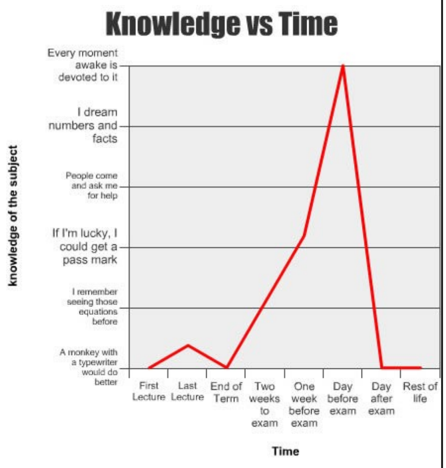
Programming with State & Golden Ages



One-Slide Summary

- The **substitution model** for evaluating Scheme does not allow us to reason about mutation. In the **environment model**:
- A **name** is a **place** for storing a value. `define`, `cons` and function application create places. **`set!`** changes the value in a place.
- Places live in **frames**. An **environment** is a frame and a pointer to a **parent frame**. The **global environment** has no parent.
- To **evaluate** a name, walk up the frames until you find a definition.
- A **golden age** is a period when knowledge or quality increases rapidly.

Outline

- Names and Places
- set! and friends
- Environment Model
- Golden Ages
- There will ***not*** be normally scheduled lab hours or office hours over Fall Reading Days.
- If you are interested in **tutoring**, email cs1120-staff@cs.virginia.edu for more information!

Reading Quiz

- Write your UVA ID on a piece of paper.
- In the Neil deGrass Tyson essay *Science's Endless Golden Age* (assigned reading before today's class), the author focuses primarily on one law. Name it. (Note that multiple laws are mentioned, but one is at the heart of the matter.)

From Lecture 3:

Evaluation Rule 2: Names

If the expression is a ***name***, it evaluates to the value associated with that name.

```
> (define two 2)
> two
2
```

This is called the **substitution model**. You can reason about Scheme expressions by substituting the definition in whenever it is used.

Names and Places

- A name is not just a value, it is a ***place*** for storing a value.
- ***define*** creates a new place, associates a name with that place, and stores a value in that place

(define x 3)

$\boxed{\text{x: 3}}$

Bang!

set! ("set bang") changes the value associated with a place

```
> (define x 3)
> x
3
> (set! x 7)
> x
7
```

x: 7

set! should make you nervous

```
> (define x 2)
> (nextx)
3
> (nextx)
4
> x
4
```

Before **set!** all procedures were pure functions (except for some with side-effects). The value of (f) was the same every time you evaluated it. Now it might be different!

Defining nextx

```
(define (nextx)
  (set! x (+ x 1))
  x)
```

syntactic sugar for

```
(define nextx
  (lambda ()
    (begin
      (set! x (+ x 1))
      x))))
```


Evaluation Rules

```
> (define x 3)
> (+ (nextx) x)
7
or 8
> (+ x (nextx))
9
or 10
```

Evaluation Rules

```
> (define x 3)
> (+ (nextx) x)
7
or 8
> (+ x (nextx))
9
or 10
```

DrScheme evaluates application subexpressions left to right, but Scheme evaluation rules allow any order.

set-car! and **set-cdr!**

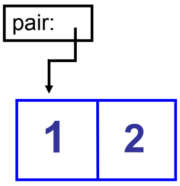
(***set-car!*** *p v*)

Replaces the car of the cons *p* with *v*.

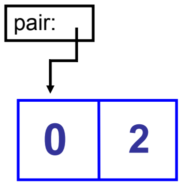
(***set-cdr!*** *p v*)

Replaces the cdr of the cons *p* with *v*. These should scare you even more then set! !

