

# LAB 11

---

## Scheme


Mingxiao Wei

[mingxiaowei@berkeley.edu](mailto:mingxiaowei@berkeley.edu)

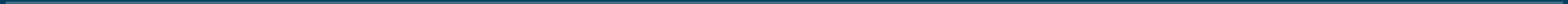
Apr 10, 2023

# LOGISTICS

---

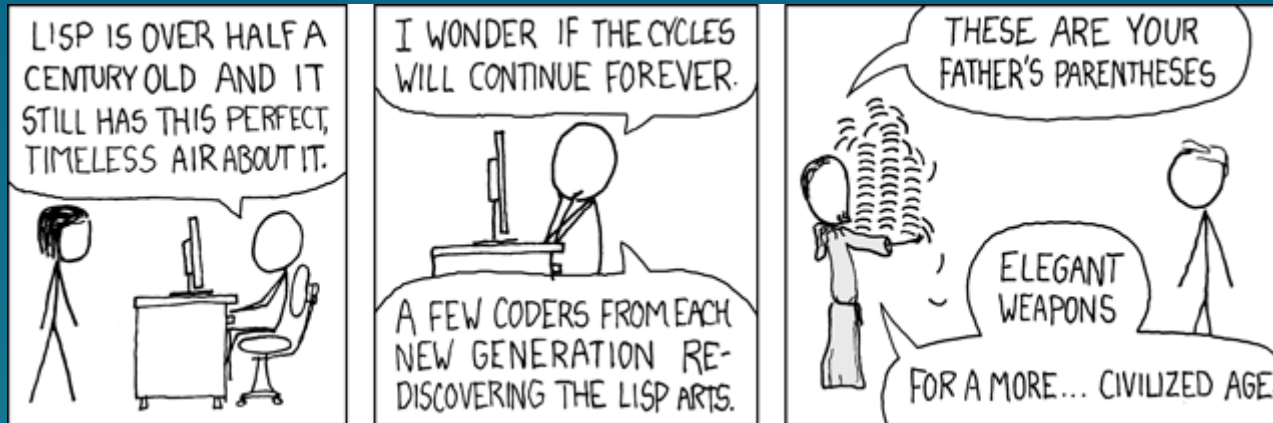
- Congrats on finishing the midterm!
- Welcome to the world of (scheme) 
- Lab 11 due tomorrow 4/12
- Homework 08 due Thu 4/13

# SCHEME



# SCHEME - INTRO

- A dialect of Lisp (a very old language)
- Uses **prefix** notations
  - `(func op1 op2)`
  - (many) nested parenthesis as a result



- Features **tail-call** optimizations - may or may not be covered later
- **EVERYTHING IN SCHEME EVALUATES TO A VALUE**
  - In contrast, statements in python do not evaluate to anything

# SCHEME - LOGISTICS

---

For assignment with Scheme questions, in the assignment directory:

- start a Scheme interpreter - `python3 scheme`
- run a program interactively - `python3 scheme -i <file.scm>`
- exit the interpreter - `(exit)`
- start the editor - `python3 editor`
  - This is how you can edit the `.scm` file for an assignment
  - save your work (locally) - `Ctrl/Cmd + s`
  - quit the editor - `Ctrl + c`
  - test without quitting the editor - run the ok commands in a separate terminal window
- [Scheme Built-In Procedure Reference](#)
- [Scheme Specification](#)

PS: You can also edit the `.scm` file in your own editor, but our version has visualizations that makes parenthesis matching much easier!

# SCHEME - PRIMITIVE EXPRESSIONS

---

- Primitive Expressions:
  - self-evaluating
  - Includes numbers, booleans, symbols

```
scm> 6
```

```
6
```

```
scm> 10.0
```

```
10.0
```

# SCHEME - PRIMITIVE EXPRESSIONS

---

- Booleans
  - `#t` in Scheme  $\leftrightarrow$  `True` in Python
  - `#f` in Scheme  $\leftrightarrow$  `False` in Python
  - `#f` is THE ONLY FALSY VALUE in Scheme! (0 is truthy!)

```
scm> #t
#t
scm> #f
#f
```

