Infinite Streams in Scheme

Comparing Different Methods to Evaluate accurately the Nth Digit of the Golden Ratio

Programming Language Paradigms

Douglas R Troeger

April 24th 2009

Kanhar Munshi
Asher Snyder

Project Objective:

To implement and evaluate an algorithm employing the concept of infinite streams in Scheme in order to evaluate precisely, the nth digit of an infinite series of numbers.

The series we specifically plan to evaluate is the Golden Ratio or the limit of the ratio of the Fibonacci numbers F_{n+1} over F_n as n tends to infinity.

It is important that our solution does not approximate this value.

We also plan to compare the correctness of our solution by using a more precise mathematical approach to determine the value of the Golden Ratio, exploiting an intrinsic property of the series whereby, the Limit of the Golden Ratio as the number of terms tends to infinity is represented by the formula

```
                       1 + sqrt{5}
Golden Ratio Value =   ------------
                            2
```

Introduction

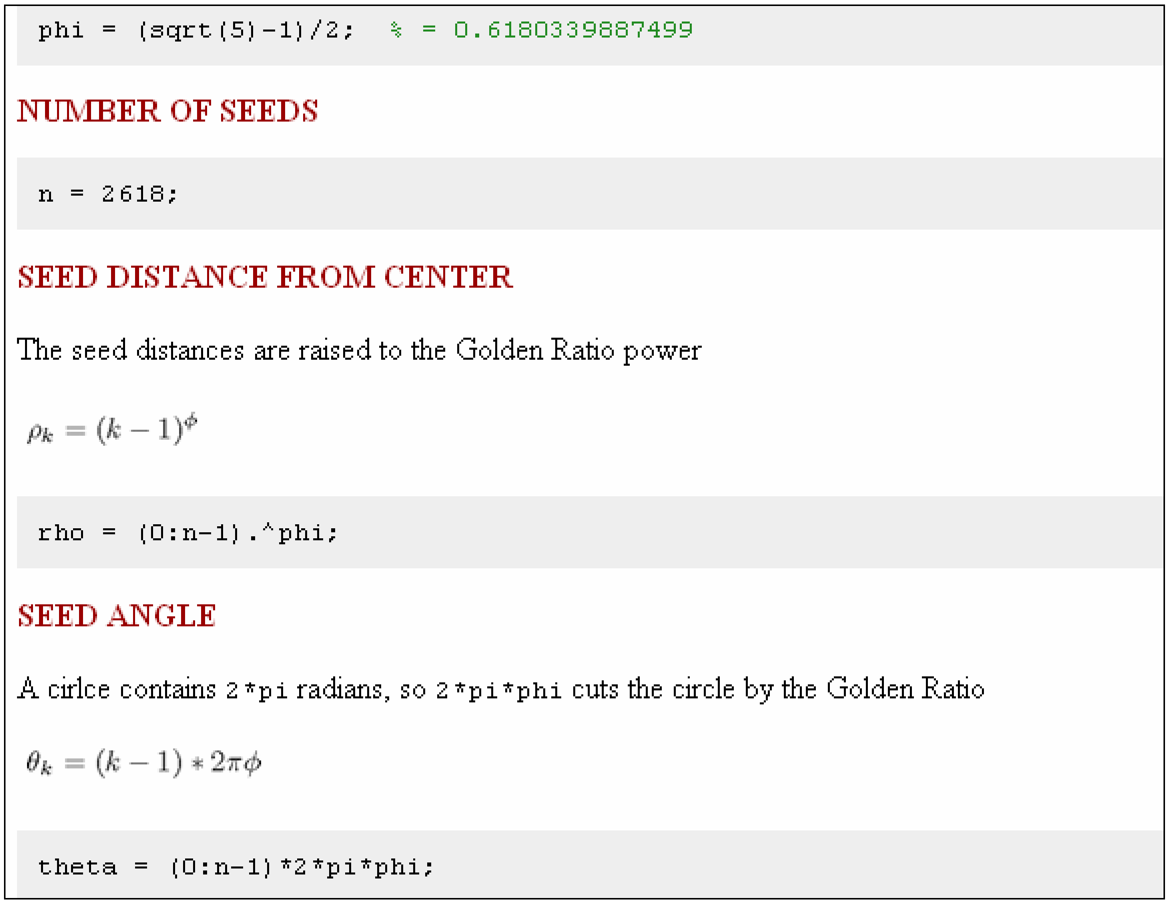
The Golden Ratio, number that arises naturally in mathematics, is one of the most fascinating numbers in the discipline, and because of its occurrence in the natural world, where it is observed in the angular geometry of seeds in sunflowers, coneflowers and pinecones, and numerous other places. It is also used heavily in architecture, and the there is now some proof that starting from the Greek's the ratio was used in some form or the other in most architectural works after.

