1. 构造过程抽象(1)

本次课讨论基本 Scheme 程序设计，重点是构造过程抽象

- 基本表达式，命名和环境
- 组合式的求值
- 过程的定义
- 复合过程求值的代换模型
- 条件表达式和谓词
- 过程抽象
- 内部定义和块结构

Scheme

- Scheme 是交互式编程语言，其解释器运行时反复执行一个“读入-求值-打印循环”（Read-Evaluate-Print Loop，REPL）。每次循环：
  - 读入一个完整的输入表达式（即，一个程序）
  - 对其进行求值（计算），得到一个值（还可能有其他效果）
  - 输出求得的值（也是一个表达式）
- 在 Scheme 里编程就是构造各种表达式
- Scheme 的功能由三类编程机制组成：
  - 基本表达式形式，是构造各种程序的基础
  - 组合机制，用于从较简单的表达式构造更复杂的表达式
  - 抽象机制，为复杂的结构命名，使人可以通过简单方式使用它们
- 任何足够强大的编程语言都需要类似的三类机制
- 常可区分“过程”（操作）和“数据”，本章主要研究过程的构造

简单表达式

入门 Scheme 的最直接方式是看一些简单表达式计算：

- 数是基本表达式

```
> 235
235
```

- 简单算术表达式（简单组合式）

```
> (+ 137 248)
385
> (+ 2.9 10)
12.9
```

- Scheme 表达式统一采用带括号的前缀形式，括号里第一个元素表示操作（运算），后面是参数（运算对象）。

  运算符和参数之间、参数之间都用空格分隔

简单表达式

- 有些运算符允许任意多个参数

```
(+ 2 3 4 29)
(* 3 7 19 6 3)
```

- 表达式可以任意嵌套

```
(+ 2.9 (* 15 10))
```

- 可以写任意复杂的表达式（组合式），如

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

复杂表达式容易写错。采用适当格式有利于正确书写和阅读：

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

子表达式之间加入任意的换行和空格不影响表达式的意义

命名和环境

- 实用的编程语言必须提供为计算对象命名的机制，这是最基本的抽象机制。这里把名字标识符称为变量，其值就是与之关联的对象
- Scheme 里通过 define 为对象命名，如：

  > (define size 10)

  此后就可以用 size 引用相关的值，如：

  > size

  10

  > (* size 3)

  30
- 可用任意复杂的表达式计算出要求关联于变量的对象：

  (define num (* size 30))

  这使 num 的值是 300

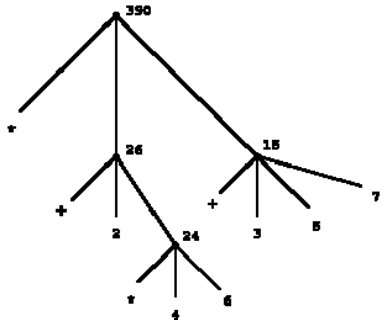
命名和环境

- 计算对象可能有任意复杂的结构，可能是经过复杂而费时的计算得到
  - 每次需要用时都重新计算，既费时又费力
  - 给一次计算得到的结果命名，能方便地多次使用
- 构造复杂的程序，最终是为构造出复杂的不易得到的对象。通过逐步构造和命名，可以分解构造过程，使之可以逐步地递增地进行。建立对象与名字的关联是这种过程中最重要的抽象手段
- 能把构造出来的值存入变量供以后取用，说明 Scheme 解释器有存储能力。这种存储称为“环境”，表达式是在环境中求值
  - define 建立或修改环境（全局环境）中名字与值的关联
  - 表达式总在当前环境中进行求值（后面将详细讨论环境概念），变量的值由环境中获得
  - Scheme 系统的全局环境里预先定义了一批名字-对象关联（预定义的对象），主要是预定义的运算（过程）

组合式的求值

- 一般而言，需要求值的是一个组合式，解释器的工作过程是：
  - 求值该组合式的各子表达式
  - 将最左子表达式的值（运算符的值，应是一个过程）作用于相应的实际参数，即由其他子表达式求出的那些值
- 上述规则说明计算过程的一些情况：

