Some Cute Programs

Jayadev Misra

Department of Computer Science
University of Texas at Austin

http://orc.csres.utexas.edu

July 16 – 20, 2012
WG 2.3 Meeting, Seattle

Orc Basics

- Site: Basic service or component.
- Concurrency combinators for integrating sites.
- Theory includes nothing other than the combinators.

No notion of data type, thread, process, channel, synchronization, parallelism $\cdots$

New concepts are programmed using new sites.

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning
if f halts without publishing do g	$f\ ;\ g$	Otherwise

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning
if f halts without publishing do g	$f \; ; \; g$	Otherwise

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning
if f halts without publishing do g	$f \; ; \; g$	Otherwise

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f >x> g$	Sequential composition
for some x from g do f	$f <x< g$	Pruning
if f halts without publishing do g	$f\ ;\ g$	Otherwise

Structure of Orc Expression

- Simple: just a site call, $CNN(d)$
  Publishes the value returned by the site.

- Composition of two Orc expressions:

do f and g in parallel	$f \mid g$	Symmetric composition
for all x from f do g	$f > x > g$	Sequential composition
for some x from g do f	$f < x < g$	Pruning
if f halts without publishing do g	$f \; ; \; g$	Otherwise

Symmetric composition: $f \mid g$

- Evaluate f and g independently.
- Publish all values from both.
- No direct communication or interaction between f and g. They can communicate only through sites.

Example: $CNN(d) \mid BBC(d)$

Calls both CNN and BBC simultaneously.
Publishes values returned by both sites. (0, 1 or 2 values)

Sequential composition: $f \ {>}x{>} \ g$

For all values published by f do g.
Publish only the values from g.

- $CNN(d) \ {>}x{>} \ Email(address, x)$
  - Call $CNN(d)$.
  - Bind result (if any) to x.
  - Call $Email(address, x)$.
  - Publish the value, if any, returned by $Email$.

- $(CNN(d) \mid BBC(d)) \ {>}x{>} \ Email(address, x)$
  - May call $Email$ twice.
  - Publishes up to two values from $Email$.

Notation: $f \gg g$ for $f \ {>}x{>} \ g$, if x is unused in g.

Schematic of Sequential composition

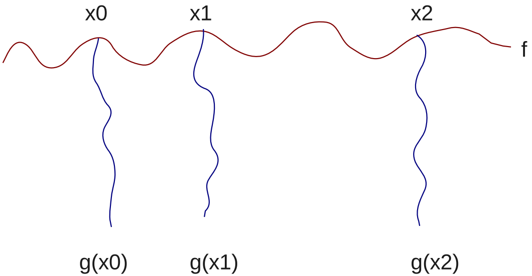


Figure: Schematic of $f \; >x> \; g$

Pruning: $f \;\; {<}x{<} \; g$

For some value published by g do f.

- Evaluate f and g in parallel.
  - Site calls that need x are suspended.
    Consider $(M() \mid N(x)) \;\; {<}x{<} \; g$
- When g returns a (first) value:
  - Bind the value to x.
  - Kill g.
  - Resume suspended calls.
- Values published by f are the values of $(f \;\; {<}x{<} \; g)$.

Notation: $f \ll g$ for $f \;\; {<}x{<} \; g$, if x is unused in g.

Example of Pruning

$$Email(address, x) \;\; <x< \; (CNN(d) \mid BBC(d))$$

Binds x to the first value from $CNN(d) \mid BBC(d)$.
Sends at most one email.

Otherwise: f ; g

Do f. If f halts without publishing then do g.

- An expression halts if
  - its execution can take no more steps, and
  - all called sites have either responded, or will never respond.
- A site call may respond with a value, indicate that it will never respond (helpful), or do neither.
- All library sites in Orc are helpful.

Orc program

- Orc program has
  - a goal expression,
  - a set of definitions.
- The goal expression is executed. Its execution
  - calls sites,
  - publishes values.

Some Fundamental Sites

- $Ift(b)$, $Iff(b)$: boolean b,
  Returns a signal if b is true/false; remains silent otherwise.
  Site is helpful: indicates when it will never respond.

- $Rwait(t)$: integer t, $t \geq 0$, returns a signal t time units later.

