コンパイラ理論 補足
Ruby 入門

櫻井彰人

Rubyプログラムの実行

- ruby とは Ruby プログラムを実行するコマンド
  - 指定されたファイルの中身をみて、それに従った動作をする

```
G:¥Ruby>ruby sample1.rb
こんにちは

G:¥Ruby>_
```

```
G:¥Ruby>ruby Dendai1.rb
こんにちは

G:¥Ruby>ruby Dendai2.rb
Dendai2.rb:1: Invalid char `¥201' in expression
Dendai2.rb:1: syntax error, unexpected $undefined, expecting $end
puts   ”こんにちは”
       ^

G:¥Ruby>
```

```
# coding: Windows-31J
# Dendai1.rb
puts   "こんにちは"
```

```
# coding: Windows-31J
# Dendai2.rb
puts   "こんにちは"
```

日本語全角文字を扱うためのおまじない
以下では省略する

```
G:¥Ruby>ruby Dendai4.rb
Dendai4.rb:2: unterminated string meets end of file

G:¥Ruby>
```

```
# coding: Windows-31J
# Dendai4.rb
puts   "こんにちは”
```

どこが間違いか？

```
G:¥Ruby>ruby Dendai5.rb
Dendai5.rb:2: Invalid char `¥201' in expression
Dendai5.rb:2: Invalid char `¥202' in expression
Dendai5.rb:2: Invalid char `¥261' in expression
Dendai5.rb:2: Invalid char `¥202' in expression
Dendai5.rb:2: Invalid char `¥361' in expression
Dendai5.rb:2: Invalid char `¥202' in expression
Dendai5.rb:2: Invalid char `¥311' in expression
Dendai5.rb:2: Invalid char `¥202' in expression
Dendai5.rb:2: Invalid char `¥277' in expression
Dendai5.rb:2: Invalid char `¥202' in expression
Dendai5.rb:2: Invalid char `¥315' in expression
Dendai5.rb:2: unterminated string meets end of file
Dendai5.rb:2: warning: parenthesize argument(s) for future version

G:¥Ruby>
```

```
# coding: Windows-31J
# Dendai5.rb
puts   ” こんにちは"
```

どこが間違っているのかな？

他の例題

```
# coding: Windows-31J
# sample12.rb
print  "春の"
print  "うららの"
puts  "隅田川"
```

```
G:¥Ruby>ruby sample12.rb
春のうららの隅田川

G:¥Ruby>
```

```
# coding: Windows-31J
# sample13.rb
print  "春の" + "うららの"
puts  "隅田川"
```

```
G:¥Ruby>ruby sample13.rb
春のうららの隅田川

G:¥Ruby>
```

```
# coding: Windows-31J
# sample14.rb
puts  "春の"
puts  "うららの"
puts  "隅田川"
```

```
G:¥Ruby>ruby sample14.rb
春の
うららの
隅田川

G:¥Ruby>
```

irb

- プログラムは、多数の行で構成されている。
- しかし、中には、実行したいことが、一行で書けてしまうこともある
- Ruby には、この「一行プログラム」の実行ができるツールが提供されている。
  - 実は、複数行に渡っても実行できる、優れもの
- それが、irb (interactive Ruby)

```
G:¥Ruby>irb
irb(main):001:0> print( 2+3+4+5+6+7+8+9 )
44=> nil
irb(main):002:0>
```

irb の実行例

- 電卓がわりにも使える

```
irb(main):001:0> 1+2+3+4
=> 10
irb(main):002:0> 2*3*4*5*6*7*8*9*10
=> 3628800
irb(main):003:0> x=2.0
=> 2.0
irb(main):004:0> Math.sqrt(x)
=> 1.4142135623731
irb(main):005:0> Math::PI
=> 3.14159265358979
irb(main):006:0> Math.sin(Math::PI/4)
=> 0.707106781186547
irb(main):007:0> Math.sqrt(2)/2
=> 0.707106781186548
irb(main):008:0>
```

http://www.ruby-lang.org/ja/man/?cmd=view;name=Math

定数と変数

数値には型がある

- さて、次のようになる理由を考えてみよう

```
irb(main):005:0> 3
=> 3
irb(main):006:0> 3.0
=> 3.0
irb(main):007:0> 3/2
=> 1
irb(main):008:0> 3.0/2
=> 1.5
irb(main):009:0> 3.0/2.0
=> 1.5
irb(main):010:0> 2/3
=> 0
irb(main):011:0> 2.0/3
=> 0.666666666666667
irb(main):012:0> 2/3.0
=> 0.666666666666667
irb(main):013:0>
```

ヒント:
小数点がある数と小数点がない数に違いがある

データには型がある 例

```
irb(main):001:0> 2+3
=> 5
irb(main):002:0> 2+3.0
=> 5.0
irb(main):003:0> 2.0+3
=> 5.0
irb(main):004:0>
```

文字列にも演算

```
irb(main):017:0> "abcd"*2
=> "abcdabcd"
irb(main):018:0> "abcd"+"efgh"
=> "abcdefgh"
irb(main):019:0> "Abc"*3
=> "AbcAbcAbc"
```

```
irb(main):020:0> "abcd"-"ab"
NoMethodError: undefined method `-' for "abcd":String
        from (irb):20
        from :0
irb(main):021:0> 2*"abcd"
TypeError: String can't be coerced into Fixnum
        from (irb):21:in `*'
        from (irb):21
        from :0
irb(main):022:0>
```