There of course exist many ways to calculate this ratio (which incidentally is an irrational number) and so is represented by the convergence of either an infinite series or by formulae involving irrational numbers. The ratio is however real, and does not include an imaginary component.

Below are a few ways to evaluate this ratio.

$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\cdots}}}}}}$	1 2 3/2 5/3 8/5 13/8 21/13 34/21 55/34 89/55
Figure 1: The limit of this fraction as the number of Terms tends to infinity.	**Figure 2: The ratio of the F(n+1) th Fibonacci over the F(n) th Fibonacci Term**
$F_n = \frac{\left(\frac{1}{2}\left(1+\sqrt{5}\right)\right)^n - \left(\frac{1}{2}\left(1-\sqrt{5}\right)\right)^n}{\sqrt{5}}$ for $n \in \mathbb{Z}$	$F_{n-1}\,F_{n+1} - (F_n)^2 = (-1)^n$ for (n element Z and n>=0)
Figure 3: Binet's Fibonacci number formula	**Figure 4: Cassini's formula**
$\lim_{n\to\infty} \frac{F(n+1)}{F(n)} = \varphi,$	
Figure 5: Johannes Kepler's limit of consecutive quotients.	

Applications



```
phi = (sqrt(5)-1)/2;  % = 0.6180339887499
```

NUMBER OF SEEDS

```
n = 2618;
```

SEED DISTANCE FROM CENTER

The seed distances are raised to the Golden Ratio power

$$\rho_k = (k-1)^{\phi}$$

```
rho = (0:n-1).^phi;
```

SEED ANGLE

A cirlce contains `2*pi` radians, so `2*pi*phi` cuts the circle by the Golden Ratio

