3。模块化，对象和状态(3)

这里讨论：

- 基于状态的模拟：约束传播语言
  - 建立一组“变量”之间的约束关系，并在一些变量的值改变之后根据约束关系自动修改其他变量的值
  - 实现一种无方向性的计算
  - 这是一种“另类的”语言和计算模型
- 计算中的时间问题
- 并发系统及其与时间有关的一些情况

  非本课程主题，简单介绍

实例：约束传播语言

- 常规程序实现的都是单方向的计算
  - 表达式描述如何基于变量和常量的值算出结果
  - 函数是从参数出发计算得到返回值
  - 程序是从输入计算出输出
- 实际中也经常需要模拟一些量之间的关系
  - 例如：
    - 金属杆偏转量 **d**，作用于杆的力 **F**，杆的长度 **L**，截面积 **A** 和弹性模数 **E**
    - 几个量之间的关系为 **dAE = FL**
    - 给定了任意 **4** 个量就可确定第**5**个
  - 用常规语言描述这类关系，必须确定特定计算顺序，区分参数和计算结果。这种等式实际上表达了多个“单方向的计算过程”

约束传播语言

- 现在讨论一种约束传播语言
- 其中的“程序”描述的
  - 不是从一些量计算出另一些量的（有方向的）计算过程，而是一批不同的量之间的（无方向的约束）关系
  - 只要给定了除（任意的）一个量之外其他的量，有关“程序”的执行就可以计算出这个缺失的量
- 最重要想法是“约束传播”，认为：
  - 一组变量通过某些方式相互约束
  - 一旦某变量有了值，其值可以通过相关约束传播到其他变量
  - 如果已有的约束足够强，就可以确定某些未定变量的值
  - 如果一个变量的值改变，这种改变也会通过约束传播出去

基本语言结构

- 约束传播语言的基本概念是“约束”

  一个约束有几个端点，描述各端点值之间的某种关系

  对应于 **Scheme** 的基本概念是“表达式”

  在这里写“程序”也就是定义约束

- 语言的基本元素是基本约束，例如：

  (adder a b c) 表示 **a**、**b**、**c** 之间有关系 **a + b = c**

  (multiplier x y z) 表示**x**、**y**、**z** 之间有关系 **x** × **y = z**

  (constant 3.14 x) 表示 **x** 的值永远是 **3.14**

- 提供组合约束的方法：

  提供一种连接器，用于连接不同的约束

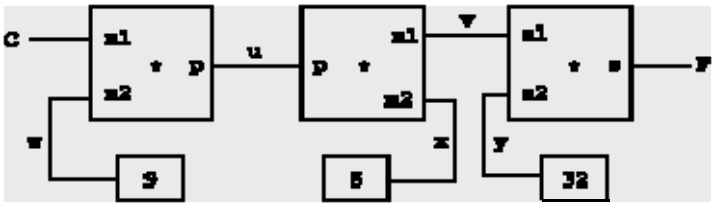
  连接器是一种对象，能保存一个值，使这个值可以参与多个约束

基本语言结构

- 基本约束连接成约束网络（复合约束）
  - 可建立任意复杂的约束网络
  - 约束网络描述任意多个变量之间的复杂约束关系
- 注意：约束网络的表达能力受到基本约束集合的限制
  - 只能是基本约束的组合
  - 基于不同的基本约束集合，可能做不同的事情
- 基本约束必须能支持“多方向”计算
  - 而且在各个方向上具有唯一确定性
  - 完全可能出现各种特殊情况，如
    - 有些约束关系，对一些特殊值在某些方向上无法计算
    - 例如乘法，由于有 0 元素，在一些方向上不可逆

约束传播语言

- 例：华氏温度和摄氏温度之间的关系是 9C = 5(F – 32)，其关系网络：



- 考虑这种网络（连接器网络）
  - 如果某个端点（连接器）取得了一个值
  - 与之关联的约束部件（如果某些相关端点也有值）就有可能确定其他连接器的值，这样得到的值还可能继续向前传
- 假设 C 位置得到值 10，它将导致 u 线得到值 90，……
- 注意，约束部件定义了它的端点间的一种关系

