Practical Common Lisp 个人翻译版

Huang Z

2011 年 10 月 15 日

原文連結:

PCL 个人翻译版(豆瓣)
因为 Practical Common Lisp 已经有译本出版,本文停止更新。

閱讀更多:

Toy of Clojure 个人翻译版
Real World Haskell 读书笔记
Redis 命令参考简体中文版

聯繫作者:

gmail/gtalk: huangz1990
twitter: @huangz1990
douban: i_m_huangz

1 引言

1.1 Lisp?! 啥玩意?

如果你也认为简洁优雅的代码是你的追求,那 Lisp 就是你的最佳选择。

使用 Lisp,你可以写出比其他语言更简洁的代码,同时多快好省地完成工作,改善你的睡眠和性生活。

这似乎有点耸人听闻,你仿佛看到了在三万米高空上漂浮的奶牛 —我能证明 Lisp 比其他语言更好吗?

这本书就是我的答案,当你读完整本书之后,你就会明白我的意思了。

现在,我得给你讲讲我和 Lisp 邂逅的陈年往事,回想当年,从我还是一个计算机系的菜鸟开始...

我算的上是为数众少 (注意,是众少) 的第二代 Lisp 黑客之一,主要是受我那乐于助人修电脑的程序员爹爹遗传。他当时接了一个资金达四千万的项目,规模足以和绿 8 比美,刚开始他使用 FORTRAN 编程,但是后来他发现 FORTRAN 写的程序就是一坨烂泥,跑起来就像老牛拉车,于是他想着用别的语言重写系统。

当时正是 1Q8X 年代,Lisp 正是人工智能领域的当红炸子鸡,于是我爹爹跑到一间由煤老板和高铁头头合伙开办的黑心大学 (CMU),想和里面的人探讨一下 Lisp 是否适合项目使用。CMU 的研究僧们将正在研究的项目

show 出来给我爹爹看,我爹爹一下子就被震住了,果断认为 Lisp 必将是明日之星。

他跑回公司报告老板,决定赌上我爷爷金田一侦探的名义,要求项目改用 Lisp。老板拗不过他,决定放手让他去办。

一年之后,该项目面临咔嚓的命运被逆转,爹爹和他的七个基友们成功使用 Lisp 完成了不可能的任务。而我爹爹认为,这一切都得归功于 Lisp 。

当然,我爹爹的意见有可能是错误的,况且,那都是 1980 年代的事情了—现在已经是闪亮的二十一世纪,连我的儿子都会打酱油了,更何况有 Ruby 和 Rails 齐飞,蟒蛇 (Python) 和地鼠 (Go) 共一窝,也许 Lisp 之勇已经不及当年了呢?

如果你也这么想,请继续听我细细道来。

我高中时候正在苦心研究魔兽争霸,所以没有时间学习编程。浑浑噩噩地从野鸡大学魔兽世界专业毕业之后,我进了一间网页公司,下海搞起了苦逼的 web 开发。

在那之后,我做了不少网站 (N!ke,阿滴打死和你拧都是我的客户) 并趁机和广告公司的各种前台 MM 邂逅过,她们的名字今天想起来还沥沥在目:Java、C、C++、Python、Smalltalk、Eiffel 以及 Beta。

但是当激情过后,黯然回首,我发觉自己最喜欢的还是 Lisp,也许是受我爹爹影响吧,因为他也是一个萝莉控。

但讽刺的是,我虽然喜欢 Lisp,但我对 Lisp 却一点也不理解,于是我开始借各种业余时间接近 Lisp ,我相信水滴石穿,事在人为,总有一天...

我慢慢发现 Lisp 果然名不虚传,她让我觉得前几年奋力追求 Java 简直

是荒废时日、不堪回首:我用 Lisp 写了一个足球算法,尝试计算出中国足球打入世界杯之日,虽然最后因为计算溢出而没有得到答案,但我发现用 Lisp 写程序的确比 Java 效率高得多。

还有一次类似的经历:我用 Java 写了一个小程序,给我的各种各样的盗版 mp3 自动管理 ID3 标签,而这个 Java 程序一直不好用,我试图重构这个程序,但最终因为难度太大而失败。

后来我用 Lisp 重写了一个,发现 Lisp 只需要两天就完成了我所需要的功能! 耶!

1.2 为神马要用 **Lisp**?

Lisp 到底有什么好,说来话长,这种事只有玩过 Lisp 的人才会知道。其中乐趣,只可意会,不可言传,更是不足为外人道也。

可惜但凡世人,十之八九,多是无利不起早之徒,所以在引诱你学习 Lisp 之前,我得先给你讲讲 Lisp 的好处,科普科普,以免你买了我的书之后再要求退货,手续不好办。

对于一些普通语言,比如 C、C++ 和 JAVA,好处是显而意见的,你可以用 C 或者 C++ 写底层,玩儿安桌,或者用 JAVA,然后去 IBM 找份朝九晚五的工作。

另外一些非主流语言,则很难对它们进行归类,比如 Perl 就声称自己一种玩意可以有多种玩法,花样百出,换言之,各种各样的 Play,无一不精,无一不通。

而 Ruby 作为日本除动漫和 AV 之外的第三产业,也受到了很多人的追

捧。

至于那些用 Python 编程的家伙,你看看 python.com 就明白了。

那么,话锋一转,说回来,为什么要用 Lisp? 其实,Lisp 语言比起其他烂大街的语言,有一个其他语言都没有的杀手级的特性 —装 B。

每次三五个死程聚首一堂,肯定要聊聊最近有什么新欢旧爱,NODE.JS,Erlang,Go 什么的在你眼里只不过是渣渣。

“噢,我最近玩儿 Lisp ,函数式编程语言的始祖,人工智能的巨子,你们知道吗? ” 简单一句话,霸气四泄,别说搞 JAVA 的羞愧难当了,搞 PHP 的简直不敢直视你的双眼。

不但如此,自从最近一本叫《Coder 与 Loser》的书出版之后,此风更甚,大小论坛,豆瓣微博,总有几个 FP 菜鸟弱弱的问,“怎么学 Lisp ?”,“我听说 Lisp 很牛”,“Lisp 比 Ruby 好吗” 之类的弱到爆的问题。

“Lisp 肯定比 Ruby 好,菜鸟们。Meta Programming 连和 Macro 提鞋的资格都不够。”

