

函数式编程入门



卢小海，绿盟科技

密级：内部使用

1 历史背景

2 编程范式

3 基本概念

4 真实案例

5 参考资料

如何应对计算机程序日益增加的规模和复杂性

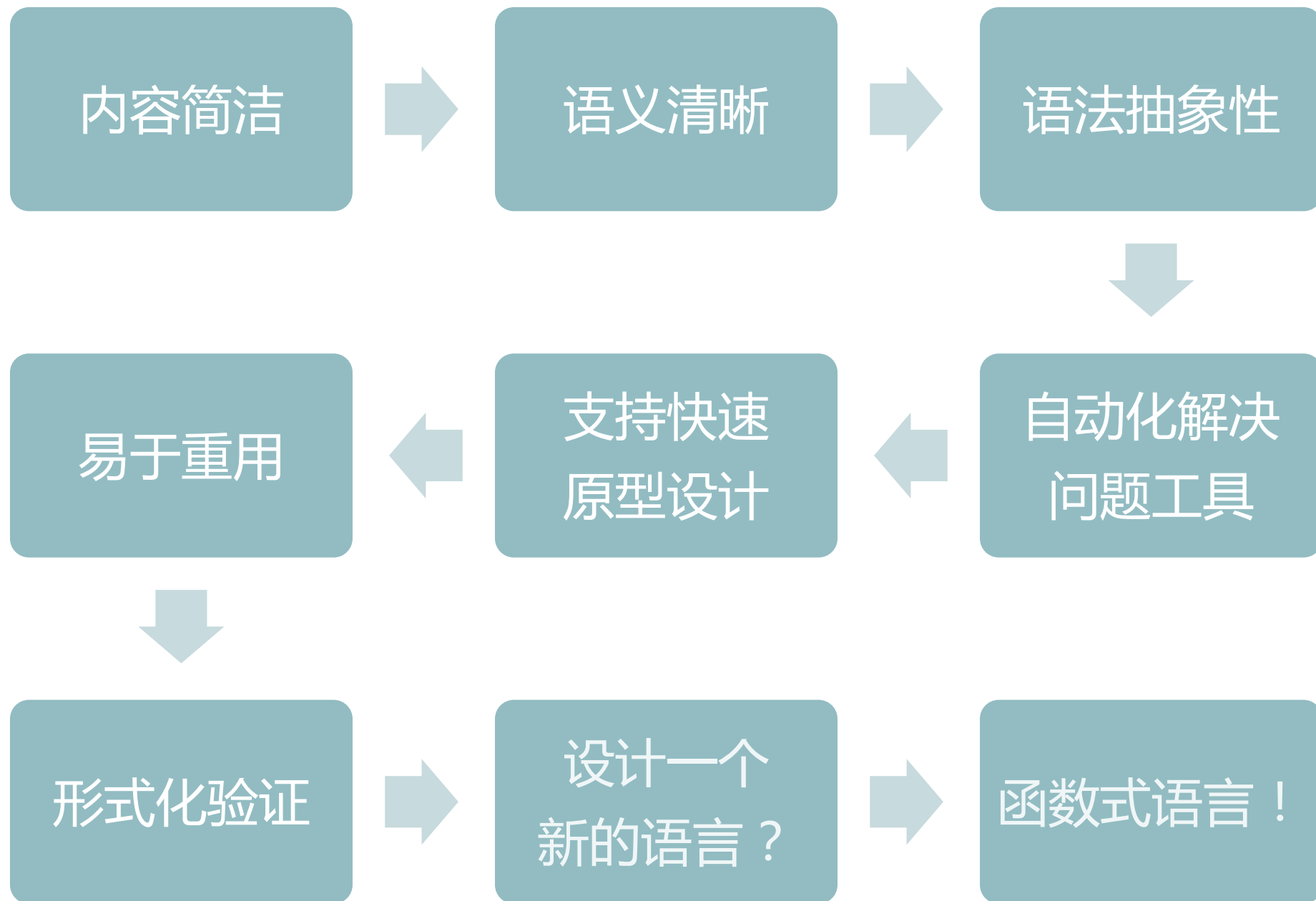
- LOC, line of code. Window 3.1(1992) 3million LOC, Windows 2000 30-50million LOC