  这里利用了等式表示的多义性：它既可以看作是描述了一种关系（无方向性），又可以看作是描述了一种计算过程（有方向性）

约束传播语言

- 基本约束部件有加法、乘法等，表示相应约束关系
  - 约束部件的一个端点可以连接到一个连接器（连线）
  - 一个连接器可以连接到一个或多个约束部件
- 连接器网络的计算方式：
  - 一旦某个连接器得到新值，就唤醒与之关联的约束部件
  - 被唤醒约束部件逐个处理与它关联的连接器，如果发现已经得到足够的信息为某个连接器确定一个新值，就设置该连接器
  - 得到值的连接器又去唤醒与之关联的约束
- 这个过程可以将信息不断向前传播
  - 信息传播的条件是约束部件已经得到足够的信息
  - 不断传播将信息变化直接或间接地传到网络中所有可能的地方
- 下面先看使用这种语言的实例

约束传播语言的使用

- 假设 make-connector 创建新连接器。下面操作构造了一个约束网络：

```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F)
ok
```

- 创建计算摄氏/华氏转换的约束网络：

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)) )
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

- 为 C 和 F 安装监视器：

```
(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)
```

- 设置 C 为 25：

```
(set-value! C 25 'user)
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done
```

- 设置 F 为 212，导致矛盾

```
(set-value! F 212 'user)
Error! Contradiction (77 212)
```

- 要求系统忘记一些值：

```
(forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
```

- 再设置 F 为 212

```
(set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100
done
```

实现：连接器抽象

- 现在考虑实现

  首先考虑连接器抽象并定义其接口过程

- 连接器是数据抽象，接口过程如下：
  - **(has-value? <*connector*>)** 确定连接器当时是否有值
  - **(get-value <*connector*>)** 取得连接器的当前值
  - **(set-value! <*connector*> <*new-value*> <*informant*>)**

    表明信息源**(informant)** 要求将连接器设置为新值
  - **(forget-value! <*connector*> <*retractor*>)**

    撤销源 **(retractor)** 要求连接器忘记现值
  - **(connect <*connector*> <*new-constraint*>)**

    通知连接器，一个新约束 **<*new-constraint*>** 与之关联

实现：连接器抽象

- 连接器对约束的操作：
  - 连接器得到新值时

    用过程 **inform-about-value** 通知它关联的所有约束
  - 失去原有值时

    用过程 **inform-about-no-value** 通知它关联的所有约束

  约束应该提供相应的过程支持这两个操作
- 考虑几个基本约束器的实现
  - 加法约束
  - 乘法约束
  - 常量约束
  - 监视器

加法约束

- 加法约束器（由一个生成器生成）是一个数据抽象
  - 生成器用过程 **adder** 实现
  - **adder** 返回一个带有内部状态的过程（数据对象）
  - **adder** 有三个连接器参数

    将创建的加法约束连接到这三个指定连接器上并返回该约束
- 加法约束得知相关的某连接器有新值时调用 **process-new-value**

  该过程在确定了至少两个连接器有值时，去设置第三个连接器
- 加法约束被通知放弃值时

  调用内部过程 **process-forget-value**

  通知与之关联的三个连接器

    可以通知或不通知送来放弃值信号的连接器，通知也没关系

加法约束

- 加法约束器在被求和连接器 **a1**、**a2** 与连接器 **sum** 间建加法约束关系

```
(define (adder a1 a2 sum)
  (define (process-new-value)
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum (+ (get-value a1) (get-value a2)) me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2 (- (get-value sum) (get-value a1)) me))
          ((and (has-value? a2) (has-value? sum))
           (set-value! a1 (- (get-value sum) (get-value a2)) me))))
  (define (process-forget-value)
    (forget-value! sum me) (forget-value! a1 me)
    (forget-value! a2 me) (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value) (process-new-value))
          ((eq? request 'I-lost-my-value) (process-forget-value))
          (else (error "Unknown request -- ADDER" request))))
  (connect a1 me) (connect a2 me)
  (connect sum me)
  me)
```

乘法约束

- 乘法约束器（与加法约束器类似）

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
                    (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product (* (get-value m1) (get-value m2)) me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2 (/ (get-value product) (get-value m1)) me))
          ((and (has-value? product) (has-value? m2))
           (set-value! m1 (/ (get-value product) (get-value m2)) me))))
  (define (process-forget-value)
    (forget-value! product me) (forget-value! m1 me)
    (forget-value! m2 me) (process-new-value))
  (define (me request)
    (cond ((eq? request 'I-have-a-value) (process-new-value))
          ((eq? request 'I-lost-my-value) (process-forget-value))
          (else (error "Unknown request -- MULTIPLIER" request))))
  (connect m1 me) (connect m2 me)
  (connect product me)
  me)
```


常量约束

- 常量约束简单地设置连接器的值：