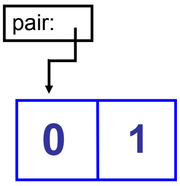
```
> (define pair (cons 1 2))
> pair
(1 . 2)
```




```
> (define pair (cons 1 2))
> pair
(1 . 2)
> (set-car! pair 0)
> (car pair)
0
> (cdr pair)
2
```




```
> (define pair (cons 1 2))
> pair
(1 . 2)
> (set-car! pair 0)
> (car pair)
0
> (cdr pair)
2
> (set-cdr! pair 1)
> pair
(0 . 1)
```



Functional vs. Imperative

Functional Solution: A procedure that takes a procedure of one argument and a list, and returns a list of the results produced by applying the procedure to each element in the list.

```
(define (map proc lst)
  (if (null? lst) null
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```


Imperative Solution

```
(define (map proc lst)
  (if (null? lst) null
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```

A procedure that takes a procedure and list as arguments, and ***replaces*** each element in the list with the value of the procedure applied to that element.

```
(define (map! f lst)
  (if (null? lst) (void)
      (begin
        (set-car! lst (f (car lst)))
        (map! f (cdr lst)))))
```


Programming with Mutation

Imperative

```
> (map! square (intsto 4))
> (define i4 (intsto 4))
> (map! square i4)
> i4
(1 4 9 16)
```

Functional

```
> (define i4 (intsto 4))
> (map square i4)
(1 4 9 16)
> i4
(1 2 3 4)
```

Mutation Changes Everything!

- We can no longer talk about the "value of an expression"
  - The value of a give expression can change!
  - We need to talk about "the value of an expression in an ***execution environment***"
    - "execution environment" = "context so far"
- The order in which expressions are evaluated now matters

Why Substitution Fails

```
> (define (nextx) (set! x (+ x 1)) x)
> (define x 0)
> ((lambda (x) (+ x x)) (nextx))
2
```

Substitution model for evaluation would predict:

```
(+ (nextx) (nextx))
(+ (begin (set! x (+ x 1)) x) (begin (set! x (+ x 1)) x))
(+ (begin (set! 0 (+ 0 1)) 0) (begin (set! 0 (+ 0 1)) 0))
(+ 0 0)
0
```


Liberal Arts Trivia: Astrophysics

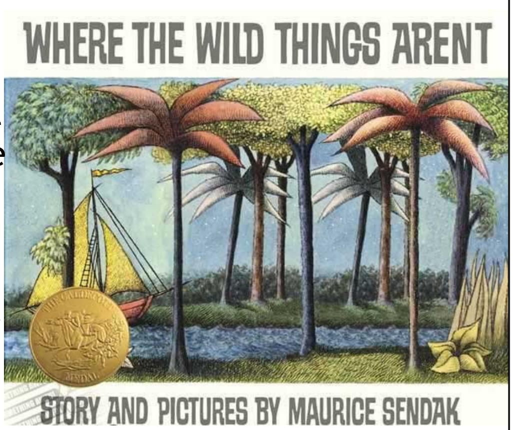
- According to this 1915 theory (be specific), the observed gravitational attraction between masses results from the warping of space and time by those masses. This theory helps to explain observed phenomena, such as anomalies in the orbit of Mercury, that are not predicted by Newton's Laws, and can deal with accelerated reference frames. It is part of the framework of the standard Big Bang model of Cosmology.

Liberal Arts Trivia: Rhetoric

- This type of "values" debate traditionally places a heavy emphasis on logic, ethical values and philosophy. It is a one-on-one debate practiced in National Forensic League competitions. The format was named for the series of seven debates in 1858 for the Illinois seat in the United State Senate.

Very Scary!

- The old substitution model does not explain Scheme programs that contain mutation.
- We need a new **environment model**.



Names and Places

- A name is a **place** for storing a value.
- **define** creates a new place
- **cons** creates two new places, the **car** and the **cdr**
- **(set!** *name expr***)** changes the value in the place *name* to the value of *expr*
- **(set-car!** *pair expr***)** changes the value in the **car** place of *pair* to the value of *expr*

Lambda and Places

- (lambda (x) …) *also* creates a new place named x
- The passed argument is put in that place

> (define x 3)
> ((lambda (x) x) 4)
4
> x
3

x : 3

x : 4

How are these places different?

Location, Location, Location

- Places live in **frames**
- An **environment** is a frame and a pointer to a parent environment
- All environments except the global environment have exactly one **parent environment**, global environment has no parent
- Application creates a new environment

Environments

global environment

+ : #<primitive:+>

null? : #<primitive:null?>

The **global environment** points to the outermost frame. It starts with all Scheme primitives.

Environments

global environment

+ : #<primitive:+>

null? : #<primitive:null?>

x : 3

The global environment points to the outermost frame. It starts with all Scheme primitives.

> (define x 3)
>

Evaluation Rule 2: Names

A *name* expression evaluates to the value associated with that name.

