# Programming Language Paradigm CSC335

Spring 2018

## Stock Analysis with Stream Processing

By: Bohan Chen
Alan Lau
Rupesh Basnet

## What is Stream ?

In computer science, a stream is a potentially infinite sequence of data that will be processing when required. Stream are implemented using promises, or delay evaluation.

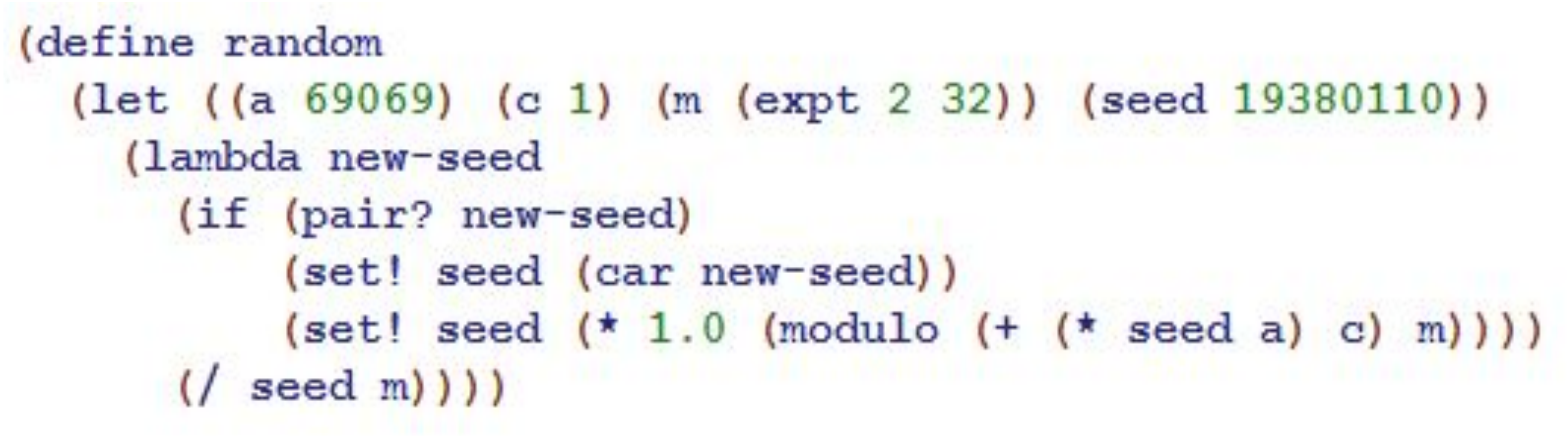
## Project Inspiration

Our idea was inspired by the research paper  "Lazy Evaluation Methods for Detecting Complex Events" by Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster.  This research paper is about using complex event processing system to detect complex patterns over a stream of primitive events. This system also employ Non-deterministic Finite Automata to detect sequences.

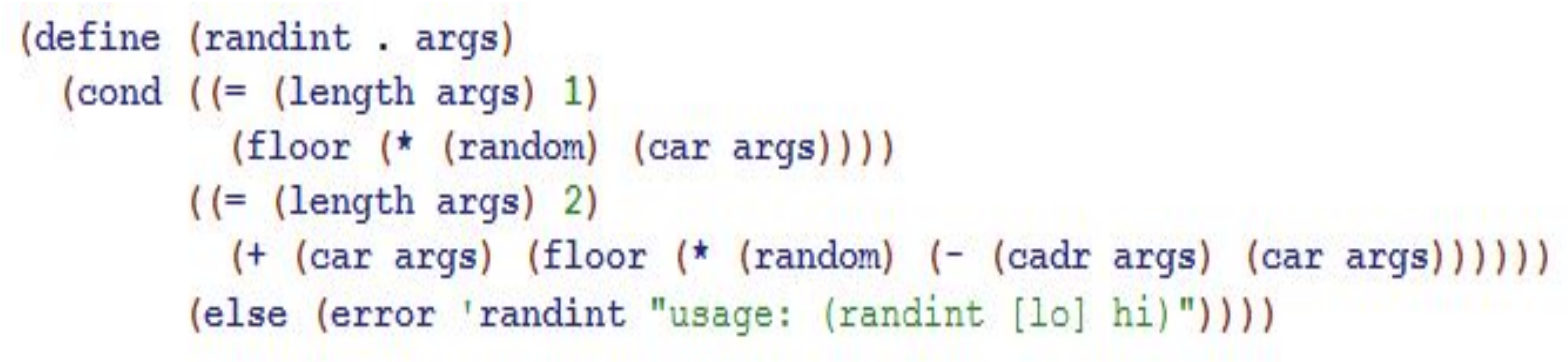
## Our idea

We plan to build a system that simulate the actual system that will read the price of the stock and analyze it. Instead of reading whole bunch of data at once, we will use the method of lazy evaluation by reading the data only if it needed. Since we can’t read the actual stock’s data so we decided to use random function to generate our own data.