如何降低开发软件和维护软件的费用

- maintenance can be up to 70%, including understanding, debugging and modifying the code

如何增强我们对于软件正确性的信心

- The European Ariane 5(1996), cost 10 years and \$7billion, explode after 40s in its maiden voyage; the floating point division in the Pentium processor cost Intel around 470 million



- functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.
- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments,

- Java
 - `total = 0;`
 `for (i = 1; i <= 10; ++i)`
 `total = total+i;`
- Haskell
 - `sum [1..10]`
- Python
 - `sum(range(10))`
 - `reduce(lambda x, y: x+y, range(10))`

1920s–1940s:



Alonzo Church

and



Haskell Curry

developed lambda calculus (λ 演算), 它是一个函数的理论, 也是串行计算的一个模型。

1960s:



John McCarthy

develops the first functional language Lisp, influenced by **lambda calculus**, but still with **variable assignments**.

- Late 1960s: Peter Landin develops ISWIM, the first pure functional language, based strongly on **lambda calculus**
- 1978 John Backus publishes FP, a functional language that emphasized **high-order functions** and calculating with programs.
- Mid1970s: Robin Milner develops ML, the first of the modern functional languages, which introduced **type inference and polymorphic types**.

- David Turner develops Miranda
- 1988: A committee of prominent researchers publishes the first definition of Haskell, a standard **lazy functional** language.
- 1999: The committee publishes the definition of Haskell98

Haskell

A Purely Functional Language

featuring static typing, higher-order functions,
polymorphism, type classes and monadic effects

Ancestor

- Lambda Calculus, Alonzo Church, 1933-1934
- $f = \lambda x. x+1 \neq g = \lambda x. (n:=x; x + 1)$

Early FL

- LISP, McCarthy, 1950s, lambda notation
- FP, Backus, 1979, higher-order functions

Modern FL

- ML, U. of Edinburgh, 1970s, Static type systems, Algebraic datatypes and pattern matching
- Miranda, David Turner, 1985, lazy evaluation

1 历史背景

2 编程范式

3 基本概念

5 真实案例

6 参考资料

声明式 (Declarative programming)