你不屑地打上几个字回答菜鸟们的问题,隔着冰冷的屏幕和 2000 米长的网线,你仍然能听到菜鸟们的尖叫。

这,就是 Lisp 的杀手级特性,而正是这个特性,让 Lisp 一直引领潮流,历久如新。

前推 200 年,后推 200 年,Lisp 仍然会是最 NB 的语言,没有之一,对此,你应该深信不疑。

2 洗刷刷,洗刷刷,REPL 入门指南

在这一章我会教你整装、设置 Lisp 环境,准备进入 Lisp 世界。勒紧你的腰带,我们准备往下跳了。

2.1 选择一个 Lisp 实现

Lisp 有很多实现,因为程序员都喜欢重复造车轮,而且软件一般是开源的,所以没有什么可以阻挡他们写代码的欲望。

于是各种各样的 Lisp 实现如雨后春笋般冒出来,美国牙防组 (ANSI) 见如此好的机会,赶紧跳出来制定标准,并向各大厂商发送催款通知书。

ANSI 固然可恶,但多得它标准,只要你使用的 Lisp 实现是通过牙防组验证的,那么它在其他被验证的 Lisp 实现上,会获得一样的结果。

不过俗语说得好,上有政策,下有对策,即便牙防组百加阻挠,仍然挡不住一些 Lisp 实现给自己增加一些私有特性,当然这些特性有时候是有用,有时候是没用的,而且在其他平台不一定兼容,所以你要自己斟酌使用,后果自负。

2.2 目田 (frEE) 你的心:交互式编程

很多呆呆的编程语言只能先编译再执行,比如 C、C++ 和 JAVA。

而 Lisp 不同,它是一个可以边写边玩,code and play 的编程语言,它有一个程序,称之为 REPL 环境,执行“读入 -求值 -打印”循环,你可以在 REPL

中定义变量、函数以及表达式,DEBUG,诸如此类。

使用 REPL,你可以先写一个程序,马上试试,看有没有出错,如果一切顺利,你接着写下一个,如此循环往复,这就是 Lisp 程序员写程序的方式。

反观那些使用编译型语言的程序员,它们的日常工作就是:编译 -喝咖啡 -编译 -喝咖啡,哼哼!

现在,你明白为什么一个 Lisp 程序员可以单挑十个 JAVA 程序员了吧—因为 Lisp 程序员不喝咖啡!

2.3 REPL 初探

为了试试 REPL 是否如传说中的好用,我们这就来试试它。

先喂给它一个简单的数字值:10

```
[1]> 10
10
```

Lisp 首先接收你的输入 10 ,然后计算这个值,发现它是数字值 10,然后,Lisp 返回 10 。

像 10 这样的 Lisp 对象称之为“自求值对象”,指的是当你将这个值对象传给 Lisp 的时候,它会将这个值原封不动还给你。

我们可以再试试更高难度的招式,这次,我们将表达式 (+ 2 3) 传给 Lisp:

```
[2]> (+ 2 3)
5
```

被括号包围的东西称之为表 (list),在这个例子之中,表包含符号 (symbol) +,以及数字值 2 和 3,一般来说,表的第一个元素是一个函数名,而剩余的其他元素则是该函数的参数。

在这个例子中,符号 + 执行加法计算,2 和 3 作为参数被传入到 + 函数当中,最后求出 5 。

OK,说得够多了,但是你肯定还在疑惑,为什么我们将 2 + 3,写成 (+ 2 3),这不是不符合常识么?

嘿,我的朋友,不这么写的话,怎么彰显 Lisp 的独树一格呢,我这么说的话,你明白了吗?

请时刻铭记 Lisp 的杀手级功能,我的朋友们。

2.4 哈啰 - 沃德 - 阿拉法特

除了将数字值喂给 Lisp 之外,你还可以试试字符串值:

```
[3]> "hello world!"
"hello world!"
```

字符串值也是一个自求值对象,它用双引号包裹,当 Lisp 读到用双引号包裹的值时,它就会知道是一个字符串,给这个字符串值在内存中分配一个位置,然后对其进行求值。

双引号不是字符串值的一部分,它只是用来识别 (marked) 字符串字面量 (literal) 的语法 (syntax)。而输出时带的双引号,是为了和输入保持一致。

严格的来说的话,我们的 “hello world” 其实只是一个 REPL 程序求值得出来的结果,它不是一个“好看的”输出值,如果想要格式化这个 “hello world”,让它漂漂亮亮地输出,我们还需要 FORMAT 函数。

FORMAT 如同其他语言的输出函数一样,有 300 个或以上的格式化参数,我们这里只简单介绍几样。

其中一个就是,当你将 t 作为 FORMAT 函数的第一个参数的时候,它会将输出打印到标准输出。

```
[4]> (format t "hello world")
hello world
NIL
```

这次的 hello world 比上次的好看多了,它是一个单纯的输出,没有带双引号或者其他乱七八糟的东西。

噢,慢着,不对,输出后面怎么多了个 NIL,好吧,这其实是 FORMAT 函数的返回值,在 Lisp 中,所有东西都是表达式,而所有表达式都返回一个值(不管它有没有用,你需要不需要),如果你玩过 Ruby 或者其他 FP 语言,你就明白了。

而这个 NIL,类似于 C 中的 NULL,Python 中的 None,Ruby 中的 nil,带表“无”。

2.5 函数定义

OK,一切顺利,让我们开始上主菜,写函数。

Lisp 中的函数以 DEFUN 表达式定义,像这样:

```
[5]> (defun hello-world () (format t "hello world"))
HELLO-WORLD
```

DEFUN 之后的 hello-world 是函数名。在第四章我们会详细跟你说明那些字符可以用作函数名,而那些又不可以。但在此之前,我们得先解释一下 "-"。

如果你有其他语言的编程经验,你多数会发现,在你的编程语言中,不可以使用 "-" 作为变量或者函数的名字,如果你要写一个像这里的 hello-world 一样的函数,你只能用毫无想象力的 hello_world 或者丑陋的 helloWorld 来代替 —这也是 Lisp 的杀手特性之一,赶紧找个本子把这一项记下来吧,光是这一招,就可以秒杀市面上大部分平庸编程语言,万试万灵!

至于函数名之后的双括号 (),它们用来包裹函数参数,因为我们的 hello-world 函数暂时不需要参数,所以它们目前还只是用来唱唱空城计。

剩下的部分,就是函数体 (body),它决定了函数做什么,在我们的例子中,它只是输出一个语句。

一旦你定义了一个函数之后,你就可以随你喜欢使用它,多少次都行,别客气:

```
[6]> (hello-world)
hello world
NIL
```

还有一件事要说明,有些天才程序员读了我的书,给我写信,说我打字不

小心,FORMAT 和 format 不分,但是,实际上,在 Lisp 中,字母不区分大小写,FORMAT 和 format 或者 Format 、FoRmAt 都是同义的 (这里有 26 种组合,时间关系,我就不一一列出了)。

2.6 保存

在上一节,我们定义了一个 hello-world 函数,但你会发现,当你退出 Lisp 然后再次进入,上次定义的函数就没办法用了。

如果有办法保存写过的程序,那可就爽了 —的确有这样的办法。

很简单,你只要用一个文件 (以.lisp 或者.cl) 结尾,将你的函数定义写进里面,当你使用的时候就载入进 Lisp ,而当你退出的时候,你定义的函数也不会消失,就是这样。

现在我们重写 hello-world 的“不会消失”版本。

首先,在文件夹内,新建一个 hello.lisp ,用你喜欢的编辑器,打开,输入以下内容,然后保存:

```
(DEFUN hello-world () (format t "hello world"))
```

