



# 函数式程序设计

郭 炜

<http://weibo.com/guoweiofpku>

<http://blog.sina.com.cn/u/3266490431>

guo\_wei@pku.edu.cn

# 课程安排

教材：《计算机程序的构造和解释》

Harold Abelson      Gerald Jay Sussman  
Julie Sussman 著  
裘宗燕 译      机械工业出版社

课程网站：北大教学网

作业和考试网站： [lisp.test.openjudge.org](http://lisp.test.openjudge.org)

课本网站： [http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%\\_sec\\_1.3](http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-12.html#%_sec_1.3)



# 课程安排

## 成绩组成：

- 期末机考50%
- 平时作业35-40%
- 课堂测验10-15%

课堂测验可以任选一次课的去掉不算。选择题必需一次过。

# 课程安排

- 平时作业35-40%

出题作业：有的章节是每人都要出，有的章节是大家轮流出。题目质量和得分相关。



北京大学  
PEKING UNIVERSITY

信息科学技术学院《函数式程序设计》 郭炜

# 第一讲

# 什么是函数式程序设计

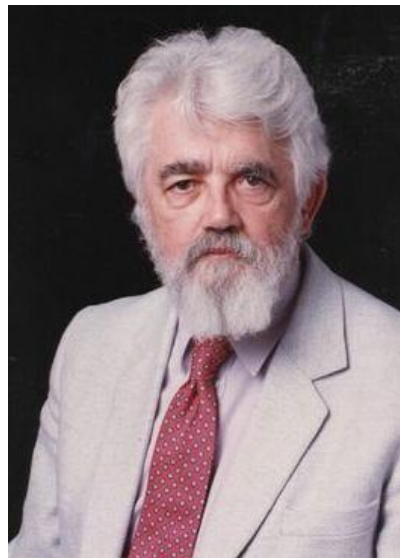
函数式程序设计 (Functional programming) 将计算机的计算视为数学上的函数计算，函数的计算结果只和函数的自变量（参数）有关，与其他一切（比如何时被调用，调用了多少次）无关。函数式程序设计避免使用程序状态(值可变的变量)。函数编程语言最重要的基础是 $\lambda$  演算 (lambda calculus)。  $\lambda$  演算是一种等价于图灵机的演算系统。 $\lambda$  演算的函数可以接受函数当作输入（参数）和输出（返回值）。

# 什么是函数式程序设计

“函数式编程”是一种“编程范式”（programming paradigm），也就是如何编写程序的方法论。它属于“结构化编程”的一种，主要思想是把运算过程尽量写成一系列嵌套和递归的函数调用。纯的函数式设计语言没有赋值操作，没有语句，只有表达式，几乎一切计算都是递归。有分支的概念，没有循环的概念。

# 函数式程序设计语言的发明

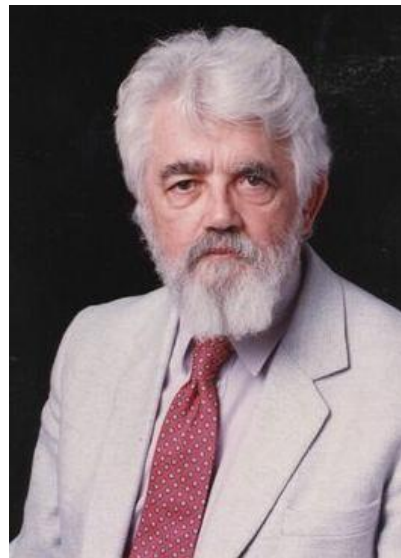
- 1958年，约翰·麦卡锡(John McCarthy)在麻省理工学院发明Lisp，通过Lisp，他证明了，图灵完备的系统可以仅仅由几个简单的算子与函数定义功能组成。
- 后来他获得图灵奖。





