COMPUTER SCIENCE MENTORS 61A

October 31–November 4, 2022

Recommended Timeline

- Scheme mini-lecture: 15 min
- What Would Scheme Do? (Basics): 7 min
- Apply Multiple: 10 min
- Hailstone: 13 min
- Scheme lists mini-lecture: 10 min
- What Would Scheme Do? (Lists): 7 min
- Is Prefix?: 10 min
- Argmax: 15 min

1 Scheme

The text explanations on this worksheet are very lengthy. **Do not go over all of this material in mini-lecture.** That would be an incredible waste of time. Instead, ask your students what they would like you to go over and focus on that material in mini-lecture. They will have already been exposed to much of this material in lecture and discussion, so they shouldn't need to be completely retaught it again. **Teaching Tips**

1. To ease in Scheme, it can help to start by comparing and contrasting with Python
   (a) Have students write a basic function in Python (like an iterative countdown), then replicate it in Scheme
   (b) Have students list language features of Python (variable assignments, conditional statements, logic operators, etc.), and explain how Scheme implements those features

Created by Gabe Classon, Aditya Balasubramanian, Alyssa Smith, Esther Shen, Maya Romero, and Manas Khatore

   (c) Make sure to give a disclaimer that while high level features may be analogous, the internals are different!

2. Scheme features break into three broad categories: Primitives, Call Expressions, and Special Forms (the latter two are called Compound Expressions)
   (a) Primitives evaluate to themselves (4 evaluates to 4, #t to #t, etc.)
   (b) Call Expressions begin with a function name and are followed by arguments- evaluate function name, evaluate arguments, and apply function to arguments
   (c) Special Forms begin with a keyword and are followed by subexpressions, which are evaluated in a way based on the specific keyword
3. Useful Links
   (a) Scheme Specification (for overfiew, types, and special forms)
   (b) Scheme Built-In Procedure Reference (for built-in procedures)

Scheme is a *functional* language, as opposed to Python, which is an *imperative* language. A Python program is comprised of *statements* or "instructions," each of which directs the computer to take some action. (An example of a statement would be something like `x = 3` in Python. This does not evaluate to any value but just instructs Python to create a variable `x` with the value 3.) In contrast, a Scheme program is composed solely of (often heavily nested) *expressions*, each of which simply evaluates to a value.

The three basic types of expressions in Scheme are atomics/primitive expressions, call expressions, and special forms.

1.1 Atomics

Atomics are the simplest expressions. Some atomics, such as numbers and booleans, are called "self-evaluating" because they evaluate to themselves:

- `123` → `123`
- `-3.14` → `-3.14`
- `#t` → `#t ;booleans in scheme are #t and #f`

Symbols (variables) are also atomic expressions; they evaluate to the values to which they are bound. For example, the symbol + evaluates to the addition **procedure** (function) `#[+]`.

1.2 Call Expressions

A call expression is denoted with parentheses and is formed like so:

```
(<operator> <operand0> <operand1> ... <operand>)
```

Each "element" of a call expression is an expression itself and is separated from its neighbors by whitespace. All call expressions are evaluated in the same way:

1. Evaluate the operator, which will return a procedure.
2. Evaluate the operands.
3. Apply operator on operands.

For example, to evaluate the call expression `(+ (+ 1 2)2)`, we first evaluate the operator `+`, which returns the procedure `#[+]`. Then we evaluate the first operand `(+ 1 2)`, which returns `3`. Then, we evaluate the last operand `2`, receiving `2`. Finally, we apply the `#[+]` procedure to `3` and `2`, which returns `5`. So this call expression evaluates to `5`.

Note that in order to add two numbers, we had to call a function. In Python, `+` is a binary operator that can add two numbers without calling a function. Scheme has no such constructs, so even the most basic arithmetic requires you to call a function. The other math operators, including `-` (both subtraction and negation), `*`, `/`, `expt` (exponentiation), =, <, >, <=, and >= function in the same way.