- 程序定义如何将输入转换为输出，是一个表达式
- 关注焦点: *计算什么的逻辑，而非控制流*
- SQL, Scheme, Haskell

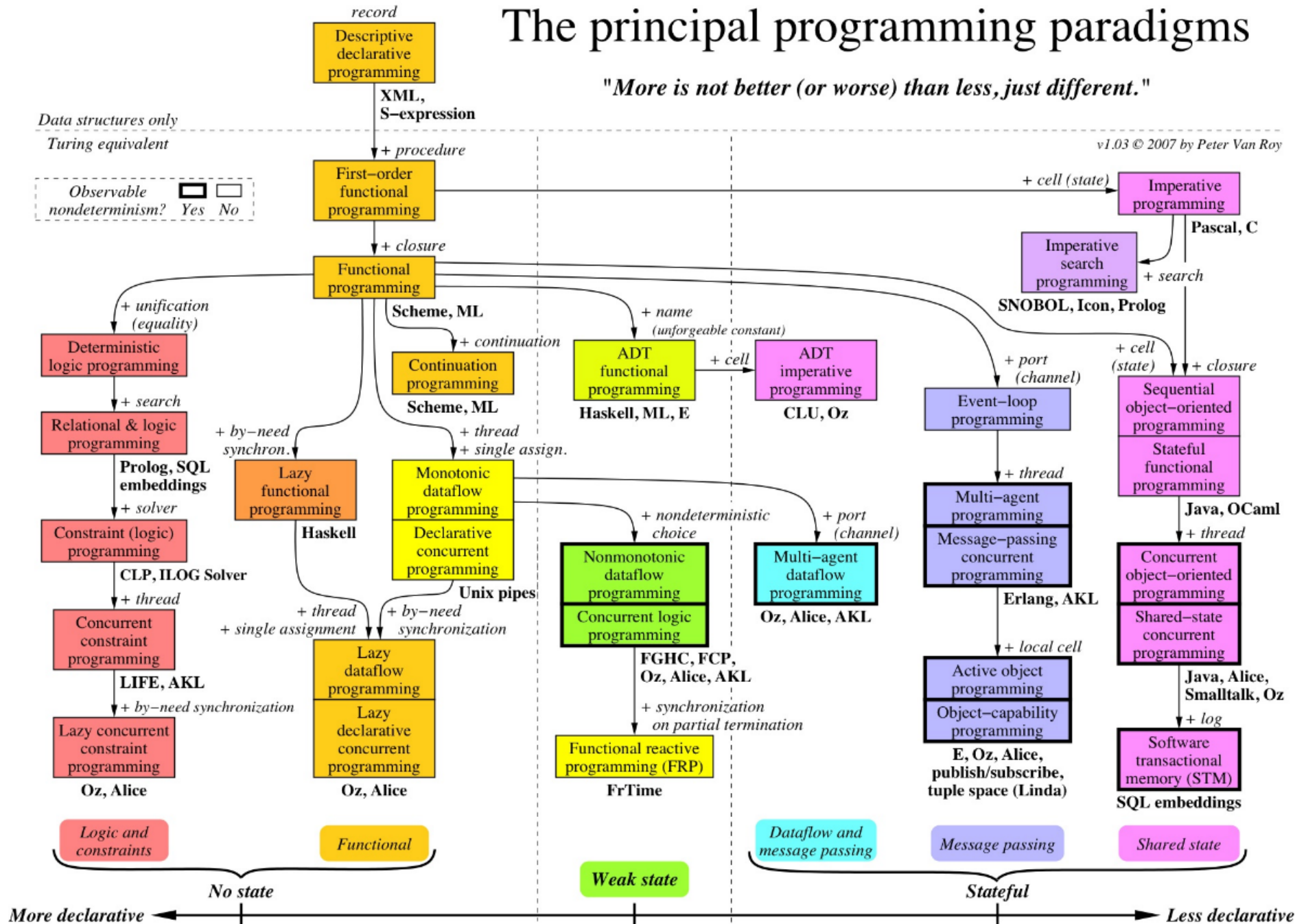
命令式 (Imperative programming)