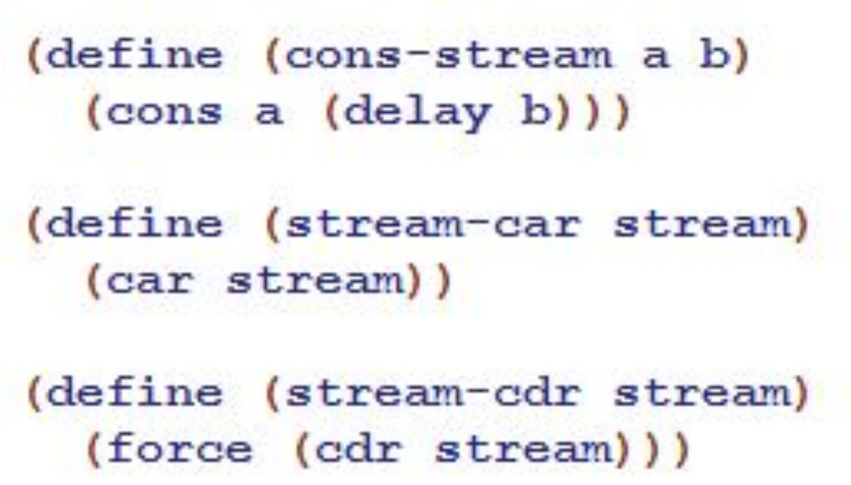
## Program Function



```
(define random
  (let ((a 69069) (c 1) (m (expt 2 32)) (seed 19380110))
    (lambda new-seed
      (if (pair? new-seed)
          (set! seed (car new-seed))
          (set! seed (* 1.0 (modulo (+ (* seed a) c) m))))
      (/ seed m))))
```



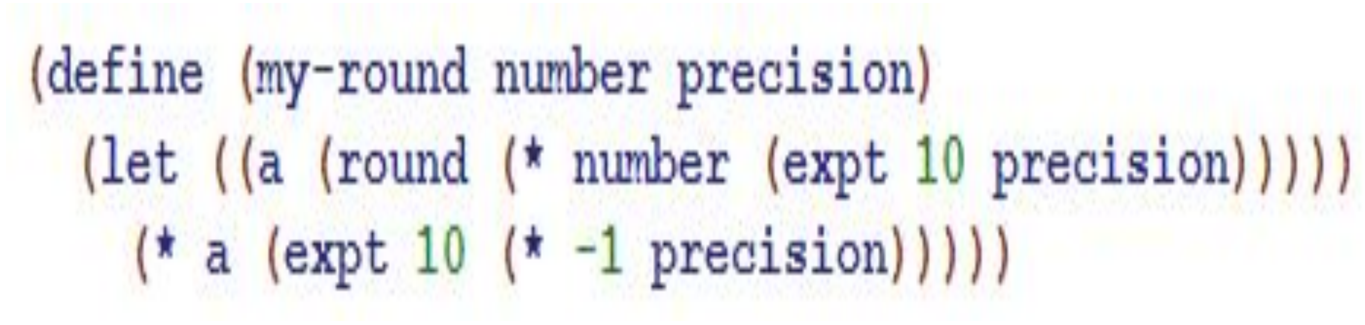
```scheme
(define (randint . args)
  (cond ((= (length args) 1)
          (floor (* (random) (car args))))
        ((= (length args) 2)
          (+ (car args) (floor (* (random) (- (cadr args) (car args))))))
        (else (error 'randint "usage: (randint [lo] hi)"))))
```



```
(define (cons-stream a b)
  (cons a (delay b)))

(define (stream-car stream)
  (car stream))

(define (stream-cdr stream)
  (force (cdr stream)))
```