1.3 Special Forms

Special forms *look* just like call expressions but are distinct in two ways:

1. One of the following keywords appears in the operator slot: **`define`**, **`if`**, **`cond`**, **`and`**, **`or`**, **`let`**, **`begin`**, **`lambda`**, **`quote`**, **`quasiquote`**, **`unquote`**, **`mu`**, **`define-macro`**, **`expect`**, **`unquote-splicing`**, **`delay`**, **`cons-stream`**, **`set!`**
2. They do not follow the evaluation rules for call expressions.

Below, we will go through a few commonly seen special forms.

1.3.1 `if` expression

```
(if <predicate> <true-expr> <false-expr>)
```

An **`if`** expression is similar to a Python **`if`** statement. First, evaluate `<predicate>`.

- If `<predicate>` is true, evaluate and return `<true-expr>`.
- If `<predicate>` is false, evaluate and return `<false-expr>`.

Note that everything in Scheme is truthy (including `0`) except for `#f`.

Also note that in Python, **`if`** is a statement, whereas in Scheme, **`if`** is an expression that evaluates to a value like any other expression would. In Scheme, you can then write something like this:

```
scm> (+ 1 (if #t 9 99))
10
```

Other special forms are also expressions that evaluate to values. Therefore, when we say "returns x," we mean "the special form evaluates to x."

1.3.2 cond expression

```
(cond
    (<predicate1> <expr1>)
    ...
    (<predicateN> <exprN>)
    (else <else-expr>))
```

A **`cond`** expression is similar to a Python **`if`**-`elif`-**`else`** statement. It is an alternative to using many nested **`if`** expressions.

- Evaluate `<predicate1>`. If it is true, evaluate and return `<expr1>`.
- Otherwise, continue down the list by evaluating `<predicate2>`. If it is true, evaluate and return `<expr2>`.
- Continue in this fashion down the list until you hit a true predicate.
- If every predicate is false, return `<else-expr>`.

1.3.3 and expression

```
(and <expr1> ... <exprN>)
```

`and` in Scheme works similarly to **`and`** in Python. Evaluate the expressions in order and return the value of the first false expression. If all of the values are true, return the last value. If no operands are provided, return `#t`.

1.3.4 `or` expression

```
(or <expr1> ... <exprN>)
```

`or` in Scheme works similarly to **`or`** in Python. Evaluate the expressions in order and return the value of the first true expression. If all of the values are false, return the last value. If no operands are provided, return `#f`.

1.3.5 `define` expression

`define` does two things. It can define variables, similar to the Python = assignment operator:

```
(define <symbol> <expr>)
```

This will evaluate `<expr>` and bind the resulting value to `<symbol>` in the current frame.

`define` is also used to define procedures.

```scheme
(define (<symbol> <op1> ... <opN>)
    <body>)
```

This code will create a new procedure that takes in the formal parameters <op1> ... <opN> and bind it to <symbol> in the current frame. When that procedure is called, the <body>, which may have multiple expressions, will be executed with the provided arguments bound to <op1> ... <opN>. The value of the final expression of <body> will be returned.

With either version of **define**, <symbol> is returned.

Dealing with the different types of **define** can be tricky. Scheme differentiates between the two by whether the first operand is a symbol or a list:

```scheme
(define (x) 1) ; like x = lambda: 1
(define x 1) ; like x = 1
```

1.3.6 lambda expressions

```scheme
(lambda (<op1> ... <opN>)
    <body>)
```

Returns a new procedure that takes in the formal parameters <op1> ... <opN>. When that procedure is called, the <body>, which may have multiple expressions, will be executed with the provided arguments bound to <op1> ... <opN>. The value of the final expression of <body> will be returned.

1.3.7 begin special form

```scheme
(begin
    <expr1>
    ...
    <exprN>)
```

Evaluates <expr1>, <expr2>, ..., <exprN> in order in the current environment. Returns the value of <exprN>.