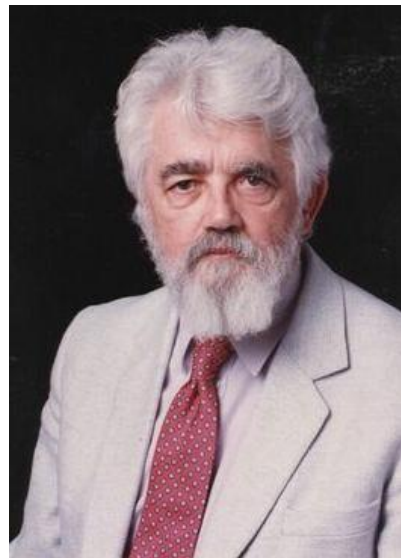
# 函数式程序设计语言的发明

- 约翰·麦卡锡并未将Lisp当作程序设计语言，他只是想用更简单的方式定义图灵机
- 约翰·麦卡锡说：Lisp比图灵机表达起来更简洁。证明这一点的一种方法就是写一个Lisp通用函数，证明它比图灵机的一般描述更短，更易懂。这个Lisp函数就是eval…。它用来计算Lisp表达式的值……。编写eval函数需要发明一种表示法，能够把Lisp函数表示成Lisp数据。设计这种书写法完全是为了满足论文写作的需要。根本没有想过用它来编写程序并在计算机上运行。
- 因此Lisp本质上是数学，数学是不会过时的



# 函数式程序设计语言的发明

- 约翰·麦卡锡的学生史蒂夫·拉塞尔把eval函数翻译成机器语言，并进一步做出了Lisp的解释器。
- 1962年，提姆·哈特（Tim Hart）与麦克·莱文（Mike Levin）在麻省理工学院，用Lisp语言，实做出第一个完整的lisp编译器。Lisp成为一种出乎意料强大的语言。



# 函数式程序设计语言的发展

- Scheme的发明

Scheme最早由麻省理工学院的盖伊·史提尔二世与杰拉德·杰伊·萨斯曼（Sussman）在1970年代发展出来。Scheme语言与 $\lambda$ 演算关系十分密切。小写字母“ $\lambda$ ”是Scheme语言的标志。本来名为Schemer，后因操作系统字数限制改为Scheme。

Scheme遵循极简主义哲学，以一个小型语言核心作为标准，加上各种强力语言工具（语法糖）来扩展语言本身。

Scheme不是纯函数式的，有赋值语句，甚至全局变量。

Scheme是极度简化的语言。他的规范文档不过47页。Common Lisp上千页。

# 函数式程序设计语言的发展

- 其他函数式程序设计语言

**Common Lisp** : 试图将Lisp的各种方言标准化

**Miranda**: 惰性求值的纯粹函数编程语言，由英国学者大卫·特纳（David Turner）所设计。1985年由英国的研究软件公司（Research Software Ltd.）发布

**Haskell**: 纯函数式，1990年在编程语言Miranda的基础上标准化而来。  
现在比较流行

**Ocaml**: 非纯函数式，1996年发布，效率可比C/C++

**F#**: 微软公司用于.net平台开发的函数是程序设计语言，同时对OO也有很好支持

# 函数式程序设计语言的发展

- 具有部分函数式语言特征的程序设计语言

**Python**：面向对象的解释型语言，但也支持函数是程序设计

**Rubby**：日本人源于Python和Perl的发明

**C++ 11**：引入 $\lambda$ 表达式

**JavaScript**

.....

函数式程序设计方法近年来越来越受重视！

参考阅读：<http://blog.csdn.net/g9yuayon/article/details/1676688>

# 函数式程序设计为什么重要

命令式程序设计语言：有赋值，if，goto就够用了。难于建立数学模型来分析程序的正确性。

数学上的函数只要自变量相同，结果总是相同。但是过程式设计语言的函数不是这样。

应寻求与求值顺序(时间)无关的程序表达方式，才能不用运行，从数学上就判断程序的正确性

函数式语言：变量，表达式，条件表达式，递归机制

# 函数式程序设计为什么重要

《Why Functional Programming Matters》，1990

<http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>

译文

<https://www.byvoid.com/blog/why-functional-programming/>

本文提到的FP最重要特点：高阶函数和惰性求值。

本文提到的FP最重要优点：程序便于模块化且简洁。

# 函数式程序设计为什么重要

《黑客与画家》 Paul Graham著， 阮一峰译

Paul Graham, 硅谷创业之父, 1995年和Robert Morris创建Viaweb公司, 1998年被Yahoo以4900万美元收购, 现从事风险投资 (Y combinator, YC) 和创业辅导。



