ソースコード解説

ソースコードはそのままCoursepowerにあげているのでメソッド毎にソースを解説する。基本的な形は千代先生から頂いたcalculater.rbと同じもの。

構文式

```
文列 = 文 (文)*
文 = print文 | while文 | if文 | function文 | call文 | for文 | ‘{‘ 文列 ‘}’ |
     全ての式（sentence()の項で説明）
print文 = ‘print’ andor(式5)
while文 = ‘while’ andor(式5) 文
if文 = ‘if’ andor(式5) 文 ( ‘else’ 文 )?
function文 = ‘function’ 変数 文
call文 = ‘call’ 変数
for文 = ‘for’ 文 ‘|’ 文 ‘|’ 文 ‘|’ 文
andor(式5) = overexp(式4) ( [ ‘&&’ | ‘||’ ] overexp(式4) )?
overexp(式4) = expression(式3) ( [ ‘<-’ | ‘=--‘ | ‘=!=‘ |
               ‘>’ | ‘>=‘ | ‘>’ | ‘>=‘ ] expression(式3) )?
expression(式3) = term(式2) ( [ ‘+’ | ‘-‘ ] term(式2) )?

term(式2) = factor(式1) ( [ ‘*’ | ‘/‘ | ‘%’ ] factor(式1) )?
factor(式1) = 数字 | 文字列 | 変数 | グローバル変数 | ‘(‘ andor(式5) ‘)’
文字列 = ‘’’ 空白文字以外 ‘’’
変数 = 大文字を含めたa-zで始まる英数字
グローバル変数 = ‘#’ 大文字を含めたa-zで始まる英数字
```

変数

`@@code`：受け付けたソースコードを文字列として格納するためのクラス変数。

`@@space_f`：関数を保持しておくためのクラス変数。

`@@global`：この処理系(Cherry)のグローバル変数を格納するためのクラス変数。

`@@keywords`：Cherryの予約語を保持しておくグローバル変数。

`@@operator`：Cherryの演算子を保持しておくグローバル変数。

`@@infunction`：function文(下で紹介)の中かどうかを判別するクラス変数。Boolean型。

`@space`：この処理系(Cherry)のインスタンス変数を格納するためのインスタンス変数。
この変数はクラスの中のinitialize()で宣言される。

get_token()

```
def get_token()
  if @@code =~ /\A\s*(#{@@keywords.keys.map{|t|Regexp.escape(t)}.join('|')})/
    @@code = $'
    return @@keywords[$1]
  elsif @@code =~ /\A\s*([0-9.]+)/
    @@code = $'
    return $1.to_f
  elsif @@code =~ /\A\s*([A-Za-z]+\w*)\(\)/
    @@code = $'
    unget_token($1)
    return :call
  elsif @@code =~ /\A\s*(\#?[A-Za-z]+\w*(?:\[\w*\])?)/
    @@code = $'
    return $1.to_s
  elsif @@code =~ /\A\s*(\'\S*\')/
    @@code = $'
    return $1.to_s
  elsif @@code =~ /\A\s*\z/
    return nil
  end
  return :bad_token
end
```

get_token()は@@codeの先頭から正規表現に沿って切り出し、切り出したものを返す関数。切り出したものが予約語の場合は、予約語を@@keywordsから探しそれと一致したシンボルを返す。数字の場合は数字をそのまま返す。関数呼び出し（関数名＋()）の場合は、末尾の「()」を取り除いたものを@@codeに戻し、:callを返す。変数（またはグローバル変数）の場合はそれをそのまま返す。このとき、配列やハッシュのように[英数字*]が最後についていても許容する。（しかし、内部では他の変数同様の処理となる）その文字列が「''」で囲まれている場合は、それを変数ではなく、ただの文字列と解釈する。空白文字だけの場合はnilを返し、それ以外は:bad_tokenを返す。

unget_token(token)

```
def unget_token(token)
  if token.is_a? Numeric
    @@code = token.to_s + @@code
  elsif token.is_a?(String)
    @@code = token + ' ' + @@code
  else
    @@code = @@keywords.key(token) ? @@keywords.key(token) + @@code : @@code
  end
end
```

