What’s happening today?

• We’re learning a new language!
• After you know one language (Python), learning your second (Scheme) is much faster
• Learn by doing – have a sheet of paper ready
• Solutions in the code supplement for this lecture
Scheme Is a Dialect of Lisp

“The greatest single programming language ever designed.”
- Alan Kay, co-inventor of OOP

“The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages.”
- Richard Stallman, founder of the Free Software movement

“Probably my favorite programming language.”
- Eric Tzeng, CS61A Instructor

-Steven Tang, CS61A Instructor

http://imgs.xkcd.com/comics/lisp_cycles.png
Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- **Primitive expressions:** 2, 3.3, true, +, quotient ...
- **Combinations:** (quotient 10 2), (not true), ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

```scheme
> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3 (+ (* 2 4) (+ 3 5)))
(+) (- 10 7)
6))
```

“quotient” names Scheme’s built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn’t matter)
Special Forms

A combination that is not a call expression is a *special form*:

- **If expression:**
  
  ```
  (if <predicate> <consequent> <alternative>)
  ```

- **And and or:**
  
  ```
  (and <e_1> ... <e_n>)  
  (or <e_1> ... <e_n>)
  ```

- **Binding names:**
  
  ```
  (define <name> <expression>)
  ```

- **New procedures:**
  
  ```
  (define (<name> (<formal parameters>)) <body>)
  ```

> (define pi 3.14)
> (* pi 2)
> 6.28

> (define (abs x)
>   (if (< x 0)
>     (- x)
>     x))
> (abs -3)
> 3

The name “pi” is bound to 3.14 in the global frame

A procedure is created and bound to the name “abs”
Try it!

Translate the following Python functions into Scheme:

```python
def one():
    return 1

def two(x, y, z):
    return x + y * z

def three(n):
    if n == 0:
        return 0
    return (n % 10) + 2 * three(n // 10)
```

In Scheme:
- `remainder`  
- `quotient`
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[
(\text{lambda} \ (<\text{formal-parameters}>)) \ <\text{body}>)
\]

Two equivalent expressions:

\[
(\text{define} \ (\text{plus4} \ x) \ (+ \ x \ 4))
\]

\[
(\text{define} \ \text{plus4} \ (\text{lambda} \ (x) \ (+ \ x \ 4)))
\]

An operator can be a combination too:

\[
((\text{lambda} \ (x \ y \ z) \ (+ \ x \ y \ (\text{square} \ z))) \ 1 \ 2 \ 3)
\]

Evaluates to the \textit{add-x-\&-y-\&-z^2} procedure
Syntactic sugar: defining procedures

- In Python, lambda expressions are fundamentally different than def statements:
  - The body of a lambda must be a single expression
  - The value of that expression is always returned
- In Scheme, defining procedures is actually syntactic sugar for a define statement and a lambda expression

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

(define square (lambda (x) (* x x)))

"Define the function square"

"Define a function and give it the name square"
Practice with lambdas

- Complete the definition of \( f \) so that \(((f) \ 3)\) evaluates to 1.

  \[
  \text{(define } f \text{ ???)}
  \]

- Complete the definition of \( g \) so that \(((g \ g) \ g)\) evaluates to 42.

  \[
  \text{(define } g \text{ ???)}
  \]
Pairs

We can implement pairs functionally:

```scheme
(define (pair x y) (lambda (m) (if (= m 0) x y)))
(define (first p) (p 0))
(define (second p) (p 1))
```

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

A pair is represented by a dot between the elements, all in parens

```scheme
> (cons 1 2)
(1 . 2)
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```
Pairs practice

• Suppose \( x \) is the following pair:

\[
\begin{array}{c}
\text{x} \\
\downarrow \\
1 \quad 2
\end{array} \quad \begin{array}{c}
\downarrow \\
3
\end{array} \quad \begin{array}{c}
6 \quad 7
\end{array} \\
\downarrow \\
4 \quad 5
\]

• How would you select 1 from \( x \)?
• 3?
• 7?
• How would you define \( x \) in the first place?
Recursive Lists

A recursive list can be represented as a pair in which the second element is a recursive list or the empty list.

Scheme lists are recursive lists:
- `nil` is the empty list
- A non-empty Scheme list is a pair in which the second element is `nil` or a Scheme list.

Scheme lists are written as space-separated combinations:

