Personal Statement

I am a 34-year-old Macau citizen who resides in Macau and has been programming for 9 years now. I once enrolled in an undergraduate program majoring in mathematics in the UK but dropped out. After that, I worked as an iOS developer for 4 years in Beijing. I am currently self-studying computer science in Macau and I am interested in the undergraduate computer science program offered by the University of Macau.

I was born in Macau but raised in Beijing, where I completed my primary, middle, and high school education. Following my high school graduation in 2009, I pursued a pre-university course abroad in the UK. Subsequently, in 2010, I secured admission to pursue a degree program specializing in mathematics at the University of Keele in the UK. After two years of study, I dropped out from the program. I was not as mature as I wished for, and it was not clear to me what my career path would be after graduating from Keele on a math degree.

I self-studied programming after moving back to Beijing and immediately became very interested in computer science and engineering. I have worked as an iOS developer in two companies in Beijing. One of them was a technology company focused on building an online medical consultation platform, while the other one was a sports media company. I gained valuable experiences from both job positions. In the early days at the medical technology company, I served as an entry-level software engineer. Later on at the sports media company, I took charge of app development and I successfully launched a sports news app for which I served as the architect and made most of the design choices.

My working experience made me realize that my lack of formal and systematic training in computer science greatly limits my potential to grow further. I turned to online resources and started to learn from online courses. After 4 years of work, I decided to quit my job and move back to Macau in order to focus on a fascinating MIT open course called "Structure and Interpretation of Computer Programs" If you are interested in the details, please refer to the document "`My Journey of Lisp Programming`". For example, over the past year I've learned and mastered **concurrency and sharing**, which are critical and lay the foundation for the more advanced courses such as operating systems, computer networks, and database systems in your program.

As I was introduced to various computer science topics through the practice of functional programming, not only was I inspired to learn computer science and AI, but I also unlearned many bad habits that I had developed during my time as an iOS developer. Now, I pay more attention to writing readable code, practicing defensive programming, testing, and documenting. Furthermore, I have also incorporated a learning habit into my daily routine.

To obtain a more systematic training, I am interested in the program at the University of Macau (UM), because it focuses on computer science, along with artificial intelligence topics which I am interested in. I believe the coursework UM offers (e.g., computer organization, algorithms and analysis, operating systems, different types of maths, computer networks, etc.) can help me succeed in my future career upon graduation. I would also be more than willing to serve as a research assistant if there is an opportunity. Additionally, the affordability of the program is also a major advantage for me.

Overall, I believe I am proficient in programming and highly likely to succeed in a challenging CS program like the one offered by the University of Macau. From my industry experience, I know how competitive and rewarding a career in software engineering can be, and I am strongly motivated to perform well in the program. Furthermore, I possess strong teamwork skills and language skills (I am fluent in English, Cantonese, and Mandarin), which will enable me to effectively collaborate with my younger classmates all over the world. I sincerely hope that UM will consider my application and thank you for your patience in reading until the end.

My journey of Lisp programming

I have been self-studying computer science primarily concentrating on the text named "Structure and Interpretation of Computer Programs". Currently, I am studying chapter 3.5 (streams) and 4.1 (metacircular-evaluator). As proof of my progress, I have uploaded some of my previous works. Recent projects are marked as "new". Now, let me briefly talk about what I have learned up to now:

The first two chapters of SICP focus on procedural abstraction and data abstraction. In my opinion, procedural abstraction involves writing a logical description of a process that emphasizes readability and correctness. Data abstraction, on the other hand, entails constructing data with operations that can be intuitively used by users. Additionally, the concept of using procedures as data complements both procedural abstraction and data abstraction effectively.

Higher order procedures are powerful tools for achieving procedural abstraction. A higher order procedure called "map," when combined with recursion, allows for elegant expression of tree recursive processes. Codes and test cases can be found in "`lab5`". The MAP procedure also aids in understanding the interpretation process of the Scheme programming language. Test cases related to a simple Scheme evaluator that I extended to include special forms AND, LET, and MAP can be found in "`transcript-interpreter-extension`".

Understanding data abstraction has helped me clarify my confusion about issues in the software engineering process. During my time as an iOS developer (Objective-C), I always wondered what the ultimate architecture was that I could rely on to handle the constantly changing demands from product managers. I tried an architecture called MVVM, which worked out well and was better than the traditional MVC model, but I still hoped for something even better. After studying the principle of data abstraction, which I believed was to defer design decisions by means of building abstraction barriers, I realized that it is impossible to design everything in advance. Instead, I should build simple components and then modify their internals at a later time. Additionally, I learned about the abstract nature of data: Data is independent of its internal representation as long as its external behaviors are coherent and consistent. I encountered an interesting problem involving church numerals and its test case in "`church-numerals`" effectively demonstrates the abstract nature of data.

I extensively practiced on various problem sets and assignments that I could find on the internet. Some assignments impressed me more than others. For example:

1."`Concurrency(new)`" is by far the most time-consuming project I have ever been involved in. This project encourages learners to self-explore issues related to concurrency, sharing, and machine learning. For example, I utilized a feedback loop and a concurrent data structure to effectively and accurately handle the race condition in this project. I also documented the background of the project and my thought process of the problems I encountered as clearly and simply as possible. Thus, this document demonstrates my documentation skills and my latest advancement in self-studying computer science.

2. A conversational system based on a pattern matcher can be found in "`transcript-freshmanAdvisor`". The recursive process involved in the pattern matcher was very complicated, which made me want to restudy proofs that can help reason such a process. Therefore, I studied "Mathematical Thinking" on Coursera and currently, I am at week 8
(introduction to real-analysis).

3. "`An elementary study(`new`)`" on how the Scheme programming language can implement object-oriented programming in various ways.

4. A practice relate to data-directed programming and generic operations in the context of polynomial calculations can be found in "`transcript-2.5`".

Streams enable efficient sequence manipulations. On the other hand, the implementation of algorithms seem to be dependent on the order of evaluation of the programming language. Using streams to solve nth-queens puzzle can be found in "`implicit-style—streams(`new`)`". Using streams to model RLC circuit can be found in "`exercise-3.80(`new`)`".

Overall, if I were to pick one of the most prominent benefits I have gained from programming in Scheme so far, I would claim that it is the habitual separation of syntax and semantics when encountering novel languages. Now, when I come across an unfamiliar domain, I focus on what problem the domain aims to solve and what consistency and coherence it intends to convey.

Links I study from:

https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/index.html

https://people.eecs.berkeley.edu/~bh/61a-pages/

https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computer-programs-spring-2005/

https://groups.csail.mit.edu/mac/classes/6.001/FT98/psets/

```
;;This is an exercise from the textbook exercise2.27
;;For reference see https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-15.html#%_sec_2.2.2

(define (reverse l)
  (define (iter in out)
    (if (null? in) out
        (iter (cdr in) (cons (car in) out))))
  (iter l nil))

STk> (reverse '(1 2 3 4))
(4 3 2 1)
STk> (reverse '(1 (2 (3 4) 5 6) 7 8))  ;;reverse can only reverse outer list but not inner list. *1*
(8 7 (2 (3 4) 5 6) 1)


(define (deep-map-reverse deeplist)
  (map (lambda (element)                ;;step2:for each element in reversed-list
         (if (list? element)            ;;step3:if element contain a list
           (deep-map-reverse element)   ;;step4:do it again by reverse inner list.
            element)                    ;;step4:otherwise, done.
        )
   (reverse deeplist)))                 ;;step1:reverse input list. e.g. *1*

STk> (deep-map-reverse '(1 (2 (3 4) 5 6) 7 8))
(8 7 (6 5 (4 3) 2) 1)
```

<<<<——BACK—TO—MY—JOURNEY—OF—LISP—PROGRAMMING

<<<<——BACK—TO—MY—JOURNEY—OF—LISP—PROGRAMMING

;;This file has 2 parts:
;;part1: before loading mywork–lab6, error occurs.
;;part2: after loading mywork–lab6, functions work now.
;;To see my code, view mywork–lab6 in the same file.

;;part 1
;;——————before–––––loading––mywork–lab6–––––AND,LET,MAP–1 special form ––– don't work
;;Errors occur when evaluating expression (and .....) (let .....) (map–1 ....) by scheme–1

```
STk> (load "~/cs61a/lab/lab6/scheme1.scm")
okay

STk> (scheme–1)
Scheme–1:

(and 1 2 false)                    ;;evaluating AND expression by Scheme–1

*** Error:
    bad proc: #[subr and]
Current eval stack:
__________________
  0    (apply stk–error args)
  1    (print (eval–1 (read)))
  2    (scheme–1)

STk> (scheme–1)
Scheme–1:

(let ((x 1) (y 2))                 ;;evaluating LET expression by Scheme–1
             (+ x y))

*** Error:
    unbound variable: x
Current eval stack:
__________________
  0    x
  1    (eval exp)
  2    (apply–1 (eval–1 (car exp)) (map eval–1 (cdr exp)))
  3    (apply–1 (eval–1 (car exp)) (map eval–1 (cdr exp)))
  4    (map eval–1 (cdr exp))
  5    (apply–1 (eval–1 (car exp)) (map eval–1 (cdr exp)))
  6    (print (eval–1 (read)))
  7    (scheme–1)
```

```
STk> (scheme-1)

Scheme-1:
(map-1 (lambda (x) (* x x)) (list 1 2 3))                  ;;evaluating map-1 expression by Scheme-1

*** Error:
    unbound variable: map-1
Current eval stack:
__________________
  0    map-1
  1    (eval exp)
  2    (apply-1 (eval-1 (car exp)) (map eval-1 (cdr exp)))
  3    (print (eval-1 (read)))
  4    (scheme-1)


;;part 2
——————————————after--loading--mywork-lab6---AND,LET,MAP-1 special forms
work-------
;; (and ....) (let ....) (map-1 ...) work now using scheme-1.

STk> (load "~/cs61a/lab/lab6/mywork-lab6.scm")
okay

STk> (scheme-1)

Scheme-1:
(and 1 2 false)

#f

Scheme-1:
(and (if (> 2 1) true false) (= (+ 3 3) 6))

#t

Scheme-1:
(let ((x 1) (y 2))
          (+ x y))
3

Scheme-1:
(map-1 (lambda (x) (* x x)) (list 1 2 3))
(1 4 9)
```

<<<<——BACK—TO—TRANSCRIPT–INTERPRETER–EXTENSION

;;;To see original eval–1 scheme interpreter provided by cs61a, view scheme1 in the same file,
;;;OR visit https://people.eecs.berkeley.edu/~bh/61a-pages/Volume1/labs.pdf , then
;;;see week6

;;created 03/08/2022

;;;Implementation of map–1 involve using a trick called Y–combinator I learnt from week2.
;;;AND, LET, MAP special forms were added in eval–1.
;;;map–1 is to avoid name conflict with built–in map.

```
(load "~/cs61a/lab/lab6/scheme1.scm")


;; AND special form added
(define (eval–1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp))              ; use underlying Scheme's EVAL
        ((quote–exp? exp) (cadr exp))
        ((if–exp? exp)
         (if (eval–1 (cadr exp))
             (eval–1 (caddr exp))
             (eval–1 (cadddr exp))))
        ((lambda–exp? exp) exp)
        ((define–exp? exp)
         (eval (list 'define (cadr exp) (maybe–quote (eval–1 (caddr exp))))))
;;————————separating—line————from——original—scheme–1——code————————
;;————AND Special form: ————————
        ((and–exp? exp)
;;try and verify–falsity form mutual recursion in order to return false as soon as possible.
         (let ((try
                  (if (null? (cdr exp))
                      true
                      (verify–falsity (cadr exp)))))

           (cond ((null? (cdr exp)) #t)
                 ((equal? #f try) #f)
;;reevaluate the AND special form excluding the verified element.e.g (and x2 x3)
 ;;we have to keep the first word AND in order to be an AND special form.
                 (else (eval–1 (cons (car exp) (cddr exp)))))))
;;————AND Special form ————————————

;;————LET Special form ——————————————
        ((let–exp? exp) (eval–1 (substitute (let–body exp)
```

```
                                        (map car (let-bindings exp))
                              (map eval-1 (map cadr (let-bindings exp)))
                                        '())))
;;————LET Special form——————————
;;——————MAP Special form————————————–
     ((map-1? exp) (cond ((= (length exp) 3)
                          (apply-1 (eval-1 combinator-map)
                                   (map eval-1 (cdr exp))))

                         ((> (length exp) 3)
                          (apply-1 (eval-1 combinator-accumulate-n)

                                   (list (eval-1 (cadr exp))
                                         (init-value (cadr exp))
                                         (map eval-1 (cddr exp)))))



                         (else (error "map-1 wrong argument"))))

;;——————MAP Special form—————————————————

;;————original——eval-1——code——from——scheme-1————————
     ((pair? exp) (apply-1 (eval-1 (car exp))      ; eval the operator
                           (map eval-1 (cdr exp))))
     (else (error "bad expr: " exp))))

;;————————end————eval-1———————————————————

;;——predicate used in And special form
(define and-exp? (exp-checker 'and))

(define (verify-falsity exp)
  ;;if expression is false then return #f symbol,
  ;;else evaluate the expression
  (if (equal? exp #f)
      #f
      (eval-1 exp)))



;;——predicate used in LET special form
(define let-exp? (exp-checker 'let))

;;——selectors of let expression
```

```
(define let–bindings cadr)
(define let–body caddr)
```

;;——predicate used in MAP–1 special form

```
(define map–1? (exp–checker 'map–1))
```

;;——define higher–order procedure “map” in terms of Y–combinator

```
(define combinator–map
  '(lambda (f n)
     ((lambda (map) (map map f n))
      (lambda (map f n)
          (if (null? n)
              '()
              (cons (f (car n)) (map map f (cdr n))))))))
```

;;——define higher–order procedure “accumulate” using Y–combinator

```
(define combinator–accumulate–n
  '(lambda (op initial sequence)
     ((lambda (f) (f f op initial sequence))
      (lambda (accumulate–n op init seqs)
          (if (null? (car seqs))
              nil
              (cons ((lambda (op initital sequence)  ;;accumulate
                         ((lambda (f) (f f op initital sequence))
                          (lambda (accumulate op initial sequence)
                              (if (null? sequence)
                                  initial
                                  (op (car sequence)
                                        (accumulate accumulate op initial (cdr sequence)))))))
                     op init (map car seqs));; apply accumulate to actual argument
                    (accumulate–n accumulate–n op init (map cdr seqs))))))))
```

;;——procedure used for MAP–1

```
(define (init–value op)
  (cond ((equal? op '+) 0)
        ((equal? op '–) 0)
        ((equal? op '*) 1)
        ((equal? op '/) 1)
        ((equal? op 'cons) nil)
        (else (error "wrong init value"))))
```

<<<—BACK—TO—FIRST

<<<——BACK—TO—MYWORK-–LAB6

```
;; Simple evaluator for Scheme without DEFINE, using substitution model.
;; Version 1: No DEFINE, only primitive names are global.

;; The "read-eval-print loop" (REPL):

(define (scheme-1)
  (display "Scheme-1: ")
  (flush)
  (print (eval-1 (read)))
  (scheme-1))

;; Two important procedures:
;; EVAL-1 takes an expression and returns its value.
;; APPLY-1 takes a procedure and a list of actual argument values, and
;;  calls the procedure.
;; They have these names to avoid conflict with STk's EVAL and APPLY,
;;  which have similar meanings.

;; Comments on EVAL-1:

;; There are four basic expression types in Scheme:
;;    1. self-evaluating (a/k/a constant) expressions: numbers, #t, etc.
;;    2. symbols (variables)
;;    3. special forms (in this evaluator, just QUOTE, IF, and LAMBDA)
;;    4. procedure calls (can call a primitive or a LAMBDA-generated procedure)

;; 1.  The value of a constant is itself.  Unlike real Scheme, an STk
;; procedure is here considered a constant expression.  You can't type in
;; procedure values, but the value of a global variable can be a procedure,
;; and that value might get substituted for a parameter in the body of a
;; higher-order function such as MAP, so the evaluator has to be ready to
;; see a built-in procedure as an "expression."  Therefore, the procedure
;; CONSTANT? includes a check for (PROCEDURE? EXP).

;; 2.  In the substitution model, we should never actually evaluate a *local*
;; variable name, because we should have substituted the actual value for
;; the parameter name before evaluating the procedure body.

;; In this simple evaluator, there is no DEFINE, and so the only *global*
;; symbols are the ones representing primitive procedures.  We cheat a little
;; by using STk's EVAL to get the values of these variables.

;; 3.  The value of the expression (QUOTE FOO) is FOO -- the second element of
```

;; the expression.

;; To evaluate the expression (IF A B C) we first evaluate A; then, if A is
;; true, we evaluate B; if A is false, we evaluate C.

;; The value of a LAMBDA expression is the expression itself. There is no
;; work to do until we actually call the procedure. (This won't be true
;; when we write a more realistic interpreter that handles more Scheme
;; features, but it works in the substitution model.)

;; 4. To evaluate a procedure call, we recursively evaluate all the
;; subexpressions. We call APPLY-1 to handle the actual procedure invocation.

```
(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp))    ; use underlying Scheme's EVAL
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
         (if (eval-1 (cadr exp))
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))
        ((lambda-exp? exp) exp)
        ((define-exp? exp)
         (eval (list 'define (cadr exp) (maybe-quote (eval-1 (caddr exp))))))
        ((pair? exp) (apply-1 (eval-1 (car exp))      ; eval the operator
                              (map eval-1 (cdr exp))))
        (else (error "bad expr: " exp))))
```

;; Comments on APPLY-1:

;; There are two kinds of procedures: primitive and LAMBDA-created.

;; We recognize a primitive procedure using the PROCEDURE? predicate in
;; the underlying STk interpreter.

;; If the procedure isn't primitive, then it must be LAMBDA-created.
;; In this interpreter (but not in later, more realistic ones), the value
;; of a LAMBDA expression is the expression itself. So (CADR PROC) is
;; the formal parameter list, and (CADDR PROC) is the expression in the
;; procedure body.

;; To call the procedure, we must substitute the actual arguments for
;; the formal parameters in the body; the result of this substitution is
;; an expression which we can then evaluate with EVAL-1.

```
(define (apply-1 proc args)
  (cond ((procedure? proc)   ; use underlying Scheme's APPLY
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc)   ; the body
                             (cadr proc)    ; the formal parameters
                             args           ; the actual arguments
                             '())))         ; bound-vars, see below
        (else (error "bad proc: " proc))))


;; Some trivial helper procedures:

(define (constant? exp)
  (or (number? exp) (boolean? exp) (string? exp) (procedure? exp)))

(define (exp-checker type)
  (lambda (exp) (and (pair? exp) (eq? (car exp) type))))

(define quote-exp? (exp-checker 'quote))
(define if-exp? (exp-checker 'if))
(define lambda-exp? (exp-checker 'lambda))
(define define-exp? (exp-checker 'define))


;; SUBSTITUTE substitutes actual arguments for *free* references to the
;; corresponding formal parameters.  For example, given the expression
;;
;;      ((lambda (x y)
;;         ((lambda (x) (+ x y))
;;          (* x y)))
;;       5 8)
;;
;; the body of the procedure we're calling is
;;
;;         ((lambda (x) (+ x y))
;;          (* x y))
;;
;; and we want to substitute 5 for X and 8 for Y, but the result should be
;;
;;         ((lambda (x) (+ x 8))
;;          (* 5 8))
;;
;; and *NOT*
```

```
;;
;;          ((lambda (5) (+ 5 8))
;;           (* 5 8))
;;
;; The X in (* X Y) is a "free reference," but the X in (LAMBDA (X) (+ X Y))
;; is a "bound reference."
;;
;; To make this work, in its recursive calls, SUBSTITUTE keeps a list of
;; bound variables in the current subexpression -- ones that shouldn't be
;; substituted for -- in its argument BOUND.  This argument is the empty
;; list in the top-level call to SUBSTITUTE from APPLY-1.

;; Another complication is that when an argument value isn't a self-evaluating
;; expression, we actually want to substitute the value *quoted*.  For example,
;; consider the expression
;;
;;       ((lambda (x) (first x)) 'foo)
;;
;; The actual argument value is FOO, but we want the result of the
;; substitution to be
;;
;;       (first 'foo)
;;
;; and not
;;
;;       (first foo)
;;
;; because what we're going to do with this expression is try to evaluate
;; it, and FOO would be an unbound variable.

;; There is a strangeness in MAYBE-QUOTE, which must handle the
;; case of a primitive procedure as the actual argument value; these
;; procedures shouldn't be quoted.

(define (substitute exp params args bound)
  (cond ((constant? exp) exp)
        ((symbol? exp)
         (if (memq exp bound)
             exp
             (lookup exp params args)))
        ((quote-exp? exp) exp)
        ((lambda-exp? exp)
         (list 'lambda
               (cadr exp)
               (substitute (caddr exp) params args (append bound (cadr exp)))))
```

```
        (else (map (lambda (subexp) (substitute subexp params args bound))
                   exp))))

(define (lookup name params args)
  (cond ((null? params) name)
        ((eq? name (car params)) (maybe-quote (car args)))
        (else (lookup name (cdr params) (cdr args)))))

(define (maybe-quote value)
  (cond ((lambda-exp? value) value)
        ((constant? value) value)
        ((procedure? value) value)   ; real Scheme primitive procedure
        (else (list 'quote value))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Sample evaluation, computing factorial of 5:

; Scheme-1: ((lambda (n)
;              ((lambda (f) (f f n))
;                (lambda (f n)
;                  (if (= n 0)
;                      1
;                      (* n (f f (- n 1))) )) ))
;            5)
; 120

;; Sample evaluation, using a primitive as argument to MAP:

; Scheme-1: ((lambda (f n)
;              ((lambda (map) (map map f n))
;                  (lambda (map f n)
;                    (if (null? n)
;                        '()
;                        (cons (f (car n)) (map map f (cdr n))) )) ))
;            first
;            '(the rain in spain))
; (t r i s)
```

<<<—BACK—TO—FIRST

For reference, see WEEK 4 ExtraForExpert in https://people.eecs.berkeley.edu/~bh/61a-pages/Volume1/hw.pdf OR Exercise2.6 in https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-14.html#%_sec_2.1.3

Description: Arithmetic implemented by Church representation of numbers.

```
STk> (define zero (lambda (f) (lambda (x) x)))       ;;definition of number zero

STk> (define one (lambda (f) (lambda (x) (f x))))   ;;definition of number one

STk> (define two (lambda (f) (lambda (x) (f (f x)))))  ;;definition of number two.

STk> (define (add-1 n)                               ;;definition of add-1
  (lambda (f) (lambda (x) (f ((n f) x)))))

STk> (define (add a b)                               ;;definition of add.
  (lambda (f)
    (lambda (x)
      ((a f) ((b f) x)))))

STk> (define (multiply a b)                          ;;definition of multiply
  ((a (b add-1)) zero))

STk> (define (exponent b n)                          ;;definition of exponent
  (((n b) add-1) zero))

STk> (define (print-num num)            ;; definition of print-num, 1+ is increment
                                        ;;procedure built-in in Scheme.
  ((num 1+) 0))                         ;; 0 is built-in number in Scheme.

STk> (define five (add one (add two two)))     ;;5 = (1 + (2 + 2))
five
STk> (print-num five)                          ;;print the value of five.
5
STk> (print-num (multiply five five))          ;;print the value of five multiply five.
25
STk> (print-num (exponent five two))         ;; print the value of square of five.
25
```

<<<<——BACK—TO—MY—JOURNEY—OF—LISP—PROGRAMMING

<<—BACK TO “My Journal Of Lisp Programming”

I have simplified the narrative by integrating the solution to present an overall picture of this project. Thus, the original text should merely serve as a reference. However, if you find my writing confusing, please refer to https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/psets/ps7/readme.html . I have uploaded source codes for convenience.

The system provided by the project originally can simulate process of arbitrage between an arbitrager(named Nick Leeson) and two markets(Tokyo and Singapore).

```
(start-world)
;Value: running

Nick Leeson     Balance: 1000000000.     Contracts: 0.     audit nick-leeson
Tokyo:Nikkei-225                 16681.455535090536        ticker tokyo
Singapore:Nikkei-225             16681.53583305463         ticker singapore
Nick Leeson     Balance: 1000090000.8151431     Contracts: 0.
Tokyo:Nikkei-225                 16682.74592944109
Singapore:Nikkei-225             16682.5313518813
Nick Leeson     Balance: 1000107887.4073222     Contracts: 0.
Nick Leeson     Balance: 1000107887.4073222     Contracts: 0.
Tokyo:Nikkei-225                 16683.941514305898
Singapore:Nikkei-225             16684.060937094866
Nick Leeson     Balance: 1000107887.4073222     Contracts: 0.
Tokyo:Nikkei-225                 16685.6041972343
Singapore:Nikkei-225             16686.97311483555
Nick Leeson     Balance: 1000207886.838094      Contracts: 0.
Nick Leeson     Balance: 1000217886.7811712     Contracts: 0.
Tokyo:Nikkei-225                 16688.30791615898
```

Running function “start-world” starts the simulation of Nick Leeson(the arbitrager) making profits by buying contracts from lower price market and selling to the higher price market. (See class diagram of market and arbitrager)

The following code of “start–world” shows that it essentially runs all use cases concurrently.(See use case diagram)

```
(define (start-world)
    (if stop-world (stop-world))
    (set! stop-world
          (parallel-execute
           (do-aperiodically (lambda ()
                               (nikkei-update))   key
                             3000)
           (do-aperiodically (lambda ()
                               (nick-leeson 'consider-a-trade))   key
                             4000)
           (do-aperiodically (lambda ()
                               (audit nick-leeson))
                             20000)
           (do-aperiodically (lambda ()
                               (tokyo 'execute-an-order))   key
                             2000)
           (do-aperiodically (lambda ()
                               (singapore 'execute-an-order))   key
                             2500)
           (do-periodically (lambda ()
                              (ticker tokyo))
                            10000)
           (do-periodically (lambda ()
                              (ticker singapore))
                            10000)
           ))
    'running)
```

“do-aperiodically” means thread sleep at most xxxx milliseconds before next invocation.

“do-periodically” means thread repeats every xxxxx milliseconds.

“Start–world” simulate the process of arbitrage repeatedly with property of randomness by specifying aperiodic timing constants (number) of key functions (labeled in red).

`Lab-exercise 1` discusses such simple concurrent program is actually complex.

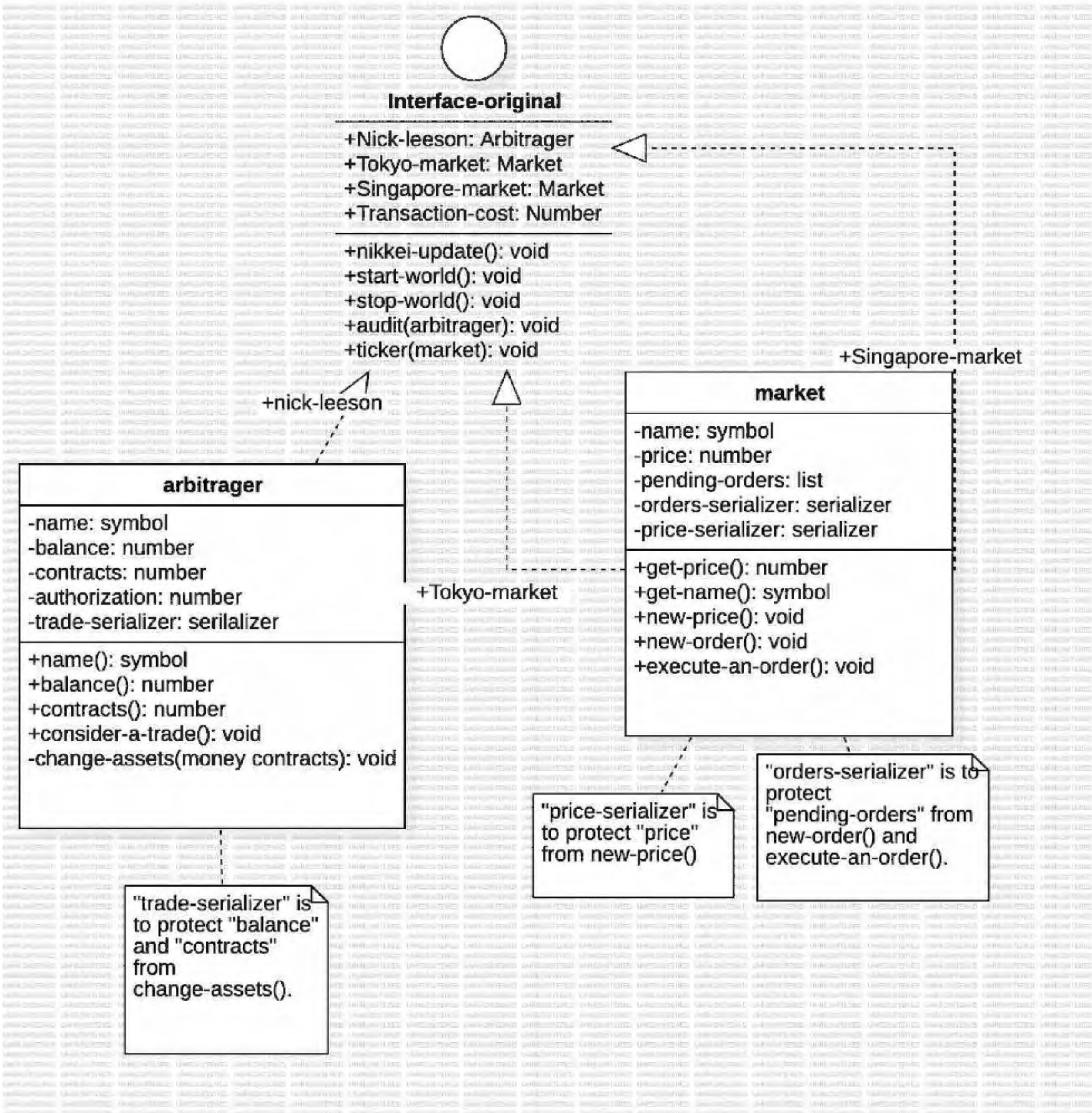
`Lab-exercise 3` uses feedback loop and concurrent data structure to solve race condition efficiently and correctly.

`Lab-exercise 4` briefly explores elementary analysis of data through **polynomial regression** to predict the future value of market price.

This project is based on codes “nikkei–sourceCode0”, “parallel–scm”, “gauss–scm”.