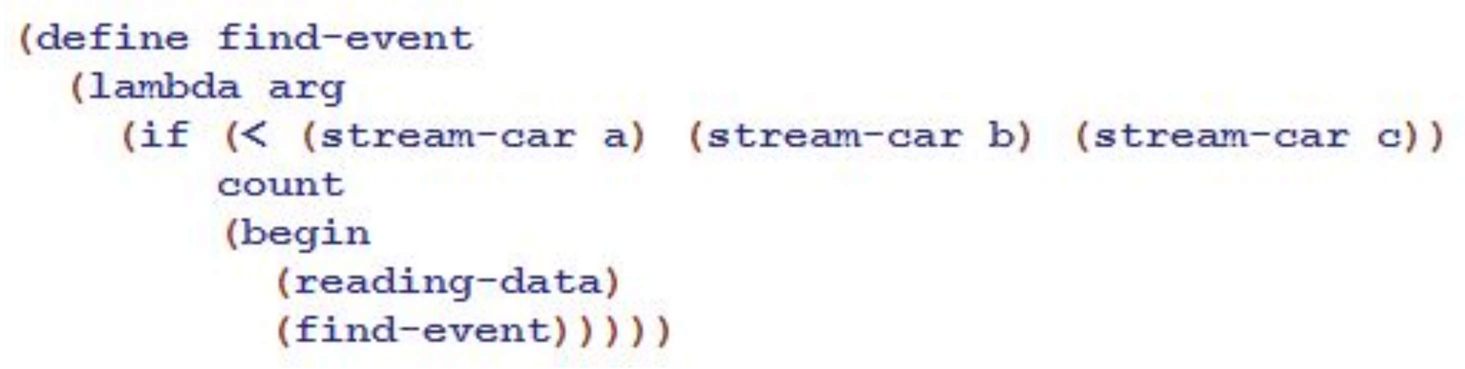


```
(define (my-round number precision)
  (let ((a (round (* number (expt 10 precision)))))
    (* a (expt 10 (* -1 precision)))))
```