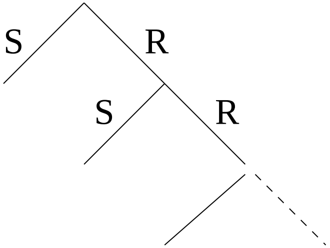
- $stop$: never responds. Same as $Ift(false)$ or $Iff(true)$.

- $signal$: returns a signal immediately.
  Same as $Ift(true)$ or $Iff(false)$.

Example of a Definition: Metronome

Publish a signal every unit.

$$\textit{def}\ \ Metronome() = \underbrace{signal}_{S} \mid (\underbrace{Rwait(1) \gg Metronome()}_{R})$$



Unending string of Random digits

$Metronome() \gg Random(10)$ – one every second

$def \; rand\_seq() =$ – at a specified rate
$\quad Random(10) \mid Rwait(dd) \gg rand\_seq()$

Logical Connectives; 2-valued Logic

And: Publish a signal if both sites do.
Or: Publish a signal if either site does.

$M() \gg N()$ – "and"

$b \ {<}b{<}\ (M() \mid N())$ – "or"

$M() \ ; \ N()$ – "or" with helpful M

$(M() \gg \mathit{true} \ ; \ \mathit{false}) \ {>}b{>}\ \mathit{Iff}(b)$ – "not" with helpful M

Orc Language

- Data Types: Number, Boolean, String, with Java operators
- Conditional Expression: *if* b *then* f *else* g
- Data structures: Tuple, List, Record
- Pattern Matching; Clausal Definition
- Function Closure
- Comingling functional and Orc expressions
- Class for active objects

Implicit Concurrency

- An experiment tosses two dice.
  Experiment is a success if and only if sum of the two dice thrown is 7.
- $exp(n)$ runs n experiments and reports the number of successes.

$$def \;\; toss() = Random(6) + 1$$

-- *toss* returns a random number between 1 and 6

$$def \;\; exp(0) = 0$$
$$def \;\; exp(n) = exp(n-1) + (if \; toss() + toss() = 7 \; then \; 1 \; else \; 0)$$

Translation of the dice throw program

```
def toss() = add(x, 1) <x< Random(6)
def exp(n) =
   ( Ift(b) >> 0
   | Iff(b) >>
     ( add(x, y)
         <x< ( exp(m) <m< sub(n, 1) )
         <y< ( Ift(bb) >> 1 | Iff(bb) >> 0 )
           <bb< equals(p, 7)
             <p< add(q, r)
               <q< toss()
               <r< toss()
     )

   ) <b< equals(n, 0)
```

Note: $2n$ parallel calls to *toss*().

Deflation

- Expression $C(\ldots, e, ..)$,
- single value expected at e
- translate to $C(\ldots, x, ..)$ $<x<$ e where x is fresh
- applicable hierarchically.

$(1|2) * (10|100)$ is
$(Times(x,y)$ $<x<$ $(1 \mid 2))$ $<y<$ $(10 \mid 100)$, or
$Times(x,y)$ $<x<$ $(1 \mid 2)$ $<y<$ $(10 \mid 100)$

Pattern Matching, clausal definition

$$type\ Tree = Node(Tree, Tree) \mid Leaf() \mid NonTree()$$

$$\begin{aligned}
&def\ \ tc(_, [\,]) = \ NonTree()\\
&def\ \ tc([\,], [(v, t)]) = \ if\ \ (v = 0)\ then\ \ t\ else\ \ NonTree()\\
&def\ \ tc([\,], v : right) = \ tc([v], right)
\end{aligned}$$

$$\begin{aligned}
&def\ \ tc((u, t) : left, (v, t') : right) =\\
&\qquad if\ u = v\ then\ \ tc(left, (v-1, Node(t, t')) : right)\\
&\qquad else\ \ tc((v, t') : (u, t) : left, right)
\end{aligned}$$

val, tuple, closure

$def \;\; circle =$

$\quad val \;\; pi = \;\; 3.1416$

$\quad def \;\; perim(r) = \;\; 2 * pi * r$

$\quad def \;\; area(r) = \;\; pi * r **2$ #

$(perim, area)$

Examples

- Combinatorial
- Mutable store manipulation
- Synchronization, Communication

List map

$$\mathit{def}\ \ \mathit{parmap}(\_, [\,]) = [\,]$$

$$\mathit{def}\ \ \mathit{parmap}(f, x : xs) = f(x) : \mathit{parmap}(f, xs)$$

List map (Contd.)

