# Scheme 语言概要(上)

**简介：** Scheme 语言是 LISP 语言的一个方言(或说成变种)，它诞生于 1975 年的 MIT，对于这个有近三十年历史的编程语言来说，它并没有象 C++，java，C#那样受到商业领域的青睐，在国内更是显为人知。但它在国外的计算机教育领域内却是有着广泛应用的，有很多人学的第一门计算机语言就是 Scheme 语言。

作为 Lisp 变体，Scheme 是一门非常简洁的计算语言，使用它的编程人员可以摆脱语言本身的复杂性，把注意力集中到更重要的问题上，从而使语言真正成为解决问题的工具。本文分为上、 下两部分来介绍 scheme 语言。

## 一．Scheme 语言的特点

Scheme 语言是 LISP 语言的一个方言(或说成变种)，它诞生于 1975 年的 MIT，对于这个有近三十年历史的编程语言来说，它并没有象 C++，java，C#那样受到商业领域的青睐，在国内更是显为人知。但它在国外的计算机教育领域内却是有着广泛应用的，有很多人学的第一门计算机语言就是 Scheme 语言。
它是一个小巧而又强大的语言，作为一个多用途的编程语言，它可以作为脚本语言使用，也可以作为应用软件的扩展语言来使用，它具有元语言特性，还有很多独到的特色,以致于它被称为编程语言中的"皇后"。
下面是洪峰对 Scheme 语言的编程特色的归纳：

- 词法定界（Lexical Scoping）
- 动态类型（Dynamic Typing）
- 良好的可扩展性
- 尾递归（Tail Recursive）
- 函数可以作为值返回
- 支持一流的计算连续
- 传值调用（passing-by-value）
- 算术运算相对独立

本文的目的是让有编程基础（那怕是一点点）的朋友能尽快的掌握 Scheme 语言的语法规则，如果您在读完本文后，发现自己已经会用 Scheme 语言了，那么我的目的就达到了。

## 二．Scheme 语言的标准与实现

### R5RS (Revised(5) Report on the Algorithmic Language Scheme)

Scheme 语言的语法规则的第 5 次修正稿 ,1998 年制定 ,即 Scheme 语言的现行标准 ,目前大多数 Scheme 语言的实现都将达到或遵循此标准，并且几乎都加入了一些属于自己的扩展特色。

### Guile (GNU's extension language)

Guile 是 GNU 工程的一个项目，它是 GNU 扩展语言库，它也是 Scheme 语言的一个具体实现；如果你将它作为一个库打包，可以把它链接到你的应用程序中去，使你的应用程序具有自己的脚本语言，这个脚本语言目前就是 Scheme 语言。
Guile 可以在 LINUX 和一些 UNIX 系统上运行，下面是简单的安装过程：
下载 guile-1.6.4 版，文件名为 guile-1.6.4.tar.gz，执行下面的命令：

```
tar xvfz guile-1.6.4.tar.gz
cd guile-1.6.4
./configure
make
make install
```

如此，即可以执行命令 guile，进入 guile>提示符状态，输入调试 Scheme 程序代码了，本文的所有代码都是在 guile 下调试通过。

### 其它实现

除了 Guile 外，Scheme 语言的实现还有很多，如：GNU/MIT-Scheme， SCI，Scheme48，DrScheme 等，它们大多是开源的，可以自由下载安装使用，并且跨平台的实现也很多。你会发现既有象 basic 的 Scheme 语言解释器，也有将 Scheme 语言编译成 C 语言的编译器，也有象 JAVA 那样将 Scheme 语言代码编译成虚拟机代码的编译器。

## 三．基本概念

### 注释

Scheme 语言中的注释是单行注释，以分号[;]开始一直到行尾结束，其中间的内容为注释，在程序运行时不做处理，如：

```
                ; this is a scheme comment line.
```

标准的 Scheme 语言定义中没有多行注释，不过在它的实现中几乎都有。在 Guile 中就有多行注释，以符号组合"#!"开始，以相反的另一符号组合"!#"结束，其中内容为注释，如：

```
#!
there are scheme comment area.
you can write mulity lines here .
!#
```

注意的是，符号组合"#!"和"!#"一定分做两行来写。

### Scheme 用做脚本语言

Scheme 语言可以象 sh，perl，python 等语言那样作为一种脚本语言来使用，用它来编写可执行脚本，在 Linux 中如果通过 Guile 用 Scheme 语言写可执行脚本 ,它的第一行和第二行一般是类似下面的内容：

```
#! /usr/local/bin/guile -s
!#
```

这样的话代码在运行时会自动调用 Guile 来解释执行，标准的文件尾缀是".scm"。

## 块(form)

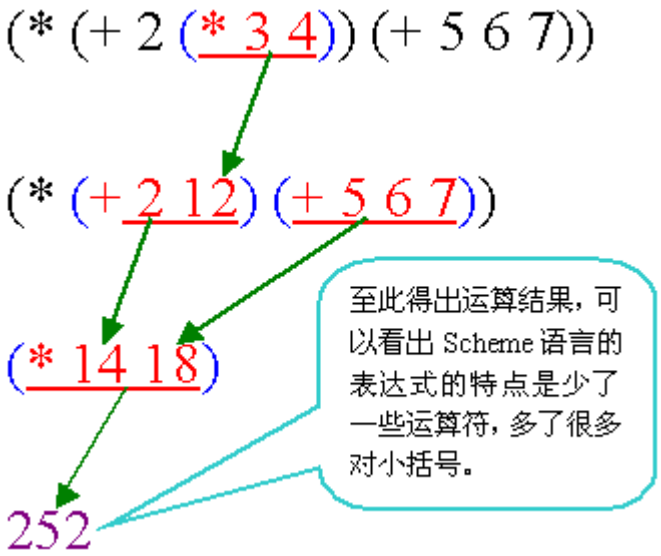
块(form)是 Scheme 语言中的最小程序单元，一个 Scheme 语言程序是由一个或多个 form 构成。没有特殊说明的情况下 form 都由小括号括起来，形如：

```
(define x 123)
(+ 1 2)
(* 4 5 6)
(display "hello world")
```