$$\theta_k = (k-1) * 2\pi\phi$$

```
theta = (0:n-1)*2*pi*phi;
```

Figure 6- Matlab code for sunflower

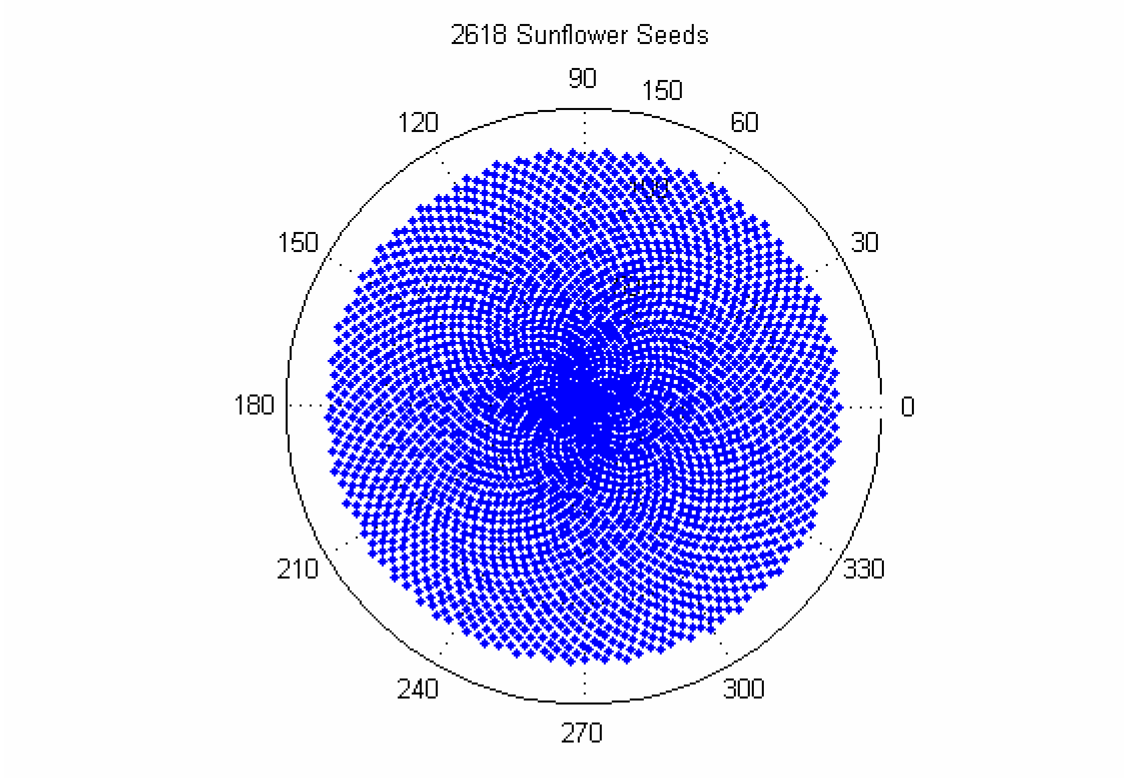


Figure 7- Sunflower generation from Matlab

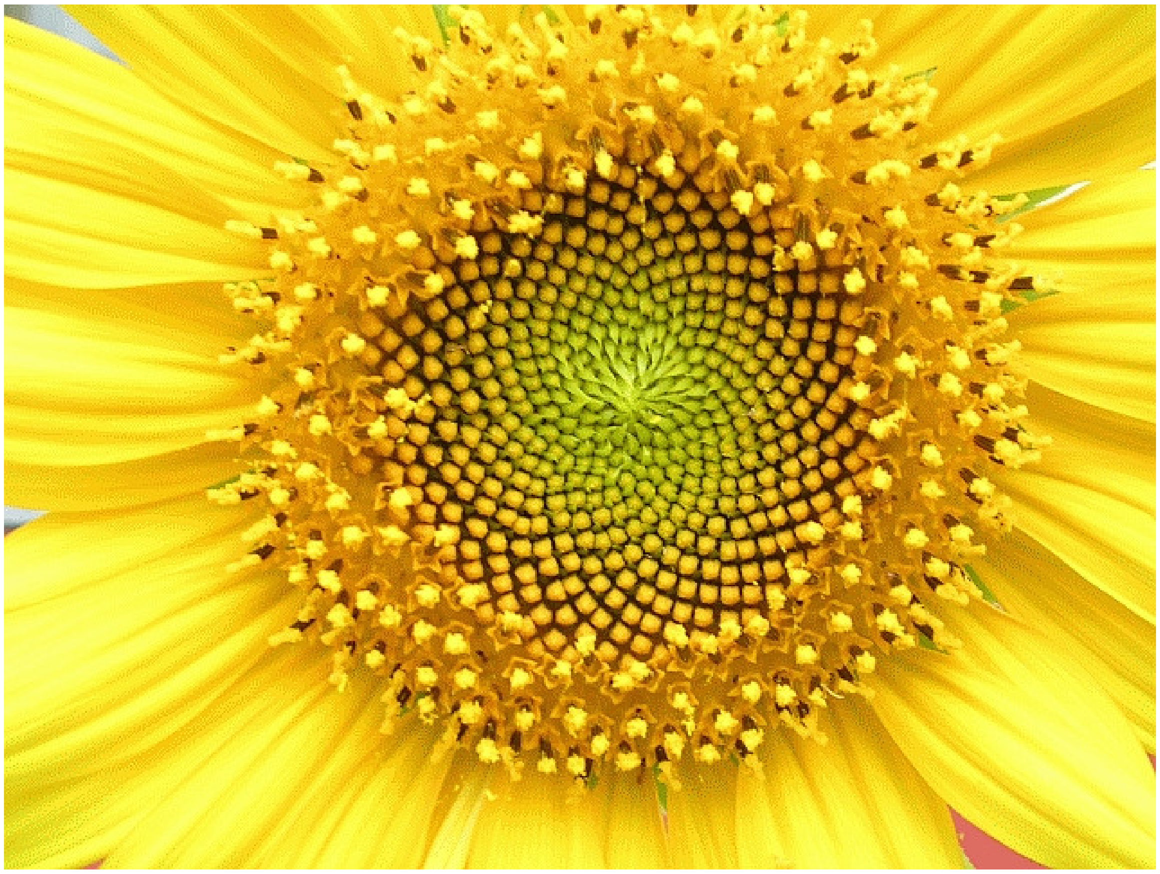


Figure 8- Sunflower head displaying florets in 34 & 35

Implementation

We zeroed in on two methods to approach the problem of generating an infinite Stream that represents the Golden Ratio.

The first method involves computing the ratio of the (N+1)th Fibonacci number to the N th Fibonacci number.

The second method involves computing the ratio (1 + √5)/2. The focus in this method is the computation of the Square Root of Root 5 which we represent by an infinite Stream.

For the implementation of both the above we make use of the following concepts for calculation and for efficiency in our recursion techniques.

1) Infinite Streams
2) Call With Current Continuation
3) Function First-Classness (Improve Guess)

1) Infinite Streams

Scheme allows for delayed evaluation of expressions through the use of keywords **delay,** and then for the evaluation of the expression, through the use of keyword **force**.

a) Cons-Stream
Streams, essentially comprises of one already evaluated value as the first value in the stream, and the promise of another value that would be returned if the stream would be forced to return another value.

