# Lisp 简明教程

整理人：Chaobs

邮箱：chaobs@outlook.com

博客：www.cnblogs.com/Chaobs

资料主要来源：http://www.yiibai.com/lisp/

版本：0.1.0

# 前言

为什么 **Lisp** 语言如此先进？

（节选自《黑客与画家》中译本）

译者原文:http://www.ruanyifeng.com/blog/2010/10/why_lisp_is_superior.html

一、

如果我们把流行的编程语言，以这样的顺序排列：Java、Perl、Python、Ruby。你会发现，排在越后面的语言，越像 Lisp。

Python 模仿 Lisp，甚至把许多 Lisp 黑客认为属于设计错误的功能，也一起模仿了。至于 Ruby，如果回到 1975 年，你声称它是一种 Lisp 方言，没有人会反对。

编程语言现在的发展，不过刚刚赶上 1958 年 Lisp 语言的水平。

二、

1958 年，John McCarthy 设计了 Lisp 语言。我认为，当前最新潮的编程语言，只是实现了他在 1958 年的设想而已。

这怎么可能呢？计算机技术的发展，不是日新月异吗？1958 年的技术，怎么可能超过今天的水平呢？

让我告诉你原因。

这是因为 John McCarthy 本来没打算把 Lisp 设计成编程语言，至少不是我们现在意义上的编程语言。他的原意只是想做一种理论演算，用更简洁的方式定义图灵机。

所以，为什么上个世纪 50 年代的编程语言，到现在还没有过时？简单说，因为这种语言本质上不是一种技术，而是数学。数学是不会过时的。你不应该把 Lisp 语言与 50 年代的硬件联系在一起，而是应该把它与快速排序（Quicksort）算法进行类比。这种算法是 1960 年提出的，至今仍然是最快的通用排序方法。

三、

Fortran 语言也是上个世纪 50 年代出现的，并且一直使用至今。它代表了语言设计的一种完全不同的方向。Lisp 是无意中从纯理论发展为编程语言，而 Fortran 从一开始就是作为编程语言设计出来的。但是，今天我们把 Lisp 看成高级语言，而把 Fortran 看成一种相当低层次的语言。

1956 年，Fortran 刚诞生的时候，叫做 Fortran I，与今天的 Fortran 语言差别极大。Fortran I 实际上是汇编语言加上数学，在某些方面，还不如今天的汇编语言强大。比如，它不支持子程序，只有分支跳转结构（branch）。

Lisp 和 Fortran 代表了编程语言发展的两大方向。前者的基础是数学，后者的基础是硬件架构。从那时起，这两大方向一直在互相靠拢。Lisp 刚设计出来的时候，就很强大，接下来的二十年，它提高了自己的运行速度。而那些所谓的主流语言，把更快的运行速度作为设计的出发点，然后再用超过四十年的时间，一步步变得更强大。

直到今天，最高级的主流语言，也只是刚刚接近 Lisp 的水平。虽然已经很接近了，但还是没有 Lisp 那样强大。

四、

Lisp 语言诞生的时候，就包含了 9 种新思想。其中一些我们今天已经习以为常，另一些则刚刚在其他高级语言中出现，至今还有 2 种是 Lisp 独有的。按照被大众接受的程度，这 9 种思想依次是：

1. 条件结构（即"if-then-else"结构）。现在大家都觉得这是理所当然的，但是 Fortran I 就没有这个结构，它只有基于底层机器指令的 goto 结构。

2. 函数也是一种数据类型。在 Lisp 语言中，函数与整数或字符串一样，也属于数据类型的一种。它有自己的字面表示形式（literal representation），能够储存在变量中，也能当作参数传递。一种数据类型应该有的功能，它都有。

3. 递归。Lisp 是第一种支持递归函数的高级语言。

4. 变量的动态类型。在 Lisp 语言中，所有变量实际上都是指针，所指向的值有类型之分，而变量本身没有。复制变量就相当于复制指针，而不是复制它们指向的数据。

5. 垃圾回收机制。

6. 程序由表达式（expression）组成。Lisp 程序是一些表达式区块的集合，每个表达式都返回一个值。这与 Fortran 和大多数后来的语言都截然不同，它们的程序由表达式和语句（statement）组成。

区分表达式和语句，在 Fortran I 中是很自然的，因为它不支持语句嵌套。所以，如果你需要用数学式子计算一个值，那就只有用表达式返回这个值，没有其他语法结构可用，因为否则就无法处理这个值。

后来，新的编程语言支持区块结构（block），这种限制当然也就不存在了。但是为时已晚，表达式和语句的区分已经根深蒂固。它从 Fortran 扩散到 Algol 语言，接着又扩散到它们两者的后继语言。

7. 符号（symbol）类型。符号实际上是一种指针，指向储存在哈希表中的字符串。所以，比较两个符号是否相等，只要看它们的指针是否一样就行了，不用逐个字符地比较。

8. 代码使用符号和常量组成的树形表示法（notation）。

9. 无论什么时候，整个语言都是可用的。Lisp 并不真正区分读取期、编译期和运行期。你可以在读取期编译或运行代码；也可以在编译期读取或运行代码；还可以在运行期读取或者编译代码。

在读取期运行代码，使得用户可以重新调整（reprogram）Lisp 的语法；在编译期运行代码，则是 Lisp 宏的工作基础；在运行期编译代码，使得 Lisp 可以在 Emacs 这样的程序中，充当扩展语言（extension language）；在运行期读取代码，使得程序之间可以用 S-表达式（S-expression）通信，近来 XML 格式的出现使得这个概念被重新"发明"出来了。

五、

Lisp 语言刚出现的时候，它的思想与其他编程语言大相径庭。后者的设计思想主要由 50 年代后期的硬件决定。随着时间流逝，流行的编程语言不断更新换代，语言设计思想逐渐向 Lisp 靠拢。

思想 1 到思想 5 已经被广泛接受，思想 6 开始在主流编程语言中出现，思想 7 在 Python 语言中有所实现，不过似乎没有专用的语法。

思想 8 可能是最有意思的一点。它与思想 9 只是由于偶然原因，才成为 Lisp 语言的一部分，因为它们不属于 John McCarthy 的原始构想，是由他的学生 Steve Russell 自行添加的。它们从此使得 Lisp 看上去很古怪，但也成为了这种语言最独一无二的特点。Lisp 古怪的形式，倒不是因为它的语法很古怪，而是因为它根本没有语法，程序直接以解析树（parse tree）的形式表达出来。在其他语言中，这种形式只是经过解析在后台产生，但是 Lisp 直接采用它作为表达形式。它由列表构成，而列表则是 Lisp 的基本数据结构。

用一门语言自己的数据结构来表达该语言，这被证明是非常强大的功能。思想 8 和思想 9，意味着你可以写出一种能够自己编程的程序。这可能听起来很怪异，但是对于 Lisp 语言却是再普通不过。最常用的做法就是使用宏。

术语"宏"在 Lisp 语言中，与其他语言中的意思不一样。Lisp 宏无所不包，它既可能是某样表达式的缩略形式，也可能是一种新语言的编译器。如果你想真正地理解 Lisp 语言，或者想拓宽你的编程视野，那么你必须学习宏。

就我所知，宏（采用 Lisp 语言的定义）目前仍然是 Lisp 独有的。一个原因是为了使用宏，你大概不得不让你

的语言看上去像 Lisp 一样古怪。另一个可能的原因是，如果你想为自己的语言添上这种终极武器，你从此就不能声称自己发明了新语言，只能说发明了一种 Lisp 的新方言。

我把这件事当作笑话说出来，但是事实就是如此。如果你创造了一种新语言，其中有 car、cdr、cons、quote、cond、atom、eq 这样的功能，还有一种把函数写成列表的表示方法，那么在它们的基础上，你完全可以推导出 Lisp 语言的所有其他部分。事实上，Lisp 语言就是这样定义的，John McCarthy 把语言设计成这个样子，就是为了让这种推导成为可能。

六、

就算 Lisp 确实代表了目前主流编程语言不断靠近的一个方向，这是否意味着你就应该用它编程呢？

如果使用一种不那么强大的语言，你又会有多少损失呢？有时不采用最尖端的技术，不也是一种明智的选择吗？这么多人使用主流编程语言，这本身不也说明那些语言有可取之处吗？

另一方面，选择哪一种编程语言，许多项目是无所谓的，反正不同的语言都能完成工作。一般来说，条件越苛刻的项目，强大的编程语言就越能发挥作用。但是，无数的项目根本没有苛刻条件的限制。大多数的编程任务，可能只要写一些很小的程序，然后用胶水语言把这些小程序连起来就行了。你可以用自己熟悉的编程语言，或者用对于特定项目来说有着最强大函数库的语言，来写这些小程序。如果你只是需要在 Windows 应用程序之间传递数据，使用 Visual Basic 照样能达到目的。

那么，Lisp 的编程优势体现在哪里呢？

七、

语言的编程能力越强大，写出来的程序就越短（当然不是指字符数量，而是指独立的语法单位）。

代码的数量很重要，因为开发一个程序耗费的时间，主要取决于程序的长度。如果同一个软件，一种语言写出来的代码比另一种语言长三倍，这意味着你开发它耗费的时间也会多三倍。而且即使你多雇佣人手，也无助于减少开发时间，因为当团队规模超过某个门槛时，再增加人手只会带来净损失。Fred Brooks 在他的名著《人月神话》（The Mythical Man-Month）中，描述了这种现象，我的所见所闻印证了他的说法。

如果使用 Lisp 语言，能让程序变得多短？以 Lisp 和 C 的比较为例，我听到的大多数说法是 C 代码的长度是 Lisp 的 7 倍到 10 倍。但是最近，New Architect 杂志上有一篇介绍 ITA 软件公司的文章，里面说"一行 Lisp 代码相当于 20 行 C 代码"，因为此文都是引用 ITA 总裁的话，所以我想这个数字来自 ITA 的编程实践。 如果真是这样，那么我们可以相信这句话。ITA 的软件，不仅使用 Lisp 语言，还同时大量使用 C 和 C++，所以这是他们的经验谈。

根据上面的这个数字，如果你与 ITA 竞争，而且你使用 C 语言开发软件，那么 ITA 的开发速度将比你快 20 倍。

如果你需要一年时间实现某个功能，它只需要不到三星期。反过来说，如果某个新功能，它开发了三个月，那么你需要五年才能做出来。

你知道吗？上面的对比，还只是考虑到最好的情况。当我们只比较代码数量的时候，言下之意就是假设使用功能较弱的语言，也能开发出同样的软件。但是事实上，程序员使用某种语言能做到的事情，是有极限的。如果你想用一种低层次的语言，解决一个很难的问题，那么你将会面临各种情况极其复杂、乃至想不清楚的窘境。

所以，当我说假定你与 ITA 竞争，你用五年时间做出的东西，ITA 在 Lisp 语言的帮助下只用三个月就完成了，我指的五年还是一切顺利、没有犯错误、也没有遇到太大麻烦的五年。事实上，按照大多数公司的实际情况，计划中五年完成的项目，很可能永远都不会完成。

我承认，上面的例子太极端。ITA 似乎有一批非常聪明的黑客，而 C 语言又是一种很低层次的语言。但是，在一个高度竞争的市场中，即使开发速度只相差两三倍，也足以使得你永远处在落后的位置。

附录：编程能力

为了解释我所说的语言编程能力不一样，请考虑下面的问题。我们需要写一个函数，它能够生成累加器，即这个函数接受一个参数 n，然后返回另一个函数，后者接受参数 i，然后返回 n 增加（increment）了 i 后的值。

Common Lisp 的写法如下：

```
(defun foo (n)
    (lambda (i) (incf n i)))
```

Ruby 的写法几乎完全相同：

```
def foo (n)
    lambda {|i| n +=
i } end
```

Perl 5 的写法则是：

```
sub foo {
    my ($n) = @_;
    sub {$n += shif
t}
}
```

这比 Lisp 和 Ruby 的版本，有更多的语法元素，因为在 Perl 语言中，你不得不手工提取参数。

Smalltalk 的写法稍微比 Lisp 和 Ruby 的长一点：

```
foo: n
    |s|
    s := n.
    ^[:i| s := s+i. ]
```

因为在 Smalltalk 中，局部变量（lexical variable）是有效的，但是你无法给一个参数赋值，因此不得不设置了一个新变量，接受累加后的值。

Javascript 的写法也比 Lisp 和 Ruby 稍微长一点，因为 Javascript 依然区分语句和表达式，所以你需要明确指定 return 语句，来返回一个值：

```
function foo (n) {
        return function (i)
{
        return n += i } }
```

（实事求是地说，Perl 也保留了语句和表达式的区别，但是使用了典型的 Perl 方式处理，使你可以省略 return。）

如果想把 Lisp/Ruby/Perl/Smalltalk/Javascript 的版本改成 Python，你会遇到一些限制。因为 Python 并不完全支持局部变量，你不得不创造一种数据结构，来接受 n 的值。而且尽管 Python 确实支持函数数据类型，但是没有一种字面量的表示方式（literal representation）可以生成函数（除非函数体只有一个表达式），所以你需要创造一个命名函数，把它返回。最后的写法如下：

```
def foo (n):
    s = [n]
    def bar (i):
        s[0] += i
        return s
[0]
    return bar
```

Python 用户完全可以合理地质疑，为什么不能写成下面这样：

```
def foo (n):
    return lambda i: return n += i
```

或者：

```
def foo (n):
    lambda i: n += i
```

我猜想，Python 有一天会支持这样的写法。（如果你不想等到 Python 慢慢进化到更像 Lisp，你总是可以直接……）

在面向对象编程的语言中，你能够在有限程度上模拟一个闭包（即一个函数，通过它可以引用由包含这个函数的代码所定义的变量）。你定义一个类（class），里面有一个方法和一个属性，用于替换封闭作用域（enclosing scope）中的所有变量。这有点类似于让程序员自己做代码分析，本来这应该是由支持局部作用域的编译器完成的。如果有多个函数，同时指向相同的变量，那么这种方法就会失效，但是在这个简单的例子中，它已经足够了。

Python 高手看来也同意，这是解决这个问题的比较好的方法，写法如下：

```
def foo (n):
    class acc:
        def _ _init_ _ (self, s):
            self.s = s
        def inc (self, i):
            self.s += i
            return self.s
    return acc (n).inc
```

或者

```
class foo:
    def _ _init_ _ (self, n):
        self.n = n
    def _ _call_ _ (self, i):
        self.n += i
        return self.n
```

我添加这一段，原因是想避免 Python 爱好者说我误解这种语言。但是，在我看来，这两种写法好像都比第一个版本更复杂。你实际上就是在做同样的事，只不过划出了一个独立的区域，保存累加器函数，区别只是保存在对象的一个属性中，而不是保存在列表（list）的头（head）中。使用这些特殊的内部属性名（尤其是__call__），看上去并不像常规的解法，更像是一种破解。

在 Perl 和 Python 的较量中，Python 黑客的观点似乎是认为 Python 比 Perl 更优雅，但是这个例子表明，最终来说，编程能力决定了优雅。Perl 的写法更简单（包含更少的语法元素），尽管它的语法有一点丑陋。

其他语言怎么样？前文曾经提到过 Fortran、C、C++、Java 和 Visual Basic，看上去使用它们，根本无法解决这个问题。Ken Anderson 说，Java 只能写出一个近似的解法：

```
public interface Inttoint {
    public int call (int i);
}

public static Inttoint foo (final int n) {
    return new Inttoint () {
    int s = n;
    public int call (int i) {
    s = s + i;
    return s;
    }};
}
```

这种写法不符合题目要求，因为它只对整数有效。

当然，我说使用其他语言无法解决这个问题，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。

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。"

如果你想解决一个困难的问题，关键不是你使用的语言是否强大，而是好几个因素同时发挥作用（a）使用一种强大的语言，（b）为这个难题写一个事实上的解释器，或者（c）你自己变成这个难题的人肉编译器。在 Python 的例子中，这样的处理方法已经开始出现了，我们实际上就是自己写代码，模拟出编译器实现局部变量的功能。这种实践不仅很普遍，而且已经制度化了。举例来说，在面向对象编程的世界中，我们大量听到"模式"（pattern）这个词，我觉得那些"模式"就是现实中的因素（c），也就是人肉编译器。 当我在自己的程序中，发现用到了模式，我觉得这就表明某个地方出错了。程序的形式，应该仅仅反映它所要解决的问题。代码中其他任何外加的形式，都是一个信号，（至少对我来说）表明我对问题的抽象还不够深，也经常提醒我，自己正在手工完成的事情，本应该写代码，通过宏的扩展自动实现。

来源：http://www.cnblogs.com/syeerzy/articles/3548899.html

# 编者记

我一直都很喜欢 Lisp 这样的语言。

很多人会问 XX 语言流行吗？XX 语言能赚钱吗？XX 语言前景怎么样？其实，我们需要问的是：

*这种语言好用吗？*

*这种语言强大吗？*

*这种语言的思维方式是什么？*

当你能清楚的回答这样的问题时，这就是一种合适的语言了。Lisp 就是这样的一门语言，具体的观点各位读者可以从前面的这篇《为什么 **Lisp** 语言如此先进？》窥见一二，但 Lisp 的真正魅力无疑还须各位亲自领略。我学习 Lisp 以来发现在国内学习 Lisp 最大的难处就是资料少，目前比较好买的书就是《实用 Common Lisp 编程》，其它的大多老旧或者是某一特定领域的 Lisp。即使是国内规模比较大的 Lisp 中文社区上，想要找到一份详尽且适合初学者的 Lisp 也并不是那么简单的。我很早就萌发了自己撰写一部关于 Lisp 编程的书籍的念头，正好在易百网（http://yiibai.com/）上发现了这一系列 Lisp 教程，这可真是雪中送炭。我将它们搜集起来一起编辑成这份文档，希望能各位热爱 Lisp 的朋友提供一点帮助。

在深入学习这份文档前，容我指出这份文档的不足：

1.没有给出 Lisp 环境搭建的指导，这方面的内容读者可以参见《实用 Common Lisp 编程》或者自行搜索 SBCL，GCL 等 CL 实现，在这份文档的下一版本中我会把这个坑给填上的，第一版时间紧促就不管实现了:-)；