変わった演算

- 比較も演算。

```
irb(main):029:0> 2 == 2
=> true
irb(main):030:0> 2 < 3
=> true
irb(main):031:0> 2 >= 3
=> false
irb(main):032:0> 2 != 3
=> true
irb(main):033:0> "ab" == "ab"
=> true
irb(main):034:0> "ab" == "abc"
=> false
irb(main):035:0> "ab" < "abc"
=> true
irb(main):036:0>
```

```
irb(main):036:0> 2.1 < 2.2
=> true
irb(main):037:0> 2.1 > 2.2
=> false
irb(main):038:0> 2.0 == 2
=> true
irb(main):039:0> 2 < "2"
ArgumentError: comparison of Fixnum
 with String failed
        from (irb):39:in `<'
        from (irb):39
        from :0
irb(main):040:0>
```

データ型

- このように、施しうる演算・操作で、対象がどんなものかが決まるような場合に、それ（演算・操作の集合、または、その対象）を型という。
- Rubyにあるデータ型のうち、既出は:
  - 小数点のない数: 整数、固定小数点数
    - integer or fixed-point number
  - 小数点のある数: 浮動小数点数
    - floating-point number
  - 文字列

変数の型

- Ruby は動的型付けの言語であり、変数は静的な型を持たない。従って、次のようにしても何の問題もない

```
irb(main):045:0> x=1.0; print x.to_s, "  "; x=1; print x.to_s
1.0  1=> nil
```

少々不思議な結果

```
irb(main):001:0> 0.9
=> 0.9
irb(main):002:0> 0.99
=> 0.99
irb(main):003:0> 0.999
=> 0.999
irb(main):004:0> 0.9999
=> 0.9999
irb(main):005:0> 0.99999
=> 0.99999
irb(main):006:0> 0.999999
=> 0.999999
irb(main):007:0> 0.9999999
=> 0.9999999
irb(main):008:0> 0.99999999
=> 0.99999999
irb(main):009:0> 0.999999999
=> 0.999999999
```

```
irb(main):010:0> 0.9999999999
=> 0.9999999999
irb(main):011:0> 0.99999999999
=> 0.99999999999
irb(main):012:0> 0.999999999999
=> 0.999999999999
irb(main):013:0> 0.9999999999999
=> 0.9999999999999
irb(main):014:0> 0.99999999999999
=> 0.99999999999999
irb(main):015:0> 0.999999999999999
=> 0.999999999999999
irb(main):016:0> 0.9999999999999999
=> 1.0
irb(main):017:0> 0.99999999999999999
=> 1.0
```

少々不思議な比較結果

Ruby 1.9 では 6 が続く

```
irb(main):040:0> 2.0/3
=> 0.666666666666667
irb(main):041:0> 2.0/3 == 0.66666666666666
=> false
irb(main):042:0> 2.0/3 == 0.666666666666666666
=> true
irb(main):043:0>
```

有限桁数かつ四捨五入だから

閑話休題: irbでの編集

- irb で入力した文字を修正する方法
  - backspace: カーソル直左の一文字消去
  - ←→: カーソルの移動
    - Ctrl+b, Ctrl+f と同じ
  - ↓ ↑: 過去入力した行への切替
    - Ctrl+n, Ctrl+p と同じ
  - delete: カーソル位置の一文字を消去
  - Home, End: 入力行の先頭、末尾へ
    - Ctrl+a, Ctrl+e と同じ
  - Ctrl+k でカーソルから右側全部削除

変数の例

```
irb(main):056:0> x=25
=> 25
irb(main):057:0> print( "xの値は ",x,"  です" )
xの値は 25  です=> nil
irb(main):058:0> x=30
=> 30
irb(main):059:0> print( "xの値は ",x,"  に変わりました" )
xの値は 30  に変わりました=> nil
```

```
irb(main):060:0> x=20; print( "xの値は ",x,"  です" )
xの値は 20  です=> nil
irb(main):061:0> x=30; print( "xの値は ",x,"  に変わりました" )
xの値は 30  に変わりました=> nil
```

```
irb(main):062:0> x=20; print( "xの値は ",x,"  です" ); x=30;
print( "xの値は ",x," に変わりました" )
xの値は 20  ですxの値は 30  に変わりました=> nil
```

いろいろのデータを一つの変数に

この行は、前の行からの続きであることを表している。Ruby は行が終わるのを待っている

このセミコロンがあると、Ruby は、まだ行が続くと考える

```
irb(main):063:0> x=2C;
irb(main):064:C* print( "x is now ",x );
irb(main):065:0* x=3.14;
irb(main):066:0* print( " but x is now ",x );
irb(main):067:0* x="Hi!";
irb(main):068:0* print( " but now x is ",x )
x is now 20 but x is now 3.14 but now x is Hi!=> nil
```

ここにセミコロンはないし、前に書いてある式も一まとまりになっているので、Ruby は、これで終わり！と考える

整数型の可能性

- Ruby の整数型は、非常に大きな数を表現できる。

これは、後ほど

```
irb(main):079:0> def factorial(n) (1..n).inject{|a,b| a*b} end
=> nil
irb(main):080:0> factorial(10)
=> 3628800
irb(main):081:0> factorial(100)
=>
933262154439441526816992388562667004907159682643816214685929638952
175999932299156089414639761565182862536979208272237582511852109168
64000000000000000000000000
```

参考

```
irb(main):002:0> def factorial(n) if n<=1 then 1 else n*factorial(n-1) end end
=> nil
irb(main):003:0> factorial(10)
=> 3628800
```

```
irb(main):004:0> def factorial(n) r=1; (2..n).each{ |i| r*=i }; r end
=> nil
irb(main):005:0> factorial(10)
=> 3628800
```

```
irb(main):007:0> def factorial(n) r=1; for i in 2..n; r*=i end; r end
=> nil
irb(main):008:0> factorial(10)
=> 3628800
```

