8 门编程语言的设计思考

宋方睿
http://maskray.tk
emacs.ray@gmail.com

2012 年 7 月 21 日

Outline

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

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

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

Scheme

- Appeared in 1975
- Functional

Scheme

- Appeared in 1975
- Functional
- Imperative

理念

- minimalism

Homoiconicity

- (name “mad” age 60)

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))
- (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))
- (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
- Douglas McIlroy (1960) Macro Instruction Extensions of Compiler Languages

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))
- (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
- Douglas McIlroy (1960) Macro Instruction Extensions of Compiler Languages
- 程序形式和数据结构一致

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))
- (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
- Douglas McIlroy (1960) Macro Instruction Extensions of Compiler Languages
- 程序形式和数据结构一致
- 易于扩展

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))
- (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
- Douglas McIlroy (1960) Macro Instruction Extensions of Compiler Languages
- 程序形式和数据结构一致
- 易于扩展
- 容易实现 meta programming

Homoiconicity

- (name “mad” age 60)
- ((name . “foo”) (name . “bar”) (name . “metasyntactic”))
- (define (fib n) (if (< n 2) n (+ (fib (- n 1)) (fib (- n 2)))))
- Douglas McIlroy (1960) Macro Instruction Extensions of Compiler Languages
- 程序形式和数据结构一致
- 易于扩展
- 容易实现 meta programming

- tail recursion

First-class continuation

```
(display
  (call/cc (lambda (cc)
             (cc "escape from the point")
           (display "This sentence won't get displayed"))))

(let ((k #f))
  (call/cc (lambda (cc) (set! k cc)))
  (display "loop\n")
  (k))
```

First-class continuation

```
(define cont #f)
(define (f)
  (let ((i 0))
    (call/cc (lambda (k) (set! cont k)))
    (set! i (1+ i))
    i))
(define a (f)) (display a) (newline)
(cont) (cont) (cont) (display a) (newline)
```

- Icon 的 generator

First-class continuation

```
(define cont #f)
(define (f)
  (let ((i 0))
    (call/cc (lambda (k) (set! cont k)))
    (set! i (1+ i))
    i))
(define a (f)) (display a) (newline)
(cont) (cont) (cont) (display a) (newline)
```

- Icon 的 generator
- 更熟悉的名字是 Python 里的 yield

First-class continuation

```
(define cont #f)
(define (f)
  (let ((i 0))
    (call/cc (lambda (k) (set! cont k)))
    (set! i (1+ i))
    i))
(define a (f)) (display a) (newline)
(cont) (cont) (cont) (display a) (newline)
```

- Icon 的 generator
- 更熟悉的名字是 Python 里的 yield
- coroutine

First-class continuation

```
(define cont #f)
(define (f)
  (let ((i 0))
    (call/cc (lambda (k) (set! cont k)))
    (set! i (1+ i))
    i))
(define a (f)) (display a) (newline)
(cont) (cont) (cont) (display a) (newline)
```

- Icon 的 generator
- 更熟悉的名字是 Python 里的 yield
- coroutine
- exception handling

First-class continuation

```
(define cont #f)
(define (f)
  (let ((i 0))
    (call/cc (lambda (k) (set! cont k)))
    (set! i (1+ i))
    i))
(define a (f)) (display a) (newline)
(cont) (cont) (cont) (display a) (newline)
```

- Icon 的 generator
- 更熟悉的名字是 Python 里的 yield
- coroutine
- exception handling
- delimited continuation: reset/shift

Lazy evaluation

- What is lazy evaluation? (delay/force)

Lazy evaluation

- What is lazy evaluation? (delay/force)
- I'm too lazy to answer the question right now.

Macro

Naming convention

- 谓词名以? 结尾

Naming convention

- 谓词名以? 结尾
- eq? zero?

Naming convention

- 谓词名以? 结尾
- eq? zero?
- 类型转换函数名称中间有 ->

Naming convention

- 谓词名以? 结尾
- eq? zero?
- 类型转换函数名称中间有 ->
- list->vector

Naming convention

- 谓词名以? 结尾
- eq? zero?
- 类型转换函数名称中间有 ->
- list->vector
- 带副作用的函数

Naming convention

- 谓词名以? 结尾
- eq? zero?
- 类型转换函数名称中间有 ->
- list->vector
- 带副作用的函数
- set! vector-set!

40 - 32 / 2 = 4 !

Example (Twitter)

- 40 - 32 / 2 = 4 !

40 - 32 / 2 = 4 !

