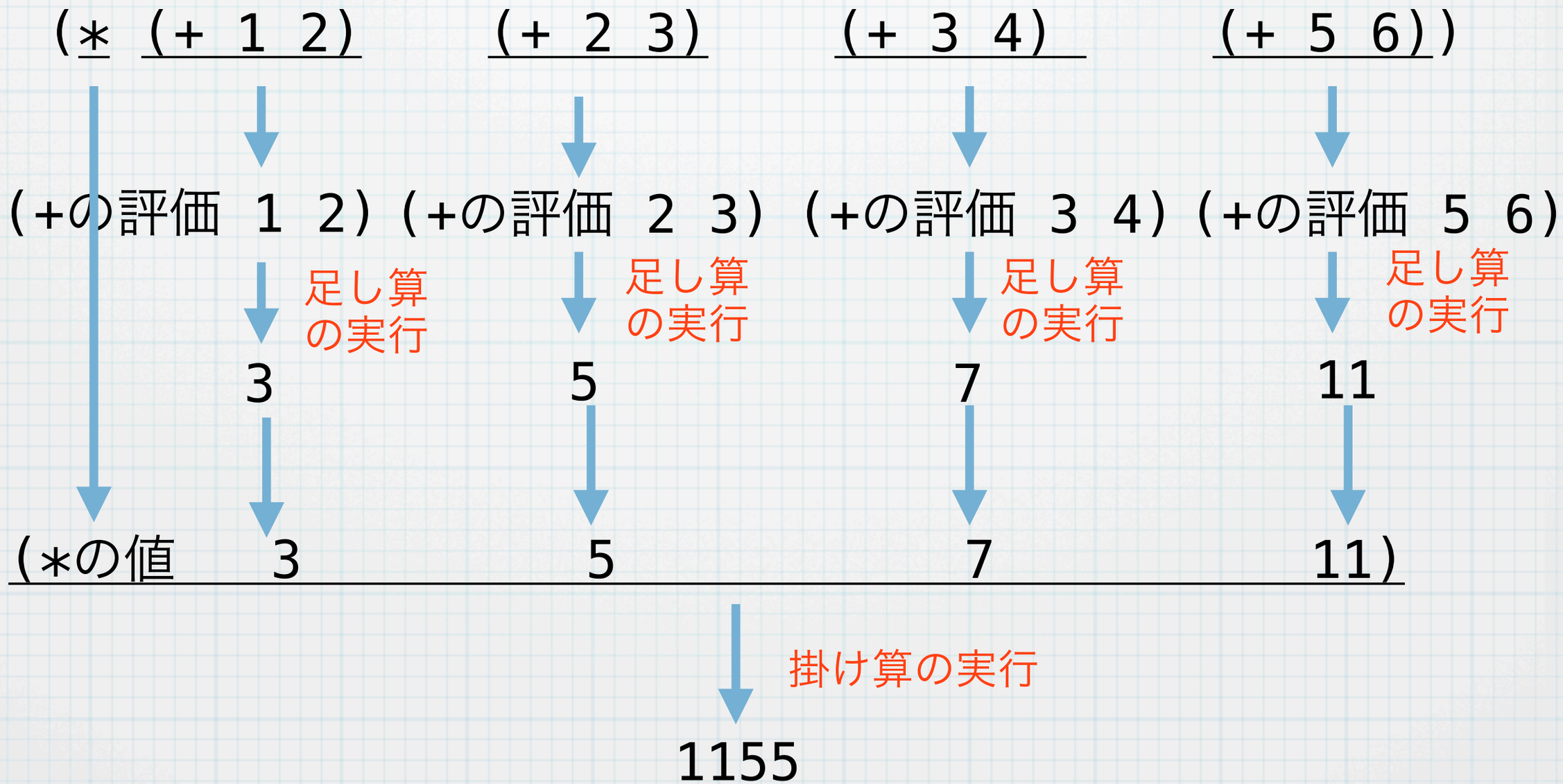
```
(func1 a1 a2 ... an)
```

1. S式のなかのそれぞれの部分式を評価する. 部分式が変数であれば, それが結び付けられている値を取り出し, S式であれば, それをさらに評価する. また, 数であればその数そのもの, 文字列であれば, その文字列とする.

2. 最初に置かれている値（func1に対応）を関数の実体だと考え, それに続く値（a1, a2,..., anに対応）を引数の値だと考えて, 関数を実行しその値をこの式の値とする.