さらに参考

```
irb(main):002:0> def fct(n,f=1) if n<=1 then f else fct(n-1,n*f) end end
=> nil
irb(main):003:0> fct(10)
=> 3628800
```

```
irb(main):002:0> def factorial(n) (1..n).inject{|x,y| x*y} end
=> nil
irb(main):003:0> factorial(10)
=> 3628800
```

```
irb(main):002:0> def factorial(n) eval( [*(1..n)].join("*") ) end
=> nil
irb(main):003:0> factorial(10)
=> 3628800
```

浮動小数点型 Ruby の場合

- 10進数で約15桁の精度があります。
  （C や Java の double と同じです）

```
irb(main):009:0> printf("%.50f¥n", 1.0/3)
0.33333333333333331000000000000000000000000000000000
=> nil
```

Ruby 1.9では
0.33333333333333331482961625624739099293947219848633

浮動小数点型 Ruby の場合

- 指数も同じく

```
irb(main):012:0> 10.0**308
=> 1.0e+308
irb(main):013:0> 10.0**309
=> Infinity
```

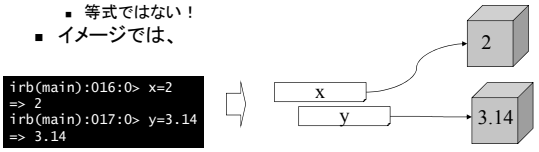
```
irb(main):014:0> 10.0**(-323)
=> 9.88131291682493e-324
irb(main):015:0> 10.0**(-324)
=> 0.0
```

Ruby 1.9 では
irb(main):014:0> 10.0**(-323)
=> 1.0e-323

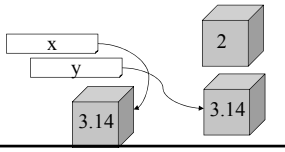
他の変数の値を代入(1)

- 変数に他の変数の値を代入する
  - 等式ではない！
- イメージでは、

```
irb(main):016:0> x=2
=> 2
irb(main):017:0> y=3.14
=> 3.14
```



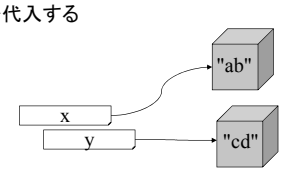

```
irb(main):018:0> x=y
=> 3.14
```



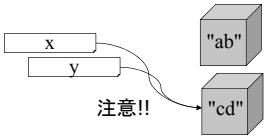
他の変数の値を代入(2)

- 変数に他の変数の値を代入する
  - 等式ではない！
- イメージでは、

```
irb(main):016:0> x="ab"
=> "ab"
irb(main):017:0> y="cd"
=> "cd"
```




```
irb(main):018:0> x=y
=> "cd"
```



式

- 式は**演算子**（オペレータ。演算内容を表すもの）と**オペランド**（演算の対象）の組み合わせ
- 例： 3+4
  - 「3」と「4」がオペランド、「+」が演算子

整数型の算術演算子

演算子	用途	例	演算結果
+	加算	3+2	5
-	減算	4-2	2
*	乗算	2*2	4
/	除算	4/2	2
%	剰余	5%2	1
**	べき	5**3	125

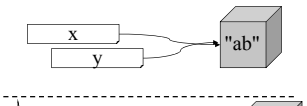
浮動小数点数型の算術演算子

演算子	用途	例	演算結果
+	加算	3.1+2.2	5.3
-	減算	4.2-2.1	2.1
*	乗算	2.1*2.1	4.41
/	除算	4.2/2.1	2.0
%	剰余	5.0%2.1	0.8
**	冪	2.1**0.5	1.44913

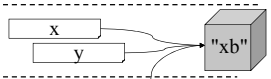
他の変数の値を代入(3)

- 変数に他の変数の値を代入する
  - 等式ではない！
- イメージでは、

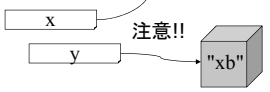
```
irb(main):016:0> x="ab"
=> "ab"
irb(main):017:0> y=x
=> "ab"
```




```
irb(main):018:0> x[0,1]="x"
=>"x"
```




```
irb(main):019:0> y="xb"
=> "xb"
```



論理式

- 論理値(真偽値)を計算する式を論理式と呼ぶことにします。
- 論理式の基本は、数式または文字列式(の値)と数式または文字列式(の値)とを比較演算子を用いて比較する式です。これを and/or/not で組み合わせます
- また、if 論理式 then 数式 else 数式 end とすると数式になります。

```
irb(main):003:0> x=-10
=> -10
irb(main):004:0> if x<0 then -x else x end
=> 10
```

比較演算子

演算子	用途	例	演算結果
==	等	3==2	false
>	大	4 > 2	true
<	小	4 < 2	false
>=	大or等	4>=2	true
<=	小or等	4<=2	false
!=	非等	3 != 2	true

論理演算子

演算子	用途	例	演算結果
!	否定	! 3==2	true
&&	かつ	2==2 && 4>2	true
||	または	2==3 || 4>2	true
not	否定	not 3==2	true
and	かつ	2==2 and 4>2	true
or	または	2==3 or 4>2	true

Ruby で括弧が省略できる場所

- いくつかある。下記は一例

```
irb(main):040:0> print "without () "; print( "or with ()" )
without () or with ()=> nil
irb(main):041:0> puts "without () "; puts( "or with ()" )
without ()
or with ()
=> nil
```