2.对于一些深入的主题没有初级，毕竟这只是一份“简易”的教程，想要深入学习的强烈推荐 ANSI 的那本 Common Lisp 手册，不过只有英文版的，《计算机程序的构造与解释》，这本书我没看过，单据说是经典，还有一本《Common Lisp 符号计算引论》太复杂了，喜欢的可以自己搜；

3.没有比较系统的案例，这点我觉得《实用 Common Lisp 编程》已经写得很好了，下一版本时我也会补充上的；

4.糟糕的排版，这个全怪我，我也没学过什么 LaTex、Word 排版，以前论文排版也是乱七八糟的，还是别人帮我排的，如果你觉得不爽也请通过邮箱（chaobs@outlook.com）联系我，帮我一起排版！

Chaobs

CUCS

2015 年 10 月

# 鸣谢

一切荣耀属于网友

长工 http://www.yiibai.com/lisp/lisp_overview.html

YeaWind http://www.yiibai.com/lisp/lisp_program_structure.html

ache038 http://www.yiibai.com/lisp/lisp_basic_syntax.html

逝风 123 http://www.yiibai.com/lisp/lisp_data_types.html

曦花 http://www.yiibai.com/lisp/lisp_macros.html

yak http://www.yiibai.com/lisp/lisp_variables.html

黑狗 http://www.yiibai.com/lisp/lisp_constants.html

WiJQ http://www.yiibai.com/lisp/lisp_operators.html

快乐学习 http://www.yiibai.com/lisp/lisp_decisions.html

php 小浩 http://www.yiibai.com/lisp/lisp_loops.html

stone-sun http://www.yiibai.com/lisp/lisp_functions.html

sallay http://www.yiibai.com/lisp/lisp_predicates.html

梦醒以后 http://www.yiibai.com/lisp/lisp_numbers.html

刘鑫华 http://www.yiibai.com/lisp/lisp_characters.html

绿水无痕 http://www.yiibai.com/lisp/lisp_arrays.html

kevinG http://www.yiibai.com/lisp/lisp_symbols.html

iTony http://www.yiibai.com/lisp/lisp_vectors.html

hibernate_jss http://www.yiibai.com/lisp/lisp_set.html

如是传统 http://www.yiibai.com/lisp/lisp_tree.html

郑小千 http://www.yiibai.com/lisp/lisp_hash_table.html

花田软件 http://www.yiibai.com/lisp/lisp_input_output.html

Anger_Coder http://www.yiibai.com/lisp/lisp_file_io.html

HerbertYang http://www.yiibai.com/lisp/lisp_structures.html

vigiles http://www.yiibai.com/lisp/lisp_packages.html

枫爱若雪 http://www.yiibai.com/lisp/lisp_error_handling.html

百 mumu http://www.yiibai.com/lisp/lisp_clos.html

再次对这些网友的无私贡献表示最诚挚的感谢！

# 错误反馈

没有一本书没有 BUG，这篇文档肯定存在很多知识上的漏洞、错别字、排版上的不合适，由于水平有限，欢迎指正。如果你发现任何问题或者对内容有补充，请不吝赐教！让我们一起把这本教程做大！

联系邮箱：chaobs@outlook.com ， q578836573@163.com

博客：www.cnblogs.com/Chaobs

Chaobs

CUCS

2015 年 10 月

# 目录

# 1 LISP - 概述介绍

Lisp 是 Fortran 语言之后第二古老的高级编程语言，自成立之初已发生了很大变化，和一些方言一直存在在它的历史。今天，最广为人知的通用的 Lisp 方言 Common Lisp 和 Scheme。Lisp 由约翰·麦卡锡在 1958 年发明，在麻省理工学院（MIT）。

该参考将带您通过简单实用的方法，同时学习 Lisp 程序设计语言。

Lisp 是一门历史悠久的语言，全名叫 LISt Processor，也就是“表处理语言”，它是由 John McCarthy 于 1958 年就开始设计的一门语言。和 Lisp 同时期甚至更晚出现的许多语言如 Algo 等如今大多已经消亡，又或者仅仅在一些特定的场合有一些微不足道的用途，到现在还广为人知的恐怕只剩下了 Fortran 和 COBOL。但唯独 Lisp，不但没有随着时间而衰退，反倒是一次又一次的焕发出了青春，从 Lisp 分支出来的 Scheme、ML 等语言 在很多场合的火爆程度甚至超过了许多老牌明星。那么这颗常青树 永葆青春的奥秘究竟在哪里呢？

如果你只接触过 C/C++、Pascal 这些“过程式语言”的话，Lisp 可能会让你觉得十分不同寻常，首先吸引你眼球（或者说让你觉得混乱的）一定是 Lisp 程序中异常多的括号，当然从现在的角度来讲，这种设计的确对程序员不大友好，不过考虑到五六十年代的计算机处理能力，简化语言本身的设计在那时算得上是当务之急了。

## 1.1 读者

该参考是不完全是为初学者准备的，只是帮助他们了解基本的到相关 LISP 编程语言的先进理念。但前提条件是假设你已经知道什么是计算机程序，什么是计算机编程语言，至少已有用一种高级语言编程的经历，且至少写过三个程序。

## 1.2 LISP - 历史介绍

约翰·麦卡锡发明 LISP 于 1958 年，FORTRAN 语言的发展后不久。首次由史蒂夫·拉塞尔实施在 IBM704 计算机上。它特别适合用于人工智能方案，因为它有效地处理的符号信息。Common Lisp 的起源，20 世纪 80 年代和 90 年代，分别接班人 Maclisp 像 ZetaLisp 和 NIL(Lisp 语言的新实施)等开发。

它作为一种通用语言，它可以很容易地扩展为具体实施。编写 Common Lisp 程序不依赖于机器的具体特点，如字长等。

## 1.3 Common Lisp 的特点

- 这是机器无关
- 它采用迭代设计方法，且易于扩展。
- 它允许动态更新的程序。
- 它提供了高层次的调试。
- 它提供了先进的面向对象编程。

- 它提供了方便的宏系统。
- 它提供了对象，结构，列表，向量，可调数组，哈希表和符号广泛的数据类型。
- 它是以表达为主。
- 它提供了一个面向对象的系统条件。
- 它提供完整的 I/ O 库。
- 它提供了广泛的控制结构。

## 1.4 LISP 的内置应用程序

大量成功的应用建立在 Lisp 语言。

- Emacs
- G2
- AutoCad
- Igor Engraver
- Yahoo Store

# 2 LISP - 程序结构

LISP 表达式称为符号表达式或 S-表达式。s 表达式是由三个有效对象，原子，列表和字符串。任意的 s-表达式是一个有效的程序。Lisp 程序在解释器或编译的代码运行。解释器会检查重复的循环，这也被称为读 - 计算 - 打印循环(REPL)源代码。它读取程序代码，计算，并打印由程序返回值。

## 2.1 一个简单的程序

让我们写一个 s-表达式找到的三个数字 7,9 和 11 的总和。要做到这一点，我们就可以输入在提示符的解释器 ->:

```
(+7911)
```

LISP 返回结果：

```
27
```

如果想运行同一程序的编译代码，那么创建一个名为 myprog 的一个 LISP 源代码文件。

并在其中输入如下代码：

```
(write(+7911))
```

单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
27
```

## 2.2 Lisp 使用前缀表示法

可能已经注意到，使用 LISP 前缀符号。在上面的程序中的+符号可以作为对数的求和过程中的函数名。在前缀表示法，运算符在自己操作数前写。例如，表达式，

```
a * ( b + c ) / d
```

将被写为：

```
(/ (* a (+ b c) ) d)
```

让我们再举一个例子，让我们写的代码转换为 60o F 华氏温度到摄氏刻度：

*此转换的数学表达式为：*

```
(60 * 9 / 5) + 32
```

创建一个名为 main.lisp 一个源代码文件，并在其中输入如下代码：

```
(write(+ (* (/ 9 5) 60) 32))
```

当单击 Execute 按钮，或按下 Ctrl+ E，MATLAB 立即执行它，返回的结果是：

```
140
```

## 2.3 计算 Lisp 程序

计算 LISP 程序有两部分：

- 程序文本由一个读取器程序转换成 Lisp 对象
- 语言的语义在这些对象中的条款执行求值程序

计算过程采用下面的步骤：

- 读取器转换字符到 LISP 对象或 S-表达式的字符串。
- 求值器定义为那些从 s-表达式内置的 Lisp 语法形式。

计算第二个级别定义的语法决定了 S-表达式是 LISP 语言形式。求值器可以作为一个函数，它接受一个有效的 LISP 语言的形式作为参数并返回一个值。这就是为什么我们把括号中的 LISP 语言表达，因为我们要发送的整个表达式/形式向求值作为参数的原因。

## 2.4 'Hello World' 程序

学习一门新的编程语言并没有真正起飞，直到学会如何迎接语言的整个世界，对吧！所以，创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
(write-line "Hello World") (write-line "I am at 'Tutorials Yiibai'! Learning LISP")
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
Hello World I am at 'Tutorials Yiibai'! Learning LISP
```

# 3 LISP - 基本语法

## 3.1 LISP 基本构建块

Lisp 程序是由三个基本构建块：

- atom
- list
- string

一个原子是一个数字连续字符或字符串。它包括数字和特殊字符。以下是一些有效的原子的例子：

```
hello-from-tutorials-yiibai
name

123008907 *hello* Block#221 abc123
```

列表是包含在括号中的原子和/或其他列表的序列。以下是一些有效的列表的示例：

```
( i am a list) (a ( a b c) d e fgh) (father tom ( susan bill joe)) (sun mon tue wed thur fri sat) ( )
```

字符串是一组括在双引号字符。以下是一些有效的字符串的例子：

```
" I am a string" "a ba c d efg #$%^&!" "Please enter the following details :" "Hello from 'Tutorials Yiibai'! "
```

## 3.2 添加注释

分号符号(;)是用于表示一个注释行。

例如，

```
(write-line "Hello World") ; greet the world
; tell them your whereabouts

(write-line "I am at 'Tutorials Yiibai'! Learning LISP")
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
Hello World I am at 'Tutorials Yiibai'! Learning LISP
```

## 3.3 移动到下一节之前的一些值得注意的要点

以下是一些要点需要注意：

- 在 LISP 语言的基本数学运算是 +，-，*，和 /
- Lisp 实际上是一个函数调用 f(x) 为 (f x)，例如 cos(45) 被写入为 cos 45
- LISP 表达式是不区分大小写的，cos 45 或 COS 45 是相同的。
- LISP 尝试计算一切，包括函数的参数。只有三种类型的元素是常数，总是返回自己的值：
- 数字
- 字母 t，即表示逻辑真
- 该值为 nil，这表示逻辑 false，还有一个空的列表。

## 3.4 稍微介绍一下 LISP 形式

在前面的章节中，我们提到 LISP 代码计算过程中采取以下步骤：读取器转换字符到 LISP 对象的字符串或 s-expressions.求值器定义为那些从 s-表达式内置的 Lisp 语法形式。计算第二个级别定义的语法决定了 S-表达式是 LISP 语言形式。

现在，一个 LISP 的形式可以是：

- 一个原子
- 空或非名单
- 有符号作为它的第一个元素的任何列表

求值器可以作为一个函数，它接受一个有效的 LISP 语言的形式作为参数，并返回一个值。这个就是为什么我们把括号中的 LISP 语言表达，因为我们要发送的整个表达式/形式向求值作为参数的原因。

## 3.5 LISP 命名约定

名称或符号可以包含任意数量的空白相比，开放和右括号，双引号和单引号，反斜杠，逗号，冒号，分号和竖线其他字母数字字符。若要在名称中使用这些字符，需要使用转义字符（ ）。一个名字可以包含数字，但不能全部由数字组成，因为那样的话它会被解读为一个数字。同样的名称可以具有周期，但周期不能完全进行。

## 3.6 使用单引号

LISP 计算一切，包括函数的参数和列表的成员。有时，我们需要采取原子或列表字面上，不希望他们求值或当作函数调用。要做到这一点，我们需要先原子或列表中带有单引号。

下面的例子演示了这一点：

创建一个名为 main.lisp 文件，并键入下面的代码进去：

```
write-line "single quote used, it inhibits evaluation") (write '(* 2 3))
(write-line " ")
(write-line "single quote not used, so expression evaluated")

(write (* 2 3))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
single quote used, it inhibits evaluation
(* 2 3)
single quote not used, so expression evaluated

6
```

# 4 LISP - 数据类型

在 LISP 中，变量没有类型的，但有数据对象。LISP 数据类型可分类为：

- 标量类型 – 例如，数字类型，字符，符号等。

- 数据结构 - 例如，列表，向量，比特向量和字符串。

任何变量都可以采取任何的 Lisp 对象作为它的值，除非明确地声明它。虽然，这是没有必要指定一个 Lisp 变量的数据类型，但是，它有助于在一定的循环扩展，在方法声明和其他一些情况下，我们将在后面的章节中讨论。 该数据类型被布置成层次结构。数据类型是一组 LISP 对象和多个对象可能属于这样的一套。

- typep 谓词用于发现一个对象是否属于一个特定的类型。

- type-of 函数，返回给定对象的数据类型的类型。

## 4.1 在 LISP 类型说明符

类型说明符是数据类型的系统定义的符号。

| | | | |
|---|---|---|---|
| array | fixnum | package | simple-string |
| atom | float | pathname | simple-vector |
| bignum | function | random-state | single-float |
| bit | hash-table | ratio | standard-char |
| bit-vector | integer | rational | stream |
| character | keyword | readtable | string |
| [common] | list | sequence | [string-char] |
| compiled-function | long-float | short-float | symbol |
| complex | nill | signed-byte | t |
| cons | null | simple-array | unsigned-byte |
| double-float | number | simple-bit-vector | vector |

除了这些系统定义的类型，可以创建自己的数据类型。当一个结构类型是使用 defstruct 函数定义，结构类型的名称将成为一个有效的类型符号。

**示例 1**

创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
(setq x 10) (setq y 34.567) (setq ch nil) (setq n 123.78) (setq bg 11.0e+4) (setq r 124/2) (print x)
```

```
(print y) (print n) (print ch) (print bg) (print r)
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
10
34.567
123.78
NIL
110000.0

62
```

**实例 2**

接下来让我们看看前面的例子中使用的变量的类型。创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
(setq x 10) (setq y 34.567) (setq ch nil) (setq n 123.78) (setq bg 11.0e+4) (setq r 124/2) (print
(type-of x)) (print (type-of y)) (print (type-of n)) (print (type-of ch)) (print (type-of bg)) (print (type-
of r))
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
(INTEGER 0 281474976710655)
SINGLE-FLOAT
SINGLE-FLOAT
NULL
SINGLE-FLOAT

(INTEGER 0 281474976710655)
```

# 5 LISP - 宏

宏可以扩展标准 LISP 的语法。从技术上讲，宏是一个函数，它接受一个 s-expression 作为参数，并返回一个 LISP 的形式，然后进行评估计算。

## 5.1 定义一个宏

在 LISP 中，一个名为宏使用另一个名为 defmacro 宏定义。定义一个宏的语法：

```
(defmacro macro-name (parameter-list)  "Optional documentation string."  body-form)
```

宏定义包含宏的名称，参数列表，可选的文档字符串，和 Lisp 表达式的体，它定义要由宏执行的任务。

**实例**

让我们写了一个名为 setTo10 简单的宏，将采取一系列并将其值设置为 10。创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
defmacro setTo10(num)  (setq num 10)(print num))  (setq x 25)  (print x)  (setTo10 x)
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
25

10
```

# 6 LISP - 变量

在 LISP 中，每个变量由一个' 符号' 表示。变量的名称是符号的名字，并将其存储在码元的存储单元。

## 6.1 全局变量

全局变量有永久值在整个 LISP 系统，并保持有效，直到指定的新值。全局变量是使用 defvar 结构一般声明。

例如：

```
(defvar x 234) (write x)
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
234
```

由于没有类型声明在 LISP 变量，可直接用 setq 一样构建一个符号指定一个值

例如，

```
->(setq x 10)
```

上面的表达式的值 10 赋给变量 x，也可以使用符号本身作为一个表达式来引用该变量。

符号值函数允许提取存储在符号存储位置的值。

**示例**

创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
(setq x 10) (setq y 20) (format t "x = ~2d y = ~2d ~%" x y) (setq x 100) (setq y 200) (format t "x = ~2d y = ~2d" x y)
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
x = 10 y = 20
x = 100 y = 200
```

## 6.2 局部变量

局部变量在给定的过程中定义。被命名为一个函数定义中参数的参数也是局部变量。局部变量只能访问内相应的功能。像的全局变量，也可以使用本 setq 一样构建体被创建的局部变量。还有其他两种结构- let 和 prog 创建局部变量。

该 let 结构的语法如下：

```
(let ((var1  val11) (var2  val12).. (varn  valn))<s-expressions>)
```

其中 var1，var2，..varn 是变量名和 val1，val12，.. valn 是分配给相应的变量的初始值。

当执行 let，每个变量被分配了各自的值，最后的 s-expression。则返回最后一个表达式的值。

如果不包括的变量的初始值，它被分配到 nil。

**例子**

创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
(let ((x 'a)
      (y 'b)          (z 'c))

(format t "x = ~a y = ~a z = ~a" x y z))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
x = A y = B z = C
```

该编结构也有局部变量作为第一个参数，它后面是 prog 的主体，以及任意数量 s-expressions 的列表。

该编函数执行 s-expressions 序列的列表，并返回零，除非遇到函数调用名返回。然后函数参数计算并返回。

**例子**

创建一个名为 main.lisp 新的源代码文件，并在其中输入如下代码：

```
(prog ((x '(a b c))
      (y '(1 2 3))          (z '(p q 10)))

(format t "x = ~a y = ~a z = ~a" x y z))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
x = (A B C) y = (1 2 3) z = (P Q 10)
```

# 7 LISP - 常量

在 LISP 中，常量变量在程序执行期间，从来没有改变它们的值。常量使用 defconstant

结构声明。

### 例子

下面的例子显示了声明一个全局常量 PI 和以后使用的函数命名 area-circle 计算圆的面积的值。该函数 defun 结构用于定义一个函数，我们将看看它在“函数”一章。创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defconstant PI 3.141592) (defun area-circle(rad) (terpri) (format t "Radius: ~5f" rad) (format t "~%Area: ~10f" (* PI rad rad))) (area-circle 10)
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
Radius:  10.0  Area:  314.1592
```