This is implemented by the function

```
(define-syntax cons-stream
(syntax-rules ()
  ((cons-stream x y)
   (cons x (delay y)))))
```

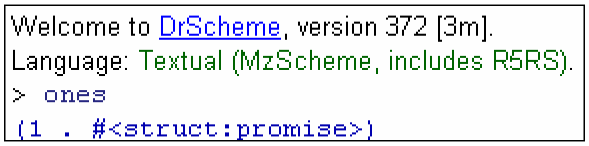
Two things happen above:-
1) It adds cons-stream as a new syntax rule for use again later in the environment (as a syntax)
2) It sets up the definition of cons-stream to imply a pair of objects, where the first object represents a tangible value that can be retrieved immediately while the second object is the promise to return another value if the stream is asked to of course do so. (This is essentially the implication of the keyword delay).

For instance a stream of infinite ones would look like :-

b) Example (Ones)

```scheme
(define ones (cons-stream 1 ones))
```

And we will see the interactive Scheme compiler return the following if we type in the word ones :-



```
Welcome to DrScheme, version 372 [3m].
Language: Textual (MzScheme, includes R5RS).
> ones
(1 . #<struct:promise>)
```

This is the core concept of our implementation.

c) Car | Cdr Equivalent

Of course now we need new methods to access the stream, and we write head and tail functions that return the car and cdr of the stream respectively. Of special importance is the tail function which forces the evaluation of the cdr of the list, one at a time.

```scheme
;Just the First element of the Stream
(define head
  (lambda(stream)
    (car stream)))

;Forces Evaluation of the Cdr of the Stream
(define tail
  (lambda(stream)
    (force(cdr stream))))
```

d) Add | Multiply Support

```scheme
Adds two Streams (in Reality it only adds the First Element of Both Streams,
but Promises to add the Rest of the Stream later, by delaying it)
(define add
  (lambda (s1 s2)
    (cons-stream (+ (head s1) (head s2) )      (add (tail s1) (tail s2)))))

(define multiply
  (lambda (s1 s2)
    (cons-stream (* (head s1) (head s2) )      (multiply (tail s1) (tail s2)))))
```

e) Generic Procedure Support

As you can tell above both add and multiply are virtually similar and I thought to extend this into a function that applies any procedure to the Stream. The result was the function applyproc :-

```
 ;Applies any procedure to the Car of the Stream
 ;and promises to apply it to the cdr of the Stream
(define applyproc
  (lambda (procedure stream)
    (cond
      ((null? stream) '())
      (else (cons-stream (procedure (head stream)) (applyproc procedure (tail stream)))))))
```

Finally, we needed a way to access a) n elements of a stream and b) the nth element of the stream which we accomplished by the methods get and getn appropriately named.

```
;Gets the First N Characters of any Stream, by recursively Forcing
 ;evaluation of the Stream one at a time.
(define get
  (lambda(stream n)
    (cond
      ((= n 0) '())
      (else (cons (head stream) (get (tail stream) (- n 1)))))))

;Gets the Nth Character of any Stream, by discarding all characters
  ;until it reaches the Nth character. N here serves as a counter.
(define getn
  (lambda(stream n)
    (cond
      ((= n 0) (head stream))
      (else (getn (tail stream) (- n 1))))))
```

As you infer both functions are similar except get cons's the results of the natural recursion whereas getn does not instead returning only the last value as n approached zero. Thus it returns the value of the stream at the nth recursion, i.e the nth element of the stream.

<u>f) An Infinite Stream representing the Division of two integers</u>

In needing a way to find the exact representation (although infinite) of the division of two non-zero integers a,b, we can either use the exact-> inexact function provided by Scheme, which has a default precision, or build our own function. Since accuracy is important in our project, we decided to implement our own division support.

Our concept was based on the natural long division method as follows:-

```
;         0.14285
; ----------------
; 7|      10
;          7
;         --
;          30
;          28
;          --
;           20
;           14
;           --
;            60
;            56
;            --
;             40
;             35
;
```

In order to do this, again we used the infinite streams that we have already defined by virtue of cons-stream.

```
;Is used to divide fractional numbers (only where the numerator < denominator)
(define div
  (lambda(a b)
    (cons-stream (quotient a b) (div (* a 10) b))))

;Divide x by y to Generate an Infinite Stream of length n (where y is not zero)
(define divide
  (lambda(x y n)
    (cond
      ((= y 0) 'Division_By_Zero_Error)
      ((< x y) (cons 0 (getn (div x y) n)))
      (else (cons (quotient x y) (getn (div (remainder x y) y) n))))))
```

As is evident cons-stream conses the quotient onto the infinite stream, one quotient at a team, and revalues the expression after multiplying the remainder by a factor of 10 (since these are base 10 numbers) using the same method. The result is an infinite stream representing the division of two integers. Note, there is no truncation error in this approach, since only exact numbers (integers) are dealt with, and the decimal places are in effect just a stream of numbers.

g) Module support

Furthermore, to extend on the principles of Object Oriented Programming and for re-usability purposes, I felt compelled to delineate my functions into two categories those that are concerned with the implementation of stream functionality into my program are placed in a module aptly titled ‘stream’ the contents of which are posted in the appendix.