Ruby プログラム

- Ruby のプログラムは「式」の列である
- 例:
  - 0.3
  - 2 * 3 * 4
  - x = 5 * 6
- しかし、非常に困ったことに、ruby.exe は各式の値を計算するのだが、それを、我々にわかるように表示してはくれない
  - irb は、一行ずつ入力された式の値は表示してくれる。
- 不親切なようだが、必要に迫られてのこと。
  - 大きなプログラムでは、全ての式の値を表示されたら、ごみの山
- そこで、式の値を表示する特別な式が用意された。
  - それが、print や puts

演算子の優先順位

高 ↕ 低

( ): 括弧, ( ): 引数
+, -: 符号（符号同一、符号反転）
**:冪乗
*:乗算　/:除算　%:剰余
+:加算　-:減算
=:代入

- 同じレベルの演算子は左から順に計算する。従って
  - 2-4 * 3　は 2 - (4*3)，　3 / 4*6　は (3/4)*6．
    なお、a**-b は a**(-b)，a--+-b は a-(-(+(-b)))

文字列連結演算子

- 「+」は、文字列を連結する役割がある
  「＊」は、文字列を繰り返す役割がある
- 既にサンプルプログラムで使っている通り
  - **print("iは " + i.to_s + " です¥n")**
    - " "で囲まれた文字列または変数の文字列値を連結する
    - 変数を文字列とするには、n.to_s のように変数名の後ろに、「.to_s」をつける
  - **print("にわ" *4 + "とりがいる¥n")**

Ruby の工夫

- Ruby では文字列（これは定数です）中に、変数を書くことができる。

```
irb(main):009:0> n=123; print( "n の値は #{n} です¥n" )
n の値は 123 です
=> nil
```

- 上記のように、文字列中に #{ と } で挟んだ式を書けばよい

```
irb(main):001:0> n=123; print( "#{n} は #{if n%2==0 then "even" else "odd" end} です¥n" )
123 は odd です
=> nil
```

```
irb(main):004:0> n=123;
irb(main):005:0* print( "#{n} は #{if n%2==0
irb(main):006:0"  then "even" else "odd" end} です¥n" )
123 は odd です
=> nil
```

出力フォーマット

- より凝って出力したい場合には、
  - printf("%+d", 1)

  のように、printf を用います
- " … " の部分がフォーマット（書式）です。
- 詳細は、

  http://www.ruby-lang.org/ja/man/
  ⇒「目次」中の「付録」中の sprintfフォーマット

代入演算子

- 「a = a 演算子 b」を「**a 演算子= b**」と記述することができる
- これを代入演算子という
  - 注意　=と演算子の間にスペースはおけない

```
a=20; b=10
a += b        # a=a+bと同じ  aは30を保持
```

代入演算子の例

代入演算子	用途	例
+=	加算代入	a += b (a=a+b)
-=	減算代入	a -= b (a=a-b)
*=	乗算代入	a *= b (a=a*b)
/=	除算代入	a /= b (a=a/b)
%=	剰余代入	a %= b (a=a%b)
**=	冪乗代入	a **=b (a=a**b)

データ型の変換

- データの右に、「.」をおき、その右に、「to_i」「to_f」「to_s」を記述した場合、そのデータ(値)を、それぞれ、整数型、浮動小数点数型、文字列への変換を行うことを意味する。
- データの代わりに変数にしても同様

```
irb(main):025:0> print 2
2=> nil
irb(main):026:0> print 2.0
2.0=> nil
irb(main):027:0> print 2.to_s
2=> nil
irb(main):028:0> print 2.to_f
2.0=> nil
irb(main):029:0> print "2".to_f
2.0=> nil
irb(main):030:0> print "2".to_i
2=> nil
```

```
irb(main):031:0> print 2.0.to_i
2=> nil
irb(main):032:0> print "2".to_i.to_f.to_s
2.0=> nil
irb(main):033:0> p "2".to_i.to_f.to_s
"2.0"
=> nil
irb(main):034:0> p "2".to_i.to_f
2.0
=> nil
```

p は、出力するとき、文字列と数値を区別して出力してくれます

自動型変換

- 整数型と浮動小数点数型が混在しているときには、浮動小数点数型に変換される。
  - これは、演算ごとに行われる。演算は左から順番に行われるため、すべての整数型データが浮動小数点型に変換されるわけではない。
  - 浮動小数点数型から整数型への変換は明示的に行わなければならない。

```
irb(main):038:0> 3.0/3 + 2/3
=> 1.0
irb(main):039:0> (3.0+2)/3
=> 1.66666666666667
```

- 文字列との自動変換は行われない

```
irb(main):040:0> "2" + 3
TypeError: can't convert Fixnum into String
        from (irb):40:in `+'
        from (irb):40
        from :0
```

実行の制御

if then else end

- if … then … else … end という式がある

```
irb(main):070:0> x=3
=> 3
irb(main):071:0> y = if x==3 then 4 else 6 end
=> 4
irb(main):072:0> x=4
=> 4
irb(main):073:0> y = if x==3 then 4 else 6 end
=> 6
```

if式

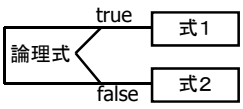
if 論理式 then 式1 else 式2 end

```
if 論理式 then
  式1  # 論理式がtrueのときの値
else
  式2  # 論理式がtrueのときの値
end
```