# 8 LISP - 运算符

运算符是一个符号，它告诉编译器执行特定的数学或逻辑操作。 LISP 允许在众多的数据业务，通过各种函数，宏和其他结构的支持。允许对数据的操作都可以归类为：

- 算术运算
- 比较操作
- 逻辑运算
- 位运算

## 8.1 算术运算

下表列出了所有支持的 LISP 算术运算符。假设变量 A=10 和变量 B=20 则：

| 运算符 | 描述 | Example |
| --- | --- | --- |
| + | 增加了两个操作数 | (+ A B) = 30 |

| | | |
|---|---|---|
| - | 从第一数减去第二个操作数 | (- A B)= -10 |
| * | 乘两个操作数 | (* A B) = 200 |
| / | 通过取消分子除以分子 | (/ B A) = 2 |
| mod,rem | 模运算符和其余整数除法后 | (mod B A ) = 0 |
| incf | 递增运算符，所指定的第二个参数增加整数值 | (incf A 3) = 13 |
| decf | 递减操作符，通过指定的第二个参数减小整数值 | (decf A 4) = 9 |

**例子**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (setq b 20) (format t "~% A + B = ~d" (+ a b)) (format t "~% A - B = ~d" (- a b))
(format t "~% A x B = ~d" (* a b)) (format t "~% B / A = ~d" (/ b a)) (format t "~% Increment A by
3 = ~d" (incf a 3)) (format t "~% Decrement A by 4 = ~d" (decf a 4))
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
A + B = 30 A - B = -10 A x B = 200 B / A = 2 Increment A by 3 = 13 Decrement A by 4 = 9
```

## 8.2 比较操作

下表列出了所有支持的 LISP 关系运算符的数字之间进行比较。然而不像其他语言的关系运算符，LISP 的比较操作符可能需要超过两个操作数，他们在只有数字工作。

假设变量 A=10 和变量 B=20，则：

| Operator | 描述 | Example |
|---|---|---|
| = | 检查如果操作数的值都相等与否，如果是的话那么条件为真。 | (= A B)= true. |
| /= | 检查如果操作数的值都不同，或没有，如果值不相等，则条件为真。 | (/= A B) =true. |
| > | 检查如果操作数的值单调递减。 | (> A B) !=true. |
| < | 检查如果操作数的值单调递增。 | (< A B) = true. |
| >= | 如有左操作数的值大于或等于下一个右操作数的值，如果是则条件检查为真。 | (>= A B) !=true. |
| <= | 如有左操作数的值小于或等于其右操作数的值，如果是，则条件检查为真。 | (<= A B) = true. |

| | | |
|---|---|---|
| max | 它比较两个或多个参数，并返回最大值。 | (max A B) 返回 20 |
| min | 它比较两个或多个参数，并返回最小值。 | (min A B) 返回 20 |

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (setq b 20) (format t "~% A = B is ~a" (= a b)) (format t "~% A /= B is ~a" (/= a b))
(format t "~% A > B is ~a" (> a b)) (format t "~% A < B is ~a" (< a b)) (format t "~% A >= B is ~a"
(>= a b)) (format t "~% A <= B is ~a" (<= a b)) (format t "~% Max of A and B is ~d" (max a b))
(format t "~% Min of A and B is ~d" (min a b))
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
A = B is NIL
A /= B is T
A > B is NIL
A < B is T
A >= B is NIL
A <= B is T

Max of A and B is 20 Min of A and B is 10
```

## 8.3 布尔值逻辑操作

Common Lisp 中提供了三种逻辑运算符：AND，OR，而不是运算符的布尔值。假定 A=nil，B=5，那么

| 运算符 | 描述 | 示例 |
|---|---|---|
| and | 这需要任意数量的参数。该参数是从左向右计算。如果所有参数的计算结果为非零，那么最后一个参数的值返回。否则就返回 nil。 | (and A B) = NIL. |
| or | 这需要任意数量的参数。该参数是从左向右计算的，直到一个计算结果为非零，则此情况下返回参数值，否则返回 nil。 | (or A B) = 5. |
| not | 它接受一个参数，并返回 t，如果参数的计算结果为 nil。 | (not A) = T. |

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (setq b 20) (format t "~% A and B is ~a" (and a b)) (format t "~% A or B is ~a" (or a b)) (format t "~% not A is ~a" (not a)) (terpri) (setq a nil) (setq b 5) (format t "~% A and B is ~a" (and a b)) (format t "~% A or B is ~a" (or a b)) (format t "~% not A is ~a" (not a)) (terpri) (setq a nil) (setq b 0) (format t "~% A and B is ~a" (and a b)) (format t "~% A or B is ~a" (or a b)) (format t "~% not A is ~a" (not a)) (terpri) (setq a 10) (setq b 0) (setq c 30) (setq d 40) (format t "~% Result of and operation on 10, 0, 30, 40 is ~a" (and a b c d)) (format t "~% Result of and operation on 10, 0, 30, 40 is ~a" (or a b c d)) (terpri) (setq a 10) (setq b 20) (setq c nil) (setq d 40) (format t "~% Result of and operation on 10, 20, nil, 40 is ~a" (and a b c d)) (format t "~% Result of and operation on 10, 20, nil, 40 is ~a" (or a b c d))
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
A and B is 20 A or B is 10 not A is NIL

A and B is NIL
A or B is 5 not A is T

A and B is NIL
A or B is 0 not A is T

Result of and operation on 10, 0, 30, 40 is 40 Result of and operation on 10, 0, 30, 40 is 10
Result of and operation on 10, 20, nil, 40 is NIL

Result of and operation on 10, 20, nil, 40 is 10
```

请注意，逻辑运算工作，布尔值，其次，数字为零，NIL 不是一样的。

## 8.4 对数位运算

位运算符位工作并进行逐位操作。对于按位与，或，和 XOR 运算的真值表如下：

| p | q | p and q | p or q | p xor q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

```
Assume if A = 60; and B = 13; now in binary format they will be as follows: A = 0011 1100 B = 0000 1101 ----------------- A and B = 0000 1100 A or B = 0011 1101 A xor B = 0011 0001 not A  = 1100 0011
```

通过 LISP 支持位运算符列于下表中。假设变量 A=60 和变量 B=13，则：

| 操作符 | 描述 | Example |
| --- | --- | --- |
| logand | 这将返回位逻辑的参数和。如果没有给出参数，则结果为-1，这是该操作的标识。 | (logand a b)) = 12 |
| logior | 这将返回位逻辑包括它的参数或。如果没有给出参数，那么结果是零，这是该操作的标识。 | (logior a b) = 61 |
| logxor | 这将返回其参数的按位逻辑异或。如果没有给出参数，那么结果是零，这是该操作的标识。 | (logxor a b) = 49 |
| lognor | 这不返回的逐位它的参数。如果没有给出参数，则结果为-1，这是该操作的标识。 | (lognor a b) = -62, |
| logeqv | 这将返回其参数的逐位逻辑相等（也称为异或非）。如果没有给出参数，则结果为-1，这是该操作的标识。 | (logeqv a b) = -50 |

**示例**

创建一个名为 `main.lisp` 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 60) (setq b 13) (format t "~% BITWISE AND of a and b is ~a" (logand a b)) (format t "~% BITWISE INCLUSIVE OR of a and b is ~a" (logior a b)) (format t "~% BITWISE EXCLUSIVE OR of a and b is ~a" (logxor a b)) (format t "~% A NOT B is ~a" (lognor a b)) (format t "~% A EQUIVALANCE B is ~a" (logeqv a b)) (terpri) (terpri) (setq a 10) (setq b 0) (setq c 30) (setq d 40) (format t "~% Result of bitwise and operation on 10, 0, 30, 40 is ~a" (logand a b c d)) (format t "~% Result of bitwise or operation on 10, 0, 30, 40 is ~a" (logior a b c d)) (format t "~% Result of bitwise xor operation on 10, 0, 30, 40 is ~a" (logxor a b c d)) (format t "~% Result of bitwise eqivalance operation on 10, 0, 30, 40 is ~a" (logeqv a b c d))
```

当您单击 `Execute` 按钮，或按下 `Ctrl+ E`，LISP 立即执行它，返回的结果是：

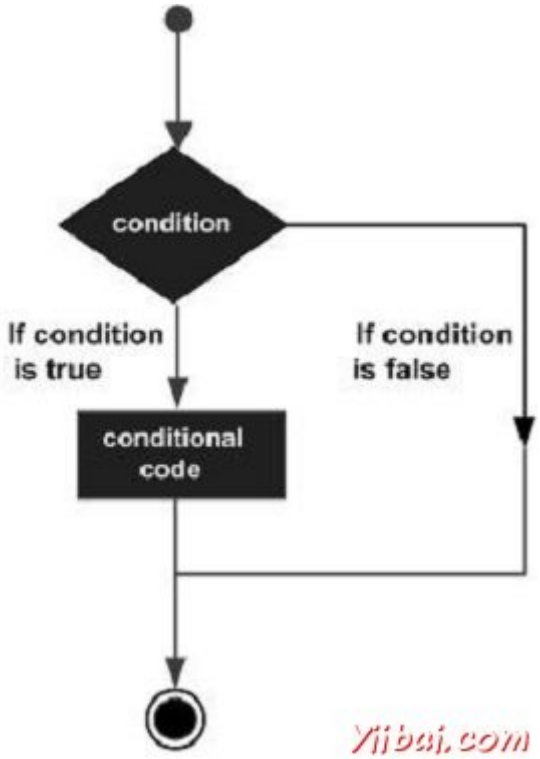
```
BITWISE AND of a and b is 12 BITWISE INCLUSIVE OR of a and b is 61 BITWISE EXCLUSIVE OR of a and b is 49 A NOT B is -62 A EQUIVALANCE B is -50

Result of bitwise and operation on 10, 0, 30, 40 is 0 Result of bitwise or operation on 10, 0, 30, 40 is 62 Result of bitwise xor operation on 10, 0, 30, 40 is 60 Result of bitwise eqivalance operation on 10, 0, 30, 40 is -61
```

# 9 LISP - 决策

决策结构需要程序员指定一个或多个条件由程序进行评估或测试，以及要执行的语句或语句如果条件被确定为 `true`，如果条件被确定为 `false` 那么选择要执行其他语句。

下面是在大多数编程语言中一个典型的决策结构的一般形式为：

LISP 提供了以下类型的决策构造。




| Construct | 描述 |
| --- | --- |
| cond | 这个结构是用于用于检查多个测试行动作条件。它可以嵌套 if 或其他编程语言语句。 |
| if | if 结构有多种形式。在最简单的形式，它后面跟着一个测试条，测试操作和一些其它相应措施(次)。如果测试子句的值为 true，那么测试的动作被执行，否则，由此产生的子句求值。 |
| when | 在最简单的形式，它后面跟着一个测试条和测试操作。如果测试子句的值为 true，那么测试的动作被执行，否则，由此产生的子句求值。 |
| case | 这种结构实现了像 cond 构造多个测试行动语句。但是，它会评估的关键形式，并允许根据该键的形式评价多个行动语句。 |

## 9.1 LISP 的 cond 特殊构造

在 LISP 语言中 cond 结构是最常用的，以允许分支。

*cond 的语法是*：

```
(cond   (test1    action1)        (test2    action2)        ...        (testn    actionn))
```

在 cond 语句中每个子句包含一个条件测试，并要执行的动作。

如果第一次测试下面的芯线，为 test1，被评估为 true，那么相关的行动的一部分，action1 执行，返回它的值，及本子句的其余部分被跳过。如果 test1 的计算结果是 nil，然后控制移动到第二个子句，而不执行 action1，和相同的流程进行后续处理。如果没有试验条件计算结果为真，那么 cond 语句返回 nil。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (cond ((> a 20)      (format t "~% a is less than 20")) (t (format t "~% value of a is ~d " a)))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
value of a is 10
```

请注意，第二个子句中 t 保证的是，如果没有其他的将最后完成的动作。

## 9.2 if 结构

如果该宏后跟一个测试子句计算为 t 或 nil。如果测试子句计算到 t，然后按照测试子句的动作被执行。如果它是零，那么下一个子句进行评估计算。

*if 的语法*：

```
(if (test-clause) (<action1) (action2))
```

**示例 1**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (if (> a 20)      (format t "~% a is less than 20")) (format t "~% value of a is ~d " a)
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
value of a is 10
```

**示例 2**

if 子句后面可以跟一个可选的 then 子句：

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (if (> a 20)  then (format t "~% a is less than 20")) (format t "~% value of a is ~d " a)
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
a is less than 20 value of a is 10
```

**示例 3**

还可以创建使用 if 子句的 if-then-else 类型声明。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 100) (if (> a 20)  (format t "~% a is greater than 20")
   (format t "~% a is less than 20")) (format t "~% value of a is ~d " a)
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
a is greater than 20 value of a is 100
```

## 9.3 when 构造

该 when 宏，后面跟着一个测试子句计算为 t 或为零。如果测试条被评估计算为 nil，则任何形式的评估及 nil 返回，但是它的测试结果为 t，则下面的测试条的动作被执行。

*when 宏的语法*：

```
(when (test-clause) (<action1) )
```

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 100) (when (> a 20)  (format t "~% a is greater than 20")) (format t "~% value of a is ~d
" a)
```

当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
a is greater than 20 value of a is 100
```

## 9.4 case 构造

case 结构实现像 cond 结构多个测试动作语句。但是，它会评估的键形式，并允许根据该键的形式评价多个动作语句。

该 case 宏的语法是：

```
The template for CASE is:
```

```
(case  (keyform) ((key1)  (action1  action2 ...) ) ((key2)  (action1  action2 ...) ) ... ((keyn)  (action1
action2 ...) ))
```

```
(setq day 4) (case day

(1 (format t "~% Monday")) (2 (format t "~% Tuesday")) (3 (format t "~% Wednesday")) (4
(format t "~% Thursday")) (5 (format t "~% Friday")) (6 (format t "~% Saturday")) (7 (format t "~
% Sunday")))
```

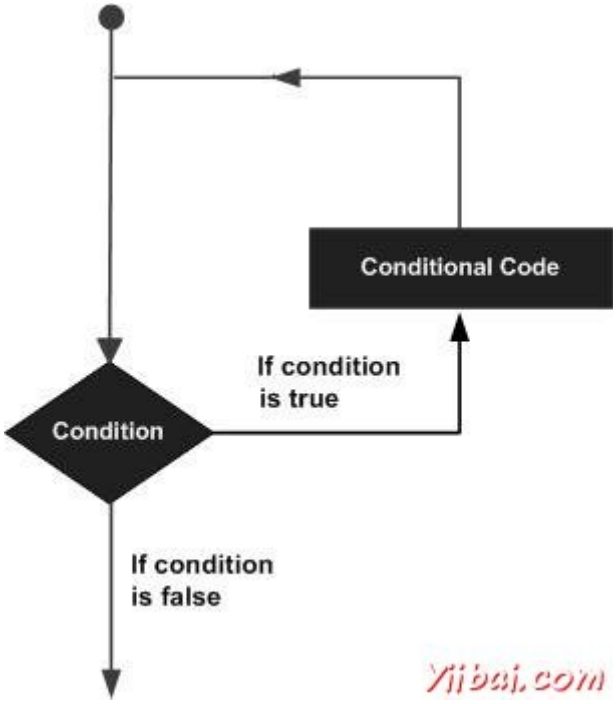
当您单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
Thursday
```

# 10 LISP - 循环

可能有一种情况，当需要执行代码块多次。循环语句可以让我们执行一个语句或语句组多次，下面是在大多数编程语言中的循环语句的一般形式为：

LISP 提供的结构来处理循环要求以下类型。




| Construct | 描述 |
| --- | --- |
| loop | 循环 loop 结构是迭代通过 LISP 提供的最简单的形式。在其最简单的形式，它可以重复执行某些语句(次)，直到找到一个 return 语句。 |
| loop for | loop 结构可以实现一个 for 循环迭代一样作为最常见于其他语言。 |
| do | do 结构也可用于使用 LISP 进行迭代。它提供了迭代的一种结构形式。 |
| dotimes | dotimes 构造允许循环一段固定的迭代次数。 |
| dolist | dolist 来构造允许迭代通过列表的每个元素。 |

## 10.1 循环 loop 结构

循环 loop 结构是迭代通过 LISP 提供的最简单的形式。在其最简单的形式，它可以重复执行某些语句(次)，直到找到一个 return 语句。它的语法如下：

```
(loop (s-expressions))
```

**例子**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a 10) (loop
   (setq a (+ a 1))    (write a)    (terpri)    (when (> a 17) (return a)))
```

当执行的代码，它返回以下结果：

```
11 12 13 14 15 16 17 18
```

请注意，没有 return 语句，循环宏会产生一个无限循环。

## 10.2 循环的构造

loop 结构可以实现一个 for 循环迭代一样作为最常见于其他语言。它可以

- 设置为迭代变量
- 指定表达式（s）表示，将有条件终止迭代
- 对于执行某些任务在每次迭代中指定表达式的结果
- 做一些任务而退出循环之前指定表达式（s）和表达式

*在 for 循环的结构如下几种语法：*

```
(loop for loop-variable in <a list>     do (action))
(loop for loop-variable from value1 to value2
   do (action))
```

**示例 1**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(loop for x in '(tom dick harry)
   do (format t " ~s" x)
```

```
)
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
TOM DICK HARRY
```

**示例 2**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(loop for a from 10 to 20       do (print a) )
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
10
11
12
13
14
15
16
17
18
19

20
```

**示例 3**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(loop for x from 1 to 20    if(evenp x)    do (print x) )
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
2
4
6
8
```