一个 form 也可以是一个表达式，一个变量定义，也可以是一个过程。

## form 嵌套

Scheme 语言中允许 form 的嵌套，这使它可以轻松的实现复杂的表达式，同时也是一种非常有自己特色的表达式。下图示意了嵌套的稍复杂一点的表达式的运算过程：




## 变量定义

可以用 define 来定义一个变量，形式如下：
(define 变量名 值)
如：(define x 123)，定义一个变量 x，其值为 123。

## 更改变量的值

可以用 set!来改变变量的值，格式如下：
(set! 变量名值)
如：(set! x "hello")，将变量 x 的值改为"hello"。

Scheme 语言是一种高级语言，和很多高级语言(如 python，perl)一样，它的变量类型不是固定的，可以随时改变。

# 四．数据类型

## 1．简单数据类型

**逻辑型(boolean)**

最基本的数据类型，也是很多计算机语言中都支持的最简单的数据类型，只能取两个值：#t，相当于其它计算机语言中的 TRUE；#f，相当于其它计算机语言中的 FALSE。

Scheme 语言中的 boolean 类型只有一种操作：not。其意为取相反的值，即：

```
(not #f) => #t
(not #t) => #f
```

not 的引用，与逻辑非运算操作类似

```
guile> (not 1)
#f
guile> (not (list 1 2 3))
#f
guile> (not 'a)
#f
```

从上面的操作中可以看出来，只要 not 后面的参数不是逻辑型，其返回值均为#f。

**数字型(number)**

它又分为四种子类型：整型(integer)，有理数型(rational)，实型(real)，复数型(complex)；它们又被统一称为数字类型(number)。

如：复数型(complex) 可以定义为 (define c 3+2i)

实数型（real）可以定义为 (define f 22/7)

有理数型（rational）可以定义为 (define p 3.1415)

整数型(integer) 可以定义为 (define i 123)

Scheme 语言中，数字类型的数据还可以按照进制分类，即二进制，八进制，十进制和十六进制，在外观形式上它们分别以符号组合 #b、 #o、 #d、 #x 来作为表示数字进制类型的前缀，其中表示十进制的#d 可以省略不写，如：二进制的 #b1010 ，八进制的 #o567，十进制的 123 或 #d123，十六进制的 #x1afc 。

Scheme 语言的这种严格按照数学定理来为数字类型进行分类的方法可以看出 Scheme 语言里面渗透着很深的数学思想，Scheme 语言是由数学家们创造出来的，在这方面表现得也比较鲜明。

**字符型(char)**

Scheme 语言中的字符型数据均以符号组合 "#\" 开始，表示单个字符，可以是字母、数字或"[ ! $ % & * + - . / : %lt; = > ? @ ^ _ ~ ]"等等其它字符，如：

#\A 表示大写字母 A，#\0 表示字符 0，

其中特殊字符有：#\space 表示空格符和 #\newline 表示换行符。

**符号型(symbol)**

符号类型是 Scheme 语言中有多种用途的符号名称，它可以是单词，用括号括起来的多个单词，也可以是无意义的字母组合或符号组合，它在某种意义上可以理解为 C 中的枚举类型。看下面的操作：

```
guile> (define a (quote xyz)) ； 定义变量 a 为符号类型，值为 xyz
guile> a
xyz
guile> (define xyz 'a) ；定义变量 xyz 为符号类型，值为 a
guile> xyz
a
```

此处也说明单引号' 与 quote 是等价的，并且更简单一些。符号类型与字符串不同的是符号类型不能象字符串那样可以取得长度或改变其中某一成员字符的值，但二者之间可以互相转换。

## 2．复合数据类型

可以说复合数据类型是由基本的简单数据类型通过某种方式加以组合形成的数据类型，特点是可以容纳多种或多个单一的简单数据类型的数据，多数是基于某一种数学模型创建的。

字符串(string) 由多个字符组成的数据类型，可以直接写成由双引号括起的内容，如："hello" 。下面是 Guile 中的字符串定义和相关操作：

```
guile> (define name "tomson")
guile> name
"tomson"
guile> (string-length name) ；取字符串的长度
6
guile> (string-set! name 0 #\g) ；更改字符串首字母(第 0 个字符)为小写字母 g (#\g)
guile> name
"gomson"
guile> (string-ref name 3) ；取得字符串左侧第 3 个字符（从 0 开始）
#\s
```

字符串还可以用下面的形式定义：

```
guile> (define other (string #\h #\e #\l #\l #\o ))
guile> other
"hello"
```

字符串中出现引号时用反斜线加引号代替，如："abc\"def" 。

**点对(pair)**

我把它译成"点对"，它是一种非常有趣的类型，也是一些其它类型的基础类型，它是由一个点和被它分隔开的两个所值组成的。形如：（1 . 2）或（a . b），注意的是点的两边有空格。

这是最简单的复合数据类型，同是它也是其它复合数据类型的基础类型，如列表类型（list）就是由它来实现的。

按照 Scheme 语言说明中的惯例，以下我们用符号组合 "=>" 来表示表达式的值。

它用 cons 来定义，如：（cons 8 9）=>(8 . 9)

其中在点前面的值被称为 car ，在点后面的值被称为 cdr，car 和 cdr 同时又成为取 pair 的这两个值的过程，如：

```
(define p (cons 4 5))      => (4 . 5)
(car p)                    => 4
(cdr p)                    => 5
```

还可以用 set-car! 和 set-cdr! 来分别设定这两个值：

```
(set-car! p "hello")
(set-cdr! p "good")
```

如此，以前定义的 p 又变成了 ("hello" . "good") 这个样子了。

**列表(list)**

列表是由多个相同或不同的数据连续组成的数据类型，它是编程中最常用的复合数据类型之一，很多过程操作都与它相关。下面是在 Guile 中列表的定义和相关操作：

```
guile> (define la (list 1 2 3 4 ))
guile> la
(1 2 3 4)
guile> (length la)  ; 取得列表的长度
4
guile> (list-ref la 3)  ; 取得列表第 3 项的值（从 0 开始）
4
guile> (list-set! la 2 99)  ; 设定列表第 2 项的值为 99
99
guile> la
(1 2 99 4)
guile> (define y (make-list 5 6))  ;创建列表
guile> y
(6 6 6 6 6)
```

make-list 用来创建列表，第一个参数是列表的长度，第二个参数是列表中添充的内容；还可以实现多重列表，即列表的元素也是列表，如：(list (list 1 2 3) (list 4 5 6))。