  例：(* (+ 2 (* 4 6)) (+ 3 5 7))
  - 组合式求值要求先求值子表达式。因此求值过程是递归的
  - 求值过程可以用树表示，首先得到终端结点的值，而后向上累积
  - 最终在树根得到整个表达式的值
  - 树是递归结构，用递归方式处理很自然



组合式的求值

- 求值中的递归最终将到达基本表达式，这时可以直接得到值：
  - 数的值就是其自身（它们所表示的数值）
  - 内部运算符的值是系统中实现相关运算的指令序列
  - 其他名字的值在当前环境里查找，如果有相应的名字-值关联，取出对应的值作为该名字的求值结果
- 后两种情况可以统一：基本运算符（如 + 和 *）和其他预定义对象（如 define）都看作名字，在环境中查找它们的关联
- 环境为程序里使用的名字提供定义。如果求值中遇到一个名字，在当时环境没有它的定义，解释器将报错
- 求值规则有例外，如 (define x 1) 中的 x 不求值，本表达式要求为名字 x 关联一个值。这说明 define 要求特殊的求值规则

  要求特殊求值规则的名字称为特殊形式（special form）。Scheme 有一组特殊形式，define 是其中一个。每个特殊形式有特殊的求值规则，下面会看到其他特殊形式

过程定义

- 表达式可能很长，复杂计算中常要写重复或类似表达式。为控制程序复杂性，需要过程描述的抽象机制，在 Scheme 里是“过程定义”
- 求平方过程的定义：

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

包括：过程名，参数，该过程做什么（如何求值）。求值这个定义表达式，将使相应计算过程关联于名字 square

- 定义好的过程可以像基本操作一样使用：

```
> (square (* (+ 3 7) 10))
10000
> (+ (square 3) (* 20 (square 2)))
89
> (define (sum-of-squares x y)
          (+ (square x) (square y)))
```


过程定义

- 新定义的 sum-of-squares 又可以像内部操作一样用

```
> (sum-of-squares 3 4)
25
(define (f a)
  (sum-of-squares (+ a 1) (* a 2)))
(f 5)
136
```

- 预定义基本过程（操作）和特殊形式是构造程序的基本构件

  编程中根据需要定义的过程扩大了这一构件集合

  只看 square 和 sum-of-squares 的使用，完全看不出它们究竟是基本操作还是用户（程序员）定义的过程（复合过程）

  复合过程具有和基本操作一样的使用方式和威力，是很好的语言特征

- 过程定义是分解和控制程序复杂性的最重要技术之一

过程应用的代换模型

- 考虑过程的应用（求值）。假定解释器实现了将基本运算应用于实际参数的功能，复合过程确定的计算规则是（代换模型）：

  用实际参数取代（代换）过程体里的参数，而后求值过程体

- 例：

  (f 5)　　用原过程体 (sum-of-squares (+ a 1) (* a 2))，代换得到

  (sum-of-squares (+ 5 1) (* 5 2))　　求值实参并代入过程体，得到：

  (+ (square 6) (square 10))　　求值实参并代入过程体，得到：

  (+ (* 6 6) (* 10 10))

  (+ 36 100)

  136

- 代换模型给出了过程定义的一种语义。很多 Scheme 过程的行为可以用这一模型描述。后面会看到，更复杂的过程需要扩充的语义模型

过程应用的代换模型

注意：

- 代换模型只是为了帮助理解过程应用，它并没有反映解释器的实际工作过程（只是为了初步的直观理解）
- 解释器是基于环境实现的，后面有进一步讨论
- 本课程后面部分将研究解释器的工作过程的一些模型，代换模型是其中最简单的一个（最容易理解）
- 代换模型有很大局限性，它不能解释带有可变数据的程序。描述带有可变数据的程序需要更精细的模型

应用序和正则序求值

- 前面说解释器先求值子表达式（运算符和各运算对象），而后把得到的运算应用于运算对象（实际参数）。这很合理，但合理的做法不唯一
- 另一可能方式是先不求值运算对象，实际需要用时再求值。按这种方式对 (f 5) 求值得到的计算序列是先展开：

```
(sum-of-squares (+ 5 1) (* 5 2))
(+   (square (+ 5 1))    (square (* 5 2)) )
(+   (* (+ 5 1) (+ 5 1))  (* (* 5 2) (* 5 2)))
```