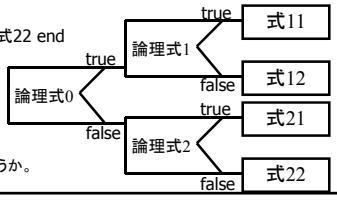
```
if 論理式
  式1  # 論理式がtrueのときの値
else
  式2  # 論理式がtrueのときの値
end
```

if 式

if 論理式 then 式1 else 式2 end




```
if 論理式0 then
  if 論理式1 then 式11 else 式12 end
else
  if 論理式2 then 式21 else 式22 end
 end
```



正式にはPAD図といいます
ここでは、分岐図とでもいいましょうか。

プログラム例(if式)

```
loop {
  print( "Enter your score: " )
  line = gets.chomp
    break if line==""
  score = line.to_f
  grade =
    if score >= 70 then
      if score >= 80 then "A" else "B" end
    else
      if score >= 60 then "C" else "D" end
    end
  print( "Your score #{score} corresponds to #{grade}¥n" )
}
```

```
G:¥Ruby>ruby -Ks sample0405.rb
Enter your score: 90
Your score 90.0 corresponds to A
Enter your score: 40
Your score 40.0 corresponds to D
Enter your score:
```

複数の式からなる式

- Ruby では、複数の式を並べたものも式となる。

```
loop{
  print( "Enter your score: " )
  line = gets.chomp
    break if line==""
  score = line.to_f
  grade =
    if score >= 70 then
      if score >= 80 then "A" else "B" end
    else
      if score >= 60 then "C" else "D" end
    end
  print( "Your score #{score} corresponds to #{grade}¥n" )
}
```

無限の繰り返し

- loop{ … }
- 上記「…」を無限に繰り返す。無限個のコピーを作ると考えてもよい。ただし、いきなり作るのではなく、必要があったら作るのですが。
- しかし、いずれにせよ、無限に作られるのは困る。
- 途中で止めなければ意味がない。
- 途中で止める道具(これも式だが、まったく式らしくない)が break です。

```
i = 0
loop{
  print( "やっほ～ " )
  if i>=10 then break end
  puts( "      Yee-ha! " )
  i = i+1
}
```

```
i = 0
loop{
  print( "やっほ～ " )
  break if i>=10
  puts( "      Yee-ha! " )
  i = i+1
}
```

これを if 修飾子という式 if 論理式 が一般形

回数がわかっている繰り返し

- 10.times

```
10.times {print "やっほ～ "; puts "      Yee-ha! " }
```

```
10.times {
  print( "やっほ～ " )
  puts( "      Yee-ha! " )
}
```

- さらに、こんなことができる

```
10.times {|i| print i; puts "番目"}
10.times {|i| puts( "#{i} 番目" ) }
10.times {|j| puts "*"*j}
```

```
0 番目
1 番目
2 番目
3 番目
4 番目
5 番目
6 番目
7 番目
8 番目
9 番目
```

```
*
**
***
****
*****
******
*******
********
*********
```

iやj=0,1,2,3,4,5,6,7,8,9 の順で、繰り返し、式の計算をする

回数が定まる繰り返し

- n.times

```
n=5
n.times {|k| print( " "*(10-k), "*"*k, "¥n" ) }
```

```
    *
   **
  ***
 ****
```

変数範囲が決まっている繰り返し

- each

```
(5..10).each {|i| print( i, if i%2==0 then " は偶数" else " は奇数" end,"¥n" ) }
```

```
(5..10).each {|i|
  print( i )
  if i%2==0 then
    print( " は偶数" )
  else
    print( " は奇数" )
  end
  print( "¥n" )
}
```

```
s = 5
e = 10
(s..e).each {|i|
  print( i )
  if i%2==0 then
    print( " は偶数" )
  else
    print( " は奇数" )
  end
  print( "¥n" )
}
```

```
5 は奇数
6 は偶数
7 は奇数
8 は偶数
9 は奇数
10 は偶数
```

```
11.times { |i|
  d = Math.sqrt( 100 - i*i ).to_i
  (1..d)     .each{ print( " " ) }
  ((d+1)..10).each{ print( "*" ) }
  print( "¥n" )
}
```

```
          *
          *
          *
          *
         **
         **
        ***
       ****
      ******
**********
```

```
11.times { |i|
  d = Math.sqrt( 400 - 4*i*i ).to_i
  (1..d)     .each{ print( " " ) }
  ((d+1)..20).each{ print( "*" ) }
  print( "¥n" )
}
```

```
                    *
                    *
                    *
                   **
                  ***
                 ****
                ******
               ********
              **********
             ************
            **************
           ****************
          ******************
**********
```

他のループ制御

- Ruby では他に、while, until, for などがある。
  - 拘る人へ: 正確には
    - Ruby の繰返制御構造は、while, until, for である
    - times, upto, downto, step は Integer のメソッド
    - each は Array 等の、Enumerable をインクルードするクラスのメソッド
- ループの中断には、break以外、next, redo, retry がある

downto, upto, and step

- 例で学ぼう

```
7.downto(3) { |i| print( i," " ) }
```
7 6 5 4 3

```
3.upto(7) { |i| print( i," " ) }
```
3 4 5 6 7

```
2.step(12,3) { |i| print( i," " ) }
```
2 5 8 11

```
12.step(2,-3) { |i| print( i," " ) }
```
12 9 6 3

while による繰り返し

```
第一回目のための準備
while 継続条件成立 do
  するべき作業
  次回への準備
end
```

```
i = 1
while i <= 5 do
  puts( i.to_s + "回目です。" )
  i += 1
end
```

```
1回目です。
2回目です。
3回目です。
4回目です。
5回目です。
```

while による繰り返し（続）

```
第一回目のための準備
while 継続条件成立 do
  するべき作業
  次回への準備
end
```

```
i = 1
while i <= 30 do
  puts( i.to_s + " です。" )
  i += i
end
```

```
1 です。
2 です。
4 です。
8 です。
16 です。
```