```
(define (constant value connector)
  (define (me request)
    (error "Unknown request -- CONSTANT" request))
  (connect connector me)
  (set-value! connector value me)
  me)
```

初始设置之后，对常量值的任何操作都被看作错误

定义两个语法接口过程（供连接器调用）：

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))

(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

监视器

- 监视器可以附在连接器上，在连接器值被设置或取消时输出信息

```
(define (probe name connector)
      (define (print-probe value)
        (newline)
        (display "Probe: ")
        (display name)
        (display " = ")
        (display value))
      (define (process-new-value)
        (print-probe (get-value connector)))
      (define (process-forget-value) (print-probe "?"))
      (define (me request)
        (cond ((eq? request 'I-have-a-value)
                (process-new-value))
              ((eq? request 'I-lost-my-value)
                (process-forget-value))
              (else
                (error "Unknown request -- PROBE" request))))
      (connect connector me)
      me)
```

连接器的实现

连接器用计算对象实现，局部变量 **value**，**informant** 和 **constraints**，分别保存其当前值、设置它的对象和所关联的约束的表

```
(define (make-connector)
  (let ((value false) (informant false) (constraints '()))
    (define (set-my-value newval setter)
      (cond ( ... ))
            (else 'ignored)))
    (define (forget-my-value retractor) ... ... 'ignored)) ; 定义见后
    (define (connect new-constraint) ... ... 'done)
    (define (me request)
      (cond ((eq? request 'has-value?) (if informant true false))
            ((eq? request 'value) value)
            ((eq? request 'set-value!) set-my-value)
            ((eq? request 'forget) forget-my-value)
            ((eq? request 'connect) connect)
            (else (error "Unknown operation -- CONNECTOR" request))))
    me))
```

连接器内部过程 (for-each-except 在下页定义)

```
(define (set-my-value newval setter)
  (cond ((not (has-value? me))
         (set! value newval) (set! informant setter)
         (for-each-except setter inform-about-value constraints))
        ((not (= value newval)) (error "Contradiction" (list value newval)))
        (else 'ignored) ))

(define (forget-my-value retractor)
  (if (eq? retractor informant)
      (begin (set! informant false)
        (for-each-except retractor inform-about-no-value constraints))
        'ignored))

(define (connect new-constraint)
  (if (not (memq new-constraint constraints))
      (set! constraints (cons new-constraint constraints)))
      (if (has-value? me) (inform-about-value new-constraint) )
  'done)
```


连接器的实现

- 设置新值时通知所有相关约束（除了设置值的那个约束）的过程

  用迭代的方式实现：

```
(define (for-each-except exception procedure list)
    (define (loop items)
      (cond ((null? items) 'done)
            ((eq? (car items) exception)
              (loop (cdr items)))
            (else
              (procedure (car items))
              (loop (cdr items)))))
    (loop list))
```

约束传播语言：连接器的实现

- 基于操作分派为连接器抽象提供的接口过程：

```
(define (has-value? connector)
    (connector 'has-value?))

(define (get-value connector)
    (connector 'value))

(define (set-value! connector new-value informant)
    ((connector 'set-value!) new-value informant))

(define (forget-value! connector retractor)
    ((connector 'forget) retractor))

(define (connect connector new-constraint)
    ((connector 'connect) new-constraint))
```

至此约束语言完成！

我们可以根据需要定义自己的约束网络，令其完成所需的计算

实例（重看）

- 设 make-connector 创建新连接器。构造如下对象：

```
(define C (make-connector))
    (define F (make-connector))
    (celsius-fahrenheit-converter C F)
    ok
```

- 创建计算摄氏/华氏转换的约束网络：

```
(define (celsius-fahrenheit-converter c f)
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)) )
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```

- 为 C 和 F 安装监视器：

```
(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)
```

- 设置 C 为 25：

```
(set-value! C 25 'user)
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done
```

- 设置 F 为 212，导致矛盾

```
(set-value! F 212 'user)
Error! Contradiction (77 212)
```

- 要求系统忘记一些值：

```
(forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
```

- 再设置 F 为 212

```
(set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100
done
```

两个实例的简单总结

- 数字电路和约束传播语言两个实例都有理论和实际意义。其中：
  - 用有局部状态的对象反映系统状态，用消息传递实现对象间交互
  - 定义了较直观易用的语言，支持所需对象的构造和组合。用过程作为组合手段。对象的构造具有闭包性质
  - 比较容易在 OO 语言里模拟
  - 也可以考虑如何在 C 语言里实现
- 有局部状态的变量很有用，但赋值也破坏了引用透明性
  - 使语言的语义再也不可能简洁清晰
  - 还带来许多不易解决的问题
  - 使检查（考虑）程序的正确性的问题变得非常复杂
- 下面进一步分析这方面的一些复杂情况，考虑一些问题

