3。模块化。对象和状态(3)

这里讨论：

- 用状态模拟的第一个大例子

  电子线路模拟

- 用状态模拟的第二个大例子

  约束传播语言

- 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

数字电路模拟器

- 介绍大型实例：构造一个数字电路模拟器

  以此作为基于状态的系统模拟的一个实例

- 复杂的数字电路应用于许多领域，其正确设计非常重要
  - 数字电路由一些简单元件构成，通过复杂连接形成复杂的行为
  - 为了理解正在设计的电路，需要做模拟
- 数字电路模拟代表了一类计算机应用：事件驱动的模拟系统

  这是一个重要的计算机应用领域

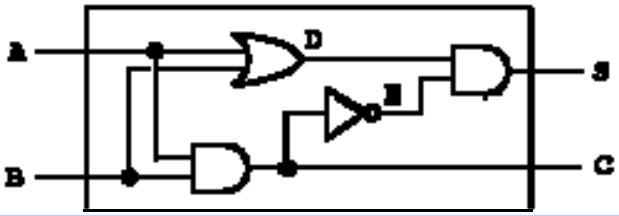
- 事件驱动的模拟的基本想法：
  - 系统的活动中将会发生一些事件
  - 事件的发生又会引发其他事件
  - 事件的发生和影响导致了系统状态的不断变化

数字电路模拟

- 实际的（数字）电子线路由一些基本功能块和它们之间的连线组成
  - 连线在基本功能块之间传递信号
  - 数字电路传递的信号只有 **0** 和 **1** 两种
  - 功能块有若干输入端口和输出端口，从输入信号计算输出
  - 功能块产生输出信号有一定延迟
- 基本功能块：反门**(inverter)**，与门**(and-gate)**，或门**(or-gate)** 等。各种逻辑门都有若干单位时间的延迟
  - 每个功能块有确定的输入端和输出端
  - 不同功能块具有不同的输出端信号与输入端信号的关系

数字电路模拟器：连线

- 电路的计算模型由一些计算对象构成
  - 需要模拟实际电路里的各种基本电路元件，模拟其功能/延迟等
  - 还需用某种机制模拟连线，模拟对象之间的数字信号传递
- 连接基本功能块，可得到更复杂的功能块。如半加器，有输入 **A** 和 **B**，输出 **S**（和）和 **C**（进位）
  - 电子线路见下
  - **A**、**B** 之一为 **1** 时 **S** 为 **1**；**A** 和 **B** 均为 **1** 时 **C** 为 **1**
  - 由于延迟，得到输出的时间可能不同
- 模拟器还需模拟电路的组合



连线

- 下面构造一个模拟数字电路的程序。其中
  - 连线用计算对象表示，它们能保持信号
  - 功能模块用过程模拟，它们产生正确的信号关系
- 在构造模拟数字电路的系统里
  - 连线是一种数据抽象
  - 连线的构造操作是 **make-wire**
- 例如构造 **6** 条连线：

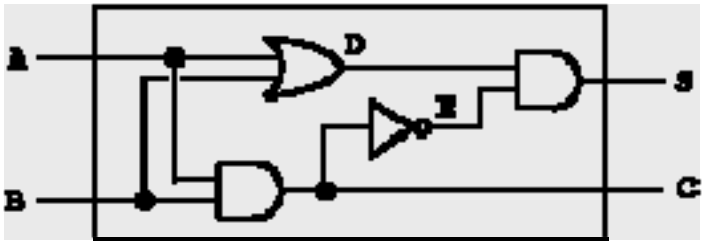
```
(define a (make-wire))   (define b (make-wire))
(define c (make-wire))   (define d (make-wire))
(define e (make-wire))   (define s (make-wire))
```

- 这里的各种基本门电路也是数据抽象
  - 假定已经定义了相应的构造函数（后面实际定义）

数字电路模拟器：组合和抽象

- 将反门、与门和或门连接起来，可以构成半加器：

```
(or-gate a b d)
ok
(and-gate a b c)
ok
(inverter c e)
ok
(and-gate d e s)
ok
```