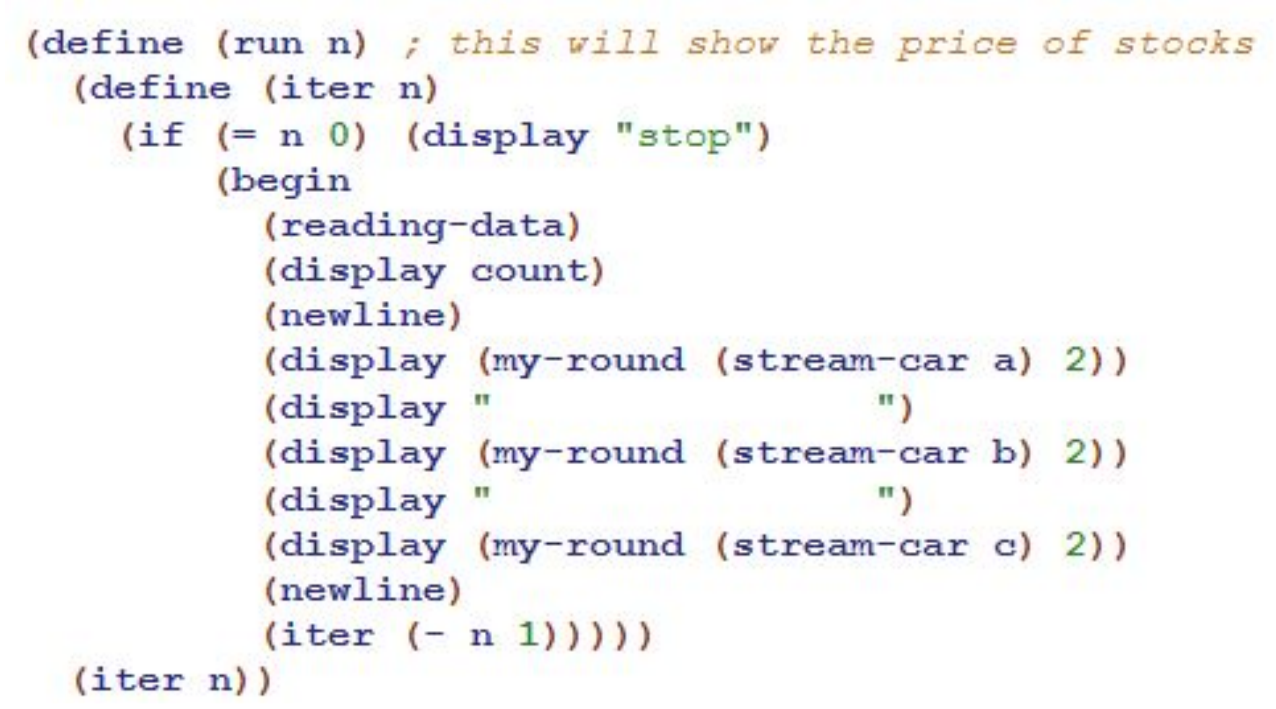
```scheme
(define s-rising
  (lambda (stock)
    (cond ((eq? stock 'a) (begin
                           (set! temp (stream-car a))
                           (cond ((< (randint 0 10) p-a-rise3%) (cons-stream (* temp 1.03) (cons-stream temp a)))
                                 ((>= (randint 0 10) p-a-rise1%) (cons-stream (* temp 1.01) (cons-stream temp a)))
                                 (else (cons-stream (* temp 1.02) (cons-stream temp a))))))
          ((eq? stock 'b) (begin
                            (set! temp (stream-car b))
                            (cond ((< (randint 0 10) p-b-rise3%) (cons-stream (* temp 1.03) (cons-stream temp b)))
                                  ((>= (randint 0 10) p-b-rise1%) (cons-stream (* temp 1.01) (cons-stream temp b)))
                                  (else (cons-stream (* temp 1.02) (cons-stream temp b))))))
          (else (begin
                  (set! temp (stream-car c))
                  (cond ((< (randint 0 10) p-c-rise3%) (cons-stream (* temp 1.03) (cons-stream temp c)))
                        ((>= (randint 0 10) p-c-rise1%) (cons-stream (* temp 1.01) (cons-stream temp c)))
                        (else (cons-stream (* temp 1.02) (cons-stream temp b)))))))))
```

```scheme
(define s-dropping
  (lambda (stock)
    (cond ((eq? stock 'a) (begin
                           (set! temp (stream-car a))
                           (cond ((< (randint 0 10) p-a-drop3%) (cons-stream (* temp 0.97) (cons-stream temp a)))
                                 ((>= (randint 0 10) p-a-drop1%) (cons-stream (* temp 0.99) (cons-stream temp a)))
                                 (else (cons-stream (* temp 0.98) (cons-stream temp a))))))
          ((eq? stock 'b) (begin
                            (set! temp (stream-car b))
                            (cond ((< (randint 0 10) p-b-drop3%) (cons-stream (* temp 0.97) (cons-stream temp b)))
                                  ((>= (randint 0 10) p-b-drop1%) (cons-stream (* temp 0.99) (cons-stream temp b)))
                                  (else (cons-stream (* temp 0.98) (cons-stream temp b))))))
          (else (begin
                  (set! temp (stream-car c))
                  (cond ((< (randint 0 10) p-c-drop3%) (cons-stream (* temp 0.97) (cons-stream temp c)))
                        ((>= (randint 0 10) p-c-drop1%) (cons-stream (* temp 0.99) (cons-stream temp c)))
                        (else (cons-stream (* temp 0.98) (cons-stream temp c)))))))))
```

```
(define reading-data
  (lambda arg
    (begin
      (set! count (+ count 1)) ;; each call will increment the count by 1
      (set! a (if (< (randint 0 10) p-for-a-to-rise)
                  ; if a is rising, determine how much it will rise
                  (s-rising 'a)
                  (s-dropping 'a)))
      (set! b (if (<  (randint 0 10) p-for-b-to-rise)
                  (s-rising 'b)
                  (s-dropping 'b)))
      (set! c (if (<  (randint 0 10) p-for-c-to-rise)
                  (s-rising 'c)
                  (s-dropping 'c)))
      )))
```



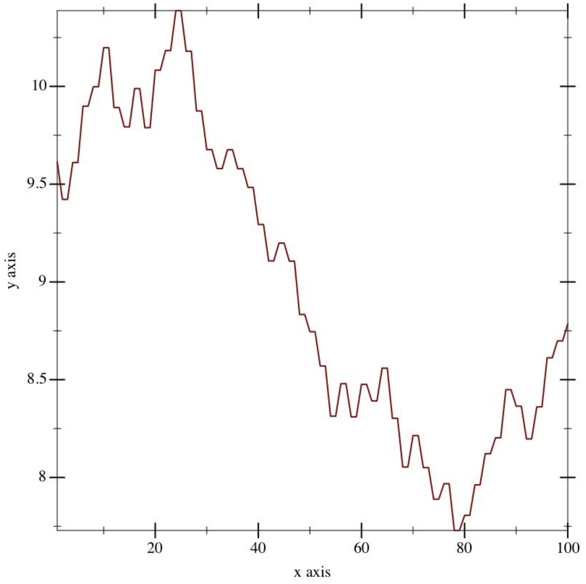
```scheme
(define find-event
  (lambda arg
    (if (< (stream-car a) (stream-car b) (stream-car c))
        count
        (begin
          (reading-data)
          (find-event)))))
```



