[读书笔记]SICP：1[B]程序设计的基本元素

根据这里下载配置好DrRacket来跑SICP上的源代码，以及这里来学习一些基础的Lisp内容。

感觉SICP的内容较为离散，比较难整理……感觉视频课程和书本内容也不是很对应，先把书本内容过一遍，再看视频吧。

本章对应于书中的1.1。

描述一种语言时，应该将注意力集中在语言的基本表达形式、组合方法和抽象方法。

这里打算学习有关计算过程的知识，使用程序设计语言Lisp，它一个**主要特征**是：计算过程的Lisp描述（又称过程）本身可作为Lisp的数据来表示和操作，由此使得Lisp称为编写那些要将其他程序当做数据去操作的程序的最佳语言，比如解释器和编译器。

程序设计语言为了能够将简单的认识组合起来形成更复杂的认识，提供了**三种机制**：

- **基本表达形式**：用于表示语言所关心的最简单的个体，比如数。
- **组合的方法**：通过它们可以从较简单的东西出发构造出复合的元素。
- **抽象的方法**：通过它们可以为复合对象命名，并将他们当做单元去操作。

程序设计语言必须能表述基本的数据和过程，还需提供对过程和数据进行组合和抽象的方法。

1 表达式

我们可以输入一个**表达式**，而解释器的响应就是将表达式**求值**结果显示出来。我们可以输入基本表达式，比如数，也可以用表示基本过程的表达式形式（比如+和*），将表示数的表达式组合起来，形成复合表达式，来表示求要把有关过程应用于这些数，这个称为**组合式**，构造方式就是用一对括号括起一些表达式，形成一个表，表示一个过程引用，表中最左侧的元素称为**运算符**，其他元素称为**运算对象**，这种形式称为**前缀表示**，它可以适用于可能带有任意个实参的过程，也可以对其扩展允许组合式嵌套。要得到组合式的值，就是将运算符刻画的过程应用于有关的**实际参数**，也就是那些运算对象的值。

2 命名和复合过程

程序设计语言需要提供一种通用名字去使用计算对象的方式，这个名字标识符称为**变量**，它的值也就是对应的**对象**，这里可以使用`define`完成

```
(define <name> <value>)
```

对这个`define`的求值过程就是将名字`和对象`关联起来。此外还能得到一种强大的抽象技术——**过程定义**（就是定义函数），通过`define`可以为复合操作提供名字得到**复合过程**（也就是函数），这样就能将它的操作和基本运算一样作为一个单元来使用了。

```
(define (<name> <formal parameters>) <body>)
```

对这个`define`的求值过程其实可以看成两步：首先创建一个过程，`然后将其和名字`进行关联。当应用这个过程时，`中由`定义的**形式参数**将用与之对应的**实际参数**取代，然后对取代后的表达式进行求值，得到这个**过程应用**的值。更一般的情况下，``可以是一系列表达式，解释器就按需对这些表达式进行求值，然后将最后一个表达式的值作为整个过程应用的值并返回。

> 过程应用就是应用该过程的过程

解释器为了保持有关的**名字-值**对偶的轨迹，它必须维护某种存储能力，称为**全局环境**，它的作用就是用来确定表达式中各个符号的意义，比如`+`、`-`、`1`等，为求值过程的进行提供了一种上下文。

它允许我们用一个简单的名字去引用一个组合运算的结果，通过它我们可以通过一系列交互式动作，逐步创建起所需的**名字-对象关联**，来一步步创建出越来越复杂的计算性对象，这就鼓励人们采用递增的方式去开发和调试程序。（简单到复杂逐步抽象）

本章的**目的**就是把与过程性思维有关的各种问题隔离出来。

3 求值过程

3.1 通用组合式求值

对于组合式求值，解释器提供了**两种求值方式**：

- **应用序求值：（先计算子表达式）**
- - 求值该组合式的各个子表达式
  - 将作为最左子表达式（运算符或复合过程）的值（运算符或复合过程表示的计算过程）的那个过程应用于相应的实际参数（其他子表达式的值，即运算对象）。
- **正则序求值：（有限替换，需要时才计算）**
- - 首先用运算对象表达式去代换形参
  - 直到得到一个只包含基本运算符的表达式再执行求值。

这种计算过程称为过程应用的**代换模型**，也就是确定过程应用"意义"的一种模型，用来帮助我们理解过程应用的情况。我们以以下代码为例

```
(define (square x) (* x x))
(define (sum-of-square x y) (+ (square x) (square y)))
(define (f a) (sum-of-square (+ a 1) (* a 2)))
```