unget_token()は、引数tokenを@@codeの先頭に戻すもの。tokenが数字の場合はそのまま戻し、文字列の場合空白を後ろに追加して戻す。（変数が結合してしまうことを防ぐ）それ以外の場合は、予約語で変換したものであれば変換前に戻し@@codeに戻す。

sentences()

```
def sentences()
  unless s = sentence()
    raise Exception, "can't find any sentences“
  end
  result = [:block, s]
  while s = sentence()
    result << s
  end
  return result
end
```

sentences()は、構文式での文列にあたる。文を順番に読み込んでいき、最後にその値を返す。一つも文が無かった場合は例外(can’t find any sentences)を発生させる。

sentence()

```
def sentence()
  token = get_token()
  case token
  when :lbra
    result = sentences()
  when :rbra
    return result
  when :print
    result = [:print, andor()]
    p result
  when :while
    result = [:while, andor(), sentence()]
  when :if
    if1 = andor()
    if get_token() == :then then
      if2 = sentence()
    else
      raise Exception, "then is missed"
    end
    if (temp = get_token()) == :else then
      if3 = sentence()
      result = [:if, if1, if2, if3]
    else
      unget_token(temp)
      result = [:if, if1, if2, nil]
    end
  when :function
    result = [:function, get_token(), sentence()]
    unless result[1].is_a? String
      raise Exception, "should use string as name of function"
    end
  when :call
    result = [:call, get_token()]
    unless result[1].is_a? String
      raise Exception, "should use string as name of function"
    end
  when :for
    temp1 = sentence()
    if get_token == :pipe then
      temp2 = sentence()
    else
      raise Exception, "missed pipe_1"
    end
    if get_token == :pipe then
      temp3 = sentence()
    else
      raise Exception, "missed pipe_2"
    end
    temp4 = sentence()
    result = [:for, temp1, temp2, temp3, temp4]
  when :bad_token
    raise Exception, "requested token is missed"
  end
  if token.is_a?(String)
    temp = token
```

```
      temp2 = get_token()
      if @@operator.include?(temp2) then
        unget_token(temp2)
        unget_token(temp)
        result = andor()
      else
        raise Exception, 'wrong sentense'
      end
    end
    return result
  end
```

sentence()は構文式の文にあたる。最初にget_token()を呼び出し、その返り値によって異なる動作をする。

・:lbra（{）のときは、新しく文列のブロックを作成する。

・:rbra（}）のときは、:lbraから始まったブロックを閉じて返す。

・:printのときは、[:print, andor(式5)]を返す。

・:whileのときは、[:while, andor(式5), 文]を返す。

・:ifのときは、[:while, 文, 文, 文]を返す。しかし、最初の文の後にthenが無かった場合は例外(then is missed)が発生する。ifの分岐が無かった場合は最後の文はnilとなる。分岐があるかどうかは2つ目の文の後のelseで判定する。

・:functionのときは、[:while, get_token(変数), 文]を返す。変数がString型のものでなかった場合は例外（should use string as name of function）が発生する。

・:callのときは、[:call, get_token(変数)]を返す。こちらも同様に、変数がString型のものでなかった場合は例外（should use string as name of function）が発生する。

・:forのときは、[:for, 文, 文, 文, 文]を返す。2つ目から4つ目の文の間に「|」が無かった場合はそれぞれ例外（missed pipe_1）（missed pipe_2）が発生する。

・:bad_tokenのときは、例外（requested token is missed）が発生する。

・返り値がString型の場合は、get_token()でもう一つ取得し、その値によってまた動作が変わる。下記のシンボルだった場合は、取得した２つの返り値を戻しandor(式5)を返す。それ以外の場合は例外（wrong　sentenes）が発生する。この記述によって、これらの演算子全てを文として扱うことが出来た。文として扱うことが出来るため構文式に書くべきか悩んだが、文としても式としても扱えるものをどのように記述すればいいのか分からなかったため、今回は記述しなかった。具体的には次のシンボルの場合。

```
:equal, :not_equal, :greater, :greater_than, :less, :less_equal,:and, :or, :assignment
```

andor()など

```
def andor()
  result = overexp()
  while true
    token = get_token()
    unless token == :and or token == :or
      unget_token(token)
      break
    end
    result = [token, result, overexp()]
  end
  return result
end

def overexp()
def expression()
def term()
def factor()
```