<<—BACK TO FIRST

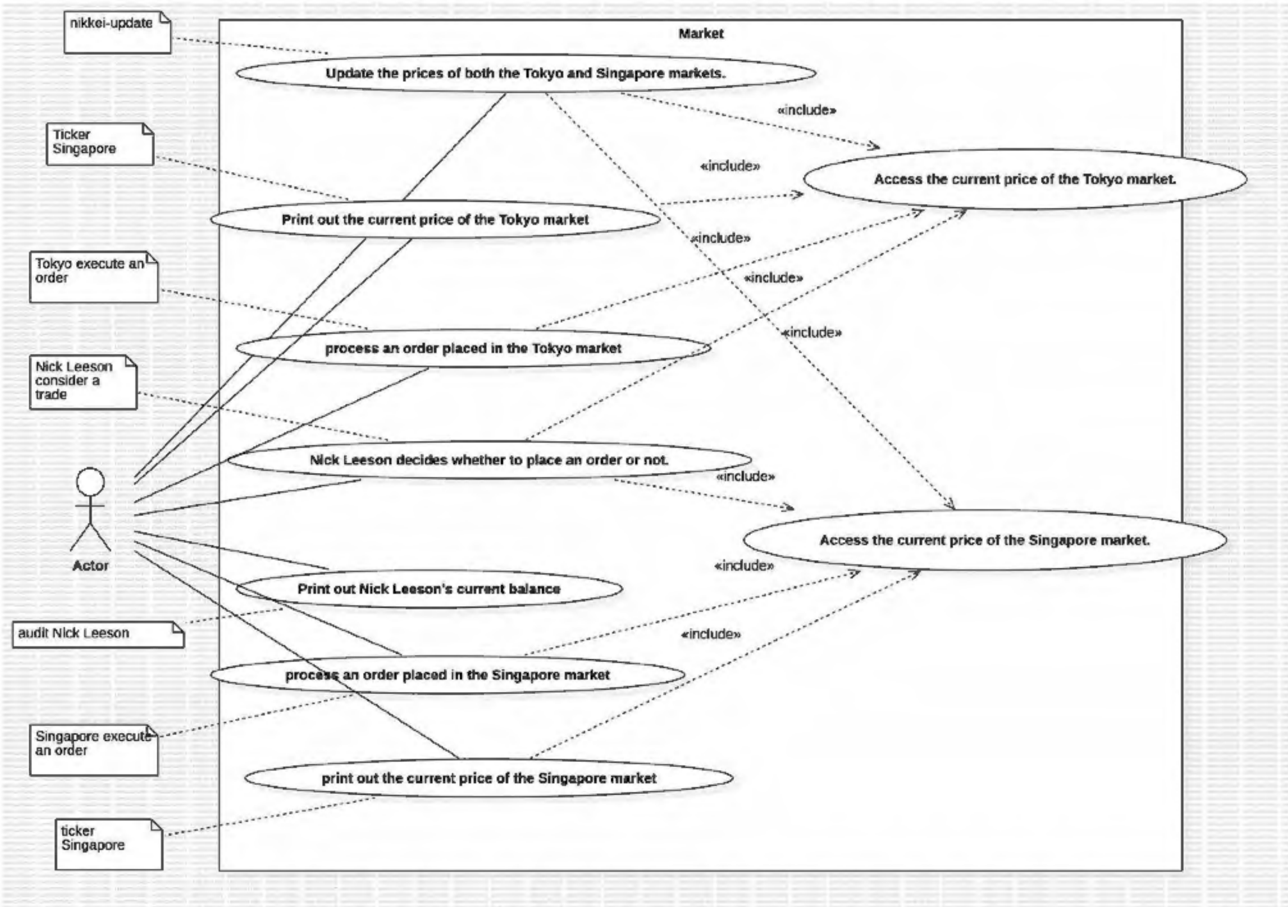
<<—BACK TO CONCURRENCY
<<—BACK TO LAB–3–VERSION–3



Interface codes, arbitrager class codes, market class codes are all in nikkei–souceCode0.

<<<——BACK TO CONCURRENCY TO–PITFALLS–OF–CONCURRENCY—>>>

Note that “nikkei–update” and “consider–a–trade” include accessing both market’s current price, “execute–an–order” and “ticker” include accessing corresponding market’s price.



Nikkei–update sequence diagram: Update the price of both markets (Access 2 market’s price).

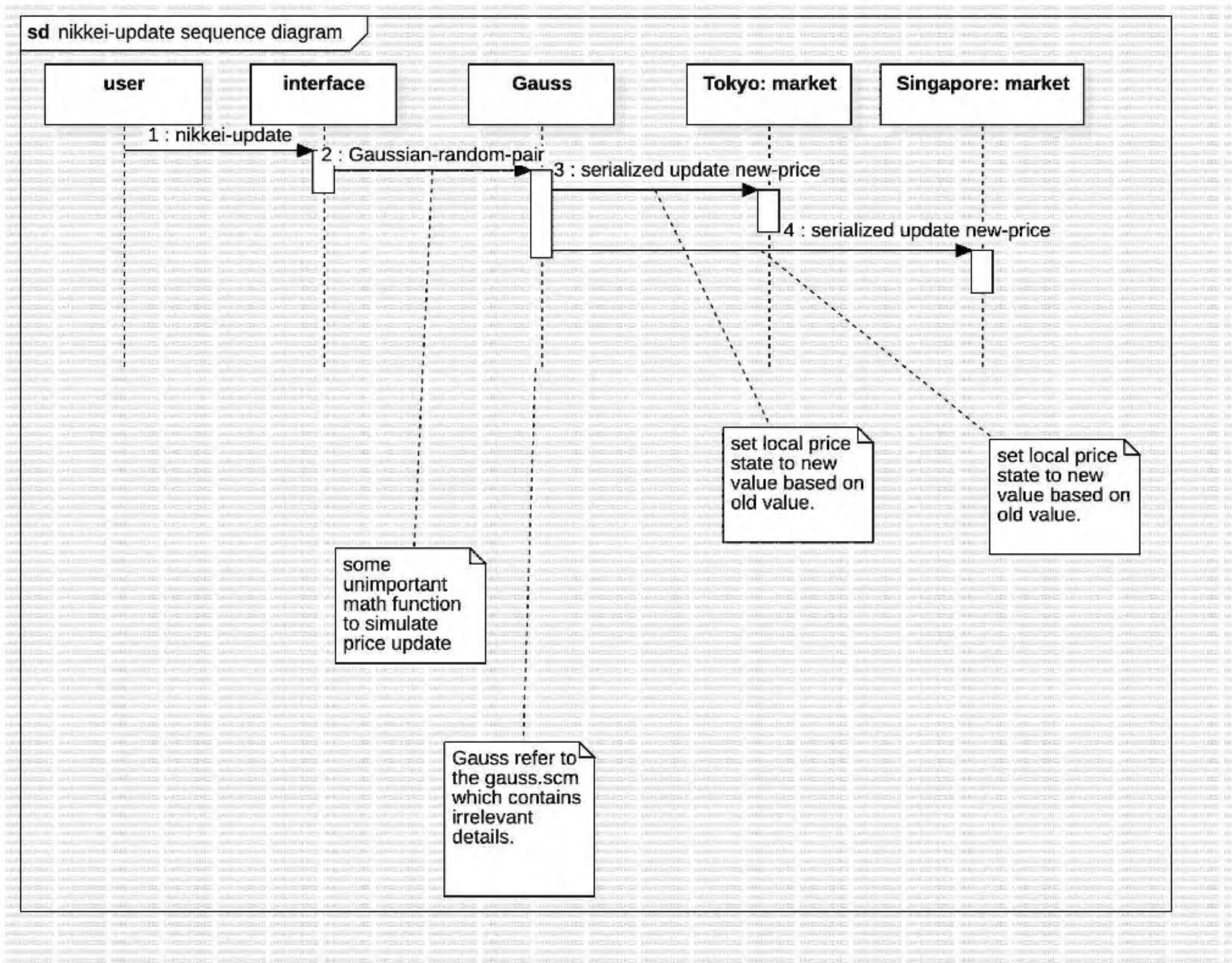
Ticker a market(code): Print out the current price of the market (Access 1 market’s price).

Execute an order sequence diagram: Process an order placed by the market (Access 1 market’s price).

Audit the arbitrager(code): Print out arbitrager’s current balance.

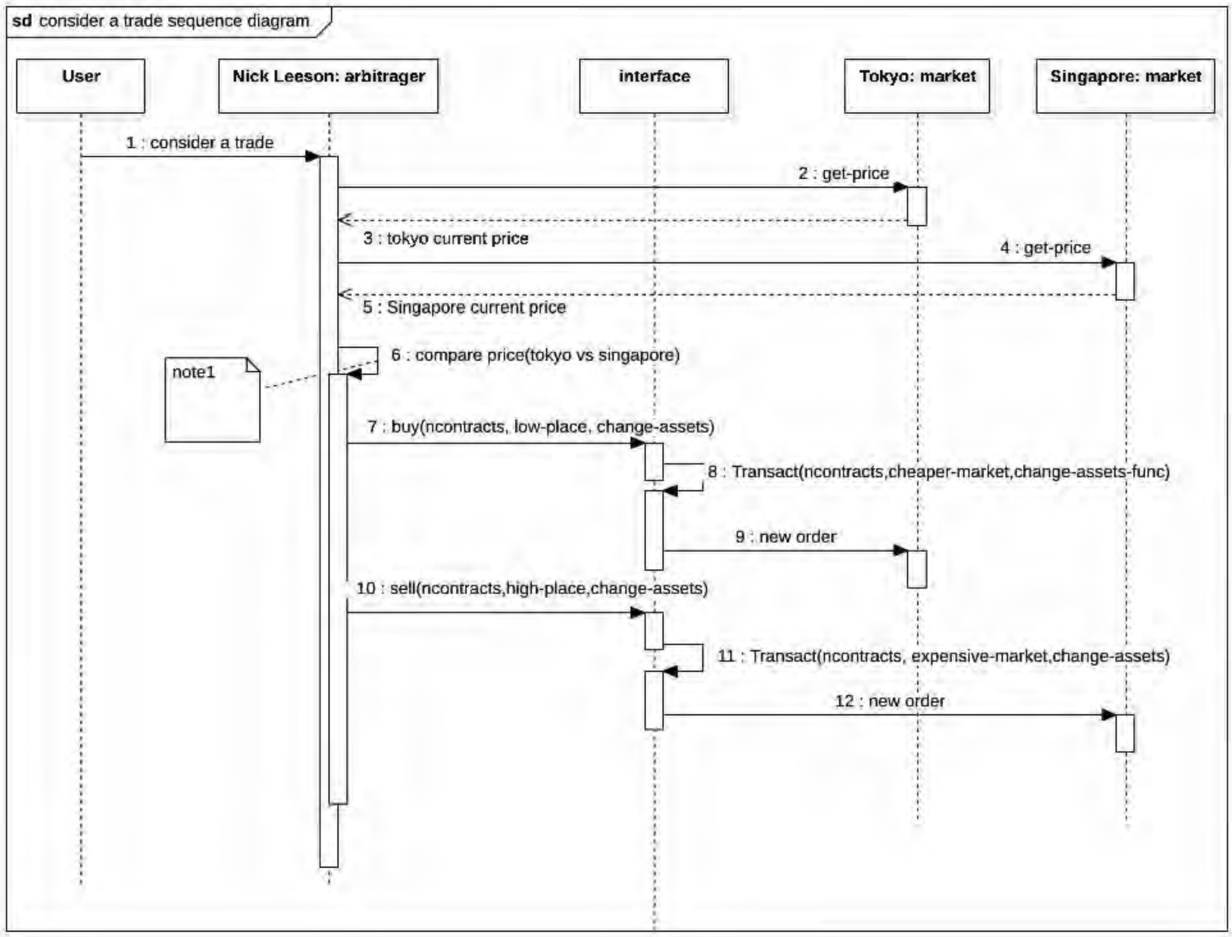
Consider a trade sequence diagram: Arbitrager decides whether to place an order or not (Access 2 market price).

BACK TO USE-CASE-DIAGRAM



This is sequence diagram of nikkei-update function in nikkei-sourcecode0.

This is sequence diagram of consider a trade.
Consider a trade is a method of class arbitrager in nikkei–sourceCode.
<<— GO BACK TO USE–CASE–DIAGRAM



Note1 : Texts in color light blue in nikkei–sourceCode0 shows that consider–trade method compare prices before calling function a<b in order to buy low and sell high.

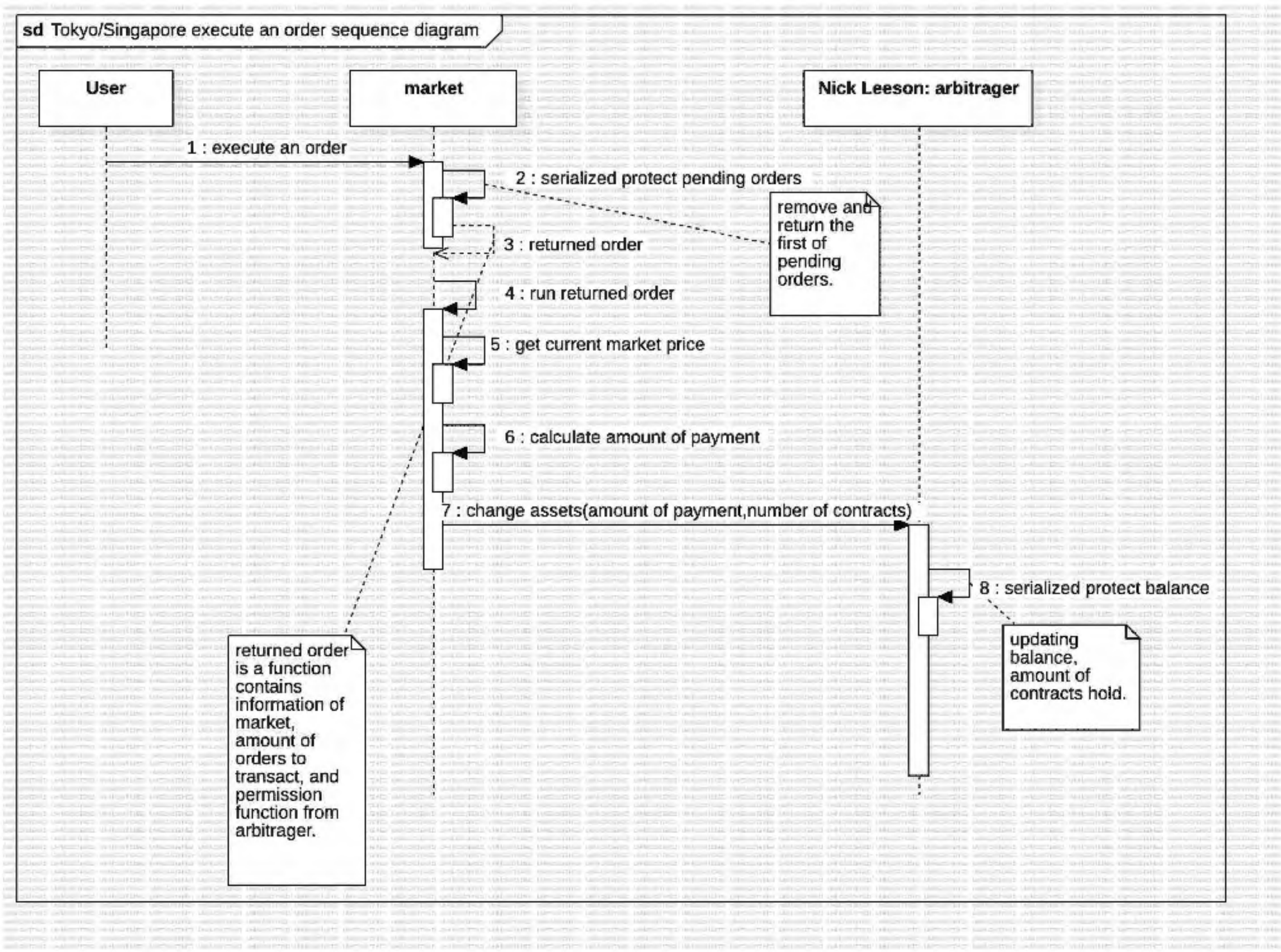
Note2: Texts in color purple in a<b function in nikkei–sourceCode0 shows that "transaction–cost" and "price difference" between two markets determine whether to place an order or not.

Note3: Texts in color red in "a<b" function in nikkei–sourceCode0 shows that ncontracts(amount of contracts to buy/sell) depends on function "minimum(10000, balance)" and "price difference" between two markets.

Example 1: ncontracts = (min 10000 1,000,000,000) / (16688.0 – 16686.5)) = (10000 / 1.5)
= 6667 (buy&sell 6667 contracts in two markets)
Example 2: ncontracts = (min 10000 –1,000,000) / (16689.20 – 16689.19))
= (–1,000,000 / 0.01) = – 100,000,000 (buy&sell –100,000,000 contracts in two markets)

<<—BACK TO WHY EXTREME VALUE OR <<—BACK TO EXECUTE–AN–ORDER DIAGRAM

Execute an order in Tokyo/Singapore market:



Note1: Lines in red within function “transact” in nikkei-sourceCode0 represent the codes of the order(function) returned from step3 above.

Note2: Function “permission” in red within function “transact” in nikkei-sourceCode0 is the step 7 above arbitrager class’s method “change assets” (line in green in nikkei-sourceCode0) that will change Nick Leeson’s balance.

Note3: “Ncontracts” in red stored in the returned order(as note1 above mentioned) within “transact” in nikkei-sourceCode0 was passed from function “buy” or function “sell” which are invoked from “a<b” by arbitrager class (see consider-trade-sequence-diagram/note3).

Note4: If “Ncontracts” computed by “a<b” is negative (e.g. consider-trade-sequence-diagram/note3/example2), the original intention of “buy low and sell high” (see consider-trade-diagram-sequence-diagram/note1) end up as “buy high and sell low” . See note2&3 above for explanation.

TO USE-CASE-DIAGRAM

Lab Exercise 1: Run the world for a bit. Does Mr. Leeson seem to lose money, gain money, or break even on the average? The time constants for the way the world runs are the numbers (of milliseconds) occuring in the procedure `start-world`. Also, Mr. Leeson's strategy depends on the value of the constant `transaction-cost`. Which of these constants could you change to make arbitrage extremely profitable? How do you think transaction-cost interacts with the timing constants? To support your argument, make a change and demonstrate the improved profits.

<<<——BACK TO CONCURRENCY

The row in red in the table below shows that the combination of timing constant within the function “start–world” set up by the project originally (as shown in the overview) causes the arbitrager’s balance to reach 1,000,399,129 after running a 2–minute simulation starting from a balance of 1,000,000,000 (All test cases in the table starting from the same balance value).

As the table shows, arbitrager’s balance can remain at 3 different states:
1. Moderate profit/break even.
2. Huge lost.
3. Huge profit.

transaction cost interact with random/constant repetition rate of 4 key functions

	Nikkei updates (repeat in milliseconds)	Consider trades (repeat in milliseconds)	Tokyo execution (repeat in milliseconds)	Singapore execution (repeat in milliseconds)		
Transaction cost = 1	Aperiodically 3000	Aperiodically 4000	Aperiodically 2000	Aperiodically 2500	Balance = 1,000,399,129	2 minutes
"	Aperiodically 3000	Aperiodically 100	Aperiodically 100	Aperiodically 700	Balance = –2.00e11	1 minute
"	Aperiodically 30	Aperiodically 10	Aperiodically 10	Aperiodically 50	Balance = 1,005,718,709	2 minutes
"	Periodically 30	Periodically 10	Periodically 10	Periodically 50	Balance = 1,008,338,837	2 minutes
Transaction cost = 0.1	Aperiodically 3000	Aperiodically 100	Aperiodically 100	Aperiodically 100	Balance = 1.14e139	Not recorded
Transaction cost = 1e–11	Aperiodically 3000	Aperiodically 2000	Aperiodically 2000	Aperiodically 4000	Balance = –6.03e14	4 minutes 30 seconds
"	Aperiodically 300	Aperiodically 2000	Aperiodically 100	Aperiodically 500	Balance = 1,003,396,966	3 minutes 30 seconds
"	Aperiodically 2	Aperiodically 2	Aperiodically 2	Aperiodically 2	Balance = 5.51e77	16 seconds

Transaction cost = 1e−14	Aperiodically 30	Aperiodically 10	Aperiodically 10	Aperiodically 50	Balance = 5.55e16	30 seconds
"	Periodically 30	Periodically 10	Periodically 10	Periodically 50	Balance = −1.10e22	Not recorded
					Nick Leeson's balance	Time taken

Yellow table cell indicate "Constant repetition rate" as opposed to "Random repetition rate" (Periodic vs Aperiodic)

In order to figure out what caused Nick Leeson's balance to reach an anomalous result, instead of reasoning from occasional print-outs (e.g. ticker, audit within "start-world" in overview), **one must insert printing among 4 key functions to print out all state's value after each update and then examine the print-outs line by line.**

Here we have a new version of a "start-world" within nikkei-sourceCode1 which can print out much richer information about the transaction as shown in new-display, after examination of new-display we can see the 3 different states of the arbitrager's balance are in temporal order (trivial details are in whyExtremeValue).

Overall, certain combinations do contribute to certain outcomes, but with a slight change in settings, a different result (categorized in 3 different states above) would be produced. In general, the correlation between combinations of timing constants and the outcome is complex, and this is probably what is called "nondeterministic computation".

Note: Colorful lines are used for explanation in WHY EXTREME VALUE.

– – – – – – – – –phase 1 –
– – small amount (< 10000) transaction –
– –
– –
– –

Nick Leeson Balance: 1002264537.0877643 Contracts: 0.
tokyo execute–an–orderTransact 4826. contracts at 16789.115554665015
tokyo–remain–orders––:1
Nick Leeson Balance: 921240265.4209509 Contracts: 4826.
singapore execute–an–orderTransact –4826. contracts at 16788.607589051226
singapore–remain–orders––:1
Nick Leeson Balance: 1002262085.6457121 Contracts: 0.

– – – – – – – – phase 2– –
– Big amount orders * market price > balance – – – – – – – – – – – – – – – – – – –
– –
– –
– – – – – – – <<<—GO BACK TO WHY EXTREME VALUE——
nikkei–update
Tokyo:Nikkei–225 16789.02853660443
Singapore:Nikkei–225 16789.03120803015
singapore execute–an–orderTransact 19686. contracts at 16789.03120803015
singapore–remain–orders––:0
Nick Leeson Balance: 671753217.2844306 Contracts: 19686.
tokyo execute–an–orderTransact –19686. contracts at 16789.02853660443
tokyo–remain–orders––:0
Nick Leeson Balance: 1002262033.0560255 Contracts: 0.
nick–leeson consider–a–trade
consider buy 3743320. contracts at price 16789.02853660443 in Tokyo:Nikkei–225
consider sell 3743320. contracts at price 16789.03120803015 in Singapore:Nikkei–225
tokyo–remain–orders––:1
singapore–remain–orders––:1
Nick Leeson Balance: 1002262033.0560255 Contracts: 0.
nikkei–update
Tokyo:Nikkei–225 16789.677085858268
Singapore:Nikkei–225 16791.31447626734
tokyo execute–an–orderTransact 3743320. contracts at 16789.677085858268
tokyo–remain–orders––:0
Nick Leeson Balance: –61846871995.97895 Contracts: 3743320.
nick–leeson consider–a–trade
consider buy –37771610029. contracts at price 16789.677085858268 in Tokyo:Nikkei–225

consider sell -37771610029. contracts at price 16791.31447626734 in Singapore:Nikkei-225
tokyo-remain-orders--:1
singapore-remain-orders--:2
Nick Leeson Balance: -61846871995.97895 Contracts: 3743320.
nikkei-update

- - - - - - - -phase 3 -
- price update(in purple) between executions cause negative balance to turn positive
- -
- -
nick-leeson consider-a-trade
consider buy -1.7153466341524468e19 contracts at price 16797.965080316477 in Tokyo:Nikkei-225
consider sell -1.7153466341524468e19 contracts at price 16799.350208740518 in Singapore:Nikkei-225
tokyo-remain-orders--:2
singapore-remain-orders--:2
Nick Leeson Balance: -2.3759753800474165e19 Contracts: 0.
singapore execute-an-orderTransact 4.204012881938181e23 contracts at 16799.350208740518
singapore-remain-orders--:1
Nick Leeson Balance: -7.062468492333354e27 Contracts: 4.204012881938181e23
tokyo execute-an-orderTransact -4.204012881938181e23 contracts at 16797.965080316477
tokyo-remain-orders--:1
Nick Leeson Balance: -5.82333533534131e23 Contracts: 0.
nikkei-update
Tokyo:Nikkei-225 16800.722716577468
Singapore:Nikkei-225 16800.74514393001
tokyo execute-an-orderTransact -1.7153466341524468e19 contracts at 16800.722716577468
tokyo-remain-orders--:0
Nick Leeson Balance: -2.941429019020339e23 Contracts: -1.7153466341524468e19
nick-leeson consider-a-trade
consider buy -1.3115364435151438e25 contracts at price 16800.722716577468 in Tokyo:Nikkei-225
consider sell -1.3115364435151438e25 contracts at price 16800.74514393001 in Singapore:Nikkei-225
tokyo-remain-orders--:1
singapore-remain-orders--:2
Nick Leeson Balance: -2.941429019020339e23 Contracts: -1.7153466341524468e19
singapore execute-an-orderTransact 1.7153466341524468e19 contracts at 16800.74514393001
singapore-remain-orders--:1
Nick Leeson Balance: -5.82333918240968e23 Contracts: 0.

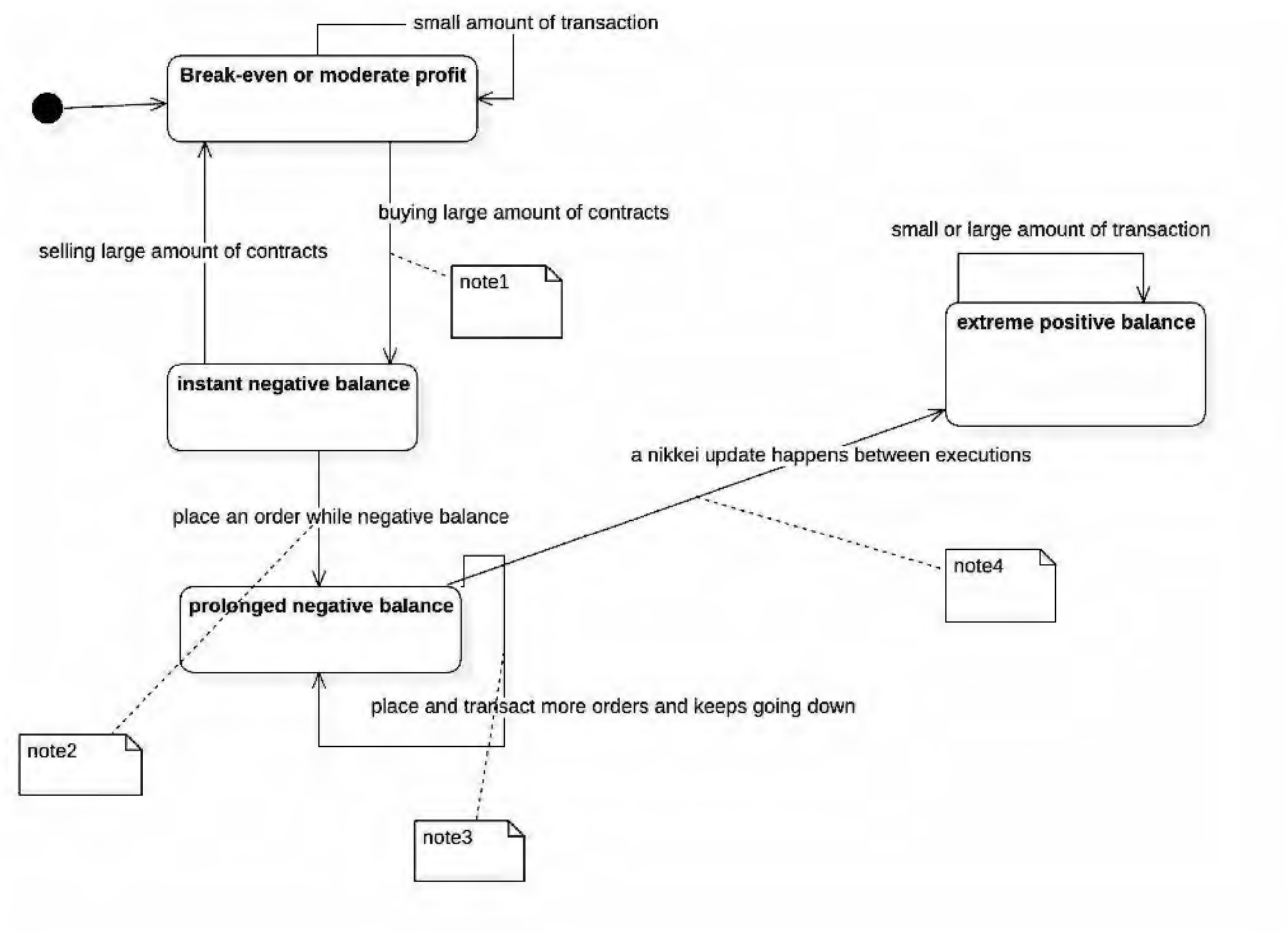
singapore execute–an–orderTransact 1.3115364435151438e25 contracts at 16800.74514393001
singapore–remain–orders––:0
Nick Leeson   Balance: –2.2034847767866113e29   Contracts: 1.3115364435151438e25
nikkei–update
Tokyo:Nikkei–225   16802.450755832535
Singapore:Nikkei–225   16801.443155802248
tokyo execute–an–orderTransact –1.3115364435151438e25 contracts at 16802.450755832535
tokyo–remain–orders––:0
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 0.

– – – – – – – – – – – phase 4 –
– – Big amount of orders are no longer able to outweigh huge balance – – – – – – – – – –
– – Balance stays at huge number –
– –

nick–leeson consider–a–trade
consider buy 9925. contracts at price 16801.443155802248 in Singapore:Nikkei–225
consider sell 9925. contracts at price 16802.450755832535 in Tokyo:Nikkei–225
tokyo–remain–orders––:1
singapore–remain–orders––:1
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 0.
singapore execute–an–orderTransact 9925. contracts at 16801.443155802248
singapore–remain–orders––:0
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 9925.
nick–leeson consider–a–trade
consider buy 9925. contracts at price 16801.443155802248 in Singapore:Nikkei–225
consider sell 9925. contracts at price 16802.450755832535 in Tokyo:Nikkei–225
tokyo–remain–orders––:2
singapore–remain–orders––:1
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 9925.
nikkei–update
Tokyo:Nikkei–225   16802.938720638394
Singapore:Nikkei–225   16802.094806618315
tokyo execute–an–orderTransact –9925. contracts at 16802.938720638394
tokyo–remain–orders––:1
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 0.
singapore execute–an–orderTransact 9925. contracts at 16802.094806618315
singapore–remain–orders––:0
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 9925.
nick–leeson consider–a–trade
consider buy 11850. contracts at price 16802.094806618315 in Singapore:Nikkei–225

<<<——BACK-TO-LAB-EXERCISE-1

consider sell 11850. contracts at price 16802.938720638394 in Tokyo:Nikkei-225
tokyo-remain-orders--:2
singapore-remain-orders--:1
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 9925.
nikkei-update
Tokyo:Nikkei-225   16805.50541861005
Singapore:Nikkei-225   16804.92179196282
tokyo execute-an-orderTransact -9925. contracts at 16805.50541861005
tokyo-remain-orders--:1
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 0.
singapore execute-an-orderTransact 11850. contracts at 16804.92179196282
singapore-remain-orders--:0
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 11850.
tokyo execute-an-orderTransact -11850. contracts at 16805.50541861005
tokyo-remain-orders--:0
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 0.
nick-leeson consider-a-trade
consider buy 17134. contracts at price 16804.92179196282 in Singapore:Nikkei-225
consider sell 17134. contracts at price 16805.50541861005 in Tokyo:Nikkei-225
tokyo-remain-orders--:1
singapore-remain-orders--:1
Nick Leeson   Balance: 2.1787387768294334e25 Contracts: 0.



Note1: Low transaction cost (<= 0.1) is key to large amount transaction. See consider–trade–sequence–diagram/note2 & note3 and new–display phase2 line in pink.

Note2: Placing orders(consider–a–trade) while balance is negative (see pink line in phase2 of new–display) will cause placing negative amount of orders (see orange lines in phase2 of new–display) due to the reason in consider–trade–sequence–diagram/note3/example2 .

Note3: Executing negative amount of orders will further decrease the balance because the execution is essentially "buying high and selling low". For detail reasons see execute–an–order diagram/note4.

Note4: Purple lines in phase3 of the new–display shows a price update occurs between the execution in Singapore market (lines in red) and execution in Tokyo market (lines in green), causing the total value of transaction in Tokyo market to outweigh the decreasing momentum (as mentioned in note 3 above), and thus remain at an extremely positive balance (phase 4 in new–display).