```
10
12
14
16
18

20
```

## 10.3 do 构造

do 结构也可用于使用 LISP 进行迭代。它提供了迭代的一种结构形式。

*do 语句的语法*：

```
(do (variable1 value1 updated-value1) (variable2 value2 updated-value2)
(variable3 value3 updated-value3) ... (test return-value) (s-expressions))
```

每个变量的初始值的计算和结合到各自的变量。每个子句中更新的值对应于一个可选的更新语句，指定变量的值将在每次迭代更新。每次迭代后，将测试结果进行评估计算，并且如果它返回一个 nil 或 true，则返回值被求值并返回。最后一个 S-表达式（s）是可选的。如果有，它们每一次迭代后执行，直到测试返回 true 值。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(do ((x 0 (+ 2 x)) (y 20 ( - y 2))) ((= x y)(- x y)) (format t "~% x = ~d  y = ~d" x y))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
x = 0  y = 20 x = 2  y = 18 x = 4  y = 16 x = 6  y = 14 x = 8  y = 12
```

## 10.4 dotimes 构造

dotimes 构造允许循环一段固定的迭代次数。

**实例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(dotimes (n 11)      (print n) (prinl (* n n)))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
0 0 1 1 2 4 3 9 4 16 5 25 6 36 7 49 8 64 9 81 10 100
```

## 10.5 dolist 构造

dolist 来构造允许迭代通过列表的每个元素。

**实例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(dolist (n '(1 2 3 4 5 6 7 8 9))
   (format t "~% Number: ~d Square: ~d" n (* n n)))
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
Number: 1 Square: 1 Number: 2 Square: 4 Number: 3 Square: 9 Number: 4 Square: 16
Number: 5 Square: 25 Number: 6 Square: 36 Number: 7 Square: 49 Number: 8 Square: 64
Number: 9 Square: 81
```

## 10.6 退出块

块返回，从允许从正常情况下的任何错误的任何嵌套块退出。块功能允许创建一个包含零个或多个语句组成的机构命名块。语法是：

```
(block block-name( ... ... ))
```

返回 – 从函数接受一个块名称和可选（默认为零）的返回值。

下面的例子演示了这一点：

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun demo-function (flag)      (print 'entering-outer-block)
   (block outer-block
     (print 'entering-inner-block)        (print (block inner-block
                 (if flag
                   (return-from outer-block 3)                  (return-from inner-block 5))
(print 'This-wil--not-be-printed)))
     (print 'left-inner-block)        (print 'leaving-outer-block)
     t))
(demo-function t)
(terpri)

(demo-function nil)
```

当单击 Execute 按钮，或按下 Ctrl+ E，LISP 立即执行它，返回的结果是：

```
ENTERING-OUTER-BLOCK
ENTERING-INNER-BLOCK

ENTERING-OUTER-BLOCK
ENTERING-INNER-BLOCK
5
LEFT-INNER-BLOCK

LEAVING-OUTER-BLOCK
```

# 11 LISP - 函数

函数是一组一起执行任务的语句。可以把代码放到单独的函数。如何划分代码之前不同的功能，但在逻辑上划分通常是这样每个函数执行特定的任务。

## 11.1 LISP-函数定义

命名函数 defun 宏用于定义函数。该函数的 defun 宏需要三个参数：

- 函数名称
- 函数的参数
- 函数的体

*defun 语法是*：

```
(defun name (parameter-list)  "Optional documentation string."  body)
```

让我们举例说明概念，简单的例子。

**例子** 1

让我们编写了一个名为 averagenum，将打印四个数字的平均值的函数。我们将会把这些数字作为参数。创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun averagenum (n1 n2 n3 n4)    (/ ( + n1 n2 n3 n4) 4)) (write(averagenum 10 20 30 40))
```

当执行的代码，它返回以下结果：

```
25
```

**示例** 2

让我们定义和调用函数，将计算出的圆的面积，圆的半径被指定作为参数的函数。创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun area-circle(rad) "Calculates area of a circle with given radius"  (terpri)  (format t "Radius:
```

```
~5f" rad)  (format t "~%Area: ~10f" (* 3.141592 rad rad)))  (area-circle 10)
```

当执行的代码，它返回以下结果：

*请注意*：

- 可以提供一个空的列表作为参数，这意味着函数没有参数，该列表是空的，表示为()。
- LISP 还允许可选，多个和关键字参数。
- 文档字符串描述了函数的目的。它与函数名相关联，并且可以使用文档函数来获得。
- 函数的主体可以包含任意数量的 Lisp 表达式。
- 在主体内的最后一个表达式的值返回函数的值。
- 还可以使用返回 - 从特殊的运算符函数返回一个值。

我们在简要讨论上述概念。更多高级主题请自行搜索或等待下一版加入（编者注）

- 可选参数
- 其余部分参数
- 关键字参数
- 从函数返回的值
- lambda 函数
- 映射函数

## 11.2 可选参数

可以使用可选参数定义一个函数。要做到这一点，需要把符号与可选的可选参数的名称之前。我们将只是显示它接收的参数的函数。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun show-members (a b &optional c d) (write (list a b c d))) (show-members 1 2 3) (terpri) (show-members 'a 'b 'c 'd) (terpri) (show-members 'a 'b) (terpri) (show-members 1 2 3 4)
```

当执行代码，它返回以下结果：

```
(1 2 3 NIL) (A B C D) (A B NIL NIL) (1 2 3 4)
```

请注意，参数 c 和 d 是在上面的例子中，是可选参数。

## 11.3 其余部分参数

有些函数需要采用可变数目的参数。例如，我们使用格式化函数需要两个必需的参数，数据流和控制字符串。然而，该字符串后，它需要一个可变数目的取决于要显示的字符串中的值的数目的参数。同样，+ 函数，或 * 函数也可以采取一个可变数目的参数。可以提供这种可变数目的使用符号与其余参数。下面的例子说明了这个概念：

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun show-members (a b &rest values) (write (list a b values))) (show-members 1 2 3) (terpri) (show-members 'a 'b 'c 'd) (terpri) (show-members 'a 'b) (terpri) (show-members 1 2 3 4) (terpri) (show-members 1 2 3 4 5 6 7 8 9)
```

当执行代码，它返回以下结果：

```
(1 2 (3)) (A B (C D)) (A B NIL) (1 2 (3 4)) (1 2 (3 4 5 6 7 8 9))
```

## 11.4 关键字参数

关键字参数允许指定哪个值与特定的参数。它使用的是 &key 符号表示。当发送的值到

该函数必须先于值 :parameter-name.下面的例子说明了这个概念。

**例子**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun show-members (&key a b c d ) (write (list a b c d))) (show-members :a 1 :c 2 :d 3) (terpri) (show-members :a 'p :b 'q :c 'r :d 's) (terpri) (show-members :a 'p :d 'q) (terpri) (show-members :a 1 :b 2)
```

当执行代码，它返回以下结果：

```
(1 NIL 2 3) (P Q R S) (P NIL NIL Q) (1 2 NIL NIL)
```

## 11.5 从函数返回的值

默认情况下，在 LISP 函数返回最后一个表达式作为返回值的值。下面的例子将证明这一点。

**示例 1**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun add-all(a b c d)   (+ a b c d)) (setq sum (add-all 10 20 30 40)) (write sum) (terpri) (write (add-all 23.4 56.7 34.9 10.0))
```

当执行代码，它返回以下结果：

```
100 125.0
```

但是，可以使用返回- 从特殊的操作符立即从函数返回任何值。

**示例 2**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun myfunc (num)    (return-from myfunc 10)    num) (write (myfunc 20))
```

当执行代码，它返回以下结果：

```
10
```

更改一点点代码：

```
(defun myfunc (num)    (return-from myfunc 10)    write num) (write (myfunc 20))
```

它仍然返回：

```
10
```

## 11.6 lambda 函数

有时，可能需要一个函数只在一个程序中的位置和功能是如此的微不足道，可能不给它一个名称，也可以不喜欢它存储在符号表中，宁可写一个未命名或匿名函数。LISP 允许编写评估计算在程序中遇到的匿名函数。这些函数被称为 Lambda 函数。可以使用 lambda 表达式创建这样的功能。lambda 表达式语法如下：

```
(lambda (parameters) body)
```

lambda 形式可以不进行评估计算，它必须出现只有在 LISP 希望找到一个函数。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write ((lambda (a b c x)    (+ (* a (* x x)) (* b x) c))    4 2 9 3))
```

当执行代码，它返回以下结果：

```
51
```

## 11.7 映射函数

映射函数是一组函数，可以连续地施加于元件中的一个或多个列表。应用这些功能列表的结果被放置在一个新的列表，而新的列表返回。

例如，mapcar 函数处理的一个或多个列表连续元素。

在 mapcar 函数的第一个参数应该是一个函数，其余的参数是该函数的应用列表（次）。

函数的参数被施加到连续的元素，结果为一个新构造的列表。如果参数列表是不相等的长度，然后映射的过程停止在达到最短的列表的末尾。结果列表将元素作为最短输入列表的数目相同。

**示例 1**

让我们从一个简单的例子和数字 1 添加到每个列表的元素( 23 34 45 56 67 78 89)。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write (mapcar '1+ '(23 34 45 56 67 78 89)))
```

当执行代码，它返回以下结果：

```
(24 35 46 57 68 79 90)
```

**示例 2**

让我们写这将多维数据集列表中的元素的函数。让我们用一个 lambda 函数用于计算数字的立方。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun cubeMylist(lst) (mapcar #'(lambda(x) (* x x x)) lst)) (write (cubeMylist '(2 3 4 5 6 7 8 9)))
```

当执行代码，它返回以下结果：

```
(8 27 64 125 216 343 512 729)
```

示例 3

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write (mapcar '+ '(1 3 5 7 9 11 13) '( 2 4 6 8)))
```

当执行代码，它返回以下结果：

```
(3 7 11 15)
```

# 12 LISP - 谓词

谓词是函数，测试其参数对一些特定的条件和返回 nil，如果条件为假，或某些非 nil 值条件为 true。

下表显示了一些最常用的谓词：

| 谓词 | 描述 |
| --- | --- |
| atom | 它接受一个参数，并返回 t 如果参数是一个原子或，否则 nil。 |
| equal | 它有两个参数，并返回 t，如果他们在结构上相同或否则 nil |
| eq | 它有两个参数，并返回 t，如果它们是相同的相同的对象，共享相同的内存位置或否则 nil |
| eql | 它有两个参数，并返回 t 如果参数相等，或者如果他们是同一类型具有相同值的数字，或者如果他们是代表相同的字符的字符对象，否则返回 nil |
| evenp | 它接受一个数字参数，并返回 t 如果参数为偶数或否则为 nil。 |
| oddp | 它接受一个数字参数，并返回 t 如果参数为奇数或否则为 nil。 |
| zerop | 它接受一个数字参数，并返回 t 如果参数是零或否则为 nil。 |
| null | 它接受一个参数，并返回 t，如果参数的计算结果为 nil，否则返回 nil。 |
| listp | 它接受一个参数，并返回 t 如果参数的计算结果为一个列表，否则返回 nil。 |
| greaterp | 这需要一个或多个参数，并返回 t，如果不是有一个单一的参数或参数是从左到右，或如果无先后，否则为 nil。 |
| lessp | 这需要一个或多个参数，并返回 t，如果不是有一个单一的参数或参数是从左到右依次更小的向右，或否则为 nil. |

| | |
|---|---|
| numberp | 它接受一个参数，并返回 t 如果参数是一个数字，否则为 nil。 |
| symbolp | 它接受一个参数，并返回 t 如果参数是一个符号，否则返回 nil。 |
| integerp | 它接受一个参数，并返回 t 如果参数是一个整数，否则返回 nil。 |
| rationalp | 它接受一个参数，并返回 t 如果参数是有理数，无论是比例或数量，否则返回 nil>。 |
| floatp | 它接受一个参数，并返回 t 当参数则返回一个浮点数否则为 nil。 |
| realp | 它接受一个参数，并返回 t 如果参数是一个实数，否则返回 nil。 |
| complexp | 它接受一个参数，并返回 t 如果参数是一个复数，否则返回 nil。 |
| characterp | 它接受一个参数，并返回 t 如果参数是一个字符，否则返回 nil。 |
| stringp | 它接受一个参数，并返回 t，如果参数是一个字符串对象，否则返回 nil。 |
| arrayp | 它接受一个参数，并返回 t 如果参数是一个数组对象，否则返回 nil。 |
| packagep | 它接受一个参数，并返回 t，如果参数是一个包，否则返回 nil。 |

**示例** 1

创建一个名为 `main.lisp` 一个新的源代码文件，并在其中输入如下代码：

```
(write (atom 'abcd))
(terpri)
(write (equal 'a 'b))
(terpri)
(write (evenp 10))
(terpri)
(write (evenp 7 ))
(terpri)
(write (oddp 7 ))
(terpri)
(write (zerop 0.0000000001))
(terpri)
(write (eq 3 3.0 ))
(terpri)
(write (equal 3 3.0 ))
(terpri)
(write (null nil ))
```

当执行以上代码，它返回以下结果：

```
T
NIL
T
NIL
T
NIL
NIL
NIL

T
```

示例 2

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun factorial (num)    (cond ((zerop num) 1)          (t ( * num (factorial (- num 1))))))) (setq n 6)
(format t "~% Factorial ~d is: ~d" n (factorial n))
```

当执行以上代码，它返回以下结果：

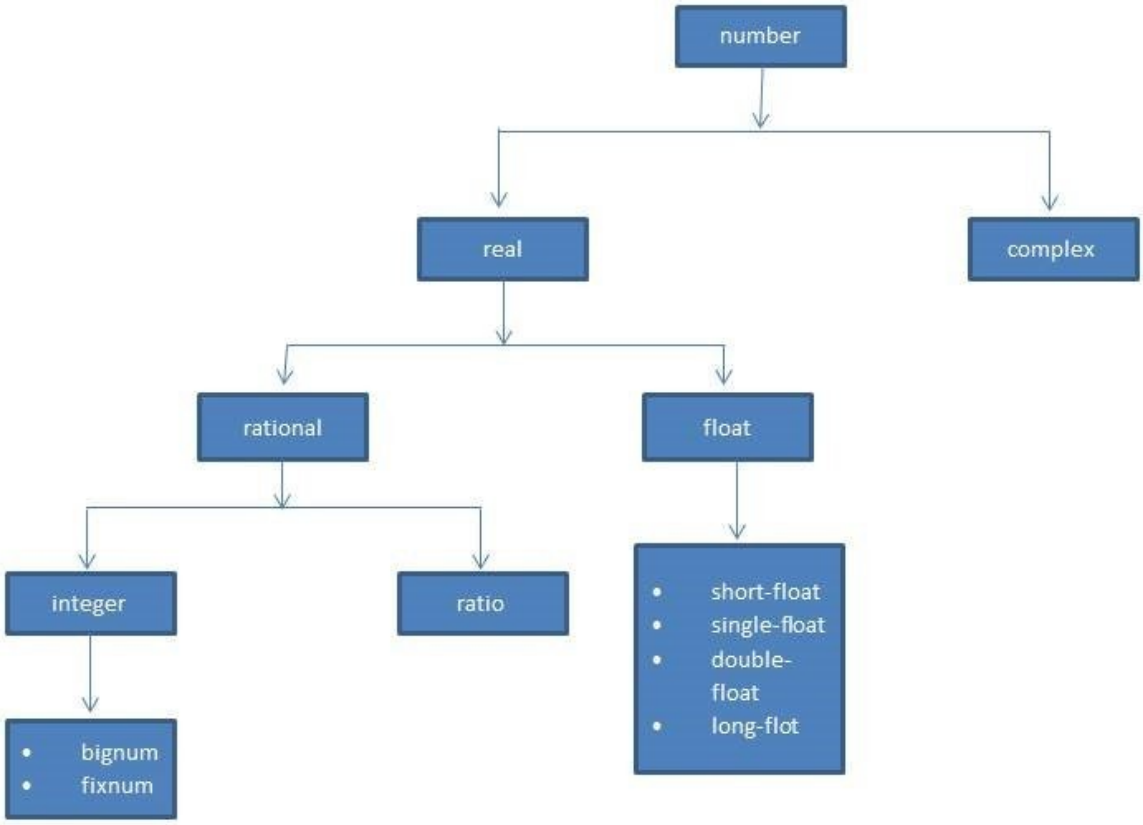
```
Factorial 6 is: 720
```

# 13 LISP - 数字

数字——通过 LISP 支持数类型是：

- Integers
- Ratios
- Floating-yiibai numbers
- Complex numbers

下图显示的数量和层次在 LISP 提供的各种数字数据类型：






## 13.1 在 LISP 各种数值类型

下表描述了 LISP 语言提供的各种数字类型的数据：

| Data type | 描述 |
|---|---|
| fixnum | 这个数据类型表示的整数哪些不是太大，大多在范围-215 到 215-1(它是依赖于机器) |
| bignum | 这些都是非常大的数字有大小受限于内存中分配 LISP 量，它们不是长整数数字。 |
| ratio | 表示两个数中的分子/分母形式的比率。在/函数总是产生结果的比率，当其参数都是整数。 |
| float | 它表示非整数。还有随着精密四个浮点数据类型。 |
| complex | 它表示复数，这是由#C 表示。实部和虚部可以是两者或者理性或浮点数。 |

**示例**

创建一个名为 `main.lisp` 一个新的源代码文件，并在其中输入如下代码：

```
(write (/ 1 2)) (terpri) (write ( + (/ 1 2) (/ 3 4))) (terpri) (write ( + #c( 1 2) #c( 3 -4)))
```

当执行以上代码，它返回以下结果：