1.3.8 let special form

```scheme
(let ((<symbol1> <expr1>)
      ...
      (<symbolN> <exprN>))
  <body>)
```

Evaluates <expr1>, ..., <exprN> in the current environment. Then, creates a new frame as a child of the current frame and binds the values of <expr1>, ..., <exprN> to <symbol1>,

..., <symbolN>, respectively, in that new frame. Finally, Scheme evaluates the <body>, which may have multiple expressions, in the new frame. The value of the final expression of <body> is be returned.

1.3.9 quote special form

```
(quote <expr>)
'<expr> ; shorthand syntax
```

Returns an expression that evaluates to <expr> *in its unevaluated form*. In other words, if you put '<expr> into the Scheme interpreter, you should get <expr> out *exactly*.

1.3.10 Summary of special forms

We have presented the main details of the most important special forms here, but this account is not comprehensive. Please see https://cs61a.org/articles/scheme-spec/ for a fuller explanation of the Scheme language.

behavior	syntax
if/else	(if <predicate> <true-expr> <false-expr>)
if/elif/else	(**cond** (<predicate1> <expr1>) ... (<predicateN> <exprN>) (**else** <else-expr>))
and	(and <expr1> ... <exprN>)
or	(or <expr1> ... <exprN>)
variable assignment	(define <symbol> <expr>)
function definition	(**define** (<symbol> <op1> ... <opN>) <body>)
lambdas	(**lambda** (<op1> ... <opN>) <body>)
evaluate many lines	(**begin** <expr1> ... <exprN>)
temporary environment	(**let** ((<symbol1> <expr1>) ... (<symbolN> <exprN>)) <body>)
quote	(quote <expr>) or '<expr>

1. What will Scheme output?

```
scm> (define pi 3.14)
```

```
pi
```

- Terminal will give you back the name of whatever you just defined

```
scm> pi
```

```
3.14
```

- In this case, you will get the value that pi is assigned to

```
scm> 'pi
```

```
pi
```

- Scheme doesn't evaluate the next expression after the quote.

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

```
3
```

```
scm> (+ 1 (* 3 4))
```

```
13
```

- This tests the order of operations in Scheme.

```
scm> (if 2 3 4)
```

```
3
```

```
scm> (if 0 3 4)
```

```
3
```

- Unlike Python, all Scheme values other than #f are true. Therefore, 0 is a true value, and this expression evaluates to 3.

```
scm> (- 5 (if #f 3 4))
```

```
1
```

```
scm> (if (= 1 1) 'hello 'goodbye)
```

```
hello
```

- In this case, there's a similar behavior to Python

- Emphasize the importance of short circuiting – it will be very important for the Scheme project

```
scm> (define (factorial n)
        (if (= n 0)
            1
            (* n (factorial (- n 1)))))
```

```
factorial
```

```
scm> (factorial 5)
```

```
120
```

Teaching Tips

- We can compare this factorial implementation in Scheme to the Python interpretation so that the students can see how the **`define`** keyword works
- If your students are not caught up with lecture/have had no practice with Scheme, try drawing parallels between control statements in Scheme and Python.

2. Define `apply-multiple` which takes in a single argument function `f`, a nonnegative integer `n`, and a value `x` and returns the result of applying `f` to `x` a total of `n` times.

```
;doctests
scm> (apply-multiple (lambda (x) (* x x)) 3 2)
256
scm> (apply-multiple (lambda (x) (+ x 1)) 10 1)
11
scm> (apply-multiple (lambda (x) (* 1000 x)) 0 5)
5
```

```
(define (apply-multiple f n x)

)
```

```
(define (apply-multiple f n x)
    (if (= n 0)
        x
        (f (apply-multiple f (- n 1) x))))
```

Alternate solution:

```
(define (apply-multiple f n x)
    (if (= n 0)
        x
        (apply-multiple f (- n 1) (f x))))
```

Teaching Tips

- Functions can get a little confusing to work with in Scheme so it helps to remind your students that they are all just pieces of data.
- Thinking about the base case first may be more helpful with this problem. When do you stop applying your function to the input? How do you know/which input will tell you when to stop? This will then provide a good idea of what the recursive calls should be.
- There are two alternate solutions to this issue that differ in what you call your function f on. Go along with whatever your students share and if possible share the alternate solutions so students can see different ways of recursing in Scheme.