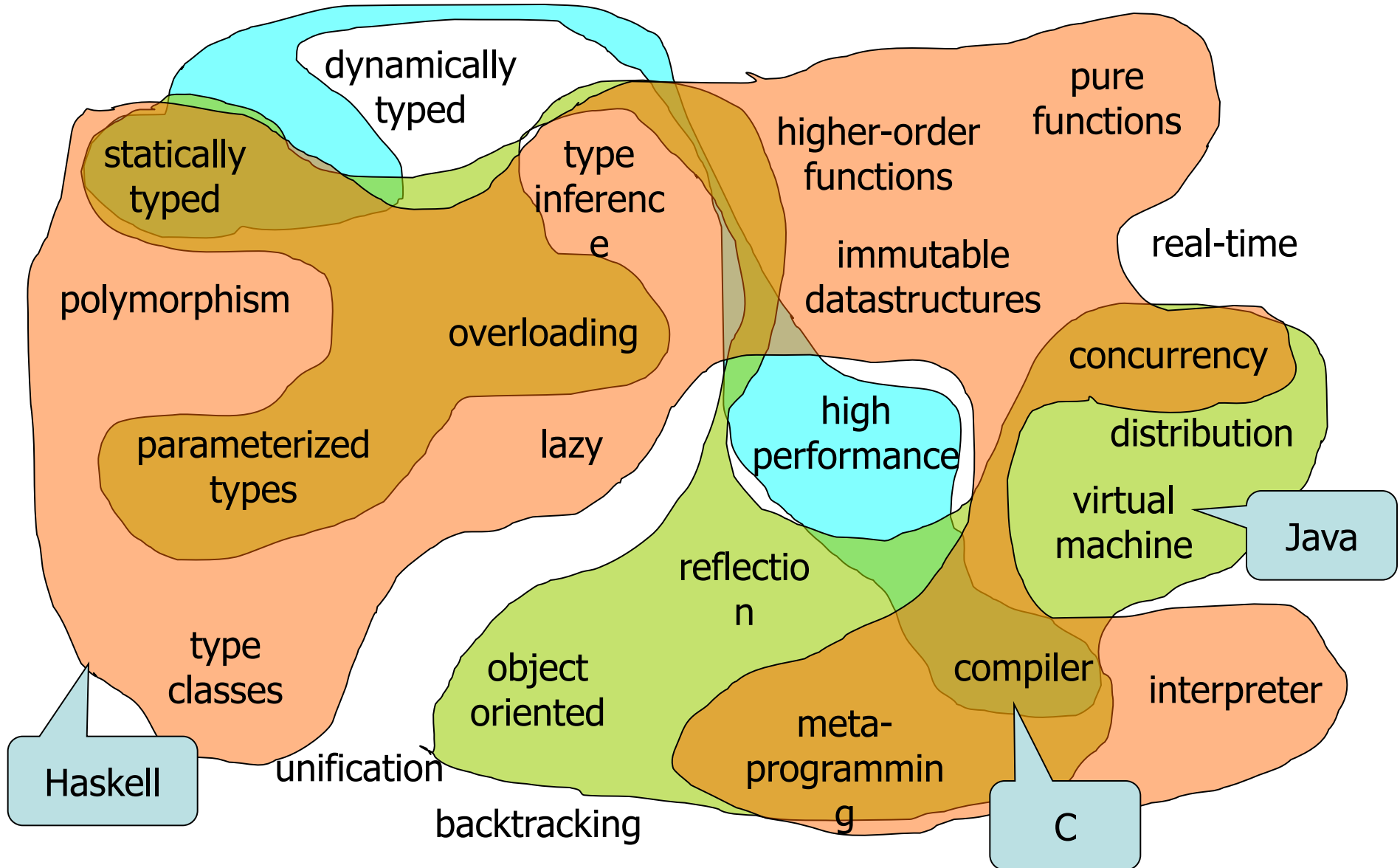
- 程序是命令的序列，运行时按顺序执行
- 关注的焦点: *如何进行计算，及其中间状态*
- C, Java, Ada and Pascal

The principal programming paradigms

"More is not better (or worse) than less, just different."

v1.03 © 2007 by Peter Van Roy





- Hot topic in PL community and industry
 - Compilers/compiler-like
 - Domain-specific languages (Haskell)
 - build *your own* programming language with little effort
 - Telecom industry (Erlang)
 - Dealing with complex protocols/data-flow
 - Need to get *right*
 - Financial industry (Haskell)
 - Dealing with complex calculations
 - Need to get *right*

1 历史背景

2 编程范式

3 基本概念

4 真实案例

5 参考资料

- 函数允许将函数作为参数传入或作为结果返回
- 类似 [first-class functions](#) 但偏重于描述数学概念

```
– class worker(object):
    def __init__(self, name = None):
        self.name = name

    def __call__(self, func):
        def wrapped(*args, **kwds):
            return func(*args, **kwds)

        wrapped.name = self.name or func.__name__
        wrapped.func = func
        wrapped.type = Worker

        return wrapped

– @worker('dns')
def dns_query(domains, dnsservers=None):
    return dict([(domain, list(dnsquery(str(domain), dnsservers))) for
domain in domains])
```

- 允许对函数参数进行逐步绑定

1. 求值函数 $f(x,y) = y / x$
2. 目标是计算 $f(2,3)$
3. 中间函数 $g(y) = f(2,y) = y / 2$
4. 进行计算 $g(3) = f(2,3) = 3 / 2$

–

```
>>> from functional import curry
>>> computation = lambda a,b,c,d: (a + b**2 + c**3
+ d**4)
>>> computation(1,2,3,5)
>>> fillZero = curry(computation)
>>> fillOne   = fillZero(1)
>>> fillTwo   = fillOne(2)
>>> fillThree = fillTwo(3)
>>> answer    = fillThree(5)
>>> answer
657
```