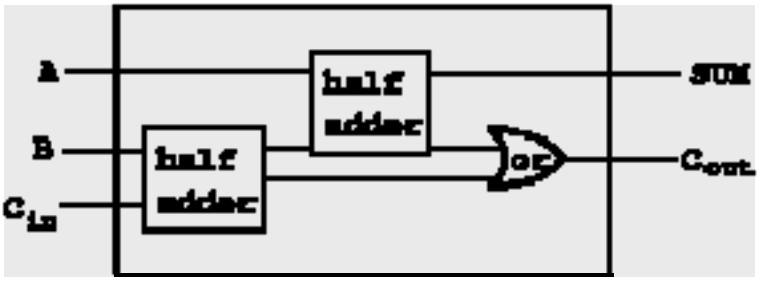
- 更好的方式，是把半加器的构造操作定义为过程，取 **4** 个连线参数：

```
(define (half-adder a b s c)
  (let ((d (make-wire)) (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)
    'ok))
```

数字电路模拟器

- 用两个半加器和一个或门可构造出一个全加器。过程定义：

```
(define (full-adder a b c-in
          sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
    (or-gate c1 c2 c-out)
    'ok))
```



数字电路模拟器：语言结构

- 以构造出的功能块作为部件，可以继续构造更复杂的电路
- 这实际上建立了一种语言，可用于构造结构任意复杂的数字电路
  - 基本功能块是这个语言的基本元素
  - 将功能块用连线连接，是这个语言的组合机制
  - 将复杂连接方式定义为过程，是这里的抽象机制
- 基本功能块实现特定效果，使一条连线的信号能影响其他连线
- 连线基本操作
  - **(get-signal <*wire*>)** 返回 **<*wire*>** 上的当前信号值
  - **(set-signal! <*wire*> <*new value*>)** 将 **<wire>** 的信号值设为新值
  - **(add-action! <*wire*> <*procedure of no arguments*>)** 要求在 **<wire>** 的信号值改变时执行指定过程（过程注册）
  - **after-delay** 要求在给定时延之后执行指定过程（两个参数）

数字电路模拟器：基本功能块

- 基本功能块基于这些操作定义
- 反门在给定时间延迟后将输入的逆送给输出

```
(define (inverter input output)
    (define (invert-input)
      (let ((new-value (logical-not (get-signal input))))
        (after-delay inverter-delay
                     (lambda () (set-signal! output new-value)))))
    (add-action! input invert-input)
    'ok)

(define (logical-not s)
    (cond ((= s 0) 1)
          ((= s 1) 0)
          (else (error "Invalid signal" s))))
```

在 input 连线对象登记无参过程 inverter-input

数字电路模拟器：基本功能块

- 与门有两个输入，这两个输入得到信号时都要执行动作
- 过程 logical-and 与 logical-not 类似，只是有两个参数

```
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value (logical-and (get-signal a1) (get-signal a2))))
      (after-delay and-gate-delay
                   (lambda () (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'ok)

(define (logical-and a1 a2) (cond ... ...))
```

- 或门、异或门等都可以类似定义
- inverter-delay 和 and-gate-delay 等是模拟中的常量，需要事先定义

数字电路模拟器：连线

- 连线是计算对象，有局部的信号值变量 signal-value 和记录一组过程的变量 action-procedures，连线信号值改变时执行这些过程：

```
(define (make-wire)
  (let ((signal-value 0) (action-procedures '())) ；初始值
    (define (set-my-signal! new-value)
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures))
          'done))
    (define (accept-action-procedure! proc)
      (set! action-procedures (cons proc action-procedures))
      (proc))
    (define (dispatch m)
      (cond ((eq? m 'get-signal) signal-value)
            ((eq? m 'set-signal!) set-my-signal!)
            ((eq? m 'add-action!) accept-action-procedure!)
            (else (error "Unknown operation -- WIRE" m))))
    dispatch))
```

辅助过程 call-each 逐个调用过程表里的各个过程（都是无参过程）

执行无参过程 proc

数字电路模拟器：连线

```
(define (call-each procedures)
    (if (null? procedures)
        'done
        (begin
          ((car procedures))
          (call-each (cdr procedures)))))
```

- 使用连线的几个过程：

```
(define (get-signal wire)
  (wire 'get-signal))
(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))
(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))
```

- 连线是典型的变动对象，保存可能变化的信号，用带局部状态的过程模拟。调用 make-wire 返回新的连线对象
- 连线状态被连接在其上的功能块共享，一个部件的活动通过交互影响连线状态，进而影响连接在这条线上的其他部件

