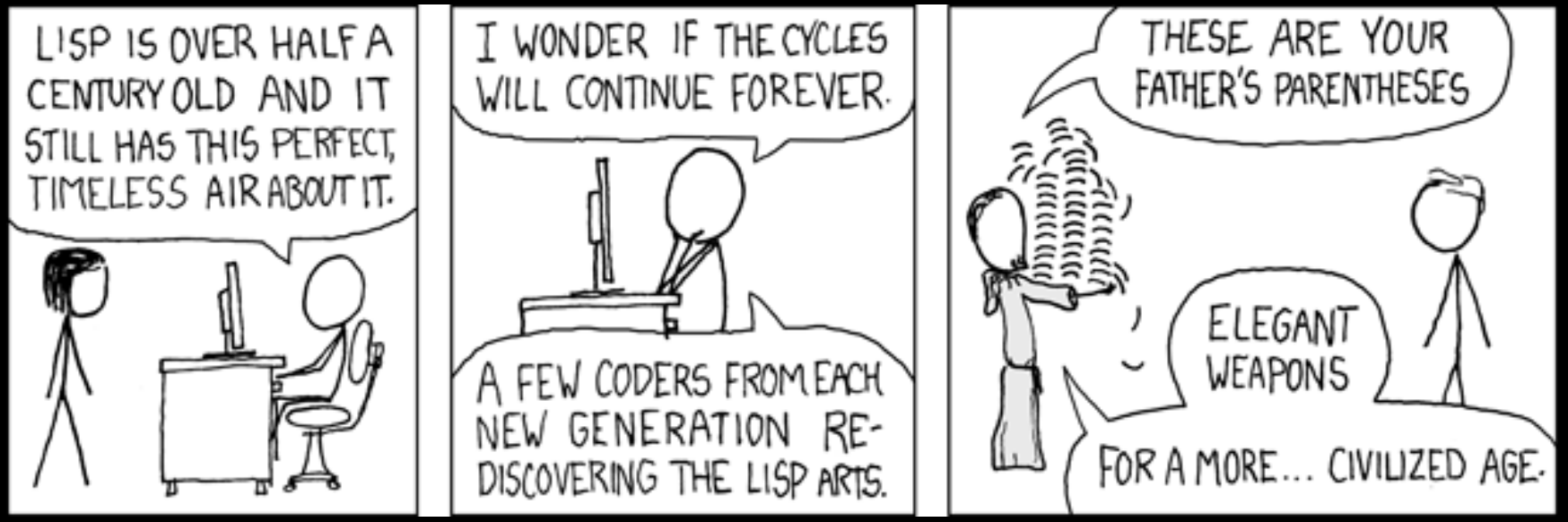


"Lisp Cycles", http://xkcd.com/297/

CS 152: *Programming Language Paradigms*

Introduction to Scheme

Prof. Tom Austin
San José State University

Key traits of Scheme

1. Functional
2. Dynamically typed
3. Minimal
4. Lots of lists!

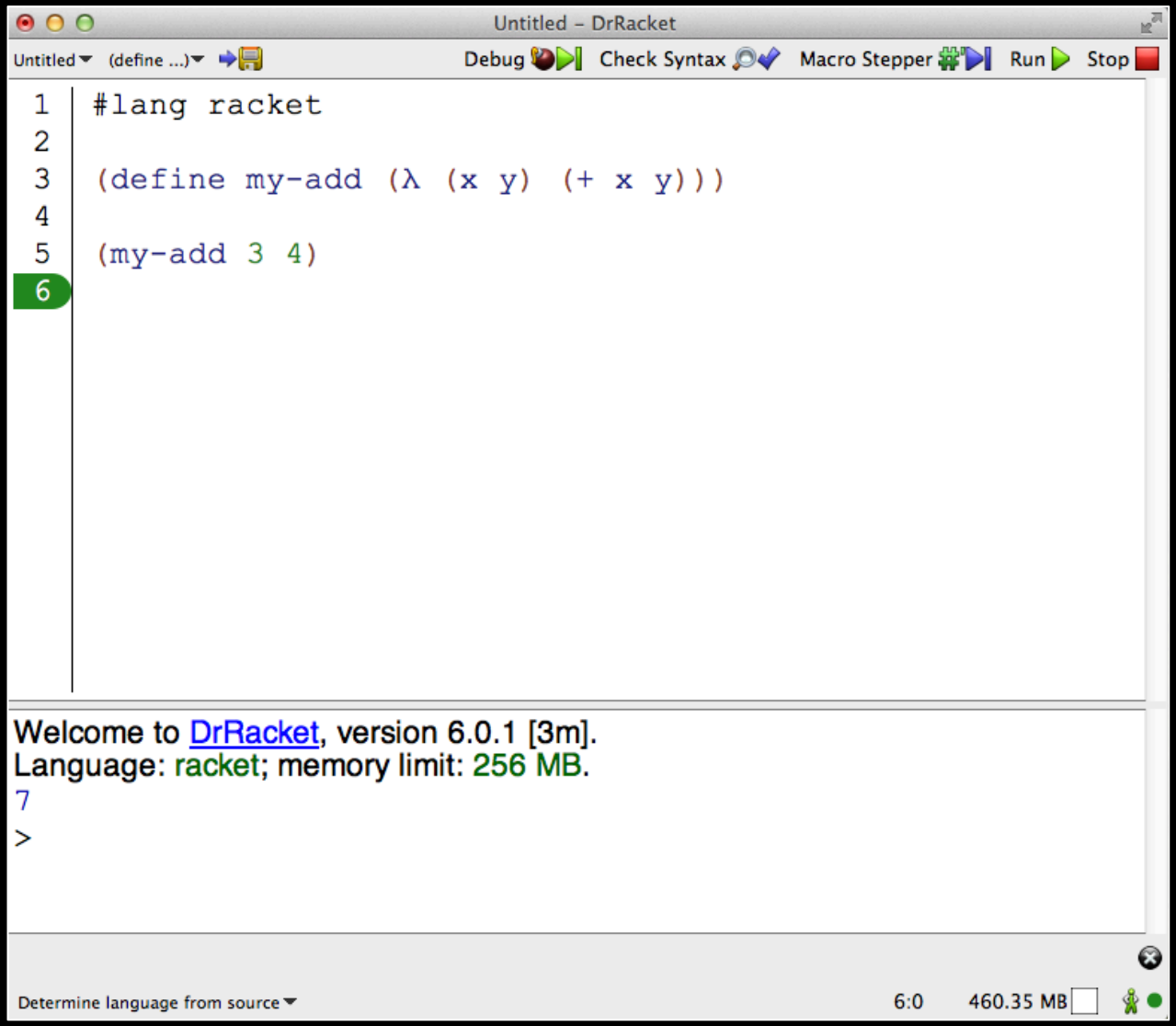
Interactive Racket

```
$ racket
Welcome to Racket
v6.0.1.
> (* (+ 2 3) 2)
10
>
```

Running Racket from Unix Command Line

```
$ cat hello-world.rkt
#lang racket
(displayln "Hello world")
$ racket hello-world.rkt
Hello world
$
```

DrRacket


Untitled – DrRacket
Untitled
(define ...)
Debug
Check Syntax
Macro Stepper
Run
Stop
1 #lang racket
2
3 (define my-add (λ (x y) (+ x y)))
4
5 (my-add 3 4)
6
Welcome to DrRacket, version 6.0.1 [3m].
Language: racket; memory limit: 256 MB.
7
>
Determine language from source
6:0
460.35 MB

Racket Simple Data Types

- Booleans: `#t, #f`
  - `(not #f)` => `#t`
  - `(and #t #f)` => `#f`
- Numbers: `32, 17`
  - `(/ 3 4)` => `3/4`
  - `(expt 2 3)` => `8`
- Characters: `#\c, #\h`

Racket Compound Data Types

- Strings
  - `(string #\S #\J #\S #\U)`
  - `"Spartans"`
- Pairs
- Lists

What is a pair?

```
> (pair? '(1 2))
#t
> (pair? '(1 2 3))
#t
> (pair? '(1))
#t
> (pair? '())
#f
> (list? '())
#t
>
```

Pair Functions

- **`cons`** – make a pair
  `(cons 3 4) -> '(3 . 4)`
- **`car`** – get first element
  `(car (cons 3 4)) -> 3`
- **`cdr`** – get second element
  `(cdr (cons 3 4)) -> 4`

Lists

A list is **either**:

- the empty list (**`null`**)
- A pair of:
  - The head of the list
  - The tail of the list
    i.e., the list containing everything except the head.

