80 分钟 8 门语言

宋方睿

2012 年 6 月 30 日

Outline

color

声明

- 对大多数语言我的理解都很肤浅，甚至无法用该语言写出一个不琐碎的程序
- 希望大家能了解到传统的 imperative、object-oriented 以外的东西
- 对编程语言有不同的认识 (如工具是要精心挑选并且需要打磨的)

(display “hello, world”)

Example (hello, world)

- `(display "hello, world")`

Scheme

- Appeared in 1975
- Functional
- Imperative

理念

- minimalism
- tail recursion
- first-class continuation
  - Icon(?) 的 generator
  - 更熟悉的名字是 Python 里的 yield
  - coroutine
  - exception handling
  - delimited continuation: reset/shift
- lazy evaluation
  - delay/force
- macro
- sigil，如 number? set!

40 - 32 / 2 = 4 !

Example (Twitter)

- 40 - 32 / 2 = 4 !
- 40 - (32 / 2) = 4*3*2*1
- (((40 - 32) / 2) = 4) !

Smalltalk

- Appeared in 1972
- Object-oriented
- Dynamically typed
- Reflective

Smalltalk

- Appeared in 1972
- Object-oriented
- Dynamically typed
- Reflective

Example (hello, world)

```
Transcript show: 'Hello, World!'
```

Message passing

Example (Unary messages)

- 42 factorial
- ‘meow’ size

Message passing

Example (Binary messages)

- $1 + 2$
- $3 = 3$

Message passing

Message passing

Message passing

Example (Keyword messages)

- 每个 selector 后有个冒号
- Transcript print: ‘hello, world’
- 0 bitAt: 3 put: 1

和常规方法比较

- 没有繁杂的优先级，只须记住 unary > binary > keyword
- 相同类型的左结合
- 函数调用时参数非匿名有什么好处？

Block

Example

- `[:x | x+1]`
- `anObject ifTrue: [block-expression-1. block-expression-2]`
- `anObject ifTrue: [block-expression-1. block-expression-2] ifFalse: [block-expression-3. block-expression-4]`
- 不是 applicative order，所以放在 block 里可以延缓求值

blocks

- First-class objects
- 可以形成 closure
- 通过 value: message 来执行
- `[3] value`
- `[:x | x+1] value: 3`
- `[:x :y | x+y] value: 3 value: 3`

List comprehensions

Example (Smalltalk way)

- (1 to: 10) fold: [:x :y | x * y].

Example (if)

```
anObject
  ifTrue: [
    block1
  ] ifFalse: [
    block2
  ]
```

影响

- object-oriented programming
- 窗口环境的设计
- virtual machine
- message passing 风格和 class hierarchy
- integrated development environment
- hot swapping

Lua

- Appeared in 1993
- imperative
- prototype-based object-oriented

Table

- Assocative array
- 数组也是 table
- Metatable，object-oriented 的基石。Javascript 的 prototype 也是个类似的设计。

Llama, alpaca, Camel

Perl

- Appeared in 1987
- Imperative
- Reflective

Regular expression

- named capture
- look-around
- executing code
- `(?>pattern)`

Sigil

```
@newAoA = splice_2D( \@AoA, 4 => 8, 7 => 12 );
my $lrr = shift;
```

- 丑陋。因为同样的理由讨厌 PHP……

Overloadding

- 滥用 Zero one infinity rule
- 重载过头了，比如 `perldoc -f open` 给的 open 的用法
- 为了不是非常常见的情况下的便利牺牲了可读性，增加了复杂性
- 表现力其实非常差，增加的表现力都是通过繁杂零碎的细节规则实现的，而非通用的。和 Scheme/Smalltalk 相比无疑是个弱者

Ruby

- Appeared in 1995
- Reflective
- Dynamic typing
- Object-oriented

高度 object-oriented

- `3.to_s`
- `4.class`
- `5.methods`
- `6.send :to_f`, Smalltalk 的 message-passing 风格