数字电路模拟器：待处理表

- 现在要实现 **after-delay** 操作。这里的想法是维护一个待处理表，记录需要处理的事项。该数据抽象提供如下操作：

  (make-agenda) 返回新建的空待处理表

  (empty-agenda? <*agenda*>) 判断待处理表是否为空

  (first-agenda-item <*agenda*>) 返回待处理表中第一个项

  (remove-first-agenda-item! <*agenda*>) 删除待处理表里的第一项

  (add-to-agenda! <*time*> <*action*> <*agenda*>) 向待处理表中加入一项，要求在给定时间运行给定过程

  (current-time <*agenda*>) 返回当前时间

- 待处理表用 **the-agenda** 表示，**after-delay** 向其中加入一个新项

```
(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time the-agenda))
                  action
                  the-agenda ) )
```


数字电路模拟器：模拟

- 模拟驱动过程 **propagate** 顺序执行待处理表中的项
  - 处理表中的项就是在“特定时间”完成“事件”要求的动作
  - 处理中可能加入新项（加入将在特定时间发生的事件）
  - 只要待处理表不空（还有事件等待发生），模拟就继续进行
- 过程定义

```
(define (propagate)
   (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
         (first-item)
         (remove-first-agenda-item! the-agenda)
         (propagate))))
```

监视器

- 实现一个“监视器”

  把它连在连线上时，该连线的值改变时它就会输出信息

```
(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (newline)
                 (display name)
                 (display ", Time = ")
                 (display (current-time the-agenda))
                 (display "  New-value = ")
                 (display (get-signal wire)))))
```


数字电路模拟器：模拟实例

- 初始化：

```
(define the-agenda (make-agenda))
(define inverter-delay 2)
(define and-gate-delay 3)
(define or-gate-delay 5)
```

- 定义 4 条线路，其中两条安装监视器：

```
(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))
(probe 'sum sum)
sum 0  New-value = 0
(probe 'carry carry)
carry 0  New-value = 0
```

- 把线路连接到半加器上：

```
(half-adder input-1 input-2 sum carry)
ok
```

- 将 input-1 的信号置为 1，而后运行这个模拟：

```
(set-signal! input-1 1)
  done
  (propagate)
  sum 8  New-value = 1
  done
```

- 这时将 input-2 上的信号置 1 并继续模拟：

```
(set-signal! input-2 1)
done
(propagate)
carry 11  New-value = 1
sum 16  New-value = 0
done
```

  时刻 11 时 carry 变为 1，时刻 16 时 sum 变为 1

数字电路模拟器：待处理表的实现

- 待处理表的功能与队列类似，其中记录要运行的过程。元素是时间段，包含一个时间值和一个队列，队列里是在该时间要执行的过程：

```
(define (make-time-segment time queue)  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))
```

- 待处理表是时间段的一维表格，其中时间段按时间排序

  为了方便，在表头记录当前时间（初始为 0）

```
(define (make-agenda) (list 0))
(define (current-time agenda) (car agenda))
(define (set-current-time! agenda time) (set-car! agenda time))
(define (segments agenda) (cdr agenda))
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))
```


数字电路模拟器：待处理表的实现

- 将动作加入待处理表的过程：

```
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments) (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments)) action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr! segments
                (cons (make-new-time-segment time action) (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments! agenda
          (cons (make-new-time-segment time action) segments))
        (add-to-segments! segments))))
```

数字电路模拟器：待处理表的实现

- 需要从待处理表删除第一项时，应删除第一个时间段队列里的第一个过程。如果删除后队列为空，就删除这个时间段：

```
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))
```