The modules syntactically are similar to defines, except for the keyword provide that requires you to list the procedures that you wish to be visible to the objects that access your module.

```
;Stream Library With Basic StreamFunctions
;List of Functions
;1) Cons-Stream
;2) Head | Tail
;3) Add | Multiply
;4) Apply Procedure
;5) Get (Get first n Elements of the Stream)
;6) Getn (Get the nth Element of the Stream)
;7) Divide(x,y,n) to Generate an Infinite Stream of length n (where y is not zero)

;These are the Header Files, sort of the Core Functions that Are Used Later on.
(module stream mzscheme

  (define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream x y)
     (cons x (delay y)))))
```

The entire code is attached in Appendix A.

2) Call With Current Continuation

In order to run automated tests of my functions I needed another function that could automatically invoke my other functions with varying parameters. In order to achieve this and to stop when the recursion is over I used Call with current continuations as follows to design a function that can accept as a parameter the function name to be automated, and the number of guesses / terms for it to be automated for.

```
;Evaluates a Function from 1 to N, spaced by a NewLine
(define (loop proc n)
  (call-with-current-continuation
   (lambda (stop)
     (define (inner-loop proc i)
       (cond ((> i n) (stop 'done))
             (else
              (display (proc i 30))
              (newline)
              (inner-loop proc (+ i 1))
             )))
     (inner-loop proc 1))))
```

a) **Fibonacci Method of Approximation**

```
;Infinite Fibonacci Stream
(define fibs
  (cons-stream 0  (cons-stream 1  (add (tail fibs) fibs))))

(define gr_fib_inexact
  (lambda (n)
    ( exact->inexact (/ (getn fibs (+ n 1)) (getn fibs n)) )))

(define gr_fib_exact
  (lambda (n precision)
    (divide (getn fibs (+ n 1)) (getn fibs n) precision )))
```

There are two implementations of the Golden Ratio both of which make use of the fibs infinite Stream, the difference between them being that one converts the exact (fractional) implementation of the ratio to a decimal using Scheme's native exact->inexact routine, whereas the other implementation uses the custom division that we implemented before.

The results show how much more accuracy can be achieved by using an infinite stream to represent the (Exact) division.

b) **Square Root Method of Approximation**

The algorithm we used to compute the Square Root of 5 was the Divide and Average algorithm known to the Babylonians in around 1700 BC.

The following is an extract of this algorithm from www.mathpath.org. The initial guess is arbitrary but must be non-zero, and while the example given below sets A = 2, I set to A =1 for simplicity.

Square roots by Divide-and-Average
This method dating back to the Babylonians (1700 B.C.) is known as the Babylonian Algorithm.
If A > 0 is a guess of the square root of the positive number Q, then a better approximation to √Q is given by
B = (A + Q/A)/2.
Now you use B as an approximation and compute C = (B + Q/B)/2, and so on to get as close as you want to the value of √Q. We have shown elsewhere why the Babylonian Algorithm works.

Example:
Let us find the square root of 5. Let A = 2.
Then B = (2 + 5/2)/2 = 2.25
C = (B + Q/B)/2 = (2.25 + 5/2.25)/2 = (2.25 + 2.22)/2 = 2.235
Notice that this value is getting closer with each iteration to the actual value 2.2360679...

You might wonder how one might even think of Q/A in connection with finding the square root of Q. Well, if A were indeed the square root of Q, then A^2 = Q and so A = Q/A. And even if A is not the square root of Q so that A ^{1} Q/A, it turns out that √Q lies strictly between A and Q/A, and their average B produces a shorter interval from B to Q/B within which √Q still lies and the average of B and Q/B produces an even shorter interval containing √Q, and so on.