```
1/2 5/4 #C(4 -2)
```

## 13.2 数字函数

下表描述了一些常用的数值函数：

| Function | 描述 |
| --- | --- |
| +, -, *, / | 各算术运算 |
| sin, cos, tan, acos, asin, atan | 相应的三角函数 |
| sinh, cosh, tanh, acosh, asinh, atanh | 相应的双曲函数 |
| exp | 幂函数，计算 ex |
| expt | 幂函数，需要基础和幂两者 |
| sqrt | 它可以计算一个数的平方根 |
| log | 对数函数。它的一个参数给出，则它计算其自然对数，否则将第二个参数被用作基数 |
| conjugate | 它计算一个数的复共轭，如有任何实数，它返回数字本身 |
| abs | 它返回一个数的绝对值（或幅度） |
| gcd | 它可以计算给定数字的最大公约数 |
| lcm | 它可以计算给定数的最小公倍数 |
| isqrt | 它提供了最大的整数小于或等于一个给定的自然数的精确平方根。 |
| floor, ceiling, truncate, round | 所有这些函数把一个数字的两个参数，并返回商;地面返回的最大整数不大于比，天花板选择较小的整数，它比比率越大，截断选择相同符号的整数的比值与最大的绝对值是小于的比值的绝对值，与圆公司选用一个整数，它是最接近比值 |
| ffloor, fceiling, ftruncate, fround | 确实与上述相同，但返回的商作为一个浮点数 |

| | |
|---|---|
| mod, rem | 返回除法运算的余数 |
| float | 将实数转换为浮点数 |
| rational, rationalize | 将实数转换为有理数 |
| numerator, denominator | 返回有理数的各个部分 |
| realpart, imagpart | 返回一个复数的实部和虚部 |

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write (/ 45 78)) (terpri) (write (floor 45 78)) (terpri) (write (/ 3456 75)) (terpri) (write (floor
3456 75)) (terpri) (write (ceiling 3456 75)) (terpri)(write (truncate 3456 75)) (terpri) (write (round
3456 75)) (terpri) (write (ffloor 3456 75)) (terpri) (write (fceiling 3456 75)) (terpri) (write
(ftruncate 3456 75)) (terpri) (write (fround 3456 75)) (terpri) (write (mod 3456 75)) (terpri) (setq c
(complex 6 7)) (write c) (terpri) (write (complex 5 -9)) (terpri) (write (realpart c)) (terpri) (write
(imagpart c))
```

当执行以上代码，它返回以下结果：

```
15/26 0 1152/25 46 47 46 46 46.0 47.0 46.0 46.0 6 #C(6 7) #C(5 -9) 6 7
```

# 14 LISP - 字符

在 LISP 中，字符被表示为字符类型的数据对象。可以记#前字符本身之前的字符的对象。例如，#一个表示字符 a。空格和其它特殊字符可以通过 # 前面的字符的名称前表示。例如，#空格代表空格字符。下面的例子演示了这一点：

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write 'a)
```

```
(terpri)
(write #a)
(terpri)
(write-char #a)
(terpri)

(write-char 'a)
```

当执行以上代码，它返回以下结果：

```
A
#a a

*** - WRITE-CHAR: argument A is not a character
```

## 14.1 特殊字符

Common Lisp 允许使用以下特殊字符在代码。他们被称为半标准字符。

- #Backspace
- #Tab
- #Linefeed
- #Page
- #Return
- #Rubout

## 14.2 字符比较函数

数字比较函数和运算符，如，< 和 >上字符不工作。 Common Lisp 提供了另外两组的功能，在代码中比较字符。一组是区分大小写的，而另一个不区分大小写。

下表提供的功能：

| Case Sensitive Functions | Case-insensitive Functions | 描述 |
| --- | --- | --- |
| char= | char-equal | 检查如果操作数的值都相等与否，如果是的话那么条件为真。 |
| char/= | char-not-equal | 检查如果操作数的值都不同，或没有，如果值不相等，则条件为真。 |
| char< | char-lessp | 检查如果操作数的值单调递减。 |
| char> | char-greaterp | 检查如果操作数的值单调递增。 |
| char<= | char-not-greaterp | 如有左操作数的值大于或等于下一个右操作数的值，如果是则条件为真检查。 |
| char>= | char-not-lessp | 如有左操作数的值小于或等于其右操作数的值，如果是，则条件为真检查。 |

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
; case-sensitive comparison
(write (char= #a #)) (terpri) (write (char= #a #a)) (terpri) (write (char= #a #A)) (terpri) ;case-
insensitive comparision

(write (char-equal #a #A)) (terpri) (write (char-equal #a #)) (terpri) (write (char-lessp #a # #c))
(terpri) (write (char-greaterp #a # #c))
```

当执行以上代码，它返回以下结果：

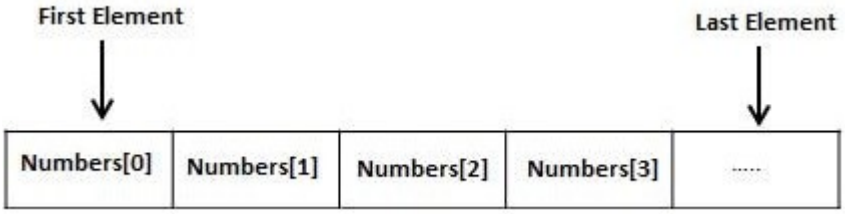
```
NIL
T
NIL
T
NIL
T

NIL
```

# 15 LISP - 数组

LISP 允许使用 make-array 函数来定义一个或多个维数组。一个数组可以任意 LISP 对象

存储为它的元素。所有数组组成的连续的存储单元。最低的地址对应于第一个元素和最高地址的最后一个元素。

数组的维数被称为它的秩。




在 LISP 语言中，
数组元素是由一个非负整数索引的顺序指定。该序列的长度必须等于数组的秩。索引从 0 开始。

例如，要创建一个数组，10 - 单元格，命名为 my-array，我们可以这样写：

```
(setf my-array (make-array '(10)))
```

aref 函数允许访问该单元格的内容。它有两个参数，数组名和索引值。

例如，要访问的第十单元格的内容，可以这样编写：

```
(aref my-array 9)
```

**示例 1**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write (setf my-array (make-array '(10))))
(terpri)
(setf (aref my-array 0) 25)
(setf (aref my-array 1) 23)
(setf (aref my-array 2) 45)
(setf (aref my-array 3) 10)
(setf (aref my-array 4) 20)
(setf (aref my-array 5) 17)
(setf (aref my-array 6) 25)
(setf (aref my-array 7) 19)
(setf (aref my-array 8) 67)
(setf (aref my-array 9) 30)

(write my-array)
```

当执行以上代码，它返回以下结果：

```
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL) #(25 23 45 10 20 17 25 19 67 30)
```

**示例 2**

让我们创建一个 3×3 数组。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setf x (make-array '(3 3)
        :initial-contents '((0 1 2 ) (3 4 5) (6 7 8)))) (write x)
```

当执行以上代码，它返回以下结果：

```
#2A((0 1 2) (3 4 5) (6 7 8))
```

**示例 3**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a (make-array '(4 3)))
(dotimes (i 4)
  (dotimes (j 3)
    (setf (aref a i j) (list i 'x j '= (* i j)))))
(dotimes (i 4)
  (dotimes (j 3)
    (print (aref a i j))))
```

当执行以上代码，它返回以下结果：

```
(0 x 0= 0)
(0 x 1 = 0)
(0 x 2 = 0)
(1 x 0 = 0)
(1 x 1 = 1)
(1 x 2 = 2)
(2 x 0 = 0)
```

```
(2 x 1 = 2)
(2 x 2 = 4)
(3 x 0 = 0)
(3 x 1 = 3)

(3 x 2 = 6)
```

## 15.1 make-array 函数完整的语法

make-array 函数需要许多其他的参数。让我们来看看这个函数的完整语法：

```
make-array dimensions :element-type :initial-element :initial-contents :adjustable :fill-yiibaier
:displaced-to :displaced-index-offset
```

除了维度参数，所有其他参数都是关键字。下表提供的参数简要说明。

| 参数 | 描述 |
| --- | --- |
| dimensions | 它给该数组的大小。它是一个数字为一维数组，而对于多维数组列表。 |
| :element-type | 它是类型说明符，默认值是 T，即任何类型 |
| :initial-element | 初始元素值。它将使一个数组的所有初始化为一个特定值的元素。 |
| :initial-content | 初始内容作为对象。 |
| :adjustable | 它有助于创造一个可调整大小(或可调)向量，其底层的内存可以调整大小。该参数是一个布尔值，表示数组是否可调与否，默认值是 nil。 |
| :fill-yiibaier | 它跟踪实际存储在一个可调整大小的矢量元素的数目 |
| :displaced-to | 它有助于创造一个移位的数组或共享数组共享其内容与指定的数组。这两个数组应该有相同的元素类型。位移到选项可能无法使用:displaced-to 或:initial-contents 选项。此参数默认为 nil。 |
| :displaced-index-offset | 它给出了索引偏移创建的共享数组。 |

**示例 4**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq myarray (make-array '(3 2 3)
        :initial-contents
        '(((a b c) (1 2 3))
              ((d e f) (4 5 6))
              ((g h i) (7 8 9))
              )))
(setq array2 (make-array 4 :displaced-to myarray
                    :displaced-index-offset 2))
```

```
(write myarray) (terpri) (write array2)
```

当执行以上代码，它返回以下结果：

```
#3A(((A B C) (1 2 3)) ((D E F) (4 5 6)) ((G H I) (7 8 9))) #(C 1 2 3)
```

若对数组是二维的：

```
(setq myarray (make-array '(3 2 3)
        :initial-contents
        '(((a b c) (1 2 3))
              ((d e f) (4 5 6))
              ((g h i) (7 8 9))
              )))
(setq array2 (make-array '(3 2) :displaced-to myarray
               :displaced-index-offset 2))
(write myarray)
(terpri)
(write array2)
```

当执行以上代码，它返回以下结果：

```
#3A(((A B C) (1 2 3)) ((D E F) (4 5 6)) ((G H I) (7 8 9))) #2A((C 1) (2 3) (D E))
```

让我们改变流离指数偏移量 5：

```
(setq myarray (make-array '(3 2 3)
        :initial-contents
        '(((a b c) (1 2 3))
              ((d e f) (4 5 6))
              ((g h i) (7 8 9))
              )))
(setq array2 (make-array '(3 2) :displaced-to myarray
               :displaced-index-offset 5))
(write myarray)
(terpri)
(write array2)
```

当执行以上代码，它返回以下结果：

#3A(((A B C) (1 2 3)) ((D E F) (4 5 6)) ((G H I) (7 8 9))) #2A((3 D) (E F) (4 5))

**示例 5**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
;a one dimensional array with 5 elements,
;initail value 5 (write (make-array 5 :initial-element 5)) (terpri) ;two dimensional array, with initial
element a
(write (make-array '(2 3) :initial-element 'a)) (terpri) ;an array of capacity 14, but fill yiibaier 5, is 5
(write(length (make-array 14 :fill-yiibaier 5))) (terpri) ;however its length is 14 (write (array-
dimensions (make-array 14 :fill-yiibaier 5))) (terpri) ; a bit array with all initial elements set to 1
(write(make-array 10 :element-type 'bit :initial-element 1))
(terpri)
; a character array with all initial elements set to a
; is a string actually
(write(make-array 10 :element-type 'character :initial-element #a)) (terpri) ; a two dimensional array
with initial values a
(setq myarray (make-array '(2 2) :initial-element 'a :adjustable t)) (write myarray) (terpri) ;readjusting
the array
(adjust-array myarray '(1 3) :initial-element 'b)

(write myarray)
```

当执行以上代码，它返回以下结果：

#(5 5 5 5 5) #2A((A A A) (A A A)) 5 (14) #*1111111111 "aaaaaaaaaa" #2A((A A) (A A)) #2A((A A B))

# 16 LISP - 符号

在 LISP 语言中，符号是表示数据对象和有趣的是它也是一个数据对象的名称。是什么使得符号特殊之处在于他们有分别叫 property list,或 plist.

## 16.1 属性列表

LISP 可以让属性，以符号分配。例如，我们有一个' 人' 的对象。希望这个' 人' 的对象有像姓名，性别，身高，体重，住址，职业等属性是一些属性名称。一个属性列表被实现为具

有元素为偶数(可能为零)的列表。每对列表中的元素构成一个条目;第一个项目是指标，而第二个是该值。当创建一个符号，它的属性列表最初是空的。属性是使用于 asetf 形式得到建立。

例如，下面的语句使我们能够分配属性标题，作者和出版商，以及相应的值，命名(符号)'书'的对象。

**示例** 1

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
((write (setf (get 'books'tit1e) '(Gone with the Wind)))

(terpri)

(write (setf (get 'books 'author) '(Margaret Miche1)))

(terpri)

(write (setf (get 'books 'publisher) '(Warner Books)))
```

当执行代码，它返回以下结果：

```
(GONE WITH THE WIND)

(MARGARET MICHEL)

(WARNER BOOKS)
```

各种属性列表功能允许你指定的属性以及检索，替换或删除一个符号的属性。

get 函数返回符号的属性列表对于一个给定的指标。它的语法如下：

```
get symbol indicator &optional default
```

get 函数查找指定的指标给定的符号的属性列表，如果找到则返回相应的值;否则默认返回(或 nil，如果没有指定默认值)。

## 示例 2

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setf (get 'books 'title) '(Gone with the Wind))

(setf (get 'books 'author) '(Margaret Micheal))

(setf (get 'books 'publisher) '(Warner Books))

(write (get 'books 'title))

(terpri)

(write (get 'books 'author))

(terpri)

(write (get 'books 'publisher))
```

当执行代码，它返回以下结果：

```
(GONE WITH THE WIND)

(MARGARET MICHEAL)

(WARNER BOOKS)
```

symbol-plist 函数可以看到一个符号的所有属性。

**示例 3**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setf (get 'annie 'age) 43)

(setf (get 'annie 'job) 'accountant)

(setf (get 'annie 'sex) 'female)

(setf (get 'annie 'children) 3)

(terpri)

(write (symbol-plist 'annie))
```

当执行代码，它返回以下结果：

```
(CHILDREN 3 SEX FEMALE JOB ACCOUNTANT AGE 43)
```

remprop 函数从符号中删除指定的属性。

**示例 4**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setf (get 'annie 'age) 43)

(setf (get 'annie 'job) 'accountant)

(setf (get 'annie 'sex) 'female)

(setf (get 'annie 'children) 3)

(terpri)

(write (symbol-plist 'annie))

(remprop 'annie 'age)

(terpri)

(write (symbol-plist 'annie))
```

当执行代码，它返回以下结果：

```
(CHILDREN 3 SEX FEMALE JOB ACCOUNTANT AGE 43)

(CHILDREN 3 SEX FEMALE JOB ACCOUNTANT)
```

# 17 LISP - 向量

向量是一维数组，数组因此子类型。向量和列表统称序列。因此，我们迄今为止所讨论的所有序列的通用函数和数组函数，工作在向量上。

## 17.1 创建向量

向量函数使可以使用特定的值固定大小的向量。这需要任意数量的参数，并返回包含这些参数的向量。

**示例 1**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setf v1 (vector 1 2 3 4 5)) (setf v2 #(a b c d e)) (setf v3 (vector 'p 'q 'r 's 't))
(write v1)
(terpri)
(write v2)
(terpri)

(write v3)
```

当执行代码，它返回以下结果：

```
#(1 2 3 4 5) #(A B C D E) #(P Q R S T)
```

请注意，LISP 使用 #(...)语法为向量的文字符号。可以使用此 #(...)语法来创建并包含在代码中的文字向量。然而，这些是文字向量，所以修改它们没有在 LISP 语言中定义。因此，对于编程，应始终使用向量函数，或者 make-array 函数来创建打算修改的向量。

*make-array 函数是比较通用的方式来创建一个矢量*。可以访问使用 aref 函数的矢量元素。

**示例 2**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a (make-array 5 :initial-element 0)) (setq b (make-array 5 :initial-element 2)) (dotimes (i 5)
(setf (aref a i) i)) (write a)(terpri) (write b) (terpri)
```

当执行代码，它返回以下结果：

```
#(0 1 2 3 4) #(2 2 2 2 2)
```

## 17.2 Fill 指针

make-array 函数允许创建一个可调整大小的矢量。

函数 fill-yiibaier 参数跟踪实际存储在向量中的元素的数量。它的下一个位置，当添加元素的向量来填充的索引。

vector-push 函数允许将元素添加到一个可调整大小的矢量的结束。它增加了填充指针加 1。

vector-pop 函数返回最近推条目，由 1 递减填充指针。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq a (make-array 5 :fill-yiibaier 0)) (write a) (vector-push 'a a)
(vector-push 'b a) (vector-push 'c a)
(terpri)
(write a)
(terpri)
(vector-push 'd a) (vector-push 'e a)
;this will not be entered as the vector limit is 5

(vector-push 'f a) (write a) (terpri) (vector-pop a) (vector-pop a) (vector-pop a) (write a)
```

当执行代码，它返回以下结果：

```
#() #(A B C) #(A B C D E) #(A B)
```

向量是序列，所有序列函数是适用于向量。请参考序列章节，对向量函数。

# 18 LISP - 集合

Common Lisp 不提供的一组数据类型。然而，它提供的函数数量，它允许一组操作，以可以在列表上执行。可以添加，删除和搜索列表中的项目，根据不同的标准。还可以执行像不同的集合运算：并，交和集合差。

## 18.2 实现 LISP 集合

集合像列表一样，一般实现的利弊单元。由于这个原因，集合操作越来越少，高效的获

取大的集合。要明白这一点，一旦我们深入研究这个问题更深一点。

*adjoin 函数可建立一个集合*。这需要一个条目和一个列表表示一组，并返回表示包含该项目，并在原设定的所有项目的集合列表。adjoin 函数首先查找的条目给定列表中，一旦找到，将返回原来的名单;否则，创建一个新的 cons 单元，其 car 作为该目条，cdr 指向原来的列表并返回这个新列表。该毗函数也需要:key 和 :test 关键字参数。这些参数用于检查该条目是否存在于原始列表。因为，adjoin 函数不会修改原来的列表，让列表本身的变化，必须指定由 adjoin 到原始列表返回的值或者可以使用宏 pushnew 将条目添加到集合。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
; creating myset as an empty list
(defparameter *myset* ()) (adjoin 1 *myset*) (adjoin 2 *myset*) ; adjoin didn't change the original set
;so it remains same
(write *myset*)
(terpri)
(setf *myset* (adjoin 1 *myset*))
(setf *myset* (adjoin 2 *myset*))
;now the original set is changed
(write *myset*)
(terpri)
;adding an existing value
(pushnew 2 *myset*)
;no duplicate allowed
(write *myset*)
(terpri)
;pushing a new value
(pushnew 3 *myset*)
(write *myset*)