List functions

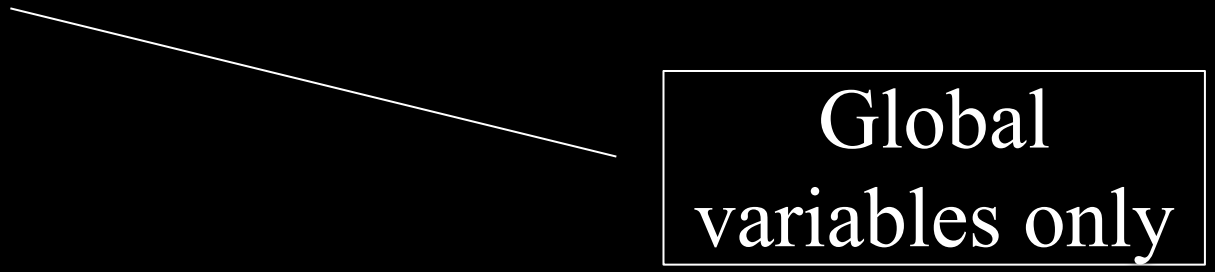
```
> (cons 1 '(2 3))
'(1 2 3)
> (cons 1 '())
'(1)
> (cons 1 null)
'(1)
> (append '(1 2) '(3 4))
'(1 2 3 4)
>
```

List functions, continued

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
'(2 3)
> (empty? '(1 2 3))
#f
> (empty? '())
#t
> (null? '())
#t
>
```

Setting values

```
(define zed "Zed")
```



Setting values

```
(define zed "Zed")
(displayln zed)

{let
   ([z2 (string-append zed zed)]
    [sum (+ 1 2 3 4 5)])
   (displayln z2)
   (displayln sum)}
```

Setting values

```
(define zed "Zed")
(displayln zed)

{let
   ([z2 (string-append zed zed)]
    [sum (+ 1 2 3 4 5)])
   (displayln z2)
   (displayln sum)}
```

List of local variables

Setting values

```
(define zed "Zed")
(displayln zed)

{let
   ([z2 (string-append zed zed)]
    [sum (+ 1 2 3 4 5)])
   (displayln z2)
   (displayln sum)}
```

Setting values

```
(define zed "Zed")
(displayln zed)

{let
   ([z2 (string-append zed zed)]
    [sum (+ 1 2 3 4 5)])
   (displayln z2)
   (displayln sum)}
```

Variable definitions

Setting values

```
(define zed "Zed")
(displayln zed)

{let
   ([z2 (string-append zed zed)]
    [sum (+ 1 2 3 4 5)])
   (displayln z2)
   (displayln sum)}
```

let and let*

```
(let   ([x 3]
        [y (* x 2)])
  (displayln y))
```

```
x: unbound identifier
in module in: x
```

let and let*

```
(let* ([x 3]
       [y (* x 2)])
  (displayln y))
```

```
6
```

All the data types discussed so far are called *s-expressions* (s for symbolic).

Note that programs themselves are also s-expressions. **Programs are data.**

Lambdas (λ)

```
(lambda
  (x)
  (* x x))
```

Also known as "functions"

Lambdas (λ)

```
(λ
  (x)
  (* x x))
```

In DrRacket, λ can be typed with Ctrl + \

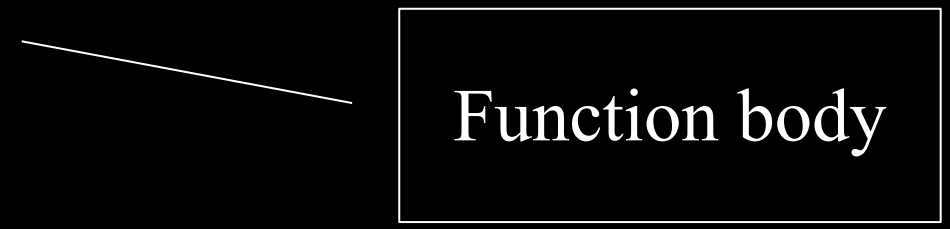
Lambdas (λ)

```
(lambda
  (x)
  (* x x))
```

Parameter list

Lambdas (λ)

```
(lambda
  (x)
  (* x x))
```



Lambdas (λ)

```
((lambda
   (x)
   (* x x))
 3)
```

Evaluates to 9

Lambdas (λ)

```
(define square
   (lambda (x)(* x x)))
(square 4)
```

Alternate Format

```
(define (square x)
   (* x x))
(square 4)
```

Java method vs. Scheme function

