编程之思想
抽象

我们写程序。我们写下用于表达我们想法的代码，然后借助于编译器，将这些符号的有机组合翻译为机器可以听懂的二进制指令并交与执行。[翻页]

程序设计是一种有关语言运用和思想表达的艺术。要成为编程大师，需要掌握所有的编程语言么？当然不！著名的图灵等价论断说：语言是对思想模型的反映，如果模型是一致的，那么语言便是等价的。不同于我们采用的自然语言，进行编程的程序设计语言是形式化的。形式化语言有一个根本的推倒规则，而语言整体由这个规则发衍而来。

程序员关心程序设计语言的效率，因此业界往往争论究竟是哪个语言更快。从这点来说，我很赞同王垠同学提出的观点：程序的效率之关键有如下亮点：

1、程序是否采用了最好的算法；
2、运行效率全在于编译器\解释器是否能产生高质量的目标代码；

第一点，显然所有语言都可以使用最好的算法，因此，算法这一点就没有一门语言占优势。而第二点，我直接借用王垠同学提出的例子：在历史上，Lisp 语言享有“龟速”的美名。有人说“Lisp 程序员知道每个东西的值，却不知道任何事情的代价”，讲的就是这个事情。但这已经是很久远的事情了，现代的 Lisp 系统能编译出非常高效的代码。比如商业的 Chez Scheme 编译器，能在 5 秒钟之内编译它自己，编译生成的目标代码非常高效。它可以直接把 Scheme 程序编译到多种处理器的机器指令，而不通过任何第三方软件。它内部的一些算法，其实比开源的 LLVM 之类的先进很多。

所以，争论语言的运行效率是毫无意义的。把这个争论留在编译器设计和优化上，可能会好一点。[翻页]

那么作为思想的形式化表达的抽象，语言所代表的思想呢？就如 SHELL 来说，你虽然知道“管道”这个操作是一个 trick，但是这个 trick 给你带来的线性编程体验是其它语言无法比拟的。所以，好的思想可以减少建模的复杂度。命令式语言大行其道，但是在 GUI 开发方面，面向对象语言似乎更加方便。

本次讨论课，我们会对不同的语言进行讨论。我们不是讨论语言本身的好坏，而是讨论它所蕴含的思想，昭示的哲理。我将探讨“抽象”究竟是什么，“抽象”的目的是什么，以及“抽象”的方法是什么。最后，如果可能的话，我们会讨论如何用一门语言去生成另一门语言，这是一种“元语言抽象”的元编程技术。[翻页]

就像哲学是什么，这是一个很哲学的问题一样。抽象是什么，抽象本身就是一个很抽象的东西。[翻页]

首先，我想给大家展示几幅画。这是苏联画家埃尔·利西茨基在 1922 年绘制的 Proun 19D。我想先听听大家对这幅画的看法。

- 询问大家对这幅画的看法 -

[翻页]第二幅是荷兰画家皮特·蒙德利安在 1942 年绘制的《百老汇爵士乐》。这幅就更加抽象了、只是通过矩形、色彩、布局来表达自己的情感。我觉得这幅画是作者表达爵士乐带来的感受。[翻页]最后一幅，也是最出名的一幅。来自于法国画家让·米勒，这是他在 1857 年完成的《拾穗者》。这就是一幅很写实的画了。我想请大家来描述一下。[互动]

前两幅图都是抽象派画作。就如大家的感觉一样，我们无法直观的描述出抽象画描述的是什么，因为它描述的都不是实在的、可感知的实物。而最后的一幅写实派的《拾穗者》，却是对一个实在场景的描述，我们能很容易感受到它。

在美术中，我们就把那些实在的东西叫做“具象”，“具象”是一个具体的东西。而“抽象”就是与这些“具象”相对的东西了。很多人认为抽象

画不美、不应该算是艺术，因为它只注意了具象的东西。我们的感官太强大了，这加强了我们对具象的体会，而忽视了抽象的思考。当然，杰出的数学家必须有相当的抽象思维。

那么，编程中的抽象是什么呢？[翻页]

我们在先来谈谈语言形式对思想的抽象。这里不得不提到近代数学对抽象化的杰出贡献。我以为，皮亚诺、莱布尼兹等先贤提出并发展的数理逻辑是人类思想的伟大进步。这种符号化系统就是对具象的一个高度抽象。例如，一个命题的蕴含式可以这样抽象：

p 蕴含 q

对于一个具体的事，我们可以给 p 和 q 这两个无意义的“符号”赋予实际意义，譬如说：“如果 Y 的平方等于 X，那么 Y 就是 X 的平方根。”对应的。p 代表“Y 的平方等于 X”这个断言，而 q 则是“Y 是 X 的平方根。”