プログラム例(while文)

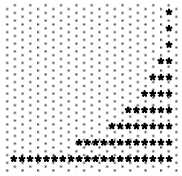
```
i = 1
while i <= 5 do
  puts i.to_s + "回目です。"
  i += 1
end
```

```
i = 1
while i <= 5 do
  puts "#{i}回目です。"
  i += 1
end
```

```
1回目です。
2回目です。
3回目です。
4回目です。
5回目です。
```

プログラム例(while文)(続)

```
11.times { |i|
  d = Math.sqrt( 400 - 4*i*i ).to_i
  (1..d)     .each{ print " " }
  ((d+1)..20).each{ print "*" }
  print "¥n"
}
```



```
11.times { |i|
  j=1
  while j*j+4*i*i<=400 do
    print " "
    j += 1
  end
  (j..20).each{ print "*" }
  print "¥n"
}
```

times と while の違い

- 同じである。ただし、コンセプトには結構な違いがある
  - times, each: 各回には、制御変数の値の違いしかない
  - while: 各回には、制御変数以外の変数で違いがある。または、制御変数を自分で作ってあげないといけない

times と while の違い (続)

- while を使うのは
  - ファイルの読み込み(後述)のように、読んでみないとわからない場合
    - 次のテーマ
  - 計算してみないとわからない場合
    - Collatz-角谷の予想
    - 素数を求める
    - 次回でてくる配列を用いる計算の中にある
  - 前の行為の結果が、影響を及ぼす場合

例: times と while の違い

- わざわざ作った例ですが

```
s = 0.0
n =10
n.times{ |i|
  r = rand()
  s  += r
}
a  = s/n
puts( "平均= #{a}" )
```

"回数" が予めわかっている

```
s = 0.0
n = 0
while s<10.0 do
  r = rand()
  s  += r
  n  += 1
end
puts( "#{n} 回目で10を越えた" )
```

"停止条件" が予めわかっている

"継続条件" が予めわかっている

while による繰り返し(続々)

```
while 継続条件 do #継続条件が真の間、処理を繰り返す
                  #継続か否かの判定は、一番最初
  式1
  式2
   .
   .
end

# よくある使い方
第一回目のための準備
while 継続条件 do
  するべき作業
  次回への準備
end
```

一行読み込む

```
open( "HumptyDumpty.txt" ) { |f|
  ln = 1; l = f.gets          # 第一回目のための準備
  while !(l.nil?) do
    print( ln, "行目:", l )   # 作業
    ln += 1; l = f.gets       # 次回への準備
  end
}
```

```
1行目:Hompty Dumpty sat on a wall,
2行目:Humpty Dompty had a great fall;
3行目:All the king's horses and all the king's men
4行目:Couldn't put Humpty together again.
```

http://www.authorama.com/through-the-looking-glass-6.html

よくある簡略表現

- 次のような簡略化が可能となる工夫がされている

```
open( "HumptyDumpty.txt" ) { |f|
  ln = 1; l = f.gets
  while ! (l.nil?) do
    print( ln, "行目:", l )
    ln += 1; l = f.gets
  end
}
```

```
open("HumptyDumpty.txt") { |f|
  ln = 1                      # 第一回目のための準備
  while l = f.gets  do        # 条件判定と次への準備
    print( ln, "行目:", l )   # 肝心の作業
    ln += 1                   # 次回への準備
  end
}
```

```
open("HumptyDumpty.txt") { |f|
  ln = 0
  while ( ln +=1; l = f.gets)  do
    print( ln, "行目:", l )
  end
}
```

禁止です。
でも、どうして動くのだろうか?

break文

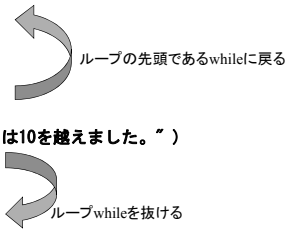
- 繰り返し（ループ）の中で使用
- そのbreakが所属するループを1つ抜ける

next

- nextはもっとも内側のループの次の繰り返しにジャンプします。
- while であれば、「継続条件」の判定の直前が再開場所です。

プログラム例(break,next)

```
n = 0
while n>=0 do
  if n<=10 then
    print( n, " ")
    n += 1
    next
  else
    print( "変数 n は10を越えました。" )
    break
  end
end
```




```
0 1 2 3 4 5 6 7 8 9 10 変数 n は10を越えました。
```

プログラム例(each と next)

```
(0..5).each { |i|
  if ( i==1 || i==4 ) then next end
  puts( "iは #{i}" )
}
```

```
iは 0
iは 2
iは 3
iは 5
```

素数を印字するプログラム

```
# 素数は、2～「自分-1」では割り切れない整数
# 2から10までの素数を印字しよう

(2..10).each{ |n|              # 素数候補
  p = 1                        # まだ素数候補
  (2..n-1).each{ |x|           # これで調べる
    (p=0; break) if n%x == 0 # 割り切れたら素数ではない
  }
  puts "#{n} は素数" if p==1  # 素数候補のままなら素数！
}
```

```
2 は素数
3 は素数
5 は素数
7 は素数
=> 2..10
```

