3.2 函数式编程

::: details INFO
译者：Mancuoj

来源：3.2 Functional Programming

对应：Disc 10、Lab 10
:::

现代计算机上运行的软件是由各种编程语言编写的。其中，有些是物理级别的语言，例如专为特定计算机设计的机器语言。这种语言主要关注如何将数据和指令转化为计算机可以识别的二进制位。使用机器语言的程序员主要考虑如何最大限度地发挥硬件的性能，以高效执行计算任务。而高级语言则基于机器语言之上，它屏蔽了数据和程序的底层表示方式。这些高级语言提供了更多的组合和抽象手段，例如函数定义，使得大型软件系统的构建更为方便和直观。

在这个章节中，我们要介绍一种偏向函数式编程的高级语言。这种语言，实际上是 Scheme 语言的一个子集，其计算模型与 Python 很相似，但它特点是只使用表达式而不使用语句，特别适合符号计算，并且它处理的数据都是不可变的（immutable）。

Scheme 是 Lisp 的一个变种，而 Lisp 是继 Fortran 之后仍然广受欢迎的第二古老的编程语言。Lisp 程序员社区几十年来持续蓬勃发展，Clojure 等 Lisp 的新方言拥有现代编程语言中增长最快的开发人员社区。如果你想亲手试试本文中的例子，可以下载一个 Scheme 的解释器 来操作。

3.2.1 表达式

Scheme 程序主要是由各种表达式构成的，这些表达式可以是函数调用或一些特殊的结构。一个函数调用通常由一个操作符和其后面跟随的零个或多个操作数组成，这点和 Python 是相似的。不过在 Scheme 中，这些操作符和操作数都被放在一对括号里：

```
(quotient 10 2)
```

Scheme 的语法一直采用前缀形式。也就是说，操作符像 `+` 和 `*` 都放在前面。函数调用可以互相嵌套，并且可能会写在多行上：

```
(+ (* 3 5) (- 10 6))
```

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

在 Scheme 中，表达式可以是基础类型，如数字，或是复合类型，如函数调用。数字字面量是基础类型，而函数调用则是可以包含任意子表达式一种复合形式。当求值一个函数调用时，Scheme 的处理方式和 Python 相似：首先对操作符和操作数进行求值，然后用操作符的结果（即函数）去处理操作数的结果（即参数）。

在 Scheme 里，`if` 表达式是一种特殊结构。尽管从语法上看，它似乎和常规的函数调用相似，但其求值方式却与众不同。一般来说，`if` 表达式的结构如下：

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

要理解 if 表达式是如何工作的，首先需要看它的 `<predicate>` 部分，即条件判断。如果这个条件成立（即为真），解释器就会执行并返回 `<consequent>` 的值；如果条件不成立，就会执行并返回 `<alternative>` 的值。

我们可以用常见的比较操作符来比较数字，但在 Scheme 中，这些操作符仍然采用前缀的形式：

```
(>= 2 1)
```

Scheme 中的布尔值有 `#t` 或者 `true` 代表真和 `#f` 或 `false` 代表假。你可以使用一些特定的布尔操作来组合它们，这些操作的执行逻辑和 Python 里的很相似。

> - `(and <e1> ... <en>)` 解释器会从左到右依次检查 `<e>` 表达式。一旦有一个 `<e>` 的结果是假，整个 `and` 表达式就直接返回假，并且剩下的 `<e>` 表达式不再检查。只有当所有 `<e>` 都是真时，`and` 表达式的返回值才是最后一个表达式的结果。
> - `(or <e1> ... <en>)` 解释器会从左到右依次检查 `<e>` 表达式。一旦有一个 `<e>` 的结果是真，`or` 表达式就直接返回那个真值，并且剩下的 `<e>` 表达式不再检查。只有当所有 `<e>` 都是假时，`or` 表达式才返回假。
> - `(not <e>)` 当 `<e>` 表达式的结果是假时，`not` 表达式就返回真，否则返回假。