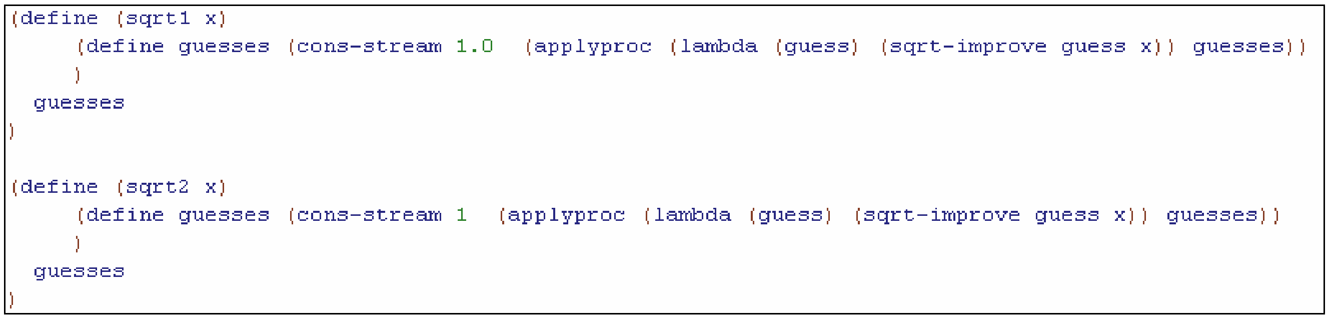
Now as you can from the figure below, and the attached excel document we have four different functions that return the Golden Ratio based on this algorithm.

The reason we have four are because of the rise of exactness as a property that can be used at two levels, inside the function itself and / or at the outer most level.

So as stated the four examples are

1) Exact Function Evaluation Inexact Recursion
2) Inexact Function Evaluation Exact Recursion
3) Inexact Function Evaluation Inexact Recursion
4) Exact Function Evaluation Exact Recursion

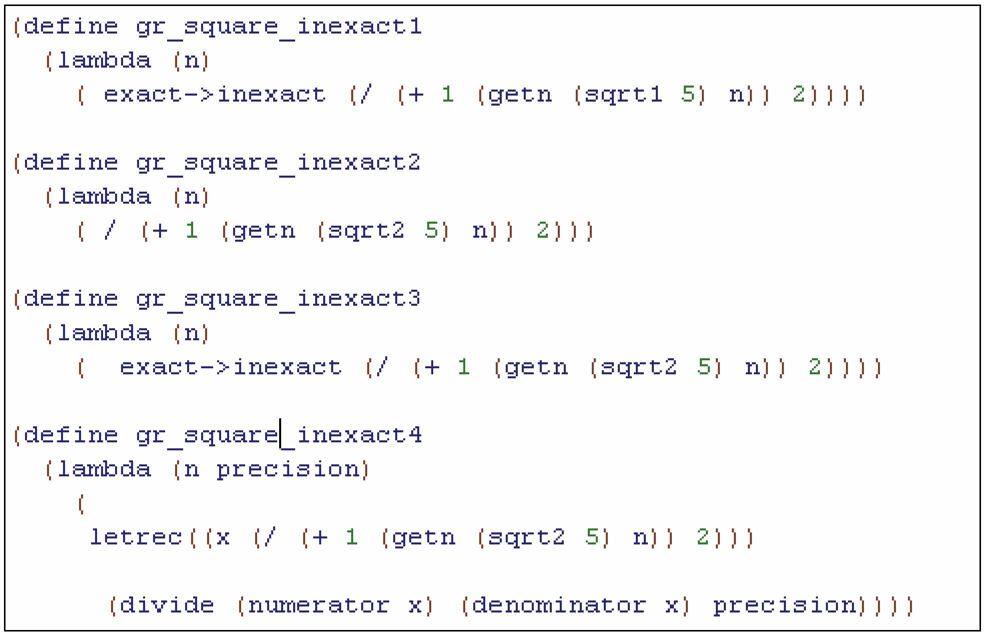
The function sqrt1 employs inexact recursion because by definition it forces conversion to decimal for each average | improve-guess call, due to the 1.0 as seen below.



```
(define (sqrt1 x)
     (define guesses (cons-stream 1.0  (applyproc (lambda (guess) (sqrt-improve guess x)) guesses))
     )
  guesses
)

(define (sqrt2 x)
     (define guesses (cons-stream 1  (applyproc (lambda (guess) (sqrt-improve guess x)) guesses))
     )
  guesses
)
```

Figure 9: Exact and Inexact Square Root Functions



```
(define gr_square_inexact1
  (lambda (n)
    ( exact->inexact (/ (+ 1 (getn (sqrt1 5) n)) 2))))

(define gr_square_inexact2
  (lambda (n)
    ( / (+ 1 (getn (sqrt2 5) n)) 2)))

(define gr_square_inexact3
  (lambda (n)
    (  exact->inexact (/ (+ 1 (getn (sqrt2 5) n)) 2))))

(define gr_square_inexact4
  (lambda (n precision)
    (
     letrec((x (/ (+ 1 (getn (sqrt2 5) n)) 2)))

      (divide (numerator x) (denominator x) precision))))
```

Figure 7: Top Level functions

Conclusion:

We conclude by looking at our data that depicts increasing terms of the Golden Ratio as evaluated, that a greater number of digits lock into certain places as the series increases, and do not change their value as the number of the terms in the series increases.