**列表与 pair 的关系**

回过头来，我们再看看下面的定义：

```
guile> (define a (cons 1 (cons 2 (cons 3 '()))))
guile> a
(1 2 3)
```

由上可见，a 本来是我们上面定义的点对，最后形成的却是列表。事实上列表是在点对的基础上形成的一种特殊格式。

再看下面的代码：

```
guile> (define ls (list 1 2 3 4))
guile>ls
(1 2 3 4)
guile> (list? ls)
#t
guile> (pair? ls)
```

```
#t
```

由此可见，list 是 pair 的子类型，list 一定是一个 pair，而 pair 不是 list。

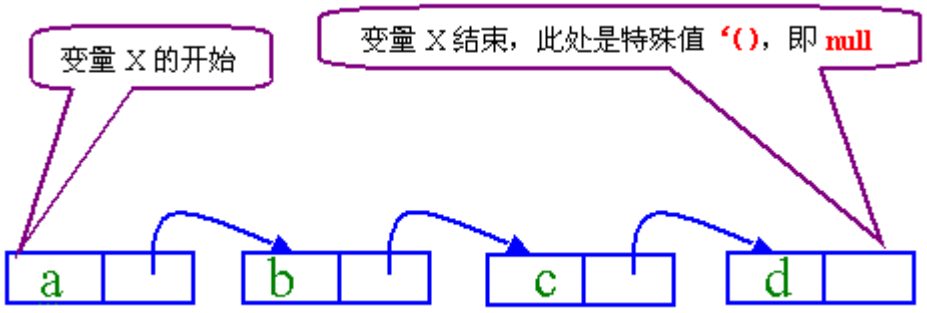
```
guile> (car ls)
1
guile> (cdrls)
(2 3 4)
```

其 cdr 又是一个列表，可见用于 pair 的操作过程大多可以用于 list。

```
        guile> (cadrls)   ; 此"点对"对象的 cdr 的 car
2
guile> (cddrls)   ; 此"点对"对象的 cdr 的 cdr
(3 4)
guile> (caddrls)   ; 此"点对"对象的 cdr 的 cdr 的 car
3
guile> (cdddrls)   ; 此"点对"对象的 cdr 的 cdr 的 cdr
(4)
```

上在的操作中用到的 cadr，cdddr 等过程是专门对 PAIR 型数据再复合形成的数据操作的过程，最多可以支持在中间加四位 a 或 d，如 cdddr，caaddr 等。

下图表示了由 pairs 定义形成的列表：




这个列表可以由 pair 定义为如下形式：

```
(define x (cons 'a (cons 'b (cons 'c (cons 'd '())))))
```

而列表的实际内容则为：(a b c d)

由 pair 类型还可以看出它可以轻松的表示树型结构，尤其是标准的二叉树。

**向量（vector）**

可以说是一个非常好用的类型，是一种元素按整数来索引的对象，异源的数据结构，在占用空间上比同样元素的列表要少，在外观上：

列表示为：（1 2 3 4）

VECTOR 表示为：#(1 2 3 4)

可以正常定义：(define v (vector 3 4 5))

也可以直接定义：(define v #(3 4 5))

vector 是一种比较常用的复合类型，它的元素索引从 0 开始，至第 n-1 结束，这一点有点类似 C 语言中的数组。

关于向量表（vector）的常用操作过程：

```
guile> (define v (vector 1 2 3 4 5))
guile> v
#(1 2 3 4 5)
guile> (vector-ref v 0)    ; 求第 n 个变量的值
1
guile> (vector-length v)  ; 求 vector 的长度
5
guile> (vector-set! v 2 "abc")  ; 设定 vector 第 n 个元素的值
guile> v
#(1 2 "abc" 4 5)
guile> (define x (make-vector 5 6))  ; 创建向量表
guile> x
#(6 6 6 6 6)
```

make-vector 用来创建一个向量表，第一个参数是数量，后一个参数是添充的值，这和列表中的 make-list 非常相似。

我们可以看出，在 Scheme 语言中，每种数据类型都有一些基本的和它相关的操作过程，如字符串，列表等相关的操作，这些操作过程都很有规律，过程名的单词之间都用-号隔开，很容易理解。对于学过 C++ 的朋友来说，更类似于某个对象的方法，只不过表现的形式不同了。

## 3．类型的判断、比较、运算、转换与方法

**类型判断**

Scheme 语言中所有判断都是用类型名加问号再加相应的常量或变量构成，形如：

```
(类型? 变量)
```

Scheme 语言在类型定义中有比较严格的界定，如在 C 语言等一些语言中数字 0 来代替逻辑类型数据 False，在 Scheme 语言中是不允许的。

以下为常见的类型判断和附加说明：

逻辑型：

```
(boolean? #t) => #t
(boolean? #f) => #t        因为#t 和#f 都是 boolean 类型，所以其值为#t
(boolean? 2)  => #f        因为 2 是数字类型，所以其值为 #f
```

字符型

```
(char? #\space)   => #t
(char? #\newline) => #t    以上两个特殊字符：空格和换行
(char? #\f)               => #t    小写字母 f
(char? #\;)               => #t    分号 ;
(char? #\5)               => #t    字符 5     ，以上这些都是正确的，所以返回值都是 #t
(char? 5)                 => #f    这是数字 5 ，不是字符类型，所以返回 #f
```

数字型

```
(integer? 1)              => #t
(integer? 2345)   => #t
(integer? -90)            => #t    以上三个数均为整数
```

```
(integer? 8.9)              => #f   8.9 不整数
(rational? 22/7)  => #t
(rational? 2.3)             => #t
(real? 1.2)                 => #t
(real? 3.14159)   => #t
(real? -198.34)             => #t     以上三个数均为实数型
(real? 23)                  => #t   因为整型属于实型
(number? 5)       => #t
(number? 2.345)   => #t
(number? 22/7)    => #t
```