- 待处理表的第一项就是其第一个时间段队列里的第一个过程。提取项时应该更新待处理表的时间：

```
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty -- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```


数字电路模拟器

- 至此数字电路模拟系统的开发完成。总结一下
- 系统类型：离散事件模拟系统
  - 事件，事件的发生，执行并可能产生新事件
  - 技术：用一个待处理事件表，维护和使用这个表
  - 这是一般性是问题和技术
- 具体系统：模拟电子线路的活动
  - 一组基本功能块，表示基本电路，用带有状态的对象模拟
  - 实际电路是更简单的电路的组合（闭包性质）
  - 定义连线对象，用于连接简单电路构成更复杂的电路。在模拟中，连线负责在部件之间传递信号
- 注意：电路模拟器的接口实际上形成了一个描述电路及其组合的语言：以基本功能块作为基本元素，以连线的方式组合，通过定义高级过程的方式建立语言抽象

总结：数字电路模拟器

- 可以考虑在其他语言里实现类似的模拟器
  - 例如 C 语言或者 C++ 语言
  - 可以参考 Scheme 实现中的想法
- 解决这个问题可以作为本课程的第二个大作业
  - 如果选做这个，现在就可以开始了
  - 后面还会提出这种类型的大作业
  - 对这种作业，一项工作是将自己完成的实现与书上的 Scheme 实现做一个比较。如考虑
    - 各自的长处和缺点
    - 设计实现的结构统一性
    - 易理解和易扩充性
    - 等等

实例：约束传播语言

- 常规程序实现的都是单方向的计算
  - 表达式根据一下变量的值算出结果
  - 函数从参数出发计算得到返回值
  - 程序从输入算出输出
- 实际中也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 个就可确定第5个
  - 用常规语言描述时必须确定计算顺序，确定参数和计算结果。可以说，这样一个表达式代表了多个单方向“计算过程”

约束传播语言

- 下面讨论一种约束传播语言
  - 其“程序”里描述的不是从一些量计算出另一些量的过程，而是一批不同的量之间的关系。执行中可以从其中一组量算出另一个量
  - 最重要想法是“约束传播”，即认为：一组变量通过某些方式相互约束，当一个变量有了值后，就可能通过相关约束传播到其他变量。如果约束足够强，就可能确定某些未定变量的值
- 约束传播语言的基本结构
  - 语言的基本元素是基本约束，例如：

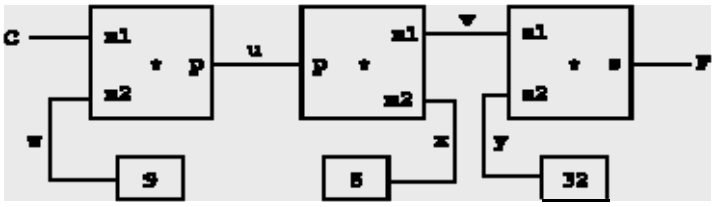
    (adder a b c) 表示 **a**、**b**、**c** 之间有关系 **a + b = c**

    (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**
  - 提供组合各种约束的方法，以描述各种复杂关系
  - 基本想法是用约束网络组合各种约束，用连接器来连接约束

实例：约束传播语言

- 连接器是一种对象，它能保存一个值，使这个值可以参与多个约束
- 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）上的情况
  - 假设某个端点（连接器）取得了一个值
  - 如果有了这个连接器值，与之关联的约束部件就可以确定其他连接器的值，相应的值就可以继续向前传
- 假设 **C** 位置得到值 **10**，它将导致 **u** 线得到值 **90**，……
- 注意，约束部件定义了端点间的一种关系，无方向性（等式的二义性）

约束传播语言

- 这里的加法、乘法等是约束部件，表示约束关系
  - 约束部件的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：
  - 某个连接器得到新值时就去唤醒与之关联的约束部件
  - 被唤醒的约束部件逐个处理与它关联的连接器，如果有已经足够信息为某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
- 这样就会使信息不断向前传播
  - 信息传播的条件是约束部件得到的信息是否足够
  - 信息的变化可以间接地传到网络中所有可能的地方
- 在定义约束传播语言的基本构件前，先看几个实例

约束传播语言：使用

- 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
```

约束传播语言：连接器抽象

- 现在考虑实现。首先考虑连接器抽象，定义其接口过程
- 连接器是数据抽象，其接口过程：
  - **(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
- 连接器对约束的操作：
  - 得到一个新值时用过程 **inform-about-value** 通知所关联的约束
  - 失去原有的值时用 **inform-about-no-value** 通知

约束传播语言：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
  - 返回一个带有内部状态的过程（数据对象）
  - **adder** 以三个连接器作为参数

    把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“来放弃值信号）的连接器，都通知也没关系）

约束传播语言：加法约束

- adder 在被求和连接器 a1、a2 与连接器 sum 之间建一个加法约束

```
(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)
```


约束传播语言：乘法约束

- 乘法连接器（与加法连接器类似）

```
(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)
```