Example (Twitter)

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

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

Smalltalk

- Appeared in 1972
- Object-oriented

Example (hello, world)

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

Smalltalk

- Appeared in 1972
- Object-oriented
- Dynamically typed

Example (hello, world)

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

Smalltalk

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

Example (hello, world)

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

Message passing

Example (Unary messages)

- 42 factorial

Message passing

Example (Unary messages)

- 42 factorial
- ‘meow’ size

Example (Binary messages)

Message passing

Example (Unary messages)

- 42 factorial
- ‘meow’ size

Example (Binary messages)

- $1 + 2$

Message passing

Example (Unary messages)

- 42 factorial
- ‘meow’ size

Example (Binary messages)

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

Message passing

Example (Keyword messages)

- 每个 selector 后有个冒号

Message passing

Example (Keyword messages)

- 每个 selector 后有个冒号
- Transcript print: ‘hello, world’

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

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

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]`

Block

Example

- `[:x | x+1]`
- `anObject ifTrue: [block-expression-1. block-expression-2]`

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]`

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(有些材料中这个词和 call by value 同义)，所以放在 block 里可以延缓求值

Blocks

- First-class objects

Blocks

- First-class objects
- 可以形成 closure

Blocks

- First-class objects
- 可以形成 closure
- 通过 value: message 来执行

Blocks

- First-class objects
- 可以形成 closure
- 通过 value: message 来执行
- `[3] value`

Blocks

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

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

影响

- object-oriented programming
- 窗口环境的设计

影响

- object-oriented programming
- 窗口环境的设计
- virtual machine

影响

- object-oriented programming
- 窗口环境的设计
- virtual machine
- message passing 风格和 class hierarchy

影响

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

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

Lua

- Appeared in 1993
- imperative

Lua

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

Table

- Assocative array

Table

- Assocative array
- 数组也是 table

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。

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。
- require

Object-oriented

```
Account = {}
function Account:new()
    o = {balance = 0}
    setmetatable(o, self)
    self.__index = self
    return o
end

function Account:deposit(v)
    self.balance = self.balance + v
end

BankPresidentAccount = Account:new()
function BankPresidentAccount:deposit(v)
    self.balance = self.balance + v * 2
end

joe = BankPresidentAccount:new()
joe:deposit(17)
print(joe.balance)
```

Llama, alpaca, Camel

Perl

- Appeared in 1987

Perl

- Appeared in 1987
- Imperative

Perl

- Appeared in 1987
- Imperative
- Reflective

Regular expression

- named capture

Regular expression

- named capture
- look-around

Regular expression

- named capture
- look-around
- executing code

Regular expression

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

Sigil

example

```
my @a \= qw/0 1 2 3/; $a[1]
```

Sigil

example

```
sub v{
   $m = pop;
   my @g = b\%t,$*;
   $s = \@g;
   $z = 0;
   map{ $x=0;
        map{
             $$s[$z][$x] = $$m[$z][$x] eq '#' && $$s[$z][$
+' : '_';
             $x++
           } @$_;
        $z++
      } @$m;
   $s
}
```

莫名其妙的全局变量

- $. ($INPUT_LINE_NUMBER)

莫名其妙的全局变量

- $. ($INPUT_LINE_NUMBER)
- $, ($OUTPUT_FIELD_SEPARATOR)

莫名其妙的全局变量

- $. ($INPUT_LINE_NUMBER)
- $, ($OUTPUT_FIELD_SEPARATOR)
- $/ ($INPUT_RECORD_SEPARATOR)

莫名其妙的全局变量

- $. ($INPUT_LINE_NUMBER)
- $, ($OUTPUT_FIELD_SEPARATOR)
- $/ ($INPUT_RECORD_SEPARATOR)
- perldoc perlvar

难以创造稍微复杂一点的数据结构

- 二维数组？

```
$a = [[1],[2]];  print $a->[1][0];
```

难以创造稍微复杂一点的数据结构

- 二维数组？

```
$a = [[1],[2]];  print $a->[1][0];
```

难以创造稍微复杂一点的数据结构

- 二维数组？

```
$a = [[1], [2]];  print $a->[1][0];
```

- perldoc perllol

Increment Operator

```
$a=3; $a++; print $a;
```

```
$a='ab'; $a++; print $a;
```

Increment Operator

```
$a=3; $a++; print $a;
```

```
$a='ab'; $a++; print $a;
```

Object-oriented

```
package Person;
sub new
{
    my $class = shift;
    my $self = {_firstName => shift, _lastName  => shift
    bless $self, $class;
    $self;
}

package Employee;
use Person;
our @ISA = qw(Person);
```

Overloading

- void/scalar/list context

Overloading

- void/scalar/list context
- 滥用 Zero one infinity rule

Overloading

- void/scalar/list context
- 滥用 Zero one infinity rule
- 重载过头了，比如 `perldoc -f open` 给的 open 的用法

Overloading

- void/scalar/list context
- 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

Overloading

- void/scalar/list context
- 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