而后归约

```
(+   (* 6 6)              (* 10 10))
(+   36                   100)
136
```

- 前一方式（先求值参数后应用运算符）称为应用序求值，后一方式（完全展开之后归约）称为正则序求值。Scheme 采用应用序求值

条件表达式和谓词

- 描述复杂的计算时，需要描述条件和选择
- Scheme 有条件表达式。绝对值函数可定义为：

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

条件表达式的一般形式：

```
(cond (<p1> <e1>)          依次求值各个 p（条件），遇到第一个非
      (<p2> <e2>)          false的条件后求值对应的 e，以其值
      ...                  作为整个cond 表达式的值
      (<pn> <en>))
```

- 绝对值函数还可定义为：

```
(define (abs x)
   (cond ((< x 0) (- x))
         (else x)))        else 表示永远成立的条件
```


条件表达式和谓词

- 另一简化的常用条件表达式形式：

(if <*predicate*> <*consequent*> <*alternative*>)

cond 和 if 都是特殊形式，有自己特定的求值规则

- 逻辑组合运算符 and 和 or 也是特殊形式，采用特殊求值方式

(and <e_1> ... <e_n>)

逐个求值 e，直到某个 e 求出假，或最后一个 e 求值完成。以最后求值的那个子表达式的值作为值

(or <e_1> ... <e_n>)

逐个求值 e，直到某个 e 求出真，或最后的 e 求值完成。以最后求值的那个子表达式的值作为值

(not <*e*>)　　如果 e 的值不是真，就得真，否则得假

- 求出真假值的过程称为谓词，各种关系运算符是基本谓词

过程定义实例：牛顿法求平方根

- 过程很像数学函数，重要差异是必须描述一种有效的计算方法
- 在数学里平方根函数通常采用说明式的定义：

$$\sqrt{x} \text{ is the } y \text{ such that } y \geq 0 \text{ and } y^2 = x$$

基于它写出的过程定义无意义（没给出计算平方根的有效方法）：

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

- 牛顿法采用猜测并不断改进猜测值的方式，一直做到满意为止。例如选初始猜测值 1 求 2 的平方根（改进猜测值的方法是求平均）

```
1        (2/1) = 2            ((2 + 1)/2) = 1.5
1.5      (2/1.5) = 1.3333     ((1.3333 + 1.5)/2) = 1.4167
1.4167   (2/1.4167) = 1.4118  ((1.4167 + 1.4118)/2) = 1.4142
1.4142   ......
```

继续这一过程，直至结果的精度满足实际需要

牛顿法求平方根

- 用 Scheme 实现：
  - 从要求开平方的数和初始猜测值 1 开始
  - 如果猜测值足够好就结束
  - 否则就改进猜测值并重复这一过程
- 写出的过程：

```
(define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x)
                   x)))
```

- 改进方式是求出猜测值和被开方数除以猜测值的平均值

```
(define (improve guess x)
    (average guess (/ x guess)))
```


牛顿法求平方根

- 求平均很简单。还需决定“足够好”的标准。例如：

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

- 用 sqrt-iter 定义 sqrt，选初始猜测（这里用 1）：

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
```

- 一些试验：

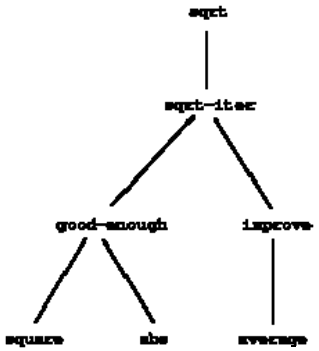
```
(sqrt 9)
3.00009155413138
(sqrt (+ 100 37))
11.704699917758145
(sqrt (+ (sqrt 2) (sqrt 3)))
1.7739279023207892
(square (sqrt 1000))
1000.000369924366
```

- 牛顿法是典型的迭代式计算过程，这里用递归方式实现
- 定义了几个辅助性过程，利用它们把一个复杂问题分解为一些更容易控制的部分
- 每个过程都有明确逻辑意义，可以用一句话明确说明

过程作为黑箱抽象

重新考察 sqrt 过程的定义，希望从中学到一些东西

- 首先，它是递归定义的，基于其自身定义。需要思考这种“自循环定义”是否真有意义，后面将详细讨论这一问题
- sqrt 分成一些部分实现，每项工作用一个独立过程完成，反映原问题的一种分解
- 分解合理与否的问题值得考虑
- 定义新过程时，把用到的已有过程看作黑箱，不关心其细节实现，只关注其功能。
- 例如，需要用求平方过程时，任何能计算平方的过程都可以用



过程作为黑箱抽象

- 只考虑功能（做什么）时，不能区分下面两个定义：

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

(define (square x)
    (exp (double (log x))))

(define (double x) (+ x x))
```