<<<——BACK–TO–LAB–EXERCISE 1

<<<——TO-NIKKEI-SOURCECODE-0 TO-NIKKEI-SOURCECODE-2—>>>

```
;;;;       England lost her Barings

;;; The collapse of the Baring Brothers establishment
;;;  provides an abject Leeson for the banking community.
;;;  -- GJS

(define (make-market name initial-price)
  (let ((price initial-price)
        (price-serializer (make-serializer))
        (pending-orders '())
        (pending-order-num 0)
        (orders-serializer (make-serializer)))
    (define (the-market m)
      (cond ((eq? m 'new-price!)
             (price-serializer
                  (lambda (update)
                    (set! price (update price)))))
            ((eq? m 'get-price)
             price)
            ((eq? m 'new-order!)
             (orders-serializer
              (lambda (new-order)
                (set! pending-order-num (+ 1 pending-order-num))
                (set! pending-orders
                      (append pending-orders
                              (list new-order))))))
            ((eq? m 'execute-an-order)
             (((orders-serializer
                (lambda ()
                  (if (not (null? pending-orders))
                      (let ((outstanding-order (car pending-orders)))
                        (set! pending-orders (cdr pending-orders))
                        (set! pending-order-num (- pending-order-num 1))
                        outstanding-order)
                      (lambda () 'nothing-to-do)))))))
            ((eq? m 'get-name) name)
            ((eq? m 'reset-order-num)
             ((orders-serializer (lambda () (set! pending-order-num 0)))))
            ((eq? m 'get-order-num)
             ((orders-serializer (lambda () pending-order-num))))
            (else
             (error "Wrong message" m))))
    the-market))
```

```
;;; Using this we make two markets, both starting at the same price:

(define nikkei-fundamental 16680.0)

(define tokyo
  (make-market "Tokyo:Nikkei-225" nikkei-fundamental))

(define singapore
  (make-market "Singapore:Nikkei-225" nikkei-fundamental))




;;; Traders buy and sell contracts at a market using TRANSACT.
;;;  The trader gives the market permission to subtract from the
;;;  trader's monetary balance the cost of the contracts purchased
;;;  and to add to the trader's stash the contracts he purchased.

(define (buy ncontracts market permission arbitrager-name)  ;;note-on-lab-exercise2.txt
  (transact ncontracts market permission arbitrager-name))

(define (sell ncontracts market permission arbitrager-name)  ;;note-on-lab-exercise2.txt
  (transact (- ncontracts) market permission arbitrager-name))

(define (transact ncontracts market permission arbitrager-name) ;;note-on-lab-exercise2.txt
  ((market 'new-order!)
   (lambda ()
     (let ((current-price (market 'get-price)))
       (newline)
       (display "Transact ")
       (display arbitrager-name)
       (display "'s order: ")
       (display ncontracts)
       (display " contracts at ")
       (display current-price)
       (permission (- (* ncontracts current-price))
                   ncontracts)))))




(define transaction-cost 1.0)
```

```
(define (make-arbitrager name balance contracts authorization)
 (let ((trader-serializer (make-serializer)))

  (define (change-assets delta-money delta-contracts)
   ((trader-serializer
     (lambda ()
      (set! balance (+ balance delta-money))
      (set! contracts (+ contracts delta-contracts))))))

  (define (a<b low-place low-price high-place high-price)
   (if (> (- high-price low-price) transaction-cost)
      (let ((amount-to-gamble (min authorization balance)))
       (let ((ncontracts             ;round to nearest integer
               (round (/ amount-to-gamble (- high-price low-price)))))
        (buy ncontracts low-place change-assets name)
        (newline)
        (display name)
        (display " consider buy ")
        (display ncontracts)
        (display " contracts at price ")
        (display low-price)
        (display " in ")
        (display (low-place 'get-name))
        (sell ncontracts high-place change-assets name)
        (newline)
        (display name)
        (display " consider sell ")
        (display ncontracts)
        (display " contracts at price ")
        (display high-price)
        (display " in ")
        (display (high-place 'get-name))
        ))))

  (define (consider-a-trade)
   (let ((nikkei-225-tokyo (tokyo 'get-price))
         (nikkei-225-singapore (singapore 'get-price)))
     (if (< nikkei-225-tokyo nikkei-225-singapore)
       (a<b tokyo nikkei-225-tokyo
            singapore nikkei-225-singapore)
       (a<b singapore nikkei-225-singapore
            tokyo nikkei-225-tokyo))))
```

<<<——BACK—TO—FIRST

```
  (define (me message)
    (cond ((eq? message 'name) name)
          ((eq? message 'balance) balance)
          ((eq? message 'contracts) contracts)
          ((eq? message 'consider-a-trade) (consider-a-trade))
          (else
           (error "Unknown message -- ARBITRAGER" message))))
  me))

(define nick-leeson
  (make-arbitrager "Nick Leeson" 1000000000. 0.0 10000.))
```

```scheme
;;; The following parameters determine the way the Nikkei average
;;;  drifts over time, and how the two markets differ.  The details of
;;;  this are probably not very much like a real market, and they
;;;  probably do not matter very much for this problem set.

(define nikkei-drift +0.1)
(define nikkei-split +1.0)

(define (nikkei-update)
```

BACK TO PITFALL1 ——>>>

```scheme
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian-random-pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei-fundamental
           (+ nikkei-fundamental
              nikkei-drift
              (* nikkei-split (+ d1 d2))))
     ;; The actual split -- cannot be modified
     ((tokyo 'new-price!)
      (lambda (old-price)
         (+ (average nikkei-fundamental old-price)
            (* nikkei-split d1))))
     ((singapore 'new-price!)
      (lambda (old-price)
            (+ (average nikkei-fundamental old-price)
               (* nikkei-split d2)))))))

(define (ticker market)
  (newline)
  (display (market 'get-name))
  (display "              ")
  (display (market 'get-price)))

(define (audit trader)
  (newline)
  (display (trader 'name))
  (display "     Balance: ")
  (display (trader 'balance))
  (display "     Contracts: ")
  (display (trader ‘contracts)))
```

<<<——BACK—TO—FIRST

```
;;; Delay times are in milliseconds.
(define (do–aperiodically thunk max–delay)
  (define (again)
    (thunk)
    (sleep–current–thread (random max–delay))
    (again))
  again)

(define (do–periodically thunk time)
  (define (again)
    (thunk)
    (sleep–current–thread time)
    (again))
  again)


;;; Students: Please ignore the following hairy expression.
;;;  It is only here to allow us to reload this file into a
;;;  running simulation.
(if (lexical–unreferenceable? user–initial–environment 'stop–world)
    (define stop–world false))

;;; The following starts a bunch of threads running in parallel
;;;  and makes a procedure to stop all of the parallel threads.

(define (start–world)
    (if stop–world (stop–world))
    (set! stop–world
            (parallel–execute
             (do–aperiodically (lambda ()
                                 (newline)
                                 (display "nikkei–update")
                                 (nikkei–update)
                                 (ticker tokyo)
                                 (ticker singapore))
                               3000)
             (do–aperiodically (lambda ()
                                 (newline)
                                 (display "nick–leeson consider–a–trade")
                                 (nick–leeson 'consider–a–trade)
                                 (tokyo–remain–orders)
                                 (singapore–remain–orders)
                                 (audit nick–leeson))
                               4000)
```

```
          (do–aperiodically (lambda ()
                              (newline)
                              (display "tokyo execute–an–order")
                              (tokyo 'execute–an–order)
                              (tokyo–remain–orders)
                              (audit nick–leeson)
                              )
                            2000)
          (do–aperiodically (lambda ()
                              (newline)
                              (display "singapore execute–an–order")
                              (singapore 'execute–an–order)
                              (singapore–remain–orders)
                              (audit nick–leeson)
                              )
                            2500)

    'running)


(define (tokyo–remain–orders)
  (newline)
  (display "tokyo–remain–orders––:")
  (display (tokyo 'get–order–num)))

(define (singapore–remain–orders)
  (newline)
  (display "singapore–remain–orders––:")
  (display (singapore 'get–order–num)))
```

<<<——BACK—TO—LAB–EXERCISE1

Lab Exercise 3: You will probably see a case where the parallel interleaving of the process threads fouls up the I/O, mixing the characters that are output by a ticker and the auditor. Can you explain this? Can you figure our a way to fix it? Write the code required to fix this bug. Be prepared to argue to your tutor that your code fixes exactly this bug and has no other consequences (such as preventing two markets from processing orders simultaneously.) (Hint: You can use a serializer, carefully.)

“Display-error” shows that racing condition can occur when printing information calling function “ticker” and “audit” in parallel as overview does. One problem I encountered is that concurrency bugs are not easy to reproduce. “Reproducing concurrency bugs” discuss more details.

By wishful thinking, it is intuitive to serialize all displaying (as version 1 does) to solve “display-error”, but serialization failed as an abstraction tool when the problem of multiple shared resources occurred. Pitfalls of Concurrency discusses sharing in further detail.

We can solve the problem of multiple shared resources by nesting relevant serializers as version 2 does. However, if we need to display not a few lines but hundreds of lines, this solution prevents two markets from processing orders simultaneously as question above reminded.

Inspired by the OOP project (also by the textbook's footnote 51), version 3 attempts to utilize the global time to store past values of state variables for later retrieval in order to delay the process of displaying messages and hopes for an improvement in efficiency. However, this solution does not guarantee an improvement in efficiency, and it violates the constraint that the question above asked for, "use a serializer" (version 3 uses 4), but this solution does lay the foundation for the final version.

Version 4 reduces the use of serializer to one, and exposes the need to change the data structure and the need to adapt the rate control algorithm to improve efficiency.

The final version uses a feedback loop and a concurrent data structure to solve Lab-Exercise 3 correctly and efficiently using one serializer and contains a proof.

<<<——BACK—TO-CONCURRENCY

<<<——BACK-TO-LAB-EXERCISE-3 TO-VERSION-2—>>>>>>

In lab-exercise 1, we showed that there was a need for displaying more information. *1*

In order to solve display-error, my first intuition is to protect(in red) displaying lines within start-world in nikkei-sourceCode2 as below:

```
(define (start-world)
    (if stop-world (stop-world))
    (set! stop-world
            (parallel-execute
             (do-aperiodically (lambda ()
                                (serialized-nikkei-update)  ;;why use serialized nikkei-update see
                                                             PITFALL1—>>>>>>
  (  (serialized (lambda () (newline)
                                (display “nikkei-update")      *1*
                                (ticker tokyo)
                                (ticker singapore))
                 ) ) )
                               3000)
             (do-aperiodically (lambda ()
                                (nick-leeson 'consider-a-trade)   TO-PITFALL2—>>>


         (  (serialized (lambda ()
                                (newline)
                                (display "nick-leeson consider-a-trade”)    *1*
                                (tokyo-remain-orders)
                                (singapore-remain-orders)
                                (audit nick-leeson))
                ) ) )
                               4000)

             (do-aperiodically (lambda ()
                                (tokyo ‘execute-an-order)    TO—PITFALL3——>>>
        (  (serialized (lambda ()
                                (newline)
                                (display "tokyo execute-an-order”)      *1*
                                (tokyo-remain-orders)
                                (audit nick-leeson)
                                )
               ) ) )
                               2000)
             (do-aperiodically (lambda ()
                                (singapore ‘execute-an-order)   TO—PITFALL3——>>>
```

```
(  (serialized (lambda ()
                     (newline)
                     (display "singapore execute–an–order”)    *1*
                     (singapore–remain–orders)
                     (audit nick–leeson)
                     )
          ) ) )
                    2500)

 ‘running)
```

<<<—BACK–TO–FIRST

<<<——BACK-TO-PITFALLS-OF-CONCURRENCY

This code snippet in nikkei-sourceCode1 has risk of race condition. Which violates the assumption stated in "pitfalls-of-concurrency" that "Tokyo and Singapore markets update prices simultaneously".

```
(define (nikkei-update)
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian-random-pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei-fundamental
           (+ nikkei-fundamental
              nikkei-drift
              (* nikkei-split (+ d1 d2))))
     ;; The actual split -- cannot be modified
     ((tokyo 'new-price!)
      (lambda (old-price)
        (+ (average nikkei-fundamental old-price)
           (* nikkei-split d1))))
                                                       <<<<——multiple shared resources problem occurs:
                                                       <<<<——"consider-a-trade" access
                                                       <<<<—— updated Tokyo-price and
                                                       <<<<—— un-updated  Singapore-price.
                                                       <<<<——See consider-trade code below

     ((singapore 'new-price!)
      (lambda (old-price)
           (+ (average nikkei-fundamental old-price)
              (* nikkei-split d2)))))))
```

Now nikkei-sourceCode1 must change to nikkei-sourceCode2 where Tokyo market and Singapore market export their internal "price-serializer" to protect whole process of function "nikkei-update"(*1*) and partial process of arbitrager's method "consider-a-trade"(*2*).

1

```
(define serialized-nikkei-update
  ((tokyo 'price-serializer)
   ((singapore 'price-serializer)
    nikkei-update)
    )
  )
```

2

```
(define (consider-a-trade)
      (let* ((price-pair ( ((tokyo 'price-serializer)
                           ((singapore 'price-serializer) (lambda () (cons ((tokyo 'get-price))
                                                                           ((singapore 'get-price))))))
                           )
                         )
             (nikkei-225-tokyo (car price-pair))
             (nikkei-225-singapore (cdr price-pair)))
        (if (< nikkei-225-tokyo nikkei-225-singapore)
            (a<b tokyo nikkei-225-tokyo
                 singapore nikkei-225-singapore)
            (a<b singapore nikkei-225-singapore
                 tokyo nikkei-225-tokyo))))

  (define (me message)
    (cond ((eq? message 'name) name)
          ((eq? message 'balance) balance)
          ((eq? message 'contracts) contracts)
          ((eq? message 'consider-a-trade) (consider-a-trade))
          (else
           (error "Unknown message -- ARBITRAGER" message))))
  me))
```

<<<——BACK-TO-PITFALLS-OF-CONCURRENCY

In order to solve “display-error”, we need to serialize all displaying message in “start-world”(version1) , and we also need to serialize any displaying within any procedures.

This sub procedure “a<b” is called within “consider-a-trade”, and “consider-a-trade” is invoked from “start-world” function in version1(purple line)

```
 (define (a<b low-place low-price high-place high-price)
     (if (> (- high-price low-price) transaction-cost)
         (let ((amount-to-gamble (min authorization balance)))
           (let ((ncontracts            ;round to nearest integer
                  (round (/ amount-to-gamble (- high-price low-price)))))
             (buy ncontracts low-place change-assets name)
(serialized (lambda ()
             (newline)
             (display name)
             (display " consider buy ")
             (display ncontracts)
             (display " contracts at price ")
             (display low-price)
             (display " in ")
             (display (low-place 'get-name))
 ) )
             (sell ncontracts high-place change-assets name)
(serialized (lambda ()
             (newline)
             (display name)
             (display " consider sell ")
             (display ncontracts)
             (display " contracts at price ")
             (display high-price)
             (display " in ")
             (display (high-place 'get-name))
) )
             ))))
```

However, this solution still has a problem: it can not guarantee the correct reflection of temporal order of events.

Incorrect temporal order of displaying message:
“Nick-leeson consider buy 3456 contracts at price: 16899.64 in Tokyo”

"Nick–leeson consider sell 3456 contracts at price: 16900.21 in Singapore"
"Nikkei–update– Tokyo: 16899.64 Singapore: 16900.21"
Where arbitrager considers prices at price X,Y earlier than markets' price update to price X,Y.

Correct temporal order of display:
"Nikkei–update– Tokyo: 16899.64 Singapore: 16900.21"
"Nick–leeson consider buy 3456 contracts at price: 16899.64 in Tokyo"
"Nick–leeson consider sell 3456 contracts at price: 16900.21 in Singapore"

OR

"Nick–leeson consider buy 3456 contracts at price: 16899.64 in Tokyo"
"Nick–leeson consider sell 3456 contracts at price: 16900.21 in Singapore"
"Nikkei–update– Tokyo: 16900.00 Singapore: 16901.22"
Where markets' have updated prices right after arbitrager considering a deal.

<<<——BACK—TO–PITTFALLS–OF–CONCURRECY

In order to solve problem of display–error, we need to serialize the messages displaying within a sub function "transact", which is called within "execute–an–order", and "execute–an–order" invoked from "start–world" in version1.

```
(define (transact ncontracts market permission arbitrager–name)
 ((market 'new–order!)
  (lambda ()
    (let ((current–price (((market 'price–serializer) (market 'get–price))))
(serialized (lambda ()
       (newline)
       (display "Transact ")
       (display arbitrager–name)
       (display "'s order: ")
       (display ncontracts)
       (display " contracts at ")
       (display current–price)
) )
       (permission (– (* ncontracts current–price))
                   ncontracts)))))
```

Serializing get–price above is to avoid situation below in nikkei–update, which is same as the problem described in pitfall1.

```
(define (nikkei–update)
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian–random–pair
   (lambda (d1 d2)
    ;; d1 and d2 are both distributed as Gaussians
    ;;  with zero mean and unity standard deviation.
    ;; The fundamental drift
    (set! nikkei–fundamental
          (+ nikkei–fundamental
            nikkei–drift
            (* nikkei–split (+ d1 d2))))
    ;; The actual split –– cannot be modified
    ((tokyo 'new–price!)
     (lambda (old–price)
       (+ (average nikkei–fundamental old–price)
         (* nikkei–split d1))))
                                            <<<<——get–Tokyo–price will result *1* below
                                            <<<<——get–Singapore–price will result *2* below

    ((singapore 'new–price!)
     (lambda (old–price)
```

```
        (+ (average nikkei-fundamental old-price)
           (* nikkei-split d2))))))
```

However, this solution can not guarantee correct behavior that price-serializer was meant to protect.

Because there are two assumptions that have been violated:
1. Tokyo and Singapore markets update prices simultaneously.
2. Transaction get price from market in real time.

Here are examples:
1
Incorrect temporal order of displaying message:
"Transact nick-leeson's order 17546 contracts at price 16880.4"
"Nikkei-update- Tokyo: 16680.4   Singapore: 16779.8"

Correct temporal order of displaying message:
"Nikkei-update- Tokyo: 16680.4   Singapore: 16779.8"
"Transact nick-leeson's order 17546 contracts at price 16880.4"
Transact depend on current price.

2
Incorrect temporal order of displaying message:
"Nikkei-update- Tokyo: 16680.4   Singapore: 16779.8"
"Nikkei-update- Tokyo: 16681.7   Singapore: 16780.9"
"Transact nick-leeson's order 17546 contracts at price 16880.4"

Correct temporal order of displaying message:
"Nikkei-update- Tokyo: 16680.4   Singapore: 16779.8"
"Transact nick-leeson's order 17546 contracts at price 16880.4"
"Nikkei-update- Tokyo: 16681.7   Singapore: 16780.9"

OR

"Nikkei-update- Tokyo: 16680.4   Singapore: 16779.8"
"Nikkei-update- Tokyo: 16681.7   Singapore: 16780.9"
"Transact nick-leeson's order 17546 contracts at price 16881.7"
Transact depend on current price.

<<<——BACK-TO-PITFALLS-OF-CONCURRENCY

In later version of lab-exercise3, we will use database to store past market price for displaying messages.

```
(define (transact ncontracts market permission name)  ;same as lab-exercise2's nikkei here.
  ((market 'new-order!)
   (lambda ()
     (let ((current-price (((market 'price-serializer) (market 'get-price))))  ;;------modified
           )

       ((screen 'add-printing-with-arguments)
          'transact name ncontracts current-price)

       (permission (- (* ncontracts current-price))
                      ncontracts))))
)
```

In above codes, screen contains a copy of most recent market's price, but as code above shown, we explicitly ask market to return a current-price and pass to screen for displaying message. Designer has a choice to use either the screen's internal stored price or the one passed in from outside.

In situations where more than one state is the same, we must make sure the copy in the screen object and the current price are the same each time.

```
;;;;        England lost her Barings

;;; The collapse of the Baring Brothers establishment
;;;  provides an abject Leeson for the banking community.
;;;  -- GJS

(define (make-market name initial-price)
  (let ((price initial-price)
        (price-serializer (make-serializer))
        (pending-orders '())
        (pending-order-num 0)
        (orders-serializer (make-serializer)))
    (define (the-market m)
      (cond ((eq? m 'new-price!)
;              (price-serializer            ;;-------------modified
                 (lambda (update)
                   (set! price (update price)))
;                )                 ;;-------------disable serializer
              )
             ((eq? m 'get-price)
              (lambda () price))     ;;-return-type-modified—for—later—serialization-use.
             ((eq? m 'new-order!)
              (orders-serializer
               (lambda (new-order)
                   (set! pending-order-num (+ 1 pending-order-num))
                   (set! pending-orders
                         (append pending-orders
                                   (list new-order))))))
             ((eq? m 'execute-an-order)
              (((orders-serializer
                   (lambda ()
                     (if (not (null? pending-orders))
                         (let ((outstanding-order (car pending-orders)))
                           (set! pending-orders (cdr pending-orders))
                           (set! pending-order-num (- pending-order-num 1))
                           outstanding-order)
                         (lambda () 'nothing-to-do)))))))
             ((eq? m 'get-name) name)
             ((eq? m 'reset-order-num)
              ((orders-serializer (lambda () (set! pending-order-num 0)))))
             ((eq? m 'get-order-num)
              ((orders-serializer (lambda () pending-order-num))))
             ((eq? m 'price-serializer) price-serializer)     ;;-----------added
```

```scheme
          (else
           (error "Wrong message" m))))
    the-market))


;;; Using this we make two markets, both starting at the same price:

(define nikkei-fundamental 16680.0)

(define tokyo
  (make-market "Tokyo:Nikkei-225" nikkei-fundamental))

(define singapore
  (make-market "Singapore:Nikkei-225" nikkei-fundamental))


;;; Traders buy and sell contracts at a market using TRANSACT.
;;;  The trader gives the market permission to subtract from the
;;;  trader's monetary balance the cost of the contracts purchased
;;;  and to add to the trader's stash the contracts he purchased.

(define (buy ncontracts market permission arbitrager-name)  ;;note-on-lab-exercise2.txt
  (transact ncontracts market permission arbitrager-name))

(define (sell ncontracts market permission arbitrager-name)  ;;note-on-lab-exercise2.txt
  (transact (- ncontracts) market permission arbitrager-name))

(define (transact ncontracts market permission arbitrager-name) ;;note-on-lab-exercise2.txt
  ((market 'new-order!)
   (lambda ()
     (let ((current-price (((market 'price-serializer) (market 'get-price))))  ;;------modified
       (newline)
       (display "Transact ")
       (display arbitrager-name)
       (display "'s order: ")
       (display ncontracts)
       (display " contracts at ")
       (display current-price)
       (permission (- (* ncontracts current-price))
                   ncontracts)))))
```

```
(define transaction-cost 1.0)

(define (make-arbitrager name balance contracts authorization)
  (let ((trader-serializer (make-serializer)))

    (define (change-assets delta-money delta-contracts)
      ((trader-serializer
         (lambda ()
           (set! balance (+ balance delta-money))
           (set! contracts (+ contracts delta-contracts))))))

    (define (a<b low-place low-price high-place high-price)
      (if (> (- high-price low-price) transaction-cost)
           (let ((amount-to-gamble (min authorization balance)))
             (let ((ncontracts            ;round to nearest integer
                      (round (/ amount-to-gamble (- high-price low-price)))))
               (buy ncontracts low-place change-assets name)
               (newline)
               (display name)
               (display " consider buy ")
               (display ncontracts)
               (display " contracts at price ")
               (display low-price)
               (display " in ")
               (display (low-place 'get-name))
               (sell ncontracts high-place change-assets name)
               (newline)
               (display name)
               (display " consider sell ")
               (display ncontracts)
               (display " contracts at price ")
               (display high-price)
               (display " in ")
               (display (high-place ‘get-name))                    <<<——BACK—TO—PITTFALL 1
               ))))
```

```
(define (consider-a-trade)
  (let* ((price-pair ( ((tokyo 'price-serializer)
                        ((singapore 'price-serializer) (lambda () (cons ((tokyo 'get-price))
                                                                        ((singapore 'get-price))))))
                       )
                      )
         (nikkei-225-tokyo (car price-pair))
         (nikkei-225-singapore (cdr price-pair)))
    (if (< nikkei-225-tokyo nikkei-225-singapore)
      (a<b tokyo nikkei-225-tokyo
           singapore nikkei-225-singapore)
      (a<b singapore nikkei-225-singapore
           tokyo nikkei-225-tokyo))))

  (define (me message)
    (cond ((eq? message 'name) name)
          ((eq? message 'balance) balance)
          ((eq? message 'contracts) contracts)
          ((eq? message 'consider-a-trade) (consider-a-trade))
          (else
           (error "Unknown message -- ARBITRAGER" message))))
  me))

(define nick-leeson
  (make-arbitrager "Nick Leeson" 1000000000. 0.0 10000.))


;;; The following parameters determine the way the Nikkei average
;;;  drifts over time, and how the two markets differ.  The details of
;;;  this are probably not very much like a real market, and they
;;;  probably do not matter very much for this problem set.

(define nikkei-drift +0.1)
(define nikkei-split +1.0)

(define (nikkei-update)                   <<<<——BACK-TO-PITFALL 1
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian-random-pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei-fundamental
           (+ nikkei-fundamental
              nikkei-drift
```

```
            (* nikkei-split (+ d1 d2))))
    ;; The actual split -- cannot be modified
    ((tokyo 'new-price!)
     (lambda (old-price)
         (+ (average nikkei-fundamental old-price)
            (* nikkei-split d1))))
    ((singapore 'new-price!)
     (lambda (old-price)
            (+ (average nikkei-fundamental old-price)
               (* nikkei-split d2)))))))

(define serialized-nikkei-update      ;;nikkei-update
  ((tokyo 'price-serializer)
   ((singapore 'price-serializer)
    nikkei-update)))


(define (ticker market)
  (newline)
  (display (market 'get-name))
  (display "                ")
  (display (market 'get-price)))

(define (audit trader)
  (newline)
  (display (trader 'name))
  (display "     Balance: ")
  (display (trader 'balance))
  (display "     Contracts: ")
  (display (trader 'contracts)))


;;; Delay times are in milliseconds.
(define (do-aperiodically thunk max-delay)
  (define (again)
    (thunk)
    (sleep-current-thread (random max-delay))
    (again))
  again)

(define (do-periodically thunk time)
  (define (again)
    (thunk)
    (sleep-current-thread time)                          <<<——BACK—TO—PITFALL 1
    (again))
```

```
  again)


;;; Students: Please ignore the following hairy expression.
;;;  It is only here to allow us to reload this file into a
;;;  running simulation.
(if (lexical-unreferenceable? user-initial-environment 'stop-world)
    (define stop-world false))

;;; The following starts a bunch of threads running in parallel
;;;  and makes a procedure to stop all of the parallel threads.

(define (start-world)
    (if stop-world (stop-world))
    (set! stop-world                              <<<<—-BACK-TO-LAB3-VERSION 1
            (parallel-execute
             (do-aperiodically (lambda ()
                                     (newline)
                                     (display "nikkei-update")
                                     (serialized-nikkei-update)
                                     (ticker tokyo)
                                     (ticker singapore))
                                   3000)
             (do-aperiodically (lambda ()
                                     (newline)
                                     (display "nick-leeson consider-a-trade")
                                     (nick-leeson 'consider-a-trade)
                                     (tokyo-remain-orders)
                                     (singapore-remain-orders)
                                     (audit nick-leeson))
                                   4000)

             (do-aperiodically (lambda ()
                                     (newline)
                                     (display "tokyo execute-an-order")
                                     (tokyo 'execute-an-order)
                                     (tokyo-remain-orders)
                                     (audit nick-leeson)
                                     )
                                   2000)
             (do-aperiodically (lambda ()
                                     (newline)
                                     (display "singapore execute-an-order")
                                     (singapore 'execute-an-order)
                                     (singapore-remain-orders)
```

```
                                    (audit nick-leeson)
                                    )
                                  2500)

    ‘running)


(define (tokyo-remain-orders)
  (newline)
  (display "tokyo-remain-orders--:")
  (display (tokyo 'get-order-num)))

(define (singapore-remain-orders)
  (newline)
  (display "singapore-remain-orders--:")
  (display (singapore 'get-order-num)))
```

<<<——BACK—TO—FIRST

<<<——BACK—TO-LAB-EXERCISE-3

Since version1 has problem of multiple shared resources, we now protect display within critical section of relevant state variable.

For example:

"improve-nikkei-display" shows that serialized displaying message has moved inside of serialized-nikkei-update with respect to version 1.

"Improve-execute-order-display" shows that serialized displaying message has moved insider of method of "execute-an-order" with respect to version 1.

Notice from the solution of this version, if we need to display not a few lines but hundreds of lines, the Tokyo market and the Singapore market can no longer execute orders concurrently as version 1 does. Even though each market uses a different "order-serializer", they share the same "display-serializer" in order to address the "display-error" problem.

<<<<——BACK–TO–VERSION—2

```
(define (nikkei–update)
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian–random–pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei–fundamental
             (+ nikkei–fundamental
                nikkei–drift
                (* nikkei–split (+ d1 d2))))
     ;; The actual split –– cannot be modified
     ((tokyo 'new–price!)
      (lambda (old–price)
          (+ (average nikkei–fundamental old–price)
             (* nikkei–split d1))))
     ((singapore 'new–price!)
      (lambda (old–price)
             (+ (average nikkei–fundamental old–price)
                (* nikkei–split d2))))
             )
   )
 (  (serialized (lambda () (newline)                <<<<——BACK–TO–VERSION–3
                              (display "nikkei–update")
                              (ticker tokyo)
                              (ticker singapore))
                     ) ) )

)

(define serialized–nikkei–update       ;;serialized–nikkei–update in version1
  ((tokyo 'price–serializer)
   ((singapore 'price–serializer)
    nikkei–update)))
```

<<<<———BACK—TO—VERSION–2

```
(define (make–market name initial–price)
  (let ((price initial–price)
          (price–serializer (make–serializer))
          (pending–orders '())
          (pending–order–num 0)
          (orders–serializer (make–serializer)))
    (define (the–market m)
      (cond ((eq? m ‘new–price!)
                     (lambda (update)
                           (set! price (update price)))          )
             ((eq? m 'get–price)
               (lambda () price))       ;;–return–type–modified—for—later—serialization–use.
              ((eq? m 'new–order!)
               (orders–serializer
                (lambda (new–order)
                    (set! pending–order–num (+ 1 pending–order–num))
                    (set! pending–orders
                          (append pending–orders
                                    (list new–order))))))
              ((eq? m 'execute–an–order)
               (((orders–serializer
                    (lambda ()
                      (if (not (null? pending–orders))
                          (let ((outstanding–order (car pending–orders)))
                              (set! pending–orders (cdr pending–orders))
                              (set! pending–order–num (– pending–order–num 1))
(  (serialized (lambda ()
                                     (newline)
                                     (display "tokyo execute–an–order")
                                     (tokyo–remain–orders)
                                     (audit nick–leeson)
                                     )
                     ) ) )
                              outstanding–order)
                          (lambda () 'nothing–to–do)))))))
              ((eq? m 'get–name) name)
              ((eq? m 'reset–order–num)
               ((orders–serializer (lambda () (set! pending–order–num 0)))))
              ((eq? m 'get–order–num)
               ((orders–serializer (lambda () pending–order–num))))
              ((eq? m 'price–serializer) price–serializer)      ;;–––––––––––added
              (else
               (error "Wrong message" m))))
    the–market))
```

<<<——BACK—TO—LAB—EXERCISE—3 TO—VERSION—4——>>>>>>

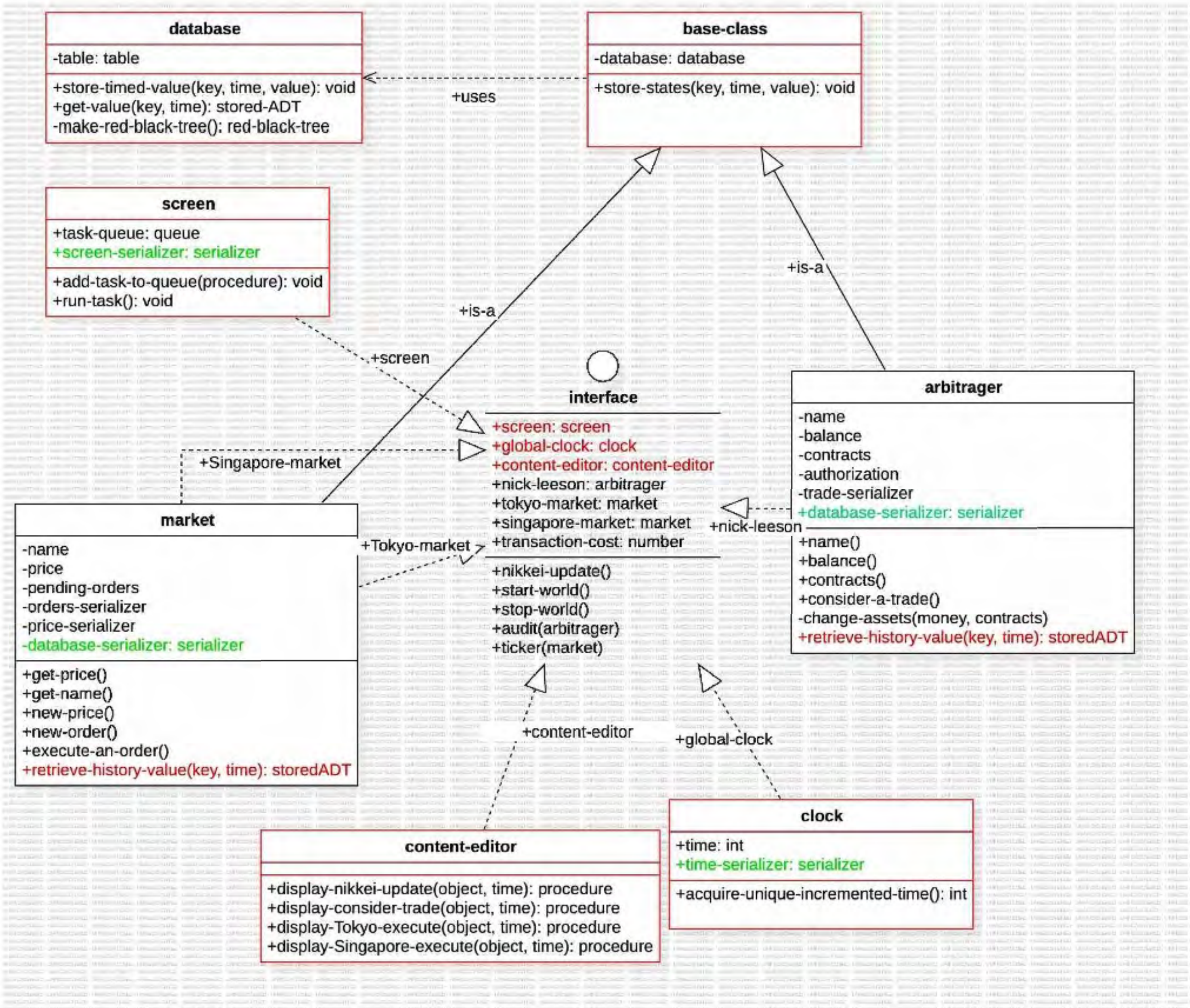
This version was meant to achieve two goal:

Goal A: Improve efficiency by delaying displaying messages

Goal B: Manage different displaying plan for future extension.