计算中的时间

- 前面说：具有内部状态的计算对象是程序模块化的有力工具
- 但使用这种对象也付出了很大代价：
  - 丧失了引用透明性
  - 代换模型不再适用
  - “同一个”的问题也不再简单清晰
- 出现这些情况，背后的核心问题是时间
  - 无赋值程序的意义与时间无关，一个表达式总算出同一个值
  - 有赋值以后就必须考虑时间的影响。赋值可改变表达式所依赖的变量的状态，同一个变量在不同时间可能有不同的值

    这样，表达式的值基于变量，因此就与求值的时间相关
- 用带有局部状态的计算对象建立计算模型时

  必须考虑时间带来的问题和影响，下面考虑一些问题

计算中的时间

- 现实世界的系统里的对象，通常有许多都在同时活动
  - 要构造出能更贴切地模拟这类系统的模型，最好是用一组计算进程（process，进程）模拟这些同时发生的活动
  - 把一个计算模型分解为一组具有独立的内部状态而且各自独立演化的部分，可能使程序进一步模块化
  - 如果能实现许多对象的独立演化，所做的模拟也会更加逼真

    为此，实际上需要能支持同时运行多个进程的并发硬件
- 即使程序在顺序计算机上运行，按能并发运行的方式写程序，也可能使程序进一步模块化，还能避免不必要的时间约束
  - 实际中，许多事件发生的顺序并不重要或原本就不能确定。一定要把这种过程写成顺序程序，实际是把原本无顺序的活动强行描述为顺序动作（所做模拟不贴切）
  - 这样做也给问题求解增加了原本没有的时间约束（做出的模拟系统可能更复杂，其行为可能更偏离实际）

计算中的时间

- 一个大趋势：多处理器系统越来越普及
  - 能并发运行的程序可以更好利用计算机能力，提高程序运行速度
  - 有可能更贴切、更模块化地模拟复杂的实际系统
  - 但是，有了并发后，赋值带来的复杂性更显严重。这是并发编程非常困难的根本原因，是今天计算机理论和实践领域的最大挑战
- 关键原因：并发活动具有时间上的非确定性
  - 不同活动的相对执行速度相互无关，带来执行顺序上的非确定性
  - 这些情况造成不同进程之间的相互影响方面的非确定性
- 抽象看，时间就像加在事件上的一种顺序。对任意事件 A 和 B
  - 或 A 在 B 之前发生，或两者同时发生，或 A 在 B 之后
  - 这实际上还是简化的看法：认为动作瞬时完成（原子动作）
  - 更一般的情况是动作本身还需要时间

并发和时间

- 例：设 **Peter** 和 **Paul** 的公用账户有 **100** 元，两人分别取**10**元和**25**元，最终余额应该是 **65** 元
  - 由提款顺序不同，余额变化过程可能是

    100 -> 90 -> 65 或 **100 -> 75 -> 65**
  - 余额变化通过对 **balance** 的赋值完成
- 如果 **Peter**、**Paul** 或还有其他人可能从不同地方访问该账户
  - 余额的变化序列将依赖于各次访问的确切时间顺序和操作的实现和进展的具体细节，具有非确定性
  - 例如：修改余额是一个时刻完成的“原子动作”，还是被实际分解为“取出当前余额”，“计算提款后的余额”，“保存计算出的余额”等若干相互分离的独立动作
- 这些问题给并发系统设计提出了严重挑战（什么是正确？不再清晰了）

并发和时间

- 假设 **Peter** 和 **Paul** 的取款工作由两个独立运行的进程完成
  - 两个进程共享变量 **balance**
  - 它们都是执行过程 **withdraw**：

```
(define (withdraw amount)
    (if (>= balance amount)
       (begin (set! balance (- balance amount)) balance)
       "Insufficient funds"))
```

