DISCUSSION 11

Scheme, Scheme Lists

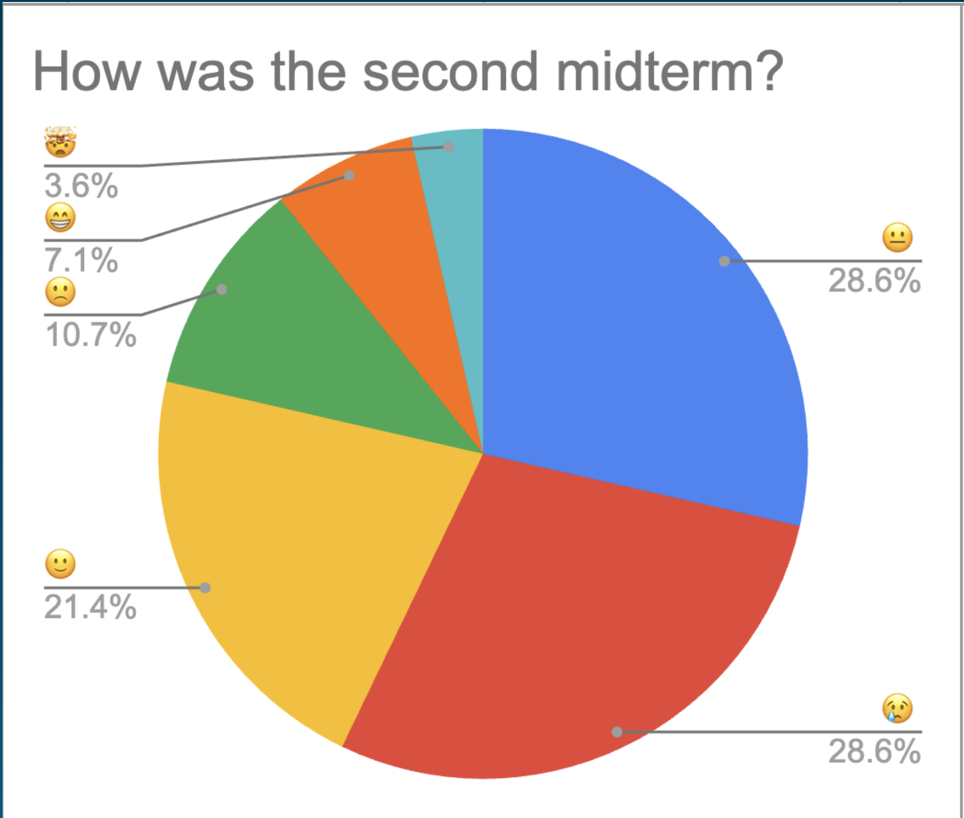
Mingxiao Wei
mingxiaowei@berkeley.edu

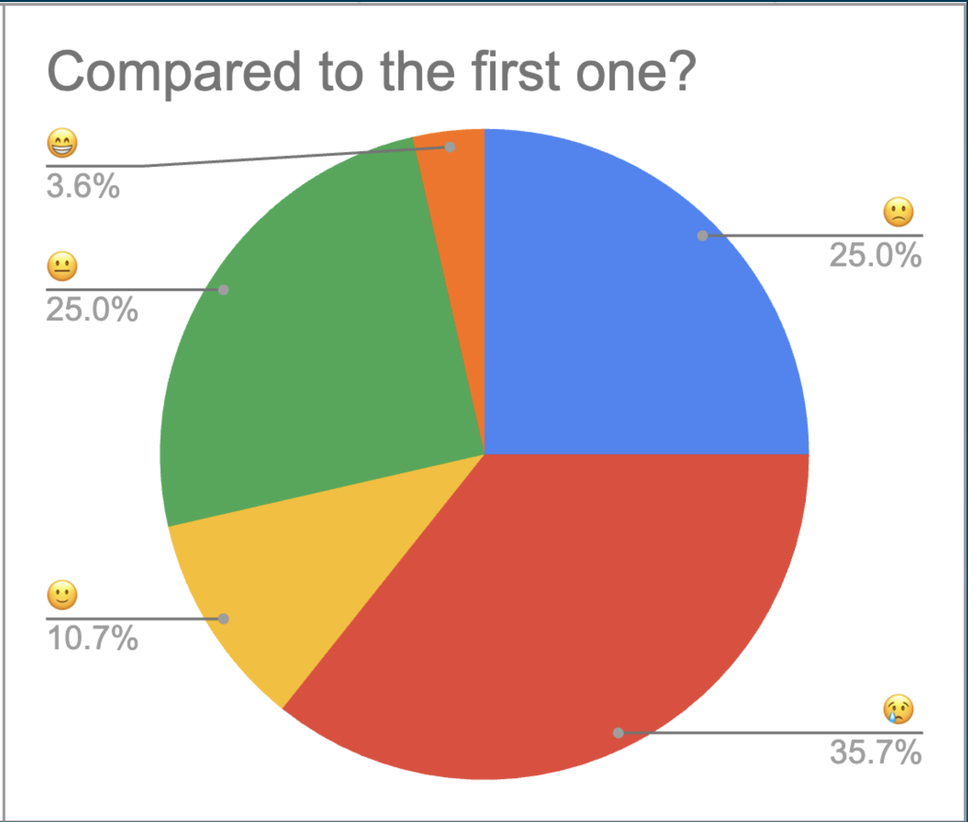
Apr 13, 2022

LOGISTICS 🏡

- Welcome to the world of `(scheme)` 🎃
- Homework 08 due today 4/13
- The Scheme project is coming up! 👀
    - Now is a good time to reach out to a project partner, if you'd like to collaborate!
    - If you like interpreter, go take CS 164 :o
- Reminder about Homework 7 recovery

FROM LAST TIME... 👀





SCHEME 🦦

SCHEME - PRIMITIVE EXPRESSIONS

- Booleans
  - `#t` in Scheme ↔ `True` in Python
  - `#f` in Scheme ↔ `False` in Python
  - `#f` is THE ONLY FALSY VALUE in Scheme!
    - 0 is truthy
    - `undefined` (Scheme's version of `None`) is also truthy

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

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
`(/ a b)`	`a / b`
`(quotient a b)`	`a // b`
`(modulo a b)`	`a % b`
`(= a b)`	`a == b`
`(not (= a b))`	`a != b`

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> (quote hello-world) ; same as above
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

<table>
<tr><th>Scheme</th><th>Python</th></tr>
<tr><td>

```scheme
(if (> x 3)
    1
    2)
```

</td><td>

```python
if x > 3:
    return 1
else:
    return 2
```

</td></tr>
<tr><td>

```scheme
(if (< x 0)
    'negative
    (if (= x 0)
        'zero
        'positive
    )
)
```

</td><td>

```python
if x < 0:
    return 'negative'
else:
    if x == 0:
        return 'zero'
    else:
        return 'positive'
```

</td></tr>
</table>

Note: Indentation / line break does NOT matter in Scheme