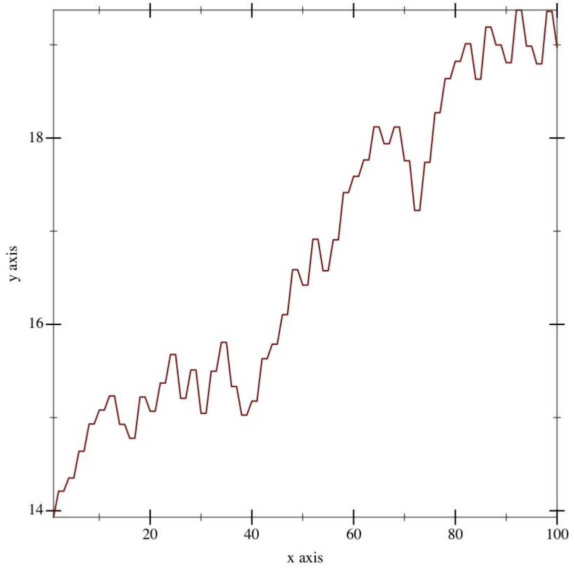
```scheme
(define (run n) ; this will show the price of stocks
  (define (iter n)
    (if (= n 0) (display "stop")
        (begin
          (reading-data)
          (display count)
          (newline)
          (display (my-round (stream-car a) 2))
          (display "                ")
          (display (my-round (stream-car b) 2))
          (display "                ")
          (display (my-round (stream-car c) 2))
          (newline)
          (iter (- n 1)))))
  (iter n))
```

