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
- 与 C 语言机制的比较，相关讨论

Scheme 基础

- Scheme 是交互式语言，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）
- Scheme 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

与 C 语言对比

- C 是一个编译型语言
  - 程序有完整的结构，简单的表达式或语句并不构成完整程序
  - 编制好的程序需要经过编译后才能投入运行
- 从语言的结构看，C 语言有
  - 描述基本计算的表达式
  - 描述基本动作的语句
  - 语句之上的各种组合机制（描述控制流）
  - 函数是语言里的抽象机制，用于把一段可能很复杂的计算抽象为一个简单形式的命令
- C 语言严格地区分了“数据”和操作数据的“过程”（代码）
  - 后面将看到，在 Scheme 里，数据和过程（代码）可以自然的来回转化：数据可变为被执行的代码，代码可以作为被处理的数据

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（> 表示提示符）

  > 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

C 语言的表达式

- C 语言的表达式采用中缀和前缀的混合形式
  - 各种二元运算符和条件运算符（?:）采用中缀形式，运算符位于运算对象之间
  - 函数调用是前缀形式，参数放在函数名后的括号内，逗号分隔
- C 语言的表达式表示不统一，但更接近数学里的常规写法
  - 表达式的结构也可以任意嵌套
  - 在写复杂的表达式时，也应该采用某种格式良好的写法
- 由于常规运算采用中缀表示
  - 需要有括号机制以便描述所需的运算顺序
  - 提供优先级的规定，以尽可能减少写括号的麻烦
  - 不能很方便地表示多元运算（如 + 和 * 等）
- 两种写法各有优点和缺点，需要习惯

命名和环境

- 编程语言必须提供为对象命名的机制，这是最基本的抽象机制
- Scheme 把名字标识符称为变量，其值就是与之关联的对象
- 用 define 为对象命名：

  > (define size 10)

  此后就可以用名字引用相关的值，如：

  > size

  10

  > (* size 3)

  30

- 可用任意复杂的表达式计算要求关联于变量的值：

  (define num (* size 30))

  这使 num 的值是 300。注意，Scheme 里的值可以是任何对象

命名和环境

- 计算对象可能有很复杂的结构，可能是经过复杂费时的计算得到
  - 如何每次用时都重新计算，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
- 写复杂的程序，常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
- 构造出的值可以存入变量供以后用，说明 Scheme 解释器有存储能力。这种存储称为“环境”，表达式在环境中求值
  - define 建立或修改环境中名字与值的关联
  - 表达式总在当前环境中求值（后面讨论）
  - 变量的值由环境中获得
  - Scheme 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），主要是预定义运算（和各种过程）