3. Define a procedure called `hailstone`, which takes in two numbers `seed` and `n` and returns the `n`th number in the hailstone sequence starting at `seed`. Assume the hailstone sequence starting at `seed` has a length of at least `n`. As a reminder, to get the next number in the sequence, divide by 2 if the current number is even. Otherwise, multiply by 3 and add 1.

   Useful procedures

   - `quotient`: floor divides, much like `//` in python

     `(quotient 103 10)` outputs 10

   - `remainder`: takes two numbers and computes the remainder of dividing the first number by the second

     `(remainder 103 10)` outputs 3

```
; The hailstone sequence starting at seed = 10 would be
; 10 => 5 => 16 => 8 => 4 => 2 => 1

; Doctests
> (hailstone 10 0)
10
> (hailstone 10 1)
5
> (hailstone 10 2)
16
> (hailstone 5 1)
16

(define (hailstone seed n)









)
```

```scheme
(define (hailstone seed n)
    (if (= n 0)
        seed
        (if (= 0 (remainder seed 2))
            (hailstone
            (quotient seed 2)
            (- n 1))
          (hailstone
          (+ 1 (* seed 3))
          (- n 1)))))

; Alternative solution with cond

(define (hailstone seed n)
    (cond
        ((= n 0) seed)
        ((= 0 (remainder seed 2))
          (hailstone
          (quotient seed 2)
          (- n 1)))
        (else
          (hailstone
          (+ 1 (* seed 3))
          (- n 1)))))
```

Students have seen hailstone before. The goal with this problem is to get the students comfortable with Scheme by having them solve a familiar problem in an unfamiliar language. However, students may find this to be boring because they have seen it before. If this is the case, you can feel free to skip this problem. **Teaching Tips**

Python version:

```
def hailstone(seed, n):
    if n == 0:
        return seed
    if seed % 2 == 0:
        return hailstone(seed//2, n - 1)
    else:
        return hailstone(3*seed + 1, n - 1)
```

- If they're confused, point them towards the % in Python and how they can get the same value back in Scheme (answer: remainder function)
- Remind them to be careful about parentheses
- If you don't use cond, how might you write an equivalent function?
- In problems like this, I like to emphasize to my students how valuable it is to indent their code so that they can keep everything well organized. Scheme can be very confusing when not properly formatted because the language seems to just be an endless stream of parentheses.

2 Scheme Lists

Unlike Python, all Scheme lists are linked lists. Recall that, in Python, a linked list is made up of `Link`s that each have a `first` and a `rest`, where the `rest` is another `Link`. Similarly, each Scheme list is a "pair" where the first element of the pair is the first element of the list, and the second element of the pair is the rest of the list (also a pair).

We use the `cons` procedure to construct Scheme lists, and `nil` to represent empty lists. The sequence $1, 2, 3$ may then be represented as follows:

```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

It's worth pointing out to your students that, unlike with the `Link` class, the `nil` must be explicitly provided at the end of the linked list.

The `car` and `cdr` procedures are used to access the elements of a Scheme list. `car` gets the first element of a list, while `cdr` gets the rest of the list:

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

```
lst
scm> (car lst)
1
scm> (cdr lst)
(2 3)
```

You can make the following analogy between linked lists in Python and Scheme:

`Link(1, Link.empty)`	`(cons 1 nil)`
`a = Link(1, Link(2, Link.empty))`	`(define a (cons 1 (cons 2 nil)))`
`a.first`	`(car a)`
`a.rest`	`(cdr a)`

The `list` procedure and quotation give us additional convenient ways to construct lists:

```
scm> (list 1 2 3)
(1 2 3)
scm> '(1 2 3)
(1 2 3)
scm> (list 1 (+ 1 1) 3)
(1 2 3)
scm> '(1 (+ 1 1) 3)
(1 (+ 1 1) 3)
```

Note that quotation will prevent any of the list items from being evaluated, which can occasionally be inconvenient.

If relevant, I like to discuss when it makes the most sense to use the different ways of constructing a list.

- `cons` is useful when you have a way to construct the first element and rest of the list, e.g. in recursive problem solving,
- `list` and quotation are useful when you want to hardcode a list into your code beforehand, but typically aren't that useful if you want to dynamically create a list based on program input.

2.1 Useful procedures

In addition to the procedures mentioned above, the following procedures are often useful when dealing with Scheme lists:

- `(null? s)`: returns true if `s` is `nil`.
- `(length s)`: returns the length of `s`.
- `(append s1 ... sn)`: returns the result of concatenating lists `s1`, ..., `sn`.

- `(map f s)`: returns the result of applying the procedure `f` to each element of `s`.
- `(filter pred s)`: returns a list containing the elements of `s` for which the single-argument procedure `pred` returns true.
- `(reduce comb s)`: combines the elements of `s` into a single value using the two-argument procedure `comb`.

2.2 Equality testing

Equality testing in Scheme is a bit confusing as it is handled by three separate procedures:

- `(= a b)`: returns true if `a` equals `b`. Both must be numbers.
- `(eq? a b)`: returns true if `a` and `b` are equivalent primitive values. For two objects, `eq?` returns true if both refer to the exactly same object in memory (like `is` in Python).
- `(equal? a b)`: returns true if `a` and `b` are equivalent. Two lists are equivalent if their elements are equivalent.

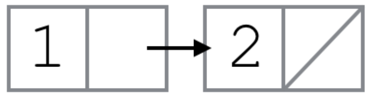
Teaching Tips

- For the love of God, please do not mini-lecture all of this stuff. This information is presented as a reference for you, and you should ask your students what they would like to go over so that you do not waste their time.
- Emphasize to students that Scheme lists are linked lists and NOT Python lists
  - Discuss the limitations (e.g. no indexing) and capabilities (e.g. recursion)
- If you're an old bear™, keep in mind that dotted lists (thank god) have been removed from the curriculum, so Scheme lists have the same functionality as linked lists
- The 61A Scheme Web interpreter is **very useful** for visualizing lists!
- If you choose to give a mini-lecture on Scheme list syntax, try using each keyword in an example instead of just talking about them!

1. What will Scheme output? Draw box-and-pointer diagrams to help determine this. (Ask your mentor if you're unsure what's going on. You aren't expected to understand this completely on your own.)

```
scm> (cons 1 (cons 2 nil))
```

```
(1 2)
```



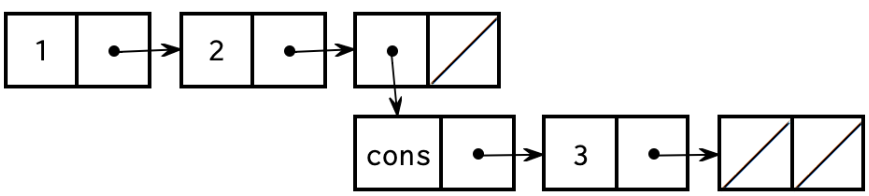
```
scm> (cons 1 '(2 3 4 5))
```

```
(1 2 3 4 5)
```

When we use the quote before the list, we are saying that we should put the literal list (2 3 4 5) in the cdr of this list. So in this case we create a list where the first element (car) is 1, and the cdr is the list (2 3 4 5).

```
scm> (cons 1 '(2 (cons 3 nil)))
```

```
(1 2 (cons 3 ()))
```



Since we also used a quote here, we do not evaluate the `(cons 3 nil)`. We keep everything inside the quotes the same so the `cdr` of this list is the list `(2 (cons 3 nil))`. That means that we add the element 2, and then the nested list `(cons 3 nil)`.

```
scm> (cons 1 (2 (cons 3 nil)))
```

```
eval: bad function in : (2 (cons 3 nil))
```

While evaluating the operands, Scheme will try to evaluate the expression `(2 (cons 3 nil))`. Since 2 is not a valid operator, this expression Errors.