## Result

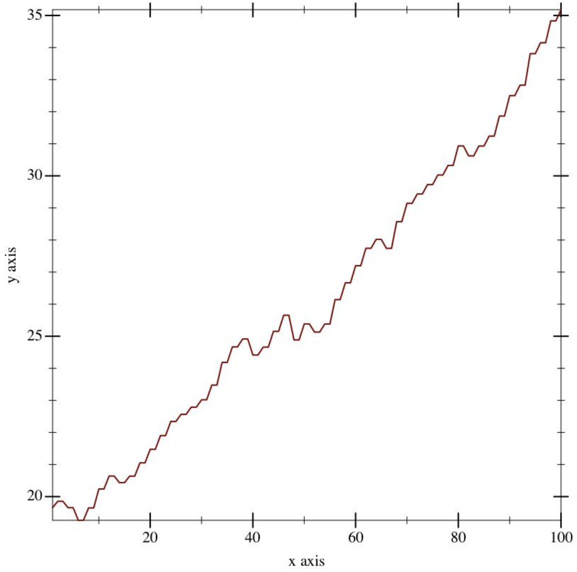
Plots



Stock A



Stock B

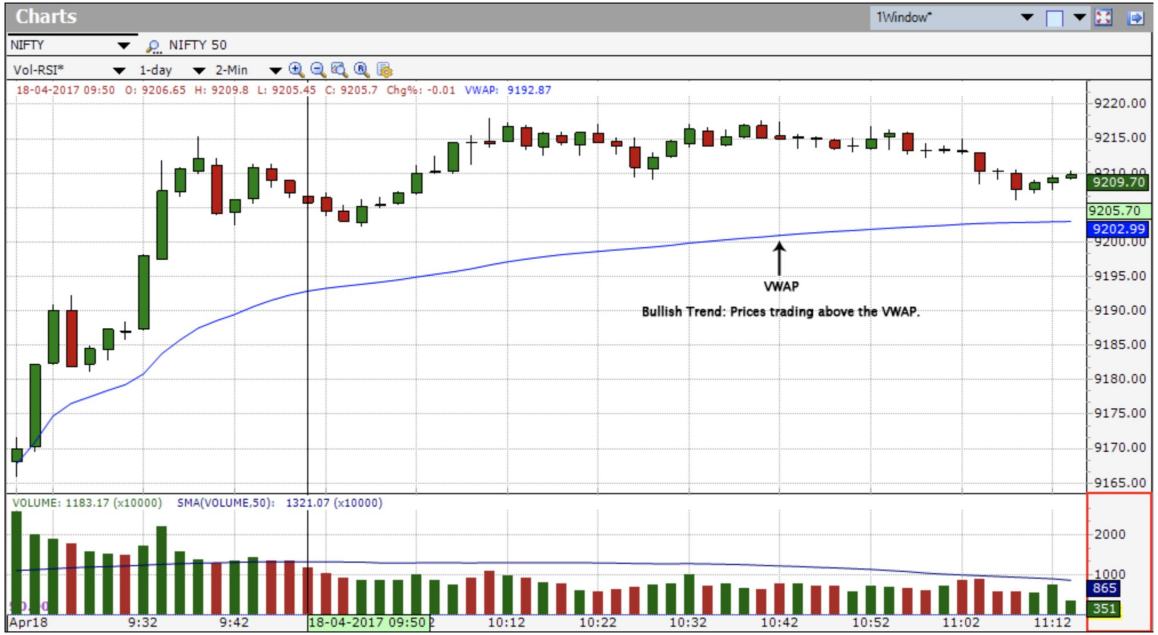


Stock C

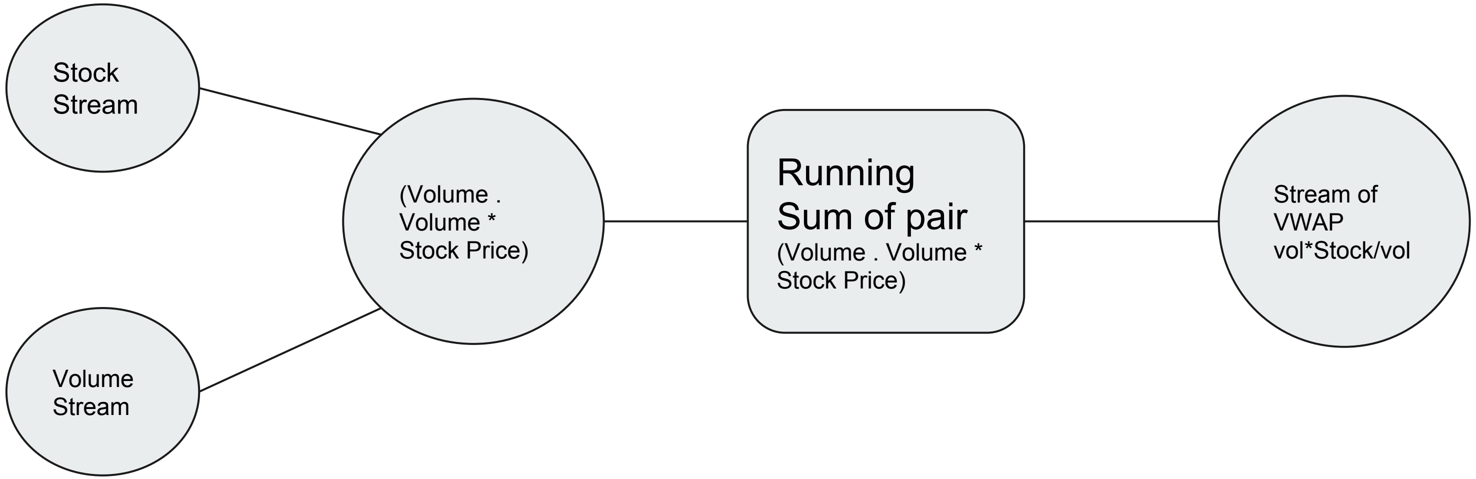
## Volume-Weighted Average Price Calculation

$$\text{VWAP} = \frac{\sum \text{Number of Shares Bought} \times \text{Share Price}}{\text{Total Shares Bought}}$$