# 函数式程序设计为什么重要

## Paul Graham 说:

- Lisp是最强大的语言，现在的主流高级语言只是接近1958年Lisp的水平。  
(如果一种特性在A语言中是内置，在B语言中需要修改解释器才能实现，那A就比B强大)
- Viaweb在竞争中获胜的关键在于使用Lisp语言进行开发，因而能比对手更快地实现各种新功能。
- C代码的长度是Lisp的7-10倍
- ITA公司成功的秘诀在于他们系统中20万行的Common Lisp核心程序。ITA的总裁说1行Lisp代码相当于20行C代码。

# 函数式程序设计为什么重要

Paul Graham 说：

我说使用其他语言无法提供Lisp的“高阶函数”这个功能，这句话并不完全正确。所有这些语言都是图灵等价的，这意味着严格地说，你能使用它们之中的任何一种语言，写出任何一个程序。那么，怎样才能做到这一点呢？就这个小小的例子而言，你可以使用这些不那么强大的语言，写一个Lisp解释器就行了。

# 函数式程序设计为什么重要

Paul Graham 说：

这样做听上去好像开玩笑，但是在大型编程项目中，却不同程度地广泛存在。因此，有人把它总结出来，起名为"格林斯潘第十定律"

(Greenspun's Tenth Rule)：

"任何C或Fortran程序复杂到一定程度之后，都会包含一个临时开发的、只有一半功能的、不完全符合规格的、到处都是bug的、运行速度很慢的Common Lisp实现。"

# 函数式程序设计语言的特点

1. “函数”是一等公民 (first class)，即有“高阶函数的概念”

函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

# 函数式程序设计语言的特点

高阶函数：

```
(define (f n)
  (lambda (x) (+ x n)))
```

`(f n)` 返回一个函数，该函数接收一个参数 $x$ ，返回  $x+n$

`((f 7) 9)`                      输出： 16

# 函数式程序设计语言的特点

高阶函数: (define (f n) (lambda (x) (+ x n)))

C++实现:

```
template <class T1, class T2>
```

```
struct Add {
```

```
    T1 n;
```

```
    auto operator() ( T2 x ) -> decltype(x + n)
```

```
    {          return x + n;    }
```

```
    Add(T1 n_) : n(n_) { }
```

```
};
```

```
template <class T1, class T2>
```

```
Add<T1, T2> f (T1 n) {
```

```
    return Add<T1, T2>(n);
```

```
}
```

```
int main()
```

```
{
```

```
    cout << f<int, int>(7)(9) << endl; //16
```

```
    cout << f<string, string> (" hello!")("world")
```

```
        << endl; // world hello!
```

```
    cout << f<char, string> (!)("world") << endl;
```

```
    return 0; //world!
```

```
}
```

# 函数式程序设计语言的特点

高阶函数:

```
(define (square x)  (* x x))  
(define (inc x)    (+ x 1))
```

```
(define (combine f g)  
  (lambda (x) (f (+ (f x) (g x))))))
```

`(combine f g)` 返回函数  $k$ ,  $k(x) = f(f(x)+g(x))$

`((combine square inc) 3)` 输出169

# 函数式程序设计语言的特点

combine的C++实现:

```
template <class T1,class T2,class T3>
struct Do
{
    T1 f;  T2 g;
    auto operator() ( T3 x ) ->
decltype(f(f(x)+g(x))) {
        return f(f(x)+g(x));
    }
    Do(T1 f_,T2 g_):f(f_),g(g_) { }
};

template <class T1,class T2,class T3>
Do<T1,T2,T3> combine(T1 f,T2 g,T3 nouse) {
    return Do<T1,T2,T3>(f,g);
}
```

```
int main()
{
    auto Square = [] (int a) { return a * a; };
    auto Inc = [] (int a) { return a + 1; };
    cout << combine(Square,Inc,1234 )(3) << endl;
    return 0;
} //输出: 169
```



