Interpreters
Announcements
Interpreting Scheme
The Structure of an Interpreter

**Eval**

**Base cases:**
- Primitive values (numbers)
- Look up values bound to symbols

**Recursive calls:**
- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

**Apply**

**Base cases:**
- Built-in primitive procedures

**Recursive calls:**
- Eval(body) of user-defined procedures

- Requires an environment for symbol lookup
- Creates a new environment each time a user-defined procedure is applied
Project 4
Pairs in Project 4: Scheme

https://cs61a.org/proj/scheme/

**Tokenization/Parsing:** Converts text into Python representation of Scheme expressions:

- Numbers are represented as numbers
- Symbols are represented as strings
- Lists are represented as instances of the Pair class (Demo)

**Evaluation:** Converts Scheme expressions to values while executing side effects:

- `scheme_eval(expr, env)` returns the value of an expression in an environment
- `scheme_apply(procedure, args)` applies a procedure to its arguments
- The Python function `scheme_apply` returns the return value of the procedure it applies (Demo)
Discussion Question: The Symbol of a Define Expression

Return the symbol of a define expression. There are two formats for define expressions:
(define x (+ 2 3)) or (define (f x) (+ x 3))

```python
def symbol(exp):
    
    """Given a define expression exp, return the symbol defined."
    >>> def_x = read_line("(define x (+ 2 3))")
    >>> def_f = read_line("(define (f x) (+ x 3))")
    >>> symbol(def_x)
    'x'
    >>> symbol(def_f)
    'f'

    """
    assert exp.first == 'define' and exp.rest is not nil and exp.rest.rest is not nil
    signature = ___________
    if scheme_symbolp(signature):
        return signature
    else:
        return ___________
```
Special Forms
Scheme Evaluation

The scheme_eval function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations

\[
\begin{align*}
\text{(if } & \text{ <predicate> } \text{ <consequent> } \text{ <alternative>)} \\
\text{(lambda } & \text{ (<formal-parameters>) } \text{ <body>)} \\
\text{(define } & \text{ <name> } \text{ <expression>)} \\
\text{(operator } & \text{ <operand 0> } \text{ ... } \text{ <operand k>)}
\end{align*}
\]

Special forms are identified by the first list element

Any combination that is not a known special form is a call expression

\[
\begin{align*}
\text{(define (demo s) (if (null? s) '3) (cons (car s) (demo (cdr s))) ) )} \\
\text{(demo (list 1 2))}
\end{align*}
\]
Lambda Expressions
Lambda Expressions

Lambda expressions evaluate to user-defined procedures

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

$$(\text{lambda} \ (x) \ (\ast \ x \ x))$$

```python
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals  # A scheme list of symbols
        self.body = body        # A scheme list of expressions
        self.env = env          # A Frame instance
```
Frames and Environments

A frame represents an environment by having a parent frame.

Frames are Python instances with methods `lookup` and `define`.

In Project 4, Frames do not hold return values.

```
g: Global frame
  y | 3
  z | 5

f1: [parent=g]
  x | 2
  z | 4
```

(Demo)
Lab 10
Lab 10: Extending the Calculator

Calculator is a subset of Scheme that doesn't have environments or special forms.

Lab 10 will have you:

• Fill in the eval function of a Calculator interpreter

• Add another procedure (floor division)

• Add a special form (and)

• Add a global frame to store bindings from symbols to values

(Demo)