# SCHEME - PRIMITIVE EXPRESSIONS

---

- Symbols
  - a type of value in scheme
  - act like variable names in Python (but not exactly the same)
  - a symbol can evaluate to a value; an expression can also evaluate to a symbol

```
scm> quotient
#[quotient] ; representation of built-in procedures
scm> 'quotient ; Scheme uses single quotation mark
quotient ; the string above evaluates to a symbol
scm> 'hello-world!
hello-world! ; also a symbol value
```

`;` in Scheme denotes comment - just like `#` in Python



# SCHEME - CALL EXPRESSIONS

---

Anatomy: `(func op1 op2 ...)`

- Operator is WITHIN the parenthesis, and comes first
- Operator/operands are separated by whitespace, NOT comma
- Same evaluation rule as in Python:
  1. Evaluate the operator, which should evaluate to a procedure\*
  2. Evaluate the operands from left to right
  3. Apply the procedure to the operands

\* In Scheme, functions are called procedures

# SCHEME - BUILT-IN PROCEDURES

Scheme	Python
<code>(/ a b)</code>	<code>a / b</code>
<code>(quotient a b)</code>	<code>a // b</code>
<code>(modulo a b)</code>	<code>a % b</code>
<code>(= a b)</code>	<code>a == b</code>
<code>(not (= a b))</code>	<code>a != b</code>

# SCHEME - CALL EXPRESSIONS

```
scm> (+ 1 (* 2 3) (- 5 4)) ; 1 + (2 * 3) + (5 - 4)  
8
```

```
scm> (- (/ 10 4) 1) ; (10 / 4) - 1  
1.5
```

```
scm> (modulo 10 4) ; modulo -> %  
2
```

```
scm> (odd? (quotient 45 2)) ; quotient -> //  
#f
```

```
scm> (not (= 1 2)) ; operands to = must be numbers  
#t
```

Exercise: translate `6 * 3 - 2 >= 0` to Scheme

# SCHEME - QUOTES

---

- use a one single quotation mark - `'<expression>`
  - only applies to the expression right after
- Equivalent form: `(quote <expression>)`
- Evaluate to the `<expression>` exactly as it is

```
scm> 'hello-world ; evaluates to a symbol value
hello-world
```

```
scm> ' (+ 1 2)
(+ 1 2)
```

# SCHEME - SPECIAL FORMS

---

- Do not follow the rules for call expressions (e.g., short-circuiting)
- [Scheme Specification](#) - complete list of special forms
- Includes `and`, `or`, `if`, `cond`, etc.

```
scm> (and 0 1 2 3) ; 0 in Scheme is truthy!
```

```
3
```

```
scm> (or 0 1 2 3)
```

```
0
```

```
scm> (and (> 1 6) (/ 1 0)) ; short-circuiting applies
```

```
#f
```

```
scm> (or (< 1 6) (/ 1 0))
```

```
#t
```

# SCHEME - CONTROL STRUCTURES

```
(if <predicate> <if-true> [if-false]) *
```

- Evaluation rules
  1. Evaluate `<predicate>`
  2. If it evaluates to a truthy value, evaluate and return `<if-true>`. Otherwise, evaluate and return `[if-false]`
  3. `[if-false]` is optional. If not provided and `<predicate>` is falsy, returns `undefined` - Scheme's version of `None` (not displayed in the interpreter unless printed)
- Only one of `<if-true>` and `[if-false]` is evaluated
  - The entire special form evaluates to either `<if-true>` or `[if-false]`
- No `elif` - if more than 2 branches, use nested `if`'s or `cond`

\* In our [Scheme Specification](#), `<>` is used to denote required components while `[ ]` is used to denote optional components

# SCHEME - CONTROL STRUCTURES

Scheme	Python
<pre>(if (&gt; x 3)     1     2)</pre>	<pre>if x &gt; 3:     return 1 else:     return 2</pre>
<pre>(if (&lt; x 0)     'negative     (if (= x 0)         'zero         'positive)     )</pre>	<pre>if x &lt; 0:     return 'negative' else:     if x == 0:         return 'zero'     else:         return 'positive'</pre>

Note: Indentation / line break does NOT matter in Scheme

# SCHEME - CONTROL STRUCTURES

Scheme <code>if</code>	Python <code>if</code>
A special form expression that evaluates to a value	Some statement that directs the flow of the program
Expects just a single expression for each of the true result and the false result	Each suite can contain multiple lines of code
No <code>elif</code>	Has <code>elif</code>