# 函数式程序设计语言的特点

## 2. 只用“表达式”，不用“语句”

表达式是单纯的计算，有返回值。语句没有返回值。

## 3. 没有副作用，易于并行和判断正确性

函数的运行结果只和函数的参数有关，不管何时调用，都是同一结果。函数不应修改函数外面的变量的值。函数的功能就是返回一个值，没有其他作用。

## 4. 不修改状态，即不修改变量，无赋值语句

碰到需要修改的情况，就重新构造一个

# 函数式程序设计语言的特点

5. 有分支但没有循环，一切靠递归

6. 支持惰性求值

需要的时候才进行求值

7. 代码的热升级

函数式编程没有副作用，只要保证接口不变，内部实现是外部无关的。所以，可以在运行状态下直接升级代码，不需要重启，也不需要停机。

8. 代码和数据的形式相同

`(t 1 2 3)` 代表调用`t`函数，以`1 2 3`为参数，`'(t 1 2 3)`表示列表

# 函数式程序设计语言的劣势

1. 不接近自然。自然界中副作用是真是存在的
2. 往往运行效率较低
3. 纯函数式程序设计在解决有些命令式能轻易解决的问题时，也要大费周折。
4. 计算机的内存本来就是可以改写的，硬要不去改写它，也许就是自寻烦恼。
5. 不一定符合现代软件开发的需要

# 最简单的Scheme程序

```
(display “hello, world!”)
```

输出：hello, world!

```
(define X 1000)
```

```
X
```

输出：1000

# Scheme的注释

单行注释，以 ‘;’ 开头，直到行末

```
(define (f n) ;(f n) 返回函数k ,  $k(x) = n + x$   
  (lambda (x) (+ n x)))
```

# Scheme的变量定义和修改

- 变量定义:

(define 变量名 值)

如: (define x 100) , 定义变量x, 其值为100。

- 修改变量的值:

(set! 变量名 值)

如: (set! x "hello") , 将变量x的值改为"hello" 。

- 变量类型可变

(define x 123)

(set! x "abc")

x ; 输出 abc

# Scheme的数据类型

- 简单数据类型

- 逻辑型 (boolean)

- 只有两个值, `#t` 和 `#f`

- 只有一个运算 `not`

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

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

- `(not 其他) => #f`

- `(not 1), (not "ok"), (not '(1 2 3)) ... => #f`

# Scheme的数据类型

- 简单数据类型

- 数字型(number)

- 整型

- (define x 123)

- (define y #b1101) ;二进制

- (define z #xab12) ;十六进制

- 有理数型

- (define a 2/13) ;输出 a 则得 2/13

- 实数型

- (define Pi 3.14)

- 复数型

- (define f 3+2i)

```
(define kxx 2+3i)
(+ kxx 1+2i) ;输出3+5i
```



# Scheme的数据类型

- 简单数据类型

- 字符型 (char)

- 以 “#\” 开头

- #\A    #\[    #\0    #\!    #\space    #\newline

# Scheme的数据类型

- 简单数据类型

- 符号型 (symbol)

```
(define x1 (quote pku))
```

x1 ; 输出 'pku

```
(define x2 'pku)
```

x2 ; 输出 'pku

符号型变量和字符串不同，不能取长度，不能改其中字符。

# Scheme的数据类型

- 复合数据类型

- 字符串 (string)

```
(define str "Hello")  
str  
(string-length str) ; 取字符串的长度  
(string-ref str 1) ; 取第1个字符 (从0开始算)
```

输出:  
"Hello"  
5  
#\e

字符串中的引号用反斜线加引号代替，如 “123\“abc”。

# Scheme的数据类型

- 复合数据类型

- 对子 (pair)

- (cons a b) 即形成一个“对子”，一个对子由两部分构成  
car 取前部，cdr取后部

- set-car!修改前部，set-cdr!修改后部

Racket中，要生成可修改的对子，以及修改对子，则使用  
mcons , mset-car! , mset-cdr!