- 通过模板和指针的方式模拟

- `int f(int a, int b) { return a + b; }`

- `int g(int a, int b, int c) { return a + b + c; }`

- 标准C++模板函数`bind1st/bind2st`

- `std::bind1st(std::ptr_fun(f), 5)(x); // f(5, x)`

- `Boost::Bind` 模板库

- `bind(f, 5, _1)(x); // f(5, x)`

- `bind(f, _2, _1)(x, y); // f(y, x)`

- `bind(g, _1, 9, _1)(x); // g(x, 9, x)`

- `bind(g, _3, _3, _3)(x, y, z); // g(z, z, z)`

- `bind(g, _1, _1, _1)(x, y, z); // g(x, x, x)`

- 纯粹的函数式函数或表达式，没有内存或I/O的副作用 ([side effects](#))
 - 结果的表达式没被使用，可以被删除并没有影响
 - 同一函数在使用相同参数调用时，返回值结果不变。可针对计算内容进行 Memoization 优化 ([示例](#))
 - 如果两个纯表达式没有数据依赖关系，则他们的顺序可以颠倒或并行执行，等同于线程安全
 - 如果整个语言不允许副作用，则可以参与任意的求值计算策略(evaluation strategy)，编译器可以自由排列或组合表达式

- Python, 使用 lambda 表达式或 generator
 - `bigmuls = lambda xs,ys: filter(lambda (x,y):x*y > 25, combine(xs,ys))`
`combine = lambda xs,ys: map(None, xs*len(ys), dupelms(ys,len(xs)))`
`dupelms = lambda lst,n: reduce(lambda s,t:s+t, map(lambda l,n=n: [l]*n, lst))`
`print bigmuls((1,2,3,4),(10,15,3,22))`

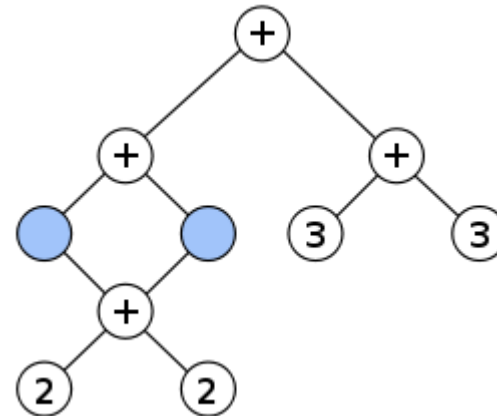
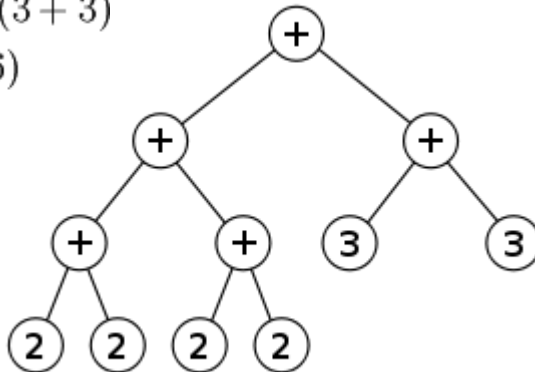
`print [(x,y) for x in (1,2,3,4) for y in (10,15,3,22) if x*y > 25]`
- GCC, 提供 [pure 属性](#)提示编译优化
 - `int square (int) __attribute__ ((pure));`

- 通过递归完成遍历(Iteration)或循环(looping)操作
- 计算 Fibonacci 序列
 - $\text{fac}(0) \rightarrow 1$;
 - $\text{fac}(N) \rightarrow N * \text{fac}(N-1)$.
- 尾递归 (tail recursion) 优化
 - 从栈调用循环函数优化为单纯循环
 - $\text{fac}(N) \rightarrow \text{fac}(N, 1)$.
 - $\text{fac}(0, A) \rightarrow A$;
 - $\text{fac}(N, A) \rightarrow \text{fac}(N-1, N * A)$.