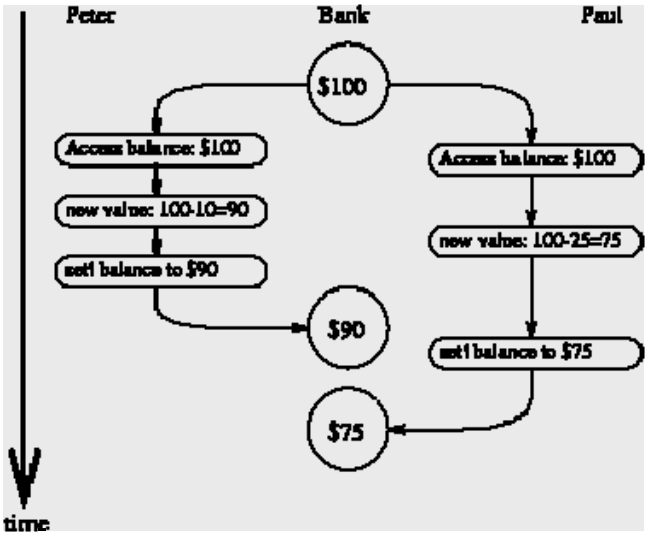
- 由于两进程独立运行，可能出现奇怪的现象。如下面动作序列
  1. **Peter** 的进程检查余额确定取 **90** 元合法（例如当时有 **100** 元）
  2. **Paul** 的进程实际地从账户中取出 **25** 元
  3. **Peter** 的进程实际取款（因为前面已经确定余额够用）

  最后这个动作已经非法（当时的余额不足）

并发带来的问题

- 设 **(set! balance (- balance amount))** 分三步完成：
  - 取变量 **balance** 的值
  - 算出新余额
  - 设置 **balance** 的新值

  如果在 **Peter** 和 **Paul** 的提款进程中这三个操作交错进行，就可能造成余额设置错误，导致银行或客户的损失
- 右图是一种可能情况，由于并发活动的相互竞争而导致系统实际出错 (这种错误称为竞态错误，**racing error)**



并发带来的问题

- 在存在赋值的情况下
  - 多个并发活动“同时”去操作共享变量，就可能产生不正确行为
  - 具体行为依赖于操作发生的顺序和操作执行的细节
- 要做出“正确”“可靠”的并发程序设计，需要定义好什么是“正确”。一种合理观点认为其中最关键的问题是
  - 保证并发运行的进程能不受到外界的不良影响
  - 并发运行能表现出“与某种独立运行一样”的行为
- 对于一个进程，其他并发运行的进程构成了它的运行环境。如果所有进程相互独立地并发运行
  - 一组进程无法自动地保证正确的操作顺序
  - 一个进程无法控制其他进程的行为
  - 一个进程也不能（一般性地）掌握其他进程的进展情况

并发带来的问题

- 要保证系统的正确行为，就要对其中并发行为增加一定限制
  - 这就是并发控制

    从进程之外对进程的活动方式加以一定的控制
  - 也就是说，进程不能自由活动，其活动要满足一定约束
- 进程控制的方法很多，首先要考虑好控制的原则

  两个基本要求：
  - 保证行为“正确”
  - 尽可能并行地执行（以利用基础硬件能力，满足实际需要）
- 可能控制原则1（考虑到问题出在修改共享变量）：

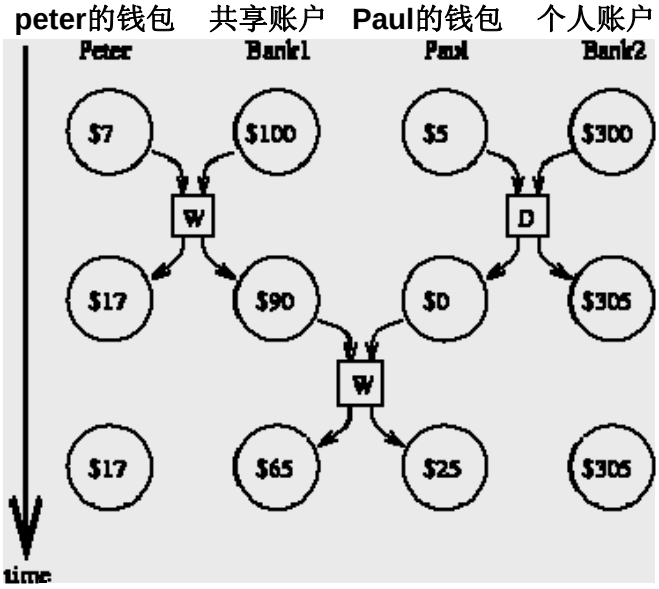
  不允许任何其他操作与修改共享变量的操作同时进行

  也就是说：当一个进程正在执行修改共享变量的操作时，其他进程都不能动，不能执行任何操作

并发带来的问题

- 这种策略是否合理？

  考虑一个场景
- 例：设 **Peter** 和 **Paul** 有一个公用账户，而 **Paul** 自己还有一个个人的账户
- 右图是两人的一系列动作



- **Paul** 的个人账户存款与 **Peter** 的共享账户取款并不冲突，应该允许同时进行
- 上述原则太严，将严重影响系统的效率

并发控制

- 可能原则2：保证并发系统的执行效果等价于不同进程的按某种方式安排的顺序运行。这个原则包含两个方面：
  - 允许不同进程并发运行
  - 要求其效果与某种顺序运行相同
- 例如：
  - 可以允许 **Paul** 在另一账户中取款的操作与 **Peter** 在两人共享账户取款的操作同时发生