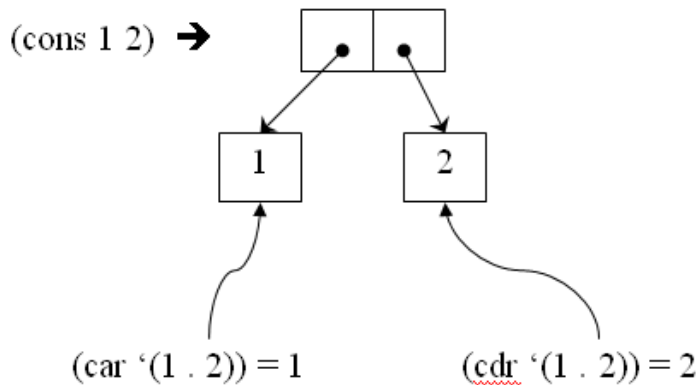
# Scheme的数据类型

- 复合数据类型

- 对子 (pair)

(cons a b) 即形成一个“对子”，一个对子由两部分(两个指针) 构成  
car 取前部，cdr取后部

set-car!修改前部， set-cdr!修改后部



# Scheme的数据类型

- 复合数据类型

- 对子 (pair)

(cons a b) 即形成一个“对子”，一个对子由两部分构成  
car 取前部，cdr取后部  
set-car!修改前部，set-cdr!修改后部

```
(define p (cons "good" 100))  
p  
(car p)  
(cdr p)
```

输出：  
'("good" . 100)  
"good"  
100

# Scheme的数据类型

- 复合数据类型

- 对子 (pair)

Racket中，要生成可修改的对子，以及修改对子，则使用 `mcons` , `mcar`, `mcdrr`, `set-mcar!`, `set-mcdr!` , 并且在程序开头写：

```
(require scheme/mpair)
```

```
(define p2 (mcons "ok" "20"))  
p2  
(set-mcar! p2 30)  
(set-mcdr! p2 "bad")  
p2
```

输出：

```
(mcons "ok" "20")  
(mcons 30 "bad")
```

# Scheme的数据类型

- 复合数据类型

- 列表 (list)

列表是由多个类型相同或不同的数据连续组成的数据类型。

```
(define lst (list 1 2 3 4 ))
```

```
lst
```

```
(length lst) ; 取得列表的长度
```

```
(list-ref lst 3) ; 取得列表第3项的值 (从0开始)
```

```
(define y (make-list 5 "a")) ; 创建列表
```

```
y
```

输出:

```
'(1 2 3 4)
```

```
4
```

```
4
```

```
'("a" "a" "a" "a" "a")
```



# Scheme的数据类型

- 复合数据类型

- 列表 (list)

列表是由多个类型相同或不同的数据连续组成的数据类型。

car用于取列表第一项，cdr取列表的剩余部分

```
(define lst (list 1 2 3 4 5))
```

```
(car lst)
```

```
(cdr lst)
```

```
(cadr lst)
```

```
(caddr lst)
```

```
(cadddr lst)
```

```
(cddr lst)
```

```
(cddddr lst)
```

```
(cddddr lst)
```

输出:

1

'(2 3 4 5)

2

3

4

'(3 4 5)

'(4 5)

'(5)

# Scheme的数据类型

- 复合数据类型

- 列表 (list)

多重列表：列表的元素可以是列表

```
(list (list 1 2 3) 4 (list "a" 200))
```

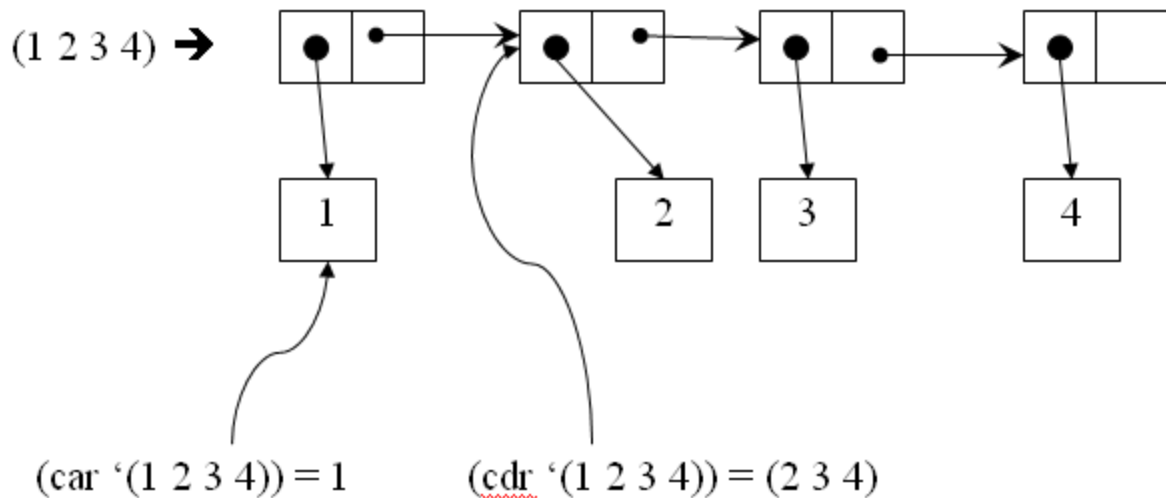
```
=> '((1 2 3) 4 ("a" 200))
```

# Scheme的数据类型

- 复合数据类型

- 列表 (list)