其它型

```
(null? '())                 => #t  ; null 意为空类型，它表示为 '() ，即括号里什么都没有的符号
(null? 5)                   => #f
(define x 123)              定义变量 x 其值为 123
(symbol? x)                 => #f
(symbol? 'x)                => #t  ; 此时 'x 为符号 x，并不表示变量 x 的值
```

在 Scheme 语言中如此众多的类型判断功能，使得 Scheme 语言有着非常好的自省功能。即在判断过程的参数是否附合过程的要求。

**比较运算**

Scheme 语言中可以用<，>，<=，>=，= 来判断数字类型值或表达式的关系，如判断变量 x 是否等于零，它的形式是这样的：(= x 0) ，如 x 的值为 0 则表达式的值为#t，否则为#f。

还有下面的操作：

```
(eqv? 34 34)   =>  #t
(= 34 34)      =>  #t
```

以上两个 form 功能相同，说明 eqv? 也可以用于数字的判断。

在 Scheme 语言中有三种相等的定义，两个变量正好是同一个对象；两个对象具有相同的值；两个对象具有相同的结构并且结构中的内容相同。除了上面提到的符号判断过程和 eqv?外，还有 eq?和 equal?也是判断是否相等的过程。

**eq?，eqv?，equal?**

eq?，eqv?和 equal?是三个判断两个参数是否相等的过程，其中 eq?和 eqv?的功能基本是相同的，只在不同的 Scheme 语言中表现不一样。

eq?是判断两个参数是否指向同一个对象，如果是才返回#t；equal?则是判断两个对象是否具有相同的结构并且结构中的内容是否相同，它用 eq?来比较结构中成员的数量；equal?多用来判断点对，列表，向量表，字符串等复合结构数据类型。

```
guile> (define v (vector 3 4 5))
guile> (define w #(3 4 5))  ; w 和 v 都是 vector 类型，具有相同的值#(3 4 5)
guile> (eq? v w)
#f                                          ; 此时 w 和 v 是两个对象
guile> (equal? v w)
#t                ; 符合 equal?的判断要求
```

以上操作说明了 eq? 和 equal? 的不同之处，下面的操作更是证明了这一点：

```
guile> (define x (make-vector 5 6))
guile> x
#(6 6 6 6 6)
guile> (eq? x x)    ; 是同一个对象，所以返回#t
#t
guile> (define z (make-vector 5 6))
guile> z
#(6 6 6 6 6)
guile> (eq? x z)    ; 不是同一个对象
#f
guile> (equal? x z)  ; 结构相同，内容相同，所以返回#t
#t
```

**算术运算**

Scheme 语言中的运算符有：

+ , - , * , / 和 expt（指数运算）

其中 - 和 / 还可以用于单目运算，如：

```
(- 4)  => -4
(/ 4)  => 1/4
```

此外还有许多扩展的库提供了很多有用的过程，

```
max 求最大 (max 8 89 90 213)  => 213
min 求最小 (min 3 4 5 6 7)  => 3
abs 求绝对值 (abs -7) ==> 7
```

除了 max，min，abs 外，还有很多数学运算过程，这要根据你用的 Scheme 语言的运行环境有关，不过它们大多是相同的。在 R5RS 中规定了很多运算过程，在 R5RS 的参考资料中可以很容易找到。

**转换**

Scheme 语言中用符号组合"->"来标明类型间的转换（很象 C 语言中的指针）的过程，就象用问号来标明类型判断过程一样。下面是一些常见的类型转换过程：

```
guile> (number->string 123)  ; 数字转换为字符串
"123"
guile> (string->number "456")  ; 字符串转换为数字
456
guile> (char->integer #\a)   ;字符转换为整型数，小写字母 a 的 ASCII 码值为 96
97
guile> (char->integer #\A)  ;大写字母 A 的值为 65
65
guile> (integer->char 97)  ;整型数转换为字符
#\a
guile> (string->list "hello")   ;字符串转换为列表
(#\h #\e #\l #\l #\o)
guile> (list->string (make-list 4 #\a)) ; 列表转换为字符串
```

```
"aaaa"
guile> (string->symbol "good")  ;字符串转换为符号类型
good
guile> (symbol->string 'better)  ;符号类型转换为字符串
"better"
```

# 五．过程定义

## 过程（Procedure）

在 Scheme 语言中，过程相当于 C 语言中的函数，不同的是 Scheme 语言过程是一种数据类型，这也是为什么 Scheme 语言将程序和数据作为同一对象处理的原因。如果我们在 Guile 提示符下输入加号然后回车，会出现下面的情况：

```
guile> +
#<primitive-procedure +>
```

这告诉我们"+"是一个过程，而且是一个原始的过程，即 Scheme 语言中最基础的过程，在 GUILE 中内部已经实现的过程，这和类型判断一样，如 boolean?等，它们都是 Scheme 语言中最基本的定义。注意：不同的 Scheme 语言实现环境，出现的提示信息可能不尽相同，但意义是一样的。
define 不仅可以定义变量，还可以定义过程，因在 Scheme 语言中过程（或函数）都是一种数据类型，所以都可以通过 define 来定义。不同的是标准的过程定义要使用 lambda 这一关键字来标识。

## Lambda 关键字

Scheme 语言中可以用 lambda 来定义过程，其格式如下：
(define 过程名 ( lambda (参数 ...) (操作过程 ...)))
我们可以自定义一个简单的过程，如下：

```
                (define add5 (lambda (x) (+ x 5)))
```

此过程需要一个参数，其功能为返回此参数加 5 的值，如：

```
          (add5 11) => 16
```

下面是简单的求平方过程 square 的定义：

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

## 与 lambda 相同的另一种方式

在 Scheme 语言中，也可以不用 lambda，而直接用 define 来定义过程，它的格式为：
(define (过程名参数) (过程内容 …))
如下面操作：

```
                (define (add6 x) (+ x 6))
                add6
```

```
                #<procedure: add6 (x)>说明 add6 是一个过程，它有一个参数 x
                (add6 23)    => 29
```

再看下面的操作：

```
guile> (define fun
             (lambda(proc x y)
                    (proc x y)))
guile> fun
#<procedure fun (proc x y)>
guile> (fun * 5 6)
30
guile> (fun / 30 3)
10
```