    因为这样做并不会影响最终效果
  - 但不允许两人同时在共享帐户存取款

    因为有可能产生不应该出现的效果

并发控制

- 并发执行的行为很难分析和处理，其根源是不同进程执行中产生的事件可能产生大量的不同交错情况
- 假定两进程的事件序列为 **(a,b,c)** 和 **(x,y,z)**

  并发执行可能产生的实际执行序列：

  (a,b,c,x,y,z) (a,x,b,y,c,z) (x,a,b,c,y,z) (x,a,y,z,b,c)
  (a,b,x,c,y,z) (a,x,b,y,c,z) (x,a,b,y,c,z) (x,y,a,b,c,z)
  (a,b,x,y,c,z) (a,x,y,b,c,z) (x,a,b,y,z,c) (x,y,a,b,z,c)
  (a,b,x,y,z,c) (a,x,y,b,c,z) (x,a,y,b,c,z) (x,y,a,z,b,c)
  (a,x,b,c,y,z) (a,x,y,z,b,c) (x,a,y,b,z,c) (x,y,z,a,b,c)
- 设计并发系统时要考虑所有这些交错行为是否都可以接受

  系统复杂时并发进程和事件都会增加，情况很难控制
- 下面讨论一种可能解决方案：

  设计某种通用机制控制并发进程之间的交错，保证系统的正确行为

并发控制

- 一种简单想法是串行控制器（**serializer**），基本思想：

  通过将程序里的过程分组，禁止不当的并发行为

- 具体做法：
  - 将与并发执行相关的过程分解为一些集合（称为互斥过程集，或者互斥过程组）
  - 任何时候，在每个集合里，只允许至多一个过程执行（注意，这里说的是过程执行，也禁止同一个过程的两个执行）
  - 如果某时刻某个集合里已经有了一个过程正在执行
    - 调用同一集合里任何过程的进程都必须“挂起”等待
    - 一个等待进程将一直挂起，直到导致它“挂起”的过程的执行结束，它才有可能开始执行调用的过程
    - 如果有多个进程等待某个过程的执行结束，该过程结束时采用某种策略选出下一个执行进程，其余的进程继续等待

并发控制：串行控制器

- 现在考虑如何通过串行化控制对共享变量访问
  - 假设程序里需要基于共享变量当前值去更新它
  - 把取该变量的值和更新的操作放入同一个过程
  - 保证修改这个变量的其他过程都不与上面过程并发运行

    有其他过程可能修改这个变量，都放入同一个串行化组

  这样就能保证在提取和更新之间变量的值不变
- **Java 的 synchronized 机制也采用了类似的思想**
  - 在对象里存放共享数据
  - 可以把有关类里的一些方法定义为 **synchronized**
  - 操作一个对象的 **synchronized** 方法不允许同时执行
  - 这种机制与面向对象机制良好集成

并发控制：串行控制器

假设 Scheme 的并行扩充增加了基本过程

(parallel-execute <p_1> <p_2> ... <p_k>)

这里的 **<p>** 都是无参过程，**parallel-execute** 为每个 **<p>** 建立一个独立进程，并令这些进程并发运行

例如 **(define x 10)**

```
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (+ x 1))))
```

- 多个可能结果，非确定性。可能行为（5种，前两种合理）：
  - 先乘后加得 **101**
  - 先加后乘得 **121**
  - **x** 加 **1** 出现在 **(* x x)** 两次访问 **x** 之间得 **110**
  - 加出现在乘和赋值之间得 **100**
  - 乘出现在求和和赋值之间得 **11**

并发控制：串行控制器

- 串行控制器的功能是限制并发过程的行为
  - 假设有一个内部过程 **make-serializer** 创建串行控制器
  - **make-serializer** 以一个过程为参数，返回同样行为的过程
    - 参数与原过程一样
    - 但保证其执行被串行化
  - 用同一 **serializer** 生成的过程受这个 **serializer** 的控制
- 下面程序保证只能产生值 **101** 或 **121**

  消除了不正确（不合理）的交错并行

```
(define x 10)

(define s (make-serializer))

(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (+ x 1)))))
```