列表和对子的关系：列表就是一个对子。对子的前部是列表的car，后部是列表的cdr。

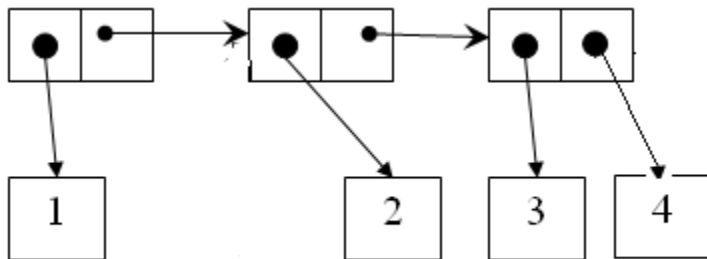


# Scheme的数据类型

- 复合数据类型

- 对子 (pair)

`(cons 1 (cons 2 (cons 3 4)))` ➔



# Scheme的数据类型

## ● 复合数据类型

### ➤ 向量 (vector)

类似于数组。

```
(define v (vector 1 2 3 4 5))
```

```
v
```

```
(vector-ref v 0) ; 求第0个元素的值
```

```
(vector-length v) ; 求vector的长度
```

```
(vector-set! v 2 "abc") ; 设置vector第2个元素的值
```

```
v
```

```
(define x (make-vector 5 "a")) ; 创建向量表
```

```
x
```

```
 #(12 xyz 34)
```

输出:

```
'#(1 2 3 4 5)
```

```
1
```

```
5
```

```
'#(1 2 "abc" 4 5)
```

```
'#("a" "a" "a" "a" "a")
```

```
'#(12 xyz 34)
```

# Scheme的运算符

## ● 类型判断运算符

(boolean? #t) => #t  
(boolean? #f) => #t  
(boolean? 2) => #f

(char? #f) => #f  
(char? #\?) => #t

(integer? 3) => #t  
(integer? 2/3) => #f

以下全为 #t  
(rational? 3)  
(rational? 2/3)  
(rational? 1.5)

(real? 3)  
(real? 2/3)

(number? 2/3)  
(number? 1.5)  
(number? 4)

(null? '()) ;#t  
(null? 5) ;#f  
(define x 124)  
(symbol? x) ;#f  
(symbol? 'x) ;#t

# Scheme的运算符

## ● 比较运算符

>, <, >=, <=, = 用于判断数字类型的变量或表达式之间的关系

```
(< 4 5) => #t
```

```
(define x 13)
```

```
(= x 12) => #f
```

# Scheme的运算符

## ● 比较运算符

`eq?` 判断两个参数是否指向同一个对象

`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。多用来判断对子，列表，向量表，字符串等复合结构数据类型。

```
(define v (list 1 2 3))  
(define w (list 1 2 3))  
(eq? v w) ;#f  
(equal? v w) ;#t
```



# Scheme的运算符

- 算术运算符

$+$ ,  $-$ ,  $*$ ,  $/$ ,

abs, remainder, quotient .....