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

<table>
<tr><th>Scheme</th><th>Python</th></tr>
<tr><td>

```
(cond
    ((> x 3) 1)
    (else 2)
)
```

</td><td>

```
if x < 3:
    return 1
else:
    return 2
```

</td></tr>
<tr><td>

```
(cond
    ((> x 0) 'positive)
    ((< x 0) 'negative)
    (else 'zero)
)
```

</td><td>

```
if x > 0:
    return 'positive'
elif x < 0:
    return 'negative'
else:
    return 'zero'
```

</td></tr>
</table>

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))
```

SCHEME - LET EXPRESSIONS

```
(let ([binding_1] ... [binding_n]) <body> ...)
```

- Each `[binding]` has the form `(<name> <expr>)`
- Evaluation rule
    1. create a new child frame whose parent is the current frame
    2. For each `binding`, bind each `name` to its corresponding evaluated `expr`
    3. In this new frame, the `body` expressions are evaluated in order, returning the result of evaluating the last expression

SCHEME - LET EXPRESSIONS

```
(let ([binding_1] ... [binding_n]) <body> ...)
```

```
scm> (define x 6)
x
scm> (define z 7)
z
scm> (let (
              (x 5) (y 10)
           )
          (print x)
          (print z)
          (- x y)
          (+ x y)
     )
5
7
15
```

SCHEME - BEGIN EXPRESSIONS

```
(begin <expr_1> ... <expr_n>)
```

- Evaluate all expressions in order in the current frame
- Return the value of the last expression

```
scm> (define x 6)
x
scm> (define y 7)
y
scm> (begin
            (print 'hello)
            (define z 8)
            (- x y z)
            (+ x y z)
     )
hello
21
```

SCHEME - BEGIN EXPRESSIONS