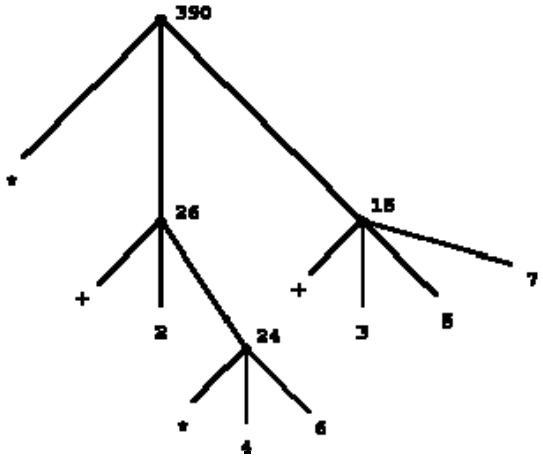
C 程序里的名字和环境

- C 语言没有明确的环境机制，但理解程序行为需要环境的概念
  - 函数，全局变量和其他全局定义的名字位于全局环境里。每个相应的声明或定义给全局环境引入一个新名字（及其定义）
  - 进入一个函数可能有新的局部定义（参数，局部变量等），形成局部环境。进入函数里的复合结构还可能有新的局部定义
  - 局部环境中的定义覆盖全局环境中同名的已有定义；内层局部环境里的定义覆盖外围环境中同名的已有定义
- 表达式在当前环境中求值，语句可能修改当前环境中有效定义的变量
- 注意 Scheme 与 C 语言的重要差异：
  - C 里定义变量要给定变量类型；Scheme 里引入变量无需说明类型
  - 类型确定变量可保存值的范围（静态性质，限定程序的动态行为）
  - 不说明类型，意味着变量取值范围无限制（可保存任意值）
  - 不限定类型带来灵活性，也使我们不可能做静态的类型检查

组合式的求值

- 通常要求求值的是组合式，解释器的工作方式是：
  - 求值该组合式的各表达式
  - 将最左子表达式的值（运算符的值，应该是一个过程）作用于相应的实际参数（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，先取得终端（叶）结点的值后向上累积
  - 最终在树根得到整个表达式的值
  - 树具有递归结构，递归处理很自然



组合式的求值

- 组合式求值的递归终将到达基本表达式，这时的值可以直接得：
  - 数的值是其自身（它们所表示的数值）
  - 内部运算符的值是系统里实现相关运算的指令序列
  - 其他名字的值在当前环境里找，找到相应名字-值关联时取出对应的值作为求值结果
- 可以把基本运算符（如 +）和其他预定义对象（如 **define**）都看作名字，在环境中查找关联的“值”。这样就统一了后两种情况
  - 环境为程序里用的名字提供定义。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
- 求值规则也有例外。如 **(define x 1)** 中的 **x** 不应该求值，是要求为名字 **x** 关联一个新值。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** 是其一。每个特殊形式有自己的求值规则

过程定义

- 表达式可能很长，复杂计算中常出现重复或类似表达式
  - 为控制程序复杂性，需要有抽象机制，**Scheme** 提供“过程定义”
- 求平方过程的定义：

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

  包括：过程名，形式参数，这一过程做什么（求值时做什么事情）

  这个定义表达式的求值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))          ; 用于计算
10000
> (+ (square 3) (* 20 (square 2)))  ; 嵌套的多次重复使用
89
> (define (sum-of-squares x y)      ; 用于定义新过程
      (+ (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

  如果只看使用，完全看不出 **square** 是基本操作还是用户定义过程

  复合过程的使用方式和威力和基本操作一样，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

- 复合过程确定的计算过程是（代换模型）：
  1. 求出各参数表达式的值（组合式求值）
  2. 找到要调用的过程的定义（根据第一个子表达式的求值结果）
  3. 用求出的实际参数代换过程体里的形式参数
  4. 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

过程应用的代换模型

- 代换模型给出了过程定义和应用的一种语义
  - 很多 Scheme 过程的行为可以用这个模型描述
  - 后面会看到，更复杂的过程需要用扩充的语义模型

注意：

- 代换模型只是为了帮助直观理解过程应用的行为
  - 它并没有反映解释器的实际工作过程
  - 实际的解释器是基于环境实现的，后面讨论
- 本课程将研究解释器的工作过程的一组模型
  - 代换模型最简单，最容易理解，但不能满足解释实际程序的需要
  - 代换模型的局限性是不能解释带有可变数据的程序
  - 需要更精细的模型（后面介绍）

应用序和正则序求值

- 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）
  - 这一做法很合理，但合理的做法并不唯一
- 另一可能方式是先不求值运算对象，直到实际需要用的时候再求值。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

C 语言表达式求值

- 求值过程就是表达式语义的实现
- C 语言的表达式求值过程牵涉的问题比较多
  - 要处理运算符的优先级、结合性、括号，确定计算顺序
  - 子表达式的求值方式也由运算符确定
- 对一般的一元/二元运算符和函数调用
  - 先求值作为运算对象的子表达式
  - 而后将运算符作用，得到运算结果
  - 也是递归定义的
- 有一批采用特殊求值规则的运算符，每个有特殊的求值方式
  - 二元逻辑运算符 **||** 和 **&&**
  - 条件运算符 **?:**　　顺序运算符 **,**
  - 各种赋值运算符和增量/减量运算符