各种运算符参考:

<http://zh.wikipedia.org/wiki/Scheme>

# Scheme的类型转换

“->” 表示类型间的转换

```
(number->string 123) ; 数字转换为字符串, ;"123"  
(string->number "456") ; 字符串转换为数字 ;456  
(char->integer #\a) ;字符转换为整型数, a的ASCII码为96 ;97  
(char->integer #\A) ;A的ASCII码为65 ;65  
(integer->char 97) ;整型数转换为字符 ;#\a  
(string->list "hello") ;字符串转换为列表 ;(#\h #\e #\l #\l #\o)  
(list->string (make-list 4 #\a)) ; 列表转换为字符串 ;"aaaa"  
(string->symbol "good") ;字符串转换为符号类型 ;good  
(symbol->string 'better) ;符号类型转换为字符串 ;"better"
```

# Scheme的表达式

## ●简单表达式

数是基本表达式 ( $\Rightarrow$ 表示输出的结果, 不是表达式的一部分)

**235  $\Rightarrow$  235**

简单算术表达式 (简单组合式)

**(+ 137 248)  $\Rightarrow$  385**

**(+ 2.9 10)  $\Rightarrow$  12.9**

表达式都是**带括号的前缀形式**。括号里第一个元素表示操作 (运算符), 后面是参数 (操作数) 操作数和参数之间、不同参数之间用空格分隔

# Scheme的表达式

- 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))

- 可以写任意复杂的组合表达式，如

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

- IDE会自动对表达式缩进以便于阅读：

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

- 换行和空格符号不影响表达式的意义

# Scheme的变量

用define 定义变量，且将其初始化

```
(define size 10)
```

变量没有类型，不能进行静态类型检查。因此其值的“类型”可变

```
(define x 10)  
set! x "hello"
```

# Scheme的变量和环境

scheme解释器在“环境”中存储变量名和其值的对应关系。

**define、set!** 建立或修改环境中的名字-值关联

“环境”有多层，表达式在当前环境求值，变量的值由环境中查到

**Scheme** 全局环境预先定义了一批名字-对象关联

全局环境中有一些预先定义的变量，主要是各种过程，如`+`,`-`,`*`,`/`,  
`abs`,`list`等

# 组合式的求值

## ●组合式的求值规则：

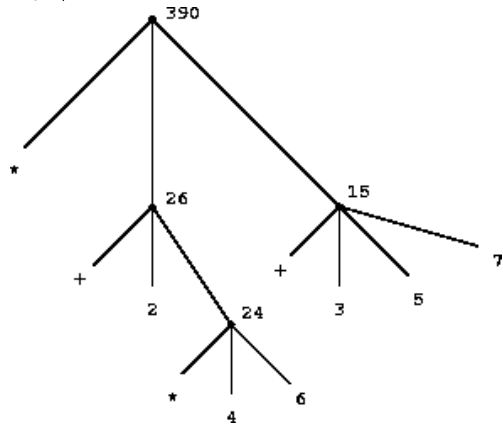
- 求值该组合式的各个子表达式(一般是从左到右)
- 最左子表达式通常会 是运算符或者函数名，其求值的结果就是一个过程。此种情况下以其他子表达式的值作为参数调用该过程。

## ● 例：

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

- 表达式求值的过程是递归的

- 求值过程可以用树表示，先取得叶结点（基本表达式）的值后向上累积



# 组合式的求值

- 常数的值是其自身（它们所表示的数值）
- 内部运算符(+, -, remainder)等的值是系统实现相关运算的指令序列
- 其他名字的值由当前环境取得，找到相应名字-值关联时取出对应的值，找不到就是错误



# 特殊形式

- 有些表达式里的子表达式不应该求值，这样的表达式称为“特殊形式” (special form)。例：

```
(define x 1)
```

`x` 不应该求值，是要求为名字`x`关联一个新值。

`scheme`有多种“特殊形式”的表达式，如`define`，`if`，`cond`等。每个特殊形式有自己的求值规则。

# 过程定义

重复使用的表达式，可以定义成一个“过程”，以便日后方便使用，简化程序的编写。可以用自定义的过程再定义新过程

```
(define (square x) (* x x))  
(square 3) ;=>9  
(square (+ 2 4)) ;=>36  
(define (sum-of-squares x y)  
  (+ (square x) (square y)))
```

```
(sum-of-squares 1 4) ;=>17  
(define (f a)  
  (sum-of-squares (+ a 1) (* a 2)))  
(f 5) ;=> 136
```

# 应用序和正则序求值

● **应用序求值**：解释器先求值子表达式（运算符和各操作数），而后把得到的运算符应用于操作数（实际参数），即先求值参数后应用运算符。

求值 (f 5) :

```
(sum-of-squares (+ a 1) (* a 2))
```

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

```
(+ (square 6) (square 10))
```