```
scm> (cons 3 (cons (cons 4 nil) nil))
```

```
(3 (4))
```

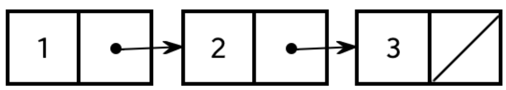
```
scm> (define a '(1 2 3))
```

```
a
```

Defines a list of elements of `(1 2 3)` and binds the list to the variable `a`. Recall that define returns the name of the symbol.

```
scm> a
```

```
(1 2 3)
```



```
scm> (car a)
```

```
1
```

```
scm> (cdr a)
```

```
(2 3)
```

```
scm> (car (cdr a))
```

```
2
```

From above, we know that `(cdr a)` is `(2 3)`. From that, we can evaluate `(car (cdr a))` to 2.

How can we get the 3 out of a?

```
(car (cdr (cdr a)))
```

To get to the pair that contains 3, we need to call `(cdr (cdr a))`. To get the element 3, we need the `car` of `(cdr (cdr a))`.

Teaching Tips

- Draw diagrams or use the 61A Scheme Web interpreter for visualizing lists
- Encourage students to ask questions and experiment with extra `cons`, `car`, and `cdr` statements to see how they change the outputs of statments!
- While unrelated to the problem, it may be helpful to teach students these keywords:
  - `(pair? arg)`, which checks if arg has a first and rest
  - `(list? arg)`, which returns true if arg is a well-formed list

2. Define `is-prefix`, which takes in a list `p` and a list `lst` and determines if `p` is a prefix of `lst`. That is, it determines if `lst` starts with all the elements in `p`.

```
; Doctests:
scm> (is-prefix '() '())
#t
scm> (is-prefix '() '(1 2))
#t
scm> (is-prefix '(1) '(1 2))
#t
scm> (is-prefix '(2) '(1 2))
#f
; Note here p is longer than lst
scm> (is-prefix '(1 2) '(1))
#f

(define (is-prefix p lst)




)
```

```scheme
; is-prefix with nested if statements
(define (is-prefix p lst)
    (if (null? p)
        #t
        (if (null? lst)
            #f
            (and
                (= (car p) (car lst))
                (is-prefix (cdr p) (cdr lst))))))

; is-prefix with a cond statement
(define (is-prefix p lst)
    (cond
        ((null? p) #t)
        ((null? lst) #f)
        (else (and (= (car p) (car lst))
            (is-prefix (cdr p) (cdr lst))))))
```

Teaching Tips

- Encourage students to think about how they would solve this problem without starter code. How would you determine if a given input matches the first part of another input? Iteration! Then translate this iteration into Scheme.
- Be sure to check for null cases or edge cases, keep track of parentheses, and keep in mind how true and false are represented in Scheme.
- Remind students also that there are two ways to go about checking different cases in Scheme: nested ifs or a cond statement.
- As a hint, also consider suggesting figuring out the logic with pseudo or Python code, then translating into Scheme.

3. Implement `argmax`, a function that takes in a list, `lst`, and returns the index of the largest element in `lst`. If there are two or more elements that are the largest element, return the index of the one that appears first in `lst`.

   You can assume all elements of `lst` are non-negative integers, and `lst` has at least 1 element and no nested lists.

```scheme
(define (argmax lst)
    (define (max-helper lst max-so-far max-index curr-index)
        (cond

            ((______________________) ______________________________)
```

```
            ((____________________) __________________________

              ___________________________________)

            (else _____________________________________________)
        )
    )

    (max-helper __________________________)
)
```

```
(define (argmax lst)
    (define (max-helper lst max-so-far max-index curr-index)
        (cond
            ((null? lst) max-index)
            ((> (car lst) max-so-far)
                (max-helper (cdr lst) (car lst) curr-index (+
                  curr-index 1)))
            (else
                (max-helper (cdr lst) max-so-far max-index (+
                  curr-index 1)))
        )
    )
    (max-helper lst 0 0 0)
)
```

It's important that students learn how to use helper functions in Scheme; since there is no iteration, most anything that cannot be done through pure recursion will have to be done via a helper function.