To achieve above goals, we need:

1. Clock: To sort events by assigning each event a unique number in ascending order.
2. Database: To store the history of all time–value pairs of each state variable for retrieval.
3. Content–Editor: To contain all display functions and to return the executable display function that will retrieve the state's past value and display appropriate messages.
4. Queue: To store and run functions returned from content–editor.



The above class–diagram was modified from original "class-diagram". Now both the arbitrager class and the market class inherit from the base–class which can store the timed–value of states. Note there are 4 serializers being used in the class–diagram above. Instead of displaying messages right after updating price as "improve–nikkei–display" in lab3–version2 does, now process of displaying message can be delayed:

```
(define (nikkei-update)
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian-random-pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei-fundamental
             (+ nikkei-fundamental
                nikkei-drift
                (* nikkei-split (+ d1 d2))))
     ;; The actual split -- cannot be modified
     ((tokyo 'new-price!)
      (lambda (old-price)
          (+ (average nikkei-fundamental old-price)
             (* nikkei-split d1))))
     ((singapore 'new-price!)
      (lambda (old-price)
             (+ (average nikkei-fundamental old-price)
                (* nikkei-split d2))))
             )
   )
(Screen's-serializer-protect
;;Step1: Get unique time
  (Let ((new-time (global-clock 'acquire-unique-incremented-time)))
;;Step2: Store states' time-value pair to database
    ( (Tokyo 'store-states) 'tokyo-price new-time (tokyo 'get-price))
;;Step2: Store states' time-value pair to database
    ( (Singapore 'store-states) 'singapore-price new-time (Singapore 'get-price))
;;Step3: Add delayed displaying task to queue
    ( (Screen 'add-task-to-queue)  (lambda () (content-editor 'display-nikkei-update) time))
  )
)

(define serialized-nikkei-update
 ((tokyo 'price-serializer)
   ((singapore 'price-serializer)
    nikkei-update)))
```

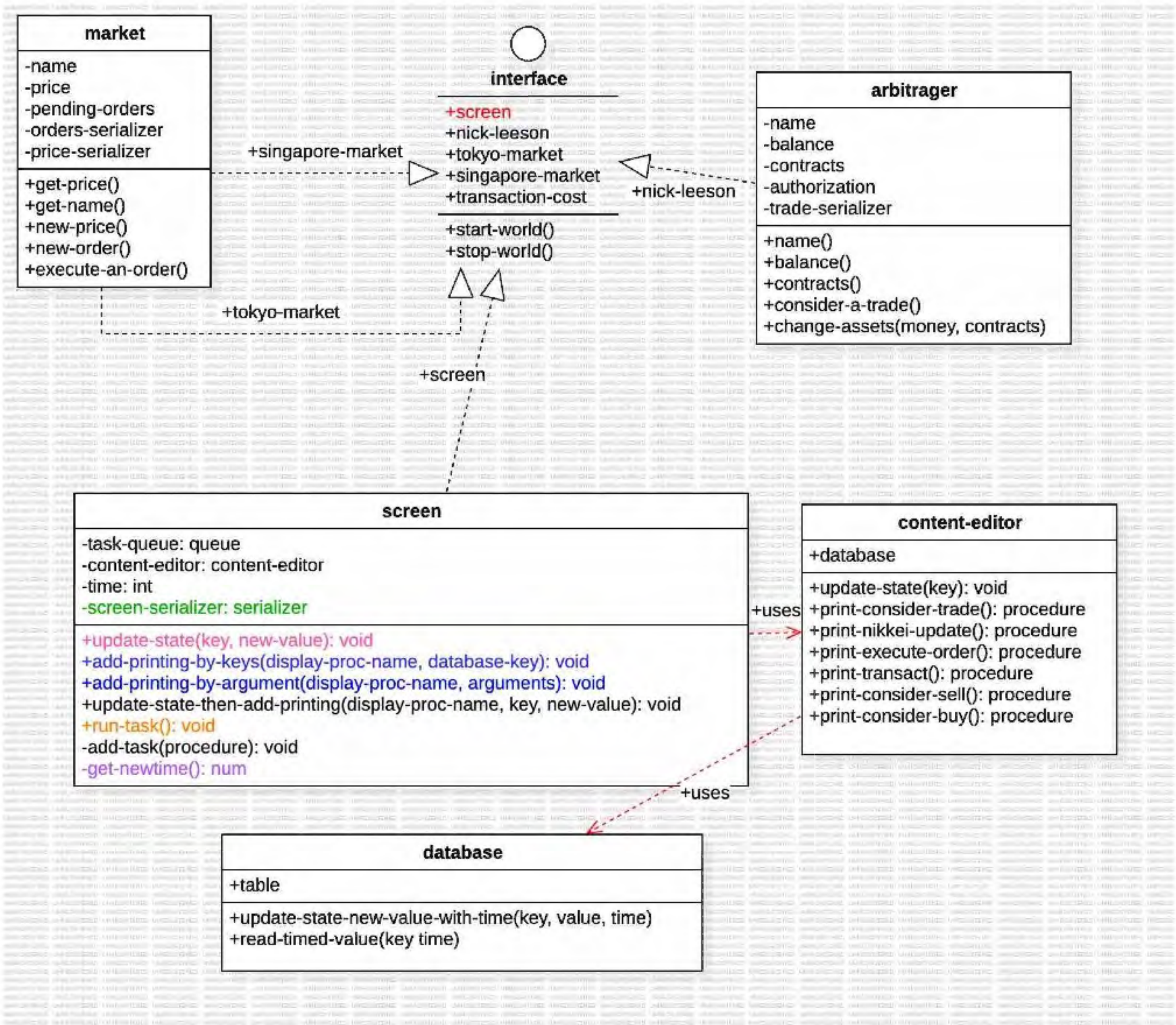
Now we need to add one more line in start-world

```
(define (start-world)
    (if stop-world (stop-world))
```

```
(set! stop-world
      (parallel-execute
       (do-aperiodically (lambda ()
                           (serialized-nikkei-update))      ;;step 1-3
                         3000)
       (do-aperiodically (lambda ()
                           (nick-leeson 'consider-a-trade))  ;;step 1-3
                         4000)

       (do-aperiodically (lambda ()
                           (tokyo 'execute-an-order))   ;;step 1-3
                         2000)
       (do-aperiodically (lambda ()
                           (singapore 'execute-an-order))  ;;step 1-3
                         2500)
      (do-aperiodically (lambda ()
                           (screen 'run-task))         ;; Step 4
                         2500)
        ))
 'running)
```

<<<——BACK—TO-FIRST

<<<——BACK–TO–LAB–EXERCISE–3 TO–FINAL–VERSION—>>>>>>

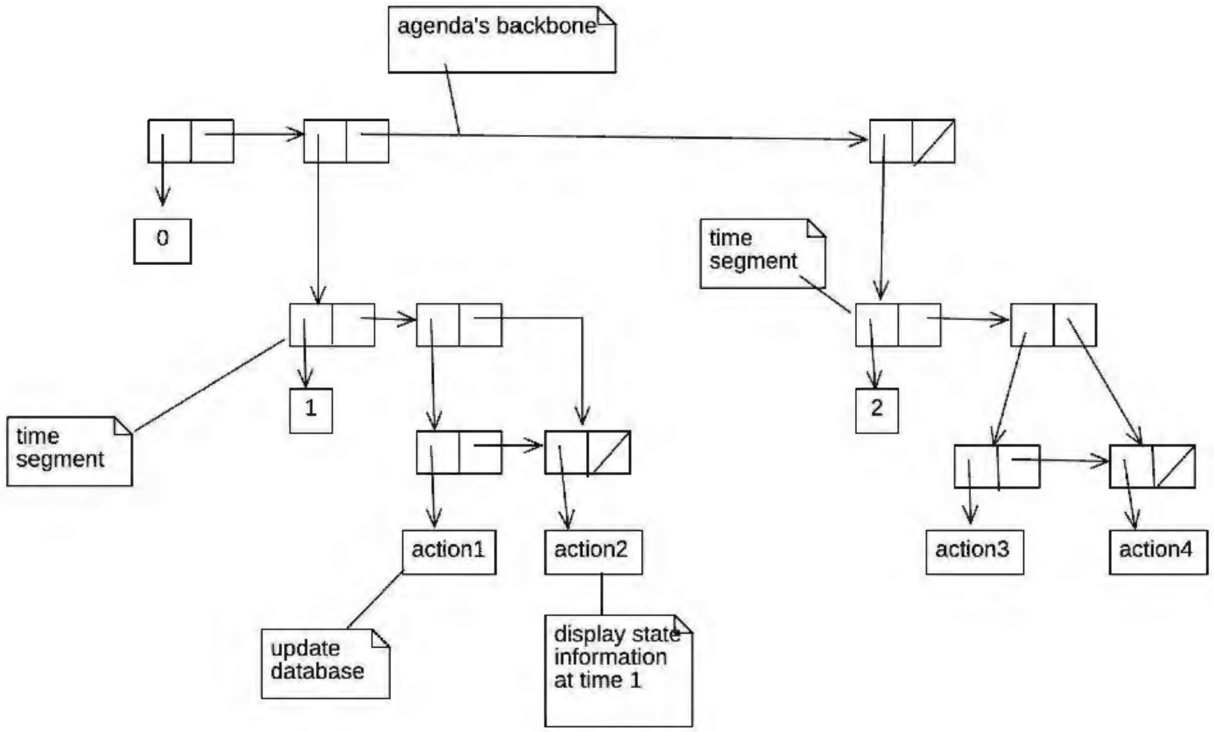


This version does exactly the same as version3 in

Step1: Get unique time(in purple).
Step2: Store time–value pair to database(in pink).
Step3: Add delayed displaying task to queue(in blue).
Step4: Run tasks in queue from a new thread(in orange).

but with constraint of using only 1 serializer (green) by having one screen object (red) in interface to handle all steps above. Now we modify nikkei–sourceCode2 to nikkei–sourceCodeFinal (red) where screen is delegated to handle all steps 1–4.

To improve efficiency, one way is to change the data structure of task–queue to a data structure shown below that can further delay the process of updating database (step 2).



(Agenda is used in text SICP3.3.4 to simulate digital circuits, which itself is a table consisting of time segments. Each time segment consists of a time and a queue.)

As shown in the diagram above, we can delay step2 by placing step 2 at the position of “action1” at time 1, and place step 3 at the position of “action2” at time 1, thereby shorten the time of “synchronization”.

Another problem we must resolve is that we need some way to dynamically adjust the rate of running tasks.

In the following lines of code, “(screen ‘run–task)” in red only run task once for each repetition. Therefore, in order to keep up with the speed of inputting tasks, “(screen ‘run–task)” must run more frequently than other threads.

In next version, we are going to solve this problem using a “feedback loop”.

```
(define (start–world)
    (if stop–world (stop–world))
    (set! stop–world
           (parallel–execute
            (do–aperiodically (lambda ()
                                   (serialized–nikkei–update))       ;;step 1–3
```

```
                      3000)
        (do-aperiodically (lambda ()
                       (nick-leeson 'consider-a-trade))  ;;step 1-3
                      4000)

        (do-aperiodically (lambda ()
                       (tokyo 'execute-an-order))   ;;step 1-3
                      2000)
        (do-aperiodically (lambda ()
                       (singapore 'execute-an-order))  ;;step 1-3
                      2500)
       (do-aperiodically (lambda ()
                       (screen 'run-task))         ;; Step 4
                      50)
         ))
   'running)
```

<<<—BACK-TO-FIRST

<<<<——BACK—TO—LAB-EXERCISE-3
<<<<——BACK—TO-VERSION-4

In final version, we are going to solve 3 problems:

1. Delay the process of updating database.
2. Concurrently adding and deleting tasks.
3. Balancing the speed of input tasks and their execution.

We are going to solve problem 1 & 2 by exploiting a data structure I call "concurrent agenda", which was modified from a data structure called "agenda" used in the textbook. For readability, please see "concurrent-agenda".

For problem 3, what we need is an adaptive-rate control algorithm, specifically a feedback loop. A rudimentary version of feedback loop codes was obtained from Chatgpt 4o. Note that the following feedback loop was modified from this "chatgpt version of feedback loop".

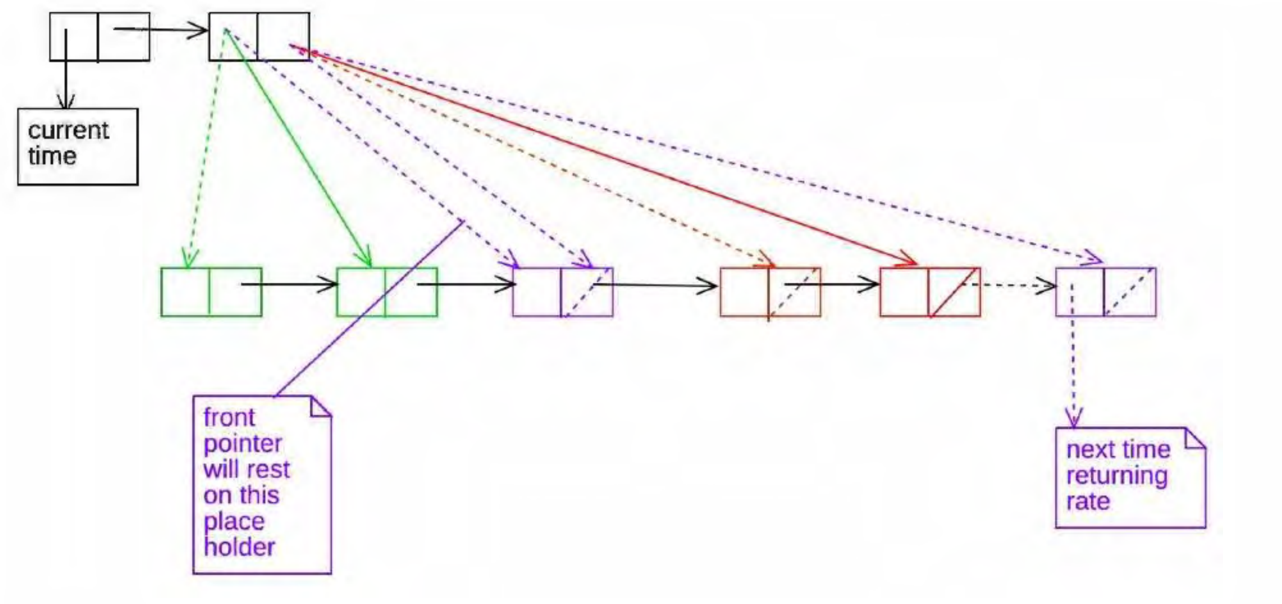
In order to synthesize the use of feedback loop and concurrent agenda, a key step is to pause the concurrent process by placing a placeholder to the concurrent-agenda whenever the feedback loop is acquiring the rate for next loop.

Detail is on next page for readability.

Here is a revised version of feedback loop:

```
(define print–loop
  (let* ((rate 0)
          (feedback (lambda ()
                        (let ((length ((agenda 'length–to–proceed)))) ;if placeholder already exists,
                         (if (= length 0)                              ;only return length.
                              (set! rate 0)
                              (set! rate length))))))

    (lambda ()
      ((agenda 'run–task) rate)  ;run and delete tasks depend on amount of rate.
      (feedback)                 ;reset rate
      (print–loop))))
```

The purple code above places a place holder to the agenda as below diagram shown, and return a rate(in this case is 2) for green code above to run. The red box below shows that while agenda is running task, new tasks are added to agenda concurrently by extending place holder. Each red box and place holder are serialized by ONE serializer.



Now we use print–loop as below:

```
(define (start–world)
    (if stop–world (stop–world))
    (set! stop–world
          (parallel–execute
           (do–aperiodically (lambda ()
                                (serialized–nikkei–update))
```

```
                    3000)
        (do–aperiodically (lambda ()
                      (nick–leeson ‘consider–a–trade))
                    4000)

        (do–aperiodically (lambda ()
                      (tokyo ‘execute–an–order))
                    2000)
        (do–aperiodically (lambda ()
                      (singapore ‘execute–an–order))
                    2500)

         (lambda () (print–loop))

         ))
    ‘running)
```

Here is a PROOF of why this solution works:

Protecting the critical section of returning the amount of tasks to process and placing the placeholder by the same serializer used for adding tasks ensures that:

1. There is no sharing when concurrently adding new tasks and running old tasks.
2. There is no change either when placing a new placeholder or when all tasks are complete or there is no task at all.

The theoretical issue of “sameness and change” suggests that as long as we don't mix sharing with change, our rational reasoning above should not have an unexpected outcome.

Making use of concurrent agenda allow chance to discard the database. The essential job for database is to store timed value of state variables and provide convenience for later displaying message. But since concurrent agenda run tasks in temporal order, which means displaying message right after updating state, there is no need to retrieve states’ past value.

For codes see “agenda–class”, “screen–class”, “content–editor”, “database”, “common–define”, “printing–database–adt”.

<<—BACK—TO—FIRST

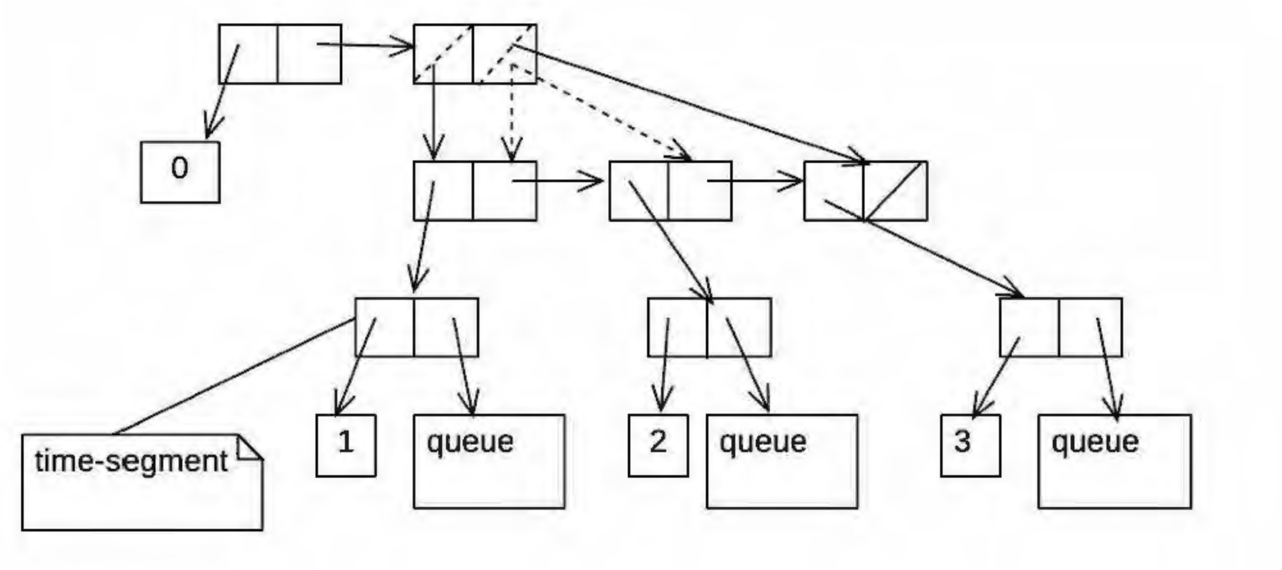
<<<<——BACK—TO—FINAL—VERSION

I ask Chatgpt 4o to generate Scheme code of feedback control loop after learning the fact that adaptive rate control might be a potential solution to the problem. Here is the code generated by Chatgpt 4o:

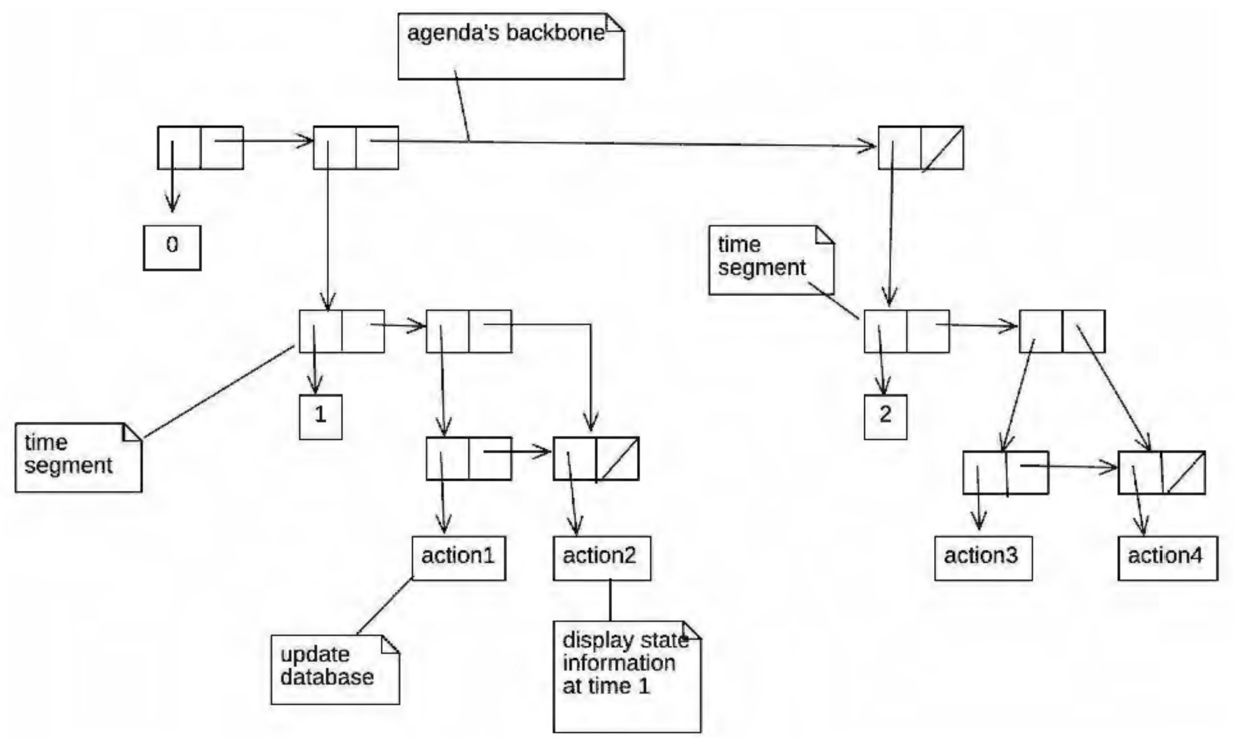
```
(define (feedback–control–loop)
  (let* ((queue '())          ; Initialize the queue
         (max–queue–length 10) ; Maximum queue length before adjusting rate
         (min–queue–length 2)  ; Minimum queue length before adjusting rate
         (rate 1)              ; Initial deletion rate
         (add–tasks (lambda (n)
                      (set! queue (append queue (make–list n 'task)))))
         (delete–tasks (lambda ()
                         (set! queue
                               (if (< rate (length queue))
                                   (drop queue rate)
                                   '()))))
         (feedback (lambda ()
                     (cond
                      ((> (length queue) max–queue–length) (set! rate (+ rate 1))) ; Increase rate
                      ((< (length queue) min–queue–length) (set! rate (max 1 (– rate 1)))))))) ;
Decrease rate

    ;; Main loop
    (let loop ((iteration 1))
      (when (< iteration 20) ; Run for 20 iterations
        (display (string–append "Iteration " (number–>string iteration) ":\n"))
        (display (string–append "  Queue: " (number–>string (length queue)) "\n"))
        (display (string–append "  Deletion rate: " (number–>string rate) "\n"))

        ;; Add and delete tasks
        (add–tasks (random 5)) ; Randomly add up to 5 tasks
        (delete–tasks)
        (feedback) ; Adjust rate based on queue length

        ;; Continue the loop
        (loop (+ iteration 1))))))

;; Run the control loop
(feedback–control–loop)
```

<<<<——BACK-TO-FINAL-VERSION

A “concurrent agenda” is essentially a pair of current time and a queue data structure. Each element within the queue is another pair of time and queue called time-segment and is sorted in order.



The concurrent agenda was modified from an “agenda” data structure used in text SICP 3.3.4 to simulate digital circuits. Where the backbone of the agenda was not a queue but a list.



<<<———BACK—TO—VERSION–4 <<<———Nikkei–SourceCode2

```
;;;;        England lost her Barings

;;; The collapse of the Baring Brothers establishment
;;;  provides an abject Leeson for the banking community.
;;;  -- GJS


(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise3\\version6\\common-define.scm")
(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise3\\version6\\content-editor.scm")


(define (make-market name initial-price)
  (let ((price initial-price)
        (price-serializer (make-serializer))
        (pending-orders '())
        (pending-order-num 0)
        (orders-serializer (make-serializer)))
    (define (the-market m)
      (cond ((eq? m 'new-price!)
                   (lambda (update)
                     (set! price (update price)))
             )
            ((eq? m 'get-price)
             (lambda () price))
            ((eq? m 'new-order!)
             (orders-serializer
              (lambda (new-order)
                  (set! pending-order-num (+ 1 pending-order-num))
                  (set! pending-orders
                        (append pending-orders
                                  (list new-order)))
                  ((screen 'update-state)
                   (list (get-key name 'remain-orders))  ;;update-remain-order
                   (list (make-stored-remain-order
                          name
                          pending-order-num)))            ;;in database
                  )))
            ((eq? m 'execute-an-order)
             (((orders-serializer
                  (lambda ()
                    (if (not (null? pending-orders))
```

```
                    (let ((outstanding-order (car pending-orders)))
                      (set! pending-orders (cdr pending-orders))
                      (set! pending-order-num (- pending-order-num 1))

                      ((screen 'update-state-then-add-printing)
                       'execute-order
                       (list (get-key name 'remain-orders))  ;;update-remain-order
                       (list (make-stored-remain-order       ;;in database
                              name
                              pending-order-num)))
                      outstanding-order)
                    (lambda () 'nothing-to-do)))))))
              ((eq? m 'get-name) name)
              ((eq? m 'reset-order-num)
               ((orders-serializer (lambda () (set! pending-order-num 0)))))
              ((eq? m 'get-order-num)
               ((orders-serializer (lambda () pending-order-num))))
              ((eq? m 'price-serializer) price-serializer)
              (else
               (error "Wrong message" m))))
  the-market))


;;; Using this we make two markets, both starting at the same price:

(define nikkei-fundamental 16680.0)

(define tokyo
  (make-market tokyo-market-name nikkei-fundamental))

(define singapore
  (make-market singapore-market-name nikkei-fundamental))
```

```
;;; Traders buy and sell contracts at a market using TRANSACT.
;;;  The trader gives the market permission to subtract from the
;;;  trader's monetary balance the cost of the contracts purchased
;;;  and to add to the trader's stash the contracts he purchased.

(define (buy ncontracts market permission name)  ;same as lab-exercise2's nikkei here.
  (transact ncontracts market permission name))

(define (sell ncontracts market permission name)  ;same as lab-exercise2's nikkei here.
  (transact (- ncontracts) market permission name))

(define (transact ncontracts market permission name)  ;same as lab-exercise2's nikkei here.
  ((market 'new-order!)
   (lambda ()
     (let ((current-price (((market 'price-serializer) (market 'get-price))))  ;;------modified
             )

       ((screen 'add-printing-with-arguments)
          'transact name ncontracts current-price)

       (permission (- (* ncontracts current-price))
                       ncontracts))))
)




(define transaction-cost 1.0)

(define (make-arbitrager name balance contracts authorization)
  (let ((trader-serializer (make-serializer)))

    (define (change-assets delta-money delta-contracts)
      ((trader-serializer
          (lambda ()
            (set! balance (+ balance delta-money))
            (set! contracts (+ contracts delta-contracts))
            ((screen 'update-state-then-add-printing)
             'audit
             (list (get-key name 'arbitrager))
             (list (make-stored-arbitrager name balance contracts)))

            ))))
```

```
(define (a<b low-place low-price high-place high-price)
  (if (> (- high-price low-price) transaction-cost)
        (let ((amount-to-gamble (min authorization balance)))
          (let ((ncontracts             ;round to nearest integer
                   (round (/ amount-to-gamble (- high-price low-price)))))

            ((screen 'add-printing-with-arguments)
             'consider-buy name ncontracts low-price (low-place 'get-name))
            ((screen 'add-printing-with-arguments)
             'consider-sell name ncontracts high-price (high-place 'get-name))

            (buy ncontracts low-place change-assets name)
            (sell ncontracts high-place change-assets name)

            ((screen 'add-printing-with-keys)
             'consider-trade
             (list (get-key name 'arbitrager))

            )))))

(define (consider-a-trade)
  (let* ((price-pair ( ((tokyo 'price-serializer)
                          ((singapore 'price-serializer) (lambda () (cons ((tokyo 'get-price))
                                                                          ((singapore 'get-price))))))
                        )
                      )
           (nikkei-225-tokyo (car price-pair))
           (nikkei-225-singapore (cdr price-pair)))
          ;;------above--modified--from--original--------
      (if (< nikkei-225-tokyo nikkei-225-singapore)
          (a<b tokyo nikkei-225-tokyo
                 singapore nikkei-225-singapore)
          (a<b singapore nikkei-225-singapore
                 tokyo nikkei-225-tokyo))))

(define (me message)
  (cond ((eq? message 'name) name)
          ((eq? message 'balance) balance)                <<<<——BACK-TO-FIRST
          ((eq? message 'contracts) contracts)
          ((eq? message 'consider-a-trade) (consider-a-trade))
          (else
           (error "Unknown message -- ARBITRAGER" message))))
me))
```

```scheme
(define nick-leeson
  (make-arbitrager nick-leeson-name 1000000000. 0.0 10000.))

(define bob-citron
  (make-arbitrager bob-citron-name 1000000000. 0.0 10000.))
```

```scheme
;;; The following parameters determine the way the Nikkei average
;;;  drifts over time, and how the two markets differ.  The details of
;;;  this are probably not very much like a real market, and they
;;;  probably do not matter very much for this problem set.

(define nikkei-drift +0.1)
(define nikkei-split +1.0)


(define (nikkei-update)
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian-random-pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei-fundamental
           (+ nikkei-fundamental
              nikkei-drift
              (* nikkei-split (+ d1 d2))))
     ;; The actual split -- cannot be modified
     ((tokyo 'new-price!)
      (lambda (old-price)
        (+ (average nikkei-fundamental old-price)
           (* nikkei-split d1))))
     ((singapore 'new-price!)
      (lambda (old-price)
```

```
            (+ (average nikkei–fundamental old–price)
               (* nikkei–split d2))))
            )
   )
  ;;Constraints:
  ;;1.add–printing should run outside of gaussian–random–pair.
  ;;  If running inside of gaussian–random–pair would cause error.
  ;;2.Since there is a multiple shared resource problem, two price–serializer
  ;;  should protect the whole nikkei–update from start–world, rather insider
  ;;  nikkei–update.
  ((screen 'update–state–then–add–printing)                ;;to–update–state–and–print
   'nikkei–update
   (list 'tokyo–price 'singapore–price)
   (list (make–stored–price (tokyo 'get–name)
                            ((tokyo 'get–price)))
         (make–stored–price (singapore 'get–name)
                            ((singapore 'get–price))))
   )

)

(define serialized–nikkei–update      ;;nikkei–update constraint 2
  ((tokyo 'price–serializer)
   ((singapore 'price–serializer)
    nikkei–update)))

(define (ticker market)
  (newline)
  (display (market 'get–name))
  (display "                ")
  (display ((market 'get–price))))

(define (audit trader)
  (newline)
  (display (trader 'name))
  (display "     Balance: ")
  (display (trader 'balance))
  (display "     Contracts: “)                                  <<<<——BACK–TO–FIRST
  (display (trader 'contracts)))
```

```
;;; Delay times are in milliseconds.
(define (do-aperiodically thunk max-delay)
  (define (again)
    (thunk)
    (sleep-current-thread (random max-delay))
    (again))
  again)

(define (do-periodically thunk time)
  (define (again)
    (thunk)
    (sleep-current-thread time)
    (again))
  again)


;;; Students: Please ignore the following hairy expression.
;;;  It is only here to allow us to reload this file into a
;;;  running simulation.
(if (lexical-unreferenceable? user-initial-environment 'stop-world)
    (define stop-world false))

;;; The following starts a bunch of threads running in parallel
;;;  and makes a procedure to stop all of the parallel threads.

(define (start-world)
    (if stop-world (stop-world))
    (set! stop-world
          (parallel-execute
           (do-aperiodically (lambda ()
                               (nikkei-update))
                             3000)
           (do-aperiodically (lambda ()
                               (nick-leeson 'consider-a-trade))
                             4000)
           (do-aperiodically (lambda ()
                               (tokyo 'execute-an-order))
                             2000)
           (do-aperiodically (lambda ()
                               (singapore 'execute-an-order))
                             2500)

           (do-aperiodically (lambda ()
                              (screen ‘run-task))
```