Ruby

- Appeared in 1995
- Reflective

Ruby

- Appeared in 1995
- Reflective
- Dynamic typing

Ruby

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

高度 object-oriented

- `3.to_s`

高度 object-oriented

- `3.to_s`
- `4.class`

高度 object-oriented

- `3.to_s`
- `4.class`
- `5.methods`

高度 object-oriented

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

理念

- Mixin。消除了不少 marker interface 的问题

理念

- 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)

理念

- 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

理念

- 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

理念

- 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

理念

- First-class continuation 来自 Scheme

理念

- First-class continuation 来自 Scheme
- Operator overloading 的方式

理念

- 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)

理念

- 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}]
```

Ruby 1.9 hash syntax

- 是不是类似 Smalltalk 的 keyword message

```
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

- 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

Prolog

- Appeared in 1972
- Logic

模型

- 事实

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

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

模型

- 事实
- 规则

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

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

模型

- 事实
- 规则
- 查询

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

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

Operator

- 一些方言中允许自定义，还能指定优先级和结合性

Data type

- 大量使用 compound term 表示自定义数据类型

Data type

- 大量使用 compound term 表示自定义数据类型
- `truck_year('Mazda', 1986)`

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

Predicate

- SLD resolution
- 相当于语言内置的 backtracking 机制

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].
```

Unification

- ```
  append([ ], L, L).
  append([X | L1], L2, [X |  L3]) :- append(L1, L2, L3).
  ```
  ```

Unification

- ```
  append([ ], L, L).
  append([X | L1], L2, [X |  L3]) :- append(L1, L2, L3).
  ```
  - ```
    ?- append([1,2],[3],X).
    ```

Unification

- ```
  append([ ], L, L).
  append([X | L1], L2, [X |  L3]) :- append(L1, L2, L3).
  ```
  - ```
    ?- append([1,2],[3],X).
    ```
  - ```
    ?- append([1,2],[3],[1,2,3].
    ```
    ```

Unification

- ```
  append([ ], L, L).
  append([X | L1], L2, [X |  L3]) :- append(L1, L2, L3).
  ```
  - `?- append([1,2],[3],X).`
  - `?- append([1,2],[3],[1,2,3].`
  - `?- append([1,2],X,[1,2,3]).`
  ```

Unification

- ```
  append([ ], L, L).
  append([X | L1], L2, [X |  L3]) :- append(L1, L2, L3).
  ```
  - `?- append([1,2],[3],X).`
  - `?- append([1,2],[3],[1,2,3].`
  - `?- append([1,2],X,[1,2,3]).`
  - `?- append(X,Y,[1,2,3]).`
  ```

Quick sort

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

Erlang

- Appeared in 1986

Erlang

- Appeared in 1986
- functional

Erlang

- Appeared in 1986
- functional
- dynamic typing

Data type

- 非常少，基本都用 tuple 表示

Data type

- 非常少，基本都用 tuple 表示
- Lua 基本都用 table 表示

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

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

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

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

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

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

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

OCaml

- Appeared in 1983
- Strong static typing

OCaml

- Appeared in 1983
- Strong static typing
- Functional

OCaml

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

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

结合 functional 和 object-oriented

- Expression problem
- Algebraic data type 与 class-based object-oriented 采取的方式正交

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

Module system

- 把相互关联的函数组织在一起 (如定义了一个数据类型以及操作这个数据类型的一系列函数)

```
module Stack = struct
  exception Stack_is_empty
  type 'a stack = Empty | Elem of 'a * 'a stack
  let push s e = Elem (e, s)
  let pop = function
    | Empty -> raise Stack_is_empty
    | Elem (_,s) -> s
  let top = function
    | Empty -> raise Stack_is_empty
    | Elem (e,_) -> e
end
```

Module system

- 把相互关联的函数组织在一起 (如定义了一个数据类型以及操作这个数据类型的一系列函数)
- separate compilation

```
module Stack = struct
  exception Stack_is_empty
  type 'a stack = Empty | Elem of 'a * 'a stack
  let push s e = Elem (e, s)
  let pop = function
    | Empty -> raise Stack_is_empty
    | Elem (_,s) -> s
  let top = function
    | Empty -> raise Stack_is_empty
    | Elem (e,_) -> e
end
```

Functor

```
module type Eq = sig
  type t
  val eq : t -> t -> bool
end
module MakeSet (E : Eq) = struct
  type t = E.t
  let empty = []
  let add x s = x::s
  let find x s = List.find (E.eq x) s
end

module StrEq = struct
  type t = string
  let eq s1 s2 = s1 = s2;
end
module StrSet = MakeSet (StrEq)
```

Encapsulation

```
# let o = object
    val xs = "ray"
    method output = print_endline xs
  end;;

val o : < output : unit > = <obj>
```

Row polymorphism

```
# fun x -> x#output;;
- : < output : 'a; .. > -> 'a = <fun>
```

Private method

single coersion