これらの関数はほとんど同じ動きをするのでまとめる。これらのメソッドでは、そのメソッドで扱うシンボルが来なかった場合は受け取った値をそのまま返し、来た場合はそのメソッドで配列の形に処理して返す。これらのメソッドを分けることで、演算子の優先度を付けることが出来る。factor()では、入力を受け付けるトークン:inputの時は:inputだけの配列を返す。String型の場合はそのままの値を返す。具体的な優先度は以下の通り。

優先度	
1	(,)
2	*, /, %
3	+, -
4	<- (代入), =-= (==), =!= (!=), >, >=, <, <=
5	&&, ll

構文解析は以上。

eval(exp) 演算子パート

```
def eval(exp)
  if exp.instance_of?(Array)
    case exp[0]
    when :add
      return eval(exp[1]) + eval(exp[2])
    when :sub
      return eval(exp[1]) - eval(exp[2])
    when :mul
      return eval(exp[1]) * eval(exp[2])
    when :div
      return eval(exp[1]) / eval(exp[2])
    when :mod
      return eval(exp[1]) % eval(exp[2])
    when :equal
      return eval(exp[1]) == eval(exp[2]) ? 1 : 0
    when :not_equal
      return eval(exp[1]) != eval(exp[2]) ? 1 : 0
    when :greater
      return eval(exp[1]) > eval(exp[2]) ? 1 : 0
    when :greater_equal
      return eval(exp[1]) >= eval(exp[2]) ? 1 : 0
    when :less
      return eval(exp[1]) < eval(exp[2]) ? 1 : 0
    when :less_equal
      return eval(exp[1]) <= eval(exp[2]) ? 1 : 0
    when :and
      return (eval(exp[1]) != 0 && eval(exp[2]) != 0) ? 1 : 0
    when :or
      return (eval(exp[1]) != 0 || eval(exp[2]) != 0) ? 1 : 0
```

eval()は意味解析にあたる。演算子の部分だけを抜粋した。このメソッドは引数によって異なる動作をする。ここでは基本的な処理は説明不要だと思われるので省略。追加した演算子は、真ならば１を、偽なら０を返す。

eval(exp) シンボル

```
      when :print
        puts eval(exp[1]) =~ /\'(\S*)\'/ ? $1 : eval(exp[1])
      when :assignment
        if(exp[1] =~ /\#([A-Za-z]+\w*(?:\[\w\])?)/) then
          @@global[$1] = eval(exp[2])
        else
          @space[exp[1]] = eval(exp[2])
        end
      when :if
        if eval(exp[1]) != 0 then
          return eval(exp[2])
        elsif exp[3] != nil
          return eval(exp[3])
        end
      when :block
        1.upto(exp.length) do |e|
          eval(exp[e])
        end
      when :while
        while eval(exp[1]) != 0
          eval(exp[2])
        end
      when :for
        eval(exp[1])
        while  eval(exp[2]) != 0
          eval(exp[4])
          eval(exp[3])
        end
      when :function
        @@infunction = true
        @@space_f[exp[1]] = exp[2]
        @@infunction = false
      when :call
        func = Cherry.new
        func.eval(@@space_f[exp[1]])
      when :input
        temp = STDIN.gets.chomp
        if temp =~ /\d+/ then
          return temp.to_f
        elsif temp.is_a?(String) then
          return "\'#{temp}\'"
        else
          raise Exception, 'please type Number or String'
        end
      end
```