50)
))
'running)

<<<<——BACK-TO-FIRST

<<<——BACK–TO–LAB–3–FINAL–VERSION

```
(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise3\\version7\\queue.scm")


;;IMPORTANT: make notes along with coding!! e.g.Why made such design choice?

(define (make-agenda)
  ;;Note: Agenda store two types of data: 1. time segment 2. place holder.
  (let ((agenda (make-concurrent-agenda)))
    (define (dispatch m)
      (cond ((eq? 'add-task m) (lambda (task time) (add-task-to-agenda task time agenda)))
            ((eq? 'add-place-holder m) (lambda () (add-place-holder agenda)))
            ((eq? 'current-time m) (current-time agenda))
            ((eq? 'run-task m) (lambda (rate) (run-task rate agenda)))
            ((eq? 'total-length m) (length (front-ptr (tasks-queue agenda)))) ;;for testing
            (else (error "wrong message to agenda"))
            )
      )

    dispatch)
  )

;;---concurrent-agenda--data--abstraction--
(define (make-concurrent-agenda)
  ;;Note: This data structure similar to "agenda" described in text SICP 3.3.4.
  ;;Type: -> pair<time, queue< place-holder| pair<time, queue <procedure>> > >
  (cons 0 (make-queue)))

(define (current-time agenda)
;;Note: interface procedure.
  (car agenda))

(define (tasks-queue agenda)
  (cdr agenda))

(define (set-current-time! agenda time)
  (set-car! agenda time))

(define (add-to-agenda-new-time-segment time task agenda)
  ;;Note: internal procedure used for "add-task-to-agenda"
  (insert-queue! (tasks-queue agenda)
                 (let ((return (make-time-segment time (make-queue))))
                   (add-task-to-time-segment task return)
```

```
                        return)))

(define (add-task-to-agenda task time agenda)
  ;;Note:This is procedure is an interface procedure used to add task to agenda,
  ;;      and is different to another procedure "add-place-holder".
  ;;      Agenda only add items using this procedure or using "add-place-holder".
  ;;Type: procedure,num,agenda-> void
  ;;Constraints:
  ;;1. Input time can only be greater or equal to most recent time.
  ;;2. input task must be a procedure.
  ;;3. There are only two types of previous-task: i. time-segment ii. place-holder.
  ;;4. check for empty case.
  (cond ((procedure? task)
         (let ((previous-task (up-to-date-task agenda)))

           (cond ((not previous-task) (add-to-agenda-new-time-segment time task
agenda)) ;constraint 4
                 ((place-holder? previous-task)      ;;constraint 3
                  (add-to-agenda-new-time-segment time task agenda))
                 ((time-segment? previous-task)      ;;constraint 3
                  (if (exist-time? time previous-task)        ;constraint 1
                      (add-task-to-time-segment task previous-task)  ;;add task to existing
time-segment.
                      (add-to-agenda-new-time-segment time task agenda)))

                 (else (error "wrong type of previous-task")))))
        (else (error "wrong type input to add-task-to-agenda")))
)


(define (add-place-holder agenda)
  ;;Note: This is an interface procedure.
  ;;Constraints:
  ;;1. If previous task is place-holder, no action.
  (let ((previous-task (up-to-date-task agenda)))
    (cond ((not previous-task) (insert-queue! (tasks-queue agenda) (make-place-holder)))
          (previous-task
           (if (not (place-holder? previous-task))
               (insert-queue! (tasks-queue agenda) (make-place-holder))))  ;;constaint 1
          (else (error "add-place-holder should not be here"))
        ))

)
```
<<—BACK—TO—FIRST

```
(define (exist-time? time last-segment)
```

```
  ;;Note: Return value of this predicate is to determine whether to extend current backbone
  ;;      or not in function "add-task-to-agenda".
  ;;Type: num, time-segment -> boolean

  (let ((last-time (segment-time last-segment)))

    (cond ((= time (+ 1 last-time)) false)
          ((= time last-time) true)
          (else (error "wrong input time value from exist-time? function")))))

(define (up-to-date-task agenda)
  ;;Note: to get this last task in the agenda's task-queue.
  ;;Constraints:
  ;;1. Check empty case, before using rear-pointer of queue data structure.

  (let ((last-pair (rear-ptr (tasks-queue agenda))))
    (if (empty-queue? (tasks-queue agenda))      ;;constraint 1
        false
        (car last-pair))))


(define (first-task agenda)
  (front-queue (tasks-queue agenda))
  )

(define (delete-task! agenda)
  (delete-queue! (tasks-queue agenda))
  )

(define (run-task rate agenda)
  ;;Note: This is an interface procedure.
  ;;Constraints:
  ;;1. when rate is 0, do noting.
  ;;2. when task's type is place-holder, don't count.
  ;;3. set new time for agenda finish each task.
  (let ((task (first-task agenda)))
    (if nikkei-debug-mode
        (begin
          (newline)
          (display "task =")
          (display task)
          (newline)
          (display "first-task =“)                                   <<—BACK—TO—FIRST
          (display (first-task agenda)))
        )
```

```
  (cond ((= rate 0) 'done)            ;;constraint 1
        ((time-segment? task) (begin
                                (let ((time (segment-time task)))
                                  (run-time-segment task)
                                  (delete-task! agenda)
                                  (set-current-time! agenda time))   ;constraint 3
                                (run-task (- rate 1) agenda)))
        ((place-holder? task)  (begin
                                 (delete-task! agenda)
                                 (run-task rate agenda)))  ;;constraint 2

        (else (error "error occur in run-task"))))

  )
;;----time-segement-data-abstraction-
(define (make-time-segment time queue)
  (list 'time-segment time queue))

(define (segment-time s) (cadr s))

(define (segment-queue s) (caddr s))

(define (add-task-to-time-segment task s)
  (insert-queue! (segment-queue s) task))

(define (time-segment? s)
  (and (pair? s)
       (equal? (car s) 'time-segment)))

(define (run-time-segment s)
  (if (not (time-segment? s))
      (error "input is not a time-segment")
      (let loop ((queue (segment-queue s)))
          (cond ((empty-queue? queue) 'done)
                (else ((front-queue queue))
                      (loop (delete-queue! queue))))))))


;;------place-holder---abstraction----
(define (make-place-holder)
  (list 'place-holder))

(define (place-holder? p)
  (and (pair? p)
       (eq? 'place-holder (car p))))
```

<<—BACK—TO—FIRST

;;------------------------------note------------
below--------------------------------------

;;think about make-agenda interface procedure by recalling constraint 2 of screen class in version7.
;;Question: is agenda class equivalent to screen class? if so, then adaptive rate control, concurrently
;;   reading/deleting can all directly modify from screen class.

;;what is the type of sub-task?
;;Note: I should specify a high level data abstraction rather than focusing on
;;   the actual data structure. Think in terms of operation!!

;;continue to implement propagate when come back!!

<<—BACK—TO—FIRST

<<<——BACK—TO—LAB–3–FINAL–VERSION

```
;;----screen-class---------------------------------

(define (make-screen typed-keys)
  ;;Input-type: list<typed-key>.
  ;;Constraints:
  ;;1.Moment/time is unique for each request.
  ;;2.There are five use cases of calling screen:
  ;;  i. Update-state.
  ;;     e.g. "new-order" in market-class
  ;;  ii. Add printing task to queue by directly passing arguments.
  ;;     e.g. "consider-buy","consider-sell" in a<b, "transact" in transact.
  ;;  iii. Add printing task to queue by using keys to retrieve data from database
  ;;       according to time.
  ;;       e.g. "consider-a-trade" in a<b in arbitrager-class.
  ;;  iv. Update-state first according to given keys, then do iii.
  ;;      e.g. nikkei-update, change-assets in make-arbitrager, execute-an-order in make-
market.
  ;;  v.  Print out printing task in queue.
  ;;      e.g. should be called from parallel-execute on a separate thread.
  (let ((agenda (make-agenda))
        (content-editor (make-editor-with-typed-key typed-keys)) ;;display editor need to
know both.
        (time 0)
        (protected (make-serializer)))

;;----internal---------procedure----------
    (define (get-new-time)
      (set! time (+ time 1))
      time)


    (define (add-task procedure time)
      ((agenda 'add-task) procedure time))

    (define (add-place-holder)
      ((agenda' add-place-holder))
      )

    (define (update-state-loop key-seq value-seq time)
      ;;Note: Used in update-state and update-state-then-add-printing.
      ;;type: list,list -> no-value
      (cond ((and (null? key-seq) (null? value-seq)) 'done)
            ((and (not (null? key-seq)) (not (null? value-seq)))
```

```
          (begin
            (if nikkei-debug-mode
                  (begin
                    (display "update-state-loop")
                    (newline)
                    (display "key-seq = ")
                    (display key-seq)
                    (display "    value-seq = ")
                    (display value-seq)))
            (let ((result
                     ((content-editor 'update-state)          ;;updating
                      (car key-seq) (car value-seq) time)))
                (if nikkei-debug-mode
                    (begin
                      (newline)
                      (display "result from update state loop")
                      (newline)
                      (display "result = ")
                      (display result)))
                )

            (update-state-loop (cdr key-seq) (cdr value-seq) time)))
         (else
          (error "key-seq and value-seq are not in the same length"))))




  (define (get-key-time-print-procedure procedure-name key-list time)
    ;;Note: This procedure is for adding printing task with acquiring information from
    ;;       database by providing key.
    (lambda () ((content-editor procedure-name) time key-list)))

  (define (get-ordinary-print-procedure procedure-name time args)
    ;;Note: This procedure is used for directly passing arguments printing tasks
    ;;       without acquiring information from database.
    ;;Constraints:
    ;;1. pass time to content-editor for future use.
    (lambda () (apply (content-editor procedure-name) (cons time args)))) ;constraint 1

;;---------internal----------procedures--------above-----------------


;;----------interface--procedure--------------------------
```

```
(define update-state
  (protected (lambda (key-list new-value-list)
               (let ((current-moment (get-new-time)))
                 (add-task (lambda () (update-state-loop key-list new-value-list current-
moment))
                           current-moment)
                 ))))

(define add-printing-with-keys
  (protected (lambda (print-procedure-name key-list)
               (let ((current-moment (get-new-time)))
                 (add-task (get-key-time-print-procedure print-procedure-name key-list
current-moment)
                           current-moment)))))



(define add-printing-with-arguments
  (protected (lambda (print-procedure-name . arguments)
               (let ((current-moment (get-new-time)))
                 (add-task (get-ordinary-print-procedure print-procedure-name current-
moment arguments)
                           current-moment)))))



(define update-state-then-add-printing
  ;;Note: update-database then add task to queue.
  ;;Type: symbol, list<key>, list<value>
  (protected (lambda (print-procedure-name key-list new-value-list)
               (let ((current-moment (get-new-time)))
                 (add-task (lambda () (update-state-loop key-list new-value-list current-
moment))
                           current-moment)

                 (add-task (get-key-time-print-procedure print-procedure-name key-list
current-moment)
                           current-moment)))))

(define (run-task)
  ;;This simply return the "real" run-task from agenda which
  ;;has type: number"rate" --> symbol"done".
  (agenda 'run-task))

(define calculate-length-and-add-place-holder
```

```
  ;;constraints:
  ;;1.add place-holder for agenda's queue's front and rear
  ;;  pointer to rest on when there is no task to run.
  ;;2. return the amount of tasks for batch processing.
  (protected (lambda ()
                (if nikkei-debug-mode
                    (begin
                        (newline)
                        (display "screen's time = ")
                        (display time)
                        (newline)
                        (display "agenda's current time =")
                        (display (agenda 'current-time)))
                    )

                (add-place-holder)
                (- time (agenda 'current-time)))))

(define (dispatch m)
  (cond ((eq? 'update-state m) update-state)               ;;constraint 2
        ((eq? 'add-printing-with-keys m) add-printing-with-keys)  ;;constraint 2
        ((eq? 'add-printing-with-arguments m)              ;;constraint 2
         add-printing-with-arguments
         )
        ((eq? 'update-state-then-add-printing m)           ;;constraint 2
         update-state-then-add-printing
         )
        ((eq? 'run-task m)          ;;should be called from parallel-execute
         (run-task)
         )
        ((eq? 'length-to-proceed m)
         calculate-length-and-add-place-holder
         )
        ((eq? 'agenda m)       ;;for testing efficiency.
         agenda
         )
        ((eq? ‘acquire-dataset m)
         (lambda (key) ((content-editor ‘get-dataset) key)))
        ((eq? 'export-content-editor m)
         content-editor)             ;;for testing
        ((eq? 'time m) time)       ;;for testing
        (else
         (error "make-screen should not be here"))))

dispatch)
```

```
  )
```

——initialization————————————————————————

```
(define screen (make-screen key-list))
((screen 'update-state) (list 'tokyo-price 'singapore-price 'tokyo-remain-order 'singapore-remain-order
                                   'nick-leeson 'bob-citron)
                              (list (make-stored-price (tokyo 'get-name)
                                                          ((tokyo 'get-price)))
                                    (make-stored-price (singapore 'get-name)
                                                          ((singapore 'get-price)))
                                    (make-stored-remain-order (tokyo 'get-name)
                                                                 0)
                                    (make-stored-remain-order (singapore 'get-name)
                                                                 0)
                                    (make-stored-arbitrager (nick-leeson 'name)
                                                               (nick-leeson 'balance)
                                                               (nick-leeson 'contracts))
                                    (make-stored-arbitrager (bob-citron 'name)
                                                               (bob-citron 'balance)
                                                               (bob-citron 'contracts))))
```

```
(define print-loop
  (let* ((rate 0)
         (feedback (lambda ()
                      (let ((length ((screen 'length-to-proceed))))
                        (if (= length 0)
                            (set! rate 0)
                            (set! rate length))))))

    (lambda ()
      ((screen 'run-task) rate)  ;when rate is 0, screen's agenda do nothing.
      (feedback)
      (print-loop))))
```

<<<——BACK—TO—FIRST

<<——BACK—TO—LAB-3-FINAL-VERSION

```
(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise4\\database.scm")

(define (make-editor-with-typed-key typed-keys)
  ;;input-type: list<typed-key>.  --typed-key = <printing-type, database-key>
  ;;constraints:
  ;;1.printing types is to categorize keys so same group of keys can be selected when printing.
  ;;2.database only need "database-key" part to store values.
  ;;3.Except "update-database" method, all methods have parameter "time" should put
  ;;  it first argument. See "get-ordinary-print-procedure" in screen-class/extra-printing-
nikkei.scm.
  (let ((data-base
          (make-database-with-keys (map (lambda (typed-key) (database-key typed-
key)) ;constraint 2
                                         typed-keys)))
         (all-arbitrager-keys (select-all-keys 'arbitrager typed-keys))
         (all-market-price-keys (select-all-keys 'market-price typed-keys))
         (all-remain-order-keys (select-all-keys 'remain-orders typed-keys)))

    (define (update-database key value time)
      ((data-base 'update-state-new-value-with-time)
       key value time))

    (define (print-all-arbitragers time key-seq)
      (if nikkei-debug-mode
            (begin
              (display "running print-all-arbitrager in content-editor.scm")
              (newline)
              (display "key-seq = ")
              (display key-seq)))
      (for-each (lambda (stored-arbitrager)
                      (print-arbitrager (name-arbitrager stored-arbitrager)
                                              (balance-arbitrager stored-arbitrager)
                                              (contracts-arbitrager stored-arbitrager)))
                    (map (lambda (key)
                           ((data-base 'read-timed-value) key time))
                         key-seq))
      )

    (define (print-all-remain-orders time key-seq)
      (if nikkei-debug-mode
            (begin
              (display "running print-all-remain-orders in content-editor.scm")
              (newline)
```

```
        (display "key-seq = ")
        (display key-seq)))
  (for-each (lambda (stored-remain-order)
                (print-remain-orders (name-remain-order stored-remain-order)
                                     (number-remain-order stored-remain-order)))
              (map (lambda (key)
                     ((data-base 'read-timed-value) key time))
                   key-seq))
  )

(define (print-nikkei-update time key-seq)
  (if nikkei-debug-mode
      (begin
        (display "running print-nikkei-update in content-editor.scm")
        (newline)
        (display "key-seq = ")
        (display key-seq)))

  (newline)
  (display "nikkei-update")
  (for-each (lambda (stored-price)
                (print-market-price (name-stored-price stored-price)
                                    (price stored-price)))
              (map (lambda (key)
                     ((data-base 'read-timed-value) key time))
                   key-seq))

  )
(define (print-consider-trade time key-seq)
  (if nikkei-debug-mode
      (begin
        (display "running print-consider-trade in content-editor.scm")
        (newline)
        (display "key-seq = ")
        (display key-seq)))

  (let ((data ((data-base 'read-timed-value) (car key-seq) time)))
    (newline)
    (display (name-arbitrager data))                          <<——BACK—TO—FIRST
    (display " consider-a-trade")
    (print-all-remain-orders time all-remain-order-keys)
    (print-arbitrager (name-arbitrager data)      ;;audit
                      (balance-arbitrager data)
                      (contracts-arbitrager data))
    )
```

```
  )

(define (print-execute-order time key-seq)
  (if nikkei-debug-mode
      (begin
        (display "running print-execute-order in content-editor.scm")
        (newline)
        (display "key-seq = ")
        (display key-seq)))

  (let ((stored-remain-order ((data-base 'read-timed-value) (car key-seq) time)))
    (newline)
    (display (name-remain-order stored-remain-order))
    (display " execute-an-test-order")
    (print-remain-orders (name-remain-order stored-remain-order)
                 (number-remain-order stored-remain-order))
    (print-all-arbitragers time all-arbitrager-keys))    ;;different to print-arbitrager.

  )


(define (print-transact time arbitrager-name ncontracts current-price)   ;;should I pass in
current-price?
  (newline)                                          ;;should I use local price?
  (display "Transact ")
  (display arbitrager-name)
  (display "'s order: ")
  (display ncontracts)
  (display " contracts at ")
  (display current-price))

(define (print-consider-buy time name ncontracts low-price low-place)
  (newline)
  (display name)
  (display " consider buy ")
  (display ncontracts)
  (display " contracts at price ")
  (display low-price)
  (display " in ")
  (display low-place))
```
<<——BACK—TO—FIRST
```
(define (print-consider-sell time name ncontracts high-price high-place)
  (newline)
  (display name)
```

```
      (display " consider sell ")
      (display ncontracts)
      (display " contracts at price ")
      (display high-price)
      (display " in ")
      (display high-place))

    (define (ticker time key-seq)
      (let ((stored-price ((data-base 'read-timed-value) (car key-seq) time)))
          (newline)
          (display (name-stored-price stored-price))
          (display "                    ")
          (display (price stored-price))
          )
      )

    (define (audit time key-seq)
      (let ((stored-arbitrager ((data-base 'read-timed-value) (car key-seq) time)))
          (newline)
          (display (name-arbitrager stored-arbitrager))
          (display "     Balance:  ")
          (display (balance-arbitrager stored-arbitrager))
          (display "     Contracts:  ")
          (display (contracts-arbitrager stored-arbitrager))
          )
      )
    (define (dispatch m)
      (cond ((eq? m 'update-state) update-database)
              ((eq? m 'consider-trade) print-consider-trade)
              ((eq? m 'nikkei-update) print-nikkei-update)
              ((eq? m 'execute-order) print-execute-order)
              ((eq? m 'ticker) ticker)
              ((eq? m 'audit) audit)
              ((eq? m 'transact) print-transact)
              ((eq? m 'consider-sell) print-consider-sell)
              ((eq? m 'consider-buy) print-consider-buy)          <<——BACK—TO—FIRST
              ((eq? m 'get-dataset) (data-base 'get-key-tree))
              ((eq? m 'export-database) data-base)     ;;for testing
              (else (error "wrong print procedure name from editor"))))
    dispatch)
  )

(define (select-all-keys type-string typed-key-seq)
  ;;type: symbol,list<typed-key> --> list<key>
  (map (lambda (typed-key) (database-key typed-key))
```

```
    (filter (lambda (typed-key) (eq? type-string (print-type typed-key)))
            typed-key-seq)))


(define (print-market-price market-name market-price)
  ;;note: This procedure function as ticker in nikkei.scm.
  (newline)
  (display market-name)
  (display "           ")
  (display market-price)
  )
(define (print-arbitrager arbitrager-name balance contracts-num)
  (newline)
  (display arbitrager-name)
  (display "    Balance  ")
  (display balance)
  (display "    Contracts: ")
  (display contracts-num))

(define (print-remain-orders market-name remain-number)
  (newline)
  (display market-name)
  (display "-remain-orders--:")
  (display remain-number)
  )
```

<<——BACK—TO—FIRST

<<——BACK—TO—LAB–3–FINAL–VERSION

```
(define (make-database-with-keys keys-seq)
  (let ((table (make-table))
        )
    (define (initialize keys)
      (if (null? keys) 'done
          (begin
            (insert! (car keys) (make-rb-tree (lambda (key1 key2)(= key1 key2))
                                              (lambda (key1 key2)(< key1 key2)))
                     table)
            (initialize (cdr keys)))))
    (define (dispatch m)
      (cond ((eq? m 'update-state-new-value-with-time)
             (if nikkei-debug-mode
                 (display "updating-database"))
             (lambda (key value time)
               (if nikkei-debug-mode
                   (begin
                     (display "key = ")
                     (display key)
                     (display " value = ")
                     (display value)
                     (display " time = ")
                     (display time)))
               (let ((record (lookup key table)))
                 (if nikkei-debug-mode
                     (begin
                       (newline)
                       (display "record = ")
                       (display record)))
                 (insert! key (update record value time) table)
                     )))
            ((eq? m 'read-timed-value)
             (lambda (key time)
               (let ((record (lookup key table)))
                 (read-latest-value record time)
                 )))
            ((eq? m 'get-key-tree)
             (lambda (key)
               (let ((record (lookup key table)))
                 (if record
                     record
                     #f))))
```

```
            ((eq? m 'table) table)   ;;for testing
            (else
             (begin
             (newline)
             (display "message =")
             (display m)
             (error "wrong message passing to database")))))


  (begin
    (initialize keys-seq)
    (if nikkei-debug-mode
          (begin
            (display "table= ")
            (display table)))
    dispatch))
  )

(define (update tree value time)
  ;;type: red-black-tree, value , num ---> red-black-tree
  ;;Constraints:
  ;;1. update should return the correct type.


    (if (rb-tree? tree)
          (begin
            (rb-tree/insert! tree time value)
            tree)                    ;;constraint 1
          (error "record is not a rb-tree"))



  )
```

<<——BACK—TO—FIRST

```
(define (read-latest-value tree time)
    (cond ((< time 0) (error "read database error from read-latest-value"))
          ((not (rb-tree? tree))
           (if nikkei-debug-mode
               (begin
                   (display "tree =")
                   (display tree)))
           (error "input tree argument is not a red-black tree"))
          (else
           (let ((result (rb-tree/lookup tree time #f)))
             (if result
                   result
```

```
                    (read-latest-value tree (- time 1)))))
    )
  )
;;-----------------------------------abstract---data-------
type----------------------
(define (make-stored-price market-name price)         ;;data type for printing market's price
  (list 'stored-price market-name price))
(define (stored-price? stored-price)
  (and (pair? stored-price) (eq? 'stored-price (car stored-price))))
(define (name-stored-price stored-price)
  (cadr stored-price))
(define (price stored-price)
  (caddr stored-price))


(define (make-stored-remain-order name remain-order-num)     ;;data type for printing
market's
  (list 'stored-remain-order name remain-order-num))         ;;remaining order number.
(define (stored-remain-order? stored-remain-order)
  (and (pair? stored-remain-order) (eq? 'stored-remain-order (car stored-remain-order))))
(define (name-remain-order stored-remain-order)
  (cadr stored-remain-order))
(define (number-remain-order stored-remain-order)
  (caddr stored-remain-order))

(define (make-stored-arbitrager name balance contracts)      ;;data-type for printing
arbitrager's
  (list 'stored-arbitrager name balance contracts))          ;;balance, name, number of
contracts hold.
(define (stored-arbitrager? stored-arbitrager)
  (and (pair? stored-arbitrager) (eq? 'stored-arbitrager (car stored-arbitrager))))
(define (name-arbitrager stored-arbitrager)
  (cadr stored-arbitrager))
(define (balance-arbitrager stored-arbitrager)
  (caddr stored-arbitrager))
(define (contracts-arbitrager stored-arbitrager)
  (cadddr stored-arbitrager))
```

<<——BACK—TO—FIRST

```
;;---------------table-------------------------
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        false)))
```

```
(define (assoc key records)
  (cond ((null? records) false)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))

(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))))
  'ok)

(define (make-table)
  (list '*table*))

;;for test.
;(define test ((screen 'export-content-editor) ‘export-database))
```

<<——BACK—TO—FIRST

<<——BACK—TO—LAB–3–FINAL–VERSION

```
(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise3\\version7\\printing-
database-adt.scm")

(define singapore-market-name "Singapore:Nikkei-225")
(define tokyo-market-name "Tokyo:Nikkei-225")
(define nick-leeson-name "Nick Leeson")
(define bob-citron-name "Bob Citron")

(define nikkei-debug-mode #f)

(define key-list
  ;;Type: list<printing-type,database-key>
  ;;constrants:
  ;;1. Database-key must be unique.

  (list (make-typed-key 'market-price 'tokyo-price tokyo-market-name)
        (make-typed-key 'market-price 'singapore-price singapore-market-name)
        (make-typed-key 'remain-orders 'tokyo-remain-order tokyo-market-name)
        (make-typed-key 'remain-orders 'singapore-remain-order singapore-market-name)
        (make-typed-key 'arbitrager 'nick-leeson nick-leeson-name)
        (make-typed-key 'arbitrager 'bob-citron bob-citron-name)
        ))


(define (get-key object-name printing-type)
  (let ((result
         (filter (lambda (typed-key) (and (eq? printing-type (print-type typed-key))
                                          (string=? object-name (obj-name typed-key))))
                 key-list)))
    (if (null? result)
        (error "wrong input-name or wrong print-type from get-key")
        (database-key (car result)))))

(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

<<——BACK—TO—LAB-3-FINAL-VERSION

```scheme
;;--Typed-key ADT is for pairing types of printing and key used in database.


(define (make-typed-key printing-type database-key obj-name)
  ;;Note: typed-key is used for content-editor to select database-key,
  ;;      and for database to initialize.
  ;;constraints:
  ;;1. printing-type is only used in content-editor.
  ;;2. database only need database-key part.

  (list 'typed-key printing-type database-key obj-name))

(define (print-type typed-key)
  (cadr typed-key))

(define (database-key typed-key)
  (caddr typed-key))

(define (obj-name typed-key)
  (cadddr typed-key))

(define (typed-key? typed-key)
  (and (pair? typed-key) (eq? 'typed-key (car typed-key))))
```

<<<<<————BACK TO LAB–EXERCISE–3

Nick Leeson   Balance: 1000033373.2553301   Contracts: 0.
Tokyo:Nikkei–225   16699.073590082524
Singapore:Nikkei–225   16699.394023862475

Nick Leeson   Balance:

1000043373.4133779Tokyo:Nikkei–225Singapore:Nikkei–225   Contracts:
0.16704.064555059056
16705.588097859585
Nick Leeson
Tokyo:Nikkei–225   Balance: Singapore:Nikkei–225   1000182114.5180726
16707.81048714844 Contracts:
16712.4327448166550.
Tokyo:Nikkei–225
Singapore:Nikkei–225   Nick Leeson   16712.90963016294 Balance:
16715.69511598564 1000349313.0177624
Tokyo:Nikkei–225   Contracts: Singapore:Nikkei–225   0.
16722.26199041812316724.79325501129
Nick Leeson
Tokyo:Nikkei–225   Balance: Singapore:Nikkei–225   1000667925.539962
16732.390419884254   Contracts:
16737.028997054440.
Tokyo:Nikkei–225
Singapore:Nikkei–225   Nick Leeson   16747.243150929655   Balance:
16747.05373673048 1000544844.486979
Tokyo:Nikkei–225   Contracts: Singapore:Nikkei–225   0.
16750.26247028707416755.226123455104
Nick Leeson
Tokyo:Nikkei–225   Balance: Singapore:Nikkei–225   1000703729.3915502
16758.70583472226 Contracts:
16761.9978996830320.
Tokyo:Nikkei–225
Singapore:Nikkei–225   Nick Leeson   16764.740815590358   Balance:
16770.84776707908 1000875970.813458
Tokyo:Nikkei–225   Contracts: Singapore:Nikkei–225   0.
16773.99767731992 16775.1866047199

<<<——BACK–TO–LAB–EXERCISE–3

Sharing becomes significant when it involves change, based on the underlying theoretical issue of “sameness and change”.

Sharing occurs within this project in 2 ways:
1:   Prices of both markets (Tokyo, Singapore) are shared by “nikkei–update”(interface function) and “consider–a–trade” (arbitrager class method). To recall see use–case diagram.

2: The display of relevant messages must be consistent with the temporal order of the state's change.

The project originally assumes that the Tokyo and Singapore markets update prices simultaneously. This assumption, combined with the above two ways of sharing, causes the pitfalls being discussed below.

#1
“Pitfall1” solve multiple–shared–resources–problem related to *1* mentioned above which was not addressed by original set up.
“Pitfall1” shows the need to use protected version of nikkei–update as shown in version1(in pink).

#2
“Pittfall2” and “pittfall3” solve another two multiple–shared–resource problems related to *2* mentioned above. As lab–exercise3 states, version1’s solution(in red) is not a good solution to the problem of “display–error”, because of problems of “pittfall2” and “pittfall3”.

#3
“Coherence” discuss a situation when there is more than one variable represent the same state, these variables must have the same value.

<<<<——BACK—TO—LAB-EXERCISE-3

It is almost impossible to reproduce the concurrency bug on a modern computer as presented in “display-error” except by configuring the CPU execution cap in the virtual machine.

In general, concurrency bugs are difficult to reproduce. For example, I spent a substantial amount of time trying to reproduce a read and write buffer error mentioned in parallel.scm but eventually failed to do so and gave up.

Here below are the description and code snippet copied from parallel.scm:

```
#|
;;; IO system is not completely interlocked, so...      ;it means in MIT-Scheme system IO
                                                         ;is not interlocked
(define (try n)
  (parallel-execute
   (lambda ()                      ;start a thread.
     (write-line ‘hi)              ;display a line “hi”.
     (let lp ((i 0))               ;run loop i = 10000.
       (if (< i 10000)
           (lp (1+ i))))
     (write-line ‘gjs))            ;display a line  “gls”.