```
public int square(int x){
   return x * x;
}
```

```
(define (square x)
    (* x x))
```

Scheme method with contracts

```
public int square(int x){
   return x * x;
}
```

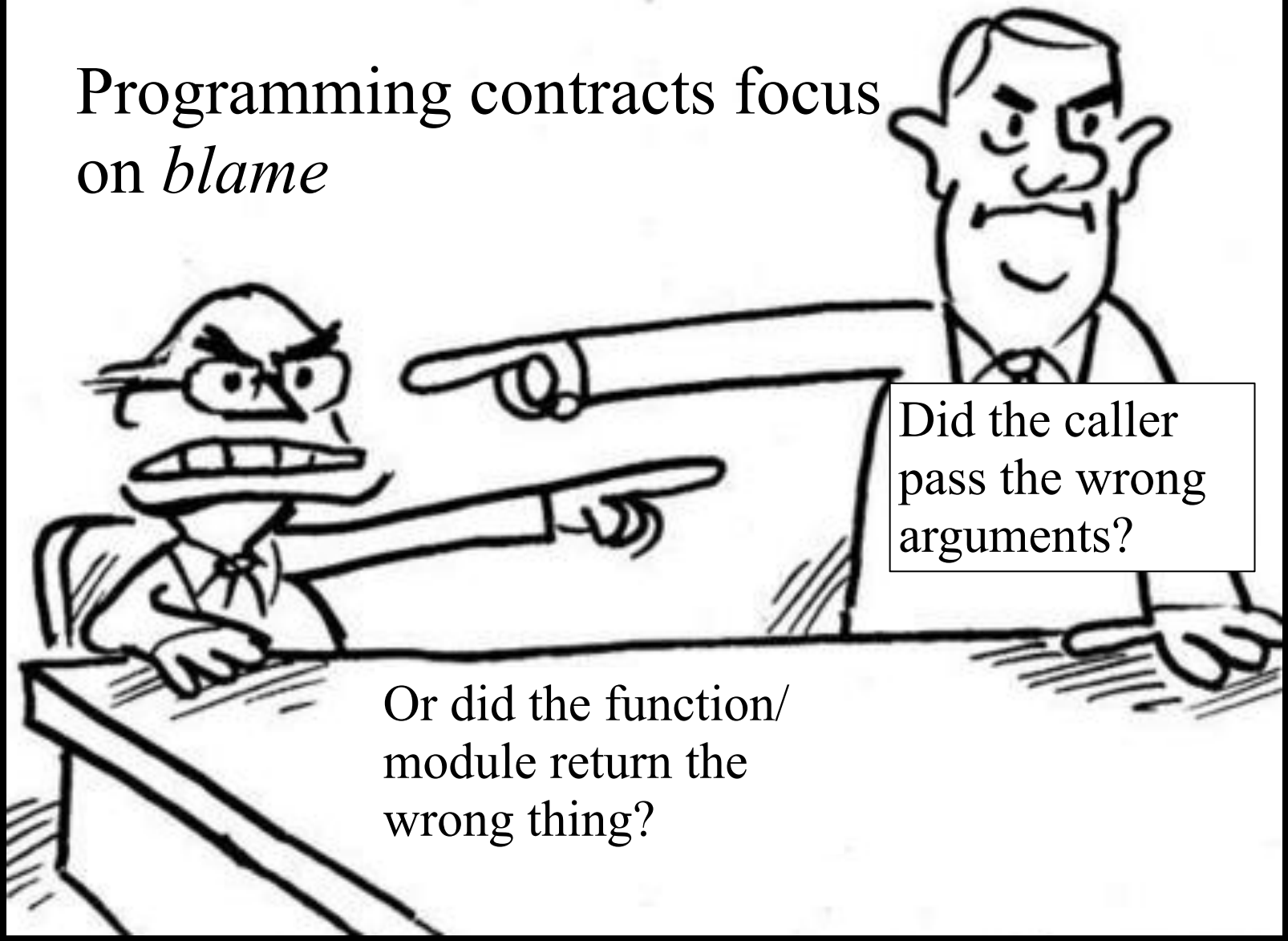
```
(define/contract (square x)
   (-> number? number?)
      (* x x))
```

Programming contracts

Similar to types, but:

- A *runtime* enforcement mechanism
- Can enforce more sophisticated restrictions

Programming contracts focus on *blame*



A programming contract consists of:

- **Preconditions**: requirements for the input
  - if they do not hold, we blame the caller
- **Postconditions**: promises for the output
  - if they do not hold, we blame the library

Sample code with contract

```
#lang racket
(define/contract (square x)
  (-> number? number?)
   (* x x))

(square 4)
(square "four")
```

```
square: contract violation
  expected: number?
  given: "four"
  in: the 1st argument of
      (-> number? number?)
  contract from: (function square)
  blaming: ./simple-contract.rkt
   (assuming the contract is correct)
  at: ./simple-contract.rkt:2.18
```

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```

If Statements