```scheme
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil))))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

Not a well-formed list!
Aside: Booleans and Boolean contexts

Boolean constants
• In Python, we had True and False as our Boolean constants
• In Scheme, we use #t and #f instead

Boolean contexts
• In Python, most objects were treated like True, but many different objects were treated as False (0, "", [], etc.)
• In Scheme, *everything* is treated like #t, with the exception of #f itself.

```scheme
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```
Recursive list practice

• Write a Scheme function append that takes two lists and returns a single list that contains the values from the first list and the second list, in order:

```scheme
STk> (append (list 1 2 3) (list 4 5 6))
(1 2 3 4 5 6)
```
Symbolic Programming

Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)

Quotation can also be applied to combinations to form lists

> (car '(a b c))
a
> (cdr '(a b c))
(b c)
Scheme Lists and Quotation

Dots can be used in a quoted list to specify the second element of the final pair

\[
> \ (\text{cdr} \ (\text{cdr} \ '(1 \ 2 \ . \ 3)))
3
\]

However, dots appear in the output only of ill-formed lists

\[
> \ '((1 \ 2 \ . \ 3))
(1 \ 2 \ . \ 3)
\]
\[
> \ '((1 \ 2 \ . \ (3 \ 4))
(1 \ 2 \ 3 \ 4)
\]
\[
> \ '((1 \ 2 \ 3 \ . \ \text{nil})
(1 \ 2 \ 3)
\]

What is the printed result of evaluating this expression?

\[
> \ (\text{cdr} \ '((1 \ 2) . (3 \ 4 \ . \ (5))))
(3 \ 4 \ 5)
\]
The Let Special Form
Let expressions introduce a new frame, with the given bindings

\[
\text{(let (((name) (exp)) \ldots) (body))}
\]

(\text{define (filter fn s)}
  (if (null? s)
      s
      (let (((first (car s))
          (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest)))
  > (filter even? '(1 2 3 4 5 6 7))
  (2 4 6)
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

(define (filter-comp comp pivot s)
  (filter (lambda (x) (comp x pivot)) s))

(define (quick-sort s)
  (if (<= (length s) 1)
    s
    (let ((pivot (car s)))
      (append (quick-sort (filter-comp < pivot s))
              (filter-comp = pivot s)
              (quick-sort (filter-comp > pivot s)))))))
Turtle graphics

- STk has built in support for basic 2D graphics!
- Turtle sits on the canvas
- As the turtle “walks” around the canvas, it leaves a trail
- Images are drawn by issuing commands to the turtle

```
(define (triangle)
  (forward 100)
  (right 120)
  (forward 100)
  (right 120)
  (forward 100)
  (right 120))
```

- Did we need the last call to `right`? Why?
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin } <\text{exp}_1> <\text{exp}_2> ... <\text{exp}_n>)\]

\[(\text{define } (\text{repeat } k \ fn))\]
\[
  (\text{if } (> k 0)
   (\text{begin } (fn) (\text{repeat } (- k 1) \ fn))
   \text{'done}))\]

\[(\text{define } (\text{tri } fn))\]
\[
  (\text{repeat } 3 (\text{lambda } () (fn) (\text{lt } 120)))\]

\[(\text{define } (\text{sier } d \ k))\]
\[
  (\text{tri } (\text{lambda } () (\text{if } (= k 1) (fd d) (\text{leg } d k))))\]

\[(\text{define } (\text{leg } d \ k))\]
\[
  (\text{sier } (/ d 2) (- k 1)) (\text{penup}) (fd d) (\text{pendown})\]