C 语言表达式求值

- C 语言的基本求值规则是规范序求值
  - 自定义函数都是规范序求值
  - Scheme 允许自己定义按正则序求值的过程（本书中没讲）
- 几个特殊运算符有各自的求值规则
  - **||** 和 **&&** 先求值左边运算对象，可确定结果就结束并得到结果；不能确定再求值右边运算对象，根据其结果确定整个表达式的结果
  - 条件运算符 **?:**　先求值条件，而后根据条件的真假选择求值一个子表达式，以该子表达式的值作为整个条件表达式的值
  - 顺序运算符 **,**　　先求值左边子表达式，而后求值右边子表达式；以后一表达式的值作为值
  - 等等
- 一般语言里都有一些采用不同的特殊求值规则的特殊运算符。学习任何语言，都需注意这方面的情况

C 语言表达式求值

- 另一个问题是运算对象的求值顺序
  - 对 Scheme，一个组合式可能有多个子表达式
  - 对 C，常规二元运算符有两个运算对象，过程可能有多个参数

  它们按什么顺序求值？有规定的顺序吗？
- 无论是 C 还是 Scheme，都没规定运算对象的求值顺序。这意味着
  - 假定它们一定采用某种顺序都是不可靠的
  - 如果写的程序只有在某种特定求值顺序下才能正确工作，这种程序在实际使用中的正确性完全没保证

  不要写假定特定求值顺序的表达式！
- C 语言里依赖于求值顺序的表达式

```
m = n ++ + n;
printf("%d, %d", n, n++);
```

  等等。这种东西“没意思”

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

- 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，有特殊的求值规则

- 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，可用 **and**、**or**、**not** 组合出各种复杂逻辑条件，可用过程定义自己的谓词

过程定义实例：牛顿法求平方根

- 过程很像数学函数，但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)))
```

- 改进就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

- average 很简单。还需要决定“足够好”的标准。如用：

```
(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

C 函数

- 很容易写出对应于上面 Scheme 程序的 C 程序。如

```
int good_enough (double guess, double x)
{  return fabs(guess*guess - x) < 0.0001; }

double average (double x, double y)
{  return (x + y) / 2; }

double improve (double guess, double x) {
   return average(guess, x/guess);
}

double sqrt_iter (double guess, double x) {
   return good_enough(guess, x)
          ? guess : sqrt_iter(improve(guess,x), x);
}

double sqrt (double x) { return sqrt_iter(1., x); }
```

这是在 C 语言里做函数式程序设计（无赋值。思考题：能走多远？）

C 函数、语句和表达式

- C 语言里计算过程的抽象机制是函数
- C 函数定义与 Scheme 过程定义的不同
  - 需要类型描述（因为变量有类型）
  - 用 return 描述返回值
    - 没有 return 就没有返回值
- 表达式和语句
  - 表达式是有关计算的描述，运行中每个表达式都算出一个值
  - 语句是命令，要求做一个动作。动作没有“值”的概念
- Scheme 基于表达式，其中的每种结构都是表达式
  - 计算就是求值，计算一个表达式就要求出一个值
- C 语言（和其他常规语言）的基本结构单元是语句，表达式只是语句的组成部分，不能独立存在

要点

- 表达式
  - 基本表达式和组合式
- 变量和值
- 环境和变量的求值（后面还会讨论，这里说的是简单情况）
- 过程抽象，技术和意义
- 简单求值过程：代换模型
- 规范序和正则序求值
- 类型
- 至今很容易用 C 语言“模拟”Scheme 程序，后面将越来越不容易。但是贯串本书的思想仍然很有参考价值

问题？