```
(if (< x 0)
    (+ x 1)
    (- x 1))
```

Cond Statements

```
(cond
     [(< x 0) "Negative"]
     [(> x 0) "Positive"]
     [else "Zero"])
```

Scheme does not let you reassign variables

"If you say that a is 5, you can't say it's something else later, because you just said it was 5. What are you, some kind of liar?"

--Miran Lipovača

Recursion

- Base case
  - when to stop
- Recursive step
  - calls function with a **smaller version** of the same problem

Algorithm to count Russian dolls

Recursive step

- Open doll
- *Count number dolls in the inner doll*
- Add 1 to the count

Base case

- No inside dolls
- return 1

An iterative definition of a `count-elems` function

Set count to 0.

For each element:

    Add 1 to the count.

The answer is the count.

BAD!!! Not the way that functional programmers do things.

A *recursive* definition of a `count-elems` function

Base case:

If a list is empty, then the answer is 0.

Recursive step:

Otherwise, the answer is 1 more than the size of the tail of the list.

Recursive Example

```
(define (count-elems lst)
  (cond [(= 0 (length lst)) 0]
        [else (+ 1
                 (count-elems (cdr lst))
)]))

(count-elems '())
(count-elems '(1 2 3 4))
```

Recursive Example


```
(define (count-elems lst)
  (cond [(= 0 (length lst)) 0]
        [else (+ 1
                 (count-elems (cdr lst)))
)]))

(count-elems '())
(count-elems '(1 2 3 4))
```

Recursive Example

```
(define (count-elems lst)
  (cond [(= 0 (length lst)) 0]
        [else (+ 1
                 (count-elems (cdr lst))
)]))
(count-elems '())
(count-elems '(1 2 3 4))
```

```
    (count-elems '(1 2 3 4))
=> (+ 1 (count-elems '(2 3 4)))
=> (+ 1 (+ 1 (count-elems '(3 4))))
=> (+ 1 (+ 1 (+ 1 (count-elems '(4)))))
=> (+ 1 (+ 1 (+ 1 (+ 1
                    (count-elems '())))))
=> (+ 1 (+ 1 (+ 1 (+ 1 0))))
=> (+ 1 (+ 1 (+ 1 1)))
=> (+ 1 (+ 1 2))
=> (+ 1 3)
=> 4
```

Mutual recursion

```
(define (is-even? n)
  (if (= n 0)
      #t
      (is-odd? (- n 1))))

(define (is-odd? n)
  (if (= n 0)
      #f
      (is-even? (- n 1))))
```

let* and letrec

```
(let*
    ([is-even?
      (lambda (n)
        (or (zero? n)
            (is-odd? (sub1 n))))]
     [is-odd?
      (lambda (n)
        (and (not (zero? n))
             (is-even? (sub1 n))))])
  (is-odd? 11))
```

```
is-odd?: unbound
identifier in
module in: is-odd?
```

let* and letrec

```
(letrec
    ([is-even?
      (lambda (n)
        (or (zero? n)
            (is-odd? (sub1 n))))]
     [is-odd?
      (lambda (n)
        (and (not (zero? n))
             (is-even? (sub1 n))))])
  (is-odd? 11))
```

```
#t
```

Lab1

Part 1: Implement a `max-num` function. (Don't use the `max` function for this lab).

Part 2: Implement the "fizzbuzz" game.
sample run:

```
> fizzbuzz 15
"1 2 fizz 4 buzz fizz 7 8
fizz buzz 11 fizz 13 14
fizzbuzz"
```

First homework

Java example with large num

```
public class Test {
 public void main(String[] args){
   System.out.println(
      99999999999999999999 * 2);
   }
}
```

```
$ javac Test.java
Test.java:3: error: integer
number too large:
99999999999999999999

System.out.println(999999999
999999999999 * 2);
                   ^
1 error
```

Racket example

```
$ racket
Welcome to Racket v6.0.1.
> (* 2 99999999999999999999)
199999999999999999998
>
```

HW1: implement a BigNum module

HW1 explores how you might support big numbers in Racket if it did *not* support them.

- Use a list of 'blocks' of digits, least significant block first. So 9,073,201 is stored as:

```
'(201 73 9)
```

- Starter code is available at on course website.

Overview of Homework

- Grade school addition
- Big number addition

- Grade school multiplication
- Big number multiplication

Before next class

- Read chapters 3-5 of *Teach Yourself Scheme*.
- If you accidentally see `set!` in chapter 5, pluck out your eyes lest you become impure.
  - Alternately, just never use `set!` (or get a 0 on your homework/exam).