我们定义了若干个复合过程，当对`(f 5)`进行求值时，应用序求值过程为

```
; 由于5是基本表达式，所以可以不用求值，直接将f中的形参a替换为5
(sum-of-square (+ 5 1) (* 5 2))
; 对两个运算对象进行求值
(sum-of-square 6 10)
; 将sum-of-square中的形参x替换为6，形参y替换为10
(+ (square 6) (square 10))
; 对两个运算对象进行求值，需要先对运算符求值，直接进行替换
(+ (* 6 6) (* 10 10))
; 对两个运算对象求值
(+ 36 100)
;得到最终结果
136
```

正则序求值过程为

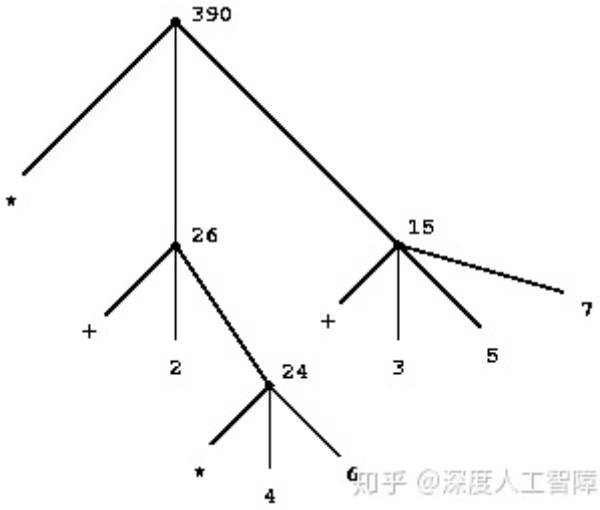
```
 ; 先进行代换
(sum-of-square (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
; 只包含基本运算符了，可以进行求值
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

对于可以通过替换去模拟，并产生合法值的过程应用，这两个求值方法会产生相同结果。而Lisp采用应用序求值，可以提高效率。

此外，**应用序求值过程具有以下含义：**

- 通过第一步可以看出该求值过程是**递归**的。我们可以用树的形式来表示以下代码

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



其中，每个**带分支节点**表示一个组合式，它的**分支**就对应于组合式中的运算符和运算对象，每个**终端节点**表示运算符或数值。从终端节点开始将运算对象的值一步步向上传递，就表示了组合式的求值过程，称为**树形积累**，也可以看成是对"求值"进行递归时的返回过程。

- 当执行完第一步时，其中一个组合式就变成了一个**基本表达式**，只包含：
- - **数**：它的值就是他所表示的数值
  - **其他名字**：它的值就是在环境中关联这个名字的对象
  - **内部运算符**：它的值就是能完成相应操作的机器指令序列，可以看成是"其他名字"的特殊形式。

> 基本表达式值包含数、内部运算符和其他名字

3.2 特殊形式

以上的求值过程是一般性的通用求值过程，还有一些**特殊形式**是不按照通用求值过程进行求值的，它们具有自身的求值规则，比如上文中的`define`

```
(define <name> <value>)
(define (<name> <formal parameters>) <body>)
```

第一行的求值结果就是将符号`和值`进行关联。

第二行的求值过程时：首先创建一个过程`，然后将其和名字`进行关联。当应用这个过程时，`中由`定义的**形式参数**将用与之对应的**实际参数**取代，然后对取代后的表达式进行求值，得到这个**过程应用**的值。更一般的情况下，``可以是一系列表达式，解释器就按需对这些表达式进行求值，然后将最后一个表达式的值作为整个过程应用的值并返回。

此外还为检测提供了以下**特殊形式**的运算符：

```
(cond
  (<p1> <e1>)
  (<p2> <e2>)
  ...
  (<pn> <en>)
  (else <e>))
```

其中`( )`称为**子句（clauses）**，`称为**谓词（predicate）**，`称为**序列表达式（consequent expression）**。对它求值的结果就是依次进入每个子句，对谓词进行求值，如果只为False，则对下一个子句的谓词进行求值，直到有一个谓词求值结果为True，就对其对应的序列表达式求值作为结果。如果没有谓词为True，则将`else`中的序列表达式的求值结果作为结果。

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

是`cond`的受限版本，只有两种情况。也是先对`进行求值，如果结果为True，则将`的求值结果作为最终结果，否则将``的求值结果作为最终结果。

```
(and <e1> ... <en>)
(or <e1> ... <en>)
```

这是一组逻辑复合运算符。对于`and`，解释器会从左到右一个个对`进行求值，如果某个`为False，则不对后续的`进行求值，最终结果直接为False。对于``or``，解释器会从左到右一个个对`进行求值，如果某个`为True，则不对后续的`进行求值，最终结果直接为True。

还有一个不是特殊形式的逻辑复合运算

```
(not <e>)
```

常规过程和特殊形式的区别：特殊形式具有自己的求值规则，而常规过程采用的是应用序求值或正则序求值顺序。如果将特殊形式定义为常规过程，就需要按照常规过程的求值方法进行求值，可能会出错。比如