$$def \;\; seqmap(\_, [\,]) = [\,]$$

$$def \;\; seqmap(f, x : xs) = f(x) \;\; >y> (y : seqmap(f, xs))$$

Infinite Set Enumeration

Enumerate all finite binary strings.
A binary string is a list of 0,1.

$$\textit{def} \;\; \textit{bin}() =$$

$$
\begin{array}{ll}
 & [\,] \\
| & \textit{bin}() \;\; >xs> \; (0 : xs \mid 1 : xs)
\end{array}
$$

Subset Sum

Given integer n and list of integers xs.

$parsum(n, xs)$ publishes all sublists of xs that sum to n.

$$def \;\; parsum(0, [\,]) = [\,]$$

$$def \;\; parsum(n, [\,]) = stop$$

$$\begin{aligned} &def \;\; parsum(n, x : xs) = \\ &\qquad parsum(n - x, xs) \;\; >ys> x : ys \;\; | \;\; parsum(n, xs) \end{aligned}$$

Subset Sum (Contd.), Backtracking

Given integer n and list of integers xs.

$seqsum(n, xs)$ publishes the first sublist of xs that sums to n.

"First" is smallest by index lexicographically.

$$def \;\; seqsum(0, [\,]) = [\,]$$

$$def \;\; seqsum(n, [\,]) = stop$$

$$\begin{aligned} &def \;\; seqsum(n, x : xs) = \\ &\quad x : seqsum(n - x, xs) \;\; ; \;\; seqsum(n, xs) \end{aligned}$$

Subset Sum (Contd.), Concurrent Backtracking

Publish the first sublist of xs that sums to n.

Run the searches concurrently.

$$def \;\; parseqsum(0, [\,]) = [\,]$$

$$def \;\; parseqsum(n, [\,]) = stop$$

$$\begin{aligned} &def \;\; parseqsum(n, x : xs) = \\ &\quad (p \;;\; q) \\ &\qquad <p< x : parseqsum(n - x, xs) \\ &\qquad <q< parseqsum(n, xs) \end{aligned}$$

Note: Neither search in the last clause may succeed.

Fold on a non-empty list

fold with binary f: $fold(+, [x_0, x_1, \cdots]) = x_0 + x_1 \cdots$

$def \;\; fold(, [x]) = x$

$def \;\; fold(f, x : xs) = f(x, fold(xs))$

Associative fold on a non-empty list

$$
\begin{aligned}
&def\ \ afold(f,[x]) = x\\
&def\ \ afold(f,xs) =\\
&\qquad def\ \ pairfold([\,]) = [\,]\\
&\qquad def\ \ pairfold([x]) = [x]\\
&\qquad def\ \ pairfold(x : y : xs) = f(x,y) : pairfold(xs)\\
&afold(f, pairfold(xs))
\end{aligned}
$$

map and associative fold: *map_afold*

Associative commutative fold over a channel

A channel has two methods: put and get.

$chFold(c, n)$ folds the first n items of channel c and publishes.

$$def\ \ chFold(c, 1) = c.get()$$

$$def\ \ chFold(c, n) = f(chFold(c, n/2), chFold(c, n - n/2))$$

Associative commutative fold over a channel

$$
\begin{aligned}
&\mathit{def}\ \ \mathit{cfold}(c, n) = \\
&\qquad \mathit{def}\ \ \mathit{threads}(0) = \mathit{stop} \\
&\qquad \mathit{def}\ \ \mathit{threads}(k) = \\
&\qquad\qquad \mathit{threads}(k-1) \\
&\qquad\quad\ \mid c.\mathit{put}(f(c.\mathit{get}(), c.\mathit{get}())) \gg \mathit{stop} \\
&\mathit{threads}(n-1)\ \ ;\ \ c.\mathit{get}()
\end{aligned}
$$

- if n is strictly more than k, $\mathit{threads}(k)$ terminates.
- at its termination the channel contains $n-k$ items whose fold yields the desired result.