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

- 函数语言可以根据是否进行惰性计算 (*lazy evaluation*) 来分类
- 是否对参数在使用前进行求值，严格(*Strict*)模式会直接计算参数 `print length([2+1, 3*2, 1/0, 5-4])`
- [Miranda](#), [Clean](#) 和 [Haskell](#) 等支持Non-strict模式.
- 适用于有大量中间计算节点的场景，例如 graph reduction 算法，用于AST优化或图裁剪

$$\begin{aligned}
 & ((2+2) + (2+2)) + (3+3) \\
 = & ((2+2) + (2+2)) + (6) \\
 = & ((2+2) + 4) + 6 \\
 = & (4+4) + 6 \\
 = & 8 + 6 \\
 = & 14
 \end{aligned}$$



$$\begin{aligned}
 & ((\text{blue} + \text{blue}) + (3+3)) \\
 & \quad \swarrow \quad \searrow \\
 & \quad (2+2) \\
 = & ((\text{blue} + \text{blue}) + 6) \\
 & \quad \swarrow \quad \searrow \\
 & \quad (2+2) \\
 = & ((\text{blue} + \text{blue}) + 6) \\
 & \quad \swarrow \quad \searrow \\
 & \quad 4 \\
 = & 8 + 6 \\
 = & 14
 \end{aligned}$$

- Python, Erlang等语言需要进行模拟
 - Python Generator w/ yield

```
def reverse(data):  
    for index in range(len(data)-1, -1, -1):  
        yield data[index]
```

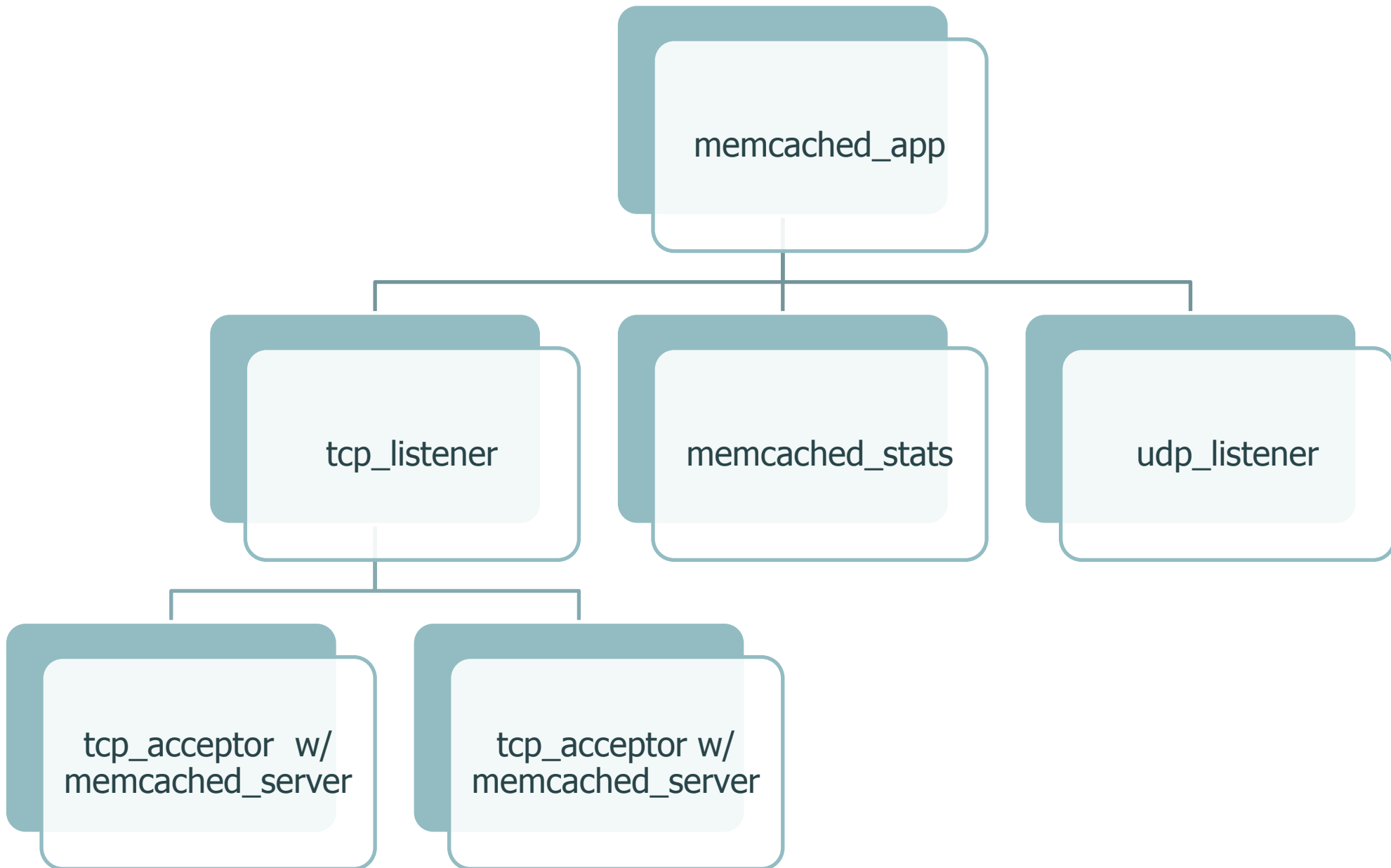
```
for char in reverse('golf'):  
    print char
```