(terpri)
```

当执行代码，它返回以下结果：

```
NIL

(2 1) (2 1) (3 2 1)
```

## 18.3 检查成员

函数的成员组允许检查一个元素是否是一个集合成员。

以下是这些函数的语法：

```
member item list &key :test :test-not :key
member-if predicate list &key :key

member-if-not predicate list &key :key
```

这些函数搜索给定列表中一个给定的项，满足了测试。它没有这样的项被找到，则函数返回 nil。否则，将返回列表中的元素作为第一个元素的尾部。搜索是只在顶层进行。这些函数可作为谓词。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(write (member 'zara '(ayan abdul zara riyan nuha))) (terpri) (write (member-if #'evenp '(3 7 2 5/3 'a)))
(terpri) (write (member-if-not #'numberp '(3 7 2 5/3 'a 'b 'c)))
```

当执行代码，它返回以下结果：

```
(ZARA RIYAN NUHA) (2 5/3 'A)

('A 'B 'C)
```

## 18.4 集合联合

联合组功能能够在作为参数提供给这些功能测试的基础上，两个列表进行集联合。

以下是这些函数的语法：

```
union list1 list2 &key :test :test-not :key

nunion list1 list2 &key :test :test-not :key
```

union 函数有两个列表，并返回一个包含所有目前无论是在列表中的元素的新列表。如果有重复，则该成员只有一个副本被保存在返回的列表。union 函数执行相同的操作，但可能会破坏参数列表。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq set1 (union '(a b c) '(c d e))) (setq set2 (union '(#(a b) #(5 6 7) #(f h))
    '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch))
(setq set3 (union '(#(a b) #(5 6 7) #(f h))

    '(#(5 6 7) #(a b) #(g h)))) (write set1) (terpri) (write set2) (terpri) (write set3)
```

当执行代码，它返回以下结果：

```
(A B C D E) (#(F H) #(5 6 7) #(A B) #(G H)) (#(A B) #(5 6 7) #(F H) #(5 6 7) #(A B) #(G H))
```

*请注意：*

对于三个向量列表 :test-not #' 不匹配的参数：如预期的 union 函数不会工作。这是因为，该名单是由 cons 单元元素，虽然值相同的外观明显，单元元素 cdr 部分不匹配，所以他们 并不完全一样，以 LISP 解释器/编译器。这是原因;实现大集全不建议使用的列表。它工作正常的小集合上。

## 18.5 交集

函数的交点组允许作为参数提供给这些函数测试的基础上，两个列表进行交点。以下是这些函数的语法：

```
intersection list1 list2 &key :test :test-not :key

nintersection list1 list2 &key :test :test-not :key
```

这些函数需要两个列表，并返回一个包含所有目前在这两个参数列表中的元素的新列表。如果任一列表中的重复项，冗余项可能会或可能不会出现在结果中。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq set1 (intersection '(a b c) '(c d e))) (setq set2 (intersection '(#(a b) #(5 6 7) #(f h))
```

```
    '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch))
(setq set3 (intersection '(#(a b) #(5 6 7) #(f h))

    '(#(5 6 7) #(a b) #(g h)))) (write set1) (terpri) (write set2) (terpri) (write set3)
```

当执行代码，它返回以下结果：

```
(C) (#(A B) #(5 6 7)) NIL
```

intersection 函数是相交的破坏性版本，也就是说，它可能会破坏原始列表。

## 18.6 差集

set-difference 组差集，可以在作为参数提供给这些功能测试的基础上，两个列表进行差集。以下是这些函数的语法：

```
set-difference list1 list2 &key :test :test-not :key

nset-difference list1 list2 &key :test :test-not :key
```

set-difference 函数返回，不会出现在第二个列表的第一个列表的元素的列表。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq set1 (set-difference '(a b c) '(c d e))) (setq set2 (set-difference '(#(a b) #(5 6 7) #(f h))
    '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch)) (setq set3 (set-difference '(#(a b) #(5 6 7) #(f h))

    '(#(5 6 7) #(a b) #(g h)))) (write set1) (terpri) (write set2) (terpri) (write set3)
```

当执行代码，它返回以下结果：

```
(A B) (#(F H)) (#(A B) #(5 6 7) #(F H))
```

# 19 LISP - 树

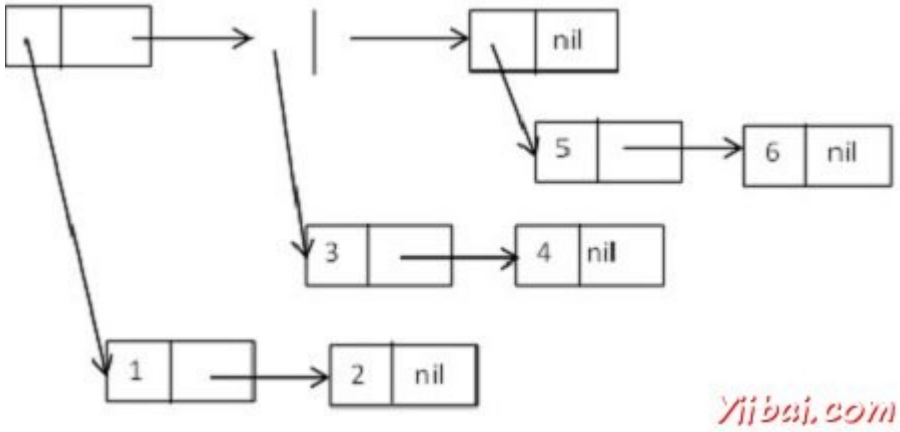
可以从 cons 单元构建树的数据结构，如清单列表。为了实现树形结构，则必须设计功能，将遍历 cons 单元，在特定的顺序，例如，前序，顺序和后序的二进制树。

## 19.1 树列表的列表

让我们考虑由 cons 单元的树状结构，形成列出的清单如下：

```
((1 2) (3 4) (5 6)).
```

图解，它可以表示为：




## 19.2 LISP 树的功能

虽然多数时候仍需要根据其它特殊需求编写自己的树的功能，LISP 提供了一些树的功能，您可以使用。

除了所有列表函数，以下是工作在树结构函数：

| 函数 | 描述 |
| --- | --- |
| copy-tree x &optional vecp | 它返回 cons 单元×树的副本。它递归地拷贝两款车和 cdr 方向。如果 x 不是一个 cons 单元，该函数只返回 x 不变。如果可选 vecp 参数为 true，这个函数将向量（递归），以及 cons 单元。 |

| | |
|---|---|
| tree-equal x y &key :test :test-not :key | 它比较两棵树的 cons 单元。如果 x 和 y 是两个 cons 单元，他们的汽车和 cdr 是递归比较。如果 x 和 y 都不是一个 cons 单元，它们是由 eql 比较，或根据指定的测试。:key 函数，如果指定，应用到这两个目录树中的元素。 |
| subst new old tree &key :test :test-not :key | 它可以代替出现给老项与新项，在树，这是 cons 单元的一棵树。 |
| nsubst new old tree &key :test :test-not :key | 它的工作原理与 subst 相同，但它破坏了原来的树。 |
| sublis alist tree &key :test :test-not :key | 它的工作原理就像 subst，只不过它采用的新旧对关联表 alist。树（应用后:key 函数，如果有的话）中的每个元素，与 alist 的车相比;如果它匹配，它被替换为相应的 cdr。 |
| nsublis alist tree &key :test :test-not :key | 它的工作原理与 sublis 相同，而是一个破坏性的版本。 |

**示例 1**

创建一个名为 `main.lisp` 一个新的源代码文件，并在其中输入如下代码：

```
(setq lst (list '(1 2) '(3 4) '(5 6)))
(setq mylst (copy-list lst))
(setq tr (copy-tree lst))
(write lst)
(terpri)
(write mylst)
(terpri)

(write tr)
```

当执行代码，它返回以下结果：

```
((1 2) (3 4) (5 6)) ((1 2) (3 4) (5 6)) ((1 2) (3 4) (5 6))
```

**示例 2**

创建一个名为 `main.lisp` 一个新的源代码文件，并在其中输入如下代码：

```
(setq tr '((1 2 (3 4 5) ((7 8) (7 8 9)))))
(write tr)
(setq trs (subst 7 1 tr))
(terpri)
(write trs)
```

当执行代码，它返回以下结果：

```
((1 2 (3 4 5) ((7 8) (7 8 9)))) ((7 2 (3 4 5) ((7 8) (7 8 9))))
```

## 19.3 建立自己的树

让我们尝试建立自己的树，使用 LISP 列表功能。

（1）首先，让我们创建一个包含一些数据的新节点：

```
(defun make-tree (item)  "it creates a new node with item."  (cons (cons item nil) nil))
```

（2）接下来让我们添加一个子节点插入到树：它会采取两种树节点，并添加第二棵树作为第一个的子树。

```
(defun add-child (tree child)  (setf (car tree) (append (car tree) child))  tree)
```

（3）接下来让我们添加一个子节点插入到树：这将需要一个树节点，并返回该节点第一个子节点，或 nil，如果这个节点没有任何子节点。

```
(defun first-child (tree)  (if (null tree)  nil  (cdr (car tree))))
```

（4）这个函数会返回一个给定节点的下一个同级节点：它需要一个树节点作为参数，并返回一个指向下一个同级节点，或者为 nil，如果该节点没有任何。

```
(defun next-sibling (tree)  (cdr tree))
```

（5）最后，我们需要一个函数来返回一个节点的信息：

```
(defun data (tree)  (car (car tree)))
```

### 示例

本示例使用上述功能：

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun make-tree (item)  "it creates a new node with item."  (cons (cons item nil) nil))  (defun first-child (tree)  (if (null tree)  nil  (cdr (car tree))))  (defun next-sibling (tree)  (cdr tree))  (defun data (tree)  (car (car tree)))  (defun add-child (tree child)  (setf (car tree) (append (car tree) child))  tree)
(setq tr '((1 2 (3 4 5) ((7 8) (7 8 9)))))
(setq mytree (make-tree 10))
(write (data mytree))
(terpri)
(write (first-child tr))
(terpri)
(setq newtree (add-child tr mytree))
(terpri)
(write newtree)
```

当执行代码，它返回以下结果：

```
10 (2 (3 4 5) ((7 8) (7 8 9)))

((1 2 (3 4 5) ((7 8) (7 8 9)) (10)))
```

# 20 LISP - 哈希表

哈希表的数据结构表示是基于键哈希代码进行组织键 - 值对的集合。它使用键来访问集合中的元素。哈希表是用于需要使用一键访问元素，可以找出一个有用的键值。在哈希表中每个项目都有一个键/值对。键是用于访问该集合中的项。

## 20.1 LISP 中创建哈希表

在 Common Lisp 中表是一种通用的集合。可以随心所欲的使用对象作为一个键或索引。当在一个哈希表中存储的值，设置键 - 值对，并将其存储在该键。以后可以从哈希表中使用相同的 key 检索值。每个键映射到一个单一的值，虽然可以在一键保存新值。哈希表，在 LISP，可分为三种类型，基于这样的键所不能 compared - eq, eql 或 equal。如果哈希表进行哈希处理的 LISP 对象然后将钥匙与 eq 或 eql 比较。如果在树结构中的哈希表散列，那么它会使用相等比较。

*make-hash-table 函数用于创建一个哈希表。*此函数语法的是：

```
make-hash-table &key :test :size :rehash-size :rehash-threshold
```

*那么：*

- key 参数提供了键。
- :test 参数确定键如何比较- 它应该有一个三个值 #'eq，#'eql 或 #'equal 或三个符号式之一，eq，eql，或 equal。如果未指定，则使用 eql。
- :size 参数设置哈希表的初始大小。这应该是一个大于零的整数。
- :rehash-size 参数指定用多少提高哈希表的大小时已满。这可以是一个大于零的整数，这是添加的项的数量，或者它可以是一个浮点数大于 1，这是新的尺寸，以旧的大小的比率。该参数的默认值是实现相关。
- :rehash-threshold 参数指定的哈希表如何能充分得到之前，它必须成长。这可以是一个大于零的整数，并且小于 :rehash-size(在这种情况下，每当该表是生长其将被缩小)，或者它可以是零和 1 之间的浮点数此默认值。参数是实现相关的。
- 也可以调用 make-hash-table 函数的无参数形式。

## 20.2 正在从项和新增项到哈希表

*gethash 函数通过搜索其键检索从哈希表中的项。*如果没有找到键，那么它返回 nil。

它的语法如下：

```
gethash key hash-table &optional default
```

*那么：*

- key:是相关联的键

- hash-table:是要被搜索的哈希表

- default:要返回的值，如果没有找到该入口，它是 nil，如果不是指定的值。

- gethash 函数实际上返回两个值，第二个是一个谓词值，如果发现一个项则是 true；如果被发现没有项目返回 false。

- 对于将项添加到哈希表中，可以使用 setf 函数及 gethash 函数。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(Charlie Brown)) (setf (gethash '002 empList) '(Freddie Seal))
(write (gethash '001 empList))
(terpri)

(write (gethash '002 empList))
```

当执行代码，它返回以下结果：

```
(CHARLIE BROWN) (FREDDIE SEAL)
```

## 20.3 删除条目

*remhash 函数删除在哈希表中的特定键的任何项。*如果是一个谓词，那么它为 true，如果没有有一个项则为 false。

其函数语法：

```
remhash key hash-table
```

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(Charlie Brown)) (setf (gethash '002 empList) '(Freddie Seal))
(setf (gethash '003 empList) '(Mark Mongoose))
(write (gethash '001 empList))
(terpri)
(write (gethash '002 empList))
(terpri) (write (gethash '003 empList))
(remhash '003 empList) (terpri) (write (gethash '003 empList))
```

当执行代码，它返回以下结果：

```
(CHARLIE BROWN) (FREDDIE SEAL) (MARK MONGOOSE) NIL
```

## 20.4 maphash 函数

*maphash 函数允许在每个键 – 值对应用一个指定的函数在一个哈希表。*

它有两个参数 – 函数和哈希表，并调用该函数一次为每个键/值对的哈希表中。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(Charlie Brown)) (setf (gethash '002 empList) '(Freddie Seal))
(setf (gethash '003 empList) '(Mark Mongoose))

(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) empList)
```

当执行代码，它返回以下结果：

```
3 => (MARK MONGOOSE) 2 => (FREDDIE SEAL) 1 => (CHARLIE BROWN)
```

# 21 LISP - 输入和输出

Common Lisp 提供了大量的输入输出功能。我们已经使用的格式功能，打印输出功能。在本节中，我们将探讨一些在 LISP 提供了最常用的输入输出功能。

## 21.1 输入函数

下表提供了 LISP 的最常用的输入功能：

| SL No. | 函数和说明 |
|---|---|
| 1 | read &optional input-stream eof-error-p eof-value recursive-p<br>它读取一个 Lisp 对象从输入流的打印形式，建立相应的 Lisp 对象，并返回该对象。 |
| 2 | read-preserving-whitespace &optional in-stream eof-error-p eof-value recursive-p<br>这是用在一些特殊情况下，最好是确定扩展令牌正好是字符结束。 |
| 3 | read-line &optional input-stream eof-error-p eof-value recursive-p<br>它读取一个文本行由换行符终止。 |
| 4 | read-char &optional input-stream eof-error-p eof-value recursive-p<br>这需要一个字符从输入流并将其作为一个字符的对象。 |
| 5 | unread-char character &optional input-stream<br>它把最近从输入流中读取的字符，到输入数据流的前部。 |
| 6 | peek-char &optional peek-type input-stream eof-error-p eof-value recursive-p<br>它返回的下一个字符被从输入流中读取，而无需实际从输入流中除去它。 |
| 7 | listen &optional input-stream<br>谓词监听为 true 如果有立即从输入流中的字符，如果不是则为 false。 |
| 8 | read-char-no-hang &optional input-stream eof-error-p eof-value recursive-p<br>它类似于 read-char 字符，但是如果它没有得到一个字符，它不会等待一个字符，但立即返回为 nil。 |
| 9 | clear-input &optional input-stream<br>它清除与输入流关联的所有缓冲的输入。 |
| 10 | read-from-string string &optional eof-error-p eof-value &key :start :end :preserve-whitespace<br>它采用字符串的字符，并相继建立一个 LISP 的对象，并返回该对象。它也返回第一个字符的索引无法读取字符串或字符串（或长度+1）的长度，视具体情况而定。 |
| 11 | parse-integer string &key :start :end :radix :junk-allowed<br>它会检查字符串的子串被分隔:start 和:end（默认为字符串的开头和结尾）。它会跳过空白字符，然后尝试解析一个整数。 |
| 12 | read-byte binary-input-stream &optional eof-error-p eof-value<br>它读取 1 字节的二进制输入流并将其返回一个整数的形式。 |

## 21.2 读取键盘的输入

*read 函数用于从键盘输入*。也可以不带任何参数。

例如，考虑代码片段：

```
(write ( + 15.0 (read)))
```

假设用户输入 10.2 来自 stdin 输入，它返回，

```
25.2
```

read 函数从输入流中读取字符，并通过解析为 Lisp 对象的表示解释它们。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
; the function AreaOfCircle ; calculates area of a circle
; when the radius is input from keyboard

(defun AreaOfCircle() (terpri) (princ "Enter Radius: ") (setq radius (read)) (setq area (* 3.1416
radius radius)) (princ "Area: ") (write area)) (AreaOfCircle)
```

当执行代码，它返回以下结果：

```
Enter Radius: 5 (STDIN Input) Area: 78.53999
```

示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(with-input-from-string (stream "Welcome to Tutorials Yiibai!") (print (read-char stream))
(print (read-char stream)) (print (read-char stream)) (print (read-char stream)) (print (read-
char stream)) (print (read-char stream)) (print (read-char stream)) (print (read-char stream))
```

```
(print (read-char stream))    (print (read-char stream))    (print (peek-char nil stream nil 'the-end))
  (values))
```

当执行代码，它返回以下结果：