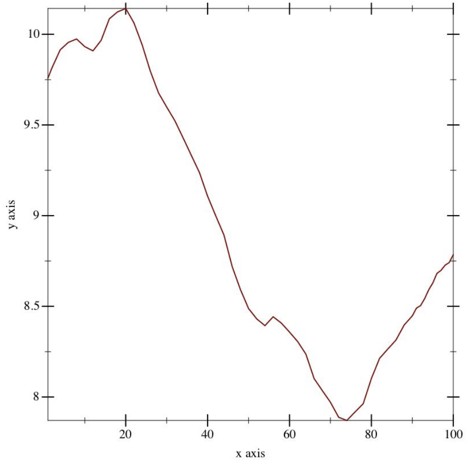
## Real world example



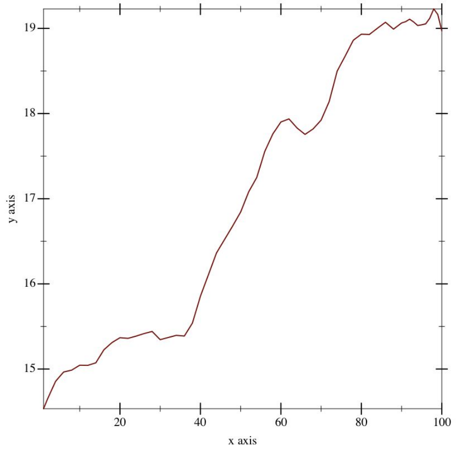
# VWAP Calculation Flow



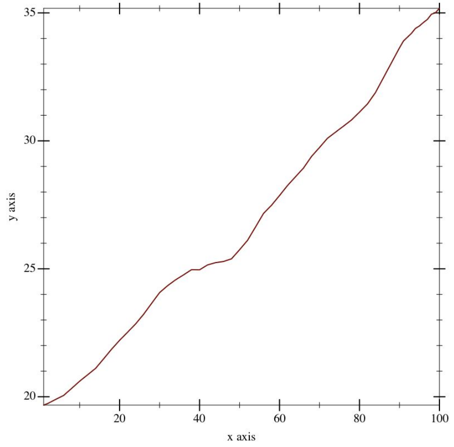
# Volume-Weighted Average Price Calculation



VWAP of Stock A

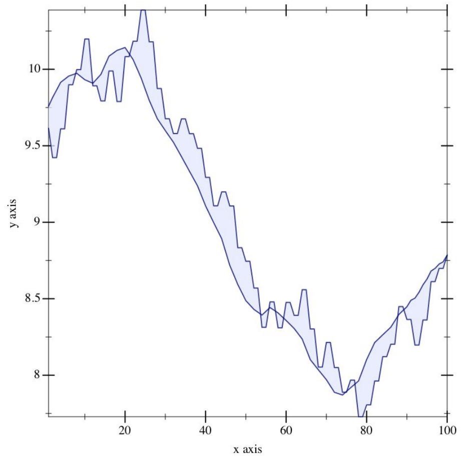


VWAP of Stock B

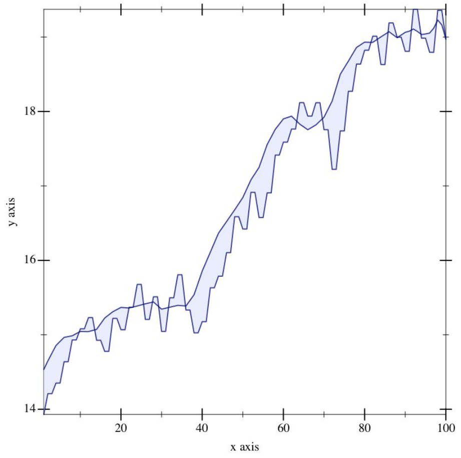


VWAP of Stock C

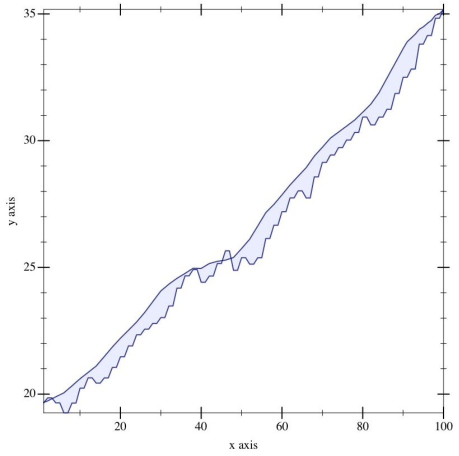
## Visualization of VWAP with the stock price