To find the value associated with a name, **look** for the name in the **frame** associated with the evaluation environment. **If** it contains a place with that name, the value of the name expression is the value in that place. **If it doesn't**, the value of the name expression is the value of the name expression evaluated in the **parent** environment if the current environment has a parent. Otherwise, the name expression evaluates to an error (the name is not defined).

Procedures

global environment

+ : #<primitive:+>

null? : #<primitive:null?>

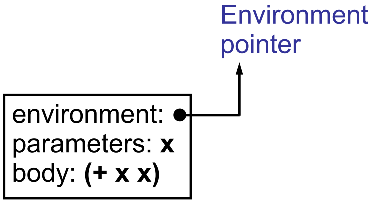
x : 3

double: ***???***

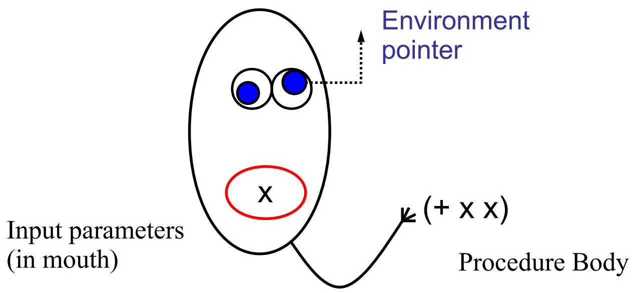
> (define x 3)
> (define double (lambda (x) (+ x x)))
>

How to Draw a Procedure

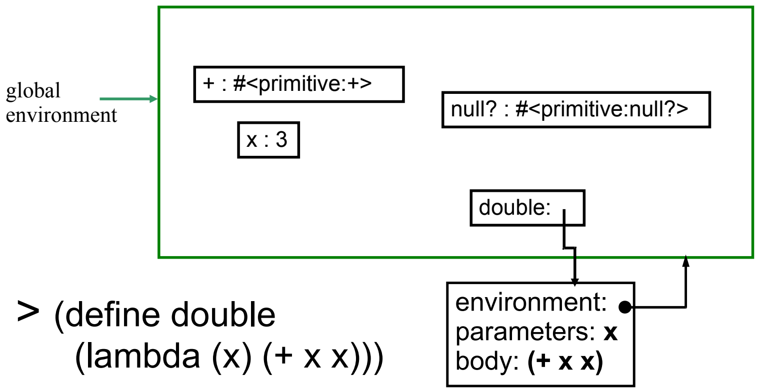
- A procedure needs **both code and an environment**
  - We'll see why soon
- We **draw procedures** like this:



How to Draw a Procedure (for artists only)



Procedures




```
> (define double
    (lambda (x) (+ x x)))
```


Application

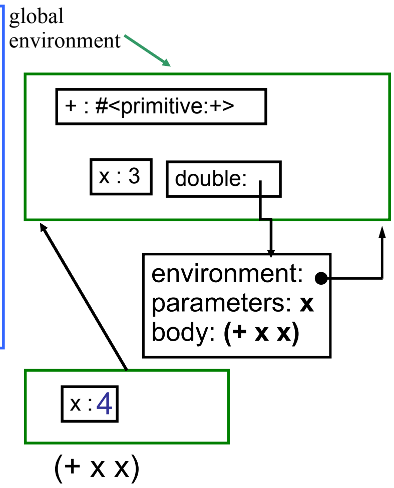
- Old rule: (Substitution model)

  Apply Rule 2: Constructed Procedures. To apply a constructed procedure, **evaluate** the body of the procedure with each formal parameter replaced by the corresponding actual argument expression value.

New Application Rule 2:

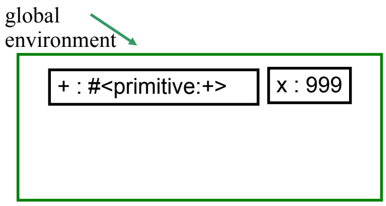
1. **Construct a new environment**, whose parent is the environment to which the environment pointer of the applied procedure points.
2. **Create places** in that frame for each parameter containing the value of the corresponding operand expression.
3. **Evaluate the body in the new environment**. Result is the value of the application.