On further inspection, we notice that if a digit d (say) at index i (i places from the decimal reading left to right) occurs more than once consecutively in two terms of the Golden Ratio (with increasing n) than it locks into that place, and increasing terms of the series does not change the value d at index i.

By referring to Appendix C, we have proved theoretically that a given d where d refers to a digit i places from the decimal, is completely accurate to the correct value of the golden ratio at that index (i places from the decimal) if the associated N for that value of d as given by the formula is greater than the value of n used to evaluate the golden ratio.

$$n = \left[\frac{5}{2}(d+1)\right]$$

APPENDIX A

```
;Stream Library With Basic Stream Functions
;List of Functions
;1) Cons-Stream
;2) Head | Tail
;3) Add | Multiply
;4) Apply Procedure
;5) Get (Get first n Elements of the Stream)
;6) Getn (Get the nth Element of the Stream)
;7) Divide(x,y,n) to Generate an Infinite Stream of length n (where y is not
zero)

;These are the Header Files, sort of the Core Functions that Are Used Later
on.
(module stream mzscheme

  (define-syntax cons-stream
  (syntax-rules ()
    ((cons-stream x y)
     (cons x (delay y)))))

;A Tutorial In Lazy Evaluation in Scheme
;http://www.cs.aau.dk/~normark/prog3-03/html/notes/eval-order_themes-delay-
stream-section.html#eval-order_sieve-ex-more_source-program_sp1
;(delay (+ 5 6))
;Returns; #<struct:promise>
;(force (delay ( + 5 6)))
;11

;Just the First element of the Stream
(define head
  (lambda(stream)
    (car stream)))

;Forces Evaluation of the Cdr of the Stream
(define tail
  (lambda(stream)
    (force(cdr stream))))

;Adds two Streams (in Reality it only adds the First Element of Both Streams,
but Promises to add the Rest of the Stream later, by delaying it)
(define add
  (lambda (s1 s2)
    (cons-stream (+ (head s1) (head s2) )     (add (tail s1) (tail s2)))))

(define multiply
  (lambda (s1 s2)
    (cons-stream (* (head s1) (head s2) )     (multiply (tail s1) (tail
s2)))))

(define applyproc
  (lambda (procedure stream)
    (cond
      ((null? stream) '())
      (else (cons-stream (procedure (head stream)) (applyproc procedure (tail
stream))))))

;Gets the First N Characters of any Stream, by recursively Forcing evaluation
of the Stream one at a time.
(define get
  (lambda(stream n)
    (cond
      ((= n 0) '())
      (else (cons (head stream) (get (tail stream) (- n 1)))))))

;Gets the Nth Character of any Stream, by discarding all characters until it
reaches the Nth character. N here serves as a counter.
(define getn
  (lambda(stream n)
    (cond
      ((= n 0) (head stream))
      (else (getn (tail stream) (- n 1))))))

;Define a List of Infinite Ones
(define ones (cons-stream 1 ones))


(define twos (cons-stream 2 twos))


(define div
  (lambda(a b)
    (cons-stream (quotient a b) (div (* a 10) b))))

;Divide x by y to Generate an Infinite Stream of length n (where y is not
zero)
(define divide
  (lambda(x y n)
    (cond
      ((= y 0) 'Division_By_Zero_Error)
      ((< x y) (cons 0 (getn (div x y) n)))
      (else (cons (quotient x y) (getn (div (remainder x y) y) n))))))


;      0.14285
; ----------------
; 7|   10
;       7
;      --
;       30
;       28
;       --
;        20
;        14
;        --
;         60
;         56
;         --
;          40
;          35
;


 (provide cons-stream get getn head tail ones twos multiply add applyproc
divide div )
 )
```