## 更多的过程定义

上面定义的过程 fun 有三个参数，其中第一个参数 proc 也是一个操作过程（因为在 Scheme 语言中过程也是一种数据，可以作为过程的参数），另外两个参数是数值，所以会出现上面的调用结果。

```
guile> (define add
             (lambda (x y)
                    (+ x y)))
guile> add
#<procedure add (x y)>
guile> (fun add 100 200)
300
```

继续上面操作，我们定义一个过程 add，将 add 作为参数传递给 fun 过程，得出和(fun + 100 200)相同的结果。

```
guile> ((lambda (x) (+ x x)) 5)
10
```

上面的 (lambda(x) (+ x x)) 事实上是简单的过程定义，在后面直接加上操作参数 5，得出结果 10，这样实现了匿名过程，直接用过程定义来操作参数，得出运算结果。

通过上面的操作，相信你已初步了解了过程的用法。既然过程是一种数据类型，所以将过程作为过程的参数是完全可以的。以下过程为判断参数是否为过程，给出一个参数，用 procedure? 来判断参数是否为过程，采用 if 结构（关于 if 结构见下面的介绍）：

```
guile> (define isp
          (lambda (x)
                 (if (procedure? x) 'isaprocedure 'notaprocedure)))
guile>isp
#<procedure isp (x)>
guile> (isp 0)
notaprocedure
guile> (isp +)
```

```
isaprocedure
```

上面的过程就体现了 Scheme 语言的参数自省（辨别）能力，'0'是数字型，所以返回 notaprocedure；而'+'是一个最基础的操作过程，所以返回 isaprocedure。

### 过程的嵌套定义

在 Scheme 语言中，过程定义也可以嵌套，一般情况下，过程的内部过程定义只有在过程内部才有效，相当 C 语言中的局部变量。
如下面的代码的最终结果是 50：

```
(define fix
        (lambda (x y z)
                (define add
                        (lambda (a b) (+ a b)))
                (- x (add y z))))
(display (fix 100 20 30))
```

此时过程 add 只在 fix 过程内部起做用，这事实上涉及了过程和变量的绑定，可以参考下面的关于过程绑定（let，let* 和 letrec）的介绍。
过程是初学者难理解的一个关键，随着过程参数的增加和功能的增强，过程的内容变得越来越复杂，小括号也会更多，如果不写出清晰的代码的话，读代码也会成为一个难题。
熟悉了 scheme 基本概念、数据类型和过程（函数）后， 下一部分我们来学习 scheme 的结构、递归调用和其他扩展功能。

# Scheme 语言概要（下）

谈完了 scheme 的基本概念、数据类型和过程，我们接着介绍 scheme 的结构、递归调用、变量和过程的绑定、输入输出等功能。

## 一．常用结构

**顺序结构**
也可以说成由多个 form 组成的 form，用 begin 来将多个 form 放在一对小括号内，最终形成一个 form。
格式为：(begin form1 form2 …)
如用 Scheme 语言写成的经典的 helloworld 程序是如下样子的：

```
(begin
        (display "Hello world!")  ; 输出"Hello world!"
        (newline))                                  ; 换行
```

**if 结构**

Scheme 语言的 if 结构有两种格式，一种格式为：(if 测试过程 1 过程 2)，即测试条件成立则执行过程 1，否则执行过程 2。例如下面代码：

```
(if (= x 0)
(display "is zero")
(display "not zero"))
```

还有另一种格式：(if 测试过程) ，即测试条件成立则执行过程。例如下面代码：

```
                (if (< x 100) (display "lower than 100"))
```

根据类型判断来实现自省功能，下面代码判断给定的参数是否为字符串：

```
(define fun
         (lambda ( x )
                  (if (string? x)
                           (display "is a string")
                           (display "not a string"))))
```

如执行 (fun 123) 则返回值为"not a string"，这样的功能在 C++或 JAVA 中实现的话可能会很费力气。

**cond 结构**

Scheme 语言中的 cond 结构类似于 C 语言中的 switch 结构，cond 的格式为：

```
                (cond ((测试) 操作) … (else 操作))
```

如下是在 Guile 中的操作：

```
guile> (define w (lambda (x)
     (cond ((< x 0) 'lower)
          ((> x 0) 'upper)
          (else 'equal))))
guile> w
#<procedure w (x)>
guile> (w 9)
upper
guile> (w -8)
lower
guile> (w 0)
equal
```

上面程序代码中，我们定义了过程 w，它有一个参数 x，如果 x 的值大于 0，则返回符号 upper，如 x 的值小于 0 则返回符号 lower，如 x 的值为 0 则返回符号 equal。

下载已做成可执行脚本的 例程。

cond 可以用 if 形式来写，上面的过程可以如下定义：

```
guile> (define ff
                      (lambda (x)
                (if (< x 0) 'lower
```

```
                        (if (> x 0) 'upper 'zero))))
guile>ff
#<procedure ff (x)>
guile> (ff 9)
upper
guile> (ff -9)
lower
guile> (ff 0)
zero
```

这在功能上是和 cond 一样的，可以看出 cond 实际上是实现了 if 的一种多重嵌套。

**case 结构**

case 结构和 cond 结构有点类似，它的格式为：

(case (表达式) ((值) 操作)) ... (else 操作)))

case 结构中的值可以是复合类型数据，如列表，向量表等，只要列表中含有表达式的这个结果，则进行相应的操作，如下面的代码：

```
(case (* 2 3)
 ((2 3 5 7) 'prime)
 ((1 4 6 8 9) 'composite))
```

上面的例子返回结果是 composite，因为列表(1 4 6 8 9)中含有表达式(* 2 3)的结果 6；下面是在 Guile 中定义的 func 过程，用到了 case 结构：

```
guile> (define func
             (lambda (x y)
                   (case (* x y)
                         ((0) 'zero)
                         (else 'nozero))))
guile>func
#<procedure func (x y)>
guile> (func 2 3)
nozero
guile> (func 2 0)
zero
guile> (func 0 9)
zero
guile> (func 2 9)
nozero
```

