PyScheme - A Scheme in Python

Danny Yoo (dyoo@hkn.eecs.berkeley.edu)

What is Scheme?

- (lisp-like? "scheme") ==> #t
- isLikePython("scheme") ==> True
- Used in quite a few schools as the "intro" language to computer science.

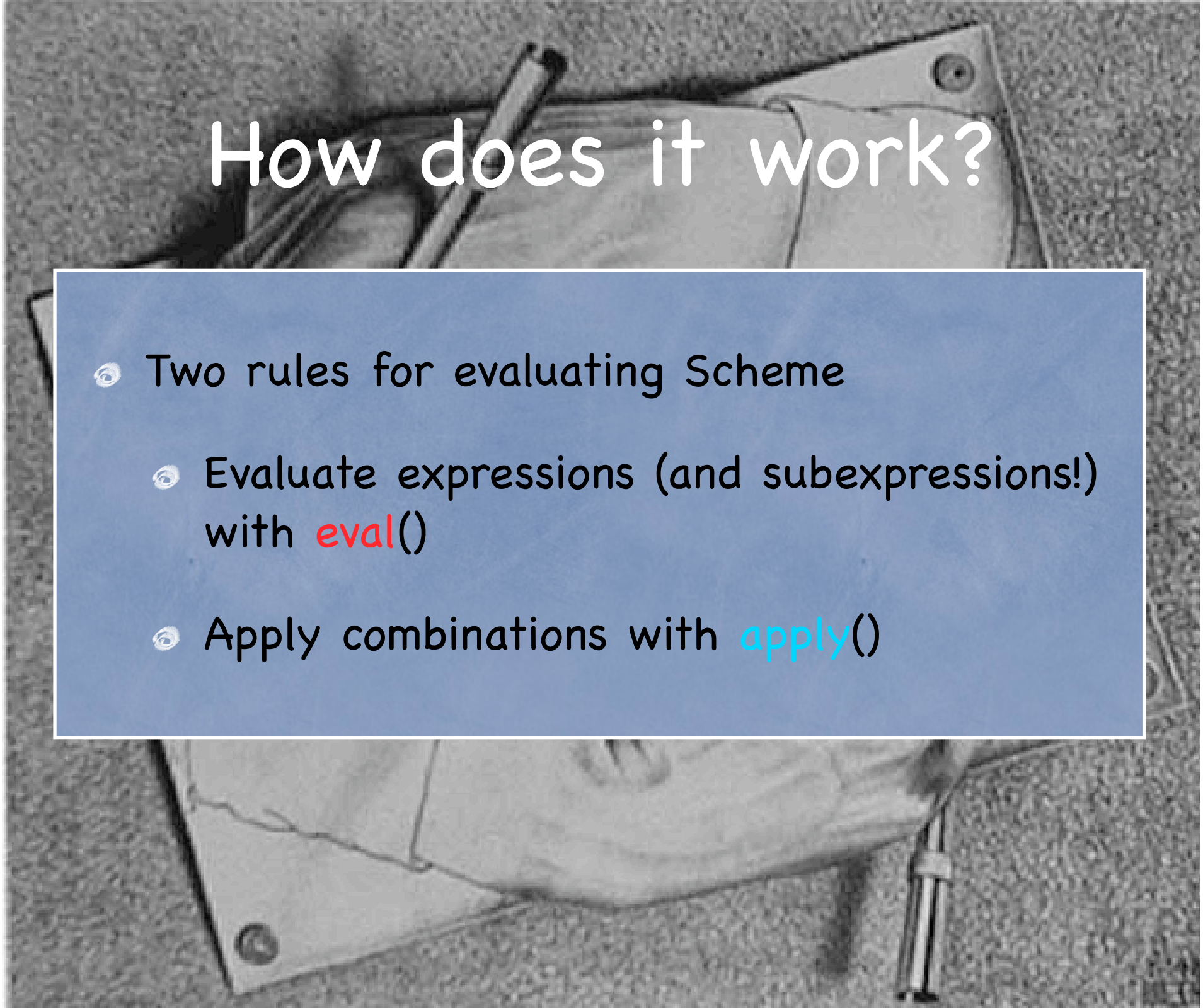
What does Scheme look like?

```
(define (say-hello)
  (display "Hello world!")
  (newline))
```


Stop here for live demo
(and pray to the demo
godesses)

Side-by-side differences

- Small core syntax in Scheme.
- Recursion is used in places where it seems weird at first, but it works.
- Emphasizes expressions and their values.
- Easier to interpret, which leads us to the question...