```
#W #e #l #c #o #m #e #Space #  #o #Space
```

## 21.3 输出功能

在 LISP 所有的输出函数都有一个称为输出流可选参数，其输出传送。如果没有提及或 nil，输出流默认为变量*标准输出*的值。

下表提供了 LISP 的最常用的输出函数：

| SLNo. | 函数和说明 |
|---|---|
| 1 | write object &key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :array<br><br>write object &key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :array :readably :right-margin :miser-width :lines :pprint-dispatch<br><br>既写对象通过指定的输出流:stream，默认为标准输出*值*。其他值默认为打印设置相应的全局变量。 |
| 2 | prin1object &optional output-stream<br><br>print object &optional output-stream<br><br>pprint object &optional output-stream<br><br>princ object &optional output-stream<br><br>所有这些函数对象的打印形式输出到输出流。但是，下面的不同之处有： |

prin1 返回对象作为其值。

print 打印与前一个换行符的目标和后跟一个空格。它返回的对象。

pprint 就像印刷不同之处在于省略了结尾间隔。

princ 就像 prin1 除了输出没有转义字符。

3 write-to-string object &key :escape :radix :base :circle :pretty :level :length :case :gensym :array

write-to-stringobject &key :escape :radix :base :circle :pretty :level :length :case :gensym :array :readably :right-margin :miser-width :lines :pprint-dispatch

prin1-to-string object

princ-to-string object

该对象被有效地打印和输出的字符被转成一个字符串，并将该字符串返回。

4 write-char character &optional output-stream

它输出的字符输出流，并返回字符。

5 write-string string &optional output-stream &key :start :end

它写入字符串的指定子字符串的字符输出流。

6 write-line string &optional output-stream &key :start :end

它的工作原理与 write-string 的方式相同，但是之后输出一个换行符。

7 terpri &optional output-stream

它输出一个换行符到 output-stream。

8 fresh-line &optional output-stream

它只输出一个换行，如果流不是已经在一行的开始。

9 finish-output &optional output-stream

force-output &optional output-stream

clear-output &optional output-stream

函数 finish-output 尝试确保发送到输出流的所有输出已达到其目标，然后才返回 nil。

函数 force-output 发起的任何内部缓冲区清空，但返回 nil，而无需等待完成或确认。

函数 clear-output 尝试中止，以便使尽可能少的输出继续到目标中的任何出色的输出操作。

10 write-byte integer binary-output-stream

它写入一个字节，整数的值。

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
; this program inputs a numbers and doubles it

(defun DoubleNumber() (terpri) (princ "Enter Number : ") (setq n1 (read)) (setq doubled (* 2.0 n1))
(princ "The Number: ") (write n1) (terpri) (princ "The Number Doubled: ") (write doubled) )
(DoubleNumber)
```

当执行代码，它返回以下结果：

```
Enter Number : 3456.78 (STDIN Input) The Number: 3456.78 The Number Doubled: 6913.56
```

## 21.4 格式化输出

*format 函数是用于生产很好的格式化文本*。它的语法如下：

```
format destination control-string &rest arguments
```

*那么，*

- destination 是一个标准输出
- control-string 持有的字符要被输出和打印指令。
- format directive 由符号（~）的，用逗号，可选的冒号（:）和符号（@）修饰符和一个字符指明了哪些指令是分开的可选前缀参数。
- 前缀参数一般都是整数，记载为可选符号十进制数。

下表提供了常用的指令的简要说明：

| 指令 | 描述 |
| --- | --- |
| ~A | 后跟 ASCII 码参数 |
| ~S | 后跟 S-表达式 |
| ~D | 为十进制参数 |
| ~B | 用于二进制参数 |
| ~O | 用于八进制参数 |
| ~X | 用于十六进制参数 |
| ~C | 用于字符参数 |
| ~F | 用于固定格式的浮点参数。 |
| ~E | 指数浮点参数 |
| ~$ | 美元和浮点参数。 |
| ~% | 被打印新的一行 |
| ~* | 被忽略的下一个参数 |
| ~? | 间接。下一个参数必须是一个字符串，一个接一个列表。 |

**示例**

让我们重写程序计算圆的面积：

创建一个名为 `main.lisp` 一个新的源代码文件，并在其中输入如下代码：

```
(defun AreaOfCircle() (terpri) (princ "Enter Radius: ") (setq radius (read)) (setq area (* 3.1416
radius radius)) (format t "Radius: = ~F~% Area = ~F" radius area) ) (AreaOfCircle)
```

当执行代码，它返回以下结果：

```
Enter Radius: 10.234 (STDIN Input) Radius: = 10.234 Area = 329.03473
```

# 22 LISP - 文件 I/O

我们已经了解如何使用标准的输入和输出是由 Common Lisp 处理的参数。所有这些函数读取和写入文本文件和二进制文件。唯一不同的是在这种情况下，我们使用流不是标准输入

或输出，但对于写入或读取文件的特定目的的流创建的。在本章中，我们将看到的 LISP 如何创建，打开，关闭文本或二进制文件的数据存储。文件代表一个字节序列，如果它是一个文本文件或二进制文件。本章将引导完成重要的功能/宏的文件管理。

## 22.1 打开文件

*可以使用 open 函数来创建一个新文件或打开一个现有的文件*。这是最基本的功能为打开一个文件。然而，with-open-file 通常更方便，更常用，因为我们将在本节后面看。当一个文件被打开，一个流对象被创建来代表它在 LISP 环境。流上的所有操作基本上等同于操作上的文件。

open 函数语法是：

```
open filename &key :direction :element-type :if-exists :if-does-not-exist :external-format
```

*那么，*

- filename 参数是要打开或创建的文件的名称。
- keyword 参数指定的数据流和错误处理方式的类型。
- :direction keyword 指定的流是否应处理的输入，输出，或两者兼而有之，它采用下列值：
- :input - 用于输入流（默认值）
- :output - 输出流
- :io - 双向流
- :probe - 只是检查一个文件是否存在;该流被打开，然后关闭。
- :element-type 指定事务单元的流类型。

- :if-exists 参数指定要采取的操作，如果 :direction 是 :output or :io 和指定的名称已存在的文件。如果方向是 direction 为 :input 或 :probe，则忽略此参数。它采用下列值：

- :error - 它发出错误信号。

- :new-version - 它将创建一个具有相同名称但大版本号的新文件。

- :rename - 它重命名现有的文件。

- :rename-and-delete - 它重命名现有的文件，然后将其删除。

- :append - 它追加到现有文件。

- :supersede - 它将取代现有的文件。

- nil - 它不创建一个文件甚至流只是返回零表示失败。

- :if-does-not-exist 参数指定，如果指定名称的文件已经不存在应采取的操作。它采用下列值：

- :error - 它发出错误信号。

- :create - 它创建具有指定名称的空文件，然后使用它。

- nil - 它不创建一个文件或流，而是简单地返回 nil 表示失败。

- :external-format 参数指定用于表示文件的字符的实施认可制度

**例如**，可以打开一个名为 myfile.txt 的存储在/ tmp 文件夹的文件：

```
(open "/tmp/myfile.txt")
```

## 22.2 写入和读取文件

*with-open-file 允许读取或写入到一个文件中，用与读/写事务相关联的流变量。*一旦这项工作完成后，它会自动关闭文件。它使用极为方便。

它的语法如下：

```
with-open-file (stream filename {options}*)

      {declaration}* {form}*
```

- filename 是要打开的文件的名称;它可以是一个字符串，一个路径，或一个流。
- options 就像 keyword 参数给函数打开的一样。

**示例 1**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(with-open-file (stream "/tmp/myfile.txt" :direction :output)

  (format stream "Welcome to Tutorials Yiibai!")

  (terpri stream)

  (format stream "This is a tutorials database")

  (terpri stream)

  (format stream "Submit your Tutorials, White Papers and Articles into our Tutorials   Directory."))
```

请注意，在前面的章节，如，terpri 和 format 讨论的所有输入输出函数正在编写到创建的文件。当执行代码，它不返回任何东西;然而，数据被写入到该文件中。 :direction :output 关键字可以做到这一点。不过，我们可以使用 read-line 函数从这个文件中读取。

**实例** 2

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(let ((in (open "/tmp/myfile.txt" :if-does-not-exist nil)))

  (when in

    (loop for line = (read-line in nil)

         while line do (format t "~a~%" line))

    (close in)))
```

当执行代码，它返回以下结果：

```
Welcome to Tutorials Yiibai!

This is a tutorials database

Submit your Tutorials, White Papers and Articles into our Tutorials   Directory.
```

## 22.3 关闭文件

*close 函数关闭一个流。*

# 23 LISP – 结构

结构是用户定义的数据类型，它让用户可以合并不同种类的数据项。结构被用于表示记录。假设要跟踪图书馆中的书籍。可能希望跟踪了解每本书的以下属性：

- 标题 - Title
- 作者 - Author
- 科目 - Subject
- 书籍编号 - Book ID

## 23.1 定义一个结构

LISP 的 defstruct 宏允许定义一个抽象的记录结构。defstruct 语句定义了一个新的数据类型，项目结构中不止一个成员。讨论 defstruct 宏的格式，编写本书的结构的定义。可以定义本书的结构为：

```
(defstruct book
  title
  author
  subject
  book-id

)
```

*请注意*:

上述声明创建一个本书结构有四个命名组件。因此，创建的每一个本书将是这个结构的对象。它定义了一个名为 book-title，book-subject，book-book-id 的书籍，这将需要一个参数，书的结构，并且将返回的字段标题，作者，主题和本书的 book-book-id 对象。这些函数被称为接入功能。符号书成为一个数据类型，它*可以使用 typep 谓词检查*。也将命名为 book-p 隐函数，这是一个谓词，将为 true，如果它的参数是本、书，则返回 false。另一个名为 make-book 隐函数将被创建，这是一种构造方法，其中，当被调用时，将创建一个数据结构具有四个组件，适于与所述接入功能的使用。

- #S 语法指的是一个结构，可以用它来读取或打印一本书的实例
- copy-book 书本参数还定义了隐函数。这需要书的对象，并创建另一个书的对象，这是第一个副本。调用此函数复印机功能。
- 可以使用 setf 改变书籍的组成结构

**例如**

```
(setf (book-book-id book3) 100)
```

**示例**

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defstruct book
  title
  author
  subject
  book-id
 ) ( setq book1 (make-book :title "C Programming"                    :author "Nuha Ali"
                           :subject "C-Programming Tutorial"                 :book-id "478"))
( setq book2 (make-book :title "Telecom Billing"                  :author "Zara Ali"
                           :subject "C-Programming Tutorial"                 :book-id "501"))
(write book1)(terpri) (write book2) (setq book3( copy-book book1)) (setf (book-book-id book3) 100)

(terpri) (write book3)
```

当执行代码，它返回以下结果：

```
#S(BOOK :TITLE "C Programming" :AUTHOR "Nuha Ali" :SUBJECT "C-Programming Tutorial"
:BOOK-ID "478") #S(BOOK :TITLE "Telecom Billing" :AUTHOR "Zara Ali" :SUBJECT "C-
Programming Tutorial" :BOOK-ID "501") #S(BOOK :TITLE "C Programming" :AUTHOR "Nuha
Ali" :SUBJECT "C-Programming Tutorial" :BOOK-ID
```

# 24 LISP - 包

在编程语言的通用术语中，包是专为提供一种方法来保持一组名从另一个分开的。在一个包中声明的符号将不会与另一个声明的相同的符号相冲突。这样的包减少独立的代码模块之间的命名冲突。LISP 读取器会维护所有已发现的符号表。当它找到一个新的字符序列，它在符号表中创建一个新的符号和存储。这个表被称为一个包。

*当前包是由特殊变量*package*引用。*

有两个预定义的包在 LISP：

common-lisp - it 包含了所有已定义的函数和变量符号。

common-lisp-user - it 采用了 common-lisp 包和其他所有的包与编辑和调试工具;它简称为 cl-user

## 24.1 LISP 包函数

下表提供了用于创建，使用和操作封装最常用的功能：

| SL No | 函数和说明 |
|---|---|
| 1 | make-package package-name &key :nicknames :use<br><br>它创建并使用指定的包名返回一个新的包。 |
| 2 | in-package package-name &key :nicknames :use<br><br>使得当前的程序包。 |
| 3 | in-package name<br><br>这个宏的原因*package*设置为名为 name 的包，它必须是一个符号或字符串。 |
| 4 | find-package name<br><br>它搜索一个包。返回包的名称或昵称;如果没有这样的程序包是否存在，find-package 返回 nil |
| 5 | rename-package package new-name &optional new-nicknames<br><br>它重命名一个包。 |
| 6 | list-all-packages<br><br>该函数返回一个当前存在于 Lisp 语言系统中的所有包的列表。 |
| 7 | delete-package package<br><br>它会删除一个包 |

## 24.2 创建一个 LISP 包

*defpackage 函数用于创建一个用户定义的程序包。*它的语法如下：

```
defpackage :package-name

  (:use :common-lisp ...)   (:export :symbol1 :symbol2 ...))
```

*那么，*

- package-name 是包的名称。
- :use 关键字指定此包需要的包，即定义在此包中使用包的代码函数。
- :export 关键字指定为外部在这个包中的符号。
- make-package 函数也可用于创建一个包。其语法函数：
- make-package package-name &key :nicknames :use
- 参数和关键字具有相同的含义。

## 24.3 使用包

一旦创建了一个包，则可以使用代码在这个包中，使其成为当前包。in-package 宏使得环境中的当前程序包。

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(make-package:tom) (make-package :dick) (make-package :harry) (in-package tom) (defun hello ()
(write-line "Hello! This is Tom's Tutorials Yiibai") ) (hello) (in-package dick) (defun hello ()
(write-line "Hello! This is Dick's Tutorials Yiibai") ) (hello) (in-package harry) (defun hello ()

(write-line "Hello! This is Harry's Tutorials Yiibai") ) (hello) (in-package tom) (hello) (in-package
```

```
dick) (hello) (in-package harry) (hello)
```

当执行代码，它返回以下结果：

```
Hello! This is Tom's Tutorials Yiibai

Hello! This is Dick's Tutorials Yiibai Hello! This is Harry's Tutorials Yiibai
```

## 24.4 删除包

*delete-package 宏允许删除一个包*。下面的例子演示了这一点：

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(make-package :tom) (make-package :dick) (make-package :harry) (in-package tom) (defun hello ()
(write-line "Hello! This is Tom's Tutorials Yiibai") ) (in-package dick) (defun hello ()
(write-line "Hello! This is Dick's Tutorials Yiibai") ) (in-package harry) (defun hello ()

(write-line "Hello! This is Harry's Tutorials Yiibai") ) (in-package tom) (hello) (in-package dick)
(hello) (in-package harry) (hello) (delete-package tom) (in-package tom) (hello)
```

当执行代码，它返回以下结果：

```
Hello! This is Tom's Tutorials Yiibai
Hello! This is Dick's Tutorials Yiibai Hello! This is Harry's Tutorials Yiibai

*** - EVAL: variable TOM has no value
```

# 25 LISP - 错误处理

## 25.1 面向对象的错误处理- LISP 条件系统

在 Common Lisp 的术语中，异常被称为条件。事实上，条件比在传统编程语言的异常更为普遍，因为一个条件表示任何事件，错误与否，这可能会影响各级函数调用堆栈。在 LISP 状态处理机制，处理的条件是用来警告信号(例如通过打印一个警告)，而在调用堆栈的上层代码可以继续工作，这样的情况下以这样一种方式。

条件处理系统中 LISP 有三个部分：

1）信号的条件

2）处理条件

3）重启进程

## 25.2 处理一个条件

让我们处理由除零所产生的条件的例子，在这里解释这些概念。需要处理的条件如下步骤：

定义条件 - “条件是一个对象，它的类表示条件的一般性质，其实例数据进行有关的特殊情况，导致被示意条件的细节信息”。

定义条件的宏用于定义一个条件，它具有以下语法：

```
(define-condition condition-name (error)  ((text :initarg :text :reader text)))
```

:initargs 参数，新的条件对象与 MAKE-CONDITION 宏，它初始化的基础上，新的条件下的插槽中创建的。

在我们的例子中，下面的代码定义的条件：

```
(define-condition on-division-by-zero (error)  ((message :initarg :message :reader message)))
```

## 25.3 编写处理程序

条件处理程序是用于处理信号的条件在其上的代码。它一般写在调用该函数出问题的上级功能之一。当条件信号发生时，该信号转导机制中搜索基于所述条件的类合适的处理器。

每个处理程序包括：

1) 类型说明符，它指示条件，它可以处理的类型

2）一个函数，它接受一个参数条件

3）当条件获得信号，该信号机制发现最近建立的处理程序与条件类型兼容，并调用它的函数。

宏处理程序的情况建立了一个条件处理程序。一个处理程序的 handler-case 形式：

```
(handler-case expression
  error-clause*)
```

那么，每个 error 从句的形式为：

```
condition-type ([var]) code)
```

## 25.4 重新启动阶段

这是真正从错误的代码中恢复程序，条件处理程序可以通过调用一个适当的重启处理的条件。重启代码一般是放置在中层或底层函数和条件处理程序被放置到应用程序的上层。

*handler-bind 宏允许提供一个重启功能，并允许继续在较低级的功能，无需解除函数的调用堆栈。*换句话说，控制流将仍然处于较低水平的功能。

handler-bind 的基本形式如下：

```
(handler-bind (binding*) form*)
```

其中每个绑定如以下列表：

1）条件类型

2）一个参数的处理函数

3）invoke-restart 宏查找并调用具有指定名称作为参数最近绑定重启功能。

4）可以有多个重新启动。

## 示例

在这个例子中，我们演示了上述概念通过写一个名为划分功能函数，则会创建错误条件，如果除数参数为零。我们有三个匿名的功能，提供三种方式来出它 - 通过返回一个值 1，通过发送一个除数 2 和重新计算，或通过返回 1。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(define-condition on-division-by-zero (error) ((message :initarg :message :reader message)))
(defun handle-infinity () (restart-case (let ((result 0)) (setf result (division-function 10 0)) (format t "Value: ~a~%" result)) (just-continue () nil)))
(defun division-function (value1 value2) (restart-case (if (/= value2 0) (/ value1 value2) (error 'on-division-by-zero :message "denominator is zero"))

  (return-zero () 0)
  (return-value (r) r)
  (recalc-using (d) (division-function value1 d))))

(defun high-level-code ()
  (handler-bind
    ((on-division-by-zero
      #'(lambda (c) (format t "error signaled: ~a~%" (message c)) (invoke-restart 'return-zero)))
    (handle-infinity))))

  (handler-bind
    ((on-division-by-zero
      #'(lambda (c) (format t "error signaled: ~a~%" (message c)) (invoke-restart 'return-value 1))))
    (handle-infinity))

  (handler-bind
    ((on-division-by-zero
      #'(lambda (c) (format t "error signaled: ~a~%" (message c)) (invoke-restart 'recalc-using 2))))
    (handle-infinity))

  (handler-bind
    ((on-division-by-zero
      #'(lambda (c) (format t "error signaled: ~a~%" (message c)) (invoke-restart 'just-continue))))
    (handle-infinity))

  (format t "Done."))
```