# SCHEME - CONTROL STRUCTURES

```
(cond
  (<p1> <e1>)
  ...
  (<pn> <en>)
  [ (else <else-expression>) ] ) ; else is optional
```

- Similar to a multi-clause if/elif/else conditional
- Takes in an arbitrary number of arguments - clauses
  - Clause: (<p> <e>)
- Evaluation rules:
  1. Evaluate the predicates <p1>, <p2>, ..., <pn> in order until a truth-y value
  2. For the first truthy predicate, evaluate and return the corresponding expression in the clause
  3. If none are truth-y and there is an else clause, evaluate and return <else-expression>; otherwise return undefined

# SCHEME - CONTROL STRUCTURES

---

```
(cond
  (<p1> <e1>)
  ...
  (<pn> <en>)
  [ (else <else-expression>) ] ) ; else is optional
```

- special form - does not evaluate every operands
  - short circuits upon reaching the first truthy predicate
- Only one clause has its `<e>` evaluated (and returned)
  - The whole `cond` special form evaluates to the value of this `<e>`
- Order of clauses matters
  - only move to the next clause if all previous predicates are falsy

# SCHEME - CONTROL STRUCTURES

Scheme	Python
<pre>(cond   ((&gt; x 3) 1)   (else 2) )</pre>	<pre>if x &lt; 3:     return 1 else:     return 2</pre>
<pre>(cond   ((&gt; x 0) 'positive)   ((&lt; x 0) 'negative)   (else 'zero) )</pre>	<pre>if x &gt; 0:     return 'positive' elif x &lt; 0:     return 'negative' else:     return 'zero'</pre>

Note: Indentation / line break does NOT matter in Scheme

# SCHEME - DEFINE VARIABLES

---

```
(define <name> <expression>)
```

- Evaluation rules
  1. Evaluate the `<expression>`
  2. Bind its value to the `<name>` in the current frame
  3. Return `<name>` as a symbol
- Evaluates to `<name>` (a symbol value)

```
scm> (define x (+ 6 1))
```

```
x
```

```
scm> x
```

```
7
```

```
scm> (+ x 2)
```

```
9
```

# SCHEME - DEFINE FUNCTIONS

---

```
(define (<func-name> <param1> <param2> ... ) <body>)
```

- Evaluation rules
  1. Create a lambda procedure with the given parameters and `<body>`
  2. Bind its procedure to the `<func-name>` in the current frame
  3. Return `<func-name>` as a symbol
- Evaluates to `<name>` (a symbol value)
- `<body>` can have multiple expressions
  - all expressions are evaluated from left to right, and the value of the last expression is returned
- Special form - function body not evaluated until the function is called

# SCHEME - DEFINE FUNCTIONS

---

```
(define (<func-name> <param1> <param2> ... ) <body>)
```

```
scm> (define (foo x y) (+ x y))
```

```
foo
```

```
scm> (foo 2 3)
```

```
5
```

```
scm> (define (bar x y) (define z (* x y)) (+ x y z))
```

```
bar
```

```
scm> (bar 2 3)
```

```
11
```

# SCHEME - LAMBDA FUNCTIONS

```
(lambda (<param1> <param2> ... ) <body>)
```

- Create and evaluate to a procedure, without altering the current environment unless we bind it to a variable.
- All Scheme procedures are lambda procedures!
- <body> can have multiple expressions
  - all expressions are evaluated from left to right, and the value of the last expression is returned

```
scm> (define foo (lambda (x y) (+ x y)))
```

```
foo
```

```
scm> (define (foo x y) (+ x y)) ; these two are equivalent
```

```
foo
```

```
scm> (foo 2 3)
```

```
5
```

```
scm> (lambda (x y) (+ x y))
```

```
(lambda (x y) (+ x y))
```

# NOW IT'S YOUR TIME 🤠

---

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: [go.cs61a.org/mingxiao-index](https://go.cs61a.org/mingxiao-index)
- Leave any anonymous feedback here: [go.cs61a.org/mingxiao-anon](https://go.cs61a.org/mingxiao-anon)



AND REMEMBER TO GET  
CHECKED OFF! 🧺

---

[go.cs61a.org/mingxiao-att](https://go.cs61a.org/mingxiao-att)

The secret phrase is ...  
(NOT 3 dots! I'll announce it 🙊)