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



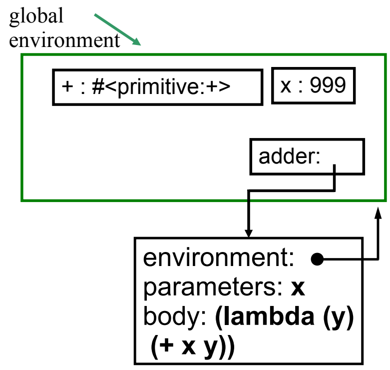

```
> (double 4)
8
```

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



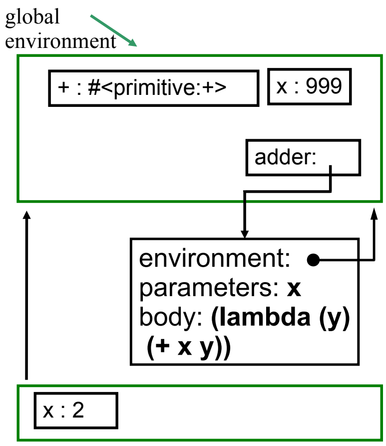
> (define x 999)

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



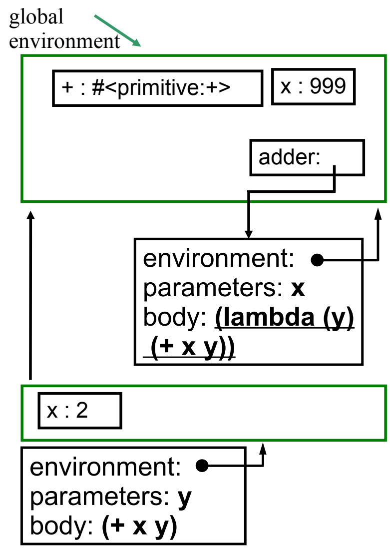
> (define x 999)
> (define (adder x)
    (lambda (y) (+ x y))))

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



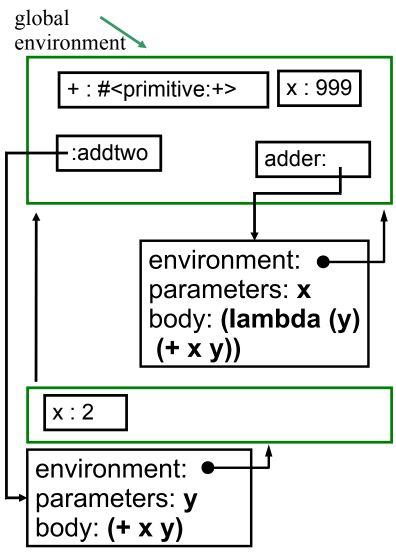
> (define x 999)
> (define (adder x)
    (lambda (y) (+ x y))))
> (define addtwo (adder 2))

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



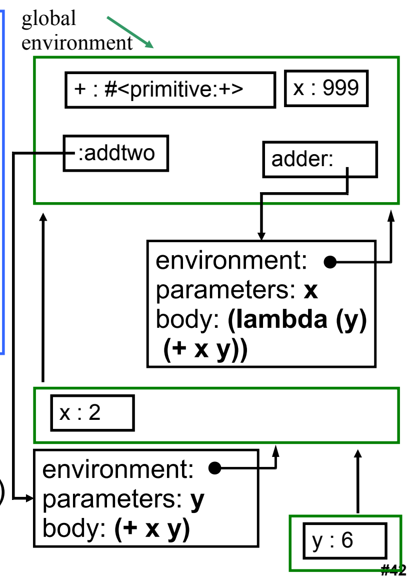
> (define x 999)
> (define (adder x)
    (lambda (y) (+ x y))))
> (define addtwo (adder 2))

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



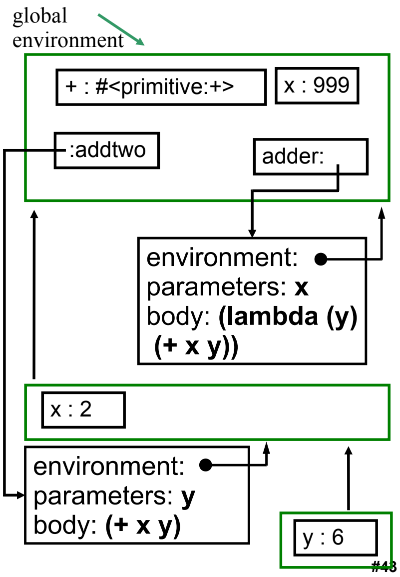
> (define x 999)
> (define (adder x)
    (lambda (y) (+ x y))))
> (define addtwo (adder 2))