嗯,和我们之前在 REPL 中写的函数定义一样,不同的是这次写在了文件当中。

然后,我们启动 Lisp ,接着,载入之前定义的 hello-world:

```
[1]> (load "hello.lisp")
;; Loading file hello.lisp ...
```

```
;; Loaded file hello.lisp
T
```

OK,一切顺利,LOAD 是用来载入写在文件中的 Lisp 程序的函数,它的返回值 T 表示载入成功,然后,我们就可以再来一遍 hello world 了:

```
[2]> (hello-world)
hello world
NIL
```

2.7 编译

“嘿嘿,Lisp 的确有两把刷子,可惜阿,你们写程序的时候不能喝咖啡。”

我听到一些编译型语言的程序员在奸笑了,哈哈哈,你们呐,t00 young, t00 simple,s0me times na!ve。

事实上,Lisp 既可以解释运行,也可以编译运行:用 COMPILE-FILE 函数编译程序,然后用 LOAD 函数载入,就那么简单 —GCC 和 MAKE 什么的连出场的机会都没有。

马上来试试编译 Lisp 程序:

```
[1]> (load (compile-file "hello.lisp"))
;; Compiling file /home/huangz/note/p_lisp/hello.lisp ...
;; Wrote file /home/huangz/note/p_lisp/hello.fas
0 errors, 0 warnings
```

```
;; Loading file /home/huangz/note/p_lisp/hello.fas ...
;; Loaded file /home/huangz/note/p_lisp/hello.fas
T
```

一如既往,T 表示一切 OK,我们可以转到文件夹,确认一下编译文件是否存在:

```
[huangz@mypad p_lisp]$ ls -l
total 24
-rw-r--r-- 1 huangz users  781 Oct 16 00:05 hello.fas
-rw-r--r-- 1 huangz users  147 Oct 16 00:05 hello.lib
-rw-r--r-- 1 huangz users   52 Oct 15 23:46 hello.lisp
```

似乎 hello.fas 和 hello.lib 似乎就是编译文件,切回 Lisp,试试我们的 hello-world 函数:

```
[2]> (hello-world)
hello world
NIL
```

嗯,干的漂亮。

某个蛮火的 JAVA 系的编程语言 (我可没说是 scala) 用可解释又可编译作为卖点,可事实是,这个功能几十年前就在 Lisp 实现了。

如果你周围有讨厌的家伙正在用这种语言 (我可没说是 scala) 并叫嚣它将延续 JAVA 的辉煌 (shit),赶紧把这个事实告诉他。

2.8 小结

在这章中我们简单介绍了 REPL 的使用方法,以及如何定义函数,还有怎么保存程序。当然我没有介绍所有细节,不过别担心,在后面的章节中,我们会慢慢领悟 Lisp 之道。马上要前往下一章了,跟紧我,别掉队。

3 实战:实现一个简单的数据库

很明显,要成为大牛,必先从菜鸟开始。

而要用 Lisp 写 NB 的程序,我们必须先学习 Lisp 语言本身开始。

但是,这样一来,不就和街边两块五一本的《谭叔叔教你学 C 语言》没什么两样了吗:这些书先介绍常量,然后教你定义函数,之后写对象,然后告诉你什么是异常、函数库。。。

喔,无聊透顶,Lisp 不应该走它们的老路,我们决定反其道而行之,从写实际的程序开始。

记住,这一章的目的是给你一个用 Lisp 写程序的直观感觉,而不是教你学习 Lisp 语法,所以这一章的叙述速度可能会有些快,但是不要担心,在下一章,我就会详细解释这里用到的各类函数和语法结构,如果遇到不明白的,囫囵吞枣地读过去就是了,别在意太多。

3.1 CD 和记录集 (Records)

假设我们打算利用盗版 CD 进攻庞大的音乐市场,很明显,我们需要一个刻录软件,将歌曲从 CD 上盗录下来,然后转刻到 MP3 或者盗版 CD 中去。

这个软件应该有以下三种功能(我们先实现数据记录功能,暂时不考虑刻录):

1. 将每张 CD 的数据保存到数据库中 2. 可以给每首歌打分 3. 可以选择歌曲,选中的歌曲才会被刻录 (防止月亮之上这类农民摇滚金曲混进我们销量百万的盗版 CD 当中)

很明显,我们需要一种方法,保存 CD 数据,Common Lisp(后称 clisp) 提供了一个称之为 Common Lisp Object System(CLOS) 的系统,可以用它来保存简单的数据。

不过我们的程序还很简单,不需要刚开始就祭出大杀器,我们可以先用简单的列表 (list) 结构表示我们的数据。

表结构用 LIST 函数构造,它接收参数,并返回一个表:

```
[1]> (list 1 2 3)
(1 2 3)
```

列表还有另一种玩法,称之为属性列表 (property list),或者简称 plist(又一个值得玩味的名字)。一个属性表就是列表中每个元素由符号 (symbol) 和值组成,每个符号前加: 符号,类似其他语言中的字典。

```
[2]> (list :a 1 :b 2 :c 3)
```

```
(:A 1 :B 2 :C 3)
```

上面的:a 就是数字值 1 的关键字符号 (keyword symbol),至于符号这玩意是什么,暂时把它当作名字就可以了,以后我们会再解释。

有了属性列表,我们还需要一个将列表中数据取出来的方法 —GETF 函数就是做这事的。

GETF 函数需要两个参数,第一个是一个 plist,然后是一个关键字符号,GETF 搜索 plist,找到和关键字符号对应的值,然后返回该值。

```
[3]> (GETF (LIST :a 1 :b 2 :c 3) :a)
1
[4]> (GETF (LIST :a 1 :b 2 :c 3) :c)
3
```

plist 和 GETF 的组合看上去似乎是一个哈希表 (hash table),但它最多只是一个阉割版的哈希表,Lisp 给我们提供了一个功能齐备的哈希表,但就目前来说,plist 已经能满足我们的胃口了。

OK,现在可以开始写我们的 CD 刻录程序了:我们要写一个 make-cd 函数,接收四个参数,并返回一个 plist ,作为 CD 数据。

```
[5]> (defun make-cd (title artist rating ripped)
       (list :title title :artist artist :rating rating :ripped ripped))
MAKE-CD
```

我们的老朋友 DEFUN 负责定义函数,函数名是 make-cd,带有四个形式参数 (parameter):title、artist、rating 和 ripped。

参数之后是函数的体,当四个实际参数 (argument) 被传入到函数当中时,它们先被四个形式参数变量绑定,然后传入函数体,生成并返回一个 plist。

好,我们来试试 make-cd 函数:

```
[6]> (make-cd "Roses" "Kathy Mattea" 7 t)
(:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T)
```

WOW,我们雄伟的盗版计划的第一小步已经迈出了,欢呼吧!