- 过程抽象的本质：
  - 定义过程时，关注所需计算的过程式描述细节（怎样做），使用时只关注其说明式描述（做什么）
  - 过程总（也应该）隐藏起一些实现细节，使用者不需要知道如何写就可以用它。被用过程可能是其他人写的或系统库提供的
  - 过程抽象既是控制和分解程序复杂性的手段，也是记录和重用已有开发成果的单位。其他抽象机制都有类似作用

过程抽象：局部名字

- 过程隐藏的最简单细节是局部的名字。下面两个定义无区别：

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

(define (square y) (* y y))
```

- 过程的形参在过程体里有重要作用：
  - 具体名字不重要，重要的是哪些位置用同一个形参
  - 形参是过程体的约束变量（来自数理逻辑的概念），其作用域是整个过程体；其他名字是自由的。约束变量统一换名不改变意义
- 重看过程 good-enough? 的定义：

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

其中用到的 x 必然与 square 里的 x 不同，否则上述过程执行时不可能得到需要的效果

过程抽象：局部名字

- 在 good-enough? 的定义里：

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

guess 和 x 是约束变量，而 <，-，abs 和 square 是自由的。保证 good-enough? 意义正确，就要保证两个约束变量（形参）名字与四个自由变量不同，且这四个自由变量（在环境里关联）的意义正确

- 形参与过程体里的自由变量重名将导致该自由变量被“捕获”，例如，下面函数的意义变了（错了）：

```
(define (good-enough? guess abs)
  (< (abs (- (square guess) abs)) 0.001))
```

- 自由变量（名字）的意义由运行时的环境确定，它可以是
  - 某个内部过程或复合过程，过程里需要应用它
  - 或者一个有约束值的变量，过程里需要它的值

过程抽象：内部定义和块结构

- sqrt 的相关定义包括几个过程：

```
(define (sqrt x)
  (sqrt-iter 1.0 x))
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
(define (improve guess x)
  (average guess (/ x guess)))
```

其中 abs 和 average 是通用的，可能在其他地方定义。

- 注意：使用者实际上只关心 sqrt，其他辅助过程出现在全局环境中只会干扰人的思维和工作（例如，不能再定义另一个同名函数）
- 写大型程序时需要控制名字的使用，控制其作用范围（作用域）

过程抽象：内部定义和块结构

- 局部于一个过程的东西应该定义在过程内部。Scheme 的做法：

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

- 这种嵌套定义形式称为块结构（block structure），是早期的重要语言 ALGOL 60 引进的概念
- 块结构是组织程序的一种重要手段。C 语言不支持局部函数定义（基于其他考虑），这种规定限制了 C 语言的程序组织方式

过程抽象：内部定义和块结构

- 函数定义局部化使程序更清晰，减少了非必要的名字污染环境，还可能简化过程定义：局部过程在形参（x）的作用域里定义，可以直接用 x（不必再作为参数传递）
- 按这种观点修改后的 sqrt 定义：

```
(define (sqrt x)
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
        guess
        (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```

- 块结构对控制程序的复杂性很有价值
- 各种新语言的设计中都为程序组织提供了许多专门的机制

关注

- 表达式
- 自由变量和约束变量
- 环境和变量的求值（这个问题后面还会讨论，这里说的是简单情况）
  - 约束变量的值就是它的约束值（实际参数值）
  - 自由变量的值是它在环境里关联的值
- 作用域
- 局部定义和块结构
- 过程抽象，技术和意义
- 简单求值过程：代换模型

有什么问题？