   (lambda ()                      ;start another thread.
     (write-line ‘there)           ;write line “there”.
     (let lp ((i 0))               ;run loop i = 9188.
       (if (< i n)
           (lp (1+ i))))
     (write-line ‘foo))))          ;write line “foo”.

(define foo (try 9188))
;Value foo

hi
there
foo
foo                                ;there is a chance foo will repeat!
gjs

(foo)
;No value
|#
```

```
(define (make-serializer)
  (let ((mutex (make-thread-mutex)))
    (define (serialized f)
      (define (serialized-f . args)
          (with-thread-mutex-locked mutex
                                    (lambda ()
                                      (apply f args))))
      serialized-f)
    serialized))
```

Now we serialize the function “write-line”.

```
(define output-serialized (make-serializer))

(define write-line
  (output-serialized write-line))

(define display
  (output-serialized display))

(define write
  (output-serialized write))
#|
;;; This solves the IO interlock problem


(define (try n)                                    ;repeat
  (parallel-execute
   (lambda ()
     (write-line 'hi)
     (let lp ((i 0))
       (if (< i 10000)
             (lp (1+ i))))
     (write-line 'gjs))
   (lambda ()
     (write-line 'there)
     (let lp ((i 0))
       (if (< i n)
             (lp (1+ i))))
     (write-line 'foo))))

(define foo (try 9197))
;Value: foo
```

```
hi
there
gjs
foo                                             ;problem solved

(define foo (try 9196))
;Value: foo

hi
there
foo
gjs
|#
```

<<<-BACK-TO-FIRST

Lab Exercise 4: Your job is to to invent another kind of trader, such as one who tries to predict the future value of the commodity from analysis of its past behavior. You may try whatever strategy you think is effective, but you may not cheat by changing the code of any other component of the system or by diddling with the parameters of the nikki-update, or by setting the price of a market. Prizes will be awarded for the most interesting trader invented. Implement your trader, install the trader as a process, and demonstrate that the program works. Show output demonstrating that interesting trades are being made.

<<<——BACK—TO—CONCURRENCY

In Lab–exercise 3, we showed that there was a need to use the database for retrieval of past states' values.

After watching videos of the first two weeks of the machine learning course on Coursera, I decided to use both linear and polynomial regression to analyze the market's price stored in the database.

The following codes show that Jack periodically analyze past Tokyo market's price and Singapore market's price.

```
(define (start–world5)
    (if stop–world (stop–world))
    (set! stop–world
            (parallel–execute
             (do–aperiodically (lambda ()
                                 (serialized–nikkei–update))
                               10)
             (do–periodically (lambda ()
                                 ((jack 'analyze–history)))
                               5000)

             (lambda () (print–loop))
              ))
    'running)
```

There are 2 functions Jack employs to fit the curves of markets' prices:

1. y = w * x + b
2. y = w1 * x + w2 * x^1/2 + b

Here y represents the price and x represents count (e.g. 1,2,3…) of calling price–update (nikkei–update).

Tokyo-price

Count	Func1 parameters w/b	Func1 predict next price (count + 1)	Func2 parameters w1/w2/b	Func2 predict next price (count +1)	Actual next price (count + 1)
542	137.9/16920.9	17160.193	157.65/-20.16/ 16920.9	17165.863	17169.587
946	248.0/17106.9	17536.882	261.46/-13.75/ 17106.9	17540.733	17524.471
1102	285.4/17175.4	17670.211	284.42/1.01/17 175.4	17669.886	17660.726

singapore-price

Count	Func1 parameters w/b	Func1 predict next price (count + 1)	Func2 parameters w1/w2/b	Func2 predict next price (count + 1)	Actual next price (count + 1)
542	137.7/16922.5	17161.44	157.10/-19.8/ 16922.5	17167.02	17172.54
946	248.0/17108.5	17538.54	262.0/-14.3/ 17108.5	17542.54	17525.27
1102	285.4/17177.1	17671.89	284.9/0.53/ 17177.1	17671.70	17660.78

It is not very clear which model performs better in the Tokyo market or the Singapore market. My intuition for the next step is to find the curve that fits seasonality (box in red), then I can determine how far forward or backward to predict (in purple).

Codes are in class-predict-future, linear-regression-price, x-plus-root-x-regression, multiple-linear-regression&feature-scaling.

<<—BACK—TO—FIRST

<<<——BACK—TO—LAB-EXERCISE-4

```
(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise4\\price-regression-model\
\linear-regression-price.scm")

(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise4\\price-regression-model\
\x-plus-root-x-regression.scm")

(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise4\\database.scm")

;;----------ADT---------------------------
;;Note: To store parameters in database within class new-arbitrager.
(define (make-store-parameter w-vector b count)
  (list 'store-parameter w-vector b count))

(define (store-parameter? store-parameter)
  (and (pair? store-parameter)
       (eq? (car store-parameter) 'sotre-parameter)))

(define (sp-get-w-vector stored-parameter)
  (cadr stored-parameter))

(define (sp-get-b stored-parameter)
  (caddr stored-parameter))

(define (sp-get-count stored-parameter)
  (cadddr stored-parameter))



;;;;----------new----class------------
(define (make-new-arbitrager name balance contracts authorization)
  ;;Note: Local database uses "tokyo-x" to denote sequence of parameters of model y= wx +
b
  ;;      for predicting Tokyo-price, and
  ;;      uses "singapore-root-x" to denote sequence of parameters of model y= w1x +
w2x^1/2 + b
  ;;      for predicting Singapore-price.

  (let ((parent (make-arbitrager name balance contracts authorization))
        (database (make-database-with-keys (list 'tokyo-x 'tokyo-root-x 'singapore-x
'singapore-root-x)))
        (analyze-protect (make-serializer)))

    (define (initialization)
```

```
  (update-database 'tokyo-x
                   (make-store-parameter (list 0) 0 0)
                   0)
  (update-database 'tokyo-root-x
                   (make-store-parameter (list 0 0) 0 0)
                   0)
  (update-database 'singapore-x
                   (make-store-parameter (list 0) 0 0)
                   0)
  (update-database 'singapore-root-x
                   (make-store-parameter (list 0 0) 0 0)
                   0)
  )
(define (update-database key value time)
  ((database 'update-state-new-value-with-time)
   key value time))

(define (transform-data-into-adt seq)
  (make-store-parameter (car seq)
                        (cadr seq)
                        (caddr seq)))

(define (dispatch message)
  (cond ((eq? message 'display) (lambda (db-key) (display-prediction db-key)))
        ((eq? message 'read)
         (lambda (key count) ((database 'read-timed-value)
                              key count)))
        ((eq? message 'analyze-history)
         (analyze-protect
          (lambda () (let ((tokyo-linear-param (linear-regression-with-tokyo-price))
                           (tokyo-rootx-param (x+x^1/2-regression-with-tokyo-price))
                           (singapore-linear-param (linear-regression-with-singapore-price))
                           (singapore-rootx-param (x+x^1/2-regression-with-singapore-
price)))
                       (begin
                         (if tokyo-linear-param
                             (let ((adt (transform-data-into-adt tokyo-linear-param)))
                               (update-database 'tokyo-x
                                                adt
                                                (sp-get-count adt)))) ;store size of dataset as
time.


                         (if tokyo-rootx-param
                             (let ((adt (transform-data-into-adt tokyo-rootx-param)))
```

```
                                        (update-database 'tokyo-root-x
                                                         adt
                                                         (sp-get-count adt)))) ;store size of dataset as
time.
                                (if singapore-linear-param
                                      (let ((adt (transform-data-into-adt singapore-linear-param)))
                                        (update-database 'singapore-x
                                                         adt
                                                         (sp-get-count adt)))) ;store size of dataset as
time.

                                (if singapore-rootx-param
                                      (let ((adt (transform-data-into-adt singapore-rootx-param)))
                                        (update-database 'singapore-root-x
                                                         adt
                                                         (sp-get-count adt)))) ;store size of dataset as
time.
                                'done
                                )))
                 ))
               ((eq? message 'export-table) (database 'table))   ;only for debug
               (else (parent message))))
    (initialization)
    dispatch)
)

(define jack (make-new-arbitrager 'jack 1000000000 0 10000))




(define (compare-prediction-with-price db-key store-parameter)
  (let* ((w-vector (sp-get-w-vector store-parameter))
          (b (sp-get-b store-parameter))
          (count (sp-get-count store-parameter))
          (market-price (cond ((or (eq? db-key 'tokyo-x)
                                   (eq? db-key 'tokyo-root-x))
                               'tokyo-price)
                              ((or (eq? db-key 'singapore-x)
                                   (eq? db-key 'singapore-root-x))
                               'singapore-price)
                              (else (error "wrong input of db-key" db-key))))

          (y-seq (->price-seq ((screen 'acquire-dataset) market-price)))
```

```
        (predict-price (cond ((or (eq? db-key 'tokyo-x)
                                  (eq? db-key 'singapore-x)) (linear-next-price count
                                                                                w-vector
                                                                                b))
                             ((or (eq? db-key 'tokyo-root-x)
                                  (eq? db-key 'singapore-root-x))
                              (root-x-predict-price count
                                                    w-vector
                                                    b))
                             (else error "wrong input of db-key" db-key))))

    (list 'count:
          count
          'predicted:
          predict-price
          'actual:
          (list-ref y-seq (- count 1)))
    )
  )


(define (->price-seq rb-tree)
  (if rb-tree
      (map price (rb-tree/datum-list rb-tree))
      false))


(define (->seq-price db-key)
  (rb-tree/datum-list (lookup db-key (jack 'export-table))))

(define (display-prediction db-key)
  (let ((seq-stored-params (filter (lambda (param) (not (= 0 (sp-get-count param))))
                                   (->seq-price db-key))))
    (for-each (lambda (x) (newline) (display x))
              (map (lambda (x)(compare-prediction-with-price db-key
                                                             x))
                   seq-stored-params))
    )
)
```

<<<——BACK—TO—FIRST

<<<——BACK—TO—LAB-EXERCISE-4

(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise4\\regression-lib\\multiple-linear-regression&feature-scaling.scm")

```
(define (linear-regression-with-tokyo-price)
  (let* ((tokyo-price-dataset ((screen 'acquire-dataset) 'tokyo-price))
         (y-seq (->price-seq tokyo-price-dataset)))
    (if (not y-seq)
        false
        (let* ((x-seq (z-scale-normalization
                       (enumerate-interval 1 (length y-seq))))
               (x-vector-seq (map (lambda (x)
                                    (list x))
                                  x-seq))
               (w-vector (list 0))
               (b 16680)
               (rate 0.9)
               (result (multiple-linear-regression w-vector
                                                   x-vector-seq
                                                   y-seq
                                                   b
                                                   (length y-seq)
                                                   0
                                                   rate))
               (new-w-vector (car result))
               (new-b (cadr result))
               (m-total (caddr result)))
          ;    (newline)
          ;    (display "predict-next-price:  ")
          ;    (display (linear-next-price m-total new-w-vector new-b))
          result)
        )))

(define (linear-regression-with-singapore-price)
  (let* ((singapore-price-dataset ((screen 'acquire-dataset) 'singapore-price))
         (y-seq (->price-seq singapore-price-dataset)))
    (if (not y-seq)
        false
        (let* ((x-seq (z-scale-normalization
                       (enumerate-interval 1 (length y-seq))))
               (x-vector-seq (map (lambda (x)
                                    (list x))
                                  x-seq))
```

```
              (w-vector (list 0))
              (b 16680)
              (rate 0.9)
              (result (multiple-linear-regression w-vector
                                                  x-vector-seq
                                                  y-seq
                                                  b
                                                  (length y-seq)
                                                  0
                                                  rate))
              (new-w-vector (car result))
              (new-b (cadr result))
              (m-total (caddr result)))
           ;    (newline)
           ;    (display "predict-next-price:  ")
           ;    (display (linear-next-price m-total new-w-vector new-b))
           result))
     ))


(define (linear-next-price m-total w-vector b)
  ;;Note: In order to feature scale new-x, we have to use old standard deviation and mean.
  ;;      This redundant recalculation below should be refactored.
  (let* ((x-seq (enumerate-interval 1 m-total))
         (mean-sd-pair (calculate-mean-and-standard-deviation x-seq))
         (x-mean (car mean-sd-pair))
         (x-sd (cdr mean-sd-pair))
         (new-x (+ 1 m-total))
         (new-x-vector (list (/ (- new-x x-mean) x-sd))))
    (+ (dot-product new-x-vector w-vector) b)))
```

<<<——BACK—TO—FIRST

<<<——BACK—TO—LAB–EXERCISE–4

(load "c:\\6.001\\work\\assignment\\concurrency\\lab-exercise4\\regression-lib\\multiple-linear-regression&feature-scaling.scm")

```
(define (root-x-predict-price total-m w-vector b)
  ;;Note: In order to feature scale new-x, we have to use old standard deviation and mean.
  ;;      This redundant recalculation below should be refactored.
  (let* ((new-x (+ total-m 1))
         (seq-x-vector (map (lambda (x) (list x (sqrt x)))
                            (enumerate-interval 1 total-m)))
         (x-seq (map (lambda (vector) (car vector))
                     seq-x-vector))
         (root-x-seq (map (lambda (vector) (cadr vector))
                          seq-x-vector))
         (x-mean-sd-pair (calculate-mean-and-standard-deviation x-seq))
         (x-mean (car x-mean-sd-pair))
         (x-sd (cdr x-mean-sd-pair))
         (root-x-sd-pair (calculate-mean-and-standard-deviation root-x-seq))
         (root-x-mean (car root-x-sd-pair))
         (root-x-sd (cdr root-x-sd-pair))
         (x-vector (list (/ (- new-x x-mean)
                            x-sd)
                         (/ (- (sqrt new-x) root-x-mean)
                            root-x-sd))))
    (+ (dot-product x-vector w-vector) b)))


(define (x+x^1/2-regression-with-tokyo-price)
  (let* ((tokyo-price-dataset ((screen 'acquire-dataset) 'tokyo-price))
         (y-seq (->price-seq tokyo-price-dataset)))
    (if (not y-seq)
        false
        (let* ((feature-engineered-x (map (lambda (x) (list x (sqrt x)))
                                          (enumerate-interval 1 (length y-seq))))
               (x-seq (z-scale-normalization
                       (map (lambda (x) (car x))
                            feature-engineered-x)))
               (root-x-seq (z-scale-normalization
                            (map (lambda (x) (cadr x))
                                 feature-engineered-x)))
               (x-vector-seq (map (lambda (x root-x)
                                    (list x root-x))
                                  x-seq
                                  root-x-seq))
```

```
              (w-vector (list 0 0))
              (b 0)
              (rate 0.9)
              (result (multiple-linear-regression w-vector
                                                  x-vector-seq
                                                  y-seq
                                                  b
                                                  (length y-seq)
                                                  0
                                                  rate))
              (new-w-vector (car result))
              (new-b (cadr result))
              (m-total (caddr result)))

          (newline)
          (display "predict-next-price:  ")
          (display (root-x-predict-price m-total new-w-vector new-b))
          result))
    ))

(define (x+x^1/2-regression-with-singapore-price)
  (let* ((singapore-price-dataset ((screen 'acquire-dataset) 'singapore-price))
         (y-seq (->price-seq singapore-price-dataset)))
    (if (not y-seq)
        false
        (let* ((feature-engineered-x (map (lambda (x) (list x (sqrt x)))
                                          (enumerate-interval 1 (length y-seq))))
              (x-seq (z-scale-normalization
                       (map (lambda (x) (car x))
                            feature-engineered-x)))
              (root-x-seq (z-scale-normalization
                            (map (lambda (x) (cadr x))
                                 feature-engineered-x)))
              (x-vector-seq (map (lambda (x root-x)
                                   (list x root-x))
                                 x-seq
                                 root-x-seq))
              (w-vector (list 0 0))
              (b 0)
              (rate 0.9)
              (result (multiple-linear-regression w-vector
                                                  x-vector-seq
                                                  y-seq
                                                  b
                                                  (length y-seq)
```

```
                                                    0
                                                    rate))
            (new-w-vector (car result))
            (new-b (cadr result))
            (m-total (caddr result)))

        ;    (newline)
        ;    (display "predict-next-price:  ")
        ;    (display (root-x-predict-price m-total new-w-vector new-b))
        result))
    ))
```

<<<——BACK—TO—FIRST

<<<——BACK—TO—LAB–EXERCISE–4

```
(define (calculate–mean–and–standard–deviation seq)
  (let* ((count (length seq))
           (mean (/ (accumulate + 0 seq) count))
           (standard–deviation
            (sqrt
             (/
              (accumulate + 0
                         (map
                          (lambda (x) (square (– x mean)))
                          seq))
              count))))
    (cons mean standard–deviation)))



(define (z–scale–normalization seq)
  (let* ((mean–mu–pair (calculate–mean–and–standard–deviation seq))
           (mean (car mean–mu–pair))
           (mu (cdr mean–mu–pair)))
    (map (lambda (x) (/ (– x mean) mu))
           seq)))



(define (multiple–linear–regression w–vector x–vector–seq y–seq b m–total last–cost–func–
value learning–rate)
  (let ((new–cost–func–value (compute–cost–function w–vector x–vector–seq y–seq b m–
total)))
;    (newline)
;    (display "cost–function––value:  ")
;    (display last–cost–func–value)
          (cond ((< (abs (– last–cost–func–value new–cost–func–value)) 0.001)

                 (list w–vector b m–total))
                (else
                 (let* ((derivatives–seq
                          (seq–values–of–partial–derivative–respect–to–w x–vector–seq
                                                                       y–seq
                                                                       w–vector
                                                                       b
                                                                       m–total))

                         (new–w–vector (map (lambda (w derivative)
```

```
                                  (- w (* learning-rate
                                         derivative)))
                                w-vector
                                derivatives-seq))

                 (new-b (- b
                            (* learning-rate
                              (value-of-partial-derivative-respect-to-b x-vector-seq
                                                                        y-seq
                                                                        w-vector
                                                                        b
                                                                        m-total))))
                 )
            (multiple-linear-regression new-w-vector
                                        x-vector-seq
                                        y-seq
                                        new-b
                                        m-total
                                        new-cost-func-value
                                        learning-rate)))))
  )
```

```
(define (seq-values-of-partial-derivative-respect-to-w x-vector-seq seq-y w-vector b m-
total)
  ;;Type-to-return: list<calculated-partial-deriv-repect-to-wi>
  (map
   (lambda (j)       ;;for each feature j1,j2....

     (/
      (accumulate +     ;;add  all <error*xj> products up
                  0
                  (map (lambda (x-vector y)        ;;for-each pair x-vector and y
                         (* (compute-error x-vector w-vector y b)
                            (list-ref x-vector (- j 1)))     ; error * x-j
                         )
                       x-vector-seq
                       seq-y
                       ))
      m-total))
```

```
   (enumerate-interval 1 (length w-vector)))  ;(1,2,3)

  )


(define (value-of-partial-derivative-respect-to-b x-vector-seq seq-y w-vector b m-total)
  (/ (accumulate +
                  0
                  (map (lambda (x-vector y)
                         (compute-error x-vector w-vector y b))
                       x-vector-seq
                       seq-y))
     m-total))

(define (compute-error x-vector w-vector y b)
  (- (+
      (dot-product x-vector w-vector)
      b)
     y)
  )

(define (dot-product v1 v2)
  (accumulate + 0 (map * v1 v2)))

(define (compute-cost-function w-vector x-vector-seq y-seq b m-num)
  (/
   (accumulate + 0 (map (lambda (x-vector y)
                            (square
                             (- (+ b (dot-product x-vector w-vector))
                                y)
                             )
                            )
                          x-vector-seq
                          y-seq))
   (* 2 m-num)))


(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define (enumerate-interval low high)
  (if (> low high)
```

```
nil
(cons low (enumerate-interval (+ low 1) high))))
```

<<<——BACK—TO—FIRST

<<<——BACK–TO–CONCURRENCY Go to nikkei–sourceCode1—>>

```
;;;;         England lost her Barings

;;; The collapse of the Baring Brothers establishment
;;;  provides an abject Leeson for the banking community.
;;;  -- GJS

(define (make-market name initial-price)
```
TO CLASS DIAGRAM—>>>>
```
  (let ((price initial-price)
        (price-serializer (make-serializer))
        (pending-orders '())
        (orders-serializer (make-serializer)))
    (define (the-market m)
      (cond ((eq? m 'new-price!)
             (price-serializer
                (lambda (update)
                  (set! price (update price)))))
            ((eq? m 'get-price)
             price)
            ((eq? m 'new-order!)
             (orders-serializer
              (lambda (new-order)
                 (set! pending-orders
                       (append pending-orders
                                 (list new-order))))))
            ((eq? m 'execute-an-order)
             (((orders-serializer
                 (lambda ()
                   (if (not (null? pending-orders))
                       (let ((outstanding-order (car pending-orders)))
                           (set! pending-orders (cdr pending-orders))
                           outstanding-order)
                       (lambda () 'nothing-to-do)))))))
            ((eq? m 'get-name) name)
            (else
             (error "Wrong message" m))))
    the-market))


;;; Using this we make two markets, both starting at the same price:

(define nikkei-fundamental 16680.0)

(define tokyo
```

```
  (make-market "Tokyo:Nikkei-225" nikkei-fundamental))

(define singapore
  (make-market "Singapore:Nikkei-225" nikkei-fundamental))
```

```
;;; Traders buy and sell contracts at a market using TRANSACT.
;;;  The trader gives the market permission to subtract from the
;;;  trader's monetary balance the cost of the contracts purchased
;;;  and to add to the trader's stash the contracts he purchased.

(define (buy ncontracts market permission)
  (transact ncontracts market permission))

(define (sell ncontracts market permission)
  (transact (- ncontracts) market permission))

(define (transact ncontracts market permission)
  ((market 'new-order!)
   (lambda ()
     (permission (- (* ncontracts (market 'get-price)))
                    ncontracts))))
```

```
(define transaction-cost 1.0)
```

<<—BACK TO FIRST

```scheme
(define (make-arbitrager name balance contracts authorization)
  (let ((trader-serializer (make-serializer)))
```

TO CLASS DIAGRAM—>>>>

```scheme
    (define (change-assets delta-money delta-contracts)
      ((trader-serializer
         (lambda ()
           (set! balance (+ balance delta-money))
           (set! contracts (+ contracts delta-contracts))))))
```

<<—BACK TO EXECUTE-AN
-ORDER DIAGRAM

```scheme
    (define (a<b low-place low-price high-place high-price)
      (if (> (- high-price low-price) transaction-cost)
           (let ((amount-to-gamble (min authorization balance)))      ;;authorization = 10000
             (let ((ncontracts                                        ;round to nearest integer
                      (round (/ amount-to-gamble (- high-price low-price)))))
               (buy ncontracts low-place change-assets)
               (sell ncontracts high-place change-assets)))))

    (define (consider-a-trade)
      (let ((nikkei-225-tokyo (tokyo ‘get-price))
              (nikkei-225-singapore (singapore 'get-price)))
         (if (< nikkei-225-tokyo nikkei-225-singapore)
              (a<b tokyo nikkei-225-tokyo
                     singapore nikkei-225-singapore)
              (a<b singapore nikkei-225-singapore
                     tokyo nikkei-225-tokyo))))
```

<<— GO BACK TO
CONSIDER-TRADE-DIAGRAM

```scheme
    (define (me message)
      (cond ((eq? message 'name) name)
              ((eq? message 'balance) balance)
              ((eq? message 'contracts) contracts)
              ((eq? message 'consider-a-trade) (consider-a-trade))
              (else
               (error "Unknown message -- ARBITRAGER" message))))
    me))

(define nick-leeson
  (make-arbitrager "Nick Leeson" 1000000000. 0.0 10000.))
```

<<—BACK TO FIRST

```
;;; The following parameters determine the way the Nikkei average
;;;  drifts over time, and how the two markets differ.  The details of
;;;  this are probably not very much like a real market, and they
;;;  probably do not matter very much for this problem set.

(define nikkei-drift +0.1)
(define nikkei-split +1.0)

(define (nikkei-update)
  (define (average x y) (/ (+ x y) 2.0))
  (gaussian-random-pair
   (lambda (d1 d2)
     ;; d1 and d2 are both distributed as Gaussians
     ;;  with zero mean and unity standard deviation.
     ;; The fundamental drift
     (set! nikkei-fundamental
           (+ nikkei-fundamental
              nikkei-drift
              (* nikkei-split (+ d1 d2))))
     ;; The actual split -- cannot be modified
     ((tokyo 'new-price!)
      (lambda (old-price)
         (+ (average nikkei-fundamental old-price)
           (* nikkei-split d1))))
     ((singapore 'new-price!)
      (lambda (old-price)
           (+ (average nikkei-fundamental old-price)
              (* nikkei-split d2)))))))

(define (ticker market)
  (newline)
  (display (market ‘get-name))
  (display "                ")
  (display (market 'get-price)))
```

<<<——BACK TO nikkei-update
<<<——sequence diagram
<<<——BACK TO USE-CASE DIAGRAM

```
(define (audit trader)
  (newline)
  (display (trader 'name))
  (display "     Balance: “)
  (display (trader 'balance))
  (display "     Contracts: ")
  (display (trader ‘contracts)))
```

<<—BACK TO FIRST

```
;;; Delay times are in milliseconds.
(define (do-aperiodically thunk max-delay)
  (define (again)
    (thunk)
    (sleep-current-thread (random max-delay))
    (again))
  again)

(define (do-periodically thunk time)
  (define (again)
    (thunk)
    (sleep-current-thread time)
    (again))
  again)


;;; Students: Please ignore the following hairy expression.
;;;  It is only here to allow us to reload this file into a
;;;  running simulation.
(if (lexical-unreferenceable? user-initial-environment 'stop-world)
    (define stop-world false))

;;; The following starts a bunch of threads running in parallel
;;;  and makes a procedure to stop all of the parallel threads.

(define (start-world)
    (if stop-world (stop-world))
    (set! stop-world
            (parallel-execute
             (do-aperiodically (lambda ()
                                (nikkei-update))
                              3000)
             (do-aperiodically (lambda ()
                                (nick-leeson 'consider-a-trade))
                              4000)
             (do-aperiodically (lambda ()
                                (audit nick-leeson))
                              20000)
             (do-aperiodically (lambda ()
                                (tokyo ‘execute-an-order))
                              2000)
             (do-aperiodically (lambda ()
                                (singapore 'execute-an-order))
                              2500)
```
<<—BACK TO FIRST

```
        (do-periodically (lambda ()
                           (ticker tokyo))
                         10000)
        (do-periodically (lambda ()
                           (ticker singapore))
                         10000)
         ))
    ‘running)
```

<<—BACK TO FIRST

<<<—BACK—TO—CONCURRENCY

```
(declare (usual-integrations))

;;; UNIFORM-RANDOM produces an inexact number x,    0 <= x < 1

#|
(define uniform-random
  (let* ((random-max (expt 2 23))
         (frandom-max (exact->inexact random-max)))
    (lambda ()
      (/ (random random-max)
         frandom-max))))
|#

(define (uniform-random) (random 1.))

(define (nonzero-uniform-random)
  (let ((x (uniform-random)))
    (if (= x 0.)
        (nonzero-uniform-random)
        x)))

;;; Given uniform random numbers, we can produce pairs of
;;; gaussian-distributed numbers, with zero mean and unit
;;; standard deviation, by the following trick:

(define 2pi (* 4.0 (atan 1.0)))

(define (gaussian-random-pair #!optional continue)
  ;; continue = (lambda (y1 y2) ...)
  (let ((continue (if (default-object? continue) cons continue))
        (x1 (uniform-random))
        (x2 (uniform-random)))
    (let ((r (sqrt (* -2.0 (log x1)))))
      (continue (* r (cos (* 2pi x2)))
                (* r (sin (* 2pi x2)))))))

(define (gaussian-random)
  (gaussian-random-pair (lambda (x y) x)))

(define (gaussian-random-list d)
  (let lp ((j d) (t '()))
    (if (fix:= j 0)
        t
```

```
         (gaussian-random-pair
          (lambda (x1 x2)
            (if (fix:= j 1)
                (cons x1 t)
                (lp (fix:- j 2) (cons x1 (cons x2 t))))))))
```

;;; Makes a list of n 2-vectors of gaussian-distributed random numbers

```
(define (gaussian-random-pairs n)
  (if (fix:= n 0)
      '()
      (cons (gaussian-random-pair vector)
            (gaussian-random-pairs (fix:- n 1)))))
```

;;; Makes a list of n d-vectors of gaussian-distributed random numbers

```
(define (gaussian-random-tuples d n)
  (if (fix:= n 0)
      '()
      (cons (list->vector (gaussian-random-list d))
            (gaussian-random-tuples d (fix:- n 1)))))
```

;;; For adding zero-mean noise with a given standard deviation to a vector.

```
(define ((add-noise sigma) v)
  (list->vector (map (lambda (signal noise)
                       (+ signal (* sigma noise)))
                     (vector->list v)
                     (gaussian-random-list (vector-length v)))))
```

<<<—BACK—TO—FIRST

<<<——BACK—TO—CONCURRENCY

```
;;; To allow parallel execution of any number of thunks, for
;;; effect.  The values are discarded.

(define disallow-preempt-current-thread
  (access disallow-preempt-current-thread
          (->environment '(runtime thread))))

(define allow-preempt-current-thread
  (access allow-preempt-current-thread
          (->environment '(runtime thread))))

(define (kill-thread thread)
  (let ((event
         (lambda ()
           (exit-current-thread 'RIP))))
    (without-interrupts
     (lambda ()
       (case (thread-execution-state thread)
         ((STOPPED) (restart-thread thread #t event))
         ((DEAD) unspecific)
         (else (signal-thread-event thread event)))))))


(define (parallel-execute . thunks)
  (let ((my-threads '()))
    (define (terminator)
      (without-interrupts
       (lambda ()
         (for-each kill-thread my-threads)
         (set! my-threads '())
         unspecific)))
    (without-interrupts
     (lambda ()
       (set! my-threads
             (map (lambda (thunk)
                    (let ((thread (create-thread #f thunk)))
                      (detach-thread thread)
                      thread))
                  thunks))
       unspecific))
    terminator))

#|
```

<<<-BACK-TO-REPRODUCING-CONCURRENCY-BUGS

```
;;; IO system is not completely interlocked, so...

