# コンパイラ理論 13
# Racc その8 (コード出力)

櫻井彰人

## 内容

- コード生成をしてみよう
- といっても、時間がないので、Ruby-likeなプログラムを出力してみよう
  - 何をやっているのかわからないといわれそうですが、ちょっと直すと C に似た言語で出力できますよ(Cは本質的には無理です。次頁)
  - それにもまして、練習になります

## CとRubyとの大きな違い

- Cは「変数に型がある、データには型がない(定数には型がある)」vs. Ruby は「変数には型がない、データには型がある」
- そのため、Cには型宣言が必要であるが、Rubyには型宣言が必要ない。
- RubyからCに変換するには、型宣言を変換プログラムが導入しないといけないが、これは、ほぼ、不可能

## 方針

- 構文木を作成してからコード生成を行う。
  - Ruby-like なコードを出すだけなら、構文木に変換してからコード出力をする必要はない。
- 今回は、練習のため。

## 再掲: 最初

```
program   : stmt_list
                {
                  result = RootNode.new( val[0] )
                }

stmt_list :
                {
                  result = []
                }
          | stmt_list stmt EOL
                {
                  result.push val[1]
                }
          | stmt_list EOL
```

ルートノード(木の根)を作る。
ノードの種類をいくつかつくり、
それごとに、メソッドを定義している。
それより細かい区分は、引数で行う

行の終わりを示す非終端記号

結果は配列に入れる

これは、前述(ではなく次頁)のように、修正する

## 再掲: if-then-else

```
if_stmt   : IF stmt THEN stmt_list else_stmt END
          {
           result = IfNode.new( @fname, val[0][0],
                                val[1], val[3], val[4] )
          }

else_stmt : ELSE stmt_list
          {
           result = val[3]
          }
            |
          {
           result = nil
          }
```

```
stmt_list :
          {
            result = []
          }
        | stmt
          {
            result=[val[0]]
          }
        | stmt_list EOL stmt
          {
            result.push val[2]
          }
        | stmt_list EOL
```

# 根っこ

```
class RootNode < Node

  def initialize(tree)
    super nil, nil
    @tree = tree
  end

  def evaluate
    exec_list Core.new, @tree
  end

end
```

- 親クラス
- "new" のときに、実行される
- 使い方から分かるように、子木の配列が渡される
- インスタンス変数に記憶する
- インタプリタ動作用。今回は、各子ノードで必ず定義する。コード生成が目的なら、ここをコード生成にする
- 親クラス Node のメソッドを使用する

追加

```
def mimic
    mimic_list Core.new, @tree
  end
```

# 大親 Node

追加

```
def mimic_list(intp, nodes)
    nodes.each {|i| i.mimic(intp); print("¥n") }
  end
```

```
class Node
  def initialize(fname, lineno)
    @filename = fname
    @lineno = lineno
  end

  attr_reader :filename
  attr_reader :lineno

  def exec_list(intp, nodes)
    v = nil
    nodes.each {|i| v = i.evaluate(intp) }
    v
  end

  def intp_error!(msg)
    raise IntpError, "in #{filename}:#{lineno}: #{msg}"
  end

  def inspect
    "#{self.class.name}/#{lineno}"
  end
end
```

- http://ruby.kyoto-wu.ac.jp/documents/ruby-man-ja/Ruby_FAQ.html の 5.6 を参照
- n = Node.new('intp', 1)
  puts n.filename, n.lineno
  などとできる
- 最後の値のみ残る
- 配列要素一個ずつ。
- 最後の値を返す

# 再掲: DefNode: 関数定義

```
class DefNode < Node

  def initialize(file, lineno, fname, func)
    super file, lineno
    @funcname = fname
    @funcobj = func
  end

  def evaluate(intp)
    intp.define_function @funcname, @funcobj
  end

end
```

Class Core

```
def define_function(fname, node)
    raise IntpError, "function #{fname} defined twice"
                              if @ftab.key?(fname)
    @ftab[fname] = node
  end
```

# DefNode追加

追加

```
def mimic(intp)
    print("def ", @funcname, "(" )
    print @funcobj.params[0]
    @funcobj.params[1..-1].map{ |i| print(", "); print(i)}
    print(") ", "¥n")
    @funcobj.body.map{ |i| i.mimic(intp); print("¥n")}
    print("end")
  end
```

# Function:関数定義

```
class Function < Node

  def initialize(file, lineno, params, body)
    super file, lineno
    @params = params
    @body = body
  end

  def call(intp, frame, args)
    unless args.size == @params.size
      raise IntpArgumentError,
        "wrong # of arg for #{frame.fname}()
             (#{args.size} for #{@params.size})"
    end
    args.each_with_index do |v,i|
      frame[@params[i]] = v
    end
    exec_list intp, @body
  end

end
```

追加

```
attr_reader :params
attr_reader :body
```

# 関数定義の仕方

```
defun : DEF IDENT param EOL stmt_list END
      {
        result = DefNode.new(@fname, val[0][0], val[1][1],
          Function.new(@fname, val[0][0], val[2], val[4]))
      }
```