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment



> (define x 999)
> (define (adder x)
    (lambda (y) (+ x y))))
> (define addtwo (adder 2))
> (addtwo 6)

1. Construct a new environment, parent is procedure's environment pointer
2. Make places in that frame with the names of each parameter, and operand values
3. Evaluate the body in the new environment




```
> (define x 999)
> (define (adder x)
     (lambda (y) (+ x y))))
> (define addtwo (adder 2))
> (addtwo 6)
8
```


Liberal Arts Trivia: Statistics

- In probability theory and statistics, *this* indicates the strength and direction of a linear relationship between two random variables. A number of different coefficients are used in different situations, the best known of which is the Pearson product-moment coefficient. Notably, this concept does not imply causation.

Liberal Arts Trivia: Art History

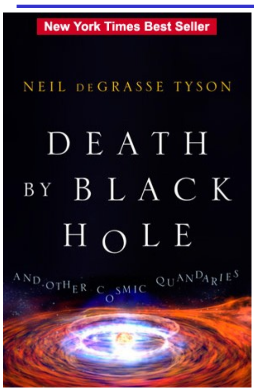
- *This* was a popular international art design movement from 1925 until the 1940s, affecting the decorative arts such as architecture, interior design and industrial design, as well as the visual arts such as fashion, painting, the graphic arts and film. At the time, this style was seen as elegant, glamorous, functional and modern.

Liberal Arts Trivia: Music

- This baroque keyboard instrument is the spiritual predecessor of the pianoforte. It produces a sound by plucking a string when each key is pressed, but unlike the piano it lacks responsiveness to keyboard touch and thus fails to produce notes at different dynamic levels.

Jan Vermeer, 1670

Science's Endless Golden Age



http://www.pbs.org/wgbh/nova/sciencenow/3313/nn-video-toda-w-220.html

Astrophysics

"If you're going to use your computer to simulate some phenomenon in the universe, then it only becomes interesting if you change the scale of that phenomenon by at least a factor of 10. … For a 3D simulation, an increase by a factor of 10 in each of the three dimensions increases your volume by a factor of 1000."

- How much work is astrophysics simulation (in Θ notation)?

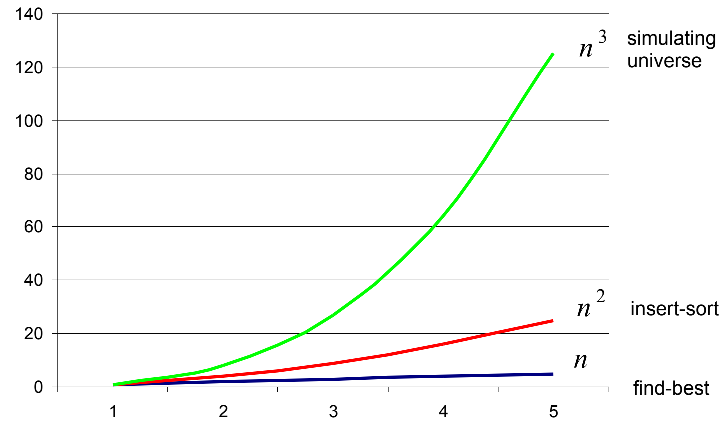
Astrophysics

- "If you're going to use your computer to simulate some phenomenon in the universe, then it only becomes interesting if you change the scale of that phenomenon by at least a factor of 10. … For a **3D** simulation, an increase by a factor of **10** in each of the three dimensions increases your volume by a factor of **1000**."
- How much work is astrophysics simulation (in Θ notation)?