Ruby

- Mixin。消除了不少 marker interface 的问题
- 对 functional 友好 (听说 Python 的 BDFL 不遗余力地想排除 map filter 这些函数，他本人学习 Haskell 的一些吐槽也被 reddit 上的人吐槽……打住，再下去就 flame war 了)
- `sort! nil?` 来自 Scheme
- `@instance_var` sigil，来自 Perl，但含义发生变化，还有 `$global` 表示 global variable
- 只有 nil 和 false 为假，`[]` `""` `{}` 等是真，更安全，nil 是 dynamic typing 和 reflective 的一个妥协
- First-class continuation 来自 Scheme
- Operator overloading 的方式
- Smalltalk 的 block，Ruby 用 do end 语法只支持一个 block(实际上这样够用了)
- 易于实现 embedded domain specific language

Ruby 1.9 hash syntax

- 是不是类似 Smalltalk 的 keyword message

```ruby
def f(*args)
  p args
end

f 3, 'a', one:5, two:8
```

```
=> [3, "a", {:one=>5, :two=>8}]
```

- {:arr => [1,2], :str => 'hello'}

Block

- 是不是 Python with statement 的通用版
-

```ruby
def fun &block
  # acquire resource
  block.call 42
  # release resource
end

fun do
  p 'hello'
end
```

Block

- `(1..5).inject(1) {|acc,x| acc*x }`

Prolog

- Appeared in 1972
- Logic

模型

- 事实
- 规则
- 查询

```
f(a,c).
f(b,c).

?- f(X,b).
X = a ;
X = c.
```

Operator

- 这个可能是方言里才有的 (Prolog 和 Lisp 一样也是方言众多)
- 可以自定义，还能指定优先级，结合性

Data type

- 大量使用 compound term 表示自定义数据类型
- `truck_year('Mazda', 1986)`
- 用 pattern matching 来提取字段

Predicate

- SLD resolution
- 相当于语言内置的 backtracking 机制
- 会尝试返回所有结果，除非使用了 cut 来剪枝

Definite clause grammar

- 像 Prolog 这么简单 (语法) 的语言也引入特殊语法

Example (normal)

```
sentence(S1,S3) :- noun_phrase(S1,S2),
  verb_phrase(S2,S3).
noun_phrase(S1,S3) :- det(S1,S2), noun(S2,S3).
verb_phrase(S1,S3) :- verb(S1,S2),
  noun_phrase(S2,S3).
det([the|X], X).
det([a|X], X).
noun([cat|X], X).
noun([bat|X], X).
verb([eats|X], X).
```

Definite clause grammar

Example (DCG)

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> det, noun.
verb_phrase --> verb, noun_phrase.
det --> [the].
det --> [a].
noun --> [cat].
noun --> [bat].
verb --> [eats].
```

Quick sort

```
qsort([]) -> [];
qsort([P|XS]) -> qsort([X || X <- XS, X < P]) ++
  [P] ++ qsort([X || X <- XS, X >= P]).
```

Erlang

- Appeared in 1986
- functional
- dynamic typing

Data type

- 非常少，基本都用 tuple 表示
- Lua 基本都用 table 表示
- Lisp 基本都用 list 表示
- 注意 list/tuple 看上去表现力是比 table/hash 弱的
- 所以采用 Lisp 的 property list 形式
- 小写字母打头的标识符是 atom，结合 tuple 能起到类似 property list 的功效
- `{hello, {x, Data}}` `{add,3,4}`

Concurrency

- 语言内置 process，Erlang VM 实现的可以看做是 green process
- 和 green thread 的差异在于 process 间没有共享的状态
- Actor model，避免 lock

OCaml

- Appeared in 1983
- Strong static typing
- Functional
- Imperative
- object-oriented

Function call

- juxtaposition

```
print_endline "hello, world"
```

结合 functional 和 object-oriented

- Expression problem
- Algebraic data type 与 class-based object-oriented 采取的方式正交
- 给你自由选择更合适的方式

Currying

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# add 3;;
- : int -> int = <fun>
```

- Ruby/CoffeeScript 也用了 juxtaposition，但是若支持 currying，由于 dynamic typing，容易产生 bug

Structural typing

- 两个 class，如果暴露给外界的接口 (public) 完全一致，那么这两个类就没有区别
- A 的接口是 B 的超集，那么 A 就是 B 的子类
- C++/Java 这类语言不得不用到很多 marker interface，而且因为 static typing，还没法动态生成类型