(define (try n)
  (parallel-execute
   (lambda ()
     (write-line 'hi)
     (let lp ((i 0))
       (if (< i 10000)
             (lp (1+ i))))
     (write-line 'gjs))
   (lambda ()
     (write-line 'there)
     (let lp ((i 0))
       (if (< i n)
             (lp (1+ i))))
     (write-line 'foo))))

(define foo (try 9188))
;Value foo

hi
there
foo
foo
gjs

(foo)
;No value
|#
```

```
(define (make-serializer)
  (let ((mutex (make-thread-mutex)))
    (define (serialized f)
      (define (serialized-f . args)
          (with-thread-mutex-locked mutex
                                          (lambda ()
                                            (apply f args))))
      serialized-f)
    serialized))

(define output-serialized (make-serializer))

(define write-line
  (output-serialized write-line))

(define display
  (output-serialized display))

(define write
  (output-serialized write))
#|
;;; This solves the IO interlock problem


(define (try n)
  (parallel-execute
   (lambda ()
     (write-line 'hi)
     (let lp ((i 0))
       (if (< i 10000)
             (lp (1+ i))))
     (write-line 'gjs))
   (lambda ()
     (write-line 'there)
     (let lp ((i 0))
       (if (< i n)
             (lp (1+ i))))
     (write-line 'foo))))

(define foo (try 9197))
;Value: foo

hi
there
```

```
gjs
foo

(define foo (try 9196))
;Value: foo

hi
there
foo
gjs
|#
```

<<<——BACK—TO—FIRST

<<<<——BACK—TO—MY—JOURNEY—OF—LISP—PROGRAMMING

;;This transcript is about a conversational system.

;The conversation system answer questions based on information of "catalog".
;Catalog contains information of different subjects taught at MIT.

;Catalog: ((8:01 physics (classical mechanics) 12 (gir physics) #f) (8:02 physics (electricity and magnetism) 12 (gir physics) (8:01 18:01)) (8:03 physics (waves) 12 (rest) (8:02 18:02)) (8:04 physics (quantum wierdness) 12 (rest) (8:03 18:03)) (18:01 math (elementary differential and integral calculus) 12 (gir calculus) #f) (18:02 math (multivariate calculus) 12 (gir calculus) (18:01)) (18:03 math (differential equations) 12 (rest) (18:02)) (18:04 math (theory of functions of a complex variable) 12 #f (18:03)) (6:001 eecs (scheming with abelson and sussman) 15 (rest) (true–grit)) (6:002 eecs (circuits) 15 (rest) (8:02 18:02)) (3:091 mms (munching and crunching stuff) 12 (gir chemistry) #f) (5:11 chemistry (smelly organic crud and goop) 12 (gir chemistry) (a–strong–stomach)) (7:012 biology (diseases and their applications) 12 (gir biology) #f) (7:013 biology (diseases and their applications) 12 (gir biology) #f) (7:014 biology (diseases and their applications) 12 (gir biology) #f) (12:001 eaps (rocks for jocks) 12 (rest) #f) (12:004 eaps (planets) 12 #f (18:03 8:02)))

```
(see–advisor 'professor)                          ;;start system
(hi professor)
(i am your freshman advisor)
(what are your plans for the semester)

** (what is 8:01 about)                           ;;question I asked
(8:01 is about classical mechanics)               ;;answered by system

** (what are 7:014 and 12:004 about)              ;;question I asked
(7:014 is about diseases and their applications)  ;;answered by system
(12:004 is about planets)                         ;;answered by system

** (how many units is 12:004)                     ;;question I asked
(12:004 is a 12 unit subject)                     ;;answered by system

** (can I take 5:11)                              ;;question I asked
(the prerequisites for 5:11 are a–strong–stomach) ;;answered by system
```

;;This conversational system can originally answer some very rigid questions like above.
;;;———

;;;However, if your questions don't match their specific form and certain keywords, you get
;;; some plausible but actually predefined sentences(randomly chosen).

```
** (can I take 8:02 if I have not taken 8:01)     ;;question asked
(students really enjoy 8:01)                      ;;predefined answer. no matter what you ask,
                                                  ;;it would answer "students really enjoy...."

** (I want to take 18:01)                         ;;question asked
(make sure to take some humanities)               ;;predefined answer

** (complex variable)                             ;;question asked
(make sure to visit harvard)                      ;;predefined answer
```

;;;;--
;;;after adding new rules, systems now can answer more types of questions
;;;based on the information defined in catalog(data base).

```
(see-advisor 'professor)                          ;;restarting system
(hi professor)
(i am your freshman advisor)
(what are your plans for the semester)

** (can I take 8:02 if I have not taken 8:01)     ;;When I ask "can I take ... if I have not taken...."
(no you cannot)                                   ;;system would recognize this pattern, then
                                                  ;;based on knowledge of "catalog",
                                                  ;;choose to answer "yes you can" or
                                                  ;;"no you cannot". Note the question form
                                                  ;;here is still rigid with very limited flexibility.

** (can I take 8:01 if I have not taken 8:02)
(yes you can)                                     ;;answered based on knowledge of "catalog"

** (i want to take 8:02)                          ;;Another rigid form "I want to take ...."
(you need the prerequisites: 8:01 18:01)          ;;answered based on knowledge of "catalog"
```

;;;;–––
;;;One particular rule wasted me lots of time is the open–ended question of problem set lab exercise 9.
;;;now, mention any words in the summury of subject, system can
;;;answer subject's basic description.(draw from the data base catalog)

** (anything munching)   ;;Notice more flexible question can be asked now
(3:091 is about munching and crunching stuff)   ;;answered based on catalog

** (looking for crud)   ;;question now can be asked naturally
(5:11 is about smelly organic crud and goop)   ;;answered based on catalog

** (jocks)   ;;or just one word
(12:001 is about rocks for jocks)   ;;answered based on catalog

** (love complex variable)   ;;another natural question.
(18:04 is about theory of functions of a complex variable)

For reference,
My code: see mywork–adv.pdf
Problem set: https://mitp–content–server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/psets/ps5adv/readme.html
Notes: Notes are in NoteOnPatternMatcher and NotesForRetrival.

<<—BACK—TO—TRANSCRIPT—FRESHMAN—ADVISOR

;;created 16/10/2022 20:55

```
(load "c:\\6.001\\work\\assignment\\freshmanadvisor\\preload.scm")


;;prelab exercise5:

(define (proc symbols)
  (reduce list-union '() (map list symbols)))

;;Answer: delete replicated element.



;;Lab Exercise 1:
(define beginnings
  '((you say)
    (why do you say)
    (i am glad to hear that)
    (fabricated-interesting when you say)
    (fabricated-good point)
    ()))


(define general-advice
  '((make sure to take some humanities)
    (mit has a lot of interesting departments)
    (make sure to get time to explore the Boston area)
    (how about a freshman seminar)
    (fabricated- make sure to visit harvard)))


;;lab Exercise2:
;;see note.txt   and   labexercise2.scm



;;lab Exercise3:
;;(what are the prerequisites for 12:004)
;;(in-catalog '6:101 (lambda () false) (lambda (x y) (* 3 3)))

(define (all-prerequisites subject)
```

```
  (in-catalog subject (lambda () '()) (lambda (dict fail)
                                               (let ((prerequisite (entry-prerequisites dict)))
                                                 (reduce list-union prerequisite (map all-prerequisites
prerequisite)))))
  )
```

;;Lab Exercise 4
;;see below end of page.

;;Lab Exercise5

```
(define (filter predicate sequence)
;;filter was defined in adv.scm as a helper procedure!
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))

(define (remove item sequence)
  (filter (lambda (x) (not (equal? x item)))
          sequence))



(define (check-circular-prerequisites? subjects)

  (define (combine a b) (and a b))

  (reduce combine true
          (reduce list-union '()
                  (map (lambda (i)
                         (let ((prereq (all-prerequisites i)))
                           (map (lambda (j)
                                  (let ((overlap (list-intersection prereq
                                                                    (list j))))
                                    (if (null? overlap) true
                                        false)))
                                (remove i subjects))))
```

```
                           subjects)
                      )
             )
  )


;;;iterative version
(define (check-circular-prerequisites2? subjects)
  (define (iter n seq)
    (display "subjects:  ") (display seq) (newline)
    (let ((current (car seq))
            (rest (cdr seq)))
      (let ((seq-current (all-prerequisites current)))
          (display "current:  ") (display current) (newline)
          (display "rest:    ") (display rest) (newline)
          (display "seq-current:   ") (display seq-current) (newline)
          (cond ((= n 0) true)
                ((not (null? (list-intersection seq-current
                                                rest)))
                 false)
                (else (iter (- n 1) (append rest (list current))))))))

  (iter (length subjects) subjects)
  )


;;more test to check-circular-prerequisites?
;;compare check-circular-prerequisites with check-...sites2?




;;Lab Exercise 6

(define (total-units subjects)
  (reduce + 0 (map (lambda (subject)
                      (in-catalog subject
                                  (lambda () 0)
                                  (lambda (dict fail) (entry-units dict))))
                    subjects)))




;;Lab Exercise 7
```

```scheme
(define (check-subject-list subjects)
  (let ((freshman-credit 54))
    (cond ((not (check-circular-prerequisites? subjects))
             (write-line '(These subjects cannot be taken at the same time)))
            ((> (total-units subjects) freshman-credit)
             (write-line '(Too much subjects for a freshman to take!)))
            (else (write-line (append '(You need the fucking prerequisites:)
                                        (reduce list-union '() (map all-prerequisites subjects))))))))


;;test more


;;Lab Exercise 8. See lab Exercise4 subject-knowledge.




;;Lab Exercise 9. see note.scm

(define (find-info subject selector entries)
  (cond ((null? entries) false)
          ((eq? subject (selector (car entries)))
           (car entries))
          (else
           (find-info subject (cdr entries)))))



(define (find-summary keyword entries)
  ;;implement in the same sence of find-subject
  ;;provided by the adv.scm
  ;;without a handy procudural abstraction called memq, is difficult
  ;;to come up with this simple control structure.
  (cond ((null? entries) false)
          ((memq keyword (entry-summary (car entries)))
           (car entries))
          (else (find-summary keyword (cdr entries)))))


(define (in-summary keyword fail succeed)
  ;;This the in-catalog style(continuation passing style) about summary
  ;;summary is description about the subject.
  (let ((entry (find-summary keyword catalog)))
```

```
  (if entry
      (succeed entry fail)
      (fail))))



(define (summarys words fail succeed)      ;;constraint 1

  ;;This procedure has a similar style compare with subject-seq in adv.scm
  ;;entry is a data type from assignment text.

  ;;type:(list<symbol>,lambda,lambda)->list<entry>

  ;;Constraints:
  ;;1.summarys should be used as a segment variable predicate.
  ;;2.rules are written in a recursive fashion.
  ;;3.first rule check if list of words has only one word. e.g (electricity)
  ;;4.second rule check rest of the words recursively since we pass the first rule.
  ;;  Note:try-rule will try rules in order.
  ;;5.we either return a element variable or segment variable from action.
  ;;  Note:Segment variable is a pattern variable denoted as (?? x predicate),
  ;;       and element variable is a pattern variable denoted as (? x predicate),
  ;;       in both cases predicate is optional.(see assignment text)
  ;;6.In subject-seq,rule action put list on the entry due to the use in
  ;;  subject-knowledge.(for-each......)It is unneccessary to do it here.
  (try-rules
   words
   (list
    (make-rule
     `((? summary , in-summary))
     (lambda (dict)
       (value 'summary dict)))       ;;constraint 6
    (make-rule
     `((? summary1) (?? more-summary , summarys))
     (lambda (dict)
       (value 'more-summary dict)) )  ;;constraint 6
    )
   fail
   succeed))


(define subject-knowledge
  (list

;;lab Exercise 9
```

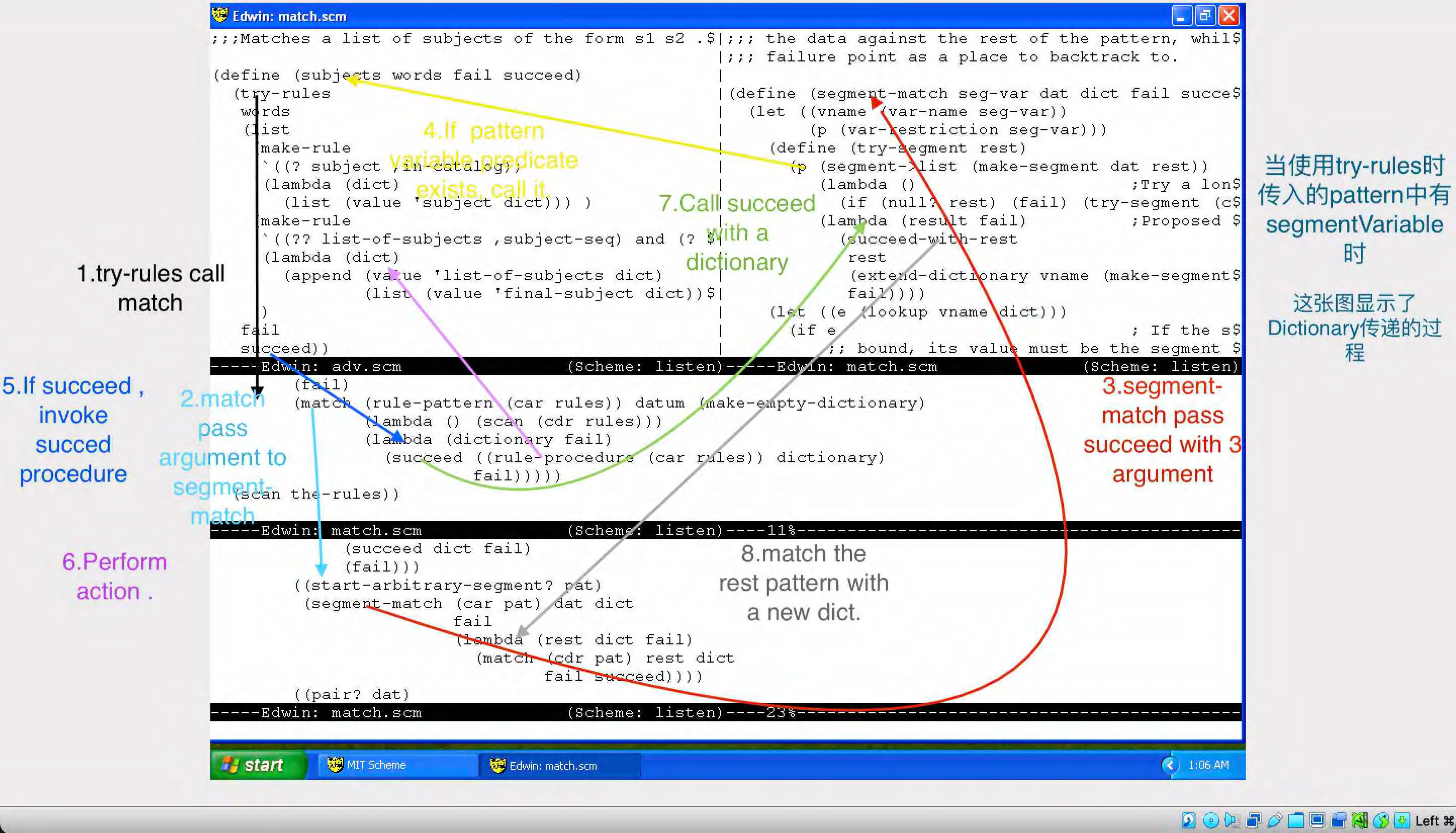
```
  (make-rule
   `((?? words , summarys))
   (lambda (dict)
;      (display (value 'words dict))
     (let ((entry (value 'words dict)))
        (write-line (append (list (entry-subject entry))
                            '(is about)
                            (entry-summary entry))))))


;;lab Exercise 8
  (make-rule
   `(i want to take (?? s , subjects))
   (lambda (dict)
     (check-subject-list (map entry-subject (value 's dict)))))


;;lab Exercise 4: Test see TestExercise4.scm
  (make-rule
   '(can I take (? x) if I have not taken (?? y))
   (lambda (dict)
     (let ((entryX (value 'x dict))
           (entryY (value 'y dict)))
        (let ((overlap (list-intersection (all-prerequisites entryX) entryY)))
          (if (not overlap) (write-line '(Yes you can))
              (write-line '(No you cannot)))))))

;;;;------original--rule---from---adv.scm---------------dividing-------------
line------------------------------
  (make-rule
   `(what is (? s ,in-catalog) about)
   (lambda (dict)
     (let ((entry (value 's dict)))
        (write-line
         (append (list (entry-subject entry))
                   '(is about)
                   (entry-summary entry))))) )

  (make-rule
   `(what are (?? s ,subjects) about)
   (lambda (dict)
     (for-each (lambda (entry)
                   (write-line
                    (append (list (entry-subject entry))
                            '(is about)
```

```
                              (entry-summary entry))))
                 (value 's dict))) )

(make-rule
 `(how many units is (? s ,in-catalog))
 (lambda (dict)
   (let ((entry (value 's dict)))
       (write-line
        (append (list (entry-subject entry))
                  '(is a)
                  (list (entry-units entry))
                  '(unit subject))))) )

(make-rule
 `(how many units are (?? s ,subjects))
 (lambda (dict)
   (for-each (lambda (entry)
                  (write-line
                   (append (list (entry-subject entry))
                             '(is a)
                             (list (entry-units entry))
                             '(unit subject))))
                 (value 's dict))) )

(make-rule
 `(what are the prerequisites for (?? s ,subjects))
 (lambda (dict)
   (for-each (lambda (entry)
                  (write-line
                   (append '(the prerequisites for)
                             (list (entry-subject entry))
                             '(are)
                             (entry-prerequisites entry))))
                 (value 's dict))) )

(make-rule
 `(can I take (? s ,in-catalog))
 (lambda (dict)
   (let ((entry (value 's dict)))
       (write-line
        (append '(the prerequisites for)
                  (list (entry-subject entry))
                  '(are)
                  (entry-prerequisites entry))))) )
```

))

;;;finished on 26/11/2022 18:20

<<<——BACK—TO—FIRST


Edwin: match.scm
;;;Matches a list of subjects of the form s1 s2 .$
(define (subjects words fail succeed)
(try-rules
words
(list
make-rule
`((? subject ,in-catalog))
(lambda (dict)
(list (value 'subject dict))))
make-rule
`((?? list-of-subjects ,subject-seq) and (? $
(lambda (dict)
(append (value 'list-of-subjects dict)
(list (value 'final-subject dict))$
)
fail
succeed))
;;; the data against the rest of the pattern, whil$
;;; failure point as a place to backtrack to.
(define (segment-match seg-var dat dict fail succe$
(let ((vname (var-name seg-var))
(p (var-restriction seg-var)))
(define (try-segment rest)
(p (segment->list (make-segment dat rest))
(lambda () ;Try a lon$
(if (null? rest) (fail) (try-segment (c$
(lambda (result fail) ;Proposed $
(succeed-with-rest
rest
(extend-dictionary vname (make-segment$
fail))))
(let ((e (lookup vname dict)))
(if e ; If the s$
;; bound, its value must be the segment $
-----Edwin: adv.scm (Scheme: listen)-----Edwin: match.scm (Scheme: listen)
(fail)
(match (rule-pattern (car rules)) datum (make-empty-dictionary)
(lambda () (scan (cdr rules)))
(lambda (dictionary fail)
(succeed ((rule-procedure (car rules)) dictionary)
fail)))))
(scan the-rules))
-----Edwin: match.scm (Scheme: listen)----11%-----
(succeed dict fail)
(fail)))
((start-arbitrary-segment? pat)
(segment-match (car pat) dat dict
fail
(lambda (rest dict fail)
(match (cdr pat) rest dict
fail succeed))))
((pair? dat)
-----Edwin: match.scm (Scheme: listen)----23%-----
1.try-rules call match
2.match pass argument to segment-match
3.segment-match pass succeed with 3 argument
4.If pattern variable predicate exists, call it.
5.If succeed , invoke succed procedure
6.Perform action .
7.Call succeed with a dictionary
8.match the rest pattern with a new dict.
当使用try-rules时传入的pattern中有segmentVariable时
这张图显示了Dictionary传递的过程
start
MIT Scheme
Edwin: match.scm
1:06 AM
Left ⌘

<<<<——BACK—TO--FRESHMAN--ADVISOR

;;created 21/10/2022
;;read lab exercise9 and plan the assignment and start. 21/20/2022

;;Read paper :lisp a language for stratified design. 23/10/2022

;;lab Exercise2:
;Understand how advisor generates the response.

;;why do I need to say xx and yyy in order to match ??s ?
;;answer: because that how it implemented.
;;For record see labexercise2.scm.   25/10/2022

;;lab Exercise3:
;;1. relationship between translate-and-run and try-rules?
;;2. in catalog-fail-succeed. succeed is a procedure.what is fail?
;;video lecture! or paper!

;;0.study the paper first,then answer the following questions if neccessary.
;;1.study the matcher and try-rules.And continuation procedure.
;;Start from translate-and-run!
;;2.I should be able to compare it with the one in lecture!
;;3.what is restrict-segment? what is restriction descibed in the lecture?
;;27/10/2022

;;Why (? s, in-catalog) work? where invoke in-catalog?
;;Answer: element-match in match.scm
;;29/10/2022

;;How does `(?? s , subjects) work? It is different to `(? s , in-catalog)!!!
;;`(? s , in-catalog) match the syntax described in the AIM-986 paper, and
;; it is handled by element-match , but `(?? s , subjects) is a magic!!!!
;;Answer: There is a procedure called var-restriction defined in the pattern
;;   abstraction, which takes input either (? s procedure) or (?? s procedure)
;;   or (? s) (?? s).
;;05/11/2022 20:08

;;;---======SUMMARY=====--------
;;There are two pattern variable predicates: in-catalog and subjects.

```
;;both of which are implenmented in continuation-passing-style.
;;The usage of rule with predicate procedure are written in a uniform way.
;;e.g.`(how many units is (? s , in-catalog))  `(how many units are (?? s, subjects))

;;Note the `(xxxxx , yy) is a scheme's quotation trick, as opposed to '(xxxxx , yy)
;; yy would be evaluated to its actual value.
;;Therefore the acutal rule returned by `(how many units is (? s , in-catalog))
;;is a nested list which is in the form of (how many units is (? s #[compound-procedure xx]))
;;where the inner list contains 3 elements.
;;06/11/2022


;;;;---Thoughts and notes for exercise9.


; TRY algebric simplification or symbolic differentiation.

;; Understand segment-match! (should come back for later study!!)
;; Why need to write 'and ? 'and is related to subjects.
;; Answer:see Note.scm on exercise2.

;;Then do Exericise 9.

;;Note: Try-rules are trying each rule to EVERY subexpression of the Input!!!
;;     if rule matches a subexpression , instantiate skeleton against the dictionary.
;;     then simplify new expression again until no longer changes.
;;Question :1. Is advisor matcher as complicated as the one presented in the lecture?
;;          2. How are subjects and subject-seq (sort of nested predicate) related to the
;;             concept of simplification as presented in the lecture?

;;Read paper and advisor assignment again then answer the question above. 08/11/2022


;;what question should user type at the prompt?
;;answer:simple response like 6:001 in conventional-wisdom.
;;The idea of rule based programming?
;;Reread the paper! 09/11/2022

;;Note: There is one succeed and one fail procedure in a matcher.
;;     both of which are passing all the way back to the original procedure
;;     which calls try-rules.And try-rules handle the instantiation before
;;     passing back the success to its original place.

;;Note:The magic of pattern variable predicate procedure "subjects" is a
;;     nested succeed procedure in the sense of closure property described
```

;;   in 2.2.4 peter henderson?
;; 11/11/2022

;;Because of the process of a matcher behave,try-rules will break
;;a sentence into pieces. ;;12/11/2022 18:05

;;search for a summary which match the keyword.
;;continuation style searching for subject->summary?
;;subject-seq refer itself in a recursive fashion!!

;;study the process of subjects and subject-seq!!12/11/2022 18:55

;;pattern match against the subjects detail.
;;(? x summary) (? x department)

;;watch the video lecture.focus on the introduction to rule programming.
;;which was the idea to view recursion as rules,
;;which have two parts, a left-hand side a right hand side.

;;before implement summarys using try rules, answer following question first:
;;1.is the analogy about try-rules true?(enzyme and stew)

;;Note for the use scenario for segment variables.
;;`(what are (?? s , subjects) about) has one segment variable.
;; but in the note there are 4 segment variables, in order to introduce
;; the notion of backtracking and continuation passing style.
;;Qestions:
;; i was thinking one segment variable! Rethink in terms of 4 cases!!
;;Answer: There was a situation that a segment variable could be used in
;;   a repeated fashion.
;;Note: 15/11/2022

;;Note for segment-match:
;;a.segment-match proceeds with the rest of pattern and data if succeeded(with segment variable stored).16/11/2022
;;b.argument of segment-match seg-var is passed in in the form of (?? x predicate) or (?? x).

;;c.if the segment variable x exists, then it is a segment data abstraction.
;;d.As the note mentioned ,stored segment variable increase the length gradually.
;;e.Think of the process of succeed in try-rules and relate it to the predicate subjects.
;;;;It is some how continued//18/11/2022.

;;Qestions for segment-match:
;;1.How is that repeat fashion relate to the implementation of
;;segment-match?
;;2.what is segment!!   e.g.is there xxx about electricity?
;;3.With a stored segment variable,what will happen when segment-match again?
;;4.what does pattern variable predicate like subject-seq return?
;;5.how is the increasing length strategy of segment variable relate to the implementation of segment-match? 18/11/2022
;;6.what is the data passed in with segment-match?
;; Answer: same with the match,from try-rules from prompt input(a list of words).
;;7.Think this question first.Use notes above(particularly e)!!!!!!!
;;Question:
;;How does the predicate subjects pass succeed all the way back to try-rules,and
;;what happen after action invoking and dictionary passing wherein try-rules take place.
;;Answer:It invokes succeed procedure in segment-match/element-match,since that's where call
;;   predicate subjects.//19/11/2022

;;Reread the paper from the beginning see whether there is hint about
;;rule programming.
;;Answer:Yes,there is.   //21/11/2022

;;Can I use the pattern of in-catalog, subjects,subject-seq in order to
;;implement the segment variable predicate summarys?//19/11/2022
;;Answer:I can study the process of how it relates to user interactions.//23/11/2022

;;rethink the process of how it works.//20/11/2022 ...see figure noteOnPatternMatcher(sicp-assignment-folder on macbook)/21/11/2022

;;Note:What happen with the rule-actions in subjects/subject-seq are passing the dict
;;   inside out between rules that are nested through pattern variable predicate.
;;Explanation:
;; I can think of it as changing the name by getting out the info in the dict, and then
;; glue(extend) it back with a new name(used by outside rule closer to the user) but the same info.
;;Also, a dict bind a name,a segment(for backtracking) and datum.

;;With this understanding in mind, I can think about the question 9 now. 23/11/2022.

;;continue to copy the fashion subject–seq and subjects.see if it works.
;;Answer:see note in summarys procedure.(constraint 6)
;;think about the dictionary passing process of subject–seq!!!!!/23/11/2022
;;Answer:It has a recursive structure, which summarys implement in a similar way.

;;1.is the analogy about try–rules true?(enzyme and stew analogy from video lecture)
;;Answer:maybe yes,rethink!!.
;;2.what is segment? Note:segment variable increase
;; its length gradually.
;;3.Study compare–segment–to–list–head
;;Answer:compare–segment–to–list–head is for avoiding name confict.
;;   Try–segment implements the general strategy for increasing length of segment variable
;;   as described in the paper "lisp:a language for stratified design"
;;   Read notes in the segment–match.For recall find the screenshot in the sicp folder on
;;   MacOs host.
;;26/11/2022.

<<<—BACK—TO—FIRST

<<—BACK TO "My Journal Of Lisp Programming"

I have practiced 4 projects on Object oriented programming:

1. Cs61a project 3 : https://people.eecs.berkeley.edu/~bh/61a-pages/Volume1/CS%2061A%20Course%20Reader,%20Volume%201.html
2. 6.001 fall 1998 problem set 7 https://groups.csail.mit.edu/mac/classes/6.001/FT98/psets/ps7web/
3. 6.001 2005 project 4 https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computer-programs-spring-2005/pages/projects/
4. SICP sample assignment https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/psets/ps6/readme.html

Each project above requires learner to study the source codes that contain :

i. The object system.
ii. The base class and setup for the project.

The object systems within 4 project can be summarized by a paper titled "Object-oriented programming in Scheme", and a study note can be found in "`RecallOfObjectOrientedSystems`".

The rest of this document is a transcript depicting a simple taxi-calling simulation similar to Uber that I expanded based on the 3rd project mentioned above. The 3rd project was originally capable of simulating people traversing different locations.

```
(setup 'yang)
;Value: ready

(ask me 'look-around)

You are in student-center
You are not holding anything.
You see stuff in the room: counter-spell
There are no other people around you.
The exits are in directions: south east
;Value: ok

(ask me 'call-taxi 'eecs-hq)                ;  destination is eecs-hq

--- the-clock Tick 0 ---
taxi calculate path                         ; calculate path involve finding all-paths
```

element==(baker bexley student–center) ; then filter out one sequence contains destination.
taxi location==broom–taxi–company

––– the–clock Tick 1 –––
taxi arrived at–– baker

––– the–clock Tick 2 –––
taxi arrived at–– bexley

––– the–clock Tick 3 –––
taxi arrived at–– student–center   ;here pick up passenger “yang” or “me”

––– the–clock Tick 4 –––
taxi calculate path
element==(lobby–7 lobby–10 great–court legal–seafood edgerton–hall 34–301 eecs–hq)
location==student–center

––– the–clock Tick 5 –––
taxi arrived at–– lobby–7
yang moves from student–center to lobby–7

––– the–clock Tick 6 –––
taxi arrived at–– lobby–10
yang moves from lobby–7 to lobby–10

––– the–clock Tick 7 –––
taxi arrived at–– great–court
yang moves from lobby–10 to great–court

––– the–clock Tick 8 –––
taxi arrived at–– legal–seafood
yang moves from great–court to legal–seafood

––– the–clock Tick 9 –––
taxi arrived at–– edgerton–hall
yang moves from legal–seafood to edgerton–hall

––– the–clock Tick 10 –––
taxi arrived at–– 34–301
yang moves from edgerton–hall to 34–301
At 34–301 yang says –– Hi ben–bitdiddle

––– the–clock Tick 11 –––
taxi arrived at–– eecs–hq

yang moves from 34-301 to eecs-hq

<<—BACK TO FIRST

<<<——BACK TO OOP-STUDY

This document notes commonalities and differences between Object systems (
1. Cs61a project 3 : https://people.eecs.berkeley.edu/~bh/61a-pages/Volume1/CS%2061A%20Course%20Reader,%20Volume%201.html
2. 6.001 fall 1998 problem set 7 https://groups.csail.mit.edu/mac/classes/6.001/FT98/psets/ps7web/
3. 6.001 2005 project 4 https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computer-programs-spring-2005/pages/projects/
4. SICP sample assignment https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/psets/ps6/readme.html

)
referring to the paper "Object-oriented programming in Scheme"(https://dl.acm.org/doi/pdf/10.1145/62678.62720).

Section 2.1 in the paper discusses how to implement class in Scheme. 1,2,3,4 all implement class in the way paper described in 2.1, but 1 translates this straightforward implementation into modern OOP syntax (https://people.eecs.berkeley.edu/~bh/61a-pages/Volume2/OOP/ref-man.pdf). To see an class underlying structure in 1, use "show-class". Note in the paper, "operation" means "message", and "method" means object's method, "Operate" means "ask".

Section 2.2 in the paper discusses how to inherit behavior from other class. All four systems implement the same style of inheritance as described in 2.2 of the paper. However, https://people.eecs.berkeley.edu/~bh/61a-pages/Volume2/OOP/belowline.pdf (inheritance and delegation) mentions that this style of inheritance is called delegation. Delegation has one drawback in which parent cannot refer to local state of the children instances. In order to figure out the difference between delegation and inheritance, I believe the papers mentioned in this section are worth exploring .

Section 2.3 in paper discusses the general need for manipulations on the instance itself when executing operations. Systems 2 and 4 implement in the same way as described in the paper. Notes.ps in https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/psets/ps6/readme.html and lecture 18 in https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computer-programs-spring-2005/pages/lecture-notes/ give more detailed examples if you don't remember what is the scenario for self reference. System 1 constructs self-reference completely differently to the previous two. See https://people.eecs.berkeley.edu/~bh/61a-pages/Volume2/OOP/belowline.pdf. System3 follows a implementation described in https://www.youtube.com/watch?v=2G5Yg-sOe9Q&list=PL7BcsI5ueSNFPCEisbaoQ0kXIDX9rR5FF&index=15 , which passing "self" as the first argument when constructing the class.

Section 2.4 in the paper discusses the scenario when the user want to make use of the inherited behavior from the parent and additional extended actions. As the paper describes, there are two common situations when deploying the parent's method: one is passing "composite" as the self argument and the other one is passing "component" as the self argument. In the above four OO systems, passing "composite" is more common, while passing "component" often leads to unintended result. See discussion of "ask vs delegate" in https://ocw.mit.edu/courses/6-001-structure-and-interpretation-of-computer-programs-spring-2005/7474a0354208c195ecd2fc1cfac7b7ae_lecture18_webhan.pdf to recall the difference.

The rest of the content of the paper describes more advanced topics related OOP which is worth studying since it relates to properties of small-talk and properties like fast-method invocation which seem relate to Java, C++. An excellent Object-Oriented programming lecture by Daniel Ingalls can be found at https://www.youtube.com/watch?v=iAQhlMqIoS0&list=PLhMnuBfGeCDNgVzLPxF9o5UNKG1b-LFY9&index=20 .

17/06/2024 17:15

<<—BACK TO FIRST

<<<——BACK—TO—MY—JOURNEY—OF—LISP—PROGRAMMING

;;This transcript has 2 parts:
;;part1 demonstrates computing greatest common divisor between 2 polynomials.

;;part2 demonstrates different implementations of polynomial can cooperate
;;without problems in doing arithmetic operation such as addition.
;;Moreover, coefficients of polynomial can be rational numbers other than only numbers.

For reference,
Part one and part two of this transcript are related to exercise on textbook 2.96 and 2.90.
See https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/full-text/book/book-Z-H-18.html#%_sec_2.5.3
I have included my solutions related to this transcript. See exercise2.90.pdf, exercise2.91.pdf, exercise 2.94.pdf, exercise 2.96.pdf.

;;part1

```
(define p1 (create-numerical-polynomial 'x '(1 -2 1)))        ;;numerical-polynomial
                                                               ;;means coefficient contains
;Value: "p1 --> (polynomial x (2 ...) (1 ...) (0 ...))"         ;;only integers.