3.2 记录多张 CD

很明显,要攻入盗版音乐市场,只有一两张 CD 是不够的,我们需要一个更大的结构,HOLD 住更多 CD 数据,这样才能在激烈的盗版行业中立于不败之地。

最简单的办法,似乎就是用一个列表包裹多个列表,看上去不错,应该可行。而且,我们这次用 DEFVAR 宏 (macro) 定义一个全局变量 (global variable)—*db* ,这样我们找起数据来就更方便了。

变量中的星号 (*) 是 Lisp 的惯例,用来表示标示全局变量。(带 * 号变量再一次秒杀所有三流语言。)

```
[7]> (defvar *db* nil)
*DB*
```

我们可以用 PUSH 函数将一个元素推入列表中,但我们最好将程序写得复杂一点,好糊弄项目经理,争取年末加薪。

于是,我们定义一个 add-record 函数,它将一个 CD 数据推入 *db* 列表当中:

```
Break 1 [9]> (defun add-record (cd) (push cd *db*))
ADD-RECORD
```

现在你可以使用 make-cd 和 add-record 组合技,打出一击漂亮的回旋踢:

```
Break 1 [9]> (add-record (make-cd "Roses" "Kathy Mattea" 7 t))
((:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))

Break 1 [9]> (add-record (make-cd "Fly" "Dixie Chicks" 8 t))
((:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))

Break 1 [9]> (add-record (make-cd "Home" "Dixie Chicks" 9 t))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

每次 add-record 之后返回的是函数的返回值,该值是 add-record 函数体执行的最后一个表达式的值,也即是 PUSH 函数的返回值。

而 PUSH 函数返回经过它修改的变量的新值,因此,你看到的其实是数据库变量 *db* 的最新情况。

3.3 database inception

你可以查看 *db* 变量的当前值:

```
[10]> *db*
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 7 :RIPPED T))
```

不过,这个输出一点也不符合 Lisp 酷酷的审美观,所以我们决定写一个 dump-db 函数,打印出数据库数据,就像这样:

```
TITLE:    Home
ARTIST:   Dixie Chicks
RATING:   9
RIPPED:   T

TITLE:    Fly
ARTIST:   Dixie Chicks
RATING:   8
RIPPED:   T

TITLE:    Roses
ARTIST:   Kathy Mattea
RATING:   7
RIPPED:   T
```

dump-db 函数的定义如下:

```
(defun dump-db ()
  (dolist (cd *db*)
      (format t "~{~a:~10t~a~%~}~%" cd)))
```

嗯,这就是我们的 dump-db 函数,它用 DOLIST 宏遍历 *db* 变量,将 *db* 中的每条 CD 数据绑定到 cd 变量中,然后使用 FORMAT 对 cd 变量进行输出。(DOLIST 相当于 python 的 for、ruby 的 each、php 和 java 的 foreach。)

至于那个复杂得像拉丁文一样的 FORMAT 参数,其实也不是太难,我们来简单说说,具体细节留到第 18 章:

第一个参数 t ,代表 *standard-output* 流。

而第二个参数当中的波浪号 (~) 相当于 C 语言当中的百分号 (%),针对不同类型的值使用不同的输出方式。(Lisp 为什么用~ 而不用土掉渣的% 的原因,想必你也已经明白了 (笑)。)

其中 ~a 参数,说明使用的是人类可读的输出方式:

```
[19]> (format t "~a" "Dixie Chicks")
Dixie Chicks
NIL
```

另外,你不但能打印字符串,你还能打印符号:

```
[22]> (format t "~a" :title)
```

```
TITLE
NIL
```

至于 ~t ,则是用来打印制表符,腾出空格的,~10t 表示打印十个制表符,为两个参数 (第一个符号,第二个是字符串) 腾出位置:

```
[23]> (format t "~a:~10t~a" :article "Dixie Chicks")
ARTICLE:  Dixie Chicks
NIL
```

其三,当 FORMAT 遇见被 ~{ 和 ~} 符号包围的内容的时候,它强制下一个被处理的参数必须得是列表。

实际上,你可以给现在我们的 dump-db 函数的输出再多用一个 ~{ 和 ~} 包围,这样的话,就用不上 DOLIST 了,因为 FORMAT 会替我们遍历整个列表,就像这样:

"~{~{~a:~10t~a~%~}~%~}"

最后,~% 输出一个换行。

好了,综合上面所说的,我们 dump-db 的最新版本出炉了:

```
(defun dump-db ()
  (format t "~{~{~a:~10t~a~%~}~%~}" *db*))
```

我们用一行代码就做了不少事情,wow,感觉不错。

3.4 增强型用户界面

我们之前定义的 add-record 干得不错,充分满足了我们记录 CD 数据的需求,但是,还有一个小问题,就是 add-record 函数太过专业了,那些没搞过编程的销售人员都抱怨它很难用。

我们的 CD 销量极好,公司正飞速膨胀中,绝不能因为这个小问题而延误了我们成为百万富翁的进程。我们得马上开发一个简单易用的输入界面,好让他们方便地录入 CD 数据:

```
(defun prompt-read (prompt)
  (format *query-io* "~a: " prompt)
  (force-output *query-io*)
  (read-line *query-io*))
```

我们一行行分析这个函数:

第一行定义了 prompt-read 函数,并接受 prompt 作为参数。

第二行使用 FORMAT,输出指向的是常量 *query-io*,这个常量指向终端输入流;

"~a: " 打印指定的提示符 (prompt)。

第三行 force-output 用来强制清空指定的流,类似于 C 语言中的 flush 函数,这里我们要清空 *query-io*,确保在它打印出提示符之前没有在等待输入。

最后一行,read-line 函数从指定的流 (这里是 *query-io*,也就是终端) 读入一行字符,并返回一个不带换行符的字符串。

整个函数就是先打印出一个指定的提示符,然后读入一段字符串,最后再返回一段字符串,就是这样,简单得连我们的销售人员也能自如使用了。

为了将用户体验再向上提升一个档次,我们可以将原有的的 make-cd 和 prompt-read 揉合起来,写一个新的 make-cd 函数:

```
(defun prompt-for-cd ()
  (make-cd
    (prompt-read "Title")
    (prompt-read "Artist")
    (prompt-read "Rating")
    (prompt-read "Ripped [y/n]")))
```

这个函数看上去不错,但有点小问题:title 和 artist 都是接受字符串的,但是 rating 和 ripped 不是,如果单纯使用 prompt-read ,我们会将字符串错误地传到 rating 和 ripped 域。

我们得想个办法,将接受到的字符串转换成数字值 —PARSE-INTEGER 就是干这事的:

```
[1]> (PARSE-INTEGER "10")
10 ;
2
```

不过如果 PARSE-INTEGER 读入的不是字符串形式的数字值的话,它将引发一个错误 (error),我们用将可选的关键字参数:junk-allowed 的值设为 t,这样的话,当 PARSE-INTEGER 读入数字值失败的时候,它只返回 nil ,而不是引起错误。

另外,我们再使用 OR 宏,当 PARSE-INTEGER 返回 nil 的时候,我们就返回 0。

```
(or (parse-integer (prompt-read "Rating") :junk-allowed t) 0)
```

来试试这个语句:

```
[7]> (or (parse-integer "10" :junk-allowed t) 0)
10
[8]> (or (parse-integer "not-a-number" :junk-allowed t) 0)
0
```

接下来,要解决读入布尔值的问题,因为 Lisp 定义了一个 Y-OR-N-P 函数,这里我们直接使用它就行,省力又省心:

```
[3]> (y-or-n-p)
n
NIL
[4]> (y-or-n-p)
y
T
```

Y-OR-N-P 接受 y, Y, n 或 N 作为它的参数。

现在,该秀一秀我们的新 prompt-for-cd 函数了:

```
(defun prompt-for-cd ()
  (make-cd
    (prompt-read "Title")
    (prompt-read "Artist")
     (or (parse-integer (prompt-read "Rating") :junk-
allowed t) 0)
    (y-or-n-p "Ripped [y/N]:")))
```

也许你已经发现了,也许你也没有发现,Lisp 的求值顺序就是自上而下读取表达式,然后对表达式进行求值,所以像上面的 prompt-for-cd 函数的执行顺序是先 (defun ...) 然后 (make-cd ...) 再 (prompt-read ...) 、(prompt-read ...) 、(or ...),最后求值 (y-or-n-p ...),然后 make-cd 的返回值作为 prompt-for-cd 的值返回,就是这样。

好吧,最后,我们要将 prompt-for-cd 加进一个无限循环里面,方便销售人员一张又一张地添加 CD (不怕告诉你,我们的音乐库存已经可以媲美阿码逊了)。

宏 LOOP 可以重复执行一段代码,直到遇到 RETURN 为止:

```
(defun add-cds ()
  (loop (add-record (prompt-for-cd))
    (if (not (y-or-n-p "Another? [y/N]: ")) (return))))
```

当第三行代码中的 y-or-n-p 执行,而用户输入 y 的时候,add-cds 函数的无限循环就会退出。

噢噢,我已经迫不及待地要试试我们新的 add-cds 函数了:

```
[1]> (add-cds)
Title: Rockin' the Suburbs
Artist: Ben Folds
Rating: 6
Ripped [y/N]:  (y/n) y
Another? [y/N]:  (y/n) y
Title: Give Us a Break
Artist: Limpopo
Rating: 10
Ripped [y/N]:  (y/n) y
Another? [y/N]:  (y/n) y
Title: Lyle Lovett
Artist: Lyle Lovett
Rating: 9
Ripped [y/N]:  (y/n) y
Another? [y/N]:  (y/n) n
NIL
```

打印出输入的数据试试:

```
[3]> (dump-db)
TITLE:    Lyle Lovett
ARTIST:   Lyle Lovett
RATING:   9
RIPPED:   T
```

```
TITLE:    Give Us a Break
ARTIST:   Limpopo
RATING:   10
RIPPED:   T

TITLE:    Rockin' the Suburbs
ARTIST:   Ben Folds
RATING:   6
RIPPED:   T

NIL
```

注:如果不记得代码了,以下是完整的 add-cds 代码段。:)

```
(defun make-cd (title artist rating ripped)
   (list :title title :artist artist :rating rating :ripped ripped))

(defvar *db* nil)

(defun add-record (cd)
    (push cd *db*))

(defun dump-db ()
    (format t "~{~{~a:~10t~a~%~}~%~}" *db*))

(defun prompt-read (prompt)
```

```
    (format *query-io* "~a: " prompt)
    (force-output *query-io*)
    (read-line *query-io*))

(defun prompt-for-cd ()
    (make-cd
        (prompt-read "Title")
        (prompt-read "Artist")
       (or (parse-integer (prompt-read "Rating") :junk-
allowed t) 0)
        (y-or-n-p "Ripped [y/N]: ")))

(defun add-cds ()
    (loop (add-record (prompt-for-cd))
     (if (not (y-or-n-p "Another? [y/N]: ")) (return))))
```

3.5 保存并载入数据库

好不容易,你刚解决了一个技术难题,没多久,公司里多愁善感的销售 MM 又过来找你麻烦,她反映说每次退出 Lisp,保存在 Lisp 里的数据就会丢失。

的确是这样,因为你还没有写将数据保存到文件里的函数,目前的数据库数据都只保存在 Lisp 的 REPL 环境里 (也即是,内存当中),一旦退出 Lisp,数据库的数据就不翼而飞了。

你告诉销售 MM 说你已经到了下班时间了,这个问题只好等到明天再解决。

谁知道销售 MM 一把抱住你的胳膊,说到:

“码农哥哥,这肯定是最后一个功能了,麻烦你,帮人家实现了嘛,谢谢你了阿,你最好人了阿。”

销售 MM 的声音简直甜美得深入骨髓,绕梁 360 度有余,面对这无法抗拒的诱惑,你一咬牙,一跺脚,又坐回小隔间,开始写代码。

你镇静了一下神经,思考 save-db 函数的实现:思路很简单,读出 *db* 变量里的数据,然后写入到文件里就可以。

```
(defun save-db (filename)
    (with-open-file (out filename
                         :direction :output
                         :if-exists :supersede)
                        (with-standard-io-syntax
                            (print *db* out))))
```

WITH-OPEN-FILE 宏打开指定文件,执行相关操作,并负责将文件关闭(无论函数体内是否发生错误)。

跟在 WITH-OPEN-FILE 之后的列表不是函数调用,而是 WITH-OPEN-FILE 语法定义的一部分,它包含了在函数体之内持有文件流的变量 (这里是 out 变量),还有一个文件名 (这里是 filename 变量),以及其他控制文件如何打开的选项::direction :output 指定你以读 (write) 模式打开文件,:if-exists :supersede 则指定当文件已存在时,将会以新文件覆写旧文件。

打开文件之后,函数唯一要做的就是执行 (print *db* out),输出 (print) 数据库中的所有内容到变量 out。

和 FORMAT 不同,使用 PRINT 打印出的 Lisp 对象,可以被 Lisp 自身读入并识别。宏 WITH-STANDARD-IO-SYNTAX 保证与 PRINT 函数相关的变量被设为标准值。等会你还会使用相同的宏,处理从文件读入 Lisp 对象的任务。

save-db 接受一个字符串值作为参数,指定数据库文件的文件夹以及文件名。

具体差异视操作系统的不同而不同,比如以下代码,就会将数据库文件保存到/home/<your_user_name>/my-cds.db :

```
CL-USER> (save-db "~/my-cds.db")
((:TITLE "Lyle Lovett" :ARTIST "Lyle Lovett" :RATING 9 :RIPPED T)
 (:TITLE "Give Us a Break" :ARTIST "Limpopo" :RATING 10 :RIPPED T)
 (:TITLE "Rockin' the Suburbs" :ARTIST "Ben Folds" :RATING 6 :RIPPED
   T)
 (:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T)
 (:TITLE "Roses" :ARTIST "Kathy Mattea" :RATING 9 :RIPPED T))
```

在 windows 下,文件名则可能是 “c:/my-cds.db” or “c:

my-cds.db.” 。

至于从文件取回数据的函数,几乎就是 save-db 的反函数而已:

```
(defun load-db (filename)
  (with-open-file (in filename)
    (with-standard-io-syntax
      (setf *db* (read in)))))
```

当从文件里读出数据时,如果不指定:direction 选项,那么:input 默认值就会被使用。至于 READ 函数,则负责从输入流变量 in 中读出数据。

这种用 with-open-file 写出、读入的方法,是 REPL 用来载入文件时所使用的方式,它适用于所有 Lisp 表达式。

不过目前来说,你还只是读入和保存表达式,而不是对表达式进行求值。(隐隐约约,你似乎觉得这种读入、写出、求值表达式的能力背后有无限潜能,的确是这样的。)

SETF 宏是 common lisp 的主要赋值操作之一,它将第二个参数的值设为第一个参数的值,类似于命令式语言的 = 。

因此,在 load-db 当中,从文件中读出的数据就是 *db* 对象的值。

注意,如果你的 *db* 变量里面已经有值 (也即是,不是 nil),而又使用 SETF 给 *db* 赋值的话,*db* 原来的值将会被 SETF 设置的新值覆盖,因此,请小心使用 load-db 。

你将代码发到了公司内部网,起身寻找销售 MM,后者早已经没了踪影。你默默拿起公文包,汇入下班人群里,悄然消失在夜色当中。

3.6 查询数据库

随着数据库内的 CD 记录数量以每天 8% 的速度增长,在数据库内手工查找特定的 CD 已经变得不可能了。我们需要一个根据特定条件,查找所有符合条件的 CD 记录的函数,比如像下面的语句,应该能返回所有演唱者为 Dixie Chicks 的唱片:

```
(select :artist "Dixie Chicks")
```

因为我们的数据库是以列表的形式组织的,而针对列表,有一个 REMOVE-IF-NOT 函数,它接受两个参数,第一个参数是一个条件 (predicate) 函数,第二个参数是一个列表,然后返回列表中所有符合条件函数条件的值。

传入到 REMOVE-IF-NOT 中的条件函数,需要接受一个值,然后根据这个值,决定返回正 (true) 还是反 (false),其中 nil 为反,其他所有值都为正。

另外,REMOVE-IF-NOT 也不修改传入的列表,它只是返回一个包含所有符合条件的值的新列表而已。

说明的够详细了,该试试 REMOVE-IF-NOT 了,下面的表达式就会选出列表中的所有偶数:

```
[1]> (remove-if-not #'evenp '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

在上面的例子中,条件函数是 EVENP,当它接受一个偶数值时返回正,否则返回 nil :

```
[2]> (evenp 10)
T
[3]> (evenp 1)
NIL
```

#' evenp | 表示 “让 Lisp 找到 EVENP 函数”。可能出乎你意料之外,用 #' 前缀标识 EVENP 函数居然不是为了别具一格!

假设,我们将 #' 去掉的话,REMOVE-IF-NOT 参数就会读入 EVENP ,而它会认为 EVENP 是一个变量,于是它就会去寻找和 EVENP 变量绑定的值,而不是 EVENP 函数,因此,我们要用 #' EVENP(寻找 EVENP 函数) 和 EVENP(寻找 EVENP 变量的值) 区别开。

还记得吗? 我们用 DEFUN 定义函数,用 DEFVAR 定义变量,它们是不同的。

另外,你还可以给 REMOVE-IF-NOT 传入一个匿名函数,比如说,下面的程序也可以完成选出所有偶数值的工作:

```
[4]> (remove-if-not #'(lambda (x) (= 0 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(2 4 6 8 10)
```

匿名函数:

```
(lambda (x) (= 0 (mod x 2)))
```

先对某个值 x 取 2 的模,然后用所得的值用 = 函数于 0 对比,检查这个值是否为偶数。

如果你想提取所有的基数,可以用下面的程序:

```
[5]> (remove-if-not #'(lambda (x) (= 1 (mod x 2))) '(1 2 3 4 5 6 7 8 9 10))
(1 3 5 7 9)
```

注意,LAMBDA 看上去和 DEFUN 有某种亲戚关系,它们的区别在于 LAMBDA 创建的函数不和任何名字绑定,这些匿名函数的使用是一次性的;而 DEFUN 创造的函数则和某个函数名绑定在一起,可以通过调用函数明多次使用。

对于那些追求哥不在江湖,但江湖中却有哥的传说的那些函数来说,LAMBDA 就是它们最好的归宿。

说回来我们的 CD 程序。

要从数据库中提取出所有 Dixie Chicks 的唱片,我们得遍历整个数据库的所有列表,然后用 REMOVE-IF-NOT 对所有记录进行过滤,而 REMOVE-IF-NOT 的条件函数,则是由 GETF 和 EQUAL 组成。

其中,GETF 负责提取属性列表中的特定字段 (比如:artist),而 EQUAL 则负责将提取出来的唱片 artist 字符串和所寻找的 artist 字符串 (这里是 “Dixie Chicks”) 进行对比,看看该唱片是否符合我们的要求,如果不是,就继续对比下一张,直到所有唱片都过滤完毕为止。

完整的程序如下:

```
(defun select-by-artist (artist)
    (remove-if-not
       #'(lambda (cd) (equal (getf cd :artist) artist))
```

```
        *db*))
```

很明显,除了 select-by-artist 之外,你还需要其他选择函数,对记录的不同数据域进行查找,比如 select-by-title 、select-by-rating ,诸如此类。

它们的定义如下:

```
(defun select-by-title (title)
    (remove-if-not
        #'(lambda (cd) (equal (getf cd :title) title))
        *db*))

(defun select-by-rating (rating)
    (remove-if-not
      #'(lambda (cd) (equal (getf cd :rating) rating))))
```

诸如此类。

但是,等一下,你发现我们的三个 select 函数非常相似,它们共享同一个结构

```
(remove-if-not
    #'(lambda (cd) (equal (getf cd :某个指定数据域) 期望的数据域值)))
```

马丁 flower 叔叔告诉我们重复代码是万恶之源,我们决定重构一下 select 类函数,泛化出一个新的 select 函数:

```
(defun select (selector-function)
    (remove-if-not selector-function *db*))
```

这样的话,我们就可以将任何我们需要的选择函数以参数的方式传递给 select 函数,以获得对不同数据域进行过滤的能力。

我们用新的 select 函数,执行之前的 select-by-title 的工作,查找 artist 为"Dixie Chicks" 的所有唱片:

```
(select #'(lambda (cd) (equal (getf cd :artist) "Dixie Chicks")))
```

不过这个新函数看上去也不太漂亮,因为每次都要传入各种各样相似的选择函数,而我们决定要将函数抽象到外太空去,因此,我们再对选择器进行包装:

```
(defun artist-selector (artist)
    #'(lambda (cd) (equal (getf cd :artist) artist)))
```

下面的程序使用了新的 select 函数和 artist-selector 函数,它的作用和之前的 select-by-title 一样,但是好看多了。

```
[5]> (select (artist-selector "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 9 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 8 :RIPPED T))
```

但是,再一次,我们还是得写各种各样相似但是作用不同的选择函数,比如 artist-selector、title-selector 等等,就像我们才刚跳出一个坑,刚准备放鞭炮庆祝,却发现外面是一个更大的坑,这有点棘手,嗯。