- 强类型，编译期检查出大部分问题
 - 使用boolean类型的函数不能使用int
 - 不提供自动类型转换等Syntactic Sugar
 - 使用Pattern Matching细化类型范围
- 简单系统，只定义基础类型
 - 沿袭LISP等设计思路，强于对列表的操作
 - 没有指针等物理实现层面的概念
 - 倾向于于数学意义上函数的定义与解释

- Pattern , 一组符合特定类型的变量集
 - Name1, [H|T], {error,Reason}
- Matching , 根据Pattern定义函数适用范围
 - % parse a complete header from raw data, we assume a header ends with \r\n (0d 0a)
extract_header(<<"\r\n", Rest/binary>>, <<>>) -> {empty, Rest, <<>>;
extract_header(<<"\r\n", Rest/binary>>, Header) -> {done, Rest, Header};
extract_header(<<>>, Header) -> {incomplete, <<>>, Header};
extract_header(<<B:8, Rest/binary>>, Header) -> Header2 = <<Header/binary, B:8>>,
extract_header(Rest, Header2).

- **parse_params([], Params) ->**
- Params;
- **parse_params([{username, Value} | L], Params) when is_list(Value) ->**
- parse_params(L, Params#amqp_params{ username = list_to_binary(Value) });
- **parse_params([{password, Value} | L], Params) when is_list(Value) ->**
- parse_params(L, Params#amqp_params{ password = list_to_binary(Value) });
- **parse_params([{virtual_host, Value} | L], Params) when is_list(Value) ->**
- parse_params(L, Params#amqp_params{ virtual_host = list_to_binary(Value) });
- **parse_params([{host, Value} | L], Params) when is_list(Value) ->**
- parse_params(L, Params#amqp_params{ host = Value });
- **parse_params([{port, Value} | L], Params) when is_integer(Value) ->**
- parse_params(L, Params#amqp_params{ port = Value });
- **parse_params([{ssl_options, Value} | L], Params) when is_atom(Value) ->**
- parse_params(L, Params#amqp_params{ ssl_options = Value }).

- [RabbitMQ](#)
 - Erlang开发的开源高性能消息队列
 - 使用AMQP协议访问队列，开发较为复杂
- [Memcached](#)
 - C 开发的开源高性能缓存服务
 - 使用基于文本或二进制的[简单数据协议](#)
 - 基本上常见语言都提供完善的客户端API支持
- [Rabbitmq-memcached](#)
 - Erlang开发的开源消息队列访问代理
 - 允许使用memcache协议访问RabbitMQ
 - 支持运行在独立代理和内嵌插件模式



- A Brief History of Functional Programming
<http://www.cse.lehigh.edu/~gtan/historyOfFP/historyOfFP.html>
- Tutorial Papers in Functional Programming
<http://www.math.chalmers.se/~rjmh/tutorials.html>
- Programming paradigm, Wiki
http://en.wikipedia.org/wiki/Programming_paradigm
- Charming Python: Functional programming in Python, [Part 1](#), [Part 2](#), [Part 3](#)
- A Better Javascript Memoizer
<http://unscriptable.com/index.php/2009/05/01/a-better-javascript-memoizer/>
- An Analysis of Lazy and Eager Evaluation in Python
<http://www.pages.drexel.edu/~kmk592/rants/lazy-python/index.html>



谢谢！