当执行代码，它返回以下结果：

```
error signaled: denominator is zero
Value: 1 error signaled: denominator is zero
Value: 5 error signaled: denominator is zero

Done.
```

除了“系统状态”，如上文所讨论，普通的 LISP 还提供了各种功能，其可被称为信令错误。当信号实现相关处理错误。

## 25.5 LISP 的错误信号功能

下表提供了常用功能的信令警告，休息，非致命和致命的错误。

用户程序指定一个错误信息（字符串）。该函数处理这个消息，并且可能/可能不会显示给用户。错误信息应该通过应用的格式化功能进行构造，不应该在开头或结尾包含一个换行符，也无需指明错误，如 LISP 系统将根据其喜好的样式利用这些服务。

| SL No. | 函数和说明 |
| --- | --- |
| 1 | error format-string &rest args<br><br>它标志着一个致命的错误。这是不可能从这种错误的继续;这样的错误将永远不会返回到其调用者。 |
| 2 | cerror continue-format-string error-format-string &rest args<br><br>它发出错误信号，并进入调试器。但是，它允许程序从调试器解决错误之后继续。 |
| 3 | warn format-string &rest args<br><br>它打印一条错误消息，但一般不会进入调试 |
| 4 | break &optional format-string &rest args<br><br>它打印的消息，并直接进入调试器，而不允许拦截由编程错误处理设施的任何可 |

```
        能性
```

**示例**

在这个例子中，阶乘函数计算一个数阶乘;但是，如果参数为负，它抛出一个错误条件。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defun factorial (x)    (cond ((or (not (typep x 'integer)) (minusp x))
        (error "~S is a negative number." x))
       ((zerop x) 1)
       (t (* x (factorial (- x 1))))))

(write(factorial 5))
(terpri)
(write(factorial -1))
```

当执行代码，它返回以下结果：

```
120 *** - -1 is a negative number.
```

# 26 LISP - 对象系统（CLOS）

Common Lisp 通过几十年的面向对象编程的推进。但是，面向对象被并入是在它最后一阶段。

## 26.1 类的定义

*defclass 宏允许创建用户定义的类*。它建立了一个类作为数据类型。它的语法如下：

```
(DEFCLASS class-name (superclass-name*)   (slot-description*)   class-option*)
```

- 插槽是存储数据变量或字段。

- slot-description 形式(插槽名称插槽选项*)，其中每个选项是一个关键字后跟一个名字，表达式和其他选项。最常用的槽选项是：

- :accessor 函数名称

- :initform 表达式

- :initarg 符号

例如，让我们定义一个 Box 类，有三个槽的长度，广度和高度。

```
(defclass Box ()
(length
breadth

height))
```

## 26.2 提供访问和读/写控制到一个插槽

除非有插槽可以访问，读取或写入的值，类是好看不中用。

当定义一个类可以为每个插槽指定访问。例如，把我们的 Box 类：

```
(defclass Box ()  ((length :accessor length)  (breadth :accessor breadth)  (height :accessor height)))
```

也可以读取和写入一个插槽指定单独的访问器的名称。

```
(defclass Box ()  ((length :reader get-length :writer set-length)  (breadth :reader get-breadth
:writer set-breadth)  (height :reader get-height :writer set-height)))
```

## 26.3 类创建实例

*通用函数 make-instance 创建并返回一个类的新实例。*

它的语法如下：

```
(make-instance class{initarg value}*)
```

### 示例

让我们创建一个 Box 类，有三个插槽，长度，宽度和高度。我们将使用三个插槽存取到这些字段设置的值。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defclass box () ((length :accessor box-length) (breadth :accessor box-breadth) (height :accessor box-height))) (setf item (make-instance 'box))
(setf (box-length item) 10)
(setf (box-breadth item) 10)
(setf (box-height item) 5)
(format t "Length of the Box is ~d~%" (box-length item))
(format t "Breadth of the Box is ~d~%" (box-breadth item))

(format t "Height of the Box is ~d~%" (box-height item))
```

当执行代码，它返回以下结果：

```
Length of the Box is 10 Breadth of the Box is 10 Height of the Box is 5
```

## 26.4 定义一个类的方法

*defmethod 宏允许在类中定义一个方法*。下面的示例扩展 Box 类包含一个方法名为 volume。

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defclass box () ((length :accessor box-length) (breadth :accessor box-breadth) (height :accessor box-height) (volume :reader volume)))
; method calculating volume

(defmethod volume ((object box)) (* (box-length object) (box-breadth object)(box-height object)))
 ;setting the values

(setf item (make-instance 'box))
(setf (box-length item) 10)
(setf (box-breadth item) 10)
(setf (box-height item) 5)
```

```
; displaying values

(format t "Length of the Box is ~d~%" (box-length item))
(format t "Breadth of the Box is ~d~%" (box-breadth item))
(format t "Height of the Box is ~d~%" (box-height item))
(format t "Volume of the Box is ~d~%" (volume item))
```

当执行代码，它返回以下结果：

```
Length of the Box is 10 Breadth of the Box is 10 Height of the Box is 5 Volume of the Box is
500
```

## 26.5 继承

LISP 允许在另一个对象来定义一个对象。这就是所谓的继承。可以通过添加功能，新的或不同的创建派生类。派生类继承了父类的功能。

下面的例子说明了这一点：

### 示例

创建一个名为 main.lisp 一个新的源代码文件，并在其中输入如下代码：

```
(defclass box ()  ((length :accessor box-length)  (breadth :accessor box-breadth)  (height :accessor
box-height)  (volume :reader volume))) ; method calculating volume
(defmethod volume ((object box))  (* (box-length object) (box-breadth object)(box-height object)))
;wooden-box class inherits the box class
(defclass wooden-box (box) ((price :accessor box-price)))
 ;setting the values
(setf item (make-instance 'wooden-box))
(setf (box-length item) 10)
(setf (box-breadth item) 10)
(setf (box-height item) 5)
(setf (box-price item) 1000)

; displaying values

(format t "Length of the Wooden Box is ~d~%" (box-length item))
(format t "Breadth of the Wooden Box is ~d~%" (box-breadth item))
(format t "Height of the Wooden Box is ~d~%" (box-height item))
(format t "Volume of the Wooden Box is ~d~%" (volume item))
```

```
(format t "Price of the Wooden Box is ~d~%" (box-price item))
```

当执行代码，它返回以下结果：

```
Length of the Wooden Box is 10 Breadth of the Wooden Box is 10 Height of the Wooden Box is 5 Volume of the Wooden Box is 500 Price of the Wooden Box is 1000
```

# 附录：为什么我喜欢 Lisp 语言

Lisp 是一种很老的语言。非常的老。Lisp 有很多变种，但如今已没有一种语言叫 Lisp 的了。事实上，有多少 Lisp 程序员，就有多少种 Lisp。这是因为，只有当你独自一人深入荒漠，用树枝在黄沙上为自己喜欢的 Lisp 方言写解释器时，你才成为一名真正的 *Lisp 程序员*。

目前主要有两种 Lisp 语言分支：Common Lisp 和 Scheme，每一种都有无数种的语言实现。各种 Common Lisp 实现都大同小异，而各种 Scheme 实现表现各异，有些看起来非常的不同，但它们的基本规则都相同。这两种语言都非常有趣，但我却没有在实际工作中用过其中的任何一种。这两种语言中分别在不同的方面让我苦恼，在所有的 Lisp 方言中，我最喜欢的是 Clojure 语言。我不想在这个问题上做更多的讨论，这是个人喜好，说起来很麻烦。Clojure，就像其它种的 Lisp 语言一样，有一个 REPL(Read Eval Print Loop)环境，你可以在里面写代码，而且能马上得到运行结果。例如：

```
1 5
2 ;=> 5
3
4 "Hello world"
5 ;=> "Hello world"
```

通常，你会看到一个提示符，就像 user>，但在本文中，我使用的是更实用的显示风格。这篇文章中的任何 REPL 代码你都可以直接拷贝到 Try Clojure 运行。

我们可以像这样调用一个函数：

```
1 (println "Hello World")
2 ; Hello World
3 ;=> nil
```

程序打印出“Hello World”，并返回 nil。我知道，这里的括弧看起来好像放错了地方，但这是有原因的，你会发现，他跟 Java 风格的代码没有多少不同：

```
1 println("Hello World")
```

这种Clojure在执行任何操作时都要用到括弧：

```
1 (+ 1 2)
2 ;=> 3
```

在Clojure中，我们同样能使用向量(vector)：

```
1 [1 2 3 4]
2 ;=> [1 2 3 4]
```

还有符号(symbol)：

```
1 'symbol
2 ;=> symbol
```

这里要用引号(')，因为Symbol跟变量一样，如果不用引号前缀，Clojure会把它变成它的值。list数据类型也一样：

```
1 '(li st)
2 ;=> (li st)
```

以及嵌套的list：

```
1 '(l (i s) t)
2 ;=> (l (i s) t)
```

定义变量和使用变量的方法像这样：

```
1 (def hello-world "Hello world")
2 ;=> #'user/hello-world
3
4 hello-world
5 ;=> "Hello world"
```

我的讲解会很快，很多细节问题都会忽略掉，有些我讲的东西可能完全是错误的。请原谅，我尽力做到最好。在Clojure中，创建函数的方法是这样：

```
1 (fn [n] (* n 2))
```

```
2 ;=> #<user$eval1$fn__2 user$eval1$fn__2@175bc6c8>
```

这显示的又长又难看的东西是被编译后的函数被打印出的样子。不要担心，你不会经常看到它们。这是个函数，使用 fn 操作符创建，有一个参数 n。这个参数和 2 相乘，并当作结果返回。Clojure 和其它所有的 Lisp 语言一样，函数的最后一个表达式产生的值会被当作返回值返回。

如果你查看一个函数如何被调用：

```
1 (println "Hello World")
```

你会发现它的形式是，括弧，函数，参数，反括弧。或者用另一种方式描述，这是一个列表序列，序列的第一位是操作符，其余的都是参数。

让我们来调用这个函数：

```
1 ((fn [n] (* n 2)) 10)
2 ;=> 20
```

我在这里所做的是定义了一个匿名函数，并立即应用它。让我们来给这个函数起个名字：

```
1 (def twice (fn [n] (* n 2)))
2 ;=> #'user/twice
```

现在我们通过这个名字来使用它：

```
1 (twice 32)
2 ;=> 64
```

正像你看到的，函数就像其它数据一样被存放到了*变量*里。因为有些操作会反复使用，我们可以使用简化写法：

```
1 (defn twice [n] (* 2 n))
2 ;=> #'user/twice
3
4 (twice 32)
5 ;=> 64
```

我们使用 if 来给这个函数设定一个最大值：

```
1 (defn twice [n] (if (> n 50) 100 (* n 2))))
```

if 操作符有三个参数：断言，当断言是 true 时将要执行的语句，当断言是 false 时将要执行的语句。也许写成这样更容易理解：

```
1 (defn twice [n]
2   (if (> n 50)
3       100
4       (* n 2)))
```

非常基础的东西。让我们来看一下更有趣的东西。假设说你想把 Lisp 语句反着写。把操作符放到最后，像这样：

```
1 (4 5 +)
```

我们且把这种语言叫做 Psil(反着写的 Lisp...我很聪明吧)。很显然，如果你试图执行这条语句，它会报错：

```
1 (4 5 +)
2 ;=> java.lang.ClassCastException: java.lang.Integer cannot be cast to
  clojure.lang.IFn (NO_SOURCE_FILE:0)
```

Clojure 会告诉你 4 不是一个函数(函数是必须是 clojure.lang.IFn 接口的实现)。

我们可以写一个简单的函数把 Psil 转变成 Lisp：

```
1 (defn psil [exp]
2   (reverse exp))
```

当我执行它时出现了问题：

```
1 (psil (4 5 +))
2 ;=> java.lang.ClassCastException: java.lang.Integer cannot be cast to
  clojure.lang.IFn (NO_SOURCE_FILE:0)
```

很明显，我弄错了一个地方，因为在 psil 被调用之前，Clojure 会先去执行它的参数，也就是 (4 5 +)，于是报错了。我们可以显式的把这个参数转化成 list，像这样：

```
1 (psil '(4 5 +))
2 ;=> (+ 5 4)
```

这回它就没有被执行，但却反转了。要想运行它并不困难：

```
1 (eval (psil '(4 5 +)))
2 ;=> 9
```

你开始发现 Lisp 的强大之处了。事实上，Lisp 代码就是一堆层层嵌套的列表序列，你可以很容易从这些序列数据中产生可以运行的程序。

如果你还没明白，你可以在你常用的语言中试一下。在数组里放入 2 个数和一个加号，通过数组来执行这个运算。你最终得到的很可能是一个被连接的字符串，或是其它怪异的结果。这种编程方式在 Lisp 是如此的非常的常见，于是 Lisp 就提供了叫做*宏(macro)*的可重用的东西来抽象出这种功能。宏是一种函数，它接受未执行的参数，而返回的结果是可执行的 Lisp 代码。

让我们把 psil 传化成宏：

```
1 (defmacro psil [exp]
2   (reverse exp))
```

唯一不同之处是我们现在使用 defmacro 来替换 defn。这是一个非常大的改动：

```
1 (psil (4 5 +))
2 ;=> 9
```

请注意，虽然参数并不是一个有效的 Clojure 参数，但程序并没有报错。这是因为参数并没有被执行，只有当 psil 处理它时才被执行。psil 把它的参数按数据看待。如果你听说过有人说 Lisp 里代码就是数据，这就是我们现在在讨论的东西了。数据可以被编辑，产生出其它的程序。这种特征使你可以在 Lisp 语言上创建出任何你需要的新型语法语言。

在 Clojure 里有一种操作符叫做 macroexpand，它可以使一个宏跳过可执行部分，这样你就能看到是什么样的代码将会被执行：

```
1 (macroexpand '(psil (4 5 +)))
2 ;=> (+ 5 4)
```

你可以把宏看作一个在编译期运行的函数。事实上，在 Lisp 里，编译期和运行期是杂混在一起的，你的程序可以在这两种状态下来回切换。我们可以让 psil 宏变的罗嗦些，让我们看看代码是如何运行的，但首先，我要先告诉你 do 这个东西。

do 是一个很简单的操作符，它接受一批语句，依次运行它们，但这些语句是被整体当作一个表达式，

例如：

```
1 (do (println "Hello") (println "world"))
2 ; Hello
3 ; world
4 ;=> nil
```

通过使用 do，我们可以使宏返回多个表达式，我们能看到更多的东西：

```
1 (defmacro psil [exp]
2   (println "compile time")
3   `(do (println "run time")
4        ~(reverse exp)))
```

新宏会打印出“compile time”，并且返回一个 do 代码块，这个代码块打印出“run time”，并且反着运行一个表达式。这个反引号`的作用很像引号'，但它的独特之处是你可以使用~符号在其内部解除引号。如果你听不明白，不要担心，让我们来运行它一下：

```
1 (psil (4 5 +))
2 ; compile time
3 ; run time
4 ;=> 9
```

如预期的结果，编译期发生在运行期之前。如果我们使用 macroexpand，或得到更清晰的信息：

```
1 (macroexpand '(psil (4 5 +)))
2 ; compile time
3 ;=> (do (clojure.core/println "run time") (+ 5 4))
```

可以看出，编译阶段已经发生，得到的是一个将要打印出“run time”的语句，然后会执行(+ 5 4)。println 也被扩展成了它的完整形式，clojure.core/println，不过你可以忽略这个。然后代码在运行期被执行。

这个宏的输出本质上是：

```
1 (do (println "run time")
2     (+ 5 4))
```

而在宏里，它需要被写成这样：

```
1 `(do (println "run time")
2       ~(reverse exp))
```

反引号实际上是产生了一种模板形式的代码，而波浪号让其中的某些部分被执行((reverse exp))，而其余部分被保留。对于宏，其实还有更令人惊奇的东西，但现在，它已经很能变戏法了。

这种技术的力量还没有被完全展现出来。按着"为什么我喜欢 Smalltalk？"的思路，我们假设 Clojure 里没有 if 语法，只有 cond 语法。也许在这里，这并不是一个太好的例子，但这个例子很简单。

cond 功能跟其它语言里的 switch 或 case 很相似：

```
1 (cond (= x 0) "It's zero"
2       (= x 1) "It's one"
3       :else "It's something else")
```

使用 cond，我们可以直接创建出 my-if 函数：

```
1 (defn my-if [predicate if-true if-false]
2   (cond predicate if-true
3         :else if-false))
```

初看起来似乎好使：

```
1 (my-if (= 0 0) "equals" "not-equals")
2 ;=> "equals"
3 (my-if (= 0 1) "equals" "not-equals")
4 ;=> "not-equals"
```

但有一个问题。你能发现它吗？my-if 执行了它所有的参数，所以，如果我们像这样做，它就不能产生预期的结果了：

```
1 (my-if (= 0 0) (println "equals") (println "not-equals"))
2 ; equals
3 ; not-equals
4 ;=> nil
```

把 my-if 转变成宏：

```
1 (defmacro my-if [predicate if-true if-false]
```

```
2    `(cond ~predicate ~if-true
3            :else ~if-false))
```

问题解决了：

```
1 (my-if (= 0 0) (println "equals") (println "not-equals"))
2 ; equals
3 ;=> nil
```

这只是对宏的强大功能的窥豹一斑。一个非常有趣的案例是，当面向对象编程被发明出来后(Lisp 的出现先于这概念)，Lisp 程序员想使用这种技术。C 程序员不得不使用他们的编译器发明出新的语言，C++和 Object C。Lisp 程序员却创建了一堆宏，就像 defclass， defmethod 等。这全都要归功于宏。变革，在 Lisp 里，只是一种进化。

来源：http://www.vaikan.com/why-i-love-lisp/