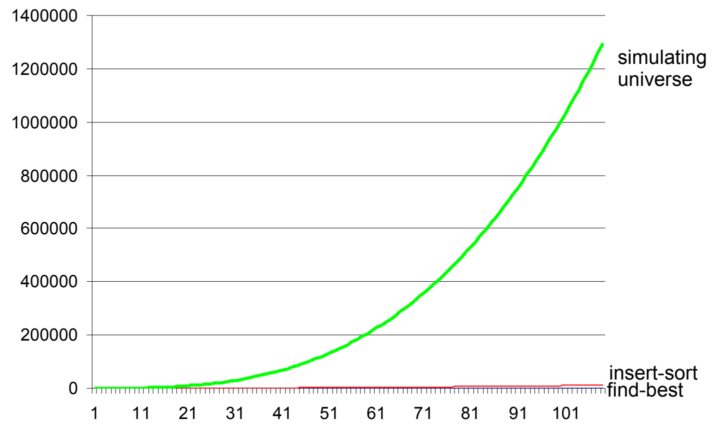
$$\Theta(n^3)$$

When we double the size of the simulation, the work octuples! (Just like oceanography octopi simulations)

Orders of Growth



Orders of Growth



Astrophysics and Moore's Law

- Simulating universe is $\Theta(n^3)$
- Moore's law: computing power doubles every 18 months
- Dr. Tyson: to understand something new about the universe, need to scale by 10x
- How long does it take to know *twice as much* about the universe?

Knowledge of the Universe

```
;;; doubling every 18 months = ~1.587 * every 12 months
(define (computing-power nyears)
  (if (= nyears 0) 1
     (* 1.587 (computing-power (- nyears 1)))))

;;; Simulation is Θ(n³) work
(define (simulation-work scale)
   (* scale scale scale))

(define (log10 x)  (/ (log x) (log 10))) ;;; log is base e
;;; knowledge of the universe is log10 the scale of universe
;;; we can simulate
(define (knowledge-of-universe scale) (log10 scale))
```


Knowledge of the Universe

```
(define (computing-power nyears)
  (if (= nyears 0) 1 (* 1.587 (computing-power (- nyears 1)))))
  ;;; doubling every 18 months = ~1.587 * every 12 months
(define (simulation-work scale) (* scale scale scale))
   ;;; Simulation is O(n^3) work
(define (log10 x)  (/ (log x) (log 10)))
   ;;; primitive log is natural (base e)
(define (knowledge-of-universe scale) (log10 scale))
  ;;; knowledge of the universe is log 10 the scale of universe we can simulate
(define (find-knowledge-of-universe nyears)
  (define (find-biggest-scale scale)
    ;;; today, can simulate size 10 universe = 1000 work
    (if (> (/ (simulation-work scale) 1000)
           (computing-power nyears))
        (- scale 1)
        (find-biggest-scale (+ scale 1))))
  (knowledge-of-universe (find-biggest-scale 1)))
```

> (find-knowledge-of-universe 0)
1.0
> (find-knowledge-of-universe 1)
1.041392685158225
> (find-knowledge-of-universe 2)
1.1139433523068367
> (find-knowledge-of-universe 5)
1.322219294733919
> (find-knowledge-of-universe 10)
1.6627578316815739
> (find-knowledge-of-universe 15)
2.0
> (find-knowledge-of-universe 30)
3.00560944536028
> (find-knowledge-of-universe 60)
5.0115366121349325
> (find-knowledge-of-universe 80)
6.348717927935257

Only two things are infinite, the universe and human stupidity, and I'm not sure about the former.
Albert Einstein

Will there be any mystery left in the Universe when you die?

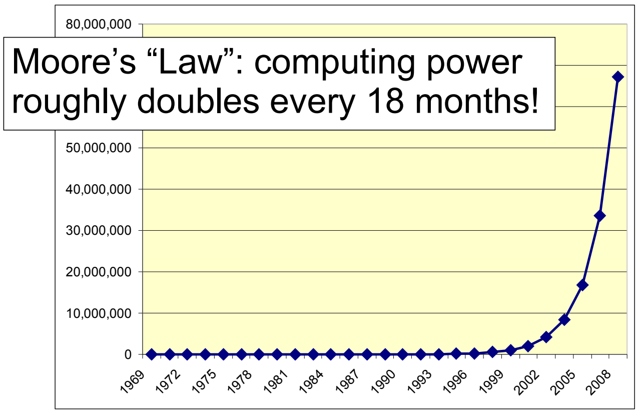
The Endless Golden Age

- **Golden Age**: period in which knowledge/quality of something doubles quickly
- At any (recent) point in history, half of what is known about astrophysics was discovered in the previous 15 years!
  - Moore's law today, but other advances previously: telescopes, photocopiers, clocks, agriculture, etc.

