第三章笔记

2019年9月1日 星期六 下午7:08

One powerful design strategy, which is particularly appropriate to the construction of programs for modeling physical systems, is to base the structure of programs on the structure of the system being modeled. For each object in the system, we construct a corresponding computational object. For each system action, we define a symbolic operation in our computational model. Our hope in using this strategy is that extending the model to accommodate new objects or new actions will require no strategic changes to the program, only the addition of the new symbolic analogs of those objects or actions. If we have been successfully in our system organization, then to add a new feature or debug an old one we will have to work on only a localized part of the system.

The difficulties of dealing with objects, change, and identity are a fundamental consequence of the need of grapple with time in our computational models. These difficulties become even greater when we allow the possibility of the concurrent execution of programs.

An object is said to "have state" if its behavior is influenced by its history.

Encapsulation reflects the general system-design principle known as the hiding principle: hiding principle: one can make a system more modular and robust by protecting parts of the system from each other; that is, by providing information access only to those parts of the system that have a "need to know".

Soon as we introduce assignment into our language, substition is no longer an adequete model of procedure application.

We have already used begin implicity in our programs, because in Scheme the body of a procedure can be a sequence of expressions. Also, the <consequent> part of each clause in a cond expression can be a sequence of expressions rather than a single expression.

Two calls to the same procedure with the same arguments always produced the same result.

Actually, this is not quite true. One exception is random number generator. Another exception involved the operation /type tables we introduced in section 2.4.3, where the values of two calls to get the same arguments depended on intervening calls to put. On the other hand, until we introduce assignment, we have no way to create such procedures ourselves.

Set!操作使我们可以去模拟带有局部状态的对象。然而，这一获益也有一个代价，它使我们的程序设计语言不能再用代换模型解释了。进一步说，热闹具有“漂亮”数学性质的简单模型，都不可能继续适合作为处理程序设计语言的对象和赋值的框架了。

只要我们不适用赋值，以同样参数对同一过程的两次求值一定产生出同样的结果，因此就可以认为是在计算数学函数。向我们在本书的前两章斤斤计的一样，不用任何赋值的程序设计成为函数式程序设计。

这里的麻烦在于，从本质上来讲，代码的最终基础就是，这一语言里的符号不过是作为值的名字。而一旦引进了set!和变量的值可以变化的想法，一个变量就不再是一个简单的名字了。现在的一个变量索引着一个可以保存值的位置，而存储在哪里的值也是可以改变的。

如果一个语言支持在表达式里“同一的东西可以相互替换”的观念，这样替换不会改变有关表达式的值，这个语言就称为具有引用**透明性**。在我们的计算机语言里包含了set!之后，也就打破了引用透明性，就使确定能否通过等价的表达式代换去简化表达式变成了一个一场错中复杂的问题了。由于这种情况，对使用赋值的程序做推理也将变得极其苦难。

有了前面有关“同一”和“变化”的论述，如果peter和Paul只能检查他们的银行账户，而不能执行修改余额的操作，那么看清这两个账户是否不同的问题就需要仔细讨论了，一般而言，如果我们绝不修改数据对象，那么就可以将一个符合数据对象完全看作是由其上段组成的一个整体。例如，一个有理数是完全由它的分子和分母确定的。如果出现了修改，这以观点也就不再合法了，此时符合数据对象有了一个“标识”，而它又是与组成这一对象的各片段都不同的东西。及是我们通过提款修改了一个账户的余额，这个账户依然是“同一个”账户，于此相反，我们也可能有两个银行账户，它们具有相同的状态信息。这种复杂性是将应工行账户看作一个对象而产生的结果，而不是程序语言的问题。例如，通常我们不将一个有理数看作具有标识的可修改对象，并不像修改其分子并保持“同一个”有理数。

将一个复合过程应用于一些实际参数，就是在用各个实际参数代换过程体力对应的形式参数之后，求值这个形式体

一旦我们把赋值引进程序设计语言之后，这一定义就不合适了。由于赋值的存在，变量已经不能再看作仅仅是某个值的名字。此时的一个变量必须以某种方式指定了一个“位置”，相应的值可以存储在那里。在我们的新求值模型里，这种位置将维持在称为环境的结构中。

一个环境就是框架(frame)的一个序列，每个框架是包含一些约束的一个表格（可能为空），这些约束将一些变量关联于对应的值（在一个框架里，任何变量至多只能有一个约束）。每个框架还包含着一个指针，指向这一框架的外部环境。如果由于当前讨论的目的，将相应的框架看作是全局的，那么它将没有外围环境。一个变量相对于某个特定环境的值，也就是这一环境中，包含着该变量的第一个框架里这个变量的约束值。如果在变量中并不存在这一变量的约束，那么我们说这个变量在该特定环境中是无约束的。

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

它不过是lambda的语法糖罢了

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

1. 局部过程的名字不会干包容他们的过程之外的名字互相干扰，这是因为这些局部过程名都是在该过程运行时创建的框架里面约束的，而不是在全局环境里约束的
2. 局部过程只需将包含他们的过程的形参作为自由变量，就可以访问该过程的实际参数。这是因为对于局部过程提的求值所在的环境是外围过程求值所在环境的下属

We implement serializers in terms of a more primitive synchronization mechanism called a mutex. A mutex is an object that supports two operations - the mutex can be acquired, and the mutex can be released. Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released.

Our mutex constructor make-mutex begins by initializing the cell contents to false. To acquire the mutex, we test the cell. If the mutex is available, we set the cell contents to true and proceed. Otherwise, we wait in a loop, attempting to acquire over and over again, until we find that the mutex is available.

In most time-shared operation systems, processes that are blocked by a mutex do not waste time "busy waiting" as above. Instead, the system schedules another process to run while the first is waiting, and the blocked process is awakened when the mutex becomes available.

We must guarantee that, once a process has tested the cell and found it to be false, the cell content will actually be set to true before any other process can test the cell.

The actual implementation of test-and-set! Depend on the details of how our system runs concurrent processes. For example, we might be executing concurrent processes on a sequential processor using a time-slicing mechanism that cycles through the processes, permitting each process to run for a short time before interrupting it and moving on to the next process. In that case, test-and-set! Can work by disabling time slicing during the testing and setting. Alternatively, multiprocessing computers provide instructions that support atomic operations directly in hardware.