Scheme
The Scheme Programming Language
Expressions

An expression is evaluated in an environment (that gives symbols meaning) to produce a value.

Local frame: "the course instructor" has a specific meaning for a particular course.
Global frame: "multiply" is an operation that everyone knows about.
Local before Global: in a particular context, "multiply" can mean something different.

Scheme programs consist of expressions, which can be:

- Self-evaluating expressions: 2  3.3  true
- Symbols:             +  -  quotient  not
- Call expressions:       (quotient 10 2)  (f x)
- Special forms:        (if a b c)   (let ((x 2)) (+ x 1))

(Demo)
Defining Functions/Procedures

No `return` in Scheme; the value of a call expression is the value of the last body expression of the procedure.

```python
>>> def sum_squares(x, y):
    ...    return x * x + y * y
```

Instead of multiple return statements, Scheme uses nested conditional expressions.

```python
>>> def fib(n):
    ...    if n == 0 or n == 1:
    ...        return n
    ...    else:
    ...        return fib(n - 2) + fib(n - 1)
```

```scheme
scm> (define (sum-squares x y)
    (+ (* x x) (* y y)))
```

```scheme
scm> (define (fib n)
    (if (or (= n 0) (= n 1))
        n
        (+ (fib (- n 2)) (fib (- n 1)))))
```
Python vs Scheme: Call Expressions

A call expression in Scheme has the parentheses on the outside.

```python
>>> def sum_squares(x, y):
...     return x * x + y * y
... sum_squares(3, 4)
25
```

```scheme
scm> (define (sum-squares x y)
        (+ (* x x) (* y y)))

sum-squares
scm> (sum-squares 3 4)
25
```

Some Scheme combinations are not call expressions because they are special forms.

```python
>>> def f(x):
...     print(x)
...     return False
... f(3) and f(4)
3 False
```

```scheme
scm> (define (f x) (print x) False)
f
scm> (and (f 3) (f 4))
3 #f
```
Python vs Scheme: Iteration

Scheme has no for/while statements, so recursion is required to iterate.

```python
def sum_first_n(n):
    return sum(range(1, n + 1))
```

```scheme
def (sum-first-n n)
    (define (f k total)
        (if (> k n)
            total
            (f (+ k 1) (+ total k)))
    (f 1 0))
sum-first-n
```

```python
def sum_first_n(n):
    total = 0
    for k in range(1, n + 1):
        total += k
    return total
```

```scheme
(sum-first-n 5)
```

```python
def sum_first_n(n):
    k = 1
    total = 0
    while k <= n:
        k, total = k + 1, total + k
    return total
```

```scheme
(sum-first-n 5)
```

```python
sum_first_n(5)
```

15
Writing Scheme
Example: A-Plus-Abs-B

a-plus-abs-b takes numbers a and b and returns $a + \text{abs}(b)$ without calling abs.

```python
def a_plus_abs_b(a, b):
    """Return $a + \text{abs}(b)$, but without calling abs."
    if b < 0:
        f = sub
    else:
        f = add
    return f(a, b)
```

(define (a-plus-abs-b a b)
  (if (< b 0) - +) a b))
Lambda Expressions
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 call expression too:

\[(\text{call} \ (\text{lambda} \ (x \ y \ z) \ (+ \ x \ y \ (\text{square} \ z))) \ 1 \ 2 \ 3) \rightarrow 12\]

Evaluates to the \(x+y+z^2\) procedure
What Would Scheme Do?

```scheme
(((lambda (g y) (g (g y))) (lambda (x) (+ x 1))) 3)

(define (f g)
  (lambda (y) (g (g y))))

(((f (lambda (x) (* x x))) 3)
```
More Special Forms
**Cond & Begin**

The cond special form that behaves like if–elif–else statements in Python

```lisp
(if x > 10:
  print('big')
(elif x > 5:
  print('medium')
(else:
  print('small'))
```

The begin special form combines multiple expressions into one expression

```lisp
(if x > 10:
  (begin
    (print 'big)
    (print 'guy)))
(else:
  (begin
    (print 'small)
    (print 'fry))))
```
Let Expressions

The let special form binds symbols to values temporarily; just for one expression.

```plaintext
a = 3
b = 2 + 2
c = math.sqrt(a * a + b * b)

(define c (let ((a 3) (b (+ 2 2)) (sqrt (+ (* a a) (* b b)))))

a and b are still bound down here
a and b are not bound down here
```