逻辑学家们就抓住了这个关键点，发明了谓词、时态逻辑、模糊逻辑等形式化系统，让我们的思想不再聚焦在一个具体的案例上。而是去思考它们的普遍模式。例如，这个就是我们所谓的“数学归纳法”的一个形式化表述。而下面则是表示一个谓词具有传递性的形式化表述。[翻页]

我们再来看看语言的不同书写形式所具有的语义（也就是所谓的表达力）。

- 你难道不喜欢看电影么？ - 不，我喜欢。
- Don't you love watching movie? Yes, I do.

两句话都想说：“我确实喜欢看电影”但从表面上的语义来看，两者的表达完全相反。

因此，从不同方面对思维进行抽象，就会表现出不同的形式，就有不同的表达力。就算书写的方法不同，所拥有的抽象表达力也是不同的。有几个

个性鲜明的例子：Pascal、Lisp。

Lisp 最引以为傲的是其前缀记法也就是所谓的 S-表达式（Symbolic Expression）。让我们回到数学记号中来。[翻页]我们有中缀记号，比如我们最熟悉的四则运算。我们有前缀记号，比如非运算，或者映射。我们有后缀记号，比如阶乘。排列和组合的记号还同时使用了上下标。极限记号不但要书写 lim 这三个字符，还有区域某个值的变量。记号太多了，记法也太复杂了，我们计算机没办法用简单的方式表达这些复杂的算法。但数学家们想了想，说：“嘿！这些记法所表示的都是函数，都是一个运算过程。我们可以用一个符号来象征这个函数，这些运算，我们把这个叫作运算符。而我们的计算都可以视为‘将运算应用到被运算的对象’上。”

换句话说，加法运算 a+b 表示的是将+所代表的运算运用到 a 和 b 两个对象上。很好，我们就可以记为：

```
(+ a b)
```

另外，如果+是变元的，我们直接线性的增加他的参数个数就行了。这种记法的中心思想是：将运算符、运算对象的组合起来，所以 S-表达式有时候也称为组合式（Combination）。要想确定是什么运算，我们对 S-表达式的第一个元素求值就好了。当然，“运算符”可能不是原子的，这个灵活的特性使得我们可以完成这种代码：

```
((if (> x y)
     +
     -) x y)
```

用对我们友好一点的 C 语言可能会这样描述：

```
if (x > y)
  return x + y;
else
  return x  −  y;
```

究竟那种写法更强大呢？我们后面会深入讨论。

S-表达式某种程度上来说是裸语法的，因为它相当于是一棵平面线性AST，也就是我们所谓的抽象语法树（Abstract Syntax Tree）。[翻页]

这里，我们有一个数学记法的算术式，右边是它对应的抽象语法树，而左下是Lisp风格的S-表达式的记法。我们看到S-表达式清晰的表达了运算结点。这样做也使得我们很容易弄明白运算顺序。在数学和其它编程语言中，运算优先级通常是由括号确定的，而Lisp则是由它的层次结构。

换句话说，其它语言中，添加括号只会使运算顺序变得更清晰，而Lisp中，添加括号则是说，扩起来的东西是个组合式。那么：

[解释图]

((+ 3 5))并不会按照你预想的返回8，而是会产生一个错误。因为最里面的那个表达式返回8，而8又被外部的括号括起来。(8)就成了一个组合式。我们对这个组合式求值，按照Lisp的规则，第一个元素是运算符——一个有效过程。然而8既不是一个有效过程，所以报错。但在诸如C语言或者Pascal语言中，你可以随意添加括号：
(((((((5+3)))))))

而相对的，平时习见的编程语言更注重思想的结构化表达。让我们来看看Pascal语言的程序吧！

Pascal语言将程序结构化设计贯彻得是如此的彻底：

这一块是程序说明，这一块是变量声明。这是子程序，这是函数定义，这是while-循环。非常清楚。然而，这儿有一段SCIP题解的Scheme代码，其实他很简单，但我还是想说：“God Bless Lisper……”

因为S-表达式直接就是一棵二维裸语法树，所以某种程度上S-表达式是反

人类的，因为相对于结构化语句来说，S-表达式并不直观，在习惯S-表达式之前，我们阅读起这些代码并不轻松。然而它是最美的、形式最简洁的。因为我们很难找找到更细化、更简洁的表达方式了。所以，抽象的一个要点，就是奥卡姆的剃刀：“因不超过果之所需。”