约束传播语言：常量约束

- 常量约束简单设置连接器的值，任何修改其值的行为都是错误：

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

语法接口过程：

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

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


约束传播语言：监视器

- 监视器可附在指定连接器上，当连接器的值被设置或取消时产生输出

```
(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)
```

约束传播语言：连接器的实现

连接器用计算对象实现，包含局部状态变量 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 ((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) ... ... 'ignored))
    (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))
```

约束传播语言：连接器的实现

两个内部过程的完整定义：

```
(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)
```

- 设置新值时通知所有相关约束（除了设置值的那个约束）。用迭代过程实现：

```
(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

- 为分派提供接口的几个过程：

```
(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
```

总结

- 两个实例都很有意思。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
  - 定义了好用的语言，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
  - 容易在 **OO** 语言里模拟
  - 也可以考虑考虑如何在 **C** 语言里实现
  - 可以作为大作业（参考前面的说明）
- 有局部状态的变量很有用，但赋值破坏了引用透明性
  - 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
  - 检查（考虑）程序的正确性，问题变得更加复杂
- 下面进一步考虑这方面的问题，考虑更复杂的情况

计算中的时间

- 前面说，内部状态的计算对象是程序模块化的有力工具
- 但如果使用这种对象，也要付出很大的代价：
  - 引用透明性不再有了
  - 代换模型不再适用
  - “同一个”的问题也不再简单清晰
- 问题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
  - 有赋值后就必须考虑时间的影响。应为赋值可改变表达式所依赖的变量的状态，从而改变表达式的值
  - 这样，表达式的值就依赖于求值的时间
- 用带有局部状态的计算对象建立计算模型时

  必须考虑时间的问题和影响

计算中的时间

- 在现实世界的系统里通常有许多同时活动的对象
  - 构造更贴切的模型，最好用一组计算进程（process进程）模拟
  - 将计算模型分解为一组具有独立的内部状态且各自独立演化的部分，可能使程序进一步模块化
- 即使程序在顺序计算机上运行，按照能并发运行的方式写程序，不但能使程序进一步模块化，还能帮助避免不必要的时间约束
  - 许多情况中事件发生的前后顺序并不重要或者不确定。如果非要写顺序性程序，就强行将它们描述为顺序进行的动作（不贴切）
  - 这样做，给问题求解增加了原本没有的时间约束
- 此外，现在多处理器越来越普及
  - 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
  - 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

并发和时间

- 在抽象层面，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
- 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，余额变化序列将依赖于各次访问的确切时间顺序和操作细节，具有非确定性