- とりたい！
- IDENT
- param
- stmt_list

# 再掲: FuncallNode: 関数呼び出し

```
class FuncallNode < Node
  def initialize(file, lineno, func, args)
    super file, lineno
    @funcname = func
    @args = args
  end
  def evaluate(intp)
    args = @args.map {|i| i.evaluate intp }
    begin
      intp.call_intp_function_or(@funcname, args) {
        if args.empty? or not args[0].respond_to?(@funcname)
          intp.call_ruby_toplevel_or(@funcname, args) {
            intp_error! "undefined function #{@funcname.id2name}"
          }
        else
          recv = args.shift
          recv.send @funcname, *args
        end
      }
    rescue IntpArgumentError, ArgumentError
      intp_error! $!.message
    end
  end

end
```

# FuncallNode: 追加

追加

```
def mimic(intp)
    if ['+','-','*','/','^','=='].include?(@funcname)
              && @args.length==2 then
      print( "(" )
      @args[0].mimic(intp)
      print( @funcname)
      @args[1].mimic(intp)
      print( ")" )
    elsif '-@'==@funcname then
      print( "(" )
      print( "-" )
      @args[0].mimic(intp)
      print( ")" )
    else
      print(@funcname,"(")
      @args[0].mimic(intp)
      @args[1..-1].map{|i|  print(", "); i.mimic( intp ) }
      print(") ")
    end
end
```

なくても動くが、Rubyはエラーとする

# FuncallNode: 関数呼び出し2

Class Core

```
  def call_intp_function_or(fname, args)
      if func = @ftab[fname]
        frame = Frame.new(fname)
        @stack.push frame
        func.call self, frame, args
        @stack.pop
      else
        yield
      end
    end

    def call_ruby_toplevel_or(fname, args)
      if @obj.respond_to? fname, true
        @obj.send fname, *args
      else
        yield
      end
    end
```

# FuncallNode: 使い方

```
expr  : expr '+' expr
      {
          result = FuncallNode.new(@fname, val[0].lineno,
                                   '+', [val[0], val[2]])
      }
```

# 再掲: IfNode: if文

```
class IfNode < Node

    def initialize(fname, lineno, cond, tstmt, fstmt)
      super fname, lineno
      @condition = cond
      @tstmt = tstmt
      @fstmt = fstmt
    end

    def evaluate(intp)
      if @condition.evaluate(intp)
        exec_list intp, @tstmt
      else
        exec_list intp, @fstmt if @fstmt
      end
    end

end
```

# IfNode: 追加

追加

```
def mimic(intp)
    print( "if " )
    @condition.mimic(intp)
    print( " then", "¥n")
      @tstmt.map {|i| i.mimic( intp ); print("¥n") }
    if @fstmt!=nil then
      print( "else", "¥n")
      @fstmt.map {|i| i.mimic( intp ); print("¥n") }
    end
    print( "end")
end
```

# IfNode: 使い方

```
if_stmt : IF stmt THEN stmt_list else_stmt END
        {
         result = IfNode.new( @fname, val[0][0],
                              val[1], val[3], val[4] )
        }
```

stmt　stmt_list　else_stmt

# WhileNode: while

```
class WhileNode < Node

  def initialize(fname, lineno, cond, body)
    super fname, lineno
    @condition = cond
    @body = body
  end

  def evaluate(intp)
    while @condition.evaluate(intp)
      exec_list intp, @body
    end
  end
end
```

追加

```
def mimic(intp)
  print("while ")
  @condition.mimic(intp)
  print("do ", "¥n" )
    @body.map{ |i| i.mimic(intp); print("¥n") }
  print("end " )
end
```

# WhileNode: 使い方

とりたい！

```
while_stmt: WHILE stmt DO EOL stmt_list END
            {
              result = WhileNode.new(@fname, val[0][0],
                                     val[1], val[4])
            }
```

# AssignNode: 代入

```
class AssignNode < Node

  def initialize(fname, lineno, vname, val)
    super fname, lineno
    @vname = vname
    @val = val
  end

  def evaluate(intp)
    intp.frame[@vname] = @val.evaluate(intp)
  end
end
```

追加

```
def mimic(intp)
  print( @vname, " = " )
  @val.mimic(intp)
end
```

# AssignNode: 使い方

```
assign   : IDENT '=' expr
    {
      result = AssignNode.new(@fname,
                              val[0][0], val[0][1], val[2])
    }
```

# VarRefNode: 変数引用

```
class VarRefNode < Node

  def initialize(fname, lineno, vname)
    super fname, lineno
    @vname = vname
  end

  def evaluate(intp)
    if intp.frame.lvar?(@vname)
      intp.frame[@vname]
    else
      intp.call_function_or(@vname, []) {
        intp_error!
       "unknown method or local variable #{@vname.id2name}"
      }
    end
  end

end
```

追加

```
def mimic(intp)
  print( @vname )
end
```

```
class Frame

  def initialize(fname)
    @fname = fname
    @lvars = {}
  end

  attr :fname

  def lvar?(name)
    @lvars.key? name
  end
```