并发控制：串行化

- make-account 的串行化版本：

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance) balance)
            (else (error "Unknown request -- MAKE-ACCOUNT" m))))
    dispatch))
```

- 这个实现里不会出现两进程同时取款或存款，避免了前面讨论的错误。此外，每个账户用一个串行化控制器，不同账户之间互不干扰

多重资源带来的复杂性

- 串行化控制器是一种有用抽象，可以隔离并发程序的复杂性
- 如果并发程序里只有一种共享资源（如一个共享变量），问题已解决。如果存在多项共享资源（也很常见），程序的行为将更难控制
- 例：假设需要交换两个账户的余额，用下面过程描述这一操作：

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance) (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

- 只有一个进程交换账户余额时不会出问题。但如果 **Peter** 要交换账户 **a1** 和 **a2** 的余额，而 **Paul** 要交换 **a2** 和 **a3** 的余额
  - 假定每个账户都已经正确地串行化
  - 如果 **Peter** 算出 **a1** 和 **a2** 的差之后 **Paul** 交换了 **a2** 和 **a3**，就会出现错误行为（可以找到许多产生错误行为的动作交错序列，还应严格定义什么是“不正确”）

多重资源带来的复杂性

- 例：假设需要交换两个账户的余额，用下面过程描述这一操作：

```
(define (exchange account1 account2)
    (let ((difference (- (account1 'balance) (account2 'balance))))
      ((account1 'withdraw) difference)
      ((account2 'deposit) difference)))
```

- 可以考虑用两个串行控制器把 exchange 串行化

  并重新安排其中的串行控制器访问

- 下面提出发做法是把串行化控制器暴露出来
  - 可以解决前面提出的问题
    - 需要在使用账户的操作层面上控制
  - 但这种做法破坏了账户对象原有的模块性
  - 其他方法也会有问题，请自己设计和分析

多重资源带来的复杂性

- 考虑一种账户实现，其中把串行控制器放入接口，可以通过消息获取，以便应用程序在账户之外实现对账户余额的保护：

```
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount)) balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Unknown request -- MAKE-
ACCOUNT"  m))))
    dispatch))
```

  暴露内部的串行控制器，增加了使用的复杂性和不安全性

多重资源带来的复杂性

- 如何安全使用带串行控制器的账户对象，是使用者的责任

  程序必须遵守复杂的协议（规则）才能保证正确串行化

- 例，考虑下面的新存款过程的实现：

```
(define (deposit account amount)
  (let ((s (account 'serializer)) (d (account 'deposit)))
    ((s d) amount)))
```

- 导出串行控制器能支持更灵活的串行化控制。**exchange** 可重写为：

```
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange)) account1 account2)))
```

  exchange 就是前面定义的过程

- 可以把这些过程再包装起来，但定义更复杂操作时又会遇到类似问题

串行化控制器的实现

- 串行控制器可以基于更基本的互斥元数据抽象实现
  - 互斥元：一种数据抽象，它可以被获取，使用后释放
  - 如果某个互斥元已被获取，想获取该互斥元的其他操作需要等待到该互斥元被释放（任何时候只能有至多一个进程获取它）
  - 假设过程 **make-mutex** 创建互斥元，获取一个互斥元的方式是给它送 **acquire** 消息，释放的方式是给它送 **release** 消息
- 实现串行控制器里需要用一个局部的互斥元：

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val) )
      serialized-p) ) )
```

本过程生成串行化控制器

- 它以过程 **p** 为参数，生成一个加了串行化控制的具有同样功能的过程
- 生成的过程在执行中获取互斥元后执行操作，然后释放互斥元，最后返回结果

互斥元的实现

- **make-mutex** 生成互斥元对象，它有一个内部状态变量 **cell**，其初始值为 **false**。接到 **acquire** 消息时试图将 **cell** 设为 **true**

```
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell) (the-mutex 'acquire)))  ; 重试
            ((eq? m 'release)  (set-car! cell false))))
    the-mutex))
```