```
let f = fun x -> x#output;; end
type 'a y = <output: 'a; input: 'a -> unit> -> 'a

f :> 'a y
```

Structural typing

- 两个类型，如果暴露给外界的接口 (public) 完全一致，那么这两个类型就认为是相同的

Structural typing

- 两个类型，如果暴露给外界的接口 (public) 完全一致，那么这两个类型就认为是相同的
- A 的接口是 B 的超集，那么 A 就是 B 的子类

Structural typing

- 两个类型，如果暴露给外界的接口 (public) 完全一致，那么这两个类型就认为是相同的
- 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

Quote

** J. Maurice Rojas

- Once you decide strongly to stick with Haskell, it will take no longer than seven lives to attain the mastery.

Haskell

- Appeared in 1990

Haskell

- Appeared in 1990
- Purely functional

Haskell

- Appeared in 1990
- Purely functional
- Strong static typing

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

- 实现 ad-hoc polymorphism 的一种方式

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。
- 单参数的 typeclass 类似 C++ 的 virtual class 或者 Java 的 interface

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。
- 单参数的 typeclass 类似 C++ 的 virtual class 或者 Java 的 interface
- 多参数、带 context、functional dependency 的情况下就需要动用 traits template 和 SFINAE principle 了

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。
- 单参数的 typeclass 类似 C++ 的 virtual class 或者 Java 的 interface
- 多参数、带 context、functional dependency 的情况下就需要动用 traits template 和 SFINAE principle 了
- C++ template 中的 partial specialization 在 typeclass 中没有直接对应物

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。
- 单参数的 typeclass 类似 C++ 的 virtual class 或者 Java 的 interface
- 多参数、带 context、functional dependency 的情况下就需要动用 traits template 和 SFINAE principle 了
- C++ template 中的 partial specialization 在 typeclass 中没有直接对应物
- 没有一般化的 overlapping instance 扩展，但可用 newtype 引入新类型，或者用 context 结合 functional dependency 解决

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。
- 单参数的 typeclass 类似 C++ 的 virtual class 或者 Java 的 interface
- 多参数、带 context、functional dependency 的情况下就需要动用 traits template 和 SFINAE principle 了
- C++ template 中的 partial specialization 在 typeclass 中没有直接对应物
- 没有一般化的 overlapping instance 扩展，但可用 newtype 引入新类型，或者用 context 结合 functional dependency 解决
- C++ template 不支持 paramorphic recursion(递归时参数类型变化)

Typeclass

- 实现 ad-hoc polymorphism 的一种方式
- 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。
- 单参数的 typeclass 类似 C++ 的 virtual class 或者 Java 的 interface
- 多参数、带 context、functional dependency 的情况下就需要动用 traits template 和 SFINAE principle 了
- C++ template 中的 partial specialization 在 typeclass 中没有直接对应物
- 没有一般化的 overlapping instance 扩展，但可用 newtype 引入新类型，或者用 context 结合 functional dependency 解决
- C++ template 不支持 paramorphic recursion(递归时参数类型变化)
- 原因是 C++ template 把一切留在编译器解决

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

Off-side rule

- 源自 ISWIN
- 使代码更清晰

Number literal

- 1 是否应该作为 Int 类型？

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。

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？

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

Section

- 二元操作符也应该拥有 currying 的能力
- Left section, (12+), 相当于 (+) 12

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`

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

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

- 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)

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

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

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

Failure

不要吐槽这段代码不 idiomatic，照顾不用 Ruby 的同学

```
def foo
  return 'Meowth' unless f 36
  return 'Persian' unless g 1
  return 'fqj' unless h 22
  return 17
end
```

Error monad

```
foo = do
  f 36
  g 1
  h 22
  return 17
```

Non-determinism

```
def bar
  [1,2,3].map {|x|
    [4,5,6].map {|y|
      x*y
    }
  }
end
```

List monad

```
baz = do
  x <- [1,2,3]
  y <- [4,5,6]
  return $ x*y

baz = [x*y | x <- [1,2,3], y <- [4,5,6]]

baz = (*) <$> [1,2,3] <*> [4,5,6]
```

Environment

```
def f(x); g(x); i(x); end
def g(x); h(x); end
def h(x); x*x; end
def i(x); x+3; end
```

Reader monad

```
f = g >> i
g = h
h = join (*) <$> ask
i = (+3) <$> ask
```

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

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 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！

Fold

- 什么样的东西能 fold？List？Tree？Foldable！被 fold 的元素有什么特征？Monoid！
- Catamorphism

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

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.”

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

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

Getter/setter

- Haskell 的 record syntax 的设计不够好
- Lens (functional reference)

```
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

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

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`