

演算子オーバーライドを DSLに活用する

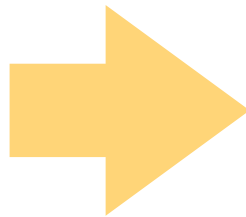
makoto kuwata

<http://www.kuwata-lab.com/>

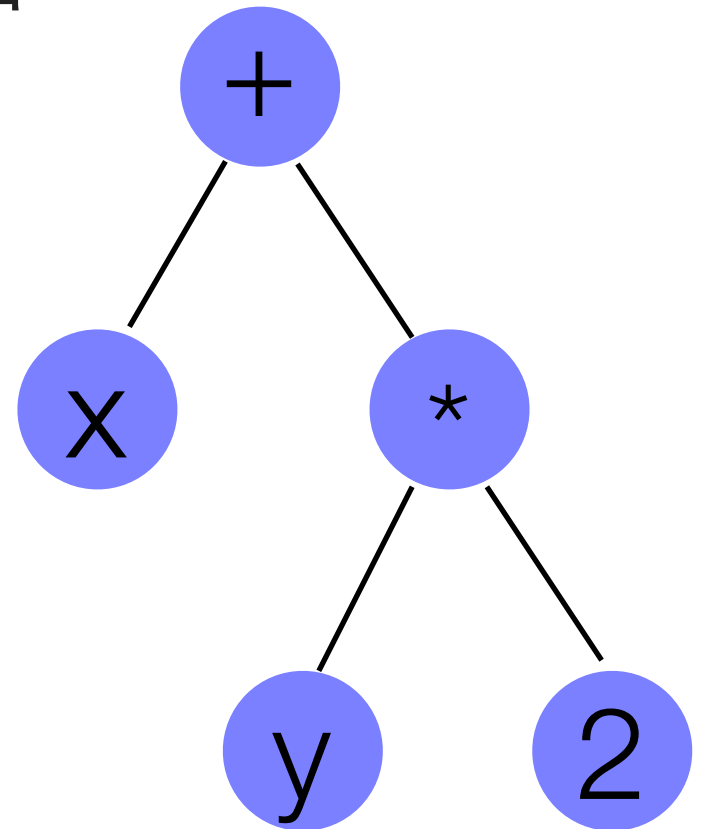
基本アイデア

- ▶ 演算結果として構文木を返す
- ▶ "構文解析" ではなく "評価"

$x + y * 2$



~~構文解析(parse)~~
評価(evaluate)



サンプルコード

サンプルコード: クラス関係

```
class Node
```

```
  ...  
end
```

構文木要素

```
class Expression < Node
```

```
  ...  
end
```

式

```
class Variable < Node
```

```
  ...  
end
```

変数

サンプルコード: Nodeクラス

```
class Node
```

```
  def +(arg)
```

```
    return Expression.new(':', '+', self, arg)
```

```
  end
```

演算結果として式を返す

```
  def *(arg)
```

```
    return Expression.new(':', '*', self, arg)
```

```
  end
```

```
  ....
```

```
end
```

サンプルコード: Expressionクラス

```
class Expression < Node
```

```
  def initialize(token, left, right)
```

```
    @token = token
```

```
    @left = left
```

```
    @right = right
```

```
  end
```

```
  attr_accessor :token, :left, :right
```

```
end
```

トークンと、
左右の Node を保持

サンプルコード: Variableクラス

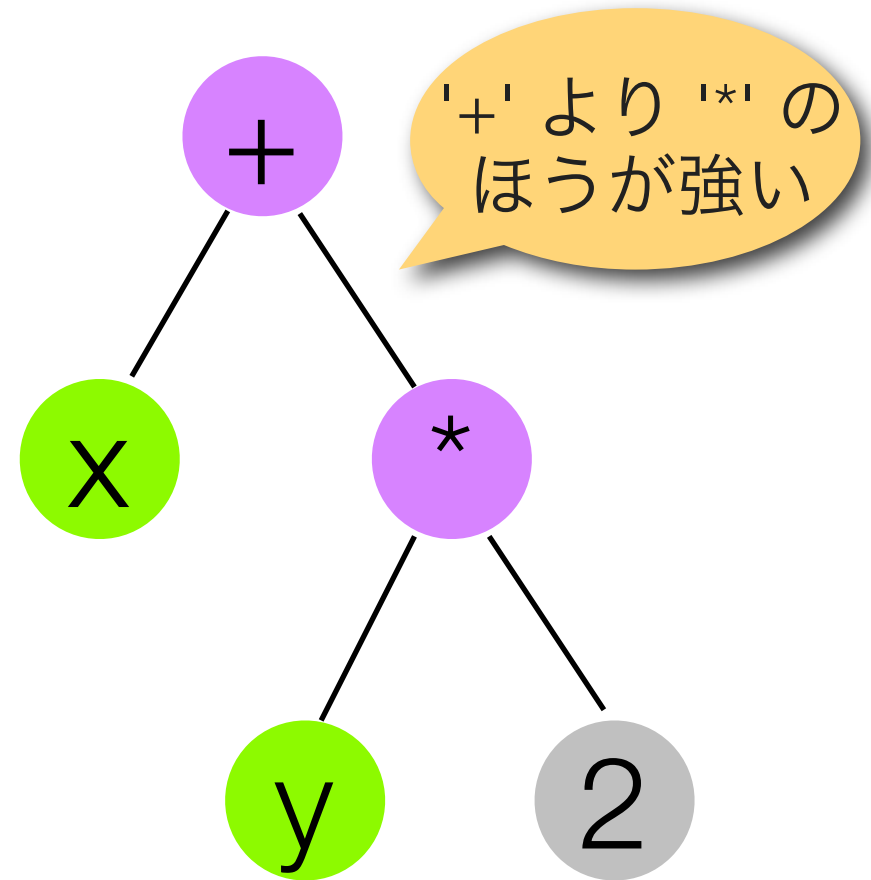
```
class Variable < Node  
  
  def initialize(name)  
    @name = name  
  end  
  
  attr_accessor :name  
  
end
```