我们再来看看不同的形式。就如同我们刚才谈到的SHELL语言所支持的管道操作一样，有时候我们需要线性的顺序表达我们的思维。这样说吧：让我们生成一个20个元素的随机数组（元素的范围在0～9），无重复的倒序输出其中的偶数（例如，如果出现了两次2，只输出一次即可）。我用了Ruby来完成这个操作（面向对象语言几乎都可以这样）。

```
Array.new(20).map{rand(10)}.uniq.sort.select{|e| e%2 == 0}.reverse.inspect
```

而用管道的思想，这个程序或者会像下面这样：

```
mka 20 -r 10 | uniq | sort | select '$1 %2 == 0' | reverse | print
```

而如果要写成C语言呢？这就有点恐怖了：

```
print(reverse(select(sort(uniq(newa(20), bd_fun)),sel_fun)))
```

从这一点来说，在表达线性的顺序思维时。方法链、管道，这种合乎人类思维习惯的表达方式有时候是很方便的，它减小了我们的编码痛苦也避免了我们思维的打断。命令式无法实现类似于方法链这样的表现行为，是因为它的基本范式是：
operate(obejct)。整个返回值依旧会作为object嵌套在另一个operate中。

这里，我们就又看到了不同思维的抽象表现在书写形式上的不同。这一部分的内容，很大程度上来自于我对程序设计语言风格的探索。Backus大神的图灵演讲论文就提出了这个疑问：编程能从冯·诺依曼体系中得到解放么？大神给出的回答是函数式编程。但除了这两者，我们还有其它的编程思想么？这个就期待在座的各位大神了。好了，这部分有什么问题么？
[Q&A]

我们再来谈谈计算过程的抽象。首先我们不得不提及三个著名的计算模型：

1. 图灵机
2. 冯·诺依曼机
3. lambda 演算

当然，如果要深入讨论的话，我们又会有很多可以讨论的话题，所以就直接谈论它们的要点。与 lambda 演算相比，图灵机和冯·诺依曼机可以算作一类，我们可以理想化的看成图灵给出了理论基础，而冯·诺依曼给出了具体的实现。另外，计算机科学家们已经证明了图灵-冯·诺依曼模型和 lambda 演算是等价的。简单的说图灵-冯·诺依曼模型依赖的是状态的变换，而 lambda 演算依赖的是函数的演算。

图灵-冯·诺依曼机中有留有某些用于存储数据的状态。比如我们对同一个函数进行连续调用可能会产生不同的结果，因为在函数内部我们可能会更改一些全局状态。甚至在局部，调用的顺序不同，也可能会产生不同的结果。一些 C 语言新手会写出下面的代码来实现平方：

```
#define square(x) ((x) * (x))
```

有经验的 C 语言程序员会告诉他，这样做是不行的，因为当调用 square(++n)的时候，你得到的不是期望的值，因为执行第一个++n 以后，n 的状态就被改变了。基于状态的计算模型太依赖于时序了，所以要实现并发比较麻烦。

lambda 演算所代表的计算模型就是函数演算。函数演算要确保，对一个表达式的求值是不会有副作用的。并且，如果采用惰性求值，我们可以对表达式进行规约：类似于 double(double(double(x)))的调用，我们就可以规约为将 x 翻为 6 倍。并且，我无论是在程序的开始还是结束还是在任何时候，只要 x 的值没有发生变化，那么这个表达式的值是不会发生变化的。

在我们平常习见的程序设计语言中，函数并不能够直接作为参数传递。例

如在C语言中，调用qsort()函数就要传递一个指向用于比较的cmp()函数的指针，间接实现“传递函数”的功能。当然，函数式编程语言中直接把函数升级为“第一级”状态，这样函数就跟普通变量一样，可以命名，可以作为输入或输出，甚至也可以包含在数据结构中。

ANSI C中没有构造匿名函数的功能，有趣的是，与之相对的lambda演算中，函数定义全是匿名的。在Lisp中，我们用lambda来构造一个匿名过程。

```
(lambda (x y)
  (+ (* x x) (* y y)))
```

（可能把lambda用make-procedure来表达更容易让人明白，但是lambda是历史沿用下来的术语）

如果，我们要给这个过程取一个名字，那么可以用define来将这个过程与一个名字绑定起来：

```
(define sum-of-square
  (lambda (x y)
    (+ (* x x) (* y y))))
```