可以下载另一个脚本文件 te.scm，参考一下。

**and 结构**

and 结构与逻辑与运算操作类似，and 后可以有多个参数，只有它后面的参数的表达式的值都为#t 时，它的返回值才为#t，否则为#f。看下面的操作：

```
guile> (and (boolean? #f) (< 8 12))
#t
```

```
guile> (and (boolean? 2) (< 8 12))
#f
guile> (and (boolean? 2) (> 8 12))
#f
```

如果表达式的值都不是 boolean 型的话，返回最后一个表达式的值，如下面的操作：

```
guile> (and (list 1 2 3) (vector 'a 'b 'c))
#(a b c)
guile> (and 1 2 3 4 )
4
guile> (and 'e 'd 'c 'b 'a)
a
```

**or 结构**

or 结构与逻辑或运算操作类似，or 后可以有多个参数，只要其中有一个参数的表达式值为#t，其结果就为#t，只有全为#f 时其结果才为#f。如下面的操作：

```
guile> (or #f #t)
#t
guile> (or #f #f)
#f
guile> (or (rational? 22/7) (< 8 12))
#t
guile> (rational? 22/7)
#t
guile> (real? 22/7)
#t
guile> (or (real? 4+5i) (integer? 3.22))
#f
```

我们还可以用 and 和 or 结构来实现较复杂的判断表达式，如在 C 语言中的表达式：

```
((x > 100) && (y < 100)) 和 ((x > 100) || (y > 100))
```

在 Scheme 中可以表示为：

```
guile> (define x 123)
guile> (define y 80)
guile> (and (> x 100) (< y 100))
#t
guile> (or (> x 100) (> y 100))
#t
```

Scheme 语言中只有 if 结构是系统原始提供的，其它的 cond，case，and，or，另外还有 do，when，unless 等都是可以用宏定义的方式来定义的，这一点充分体现了 Scheme 的元语言特性，关于 do，when 等结构的使用可以参考 R5RS。

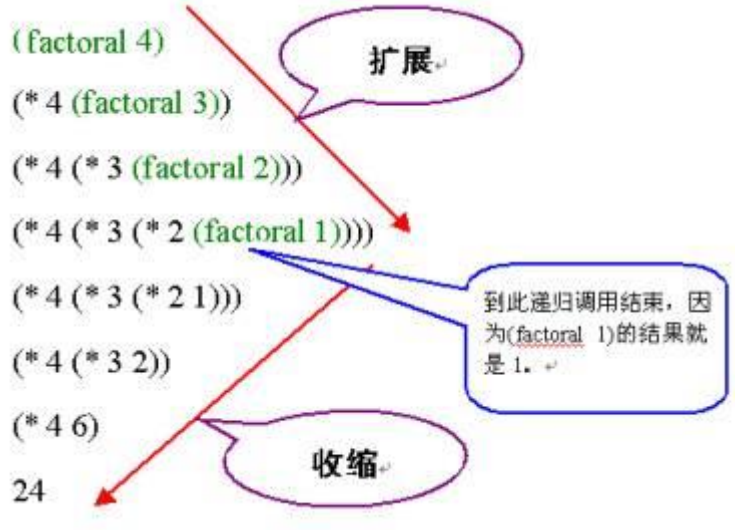
# 二．递归调用

**用递归实现阶乘**

在 Scheme 语言中，递归是一个非常重要的概念，可以编写简单的代码很轻松的实现递归调用，如下面的阶乘过程定义：

```
(define  factoral (lambda (x)
                    (if (<= x 1) 1
                       (* x (factoral (- x 1))))))
```

我们可以将下面的调用(factoral 4)，即 4 的阶乘的运算过程图示如下：




以下为 factoral 过程在 Guile 中的运行情况：

```
guile> (define factoral (lambda (x) (if (<= x 1) 1 (* x (factoral (- x 1))))))
guile>factoral
#<procedure factoral (x)>
guile> (factoral 4)
24
```

**另一种递归方式**

下面是一另一种递归方式的定义：

```
(define (factoral n)
        (define (iter product counter)
                (if (> counter n)
                        product
                        (iter (* counter product) (+ counter 1))))
        (iter 1 1))
(display (factoral 4))
```

这个定义的功能和上面的完全相同，只是实现的方法不一样了，我们在过程内部实现了一个过程 iter，它用 counter 参数来计数，调用时从 1 开始累计，这样它的展开过程正好和我们上面的递归过程的从 4 到 1 相反，而是从 1 到 4。

**循环的实现**

在 Scheme 语言中没有循环结构，不过循环结构可以用递归来很轻松的实现（在 Scheme 语言中只有通过递归才能实现循环）。对于用惯了 C 语言循环的朋友，在 Scheme 中可以用递归简单实现：

```
guile> (define loop
             (lambda(x y)
             (if (<= x y)
         (begin (display x) (display #\\space) (set! x (+ x 1))
         (loop x y)))))
guile> loop
#<procedure loop (x y)>
guile> (loop 1 10)
1 2 3 4 5 6 7 8 9 10
```

这只是一种简单的循环定义，过程有两个参数，第一个参数是循环的初始值，第二个参数是循环终止值，每次增加 1。相信读者朋友一定会写出更漂亮更实用的循环操作来的。

# 三．变量和过程的绑定

### let，let*，letrec

在多数编程语言中都有关于变量的存在的时限问题，Scheme 语言中用 let，let*和 letrec 来确定变量的存在的时限问题，即局部变量和全局变量，一般情况下，全局变量都用 define 来定义，并放在过程代码的外部；而局部变量则用 let 等绑定到过程内部使用。

用 let 可以将变量或过程绑定在过程的内部，即实现局部变量：

```
guile> let
#<primitive-macro! let>
```

从上面的操作可以看出 let 是一个原始的宏，即 guile 内部已经实现的宏定义。

下面的代码显示了 let 的用法（注意多了一层括号）：

```
guile> (let ((x 2) (y 5)) (* x y))
10
```

它的格式是：(let ((…)…) …)，下面是稍复杂的用法：

```
guile> (let ((x 5))
       (define foo (lambda (y) (bar x y)))
       (define bar (lambda (a b) (+ (* a b) a)))
       (foo (+ x 3)))
45
```