しかし、効率が悪い。余分な割り算をたくさん行っている。
エラトステネスの篩にしたい。
篩はどうすれば表現できるか？
　配列で表現します。次回の話です。

Collatz-角谷の予想

- 自然数nを選び，
  - 奇数ならば，3倍して1をたす。
  - 偶数ならば，2で割る。

これを繰り返すと，どんなnを選んでも，いつかは，1になる

```
(3..20).each{ |i|
  while i>1 do
    print( i,", " )
# ここで適切に i を変更する
  end
  puts( i )
}
```

```
3, 10, 5, 16, 8, 4, 2, 1
4, 2, 1
5, 16, 8, 4, 2, 1
6, 3, 10, 5, 16, 8, 4, 2, 1
7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
8, 4, 2, 1
9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
10, 5, 16, 8, 4, 2, 1
11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
```

配列

配列の例

定数:
```
["Perl", "Python", "Ruby", "Scheme"]
```
変数への代入:
```
names = ["Perl", "Python", "Ruby", "Scheme"]
```
印字: (print は目的(?)にあいません)
```
p ["Perl", "Python", "Ruby", "Scheme"]
```

```
irb(main):016:0> p ["Perl", "Python", "Ruby", "Scheme"]
["Perl", "Python", "Ruby", "Scheme"]
=> nil
irb(main):017:0> puts ["Perl", "Python", "Ruby", "Scheme"]
Perl
Python
Ruby
Scheme
=> nil
irb(main):018:0> print ["Perl", "Python", "Ruby", "Scheme"]
PerlPythonRubyScheme=> nil
irb(main):019:0>
```

要素の取り出し

```
["Perl", "Python", "Ruby", "Scheme"].each { | lang |
   print( "I like ", lang, "¥n" )
}
```

```
irb(main):028:0> ["Perl", "Python", "Ruby", "Scheme"].each { | lang |
irb(main):029:1*   print( "I like ", lang, "¥n" )
irb(main):030:1> }
I like Perl
I like Python
I like Ruby
I like Scheme
=> ["Perl", "Python", "Ruby", "Scheme"]
```

```
names=["Perl", "Python", "Ruby", "Scheme"]
names.each { | lang | print( "I like #{lang}¥n" ) }
```

要素の取り出し(続)

```
names=["Perl", "Python", "Ruby", "Scheme"]
4.times { | i | print( "#{i} 番目は #{ names[i] }¥n" ) }
```

```
irb(main):033:0> names=["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):034:0> 4.times { | i | print( "#{i} 番目は #{names[i]}¥n" ) }
0 番目は Perl
1 番目は Python
2 番目は Ruby
3 番目は Scheme
=> 4
irb(main):035:0>
```

要素の取り出し(続々)

```
names=["Perl", "Python", "Ruby", "Scheme"]
names.length.times { | i | print( "#{i} 番目は #{ names[i] }¥n" ) }
```

```
irb(main):035:0> names=["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):036:0> names.length.times { | i | print( "#{i} 番目は #{ names[i] }¥n"
) }
0 番目は Perl
1 番目は Python
2 番目は Ruby
3 番目は Scheme
=> 4
```

要素への代入

```
names=["Perl", "Python", "Ruby", "Scheme"]
names[0] = "Ada"
names.length.times { | i | print( "#{i} 番目は #{ names[i] }¥n" ) }
```

```
irb(main):059:0> names=["Perl", "Python", "Ruby", "Scheme"]
=> ["Perl", "Python", "Ruby", "Scheme"]
irb(main):060:0> names[0] = "Ada"
=> "Ada"
irb(main):061:0> names.length.times { | i | print( "#{i} 番目は #{ names[i] }¥n"
) }
0 番目は Ada
1 番目は Python
2 番目は Ruby
3 番目は Scheme
=> 4
```

要素への代入(エラー)

```
irb(main):062:0> primes[0]=2
NameError: undefined local variable or method `primes' for main:Object
        from (irb):62
        from :0
irb(main):063:0>
```

Ruby では(Rubyに限らずどの言語でも)、未定義の変数が使われるとエラー
　Ruby では「使う」以外に現れると、宣言と考える
Ruby では、左辺に現れる以外は、「使う」ことに相当
　従って、新しい名前を左辺に書くと、普通は、宣言になる。
　　(だから問題は発生しない)
　しかし、配列の要素として現れる(names[2])と「使う」ことになってしまう
　　(だからエラー)

配列の宣言(?)

```
names=["Perl", "Python", "Ruby", "Scheme"]
```

```
abc = Array.new(5)
```

```
irb(main):073:0> abc = Array.new( 5 )
=> [nil, nil, nil, nil, nil]
irb(main):074:0>
```

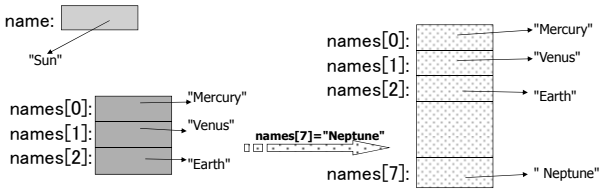
```
irb(main):075:0> abc[3]=333
=> 333
irb(main):076:0> abc
=> [nil, nil, nil, 333, nil]
irb(main):077:0>
```

Ruby の配列は柔軟

- すでに存在する要素に代入できる
  - これは当たり前
- まだ「ない要素」に代入すると、配列を拡張!して(当該要素を作って)くれる

```
irb(main):006:0> abc = ["a","b","c"]
=> ["a", "b", "c"]
irb(main):007:0> abc[3] = "d"
=> "d"
irb(main):008:0> abc
=> ["a", "b", "c", "d"]
irb(main):009:0> abc[10] = "k"
=> "k"
irb(main):010:0> abc
=> ["a", "b", "c", "d", nil, nil, nil, nil, nil, nil, "k"]
irb(main):011:0>
```