Mutable Store Manipulation

`Ref(n)`	Mutable reference with initial value `n`
`Cell()`	Write-once reference
`Array(n)`	Array of size `n` of Refs
`Table(n,f)`	Array of size `n` of immutable values of `f`
`Channel()`	Unbounded (asynchronous) channel

$Ref(3)\ >r>\ r.write(5) \gg r.read()$, or $Ref(3)\ >r>\ r := 5 \gg r?$

$Cell()\ >r>\ (r.write(5) \mid r.read())$, or $Cell()\ >r>\ r := 5 \mid r?$

$Array(3)\ >a>\ a(0) := true \gg a(1)?$

$Channel()\ >ch>\ (ch.get() \mid ch.put(3) \gg stop\ )$

Quicksort

```
def swap(i,j) = (i?,j?) >(x,y)> (i := y, j := x) ≫ signal
def quicksort(a) =
    def segmentsort(u,v) =
        def part(p,s,t) =
            def lr(i) = Ift(i < t) ≫ Ift(a(i)? ≤ p) ≫ lr(i + 1) ; i
            def rl(i) = Ift(a(i)? :> p) >> rl(i − 1) ; i                  #

            (lr(s + 1), rl(t − 1)) >(s′,t′)>
            (if (s′ < t′) then swap(a(s′), a(t′)) ≫ part(p, s′, t′)
             else t′)                                                     #

        if v − u > 1 then
            part(a(u)?, u, v) >m>
            swap(a(u), a(m)) ≫
            (segmentsort(u, m), segmentsort(m + 1, v)) ≫ signal
        else signal
    segmentsort(0, a.length?)
```

Sequential Breadth-First Traversal of a Graph

N nodes in a graph,

$root$ a specified node,

$succ(x)$ is the list of successors of x,

Publish the $parent$ of each node in Breadth-First Traversal.

$$def\ \ bfs(N, root, succ) =$$
$$val\ \ parent = Table(N, lambda(\_) = Cell())$$

– bfs' is bfs on a list of nodes

$$def\ \ bfs'([\,]) = signal$$
$$def\ \ bfs'(x : xs) = bfs'(append(xs, expand(x)))$$

$$parent(root) := N \gg bfs'([root]) \gg parent$$

Site *expand*

$$def\ \ expand(x) =$$

– $expand'(x, ys)$, ys successors of x yet to be scanned

$$\begin{array}{l} def\ \ expand'(x, [\,]) = [\,] \\ def\ \ expand'(x, z : zs) = \\ \quad parent(z) := x \ \gg\ z : expand'(x, zs)\ ;\ expand'(x, zs) \end{array}$$

$$expand'(x, succ(x))$$

Sequential Breadth-First Traversal: Complete Program

$$
\begin{array}{l}
\textit{def}\ \ \textit{bfs}(N, \textit{root}, \textit{succ}) = \\
\quad \textit{val}\ \ \textit{parent} = \ \textit{Table}(N, \textit{lambda}(\_) = \textit{Cell}()) \\
\\
\quad \textit{def}\ \ \textit{expand}(x) = \\
\quad\quad \textit{def}\ \ \textit{expand}'(x, [\,]) = \ [\,] \\
\quad\quad \textit{def}\ \ \textit{expand}'(x, z : zs) = \\
\quad\quad\quad \textit{parent}(z) := x \gg z : \textit{expand}'(x, zs) \ ; \textit{expand}'(x, zs) \\
\quad \textit{expand}'(x, \textit{succ}(x)) \qquad\qquad \text{– Goal of expand} \\
\\
\quad \textit{def}\ \ \textit{bfs}'([\,]) = \ \textit{signal} \\
\quad \textit{def}\ \ \textit{bfs}'(x : xs) = \ \textit{bfs}'(\textit{append}(xs, \textit{expand}(x))) \\
\\
\textit{parent}(\textit{root}) := N \gg \textit{bfs}'([\textit{root}]) \gg \textit{parent}
\end{array}
$$