変数名を保持

サンプルコード: 実行例

```
x = Variable.new('x')  
y = Variable.new('y')  
x + y * 2
```

```
#=> Expression.new(:'+',  
  Variable.new('x'),  
  Expression.new(':', '*',  
    Variable.new('y'),  
    1  
  )  
)
```



...構文木をさらに変換

応用例

応用例：Ruby式をSQLへ

Ruby式  構文木  SQL

Ruby

`x == 10`

`x == nil`

`x ** 3`

#=> SQL

#=> `x = 10`

#=> `x is null`

#=> `power(x, 3)`

参考: Operast (<http://github.com/kwatch/operast/>)

応用例：Ruby式をSQLへ

# Ruby	#=> SQL
<code>x > Date.new</code>	<code>x > '2008-12-06'</code>
<code>x =~ '%pattern%'</code>	<code>x like '%pattern%'</code>
<code>x.in? [1,4,7]</code>	<code>x in (1,4,7)</code>
<code>x.in? 10..20</code>	<code>x between 10 and 20</code>

参考: Operast (<http://github.com/kwatch/operast/>)

応用例：O/R Mapper の Model

```
class User
  set_table_name('users')
  @name = Variable.new('name')
  def self.name; return @name; end
  # or class <<self; attr_reader :name; end
end
```

User.name == nil #=> users.name is null

問題点、その他

言語の必要条件

- ▶ 演算子がオーバーライドできること
 - Ruby
 - Python
 - C++ (演算子オーバーロード)
- ▶ 型がないこと or 型が変更できること
 - 「==」の戻り値の型が boolean に固定されていると、実現不可能

問題点：オーバーライド不可

- ▶ オーバーライドできない演算子がある
 - 例：Ruby1.8 では、「!=」は「==の否定」に固定
- ▶ 解決策：
 - 他の演算子で代用（「^」 「<=>」 など）
 - 「x.not」 や 「x.ne」 (not equal) を提供

```
x.not == 1    #=> x != 1
```

問題点：&& と ||

- ▶ 「&&」 と 「||」 はオーバーライドできない
 - 評価のショートカットがあるため
- ▶ 解決策：
 - メソッドで代用（「.and」 「.or」）
 - bit 演算子で代用（ただし演算子の優先順位に注意）

`(x == 1) & (y == 2) #=> x = 1 and y = 2`

問題点： $1 + x$

- ▶ 「 $x + 1$ 」は OK だが 「 $1 + x$ 」は NG
- ▶ 解決策：一時的に `Integer#+` を override

```
def sandbox
  begin
    Integer#+ に alias 名をつけて退避
    Integer#+ をオーバーライド
  yield
  ensure
    Integer#+ をもとに戻す
  end
end
```

問題点：動作速度

▶ 動作速度は、たぶん遅い

- 自作 O/R Mapper で評価中

▶ 解決策：

- なさそう？
- キャッシュもできない (or 逆に遅くなる)
- 最高速である必要はない
(気にならない程度の遅さならよしとする)

まとめ

まとめ

- ▶ 演算子をオーバーライドして構文木を返す
 - 「構文解析」ではなく「評価」
 - Ruby の式をそのまま使う → DSL で使いやすい
- ▶ アイデア次第で面白い使い方ができるかも

参考

- ▶ SQLAlchemy (Python)
 - <http://www.sqlalchemy.org/>
- ▶ Sequel (Ruby)
 - <http://sequel.rubyforge.org/>
- ▶ RSpec (Ruby)
 - <http://rspec.info/>
- ▶ Operast (Ruby)
 - <http://github.com/kwatch/operast/>

one more thing...

こんな面倒なこと、本当に必要?

- ▶ Lispなら、同じことが「`'`」だけでできる
- ▶ 言語として必要な機能を Ruby が提供していないだけでは?

`(+ x (* y 2))` ;; 式が計算される

`'(+ x (* y 2))` ;; 式の構文木が返される

Lisp最強

thank you