或者直接用define的语法糖来定义：

```
(define (sum-of-square x y)
  (+ (* x x) (* y y)))
```

所以我们可以写出下面的程序：

```
((lambda (x) x) 5)
; 这是一个恒等映射，返回元素本身
```

函数式语言允许我们将函数当作参数传递，比如下面这段代码：

```
((lambda (f x) (f x))
```

```
  (lambda (y) (* 2 y))
  5)
```

Lisp 中有一个叫作 map 的高阶过程。你可以把他看作给定定义域和映射关系，返回值域。这里的域都是通过列表来实现的：

```
(map (lambda (x) (+ x x)) (list 1 2 3 4 5 6))
; (2 4 6 8 10 12)
```

有了这些知识以后，我们再来谈谈数列求和。高斯利用等差数列求和 100 及以内的正整数的故事已经被滥用了。而到了计算机时代，我们可以简单的暴力枚举求和，所以，学习 C 语言的时候，我们会被要求用 for-语句或者 while 语句写个用于计算此类问题的程序。当然，我相信在坐的各位都会写这么一个程序。在这里，为了某些方便，我们这里将其改写为 Lisp 代码：

```
(define (sum a b)
  (if (> a b)
      0
      (+ a
         (sum (+ a 1) b))))

(display (sum 1 100))
```

对于 Leibniz 定理，我们也可以暴力加和它的前 N 个有限项；

```
(define (pi-sum b)
  (define (lc-pi-sum x y)
    (if (> x y)
        0
        (+ (- (/ 1.0 x)
              (/ 1.0 (+ x 2)))
           (lc-pi-sum (+ x 4) y))))
```

```
 (* 4 (lc-pi-sum 1 b)))
```

我们现在已经有两个模式了，简单来比较一下，我们来提取一下它们的公共模式。首先，都是从一个数开始进行加算，停止于另一个数。然而它们的步进不一样，前者是1，后者是4，但我们还是折衷的认为它们有共同点，就是：“它们都有步进”。因此，我们对这种公共模式进行抽象，可以得到这么一个模板：

```
(define (<name> a b)
  (if (> a b)
    0
    (+ (<term> a)
       (<name> (<next> a) b))))
```

数学家们早就注意到了这种抽象模式，所以它们发明了“求和记号”。这使得我们能够去处理求和概念本身，而不去依赖具体的项，或者如何步进。我们用 Lisp 将这种思想形式化的表达出来，或许是这样：

```
(define (general-sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (general-sum term (next a) next b))))
```

我们来用 general-sum 过程来编写我们的加和 1 到 100 的过程。

```
(define (sum-1-to-100)
  (general-sum (lambda (x) x)
               1
               (lambda (y) (+ y 1))
               100))
```

作为对照，这样写可能更清楚一点：

```
(define (sum-1-to-100)
  (define (indent x) x)
  (define (next x)
    (+ x 1))
  (general-sum indent 1 next 100))
```

利用 general-sum 来编写 Leibniz 定理的方法如出一辙，这里就不在赘述了。但是，如果你认为抽象到了这种程度就是极限了，那么你就错了，让我们仔细审视一下：我们关心序列和的求和记号，也有关心序列积的记号。关于序列且、序列或、序列交、序列并，等等等等。这些都是对某一集合封闭的二元运算运用于所有元素所产生的计算过程。因此，我们还可以对这个二元运算符进行抽象，就可以把我们的 general-sum 过程改写为 general-operate。所以，永远别以为我们到了最顶层（或最低层）的抽象，有的时候你观察的角度不一样，看到的自然也不一样。

顺带一提，scheme 中提供了一个叫 reduce 的方法，实现了和我们的 general-operate 类似的功能。Reduce 要求给出二元关系 和一个序列，并将其应用于所有的序列。

说了这么多比较“抽象”的，再来说一个比较“具象”一点的吧！这种提取公共计算模式的抽象思想演化成了一种叫作“模板方法”的设计模式，尤其广泛的应用于面向对象编程中。这是使得子类可以从父类继承定义好的方法，于是程序员通常在父类中构造好框架，在子类中进行具体实现。

好了，我们今天探讨了对语言形式和对计算过程的抽象，我们可以发现抽象的一个基本想法：照出公共模式，给它们构建最精简的表述模型。后面我们还会看到抽象用于数据时具有的效果，抽象如何隐藏细节，抽象如何将复合数据包装得让他看起来像初等数据，抽象是如何提供接口的。最有趣的，我们将看到一种叫作“鸭子类型”的用于解决无类型变量的技术。这些将在我们将在后面讨论。Any Question?