関数評価の例（1）

単純な式の場合，それぞれのの記号に結び付けられている実体を呼び出して，関数評価が行われる．



if や defineは関数ではない

以前定義した if や define は評価の仕方が関数と異なる．すなわち，これらは関数ではない．これらは**特殊形式 (special form)** と呼ぶ．

```
(if (< a b)
    (+ a b)
    (- a b))
```

```
(define (foo x)
    (* x 2))
```

上の例の場合，(< a b)がtrue (#t) であれば(+ a b)を計算し，false (#f) であれば，(- a b)を計算するが，もし if が関数であれば，全体を計算する前に両方共計算されていることになる．以下のプログラムを実行すると foobar と出力される理由を考えよ．

```
(define (my-if cond expr1 expr2)
  (if cond expr1 expr2))

(my-if (= 10 2) (display "foo") (display "bar"))
(newline)
```

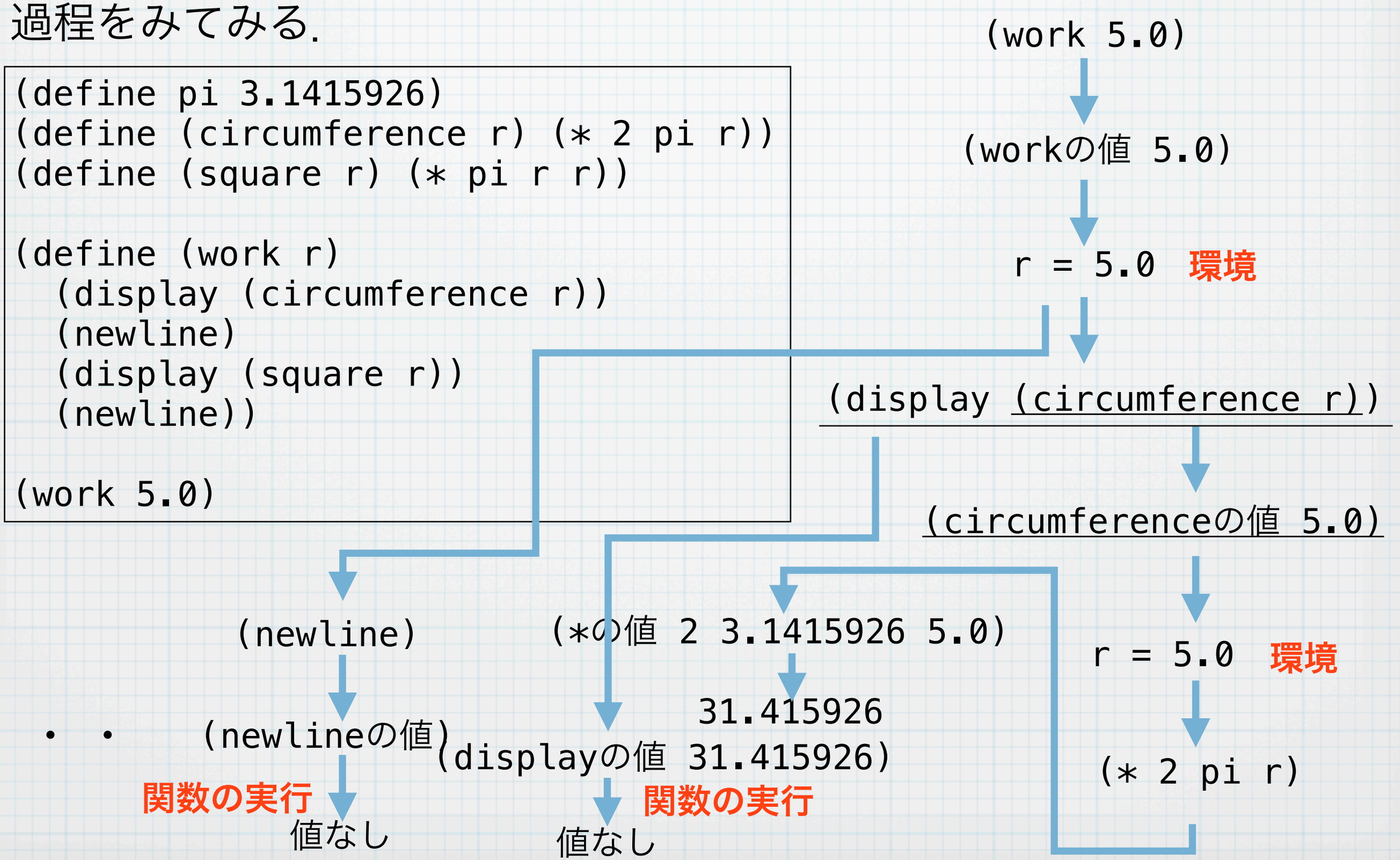
関数評価の例（２）

以下のようにいくつかの関数を定義する．関数workの式が評価される過程をみてみる．

```
(define pi 3.1415926)
(define (circumference r) (* 2 pi r))
(define (square r) (* pi r r))

(define (work r)
  (display (circumference r))
  (newline)
  (display (square r))
  (newline))

(work 5.0)
```



関数評価の例（3）

前述のbinary関数の場合，関数定義されているので，多少事情が異なる．

```
(define (binary left right f eps)
  (define m (/ (+ left right) 2))
  (if (< (- right left) eps) left
      (if (< (f m) 0)
          (binary m right f eps)
          (binary left m f eps))))
```

```
(define (f x) (- (* x x) 2))
```