Concurrent Breadth-First Traversal

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{bfs}(N, \mathit{root}, \mathit{succ}) = \\
\quad \mathit{val}\ \ \mathit{parent} = \ \mathit{Table}(N, \mathit{lambda}(\_) = \mathit{Cell}()) \\
\\
\quad \mathit{def}\ \ \mathit{expand}(x) = \\
\quad\quad \mathit{if}\ \mathit{succ}(x) = [\,]\ \mathit{then}\ [\,] \\
\quad\quad \mathit{else}\ \ \mathit{map\_afold} \\
\quad\quad\quad (\\
\quad\quad\quad\quad \mathit{lambda}(y) = \mathit{parent}(y) := x \gg [y]\ ;\ [\,], \\
\quad\quad\quad\quad \mathit{append}, \\
\quad\quad\quad\quad \mathit{succ}(x) \\
\quad\quad\quad) \\
\\
\quad \mathit{def}\ \ \mathit{bfs}'([\,]) = \ \mathit{signal} \\
\quad \mathit{def}\ \ \mathit{bfs}'(\mathit{xs}) = \ \mathit{bfs}'(\mathit{map\_afold}(\mathit{expand}, \mathit{append}, \mathit{xs})) \\
\\
\mathit{parent}(\mathit{root}) := N \gg \mathit{bfs}'([\mathit{root}]) \gg \mathit{parent}
\end{array}
$$

Memoization

Memoize calls to $f()$.

$$val\ \ done = Cell()$$
$$val\ \ res = Cell()$$

$$def\ \ memof() =$$
$$\quad res? \ll (done := signal \gg res := f())$$

Memoization of Fibonacci

$$
\begin{aligned}
&\textit{val}\;\; N = 100\\
&\textit{val}\;\; \textit{done} = \textit{Table}(N+1, \textit{lambda}(\_) = \textit{Cell}())\\
&\textit{val}\;\; \textit{res} = \textit{Table}(N+1, \textit{lambda}(\_) = \textit{Cell}())
\end{aligned}
$$

$$
\begin{aligned}
&\textit{def}\;\; \textit{mfib}(0) = 0\\
&\textit{def}\;\; \textit{mfib}(1) = 1\\
&\textit{def}\;\; \textit{mfib}(i) =\\
&\qquad \textit{res}(i)? \ll\\
&\qquad (\textit{done}(i) := \textit{signal} \gg \textit{res}(i) := \textit{mfib}(i-1) + \textit{mfib}(i-2))
\end{aligned}
$$

Synchronization, Communication

`Semaphore(n)`	Semaphore with initial value `n`
`BoundedChannel(n)`	bounded (asynchronous) channel of size `n`
`Counter()`	Counter with `inc()`, `dec()` and `onZero()`

$$Semaphore(1) \;\; >s> \; s.acquire() \;\gg\; r := 5 \;\gg\; s.release()$$

$$BoundedChannel(1) \;\; >ch> \; (ch.put(5) \mid ch.put(3))$$

$$Counter() \;\; >ctr> \; (ctr.inc() \;\gg ctr.onZero() \mid Rwait(10) \;\gg ctr.dec())$$

Rendezvous

$$
\begin{array}{l}
\textit{def}\ \ \textit{class zeroChannel}() = \\
\quad \textit{val}\ \ s = \textit{Semaphore}(0) \\
\quad \textit{val}\ \ w = \textit{BoundedChannel}(1) \\
\quad \textit{def}\ \ \textit{put}(x) = s.\textit{acquire}() \gg w.\textit{put}(x) \\
\quad \textit{def}\ \ \textit{get}() = s.\textit{release}() \gg w.\textit{get}() \\
\textit{stop}
\end{array}
$$

Pure Rendezvous

$$
\begin{array}{l}
\mathit{def}\ \ \mathit{class}\ \mathit{pairSync}() = \\
\quad \mathit{val}\ \ s = \ \mathit{Semaphore}(0) \\
\quad \mathit{val}\ \ t = \ \mathit{Semaphore}(0) \\
\quad \mathit{def}\ \ \mathit{put}() = \ s.\mathit{acquire}() \gg t.\mathit{release}() \\
\quad \mathit{def}\ \ \mathit{get}() = \ s.\mathit{release}() \gg t.\mathit{acquire}() \\
\mathit{stop}
\end{array}
$$

Reader-Writer; Call API

$val\ \ req = \ Channel()$
$val\ \ na = \ Counter()$

$def\ \ startread() =$
$\quad val\ \ s = \ Semaphore(0)$
$\quad req.put((true, s)) \gg s.acquire()$