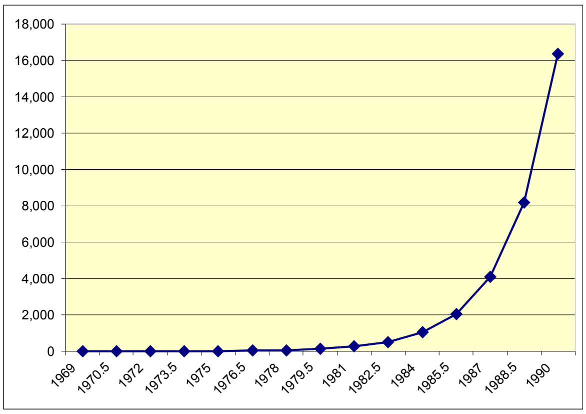
Endless/Short Golden Ages

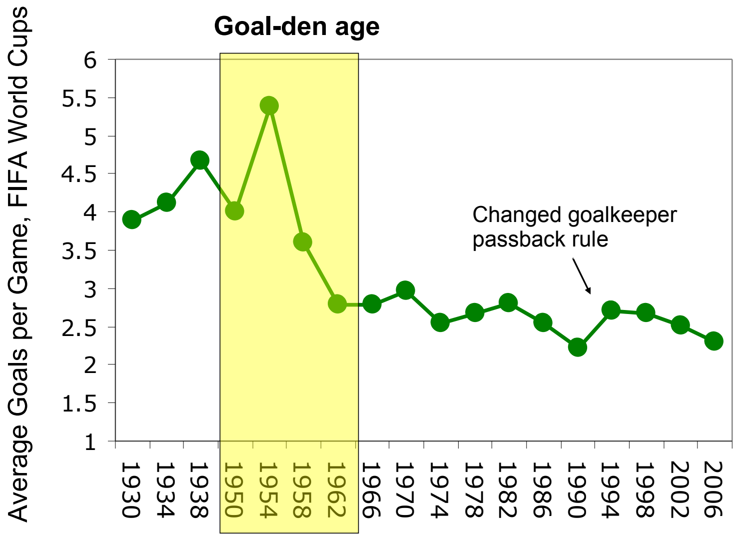
- **Endless golden age**: at any point in history, the amount known is twice what was known 15 years ago
  - Always exponential growth: $\Theta(k^n)$
    k is some constant, n is number of years
- **Short golden age**: knowledge doubles during a short, "golden" period, but only improves linearly most of the time
  - Usually linear growth: $\Theta(n)$
    n is number of years

Computing Power 1969-2008 (in Apollo Control Computer Units)



Computing Power 1969-*1990* (in Apollo Control Computer Units)





Endless Golden Age and “Grade Inflation”

- Average student gets twice as smart and well-prepared every 15 years
  - You had grade school teachers (maybe even parents) who went to college!
- If average GPA in 1980 is 2.00 what should it be today (if grading standards didn’t change)?

Grade Inflation or Deflation?

2.00	average GPA in 1980 (“gentle C”?)	
* 2	better students 1980-1995	
* 2	better students 1995-2010	
* 1.49	population increase	Virginia 1976: ~5.1M Virginia 2006: ~7.6M
* 0.74	increase in enrollment	Students 1976: 10,330 Students 2006: 13,900

Average GPA today should be: 8.82
(but our expectations should also increase)

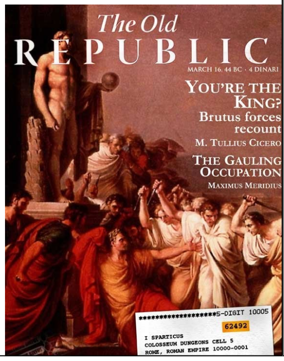
The Real Golden Rule?

Why do fields like astrophysics, medicine, biology and computer science have “endless golden ages”, but fields like ...

- rock n’ roll (1962-1973, or whatever was popular when you were 16)
- music (1775-1825)
- philosophy (400BC-350BC?)
- art (1875-1925?)
- soccer (1950-1966)
- baseball (1925-1950?)
- movies (1920-1940?)

have short golden ages?

Think about it before next class!



Homework

- Start PS 5 now!
  - Due Monday October 18th
  - It is longer than PS4.
  - If you wait, you will probably not have enough time.
- Read Course Book 9 (again) and 10