- **test-and-set!** 是特殊操作，它检查参数的 **car** 并返回其值，如果参数的 **car** 为假就在返回之前将其设为真。下面是定义：

```
(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

- 这个实现不行。**test-and-set!** 是实现并发控制的核心操作，必须以原子操作的方式执行，不能中断
- 没有这种保证，互斥元也失效

互斥元

- **test-and-set!** 需要特殊支持，它
  - 可以是一条特殊硬件指令
  - 也可能是系统软件提供的一个专门过程
- **test-and-set!** 的具体实现依赖于硬件
  - 在单处理器机器里，并发进程以轮换方式执行
    - 每个可执行进程安排一段执行时间
    - 系统控制进程的执行，适时地把控制转到其他可执行进程

    在这种环境下的 **test-and-set!** 执行中必须禁止进程切换
  - 在多处理器机器，必须有硬件支持的专门指令
- 人们开发了这一指令的许多变形，它们都能支持基本互斥操作

  支持并发的基本机制一直是一个重要的研究课题

  在软件/硬件/理论领域都有重要意义

死锁

- 即使有串行控制器，**serialized-exchange** 还是可能出麻烦
- 实例：设 **Peter** 要交换账户 **a1** 和 **a2**，同时 **Paul** 想交换 **a2** 和 **a1**
  - 假定 **Peter** 进程已进入保护 **a1** 的串行化过程，与此同时 **Paul** 也进入了保护 **a2** 的串行化过程
  - 这时 **Peter** 占着保护 **a1** 的串行化过程，等待进入保护 **a2** 的过程（等待 **Paul** 的进程释放该过程），而 **Paul** 占有保护 **a2** 的串行化过程，并等待进入保护 **a1** 的过程
  - 两人的进程将相互等待，永远也不能前进
- 两个以上的进程，由于相互等待其他方释放资源而无法继续前进，这种情况称为死锁
- 如果并发系统里使用到多重资源，那么总存在出现死锁的可能
  - 死锁是并发系统的一种不可容忍的，但也很常见的动态错误
  - 常遇到计算机系统的许多死机现象，可能就是死锁

并发性，时间和通信

- 人们开发了许多避免死锁的技术，以及许多检查死锁的技术
- 在本问题中避免死锁的一种技术是给每个账户一个唯一标识号
  - 对当前的问题，可以修改 **serialized-exchange**，让每个调用进程都先获取编号较小的账户的保护过程，再获取编号较大的账户
  - 相当于给账户一种排序，要求使用者按同样顺序操作
- 死锁问题的情况很多，控制死锁的方法需要针对问题设计
  - 一些死锁情况可能需要更复杂的技术
  - 不存在对任何情况都有效的死锁控制技术

并发性，时间和通信

- 并发编程中需要控制访问共享变量的顺序
  - 串行化控制器是一种控制工具
  - 人们还提出了许多不同的控制框架
- 实际上，在并发系统中，什么是“共享状态”的问题有时也不清楚：究竟哪些东西应该看着是共享的资源？
- 例如，常见的 **test-and-set!** 等机制要求进程能在任意时刻检查一个全局共享标志，以便控制不同进程的执行情况
  - 但是，新型高速处理器采用了许多优化技术
    - 如（多级）流水线和缓存，弱内存模型等
    - 简单的串行化技术越来越不适用（对效率影响太大）
  - 分布式系统里的共享变量可能出现在不同分布式站点的存储器里
    - 如何保证整个系统的存储一致性变成了一个大问题（首先，要保证的存储一致性条件是什么？）

并发性，时间和通信

- 以分布式银行系统为例

  一个账户可能在多个分行有当地版本
  - 不同版本存放物理上分布于不同地点的不同计算机的存储器里，但都反应着同一个账户的信息，应该相互一致
  - 由于物理分布/网络延迟/操作代价等，不可能每时每刻都一致
  - 不同版本需要定期或不定期地同步
  - 这种情况下，某时刻一个账户的余额是多少？存在不确定性
- 假如 **Peter** 和 **Paul** 分别向同一账户存款
  - 账户在某个时刻有多少钱是不清楚的（是存入时，还是下次全系统同步时？...）
  - 在不断存取钱的过程中，账户的余额也具有非确定性
- 状态、并发和共享信息的问题总是与时间密切相关的，而且时很难控制的。函数式编程中不存在时间，有优越性

总结

- 赋值和状态，背后的问题是对时间的依赖性：
  - 赋值改变变量的状态
  - 从而改变依赖于这些变量的表达式，改变计算的行为
- 并发和赋值相互作用，产生的效果更难把握
  - 无限制的并发可能带来问题，产生不希望的效果
  - 解决办法是对并发加以控制，不允许出现不希望的并发
  - 在复杂的环境里控制并发需要硬件支持
  - 并发程序设计中存在许多很难解决的问题
- 人们正在研究
  - 各种各样的并发编程模型、并发程序的检查和验证技术、并发程序开发的支持环境等等
  - 并发问题在今后很长时间内一直会是计算机领域中的最大挑战