$def\ \ startwrite() =$
$\quad val\ \ s = \ Semaphore(0)$
$\quad req.put((false, s)) \gg s.acquire()$

$def\ \ endread() = \ na.dec()$

$def\ \ endwrite() = \ na.dec()$

Reader-Writer; Main Loop

$$def\ \ manager() = grant(req.get()) \gg manager()$$

$$def\ \ grant((true, s)) = na.inc() \gg s.release()$$ – Reader

$$def\ \ grant((false, s)) =$$ – Writer

$$na.onZero() \gg na.inc() \gg s.release() \gg na.onZero()$$

Reader-Writer; Using 2 semaphores

$$
\begin{array}{l}
\textit{def}\ \ \textit{class readerWriter2}() = \\
\quad \textit{val}\ \ \textit{req} = \ \textit{Channel}() \\
\quad \textit{val}\ \ \textit{na} = \ \textit{Counter}() \\
\quad \textit{val}\ \ (r, w) = \ (\textit{Semaphore}(0), \textit{Semaphore}(0)) \\
\\
\quad \textit{def}\ \ \textit{startread}() = \ \textit{req.put}(\textit{true}) \gg r.\textit{acquire}() \\
\quad \textit{def}\ \ \textit{startwrite}() = \ \textit{req.put}(\textit{false}) \gg w.\textit{acquire}() \\
\\
\quad \textit{def}\ \ \textit{endread}() = \ \textit{na.dec}() \\
\quad \textit{def}\ \ \textit{endwrite}() = \ \textit{na.dec}() \\
\\
\quad \textit{def}\ \ \textit{grant}(\textit{true}) = \ \textit{na.inc}() >> r.\textit{release}() \quad \text{-- Reader} \\
\\
\quad \textit{def}\ \ \textit{grant}(\textit{false}) = \quad \text{-- Writer} \\
\qquad \textit{na.onZero}() \gg \textit{na.inc}() \gg w.\textit{release}() \gg \textit{na.onZero}() \\
\\
\quad \textit{def}\ \ \textit{manager}() = \ \textit{grant}(\textit{req.get}()) \gg \textit{manager}() \\
\\
\textit{manager}()
\end{array}
$$

Reader-Writer; dispense with the queue

Keep count of the number of waiting readers and writers.

Use coin toss to choose a reader or writer, instead of looking up in the queue.

Packet Reassembly Using Sequence Numbers

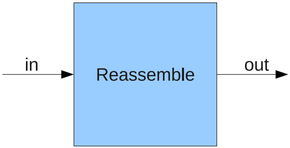


Figure: Packet Reassembler

- Packet with sequence number i is at position p_i in the input channel.
- Given: $|i - p_i| \leq k$, for some positive integer k.
- Then $p_i \leq i + k \leq p_{i+2\times k}$. Let $d = 2 \times k$.

Packet Reassembly Program

$def\ \ reassembly(read, write, d) =$ – d must be positive
$\quad val\ \ ch = Table(d, lambda(\_) = Channel())$

$\quad def\ \ input() = read()\ \ >(n, v)>\ ch(n\%d).put(v)\ \gg input()$

$\quad def\ \ output(i) = ch(i).get()\ \ >v>\ write(v)\ \gg output((i+1)\%d)$

$input()\ |\ output(0)$ – Goal expression

{- With Multiple Readers -} $read()\ |\ read()\ |\ write(0)$

Response Game

```
val  sw = Stopwatch()
val  (id, dd) = (3000, 100) – initial delay, digit delay
def  rand_seq() =   – Publish a random sequence of digits
   Random(10) | Rwait(dd) ≫ rand_seq()
def  game() =
   val  v = Random(10) – v is the seed for one game
val  (b, w) =
   Rwait(id) ≫ sw.reset() ≫ rand_seq() >x> Println(x) ≫
   Ift(x = v) ≫ sw.start() ≫ stop
 | Prompt( "Press ENTER for SEED "+v ) ≫
   sw.isrunning() >b> sw.halt() >w> (b, w)

if  b then – Goal expression of game()
   ( "Your response time = " + w + " milliseconds." )
else  ( "You jumped the gun." )
game()
```