○評価式以外の場合の動作を説明する。
・:print：文字列（'空白文字以外')の場合は、両端の「'」を取り除いて出力。それ以外の場合はこのメソッドに値を投げ、その返り値を出力。
・:assignment：受け取った変数の前に「#」がついていれば、その変数をグローバル変数として扱い、@@globalへ値を格納する。ついていない場合はインスタンス変数@spaceへ値を格納する。
・:if：最初のandor(式5)を評価し、それが真ならば2つ目の文を、偽なら3つ目の文をeval()に投げ、その返り値を返す。
・:block：前から順番に文をeval()へ投げる。
・:while：1つ目のandor(式5)が真である限り、2つ目の引数である文をこのメソッドに投げる。
・:for：1つ目の文を実行し（代入を想定）、2つ目の文が真である限り、3つ目の文をこeval()に投げる。3つ目の文を投げた直後に、4つ目の文を実行する（インクリメントを想定）。
・:function：最初に関数内であることを示すグローバル変数@@infunctionをtrueに変え、関数を格納するクラス変数@@space_fへ引数の関数（文）を代入する。その後@@infunctionをfalseにする。
・:call：関数を呼び出す。その際に新しくCherryクラスのインスタンスを生成し、そのインスタンスでeval()を呼び出す。これによってプログラム内での変数の競合を防ぐ。
・:input：標準入力を受け付ける。入力された値がString型だった場合はその両端に「'」を付けて値を返す。数字だった場合はそのまま返す。それ以外の場合は例外（please type Number or String）が発生する。

eval(exp) String

```
else
     if exp.is_a?(String)
       if exp =~ /\A\s*\'(\S*)\'/ then
         return $1
       elsif(exp =~ /\#([A-Za-z]+\w*(?:\[\w\])?)/) then
         if @@global.key?($1)
           return @@global[$1]
         elsif @@infunction == false then
           raise Exception, "not assigned(global)"
         end
       else
         if @space.key?(exp) then
           return @space[exp]
         elsif @@infunction == false then
           raise Exception, "not assigned(instance)"
         end
       end
     else
       return exp
     end
   end
 end
```

○引数expがString型の場合は次のような動作をする。
・文字列（'空白文字以外'）の場合はその値を返す。
・変数で且つ「#」がついていた場合は@@globalから対応する値を取り出し返す。なかった場合は例外（not assigned(global)）が発生する。その際に、このメソッドがfunction文内であるかどうかを@@functionで判定し、その場合は関数定義内なので例外を発生させない。
・変数で「#」がついていなかった場合は@spaceから対応する値を取り出して返す。なかった場合は例外（not assigned(instance)）が発生する。こちらも同様に、このメソッドがfunction文内であるかどうかを@@functionで判定し、その場合は関数定義内なので例外を発生させない。
○引数が数字の場合はそのままの値を返す。

initialize()など

```
  def initialize()
    @space = {}
  end

  def start()
    begin
      file = File.open(ARGV[0])
      @@code << file.read
      file.close
    rescue
      puts $!
    end
    s = sentences()
    eval(s)
  end
end

cherry = Cherry.new
cherry.start()
```

initialize()で@spaceを宣言している。これは関数毎にインスタンスを生成することに対応するため。その為プログラムでは最初の呼び出しの際にstart()メソッドを持たせ、そこでファイル読み込みを行っている。

感想

配列とハッシュを見せかけでなく、内部構造的にも対応させようと思ったが、一つのハッシュ変数に全ての値を格納しているため、そのハッシュとして格納された変数に対してさらにハッシュを繋げるとそれを呼び出す際にバグが出た。闇が深そうだったので断念した。もう一つ、while文で重要なbreakを実装したかったが、実装方法が思いつかなかった。今思うと、while文が呼び出されたときに、全てのクラス変数とインスタンス変数を渡して新しくcherryのインスタンスを呼び出して実行し、breakの時にはその呼び出したインスタンスの値を元のcherryのインスタンスに渡してexitなどで新しく生成したインスタンスを潰せば良かったのかなと思っている。これを思いついたのがレポート作成後だったので今回は実装できなかった。あとで個人的に弄ってみたい。

総合的にこの授業はとても楽しめました。15週間楽しかったです。ありがとうございました！