APPENDIX B

```
;Evaluate Golden Ratio using Different Methods
;Evaluate Golden Ratio using Different Methods
(require (lib "trace.ss"))
;Loads Relevant Stream Functions
(require "stream.scm")

;Get Square Root
(define (average x y ) (/ (+ x y) 2))

(define (sqrt-improve guess x)
     (average guess (/ x guess)))

(define (sqrt1 x)
     (define guesses (cons-stream 1.0  (applyproc (lambda (guess) (sqrt-
improve guess x)) guesses))
     )
  guesses
)

(define (sqrt2 x)
     (define guesses (cons-stream 1  (applyproc (lambda (guess) (sqrt-improve
guess x)) guesses))
     )
  guesses
)

(define gr_fib_inexact1
  (lambda (n)
    ( exact->inexact (/ (+ 1 (getn (sqrt1 5) n)) 2))))

(define gr_fib_inexact2
  (lambda (n)
    ( / (+ 1 (getn (sqrt2 5) n)) 2)))

(define gr_fib_inexact3
  (lambda (n)
    (  exact->inexact (/ (+ 1 (getn (sqrt2 5) n)) 2))))

(define gr_fib_inexact4
  (lambda (n precision)
    (
     letrec((x (/ (+ 1 (getn (sqrt2 5) n)) 2)))

      (divide (numerator x) (denominator x) precision))))

;Infinite Fibonacci Stream
(define fibs
  (cons-stream 0  (cons-stream 1  (add (tail fibs) fibs))))

(define goldenRatioStream
  (lambda (n)
    (/ (getn fibs (+ n 1)) (getn fibs n))))

(define gr_fib_inexact
  (lambda (n)
    ( exact->inexact (/ (getn fibs (+ n 1)) (getn fibs n)) )))

(define gr_fib_exact
  (lambda (n precision)
    (divide (getn fibs (+ n 1)) (getn fibs n) precision )))

(define ge2
  (lambda (n)
    ( exact->inexact (/ (+ 1 (getn (sqrt 5) n)) 2) )))

(define ge2
  (lambda (n)
    (let ((x (/ (+ 1 (getn (sqrt 5) n)) 2)))
      (divide (numerator x) (denominator x) n)
      )))

(define ratio
  (lambda(term precision)
    (display 'Golden_Ratio_using_Fibonacci_at_N= )
    (display term)
    (display "      ")
    (display 'Precision=)
    (display precision)
    (display "   ")
    (display 'is )
    (display "   ")
    (newline)
    (display (ge1 term precision))
    (newline)
    (display 'Golden_Ratio_using_Square_Root_Method )
    (display "     ")
    (display 'Precision=)
    (display precision)
    (display "   ")
    (display 'is )
    (display "   ")
    (newline)
    (display (list (ge2 term )     )
    )))


(define integers
    (lambda(n)
      (cons-stream n (integers (+ n 1)))))

;Evaluates a Function from 1 to N, spaced by a NewLine
(define (loop proc n)
  (call-with-current-continuation
   (lambda (stop)
     (define (inner-loop proc i)
       (cond ((> i n) (stop 'done))
             (else
              (display (proc i 30))
              (newline)
              (inner-loop proc (+ i 1))
             )))
     (inner-loop proc 1))))

(define minterm
    (lambda(d)
      (ceiling (* (/ 5 2) (+ d 1)))
      ))
```

APPENDIX C

Suppose we want $\frac{F(n+1)}{F(n)}$ to approximate θ up to d digits of accuracy. That is:

$$\left|\frac{F(n+1)}{F(n)} - \emptyset\right| < \frac{1}{10^{d+1}}$$

And Hardy gives that:

$$\left|\frac{F(n+1)}{F(n)} - \emptyset\right| < \frac{1}{\sqrt{5}\,F(n)^2}$$

Hence we have:

$$\frac{1}{\sqrt{5}\,F(n)^2} \leq \frac{1}{10^{d+1}}$$

$$\sqrt{5}\,F(n)^2 \geq 10^{d+1}$$

$$F(n) \geq \frac{10^{\frac{d+1}{2}}}{\sqrt[4]{5}}$$

It can be shown for any reference d, there are 4 or 5 Fibonacci having d digits in base 10. Now having the minimum n that satisfies (*) would be most efficient, but if we somehow overshoot n by a little, it would not be a huge deal. We are mostly searching for a walk that works well, not necessarily claiming it is the most efficient way.
In light of these facts, if we observer that (*) expresses information about digit length, and assume that there are 5 (=max{4, 5}) Fibonacci numbers of length d, we have a "solution" for n.

$$n = \left\lceil \frac{5}{2}(d+1) \right\rceil$$

where $\lceil \quad \rceil$ refers to the ceiling function.
What does this mean? Given d, if N=$\left\lceil \frac{5}{2}(d+1) \right\rceil$ then $\frac{F(N+1)}{F(N)} \approx \emptyset$ up to d digits.
Reference: An introduction to the thery of numbers by Hardy and Wright Theorem 193

Reference

1) Title: A Tutorial In Lazy Evaluation in Scheme
URL: http://www.cs.aau.dk/~normark/prog3-03/html/notes/eval-order_themes-delay-stream-section.html#eval-order_sieve-ex-more_source-program_sp1

2) Book: Structure and Interpretation of Computer Programs
Author: Harold Abelson and Gerald Jay Sussman
URL: http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html#%_toc_%_sec_3.5.2

3) "Fibonacci number." *Wikipedia, The Free Encyclopedia*. 19 May 2009
URL:http://en.wikipedia.org/w/index.php?title=Fibonacci_number&oldid=291426865