変数と配列の裏事情



配列の大きさが変わると、Ruby は配列を作り直している。
古い場所はそのまま捨て置かれる。
すなわち、無駄が発生している。ユーザには見えないが。

配列要素には何が代入できるか

- 変数に代入できるものなら何でも代入できる
- 整数、浮動小数点数、文字列、配列!
- しかも、混在!できる
  - CやJavaでは「混在」はできない

```
irb(main):011:0> abc = ["a","b","c"]
=> ["a", "b", "c"]
irb(main):012:0> abc[1] = 111
=> 111
irb(main):013:0> abc
=> ["a", 111, "c"]
irb(main):014:0> abc[3] = 3.33
=> 3.33
irb(main):015:0> abc
=> ["a", 111, "c", 3.33]
irb(main):016:0> abc[4] = [4,5,6]
=> [4, 5, 6]
irb(main):017:0> abc
=> ["a", 111, "c", 3.33, [4, 5, 6]]
```

要素に配列を代入したらどうなる?

- 予想通り

```
irb(main):017:0> abc
=> ["a", 111, "c", 3.33, [4, 5, 6]]
irb(main):018:0> abc[4][1]
=> 5
irb(main):019:0> abc[4][1] = 5.55
=> 5.55
irb(main):020:0> abc
=> ["a", 111, "c", 3.33, [4, 5.55, 6]]
irb(main):021:0> abc[4][5] = 8.88
=> 8.88
irb(main):022:0> abc
=> ["a", 111, "c", 3.33, [4, 5.55, 6, nil, nil, 8.88]]
```

関数（正確にはメソッド）

関数の定義

- 例えば、2数の最大値を値とする関数 max(x,y) を定義してみる

Sample0801

```
def max( x, y)
  if  x >= y then
    return x
  else
    return y
  end
end

i = 3; j = 5
x = max( i, j )
puts( "#{x} is the maximum of #{i} and #{j}" )
```

```
5 is the maximum of 3 and 5
```

関数の定義

- 3数の最大値を値とする関数 max(x,y,z)

Sample0802

```
def max( x, y, z )
  if ( x >= y ) then
    if ( y >= z ) then
      return x
    else
      if ( x >= z ) then
        return x
      else
        return z
      end
    end
  else
    if ( x >= z ) then
      return y
    else
      if ( y >= z ) then
        return y
      else
        return z
      end
    end
  end
end
```

```
i = 3; j = 5; k = 7
x = max( i, j, k)
puts( "#{x} is the maximum of #{i}, #{j}, and #{k}" )
```

```
7 is the maximum of 3, 5, and 7
```

配列を引数に

- 平均値を求める関数

Sample0803

```
def average( a )
  s=0.0
  a.each{ |x| s += x }
  return s/a.length
end
```

```
def average( a )
  s=0.0
  for i in (0..a.length-1) do
    s += a[i]
  end
  return s/a.length
end
```

```
x = [1,2,3,4,5,6,7,8,9,10]
print( "#{average( x )} is the average of: "); p( x )
```

```
5.5 is the average of: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

関数値のない関数

- 「関数値のない関数」は関数とは言い難いが。
  - サブルーチンとか副プログラムと言うのが正しい。
  - Ruby や Javaでは、どちらも、メソッド
- 仕事だけする。例えば、印字のみ。

Sample0810

```
def sayHello( n )
  n.times{ |i|
    i.times{ print( " " ) }
    puts( "Hello!" )
  }
end

sayHello( 4 )
```

```
Hello!
 Hello!
  Hello!
   Hello!
=> 4
```

関数値のない関数

- 配列に結果を戻すことはできる

Sample0811

```
def  getAverage( x,  y )
  for i in (0..x.length-1) do
    s = 0.0
    for j in (0..x[i].length-1) do
      s = s + x[i][j]
    end
    y[i] = s/x.length
  end
end
```

```
d = [ [1,2,3], [4,5,6], [7,8,9] ]
a = Array.new( d.length )

getAverage( d, a )

for i in (0..d.length-1) do
  print( "#{a[i]} is the average of: ")
  p( d[i] )
end
```

```
2.0 is the average of: [1, 2, 3]
5.0 is the average of: [4, 5, 6]
8.0 is the average of: [7, 8, 9]
=> 0..2
```

関数値のない関数

- でも、変数に値を戻すことはできない

Sample0812

```
def getAverage( x, y )
  s = 0.0
  for j in (0..x.length-1) do
    s += x[j]
  end
  y = s/x.length;
end
```

```
d = [1,2,3]
a = 0.0
getAverage( d, a )
print( "#{a} is??? the average of: "); p( d )
```

```
0.0 is??? the average of: [1, 2, 3]
```

関数値のある関数

- 配列を関数値（戻り値）とすることもできる

Sample0813

```
def getAverage( x )
  y = Array.new( x.length )
  for i in (0..x.length-1) do
    s = 0.0
    for j in (0..x[i].length-1) do
      s +=  x[i][j]
    end
    y[i] = s/x.length;
  end
  return y
end
```

```
def getAverage( x )
  y = Array.new( x.length )
  for i in (0..x.length-1) do
    s = 0.0
    x[i].each{ |a| s +=  a }
    y[i] = s/x.length;
  end
  return y
end
```

```
d = [ [1,2,3], [4,5,6], [7,8,9] ]

a = getAverage( d )

for i in (0..d.length-1) do
  print( "#{a[i]} is the average of: ")
  p( d[i] )
end
```

```
2.0 is the average of: [1, 2, 3]
5.0 is the average of: [4, 5, 6]
8.0 is the average of: [7, 8, 9]
=> 0..2
```