Module system

- 非常强大的高阶 module system，我不知道有第二个语言能与之相提并论
- Functor

Haskell

- Appeared in 1990
- Purely functional
- Strong static typing
- Lazy

All positive rationals

```
take 8 $ fix ((1:).(>>= \x->[1+x,1/(1+x)]))

[1.0,2.0,0.5,3.0,0.3333333333333333,1.5,0.6666666666666
```

Powerset

```
filterM (const [True, False]) [1..3]
```

```
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Typeclass

- Int 可以做除法，Double 可以做除法，Rational 可以做除法。需要有个 typeclass 定义哪些类型能做除法。
- 类似 C++ 的 virtual class 或者 Java 的 interface

Num [a]

Example (Example)

```
instance Num a => Num [a] where
    (f:fs) + (g:gs) = f+g : fs+gs
    fs + [] = fs
    [] + gs = gs
    (f:fs) * (g:gs) = f*g : [f]*gs + fs*(g:gs)
    _ * _ = []

[1,1]^4 == [1,4,6,4,1]
```

Re-invent imperative programming

```
do x <- get; put 5; y <- get; put 6
```

Parsec

- 解析一个 csv 文件

Example (csv)

```
csv :: CharParser st [[String]]
csv = (record `sepEndBy` newline) <* eof
  where
    record = (quoted <|> many (noneOf ",\"\n")) `sepBy`
    quoted = between (char '"') (char '"') . many $ none
```

– 可以自定义操作符，甚至优先级

Pattern matching

Example (Pattern guards)

```
add x y | 3 <- x = x + y
```

Example (View patterns)

```
mysort (sort -> xs) = xs
```

Off-side rule

- 源自 ISWIN
- 使代码更清晰

Number literal

- 1 是否应该作为 Int 类型？
- 应该是 Num 这个 Typeclass 的任意一个 instance

String literal

- “asdf” 可以是 IsString 这个 Typeclass 的任意一个 instance。
- 可以是 [Char]，可以是 ByteString，可以是你的自定义类型，只要提供了 IsString 的 instance

List comprehension

- List comprehension 的语法非常漂亮，是否应该有 set/dictionary comprehension？
- 有 Monad comprehension 就够了

Section

- 二元操作符也应该拥有 currying 的能力
- Left section, `(12+)`, 相当于 `(+) 12`
- Right section, `(*13)`, 相当于 `flip (*) 13`
- `(3 `elem`)`, 把正常函数变成 operator 再做 section

Monad

- 传统：一个函数接受参数 a 返回 b
- 上述操作可以看做一个 action(原用于 IO)
- Monad 很通用，值得拥有一个单独的语法 do(Python 的 with statement 是否通用到值得为它赋予新语法? 疑似源自 Lisp 的 unwind-protect)
- Arrow 比 Monad 更抽象，也值得拥有类似的语法 proc
- 可以看做 structured output
- 可以看做附带了一个环境

Monad transformer

- 每一个 monad 都代表一种特殊的效果。能不能把它们堆砌起来？于是就有了 monad transformer
- 比如 xmonad 中的 layout transformer，REFLECTX TABBED MIRROR NOBORDERS

Meta programming

- Lisp 的 macro 是一个好方式，但是类型不够安全，需要类型安全的 Template Haskell

Fold

- 什么样的东西能 fold？List？Tree？Foldable！被 fold 的元素有什么特征？Monoid！
- Catamorphism
- Ruby 解决这个问题的方式是 duck typing
- “When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”
- Common Lisp 解决这个问题的方式是让 map delete-if 这样的函数接受一个参数代表类型，有点丑陋

Getter/setter

- Haskell 的 record syntax 的设计不够好
- Lens

```
test :: [Int]
test = (addOne `to` everyOther) `from` [1, 2, 9, 6, 7, 8, 4
-- test == [2, 2, 10, 6, 8, 8, 5]
```

Lazy evaluation

- 消除了不必要的计算，比如和实现方式和传统毫无差异的 merge sort，如果你用它来取列表 head，那么时间复杂度是 O(n) 的
- 可以处理无限长的列表
- 一边算一边用，比如 `fibs = 0:scanl (+) 1 fibs`