```
(begin <expr_1> ... <expr_n>)
```

- Useful when only one expression is expected

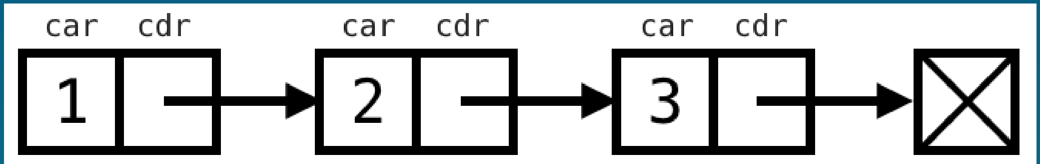
```
scm> (if (begin (print 0) 0)
         (begin (print 1) (+ 2 3))
         (begin (print 4) (+ 5 6))
     )
0
1
5
```

WORKSHEET WWSD, Q1

SCHEME LISTS 🦥

SCHEME LISTS - INTRO

- All Scheme Lists are linked lists! 😮😮
- 3 ways to construct a linked list:




```
scm> (cons 1 (cons 2 (cons 3 nil))) ; nil -> Link.empty
(1 2 3)
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
```

- `(car lst)` - returns **the first element** from the `lst`, analogous to `link.first`
- `(cdr lst)` - returns the rest of the `lst` as **another Scheme list**, analogous to `link.rest`

SCHEME LISTS - INTRO

```
scm> (define lst (cons 1 (cons 2 (cons 3 nil))))
lst
scm> lst
(1 2 3)
scm> (car lst)
1
scm> (cdr lst)
(2 3)
scm> (car (cdr (cdr a)))
3
```

SCHEME LISTS - CONSTRUCTOR

`(cons <first> <rest>)`

- Similar to a linked list constructor
- `<first>`
  - first element of the list
- `<rest>`
  - must be another Scheme list, or `nil` if empty
  - required
- Useful for recursion problems

```
scm> (define a (cons 1 (cons 'a nil)))
a
scm> a
(1 a)
scm> (cons 6 a)
(6 1 a)
```

SCHEME LISTS - CONSTRUCTOR

```
(list <ele1> <ele2> ...)
```

- Takes in an arbitrary number of elements in the list
- Evaluate each element (could be an expression) from left to right, and return them as a Scheme list
- Useful when we know exactly what elements are in the list

```
scm> (define a (+ 6 1))
a
scm> a
7
scm> (list (- a 1) a (+ a 1))
(6 7 8)
```

SCHEME LISTS - CONSTRUCTOR

`'(...)` or `(quote ...)`

- Construct the exact list given, without any evaluation

```
scm> (define a (+ 6 1))
a
scm> (list 6 a 8)
(6 7 8)
scm> '(6 a 8) ; equivalently, (quote (6 a 8))
(6 a 8)
scm> '(cons 1 2)
(cons 1 2)
scm> '(1 (2 3 4))
(1 (2 3 4))
```

SCHEME LISTS - BUILT-IN PROCEDURES

- `(null? lst)` - checks if `lst` is empty
- `(append lst1 lst2)` - concatenates two lists together and return them as a new list
- `(length lst)` - return the length of `lst`

```
scm> (null? nil)
#t
scm> (append '(c s) '(6 1 a))
(c s 6 1 a)
scm> (length '(1 (2 3) 4))
3
```

CHECKING EQUALITY

- `(= <a> <b>)`
  - Both `<a>` and `<b>` must be numbers
- `(eq? <a> <b>)`
  - Similar to `is` in Python
  - Returns `#t` if `<a>` and `<b>` are equivalent primitive values, or if they refer to the same list
- `(equal? <a> <b>)`
  - For pairs (lists) - returns `#t` if they contain the same elements, similar to `lst1 == lst2` in Python
  - For primitive values - same as `eq?`

CHECKING FOR EQUALITY

```
scm> (= (+ 2 3) (+ 1 4)) ; must be two numbers
#t
scm> (eq? (list 1 2) (list 1 2)) ; two different lists
#f
scm> (equal? (list 1 2) (list 1 2)) ; lists with the same elements
#t
scm> (define a (list 3 4))
a
scm> (define b a) ; a and b are the same list
b
scm> (eq? a b)
#t
```

PRO TIPS

- Parenthesis MATTERS A LOT in Scheme - they are used to denoted expressions in addition to grouping
  - For example, we can have `((1) + (2))` in Python, but not `(+ (1) (2))` in Scheme - correct version is `(+ 1 2)`
- NO ITERATION, ONLY RECURSION 🤔
- Make sure every call expression is wrapped in a parenthesis
- When using `cond`, make sure each clause is in its own parenthesis
- No return - can't terminate a function early. The return value has to be the value of the last expression

WORKSHEET Q2-5

ATTENDANCE! 🤠

---

go.cs61a.org/mingxiao-att

- The attendance form and slides are both linked on our section website!
- Please leave any anonymous feedback here go.cs61a.org/mingxiao-anon
- Please do remember to fill out the form by midnight today!!