```
(define (iff <p> <c> <a>) (if <p> <c> <a>))
(define (tryif a) (if (= a 0) 1 (/ 1 0)))
(define (tryiff a) (iff (= a 0) 1 (/ 1 0)))
```

这里通过`iff`将特殊形式的if变为了常规过程，然后定义了两个函数，当我们执行以下过程时，会得到不同的结果

```
> (tryif 0)
1
> (tryif 1)
. . /: division by zero
> (tryiff 0)
. . /: division by zero
> (tryiff 1)
. . /: division by zero
```

这是因为特殊形式的`if`会首先对`(= a 0)`进行求值，而常规过程的`iff`通常会使用应用序求值方法，则它的计算过程为

```
(tryiff 0)
(iff (= a 0) 1 (/ 1 0)))
; 应用序求值会先对运算对象进行求值
(iff True 1 (/ 1 0)) ;当对第三个运算对象求值时，就报错了
```

而且如果在递归过程中我们使用常规过程的`iff`，则可能会由于无限展开造成内存溢出。

综上：对各种表达式的求值规则可以描述为一个简单的通用规则和一组针对不多的特殊形式的专门规则。

4 实例：采用牛顿法求平方根

首先要明确**数学函数与计算过程的区别**：是描述一件事情的特征，与描述如何去做这件事之间的普遍性差异。数学中人们关心说明性描述（是什么），而计算机中人们关心行动性描述（怎么做）。

比如求解一个数的平方根，在数学中可以描述为
$\sqrt{x} = the\ y\ such\ that\ y \geq 0\ and\ y^2 = x$，它只告诉了你平方根是什么，但是它并没有提供如何计算平方根的方法，想要对其进行求解，就需要行动性描述

- 对结果进行猜测
- 对猜测进行更新
- 不断更新猜测直到猜测足够好

在计算机科学中，我们的任务就是形式化这种"如何做"的知识。这里可以用牛顿法来迭代逼近，不多加赘述，可以得到以下代码

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

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

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

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

(define (average x y)
  (/ (+ x y) 2))
```

这里使用`good-enough?`过程来判断当前猜测的解`guess`是否足够好，如果不够好就调用`improve`函数继续迭代，否则就得到最终结果。

但是这里不好处理`x`极大或极小的情况，可以修改`good-enough?`方法

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

可以看到，以上平方根的计算过程可以分解为若干个子问题：怎么说一个猜测是足够好的，怎么去改进一个猜测等等。使得工作中的每一个都通过一个独立过程完成，整个`sqrt`程序可以看成一族过程。注意要保证这些子过程是完成一件可以清楚表明的工作，使得它们可以被用作定义其他过程的模块，而其他过程直接将其当做**黑箱**来使用，而无需关心内部是如何实现的，只需要关注过程计算的结果，这个称为**过程抽象**，由此也提供了构建更大更复杂系统的可能性。

> 使用黑箱技术，也需要保证你的算法更具普遍性

所以一个过程定义应该能隐藏一些细节，用户在使用时无需考虑是如何实现的，由此需要保证的一点就是：过程的意义应该不依赖于其作者为形式参数所选的名字。这就要求过程的形式参数名必须只局限在有关的过程体中，否则就需要了解该过程的内部实现（比如了解如何定义形式参数，避免冲突）才能使用，就不是黑箱了。实际上形式参数的意义是受到过程的定义来约束的，与它是什么名字无关，所以称为**约束变量**，它的定义被约束于的那一集表达式称为该名字的**作用域**，其他不受约束的变量称为**自由变量**，它们才能决定过程的意义。比如`good-enough?`中的`<`、`-`、`abs`和`square`是自由变量，而`good-enough?`的意义依赖于这些的含义。

此外，还能发现，我们在`sqrt`中定义的`good-enough?`过程只适合`sqrt`过程，如果别的过程也想要一个`good-enough?`过程用于判断，此时就不能使用`good-enough?`这个名字了。所以我们希望能将这个种子过程局部化，将他们隐藏在`sqrt`过程中，使得别的过程也能使用`good-enough?`这个名字来定义自己的名字。

> 将辅助过程局部化

```
(define sqrt x)(
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (average x y)
    (/ (+ x y) 2))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

这个称为**块结构**，它除了能将所用的辅助过程定义在内部，还可以发现，`sqrt`的形式参数`x`对`sqrt`而言是约束变量而它的定义域在整个`sqrt`过程内，所以对于内部的辅助过程而言`x`是自由变量，就无需对其显示传递，所以可以修改为

```
 (define sqrt x)(
  (define (good-enough? guess)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (average y)
    (/ (+ x y) 2))
  (define (sqrt-iter guess)
    (if (good-enough? guess)
      guess
      (sqrt-iter (improve guess))))
  (sqrt-iter 1.0))
```