3.2.2 定义

你可以通过使用 `define` 这一特殊结构为某个值赋予一个名字：

```
(define pi 3.14)
(* pi 2)
```

你可以用 `define` 的另一个版本来定义新的函数（在 Scheme 中，我们称之为“过程”）。例如，如果我们想定义一个求平方的函数，可以这样写：

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

定义一个过程（procedure）的标准格式如下：

```
(define (&lt;name&gt; &lt;formal parameters&gt;) &lt;body&gt;)
```

`<name>` 是一个符号，用于表示与环境中的过程定义相关的内容。而 `<formal parameters>` 是在过程主体内部用于指代过程对应参数的名称。在形式参数被替换为实际应用到过程的参数时，`<body>` 将生成过程应用的值。 `<name>` 和 `<formal parameters>` 被括在括号内，就像实际调用所定义的过程一样。

一旦我们定义了 `square`，就可以在调用表达式中使用它了：

```
(square 21)

(square (+ 2 5))

(square (square 3))
```

用户自定义的函数可以接受多个参数，并且还可以包含特殊形式：

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

(average 1 3)
```

```
(define (abs x)
    (if (< x 0)
        (- x)
        x))
```

Scheme 支持与 Python 相同的词法作用域规则，允许进行局部定义。下面，我们使用嵌套定义和递归定义了一个用于计算平方根的迭代过程：

```
(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))
(sqrt 9)
```

匿名函数是通过 `lambda` 特殊形式创建的。`Lambda` 用于创建过程，与 `define` 相似，但不需要为过程指定名称：

```
(lambda (<formal-parameters>) <body>)
```

生成的过程与使用 `define` 创建的过程一样，唯一的区别是它没有在环境中关联任何名称。实际上，以下表达式是等效的：

```
(define (plus4 x) (+ x 4))
(define plus4 (lambda (x) (+ x 4)))
```

和任何返回过程的表达式一样，lambda 表达式可以在调用表达式中用作运算符：

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

3.2.3 复合类型

在 Scheme 语言中，`pair` 是内置的数据结构。出于历史原因，`pair` 是通过内置函数 `cons` 创建的，而元素则可以通过 `car` 和 `cdr` 进行访问：

```
(define x (cons 1 2))

x

(car x)

(cdr x)
```

Scheme 语言中也内置了递归列表，它们使用 `pair` 来构建。特殊的值 `nil` 或 `'()` 表示空列表。递归列表的值是通过将其元素放在括号内，用空格分隔开来表示的：

```
(cons 1
      (cons 2
            (cons 3
                  (cons 4 nil))))

(list 1 2 3 4)

(define one-through-four (list 1 2 3 4))

(car one-through-four)

(cdr one-through-four)

(car (cdr one-through-four))

(cons 10 one-through-four)

(cons 5 one-through-four)
```

要确定一个列表是否为空，可以使用内置的 `null?` 谓词。借助它，我们可以定义用于计算长度和选择元素的标准序列操作：

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
(define (getitem items n)
  (if (= n 0)
      (car items)
      (getitem (cdr items) (- n 1))))
(define squares (list 1 4 9 16 25))

(length squares)

(getitem squares 3)
```

3.2.4 符号数据

迄今为止，我们使用的所有复合数据对象最终都是由数字构建的。Scheme 的一个强大之处在于能够处理任意符号作为数据。

为了操作符号，我们需要语言中的一个新元素：引用数据对象的能力。假设我们想构建列表 `(a b)`。我们不能通过 `(list a b)` 来实现这一目标，因为这个表达式构建的是 `a` 和 `b` 的值，而不是它们自身的符号。在 Scheme 中，我们通过在它们前面加上一个单引号来引用符号 `a` 和 `b` 而不是它们的值：

即 quote 方法

```
(define a 1)
(define b 2)

(list a b)

(list 'a 'b)

(list 'a b)
```