(print-poly p1)                                     ;;"print-poly" is to print more readable
                                                    ;;form of polynomial
1x^2+-2x^1+1x^0.                                    ;;result is equivalent to x^2 - 2x + 1.
;Value: #[unspecified-return-value]

(define p2 (create-numerical-polynomial 'x '(11 0 7)))
;Value: "p2 --> (polynomial x (2 ...) (0 ...))"

(print-poly p2)
11x^2+7x^0.                                         ;;result is equivalent to 11x^2 + 7.
;Value: #[unspecified-return-value]

(define p3 (create-numerical-polynomial 'x '(13 5)))
;Value: "p3 --> (polynomial x (1 ...) (0 ...))"

(print-poly p3)
13x^1+5x^0.                                         ;; result is equivalent to 13x + 5.
;Value: #[unspecified-return-value]
```

```
(define product1 (mul p1 p2))                    ;;now multiply p1 by p2
;Value: "product1 --> (polynomial x (4 ...) (3 ...) (2 ...) (1 ...) (0 ...))"

(define product2 (mul p1 p3))                    ;;now multiply p1 by p3
;Value: "product2 --> (polynomial x (3 ...) (2 ...) (1 ...) (0 ...))"

(print-poly product1)
11x^4+-22x^3+18x^2+-14x^1+7x^0.
;Value: #[unspecified-return-value]

(print-poly product2)
13x^3+-21x^2+3x^1+5x^0.
;Value: #[unspecified-return-value]


;;;After we have created p1, p2, p3, product1, product2 we are going to show greatest
common divisor
;;; of product1 and product2 is equal to p1, since product1= p1*p2, product2= p1*p3, and
gcd(p2,p3)=1.
;;;Hence, gcd(product1, product2)= p1.


(greatest-common-divisor product1 product2)
;Value: (polynomial x (2 (number . 1)) (1 (number . -2)) (0 (number . 1)))

(print-poly (greatest-common-divisor product1 product2))      ; gcd(p1*p2, p1*p3)= p1
1x^2+-2x^1+1x^0.                                     ;result is equivalent to x^2 - 2x + 1
;Value: #[unspecified-return-value]

(equ? (greatest-common-divisor product1 product2) p1)   ;generic operation equ?
                                                        ;;can test different types
;Value: #t                                              ;of data include polynomial

;;--------------------------------------------------------------------------------
-----
;;part2
;;we can create different representation of polynomial: dense vs sparse.

;; dense-rep created below is equivalent to: (3/4)x^2 + 11x + 40
(define dense-rep (create-dense-polynomial 'x (list (create-rational (create-number 3)
                                                                     (create-number 4))
                                                    (create-number 11)
                                                    (create-number 40))))
```

```
;Value: "dense-rep --> (polynomial x dense 2 (rational ... number . 4) (number . 11) (number . 40))"


;; sparse-rep created below is equivalent to: (2/3)x^4 + 100x^2
(define sparse-rep (create-sparse-polynomial 'x (list (make-term 4 (create-rational
                                                                      (create-number 2)
                                                                      (create-number 3)))
                                                       (make-term 2 (create-number 100)))))
;Value: "sparse-rep --> (polynomial x sparse (term pair-term 4 rational ... number . 3) (term pair-term 2 number . 100))"


;;Now add dense-rep and sparse-rep to obtain sum of two polynomials.
;;VALUE of sum1 is COERCED to sparse representation.
;;Notice the type below (polynoial x SPARSE ...)

(define sum1 (add dense-rep sparse-rep))
;Value: "sum1 --> (polynomial x sparse (term pair-term 4 rational ... number . 3) (term pair-term 2 rational ... number . 4) (term pair-term 1 number . 11) (term pair-term 0 number . 40))"


;;VALUE of sum1 is equivalent to: (2/3)x^4 + (403/4)x^2 + 11x +40
sum1
;Value: (polynomial x sparse (term pair-term 4 rational (number . 2) number . 3) (term pair-term 2 rational (number . 403) number . 4) (term pair-term 1 number . 11) (term pair-term 0 number . 40))


;;we can also apply value to polynomial. for example, apply 2 to sum1 is equal to 5708/12.
(apply-polynomial sum1 (create-number 2))
;Value: (rational (number . 5708) number . 12)      ;;In order to obtain correct answer
                                                    ;;1427/3, polynomial need to be reduced
                                                    ;;using gcd function in part1 before applying
                                                    ;;number 2.
```

<<<——BACK-TO—FIRST

<<<—BACK—TO—TRANSCRIPT—2.5

```scheme
;;created 28/03/2023 19:56

(load "c:\\6.001\\work\\assignment\\genericarithmetic\\generic.scm")

;;-==============================generic--interface=====================
(define (create-dense-polynomial variable termlist)
  (make-polynomial (make-poly variable (create-dense-termlist termlist))))

(define (create-sparse-polynomial variable termlist)
  (make-polynomial (make-poly variable (create-sparse-termlist termlist))))

(define (create-numerical-polynomial variable L)
  (create-dense-polynomial variable (map
                                     (lambda (element)
                                       (if (number? element)
                                           (create-number element)
                                           (error
                                            "wrong type from create-numerical-polynomial: All
elements in list must be number")))
                                     L)))


(define (make-term order coeff)
  ;;note:A:term has two level tags:1.term and 2.pair-term
  ;;     which represent high level term concept and representation level concept
  ;;     which provide alternative representations' possibility.
  ;;     Term's two level tags is also helpful when its selectors order and coeff want to
  ;;     select from term in different procedures like "adjoin" which is in sparse package,
  ;;     and select from +terms which is public. Inside adjoin, term's "term" tag is stripped
  ;;     off so selector can only rely on "pair-term" tag. In +terms order and coeff can access
  ;;     parts through "term" tag. Both "term" and "pair-term" are installed for generic
selectors
  ;;     order and coeff.

  ;;     B:the two level tag benefit from mistakeversion2.90.scm
  ;;       see its notes for detail.

  ((get 'make 'term) order coeff))

(define (order term)
  (apply-generic 'order term))

(define (coeff term)
```

```
  (apply-generic 'coeff term))

(define (create-dense-termlist coeffseq)
  ((get 'make 'dense) coeffseq))

(define (create-sparse-termlist termseq)
  ((get 'make 'sparse) termseq))

(define (the-empty-termlist type)
  ;;note: There are two termlist empty type:
  ;;1.sparse--see sparse package
  ;;2.dense--see dense package
  (get 'the-empty-termlist type))

(define (empty-termlist? termlist)
  (apply-generic 'empty-termlist? termlist))

(define (adjoin-term term termlist)
  (apply-generic 'adjoin-term term termlist))

(define (first-term termlist)
  (apply-generic 'first-term termlist))

(define (rest-terms termlist)
  (apply-generic 'rest-terms termlist))

(define (map-terms proc termlist)
  ;;note:map-terms works as a coercion procedure dense->sparse or sparse->sparse
  (if (empty-termlist? termlist)
      (the-empty-termlist 'sparse)
      (adjoin-term (proc (first-term termlist))
                   (map-terms proc (rest-terms termlist)))))

(define (negate-terms termlist)
  (map-terms (lambda (term) (make-term (order term) (negate (coeff term))))
             termlist))


;;;=================termlist---operations-========================
;;Note: +terms and * terms are modified from ps5-code.scm . The only odification is coercion

(define (+terms L1 L2)
  ;;Constraints:
  ;;1.coerce to sparse-termlist when empty using map-terms
  (cond ((empty-termlist? L1) (map-terms (lambda (x) x) L2)) ;constraint1
```

```
        ((empty-termlist? L2) (map-terms (lambda (x) x) L1)) ;constraint1
        (else
         (let ((t1 (first-term L1)) (t2 (first-term L2)))
           (cond ((> (order t1) (order t2))
                  (adjoin-term t1
                               (+terms (rest-terms L1) L2)))
                 ((< (order t1) (order t2))
                  (adjoin-term t2
                               (+terms L1 (rest-terms L2))))
                 (else
                  (adjoin-term (make-term (order t1)
                                          (add (coeff t1)
                                               (coeff t2)))
                               (+terms (rest-terms L1)
                                       (rest-terms L2)))))))))

(define (*terms L1 L2)
  ;;constraints:
  ;;1. coerce to sparse-termlist when empty using the-empty-termlist by
  ;; passing tag sparse.
  (if (empty-termlist? L1)
      (the-empty-termlist 'sparse) ;;constraint 1
      (+terms (*-term-by-all-terms (first-term L1) L2)
              (*terms (rest-terms L1) L2))))


;;;   (RepTerm, RepTerms) --> RepTerms
(define (*-term-by-all-terms t1 L)
  ;;constraints:
  ;;1.coerce to sparse-termlist using map-terms
  (map-terms (lambda (term) (*term t1 term)) L)) ;;constraint1



;;;============dense-termlist-package==============================

(define (install-dense-package)

  (define (create-dense denselist)
    (make-termlist (- (length denselist) 1) denselist))

  (define (make-termlist order coefflist)
    (cons order coefflist))

  (define (order-termlist termlist)
```

```
    (car termlist))

  (define (coefflist-termlist termlist)
    (cdr termlist))

  (define (the-empty-termlist)
    (make-termlist 0 '()))

  (define (empty-termlist? termlist)
;    (display "termlist:--") (display termlist) (newline)
    (null? (coefflist-termlist termlist)))

  (define (add-zero-to-coefflist num coefflist)
    ;type: num,coefflist -> coefflist
    (cond ((= num 0) coefflist)
          ((> num 0) (cons (create-number 0)
                           (add-zero-to-coefflist (- num 1) coefflist)))
          (else (error "error from add-zero-to-coefflist"))))

  (define (adjoin-term term termlist)
  ;;type: term, dense<order,coefflist> --> dense<order,coefflist>

    (let ((o-term (order term))
          (c-term (coeff term))
          (o-termlist (order-termlist termlist))
          (cl-termlist (coefflist-termlist termlist)))
      (if (not (> o-term o-termlist))
          (error "error from adjoin-term")
          (make-termlist o-term (cons c-term
                                      (add-zero-to-coefflist (- o-term o-termlist 1)
                                                             cl-termlist))))))

  (define (first-term termlist)
    ;;type: dense<order,coefflist> --> term
    (make-term (order-termlist termlist) (car (coefflist-termlist termlist))))

  (define (rest-terms termlist)
    ;;type: dense<order,coefflist> --> dense<order,coefflist>
    (make-termlist (- (order-termlist termlist) 1)
                   (cdr (coefflist-termlist termlist))))

;;;;;;;;;;;tagged

  (define (tag x) (attach-tag 'dense x))
```

```
  (put 'make 'dense (lambda (x) (tag (create-dense x))))
  (put 'the-empty-termlist 'dense (tag (the-empty-termlist)))
  (put 'empty-termlist? '(dense) empty-termlist?)
  (put 'adjoin-term '(term dense) (lambda (x y) (tag (adjoin-term x y))))
  (put 'first-term '(dense) first-term)
  (put 'rest-terms '(dense) (lambda (x) (tag (rest-terms x))))

)


;;;;=============sparse-termlist--package========================
(define (install-sparse-package)

  (define (adjoin term term-list)
    (if (=zero? (coeff term))
          term-list
          (cons (make-term (order term) (coeff term)) term-list)))

  (define (empty) '())
  (define (first term-list) (car term-list))
  (define (rest term-list) (cdr term-list))
  (define (empty? term-list) (null? term-list))

;;;;;;;;;;tagged
  (define (tag x) (attach-tag 'sparse x))
  (put 'make 'sparse (lambda (x) (tag x)))
  (put 'the-empty-termlist 'sparse (tag (empty)))
  (put 'empty-termlist? '(sparse) empty?)
  (put 'adjoin-term '(term sparse) (lambda (x y) (tag (adjoin x y))))
  (put 'first-term '(sparse) first)
  (put 'rest-terms '(sparse) (lambda (x) (tag (rest x))))
)


;;;;========================term---package===========================

;;question1:2 level tagged term   vs    1 localized term + 1 tagged term ?
;;answer: 2 level tagged term.
;;quistion2:Do this situation relate to tower structure ?


;;install a sub-term package, make a 2 level system of term.
;; Change answer of question 1 and its related code.
```

```
(define (install-pair-term-package)
  (define (make-term order coeff)
    (cons order coeff))
  (define (order term)
    (car term))
  (define (coeff term)
    (cdr term))
  (define (tag x) (attach-tag 'pair-term x))

  (put 'make 'pair-term (lambda (x y) (tag (make-term x y))))
  (put 'order '(pair-term) order)
  (put 'coeff '(pair-term) coeff))

(define (install-term-package)
  (define (make-term order coeff)
    ((get 'make 'pair-term) order coeff))

  (define (tag x) (attach-tag 'term x))

  (put 'make 'term (lambda (x y) (tag (make-term x y))))
  (put 'order '(term) order)
  (put 'coeff '(term) coeff))


(install-pair-term-package)
(install-term-package)
(install-dense-package)
(install-sparse-package)

;;;;-----------------test---cases---from---generic.scm=====================
;;;;---rewrite--test--variable---for---the---test--cases---from----generic.scm----
===============
(define p1 (create-numerical-polynomial 'x '(1 5 0 -2)))
(define p2-mixed (create-dense-polynomial 'x (list p1 (create-number 3) (create-number
5))))
(define p2 (create-dense-polynomial 'z (list p1
                                             (create-dense-polynomial 'x (list (create-number 3)))
                                             (create-dense-polynomial 'x (list (create-number
5))))))


(define coeff1 (create-rational (create-numerical-polynomial 'y '(3))
                                (create-numerical-polynomial 'y '(1 0))))

(define coeff2 (create-rational (create-numerical-polynomial 'y '(1 0 1))
```

```scheme
                                    (create-numerical-polynomial 'y '(1 0))))

(define coeff3 (create-rational (create-numerical-polynomial 'y '(1))
                                    (create-numerical-polynomial 'y '(1 -1))))

(define coeff4 (create-rational (create-numerical-polynomial 'y '(2))
                                    (create-numerical-polynomial 'y '(1))))

(define p3 (create-dense-polynomial 'x (list coeff1 coeff2 coeff3 coeff4)))

;;---------rewrite---coercion----procedure--using---dense-termlist-
representation========
(define (repnum->reppoly var num)
  (make-poly var (create-dense-termlist (list (create-number num)))))

;;---------reinstall---coercion---procedures--for---mixed-typed--operations---
(put 'add '(polynomial number) (PPmethod->PNmethod +polynomial))
(put 'add '(number polynomial) (PPmethod->NPmethod +polynomial))
(put 'sub '(polynomial number) (PPmethod->PNmethod -polynomial))
(put 'sub '(number polynomial) (PPmethod->NPmethod -polynomial))
(put 'mul '(polynomial number) (PPmethod->PNmethod *polynomial))
(put 'mul '(number polynomial) (PPmethod->NPmethod *polynomial))
(put 'equ? '(polynomial number) (PPmethod->PNmethod equ-polynomial?))
(put 'equ? '(number polynomial) (PPmethod->NPmethod equ-polynomial?))

;;;test-cases-----from---generic.scm
;;1.(square p1)  ;;square is defined in terms of generic operation mul.
;;2.(square p2)
;;3.(square p2-mixed)
;;4.(=zero? p2-mixed)
;;5.(equ? (sub (add p1 p3) p1) p3)
;;6.(apply-polynomial p1 (create-number 2))
;;7.(apply-polynomial p2 (create-numerical-polynomial 'x '(1 1)))
;;8.(define x (create-numerical-polynomial 'x '(1 0)))
;;;(equ? (apply-polynomial p1 x) p1)


;;;;———Some——Past——notes—————
;;;;--------------some--interesting-thought---during-development----
;;1.should I install add-term to generic add to form a nice loop ?
;;answer:maybe not, but i think it is doable.

;;2.is this situation similar to the assignment or 2.86 or problemSet?
;;answer:sinsce I don't install add-term to add, so it is probably not.
```

;;3.Is this the concept of generic operation of this chapter 2.5?
;;answer:probably not. since I didnt install add-terms to add.
;;come back later for this questions.

;;4.Is this recursive data-dircted sturcture mentioned in the text?
;; or what is the difference between multiple representations and generic operations?
;;anwer:

;;reread text 2.5 and finish question 4!!! 01/04/19:20
;;then start exercise 2.91

<<<——BACK—TO—FIRST

<<<—BACK—TO—TRANSCRIPT—2.5

```
;;created 02/04/2023 17:30

(load "c:\\6.001\\work\\assignment\\genericarithmetic\\generic.scm")
;(load "c:\\6.001\\work\\exercise\\exercise2.5\\exercise2.90.scm")



(define (div–terms L1 L2)
  (if (empty–termlist? L1)
      (list (the–empty–termlist) (the–empty–termlist))
      (let ((t1 (first–term L1))
            (t2 (first–term L2)))
        (if (> (order t2) (order t1))
            (list (the–empty–termlist) L1)
            (let ((new–c (div (coeff t1) (coeff t2)))
                  (new–o (– (order t1) (order t2))))
              (let ((rest–of–result
                         (div–terms (+terms L1 (negate–terms (*–term–by–all–terms (make–term
new–o new–c)
                                                                            L2)))
                                        L2)
                     ))
                    (cons (adjoin–term (make–term new–o new–c) (car rest–of–result))
                          (cdr rest–of–result))
                ))))))

;;finish div–poly and test.
;;more test and maybe quotient abstraction needed!

(define (div–poly p1 p2)
  (if (same–variable? (variable p1) (variable p2))
      (let ((result (div–terms (term–list p1)
                               (term–list p2))))
          (let ((quotient–list (car result))
                (remainder–list (cadr result)))
            (list (make–poly (variable p1) quotient–list)
                  (make–poly (variable p2) remainder–list))))
      (error "different variable from div–poly")))

(put 'div '(polynomial polynomial) (lambda (x y) (let ((result (div–poly x y)))
                                                (list (attach–tag 'polynomial (car result))
                                                      (attach–tag 'polynomial (cadr result))))))
```

<<<—BACK—TO—TRANSCRIPT—2.5

```
;;created 06/04/2023 17:07

(load "c:\\6.001\\work\\exercise\\exercise2.5\\exercise2.91.scm")

(define (remainder-terms a b)
  (let ((result (div-terms a b)))
    (cadr result)))

(define (gcd-terms a b)
  (if (empty-termlist? b)
      a
      (gcd-terms b (remainder-terms a b))))

(define (gcd-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1) (gcd-terms (term-list p1)
                                          (term-list p2)))
      (error "different variable from gcd-poly")))

(define (greatest-common-divisor x y)
  (apply-generic 'greatest-common-divisor x y))

(put 'greatest-common-divisor '(polynomial polynomial) (lambda (x y) (attach-tag 'polynomial
                                                                                 (gcd-poly x y))))

(put 'greatest-common-divisor '(number number) (lambda (x y) (attach-tag 'number
                                                                         (gcd x y))))


(define p1 (create-numerical-polynomial 'x '(1 -1 -2 2 0)))
(define p2 (create-numerical-polynomial 'x '(1 0 -1 0)))
(greatest-common-divisor p1 p2)

;;answer is correct!
```

<<<—BACK—TO—TRANSCRIPT—2.5

```
;;created 10/04/2023 16:07

(load "c:\\6.001\\work\\exercise\\exercise2.5\\exercise2.95.scm")

;;Exercise2.96 A
(define (pseudodivision a b)
  (let ((o1 (order (first-term a)))
        (o2 (order (first-term b)))
        (c (coeff (first-term b))))
    (let ((integerizing-factor (power c (+ 1 (- o1 o2)))))
      (div-terms (map-terms (lambda (term) (make-term (order term)
                                                      (mul integerizing-factor
                                                           (coeff term))))
                            a)
                 b))))



(define (pseudoremainder-terms a b)

  (let ((result (pseudodivision a b)))
        (cadr result)))

;;Exercise 2.96B
(define (gcd-terms a b)
  ;;constraints:
  ;;1.pseudoremainder can help produce non-floating coefficient,but
  ;; it is not reduced to lowest term.
  (if (empty-termlist? b)
      (removes-common-factor a)  ;;constraint1
      (gcd-terms b (pseudoremainder-terms a b))))


(define (removes-common-factor termlist)
  ;;note:To use gcd to reduce the coefficients of a termlist to lowest term.
  ;;Type: termlist<tagged-number> ---> termlist<tagged-number>
  ;;constraints:
  ;;1.gcd take scheme-number as input, but the coefficient in termlist is
  ;; tagged number.

  (define (term-list->number-list terms)   ;constraint 1
    (cond ((empty-termlist? terms) '())
          (else
           (cons (first-term terms)
                 (term-list->number-list (rest-terms terms))))))
```

```
  (let ((common-factor (apply gcd (map (lambda (term) (contents (coeff term)))
                                          (term-list->number-list termlist))))) ;;constraint 1
    (let ((tagged-factor (create-number common-factor)))  ;constraint 1
      (map-terms (lambda (term) (make-term (order term) (div (coeff term) tagged-factor)))
                  termlist))))


(define (gcd-poly p1 p2)
  (if (same-variable? (variable p1) (variable p2))
      (make-poly (variable p1) (gcd-terms (term-list p1)
                                          (term-list p2)))
      (error "different variable from gcd-poly")))

(define (greatest-common-divisor x y)
  (apply-generic 'greatest-common-divisor x y))

(put 'greatest-common-divisor '(polynomial polynomial) (lambda (x y) (attach-tag 'polynomial
                                                                      (gcd-poly x y))))

(put 'greatest-common-divisor '(number number) (lambda (x y) (attach-tag 'number
                                                               (gcd x y))))


;;;;test-cases from exercise2.95
(define p1 (create-numerical-polynomial 'x '(1 -2 1)))
(define p2 (create-numerical-polynomial 'x '(11 0 7)))
(define p3 (create-numerical-polynomial 'x '(13 5)))

(define q1 (mul p1 p2))
(define q2 (mul p1 p3))
(greatest-common-divisor q1 q2)
(equ? (greatest-common-divisor q1 q2) p1)
```

<<—BACK—TO—FIRST

<<<——BACK-TO-MY-JOURNEY-OF-LISP-PROGRAMMING

After delving into programming in streams, my initial instinct is to modify the suggested nth-queens puzzle solution by Exercise 2.42, which is to generate all solutions at once.

```
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (cons-stream empty-board empty-stream)
        (stream-filter
         (lambda (positions) (safe? k positions))
         (stream-flatmap
          (lambda (rest-of-queens)
            (stream-map (lambda (new-row)
                          (adjoin-position new-row k rest-of-
queens))
                        (stream-enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (queen-cols board-size))
```

This modification above does improve performance. The stream version can solve the puzzle at a board size of 10 for which the original solution from Exercise 2.42 could not.

We can rewrite the above code using the implicit-style of defining streams as follows:

```
(define (queens3 board-size)
  (define queen-stream
    (cons-stream
     (cons-stream empty-board empty-stream)
     (stream-map (lambda (result new-column)
                   (stream-filter
                   (lambda (positions)
                         (safe? new-column positions board-size))
                   (stream-flatmap
                         (lambda (rest-of-queens)
                               (stream-map (lambda (new-row)
                                     (adjoin-position  new-row
                                                       new-column
                                                       rest-of-queens))
                            (stream-enumerate-interval 1 board-size)
```

```
                          ))
                result))

              )
            queen-stream
            (stream-enumerate-interval 1 board-size))
      ))
  queen-stream)
```

Exercise 2.42's solution is at “solution2.42”

<<—BACK—TO—FIRST

<<<——BACK—TO—IMPLICIT–STYLE–STREAM

```
;;––––––––––––position–––abstraction–––––––––––––––––––––
(define (make–position row column)
  (list 'position row column))

(define (get–row position)
  (if (position? position)
      (cadr position)
      (error "input is not a position")))

(define (get–column position)
  (if (position? position)
      (caddr position)
      (error "input is not a position")))

(define (position? position)
  (and (pair? position)
       (equal? 'position (car position))))

(define (position–equ? a b)
  (and (= (get–row a) (get–row b))
       (= (get–column a) (get–column b))))

(define (memq–position position seq)
  ;;Note: Position version of memq.
  (cond ((null? seq) false)
          ((position–equ? position (car seq)) true)
          (else (memq–position position (cdr seq)))))
;;––––––––––––position–––abstraction––––––––––––––––––––––

(define (adjoin–position row column set–of–positions)
  ;;Note: add new position
  ;;Type: num num list<position> –> list<position>
  (append set–of–positions (list (make–position row column))))

(define empty–board '())

(define (combine–boolean seq)
  ;;Type: list<boolean> –> boolean
  (if (null? seq) true
      (and (car seq) (combine–boolean (cdr seq)))))
```

```
(define (safe? k set-of-positions max)
  ;;Note: Safe? takes a current column k, sequence of positions and max row number.
  ;;Type: num,list<position>,num -> boolean
  ;;Constraints:
  ;;1. first column return true.
  ;;2. seperating current position from the earlier positions by dividing set-of-positions.
  ;;3. check no row matches.
  ;;4. check no diagonal matches.
  ;;5. Do not need to check column since current position is the only one.
  ;;6. downward diagonal need to know the maximun row number which is the boardsize
passing
  ;;   from outside.
  ;;7. make sure return the correct type.
;  (newline)
;  (display "predicate--safe?---start-working--------")
;  (newline)
  (if (= k 1) true     ;;constraint1
      (let ((k-position (car (filter (lambda (position) (= (get-column position) k))     ;;constraint2
                                     set-of-positions)))
            (seq-exclude-k (filter (lambda (position) (not (= (get-column position)
k)));;constraint2
                                   set-of-positions)))
         (let ((upward-diagonal (compute-diagonal-upward k-position))
               (downward-diagonal (compute-diagonal-downward k-position
max)))      ;constraint6


           (let ((result
                  (map (lambda (position)
                         (cond  ((= (get-row position) (get-row k-position))  ;;constraint3
                                 false)
                               ((memq-position position upward-diagonal)     ;;constraint4
                                 false)
                               ((memq-position position downward-diagonal)   ;;constraint4
                                 false)
                               (else true)))

                       seq-exclude-k)
                  ))


             (combine-boolean result)    ;;constraint7

             )
           ))))
```

```
(define (compute-diagonal-upward last-position)
  ;;Compute-diagonal-upward returns upward diagonal positions whose column number is
  ;;less than last-position.
  ;;Type: position-> list<position>
  (define (helper row column result)
    (if (< row 1) result
        (helper (- row 1) (- column 1) (append result (list (make-position row column))))))

  (let ((row (get-row last-position))
        (column (get-column last-position)))
    (helper (- row 1) (- column 1) nil)))


(define (compute-diagonal-downward last-position max-row)
  ;;Note:compute-diagonal-downward returns downward diagonal positions whose column
number
  ;;    is smaller than last-position and row number is smaller than or equal to max-row.
  ;;Type: position, number -> list<position>
  (define (helper row column result)
    (if (> row max-row) result
        (helper (+ row 1) (- column 1) (append result (list (make-position row column))))))
  (let ((row (get-row last-position))
        (column (get-column last-position)))
    (helper (+ row 1) (- column 1) nil)))
```

<<<——BACK—TO-FIRST

<<——BACK-TO-MY-JOURNEY-OF-LISP-PROGRAMMING

Exercise3.80 ask readers to model RLC circuit using streams without prior knowledge.

If I setup my solution as exercise 3.80 required:

(define rlc1 (rlc 1 1 0.2 0.1))   R = 1 ohm, C = 0.2 farad, L = 1 henry, dt = 0.1 second.

(define test (rlc1 10 0))   iL0 = 0 amps and vC0 = 10 volts.

then (display-stream test) will print out (voltage current) pairs.

```
(10 . 0)
(10 . 1.)
(10.5 . 1.9)
(11.45 . 2.76)
(12.829999999999998 . 3.6289999999999996)
(14.644499999999997 . 4.549099999999999)
(16.91905 . 5.558639999999999)
(19.698369999999997 . 6.694680999999999)
(23.0457105 . 7.995049899999999)
(27.043235449999997 . 9.500115959999999)
(31.79329343 . 11.254427908999999)
(37.4205073845 . 13.308314461099998)
(44.07466461505 . 15.719533753439999)
(51.93443149177 . 18.555046839600998)
(61.2119549115705 . 21.8929853048179)
(72.15844756397945 . 25.82488226549316)
.
.
.
.
```

For solution, see “solution3.80”.

<<<——BACK—TO—EXERCISE–3.80

```
(define (rlc r l c dt)
  (lambda (vc0 il0)
    (define vc (integral (delay dvc) vc0 dt))
    (define il (integral (delay dil) il0 dt))
    (define dvc (scale-stream il (/ 1 c)))
    (define dil (add-streams (scale-stream il (/ (- r) l))
                             (scale-stream vc (/ 1 l))))

    (stream-map (lambda (volt-c current-L)
                  (cons volt-c current-L))
                vc
                il)))



(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream initial-value
                 (let ((integrand (force delayed-integrand)))
                   (add-streams (scale-stream integrand dt)
                                int))))
  int)


(define (add-streams s1 s2)
  (stream-map + s1 s2))

(define (scale-stream stream factor)
  (stream-map (lambda (x) (* x factor)) stream))


(define (display-stream s)
  (stream-for-each display-line s))

(define (display-line x)
  (newline)
  (display x))
```