How does it work?

- Two rules for evaluating Scheme
  - Evaluate expressions (and subexpressions!) with eval()
  - Apply combinations with apply()

eval() in a nutshell

```
def evaluate(expression, environment):
    if expression is self-evaluating:
        return that expression
    elif expression looks like variable:
        look up variable in environment
    elif expression looks like procedure call:
        evaluate subexpressions, and
        apply procedure call on the
        results.
    [... plus a few other "special forms"
     to handle if/cond, and other special
     expressions.]
```

apply() in a nutshell

To apply() a procedure call:

1. bind parameter names and values in a new environment namespace.
2. evaluate the body expression with that environment if the procedure is user defined. Otherwise, get Python's apply() to do the primitive application against the param values.

Why is this easy?

- Evaluation is conceptually simple: it's just recursion in action. [show example with instrumented interpreter]
- We can reuse a lot of Python's objects and runtime support.
- It's been done before. *grin* All of this is reinvention, so I know it's very doable.

Why is this hard?

- Recursion in Python can be ugly.

```
>>> def factorial(x, result = 1):
...     if x == 0:
...         return result
...     return factorial(x - 1, result*x)
>>> factorial(1000)
Traceback (most recent call last):
  File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    File "<stdin>", line 4, in factorial
    ....
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  ....
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
  File "<stdin>", line 4, in factorial
RuntimeError: maximum
recursion depth exceeded
```

Ok, how do we get around this? Well...

We've actually seen trampolines before!

```
def sayHi():
    print "hi!"
    sayHi()
```

```
import Tkinter
root = Tk()
def sayHi():
    print "hi"
    root.after(10, sayHi)

sayHi()
root.mainloop()
```

Boing, boing.

```
def pogo(bouncer):
    while callable(bouncer):
        bouncer = bouncer()
    return bouncer

def bounce(f, *args):
    return lambda: f(*args)
```

Demo time again!

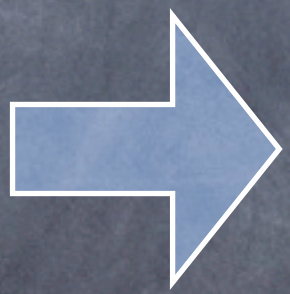
One problem with trampolines...

```
def factorial(n):
    if n == 0: return 1
    return n * factorial(n-1)
```

- Where do we bounce?
- Trampolines don't work unless the function has a certain shape. Technically, trampolines work only if all the nontrivial function calls are "tail" calls.

Rescued by... Continuation Passing Form?

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)
```



```
def identity(x): return x

def c_factorial(n, k = identity):
    if n == 0:
        return k(1)
    else:
        def c(result):
            return k(n * result)
        return c_factorial(n-1, c)
```

So...?

- A CPS'ed program can be trampolined in a fairly mechanical (mindless) way.
- So I CPSed the entire interpreter by hand.
- Isn't this ugly? Yes. Oh well.

Why in the world would you do this?

- It's a great way to learn Python as well as Scheme. There's a saying that programmers learned more about Lisp by learning Python. I'm sorta going the other direction: learning Python by implementing Lisp.
- I'm fascinated by programming languages.

References and good reading

- http://hkn.eecs.berkeley.edu/~dyoo/python/pyscheme
- SICP: Structure and Interpretation of Computer Programs
- Essential of Programming Languages
- Programming Languages - Application and Interpretation
- SICP and PLAI are both online!