以上是 Guile 中的代码实现情况。它的实现过程大致是：(foo 8) 展开后形成 (bar 5 8)，再展开后形成 (+ (* 5 8) 5) ，最后其值为 45。

再看下面的操作：

```
guile> (let ((iszero?
              (lambda(x)
              (if (= x 0) #t #f))))
       (iszero? 9))
#f
guile> (iszero? 0)  ;此时会显示出错信息
```

let 的绑定在过程内有效，过程外则无效，这和上面提到的过程的嵌套定是一样的，上面的 iszero?过程在操作过程内定义并使用的，操作结束后再另行引用则无效，显示过程未定义出错信息。

下面操作演示了 let*的用法：

```
guile> (let ((x 2) (y 5))
      (let* ((x 6)(z (+ x y)))  ;此时 x 的值已为 6，所以 z 的值应为 11，如此最后的值为 66
      (* z x)))
66
```

还有 letrec，看下面的操作过程：

```
guile> (letrec ((even?
              (lambda(x)
              (if (= x 0) #t
                     (odd? (- x 1)))))
          (odd?
              (lambda(x)
              (if (= x 0) #f
                     (even? (- x 1))))))
       (even? 88))
#t
```

上面的操作过程中，内部定义了两个判断过程 even?和 odd?，这两个过程是互相递归引用的，如果将 letrec 换成 let 或 let*都会不正常，因为 letrec 是将内部定义的过程或变量间进行相互引用的。看下面的操作：

```
guile> (letrec ((countdown
              (lambda (i)
              (if (= i 0) 'listoff
                     (begin (display i) (display ",")
                            (countdown (- i 1)))))))
       (countdown 10))
10,9,8,7,6,5,4,3,2,1,listoff
```

letrec 帮助局部过程实现递归的操作，这不仅在 letrec 绑定的过程内，而且还包括所有初始化的东西，这使得在编写较复杂的过程中经常用到 letrec，也成了理解它的一个难点。

### apply

apply 的功能是为数据赋予某一操作过程，它的第一个参数必需是一个过程，随后的其它参数必需是列表，如：

```
guile> (apply + (list 2 3 4))
9
```

```
guile> (define sum
         (lambda (x )
               (apply + x)))  ; 定义求和过程
guile> sum
#<procedure sum (x)>
guile> (define ls (list 2 3 4 5 6))
guile>ls
(2 3 4 5 6)
guile> (sum ls)
20
guile> (define avg
        (lambda(x)
              (/ (sum x) (length x))))   ; 定义求平均过程
guile>avg
#<procedure avg (x)>
guile> (avgls)
4
```

以上定义了求和过程 sum 和求平均的过程 avg，其中求和的过程 sum 中用到了 apply 来绑定"+"过程操作到列表，结果返回列表中所有数的总和。

**map**

map 的功能和 apply 有些相似，它的第一个参数也必需是一个过程，随后的参数必需是多个列表，返回的结果是此过程来操作列表后的值，如下面的操作：

```
guile> (map + (list 1 2 3) (list 4 5 6))
(5 7 9)
guile> (map car '((a . b)(c . d)(e . f)))
(a c e)
```

除了 apply，map 以外，Scheme 语言中还有很多，诸如：eval，delay，for-each，force，call-with-current-continuation 等过程绑定的操作定义，它们都无一例外的提供了相当灵活的数据处理能力，也就是另初学者望而生畏的算法，当你仔细的体会了运算过程中用到的简直妙不可言的算法后，你就会发现 Scheme 语言设计者的思想是多么伟大。

# 四．输入输出

Scheme 语言中也提供了相应的输入输出功能，是在 C 基础上的一种封装。

**端口**

Scheme 语言中输入输出中用到了端口的概念，相当于 C 中的文件指针，也就是 Linux 中的设备文件，请看下面的操作：

```
guile> (current-input-port)
#<input: standard input /dev/pts/0> ;当前的输入端口
guile> (current-output-port)
```

```
#<output: standard output /dev/pts/0> ;当前的输出端口
```

判断是否为输入输出端口,可以用下面两个过程 :input-port? 和 output-port? ,其中 input-port? 用来判断是否为输入端口，output-port?用来判断是否为输出端口。

open-input-file，open-output-file，close-input-port，close-output-port 这四个过程用来打开和关闭输入输出文件，其中打开文件的参数是文件名字符串，关闭文件的参数是打开的端口。

**输入**

打开一个输入文件后，返回的是输入端口，可以用 read 过程来输入文件的内容：

```
guile> (define port (open-input-file "readme"))
guile> port
#<input: readme 4>
guile> (read port)
GUILE 语言
```

上面的操作打开了 readme 文件，并读出了它的第一行内容。此外还可以直接用 read 过程来接收键盘输入，如下面的操作：

```
guile> (read) ; 执行后即等待键盘输入
12345
12345
guile> (define x (read)) ; 等待键盘输入并赋值给 x
12345
guile> x
12345
```

以上为用 read 来读取键入的数字，还可以输入字符串等其它类型数据：

```
guile> (define name (read))
tomson
guile> name
tomson
guile> (string? name)
#f
guile> (symbol? name)
#t
```

此时输入的 tomson 是一个符号类型，因为字符串是用引号引起来的，所以出现上面的情况。下面因为用引号了，所以(string? str)返回值为#t 。

```
guile> (define str (read))
"Johnson"
guile>str
"Johnson"
guile> (string? str)
#t
```

还可以用 load 过程来直接调用 Scheme 语言源文件并执行它，格式为：(load "filename")，还有 read-char 过程来读单个字符等等。

**输出**

常用的输出过程是 display，还有 write，它的格式是：(write 对象端口)，这里的对象是指字符串等常量或变量，端口是指输出端口或打开的文件。下面的操作过程演示了向输出文件 temp 中写入字符串 "helloworld"，并分行的实现。

```
[root@toymouse test]# guile
guile> (define port1 (open-output-file "temp"))  ; 打开文件端口赋于 port1
guile> port1
#<output: temp 3>
guile> (output-port? port1)
#t                   ; 此时证明 port1 为输出端口
guile> (write "hello\\nworld" port1)
guile> (close-output-port port1)
guile> (exit)              ; 写入数据并关闭退出
[root@toymouse test]# more temp          显示文件的内容，达到测试目的
"hello
world"
```