  这些对并发系统的设计提出了严重挑战（什么是正确？）

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"))
```

- 由于两个进程独立运行，就可能出现一些奇怪现象。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** 的进程实际取了 **25** 元
  3. **Peter** 的进程实际取款

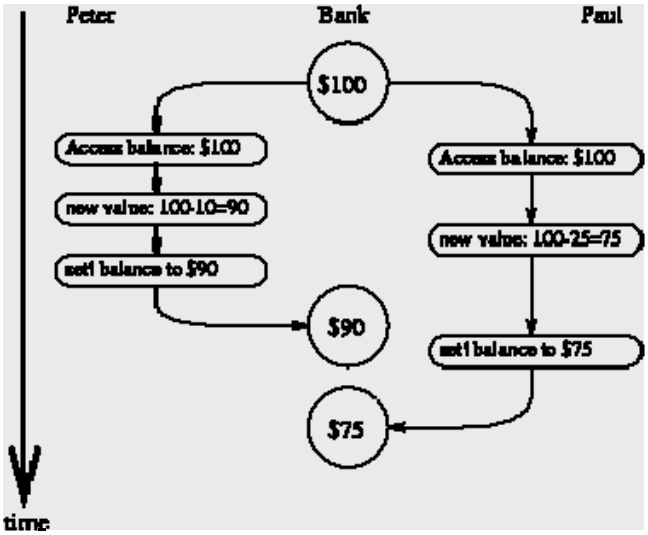
  最后这个动作已经非法了（当时的余额不足）

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 **Peter** 和 **Paul** 的提款进程中的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 (这称为竞态错误)



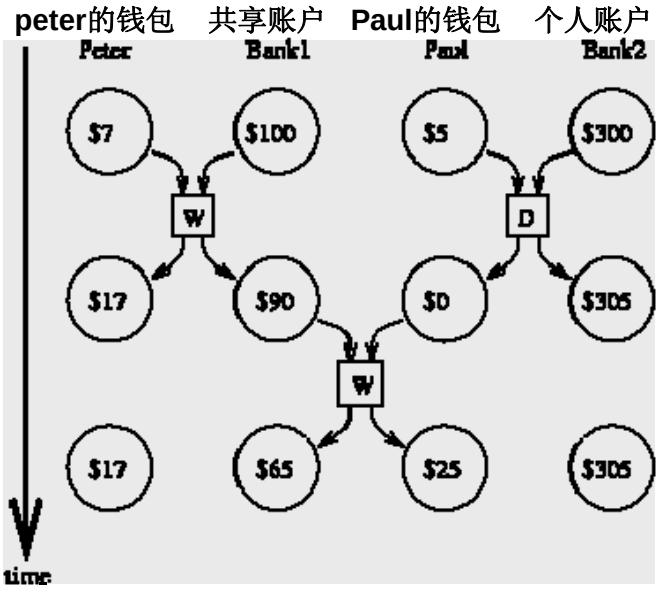
并发带来的问题

- 在存在赋值的情况下
  - 多个并发活动“同时”去操作共享变量，就可能造成不正确行为
- 在并发程序设计中，最关键是保证并发运行的进程能不受到外界的不良影响，能表现出某种“与独立运行一样”的正确行为
  - 对一个进程而言，其他并发运行的进程是它的运行环境
  - 但并发运行的一组进程无法自动地保证正确的操作顺序。因为进程相互独立，一个进程不可能控制其他进程的行为
- 要保证系统的正确行为，就需要对其并发行为增加一定限制
  - 这就是并发控制，即需要从进程之外对它们的活动加以控制
  - 也就是说，不同进程不能自由活动，其活动要受到一定约束
- 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）

并发带来的问题

- 可能控制原则**1**：不允许修改共享变量的操作与任何其他操作同时进行

  也就是说：执行修改共享变量的操作时，其他操作都不能做
- 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。右图是两人的几个动作
- **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
- 说明上述原则太严，可能严重影响系统的工作效率



并发控制

- 可能原则2：保证并发系统的执行效果等价于按某种方式安排的不同进程的顺序运行。原则有两方面：
  - 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：允许Paul 在另一账户取款与 Peter 在两人共享账户取款同时发生，因为这样不影响最终效果；但是不允许两人同时在共享帐户存款或取款，因为这样会出现不良效果
- 并发很难处理，根源是不同进程产生的事件可能产生大量交错情况

  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)

并发控制

- 设计并发系统时要考虑所有交错行为是否都可接受
- 如果系统很复杂，并发进程和事件增加，相关的事情将很难控制
  - 可能解决方法是设计一些通用机制
  - 用这些机制控制并发进程之间的交错，保证系统的正确行为
- 下面介绍一种简单想法，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：
  - 令过程分属一些集合（互斥过程集，互斥过程组）
  - 任何时候每个集合中至多允许一个过程执行
  - 如果某时刻，一个集合里的过程中有一个正在执行，调用同一集合里的过程的进程就必须等正在执行的过程结束后，所调用的过程才才能开始执行

并发控制：串行控制器

- 通过串行化来控制对共享变量的访问
  - 把提取某共享变量当前值和更新该变量的操作放入同一过程
  - 保证给该变量赋值的其他过程都不与这个过程并发运行（放入同一个并行化组）
  - 就能保证在提取和更新之间变量的值不变
- **Java** 的 **synchronized** 方法也是这种思想
  - 在对象里放共享数据
  - 操作对象的多个 **synchronized** 方法不允许同时执行

在 **Scheme** 语言里实现串行化：要假设扩充基本语言，加入过程

(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

并发控制：串行控制器

```
(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）：
  - 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**
- 使用串行控制器可以限制并发过程的行为。设过程 **make-serializer** 创建串行控制器，它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
- 下面程序只能产生值 **101** 或 **121**（消除了不正确的并行）

```
(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))
```