我们需要一种个更抽象的函数,无论数据里面有什么数据域 (因为数据库的数据总是变来变去的),我们都可以通过指定” 数据域 -值” 的方式,来筛选数据库中的数据。

让我们看看其他语言怎么做,比如说 SQL ,它是一种比 Lisp 更神秘而难懂的上古神谕,普通人很难从 SQL 中读出什么,上面记载的秘密只有一种叫 DBA 的生物才能破译。

如果用 SQL 改写我们的 select-by-title 函数的话,可能会是这样:

```
SELECT title FROM *db* WHERE title = "Home"
```

至于 select-by-artist 函数,则可能会是这样:

```
SELECT artist FROM *db* WHERE artist = "Dixie Chicks"
```

嗯,那个 WHERE field = value 的结构似乎不错,我们可以尝试微创新一下,这样的话,我们的 select 语句就会精简成这样:

```
(select (where :artist "Dixie Chicks"))
```

不但如此,我们还想抄袭 SQL 语言的组合筛选功能 (” 诚恳地复制” 是我们的座右铭):

SELECT title, artist FROM *db* WHERE title = “Home”, artist = “Dixie Chicks”

因此,我们的 select 语句也必须要支持多条件筛选:

```
(select (where :title "Home" :artist "Dixie Chicks"))
```

或者

```
(select (where :rating 10 :ripped nil))
```

诸如此类。

看上去行,但是要怎么实现这个 where 函数呢? 怎么让一个函数可以接受不固定数目的参数,而且可以接受像是“数据域 -值” 那样成双成对的过滤条件呢?

3.7 关键字参数

目前为止,我们定义的所有函数,每次调用都必须传给函数适当数量的参数,不多不少,而且,每个实际参数都是按顺序和函数的形式参数绑定的。

比如函数:

```
(defun foo (a b c) (list a b c))
```

如果我们调用

```
(foo x y z)
```

那么变量 x 的值和 a 绑定,变量 y 的值和 b 绑定,变量 z 的值和 c 绑定。而且,我们一定要给足三个参数,否则,Lisp 就会向你抱怨。

我们想要一种办法,可以使用可变数量的参数,而且,可以指定实际参数和那个形式参数绑定,不一定按照函数定义的顺序来绑定变量。

关键字参数 (Keyword parameters) 就是来干这事的,比如,下面就是关键字参数版本的 foo 函数:

```
(defun foo (&key a b c) (list a b c))
```

唯一的区别就是函数参数栏多了一个 &key ,但从现在开始,就可以随意决定传给 foo 函数的参数数量了:

```
(foo :a 1 :b 2 :c 3)  ==> (1 2 3)
(foo :c 3 :b 2 :a 1)  ==> (1 2 3)
(foo :a 1 :c 3)       ==> (1 NIL 3)
(foo)                 ==> (NIL NIL NIL)
```

正如上边的代码所示,你可以按关键字来对 foo 的参数进行赋值了,关键字赋值不必遵循函数定义时形式参数的先后顺序,而没有赋值的参数,值为 nil 。

关键字参数有两个微妙的小问题。

首先,对于没有赋值的关键字参数,参数被默认赋值为 nil ,但是,你也可以更改默认值,只要用一个列表包围参数名和默认值就可以了:

```
[6]> (defun foo (&key a (b 20) (c 30)) (list a b c))
FOO
[7]> (foo)
(NIL 20 30)
```

上面的代码示例就给 b 和 c 设置了默认值 (分别为 20 和 30),而可怜的 a 没有设置参数默认值,所以还是被默认赋值为 nil 。

另一个问题是,有时候我们想给某个参数赋值,有时候又希望这个参数留空,我们需要一种方法,识别一个参数是否被赋值。

这一次,我们将另一个变量名放进包围参数的列表当中,取决于参数是否赋值,这个变量会返回 t 或者 nil:

```
(defun foo (&key a (b 20) (c 30 c-p)) (list a b c c-p))
```

在上面的函数定义中,语句

```
(c 30 c-p)
```

表示如果 c 有参数手动传入的话 (默认值不算),c-p 变量就的值就为 T,否则,c-p 的值为 nil :

```
(foo :a 1 :b 2 :c 3)  ==> (1 2 3 T)
(foo :c 3 :b 2 :a 1)  ==> (1 2 3 T)
(foo :a 1 :c 3)       ==> (1 20 3 T)
(foo)                 ==> (NIL 20 30 NIL)
```

3.8 欢迎实现 WHERE 函数

我们绕得有点远,也许你已经把 WHERE 函数忘了,我们重新定义一下 WHERE 函数:WHERE 函数接受不定个参数,根据给定参数的值的条件,过滤 CD 数据库。

比如下面的语句,就会选择所有 artist 为 “Dixie Chicks” 的唱片:

```
(select (where :artist "Dixie Chicks"))
```

而要选择所有 rating 为 10 ,并且 ripped 为 nil 的唱片,可以用这个表达式:

```
(select (where :rating 10 :ripped nil))
```

嗯,学会了关键字参数之后,我们已经有能力实现 where 函数了:

```
(defun where (&key title artist rating (ripped nil ripped-
p))
  #'(lambda (cd)
    (and
       (if title    (equal (getf cd :title)  title)  t)
       (if artist   (equal (getf cd :artist) artist) t)
       (if rating   (equal (getf cd :rating) rating) t)
     (if ripped-p (equal (getf cd :ripped) ripped) t))))
```

WHERE 函数返回一个匿名函数,匿名函数里面包含了要对 CD 进去选择的条件。

其中,我们使用了 if 语句,来确保只有当某个参数不为 nil 的时候,才对相应的数据域进行检查,否则,它只是简单地返回 t ,略过检查。

比如如果 title 参数为 nil ,我们就不会 cd 变量的 title 数据域进行检查。

然后,我们用 equal 函数,检查 getf 对比的结果是否为真。(其中 t 代表 Lisp 中的真 (ture)。)

最后一行代码比较特别,因为 cd 变量的 ripped 数据域可能为 nil ,而 where 函数的 ripped 参数默认也为 nil,如果这里不仔细处理的话,无论是否是我们的原意,where 函数都会检查 ripped 数据域。

所以,为避免 ripped 数据域躺着中枪,我们用 ripped-p 检查 where 函数的 ripped 参数是否被手动赋值,决定是否检查 cd 变量的 ripped 数据域。

至于外层的 AND ,它提供了对多选择条件的支持。CD 唱片必须符合所有给定的条件,才会通过测试。

3.9 剑走偏锋——使用 WHERE 更新数据库记录

通过前一节,噢,不,前两节的叙述,我们有了两个漂亮的泛化函数,一个 WHERE 和一个 select 。

我们还想增加 update 函数,用于更新数据库中的记录。

update 函数和 select 很相似,因为要更新一项记录,我们得先用类似 select 的方法找到符合要求的记录,然后对记录进行修改。

以下是 update 函数的定义,我们一行行地分析它的作用:

```
(defun update (selector-function &key title artist rating (ripped nil ripped-p))
    (setf *db
        (mapcar
            #'(lambda (row)
                (when (funcall selector-function row)
                    (if title (setf (getf row :title) title))
```

```
            (if artist (setf (getf row :artist) artist))
            (if rating (setf (getf row :rating) rating))
            (if ripped-p (setf (getf row :ripped) ripped)))
            row)
      *db*)))
```