在输入输出操作方面，还有很多相关操作，读者可以参考 R5RS 的文档。

# 五．语法扩展

Scheme 语言可以自己定义象 cond，let 等功能一样的宏关键字。标准的 Scheme 语言定义中用 define-syntax 和 syntax-rules 来定义，它的格式如下：

```
                (define-syntax 宏名
                        (syntax-rules()
                                ((模板) 操作))
                                . . . ))
```

下面定义的宏 start 的功能和 begin 相同，可以用它来开始多个块的组合：

```
(define-syntax start
      (syntax-rules ()
            ((start exp1)
                  exp1)
            ((start exp1 exp2 ...)
                  (let ((temp exp1)) (start exp2 ...))) ))
```

这是一个比较简单的宏定义，但对理解宏定义来说是比较重要的，理解了他你才会进一步应用宏定义。在规则 ((start exp1) exp1) 中，(start exp1) 是一个参数时的模板，exp1 是如何处理，也就是原样搬出，不做处理。这样 (start form1) 和 (form1) 的功能就相同了。
在规则 ((start exp1 exp2 ...) (let ((temp exp1)) (start exp2 ...))) 中，(start exp1 exp2 …) 是多个参数时的模板，首先用 let 来绑定局部变量 temp 为 exp1，然后用递归实现处理多个参数，注意这里说的是宏定义中的递归，并不是过程调用中的递归。另外在宏定义中可以用省略号（三个点）来代表多个参数。

在 Scheme 的规范当中，将表达式分为原始表达式和有源表达式，Scheme 语言的标准定义中只有原始的 if 分支结构，其它均为有源型，即是用后来的宏定义成的，由此可见宏定义的重要性。附上面的定义在 GUILE 中实现的 代码。

# 六．其它功能

**1．模块扩展**

在 R5RS 中并未对如何编写模块进行说明，在诸多的 Scheme 语言的实现当中，几乎无一例外的实现了模块的加载功能。所谓模块，实际就是一些变量、宏定义和已命名的过程的集合，多数情况下它都绑定在一个 Scheme 语言的符号下（也就是名称）。在 Guile 中提供了基础的 ice-9 模块，其中包括 POSIX 系统调用和网络操作、正则表达式、线程支持等等众多功能，此外还有著名的 SFRI 模块。引用模块用 use-modules 过程，它后面的参数指定了模块名和我们要调用的功能名，如：(use-modules (ice-9 popen))，如此后，就可以应用 popen 这一系统调用了。如果你想要定义自己的模块，最好看看 ice-9 目录中的那些 tcm 文件，它们是最原始的定义。

另外 Guile 在面向对象编程方面，开发了 GOOPS( Guile Object-Oriented Programming System )，对于喜欢 OO 朋友可以研究一下它，从中可能会有新的发现。

**2．如何输出漂亮的代码**

如何编写输出漂亮的 Scheme 语言代码应该是初学者的第一个问题，这在 Guile 中可以用 ice-9 扩展包中提供的 pretty-print 过程来实现，看下面的操作：

```
guile> (use-modules (ice-9 pretty-print))    ; 引用漂亮输出模块
guile> (pretty-print '(define fix (lambda (n)
       (cond ((= n 0) 'iszero)
             ((< n 0) 'lower)
             (else 'upper)))))   ; 此处是我们输入的不规则代码
(define fix
  (lambda (n)
    (cond ((= n 0) 'iszero)
          ((< n 0) 'lower)
          (else 'upper))))     ; 输出的规则代码
```

**3．命令行参数的实现**

在把 Scheme 用做 shell 语言时，经常用到命令行参数的处理，下面是关于命令行参数的一种处理方法：

```
#! /usr/local/bin/guile -s
!#
(define cmm (command-line))
(display "应用程序名称：")
(display (car cmm))
(newline)
(define args (cdrcmm))
(define long (length args))
```

```
(define loop (lambda (count lenobj)
         (if (<= count len)
                  (begin
                           (display "参数 ")
                           (display count)
                           (display " 是 : ")
                           (display (list-ref obj (- count 1)))
                           (newline)
                           (set! count (+ count 1))
                           (loop count lenobj)))))
(loop 1 long args)
```

下面是运行后的输出结果：

```
[root@toymouse doc]# ./tz.scmabc 123 ghi
应用程序名称 : ./tz.scm
参数 1 是 : abc
参数 2 是 : 123
参数 3 是 : ghi
```

其中最主要的是用到了 command-line 过程，它的返回结果是命令参数的列表，列表的第一个成员是程序名称，其后为我们要的参数，定义 loop 递归调用形成读参数的循环，显示出参数值，达到我们要的结果。

**4．特殊之处**

**一些精确的自己计算自己的符号**

```
数字      Numbers           2 ==> 2
字符串    Strings           "hello" ==> "hello"
字符      Charactors        #\\g ==> #\\g
辑值    Booleans    #t ==> #t
量表      Vectors           #(a 2 5/2) ==> #(a 2 5/2)
```

**通过变量计算来求值的符号**

如：

```
         x ==> 9
           -list ==> ("tom" "bob" "jim")
         factoral ==> #<procedure: factoral>
                   ==> #<primitive: +>
```

**define 特殊的 form**

(define x 9) ，define 不是一个过程，它是一个不用求所有参数值的特殊的 form，它的操作步骤是，初始化空间，绑定符号 x 到此空间，然后初始此变量。

**必须记住的东西**

下面的这些定义、过程和宏等是必须记住的：

define，lambda，let，lets，letrec，quote，set!，if，case，cond，begin，and，or 等等，当然还有其它宏，必需学习，还有一些未介绍，可参考有关资料。
走进 Scheme 语言的世界，你就发现算法和数据结构的妙用随处可见，可以充分的检验你对算法和数据结构的理解。Scheme 语言虽然是古老的函数型语言的继续，但是它的里面有很多是在其它语言中学不到的东西，我想这也是为什么用它作为计算机语言教学的首选的原因吧。