VWAP with Stock A

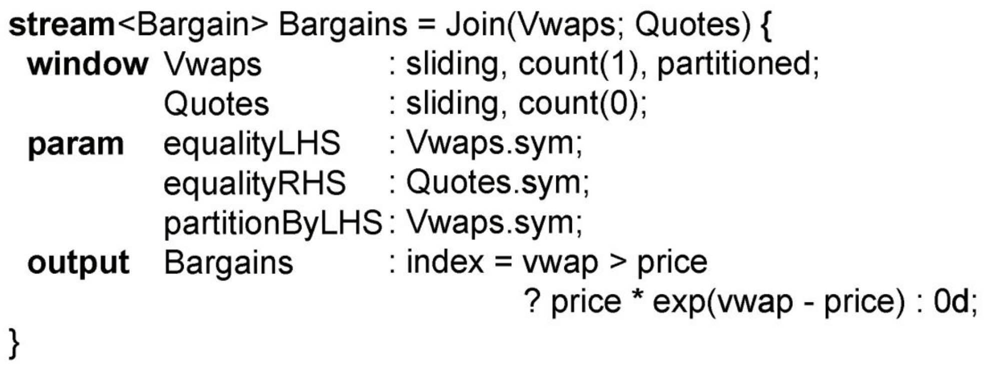


VWAP with Stock A



VWAP with Stock A

## Finding Bargain Index

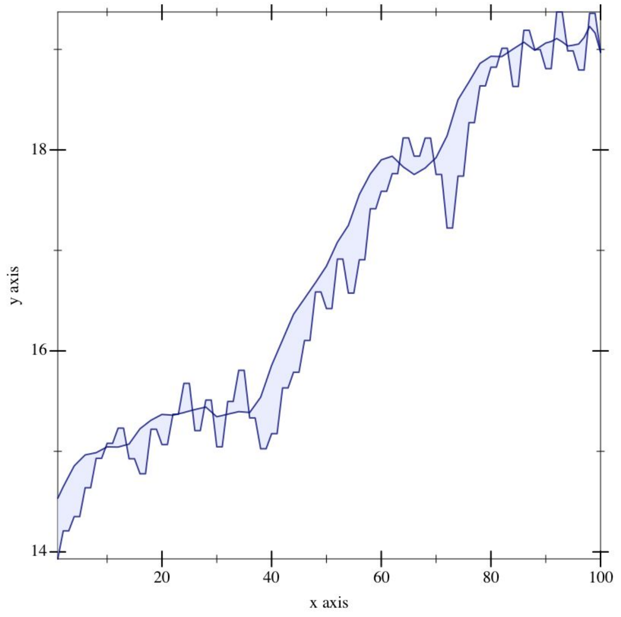


```
stream<Bargain> Bargains = Join(Vwaps; Quotes) {
 window Vwaps          : sliding, count(1), partitioned;
        Quotes         : sliding, count(0);
 param  equalityLHS    : Vwaps.sym;
        equalityRHS    : Quotes.sym;
        partitionByLHS : Vwaps.sym;
 output Bargains       : index = vwap > price
                               ? price * exp(vwap - price) : 0d;
}
```

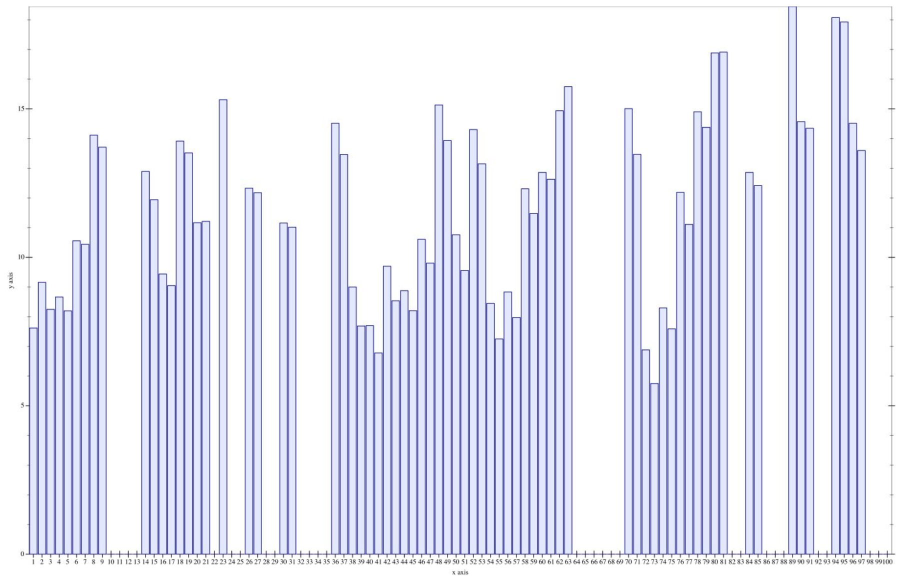
If VWAP > Price then Price * exp(VWAP - Price)
Else 0

## Bargain-index

```scheme
(define (bargain-index stock-stream running-avg-stream)
  (cond
    ((or (stream-null? stock-stream) (stream-null? running-avg-stream)) the-empty-stream)
    ((> (stream-car running-avg-stream) (stream-car stock-stream))
      (cons-stream (* (stream-car stock-stream) (exp (- (stream-car stock-stream) (stream-car running-avg-stream))))
                   (bargain-index (stream-cdr stock-stream) (stream-cdr running-avg-stream))))
    (else (cons-stream 0
                   (bargain-index (stream-cdr stock-stream) (stream-cdr running-avg-stream))))))
```



VWAP and Stock B



Bargain Index for Stock B

Thank you