- 这个实现中不会出现两个进程同时取款或存款，避免了前面说的错误。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很有用的抽象，用于隔离并发程序的复杂性。如果并发程序中只有一种共享资源（如一个共享变量），问题已可以处理。如果存在多项共享资源（常见），程序的行为控制更困难
- 例：假设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)))
```

- 只有一个进程去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
  - 设每个账户已正确地串行化
  - 若 **Peter** 算出 **a1** 和 **a2** 的差之后 **Paul** 交换了 **a2** 和 **a3**，程序就会产生不正确行为（可以找到许多产生不正确行为的动作交错序列，什么是“不正确”还需要严格定义）

多重资源带来的复杂性

- 可用两个串行控制器把 **exchange** 串行化，并重新安排对串行控制器的访问。下面做法把串行化控制器暴露出来，破坏了账户对象原有的模块性。（其他方法也会有问题，请自己设计并分析）
- 下面的账户实现把串行控制器放在接口中，可通过消息获取，以便在账户之外实现对该账户余额的保护：

```
(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

- 安全地使用这种带串行控制器的账户对象是使用者的责任，所写的程序必须按复杂的规矩（协议）使用，才能保证正确的串行化管理
- 例如，新的存款过程可以如下实现：

```
(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** 可以重写为：

```
(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
  - 如果某互斥元已被获取，企图获取这一互斥元的其他操作需要等待到该互斥元被释放（任何时候至只能多有一个进程获取它）
  - 设 **make-mutex** 创建互斥元，获取的方式是给互斥元送 **acquire** 消息，释放的方式是给它送 **release** 消息
- 每个串行控制器里需要一个局部的互斥元，其实现是：

```
(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

- 返回串行化控制器
- 它以 **p** 为参数，返回加上串行化控制的同一过程
- 执行中先获取互斥元，而后执行操作，最后释放互斥元

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!** 是一个特殊操作。它检查参数的 **car** 并返回其值，如果参数的 **car** 为假就在返回值之前将其设为真：

```
(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，互斥元也将失效

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

  具体实现依赖于硬件
- 在只有一个处理器的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
  - 而后系统将控制转给其他可执行进程

  这种环境下的 **test-and-set!** 执行中必须禁止进程切换
- 对多处理器机器，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，它们都能支持基本的互斥操作

  人们一直在研究支持并发的基本机制

死锁

- 即使有了串行控制器，**serialized-exchange** 还是可能出麻烦：

  设 **Peter** 要交换账户 **a1** 和 **a2**，同时 **Paul** 也想交换 **a2** 和 **a1**

  假定 **Peter** 进程已进入保护 **a1** 的串行化过程，同时 **Paul** 也进入了保护 **a2** 的串行化过程

  这时 **Peter** 在占有保护 **a1** 的串行化过程的状态中等待进入保护 **a2** 的过程（等待 **Paul** 的进程释放该过程），而 **Paul** 在占有保护 **a2** 的串行化过程的状态中也等待进入保护 **a1** 的过程

  两人的进程相互等待永远不能前进
- 两个以上进程由于相互等待他方释放资源而无法继续的情况称为死锁
- 使用多重资源的并发系统里总存在死锁的可能
  - 死锁是并发系统里的一种不可容忍的，但也是很常见的动态错误
  - 常见系统的许多死机现象实际上是死锁

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为每个账户确定一个唯一标识号

  对当前这个问题，需要修改**serialized-exchange**，增加一点代码，保证调用它的进程总先获取编号小的账户的保护过程
- 其他死锁可能需要更复杂的技术

  不存在在任何情况下都有效的死锁控制技术
- 并发编程中需要控制访问共享变量（表示共享状态的变量）的顺序
  - 前面提出的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，在并发中究竟什么是“共享状态”的问题常常也很不清楚
  - 究竟哪些东西应该看着是共享的？

并发性，时间和通信

- 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，以便控制进程的进展
  - 但目前的新型高速处理器采用了许多优化技术，如（多级）流水线和缓存等，串行化技术越来越不适用了（对效率影响太大）
- 在分布式系统里，共享变量可能出现在不同分布式站点的存储器里
  - 在整个系统的存储一致性变成了很大的问题
- 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，这些版本需要定期或不定期地同步。这种情况下账户余额就变成不确定的
  - 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。这些情况反过来使人想到函数式编程的优越性

总结

- 赋值和状态改变背后的问题是时间依赖性：
  - 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
- 并发和赋值的相互作用产生的效果更难把握
  - 无限制的并发可能带来各种问题，可能产生不希望的效果
  - 解决问题的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