Rubyレファレンス「クラス/メソッドの定義」中「演算子式の定義」

使用例: frame[@params[i]] = v

```
  def [](key)
    @lvars[key]
  end

  def []=(key, val)
    @lvars[key] = val
  end

end
```

# VarRefNode: 使い方

```
realprim :
      IDENT
        {
          result = VarRefNode.new(@fname, val[0][0],
                                  val[0][1])
        }
      | NUMBER
        {
          result = LiteralNode.new(@fname, *val[0])
        }
      | STRING
        {
          result = StringNode.new(@fname, *val[0])
        }
```

# StringNode と LiteralNode

```
class StringNode < Node

  def initialize(fname, lineno, str)
    super fname, lineno
    @val = str
  end

  def evaluate(intp)
    @val.dup
  end

end
```

追加

```
def mimic(intp)
  print( '"',@val,'"' )
end
```

```
class LiteralNode < Node

  def initialize(fname, lineno, val)
    super fname, lineno
    @val = val
  end

  def evaluate(intp)
    @val
  end

end
```

追加

```
def mimic(intp)
  print(@val)
end
```

# StringNode/LiteralNode : 使い方

```
realprim :
      IDENT
        {
          result = VarRefNode.new(@fname, val[0][0],
                                  val[0][1])
        }
    | NUMBER
        {
          result = LiteralNode.new(@fname, *val[0])
        }
    | STRING
        {
          result = StringNode.new(@fname, *val[0])
        }
```

# 最後の部分

```
begin
  tree = nil
  if ARGV.length>=1 then
    fname = ARGV[0]
  else
    fname = 'src.intp'
  end
  File.open(fname) {|f|
    tree = Intp::Parser.new.parse(f, fname)
  }
  tree.mimic
rescue Racc::ParseError, Intp::IntpError, Errno::ENOENT
  raise ####
  $stderr.puts "#{File.basename $0}: #{$!}"
  exit 1
end
```

もとは tree.evaluate であった

# 簡単なテスト

```
def test(x)
  print(x,x*2)
  y=x*x
  print( y )
end

a=1
b=-3-a*2
d= "abcd"
if a==1 then c=1 else c=2 end
print( c )
j=3
i=1
while i==1 do
 print( j )
 j=j-1
 if j==1 then i=0 end
end

test(2)
```

→

```
def test(x)
print(x, (x*2))
y = (x*x)
print(y)
end
a = 1
b = ((-3)-(a*2))
d = "abcd"
if (a==1) then
c = 1
else
c = 2
end
print(c)
j = 3
i = 1
while (i==1)do
print(j)
j = (j-1)
if (j==1) then
i = 0
end
end
test(2)
```

# プログラムの実行

```
sakurai@p04 /cygdrive/e/main/
$ racc intM.y -o intM.rb
1 shift/reduce conflicts

sakurai@p04 /cygdrive/e/main/
$ ruby intM.rb testM.txt  > t.txt

sakurai@p04 /cygdrive/e/main/
$ ruby testM.txt
132244
sakurai@p04 /cygdrive/e/main/
$ ruby t.txt
132244
sakurai@p04 /cygdrive/e/main/
```

# 補足

- 「FuncallNode: 追加」で「なくてもよい」と書いた部分を省略すると、出力は次のようになる。

```
$ ruby intM2.rb testM.txt
def test(x)
print(x, *(x, 2) )
y = *(x, x)
print(y)
end
a = 1
b = -(-@(3) , *(a, 2) )
d = "abcd"
if ==(a, 1)  then
c = 1
else
c = 2
end
print(c)
j = 3
i = 1
while ==(i, 1) do
print(j)
j = -(j, 1)
if ==(j, 1)  then
i = 0
end
end
test(2)
```

「追加」した状態

```
def test(x)
print(x, (x*2))
y = (x*x)
print(y)
end
a = 1
b = ((-3)-(a*2))
d = "abcd"
if (a==1) then
c = 1
else
c = 2
end
print(c)
j = 3
i = 1
while (i==1)do
print(j)
j = (j-1)
if (j==1) then
i = 0
end
end
test(2)
```

# つまり

おおもと

```
print(x,x*2)
  y=x*x

b=-3-a*2

if a==1 then

while i==1 do

j=j-1
if j==1 then
```

prefix 表現

```
print(x, *(x, 2) )
y = *(x, x)

b = -(-@(3) , *(a, 2) )

if ==(a, 1)  then

while ==(i, 1) do

j = -(j, 1)
if ==(j, 1)  then
```

infix表現

```
print(x, (x*2))
y = (x*x)

b = ((-3)-(a*2))

if (a==1) then

while (i==1)do

j = (j-1)
if (j==1) then
```

優先順位を用いないため括弧を多用

# まとめ

- LLからLRまで一通り
  - ただし、原理的なことは省略した
    - 例えば、LR表の作り方
- プログラムを作ってみた（改変ですが）
- 今後、もし機会があれば、
  - 実際のコード生成を試みてください。