首先,update 函数接受四个参数:

第一个参数 selector-function 是一个选择函数,用来选择指定的记录。

然后是关键字参数 title artist rating 和 ripped ,这些上两节都已经说过,这里不再深究。

第二行,我们对 *db* 变量使用了 setf ,将它的值设置为后面一大堆表达式计算所得的值。

第三行是 update 函数的核心,它使用 mapcar 遍历原有的 *db* 变量,并根据传入的函数 (这里它是一个长长的匿名函数),对 *db* 变量进行修改,并最终,返回一个列表,而这个列表包含了所有已经被修改过的记录。

然后,第二行的用 setf 将第三行所得的新数据设置为 *db* 的新值。

第四行中的匿名函数,它接受 row 变量作为 MAPCAR 函数所遍历的数据记录,并使用 selector-function 作为选择函数,判断该数据记录是否符合条件,如果是的话,就用之后的一串 if 对数据域进行修改。

修改数据域的过程和 where 函数的函数体类似,不同的是这里使用 setf 对数据域进行修改,而不是用 equal 进行相等测试。

嗯,总的来说,update 函数接受选择函数和需要修改的域的值作为参数,然后遍历 *db* 对象,对符合选择函数条件的数据记录进行修改,最后,将遍历所得的新列表设置为 *db* 的新值 (还记得嘛,setf 设置新值会覆盖所有

的旧值)。

另外,要解释的一点是,setf 和 getf 的作用差别相去甚远,它们有类似的名字纯属巧合。

OK,来试试我们的 update 函数, 这里,我将所有 artist 为 “Dixie Chicks” 的唱片的 rating 更新为 11:

```
CL-USER> (update (where :artist "Dixie Chicks") :rating 11)
NIL

CL-USER> (select (where :artist "Dixie Chicks"))
((:TITLE "Home" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T)
 (:TITLE "Fly" :ARTIST "Dixie Chicks" :RATING 11 :RIPPED T))
```

3.10 DELETE IT!

我们的数据库已经有了 CRUD(Create, Read, Update, Delete) 中的前三个了,很明显,我们还需要一个 delete 函数。

思路很清晰——对 *db* 变量进行遍历,过滤所有不符合给定选择条件的记录,然后剩余的记录 (它们不在删除条件之内) 设置为 *db* 变量的新值即可。

我们的老朋友 remove-if 以及 setf 很乐意为我们效劳:

```
(defun delete-rows (selector-fn)
  (setf *db* (remove-if selector-fn *db*)))
```

大功告成！我们实现了一个功能完整的数据库实现，依靠这个 NB 的数据库，我们成为盗版 CD 领头羊的目标指日可待也！

3.11 Dont't Repeat Yourself(DRY)

我们已经有了一个功能完整的数据库，但是还有一个小问题，它的效率不高，看看 where 函数，它的函数体里都有类似于这样的语句：

```
(if title ...)
(if artist ...)
(if rating ...)
(if ripped ...)
```

不管我们怎么使用 where 函数，where 函数总是要一个个判断我们传入的参数，然后判断是否使用相应的检查语句。

也即是，如果我们使用 (where :artist "huangz") 作为查询条件，where 函数就会对数据库内的每张 CD 进行如下判断：

(if title ...) 失败，掠过；(if artist ...) 正确，进行选择；(if rating ...) 失败，略过；(if ripped ...) 失败，掠过。

我们的程序多做了很多无用功，这种资源浪费对于小规模的程序来说是无关紧要的，但是对我们阿马逊级别的 CD 数据库来说却是巨大的浪费。

我们想要一些更高效的选择函数，但是，这样，我们又绕回了上一小节关于 selector-function 函数的讨论 —如果我们想获得最高的效率，我们必须写一个又一个相似但是作用不同的选择函数。

自己手写这种选择函数已经被证明是不可能的了,因为数据域经常改变,而且数量庞大,重复代码避无可避。

一种荒诞而疯狂的想法在你脑海中浮现出来:有没有一种办法,根据查询条件自动生成选择函数? 而且,生成的选择函数不多也不少,拥有最高的效率?

嗯... 你陷入了沉思当中,忽然,你耳边传来一声 whisper from the god,它悄然地说到:“Macro”。

3.12 宏,等价交互以及炼金术

说起宏 (Macro),很多人立即会想起 C 语言中的 #define、#if 等宏,而实际上,Lisp 的宏和 C 语言的宏除了名字相同之外,没有任何其他相似之处。

在 C 语言中,宏是一种文本处理机制,C 预处理器在编译之前对 C 程序的宏进行文本的替换和修改,仅次而已。

而 Lisp 的宏的用途则广泛和强大得多,它是一种代码生成机制 —当然不是无中生有,因为这不符合炼金术的等价交换原则 —实际上,你需要写一些 Lisp 代码,来生成另外一些 Lisp 代码,也即是,编写可以写程序的程序,而这,就是 Lisp 的宏。

来看一个简单的宏例子,backwards 宏将一个 Lisp 表达式倒转:

```
(defmacro backwards (expr) (reverse expr))
```

我们试试将一个 Lisp 表达式传给 backwards 宏:

```
[2]> (backwards ("hello, word" t format))
hello, word
NIL
```

很明显,这个宏将 (”hello, word” t format) 表达式转换成 (format t ”hello, word”) 然后运行了,可它是怎么做到的呢?

首先,第一个问题是,(”hello, word” t format) 是一个不合法的 Lisp 表达式,以我们对函数的想法来看,函数的参数在传入函数之前,必须先被求值,比如在表达式:

```
(+ (- 1 1) 2)
```

当中,+ 函数的第一个参数不是一个直接值,而是一个表达式,所以,必须先求值子表达式:

```
(- 1 1)
```

得到结果值 0 ,之后再传给 + 函数,得出表达式 (+ 0 2),最后,计算出结果 2 。

但是,在这里,(”hello, word” t format) 并不是一个合法的 Lisp 表达式,如果宏的参数和函数一样要先求值再传入的话,那么解释就会出错,这也就说明了,宏对于传入的参数,是不进行求值的。

因此,当执行 (backwards (”hello, word” t format)) 时,(”hello, word” t format) 不会被求值,而是直接传入 backwards 宏,然后,宏内部将 (”hello, word” t format) 传入 reverse 函数,这时候表达式就变成了 (format t “hello,

world") ,再之后 backwards 将所得的表达式作为宏的计算结果返回,接着 Lisp 对得到的这个最终表达式进行求值,最终,输出 "hello, word"。

另外,对宏的解释可以在编译时进行,因此,在运行 Lisp 程序的时候,你不会因为解释宏而损失任何效率。