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

```
(+ 36 100)
```

136

# 应用序和正则序求值

●正则序求值：先不求值操作数，推迟到必需时再求值：

●求值(f, 5)：

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

；先展开，后归约

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

136

正则序求值导致重复计算。  
Scheme采用应用序求值。

# 条件表达式

cond 条件表达式的一般形式:

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

依次求  $p_1, p_2 \dots p_n$ 。碰到第一个为真的  $p_i$ ，就求出执行对应的  $e_i$  作为返回值。  
都不满足，则无返回值。

# 条件表达式

cond 条件表达式的一般形式:

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

依次求  $p_1, p_2 \dots p_n$ 。碰到第一个为真的  $p_i$ ，就求出执行对应的  $e_i$  作为返回值。都不满足，则求出 **else** 对应的  $e_s$  作为返回值。

< $e_i$ > 中可以有多多个表达式，< $p_i$ > 为真时，依次求值，最后一个表达式的值就是  $e_i$  的值)

# 条件表达式

定义abs函数的两种方法：

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

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

# 条件表达式

if ... 条件表达式:

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

<predicate> 成立，则返回值为 <consequent>，否则返回值为<alternative>

```
(if (> 5 3)
    "ok"
    "not ok")    ;=> "ok"
```

```
(define a 6)
(if (> a 5)
    (if (< a 7)
        "good"
        "not good enough")
    "bad")    ;=> "good"
```



# 谓词

(and p1 p2 p3 p4 ...)

(or p1 p2 p3 p4 ...)

(not p)

and 和 or 都是短路计算

cond, if , and , or 都是特殊形式, 有特殊求值规则

# void函数

void库函数什么都不做，什么都不返回

```
(if (> x 3)
    (display x)
    (void))
```

$x > 3$ 时才输出 $x$ , 否则就什么都不做

# begin表达式

*(begin <exp1> <exp2> ...<expn> )*

依次求值<exp1> ... <expn>。

begin表达式的返回值是expn的值

```
(if (> x 3)
  (begin (display "x=") (display x) (newline) x)
  (void))
```

# begin表达式

*(begin <exp1> <exp2> ...<expn> )*

依次求值<exp1> ... <expn>。  
begin表达式的返回值是expn的值

# 例:牛顿迭代法求平方根

为求 $x$ 的平方根, 猜测一个值  $y_1$  作为根, 然后每次执行  $y_{i+1} = (y_i + x/y_i) / 2$  直到某  $y_i$  的平方和 $x$ 差距足够小(也可以是两 $y_i$ 和 $y_{i+1}$ 差距足够小)

# 例:牛顿迭代法求平方根

```
(define (square x) (* x x))  
(define (average a b) (/ (+ a b) 2))
```

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

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

```
(define (sqrt-iter guess x) ;guess是猜测值  
  (if (good-enough? guess x)  
      guess  
      (sqrt-iter (improve guess x)  
                  x)))  
(define (sqrt x) (sqrt-iter 1.0 x))
```

为求 $x$ 的平方根, 猜测一个值  $y_1$  作为根, 然后每次执行

$$y_{i+1} = (y_i + x/y_i) / 2$$

直到某  $y_i$  的平方和 $x$ 差距足够小 (也可以是两 $y_i$ 和 $y_{i+1}$ 差距足够小)

```
(sqrt 2)  ;=> 1.4142156862745097  
(sqrt 9)  ;=> 3.00009155413138
```

# 例:牛顿迭代法求平方根

```
(define (square x) (* x x))  
(define (average a b) (/ (+ a b) 2))
```

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

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

```
(define (sqrt-iter guess x) ;guess是猜测值  
  (if (good-enough? guess x)  
      guess  
      (sqrt-iter (improve guess x)  
                  x)))  
(define (sqrt x) (sqrt-iter 1.0 x))
```

为求 $x$ 的平方根, 猜测一个值  $y_1$  作为根, 然后每次执行

$$y_{i+1} = (y_i + x/y_i) / 2$$

直到某  $y_i$  的平方和 $x$ 差距足够小 (也可以是两 $y_i$ 和 $y_{i+1}$ 差距足够小)

```
(sqrt 2)  ;=> 1.4142156862745097  
(sqrt 9)  ;=> 3.00009155413138
```