在 Scheme 中，任何不被求值的表达式都被称为被引用。这种引用的概念源自一个经典的哲学区分，即一种事物（比如一只狗）会四处奔跑和吠叫，而"狗"这个词是一种语言构造，用来指代这种事物。当我们在引号中使用"狗"时，我们并不是指代某只特定的狗，而是指代一个词语。在语言中，引号允许我们讨论语言本身，而在 Scheme 中也是如此：

```
(list 'define 'list)
```

引用还使我们能够输入复合对象，使用传统的打印表示方式来表示列表：

```
(car '(a b c))

(cdr '(a b c))
```

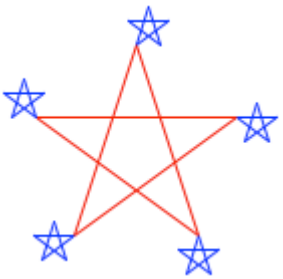
完整的 Scheme 语言包括更多功能，如变异操作（mutation operations）、向量（vectors）和映射（maps）。然而，到目前为止，我们介绍的这个子集已经提供了一个强大的函数式编程语言，可以实现我们在本文中讨论过的许多概念。

3.2.5 海龟图形

在这篇文章的 Scheme 版本中，加入了一个称为 Turtle 的图形功能（通常叫海龟），它源自 Logo 语言（Lisp 的另一种方言）中的一部分。这个"海龟"从一个画布的中心出发，根据特定的程序命令来移动和转向，并在它移动的过程中留下轨迹。尽管海龟图形最初是设计来吸引孩子学习编程的，但实际上，它对于高级程序员来说也是一个非常有趣的图形化编程工具。

当 Scheme 程序运行时，这只"海龟"会在画布上有一个特定的位置和方向。例如，使用 `forward` 和 `right` 这样的命令，可以改变"海龟"的位置和方向。为了方便，一些常用的程序命令有缩写形式，比如 `forward` 可以简写为 `fd`。在 Scheme 中，`begin` 这个特殊结构允许我们在一个表达式中串联多个子命令，这在执行多个操作时非常有用：

```
> (define (repeat k fn) (if (> k 0)
                            (begin (fn) (repeat (- k 1) fn))
                            nil))
> (repeat 5
          (lambda () (fd 100)
                     (repeat 5
                             (lambda () (fd 20) (rt 144)))
                     (rt 144)))
nil
```



Python 也内置了完整的 Turtle 功能，作为一个 turtle 库模块。

再举一个例子，Scheme 可以用其 Turtle 图形功能以非常简洁的方式来展现递归的图形。谢尔宾斯基三角形是一种分形结构，它将大的三角形分解成三个小的三角形，小三角形的顶点位于大三角形边的中点。我们可以用下面的 Scheme 程序来实现这个到一定递归深度的绘制：

```
> (define (repeat k fn)
    (if (> k 0)
        (begin (fn) (repeat (- k 1) fn))
        nil))

> (define (tri fn)
    (repeat 3 (lambda () (fn) (lt 120))))

> (define (sier d k)
    (tri (lambda ()
           (if (= k 1) (fd d) (leg d k)))))

> (define (leg d k)
    (sier (/ d 2) (- k 1))
    (penup)
    (fd d)
    (pendown))
```

`triangle` 函数是一个通用方法，它可以将某个绘图操作重复三次，每次操作后，都会让乌龟左转。`sier` 函数接受两个参数：一是长度 `d`，另一个是递归的深度 `k`。如果深度为 1，它就画一个简单的三角形；否则，它会调用 `leg` 函数来构建一个由三个小三角形组成的大三角形。`leg` 函数的功能是画一个谢尔宾斯基三角形的边，它首先递归调用 `sier` 函数来画出边的上半部分，然后移动乌龟到下一个顶点。而 `penup` 和 `pendown` 函数的作用是控制乌龟的画笔，使其在移动时可以选择是否画线。通过 `sier` 和 `leg` 的相互递归调用，我们可以得到以下效果：

```
> (sier 400 6)
```