(binary 0.0 2.0 f 1.0e-6)

(binaryの値 0.0 2.0 fの値 1.0e-6)

環境
left = 0.0, right = 2.0, f = fの値, eps = 1.0e-6

環境
m = 1.0, (f m) = (fの値, 1.0)=-1.0

(binary m right f eps)

(binaryの値 1.0 2.0 fの値 1.0e-6)

m = 1.0, (f m)

(fの値, 1.0)

環境 x = 1.0

(- (* x x) 2)

(-の値 (* x x)の評価 2)

(*の値 1.0 1.0) = 1.0

関数の実行

(-の値 1.0 2)

関数の実行

-1.0

今回のまとめ (1)

S式によるデータ（プログラム）の表現

```
(1 2 3 (5 4 3 . 8) 2)

(abc (1 2) . (def zef))
```

数の四則演算

```
(+ 1 2 3 4 5) = 1 + 2 + 3 + 4 + 5 = 15

(- 20 3 2 1) = 20 - 3 - 2 - 1 = 14

(* 2 3 4 5 6) = 2 * 3 * 4 * 5 * 6 = 720

(/ 2 3 4) = 2 / 3 / 4 = 1/6

(+ (* 2 3) (* 4 5)) = 2 * 3 + 4 * 5 = 26
```

Schemeでは出来る限り正確な演算を行う．多くの場合，有理数の割り算などでは分数をデータとして扱うことができる.

今回のまとめ (2)

変数の定義

```
(define a 20)

(define b “abcde”)
```

関数の定義

```
(define (sqr x) (* x x))

(define (関数名 引数1 引数2 ... 引数n) 式1 式2 ... 式m)
```

本体

データの表示

```
(display “abc”)

(display (+ 1 2 3))

(newline)
```

今回のまとめ (3)

定義されている変数の内容を変える

```
(set! a 34)

(set! b “meijo”)
```

成り立つ条件によって式の値を変える

```
(if (< a 0) (- a) a)

(if 条件式 式1 式2)

(when (< m 1) (set! m (+ 1 2)) (* m m))

(when 条件式 式1 式2 .. 式n)
```

whenは条件が成り立たないとき何も値を返さない.