コンパイラ理論 12
Racc その7 (ASTを作ろう)

櫻井彰人

内容

- オブジェクト指向
  - （今回に限り）データの実装を見せない、データと操作をセット
  - Rubyでの例
- 構文木との関係
  - 作成時点
- 実際のプログラム
  - 識別子の有効領域（スコープ）の管理

オブジェクト指向

- 現在では、かなり広い意味を持つ。
- 今回（プログラミング）に関して、狭い、しかし、代表的な意味では、
- データのカプセル化
  - データの実装を隠す
  - データと操作の一体化
- クラスとインスタンス

データのカプセル化

- 例えば、スタック。
- 多くの言語にはない。
- 配列で実装する（まねする）
- 操作は（例えば）、push, pop, last, isEmpty?, new
-

例えば、スタック

- 例えば、次のようにしたい
  - a = Stack.new
  - a.push(3)
  - a.length
  - a.pop

```
{}でスタック内容を表すと:
a: {1,7} → a: {1, 7, 3}
a: {1,7, 3} → 3
a: {1,7, 3} → 3; a: {1, 7}
```

```
object.method( a, b, c )
```

オブジェクト名　メソッド名

メッセージ送信

- 「メソッド名によるメソッドの起動」ではなく、
  - オブジェクトに（すべき仕事の内容を表す）メッセージを送り
  - その答えをメッセージで受け取る

という考え方がある（実は、これが基本）

- そうすると、オブジェクトが独立に（並行して）動作するということが自然に考えられる（実はこれが基本）
- この基本的考えに忠実に作成された言語が、（オリジナルの）smalltalk
- 非常に綺麗！ しかし余りに非効率的
- そこで、メッセージ送信というアイデアは尊重しつつ、実装は、関数呼び出しと同じ実装にしたのが、現代のオブジェクト指向言語

Ruby でのオブジェクト指向の実例

```
$ irb
irb(main):001:0> 2+3
=> 5
irb(main):002:0> 2.+(3)
=> 5
irb(main):003:0> 2.+ 3
=> 5
irb(main):004:0> 2.send( :+, 3)
=> 5
irb(main):005:0> 2.send( '+', 3)
=> 5
irb(main):006:0> 2.send( "+", 3)
=> 5
irb(main):007:0> 2.send :+, 3
=> 5
```

Ruby のメソッドについては
http://www.ruby-lang.org/ja/man/html/FAQ_A5E1A5BDA5C3A5C9.html

```
irb(main):008:0> 2.puts
NoMethodError: private method `puts' called for 2:Fixnum
        from (irb):8
        from :0
irb(main):009:0> a = Object.new
=> #<Object:0x7ff9e9c4>
irb(main):010:0> a.send 'puts', 2, 3
2
3
=> nil
irb(main):011:0> a.send 'puts', [ 2, 3 ]
23
=> nil
irb(main):012:0> a.send 'puts', *[ 2, 3 ]
2
3
=> nil
irb(main):013:0>
```

練習問題6

- intp.y を動かしてみてください。そして次の確認と改善をして下さい。
  - コマンドラインからファイル名を読むようにして下さい。
  - if then else end の構文が複数行に渡ることを強制しています。一行内に書いてもよいようにして下さい。
  - （山勘を働かせて）べき乗(^)を導入してください。
    - 注意: FuncallNode.newを呼ぶときの、べき乗の関数名は、^ではなく**です。

intp.yの特徴であるAST生成は、「Ruby256倍」p.110以降に現れます
また、ASTを出力するのではなく、ASTを用いてインタープリタ動作をします

ファイルの読み込み

```
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.evaluate
rescue Racc::ParseError, Intp::IntpError, Errno::ENOENT
  raise ####
  $stderr.puts "#{File.basename $0}: #{$!}"
  exit 1
end
```

変更です

べき乗

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

```
prechigh
  nonassoc UMINUS
  right    '^'
  left     '*' '/'
  left     '+' '-'
  nonassoc EQ
preclow
```

If-then-else-end: 失敗ではないが

```
if_stmt : IF stmt THEN EOL stmt_list else_stmt END
          {
            result = IfNode.new( @fname, val[0][0],
                                 val[1], val[4], val[5] )
          }
        | IF stmt THEN     stmt_list else_stmt END
          {
            result = IfNode.new( @fname, val[0][0],
                                 val[1], val[3], val[4] )
          }
```

追加した

shift/reduce conflict がたくさん出る。というのは stmt_list から EOL stmt EOL という形が導出され、追加した部分からも、もとからある部分からも、同じものが生成されうる、すなわち、曖昧な文法になっているからである

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

If-then-else-end: 失敗ではないが

```
else_stmt : ELSE EOL stmt_list
                {
                  result = val[2]
                }
          |
                {
                  result = nil
                }
          | ELSE    stmt_list
                {
                  result = val[1]
                }
```

追加した

shift/reduce conflict がたくさん出る。というのは stmt_list から EOL stmt EOL という形が導出される、すなわち、曖昧な文法になっているからである

If-then-else-end

- つまり、逆に（つまり、この問題の原因を逆手にとり）、stmt_list が空文を許すように定義されているため、if_stmt と else_stmt で EOL を落とすだけでよい！

これで万事解決かというと、そうでもない:

- if … then a=1 else … というように、else の左側に（EOLなしに）文を書くことはできない。
- これは、stmt_list の定義によると、stmt_list の最後は必ず EOL でなければならないからである。
- つまり、… a=1 else … などとするには、stmt_list の定義を書き換えて、最後にEOLが来る必要をなくせばよい。

```
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[1]
          }
           |
          {
           result = nil
          }
```

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

では、本論に

- プログラムを読んでみよう
- プログラム全体の流れは、
  - 構文木の作成
  - それを用いた翻訳実行
- 追加機能
  - 関数定義
- ほかは、基本機能に特化

intp.yの特徴であるAST生成は、「Ruby256倍」p.110以降に現れます
また、ASTを出力するのではなく、ASTを用いてインタープリタ動作をします

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、前述のように、修正する

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

大親 Node

```
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

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
```

Function:関数本体定義

```
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
```

関数定義の仕方

とりたい！

IDENT

```
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]))
      }
```

param　stmt_list

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: 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]])
      }
```

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: 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
```

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
```

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
```

```
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
```

```
class LiteralNode < Node

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

  def evaluate(intp)
    @val
  end

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])
        }
```

```
StringNode.new(@fname, val[0][0], val[0][1])
```