Introduction to Scheme Meta

Computer Science Mentors 61A

April 10–April 14, 2023

Recommended Timeline

- Midterm #2 Check-in: 5 mins
- Scheme Guided Minilecture: 25 mins
- If, and, or WWSD: 6 mins
- Define Eval: 7 mins
- Hailstone: 15 mins

This section is meant to act as your students' first introduction to Scheme, as they haven't even been exposed to it during lecture (we're just giving them a head start ;-)). You'll notice that the minilecture this time around is long. This is because we've added check-in WWSD questions to go along with explanations.

Teaching Tips

- To ease in Scheme, it can help to start by comparing and contrasting with Python
- Have students write a basic function in Python (like an iterative countdown), then replicate it in Scheme
- Have students list language features of Python (variable assignments, conditional statements, logic operators, etc.), and explain how Scheme implements those features
- Make sure to give a disclaimer that while high level features may be analogous, the internals are different!
- Scheme features break into three broad categories: Primitives, Call Expressions, and Special Forms (the latter two are called Compound Expressions)
- Primitives evaluate to themselves (4 evaluates to 4, #t to #t, etc.)
- Call Expressions begin with a function name and are followed by arguments- evaluate function name, evaluate arguments, and apply function to arguments
- Special Forms begin with a keyword and are followed by subexpressions, which are evaluated in a way based on the specific keyword

1 Scheme

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

Created by Alyssa Smith, Esther Shen, Maya Romero, Natalie Wei, Sandhya Ganesan, Kaelyn Huang, Cecilia Aiko, Aurelia Wang

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) `#[+]`.

Let's practice identifying atomic expressions in Scheme.

```
    True or False: 3.14 is an atomic expression.
```

True. Floats are atomic expressions.

```
    True or False: pi is an atomic expression.
```

False. Pi in most cases isn't naturally defined within the programming language -- rather, it's a variable name. As we haven't defined pi in this case, Scheme doesn't know what to interpret pi as, so we get an Error if we try to evaluate this on its own.

```
    True or False: - is an atomic expression.
```

True. - is an operator for subtraction.

```
    True or False: // is an atomic expression.
```

False. Scheme can only interpret the basic symbols of +, -, *, /. We'll have to define our own floor division function!

```
    True or False: 'b' is an atomic expression.
```

True. 'b' is a character, which is an atomic data type!

```
    True or False: "is this atomic?" is an atomic expression.
```

True. Even though a string is a list of characters, Scheme still treats them as an atomic. It's important to note that in Scheme's case, strings can only be denoted with ".

In general, if you want to check if an input is atomic, use (atom? [input]) in the Scheme interpreter.

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.

Let's do some practice with call expressions. What would Scheme do for each expression?

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

13

    scm> (* 1 (+ 3 4))

7

    scm> (/ 2)

0.5 || When Scheme sees only one operand in the case of multiplication and
division, it fills in <operand0> to be 1.

    scm> (- 2)

-2 || When Scheme sees only one operand in the case of addition and
subtraction, it fills in <operand0> to be 0.

    scm> (* 4 (+ 3 (- 2 (/ 1))))

16
```

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>`. The general structure follows (if `<predicate>` `<true-expr>` `<false-expr>`).

- 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>`.

Let's practice using **`if`**s and **`cond`**s to evaluate Scheme problems!

```
scm> (if 2 3 4)
3
scm> (if 0 3 4)
4
scm> (- 5 (if #f 3 4))
1
scm> (cond ((< -5 -7) 3)
           (else 4))
4
```

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`.

Let's practice and/or statements!

```
    scm> (and #t (= 3 3) (> (- 61 42) (+ 61 42)))

#f

    scm> (or #f (< 3 3) (< (- 61 42) (+ 61 42)))

#t
```

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.

```
(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:

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

1.3.6 `lambda` expressions

```
(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

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

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

1.3.8 `let` special form

```
(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?.

   (a) `(if 0 (/ 1 0) 1)`

   ```
   Error: Zero Division
   ```

   Recall that 0 is a Truth-y value in Scheme. Thus (/ 1 0) evaluates to a Zero Division Error

   (b) `(and 1 #f (/ 1 0))`

   ```
   #f
   ```

   Short-circuiting rules apply. This means that and returns the first False-y value or the last Truth-y value. In this case, the first False-y value is #f.

   (c) `(and 1 2 3)`

   ```
   3
   ```

   Short-circuiting rules apply. This means that and returns the first False-y value or the last Truth-y value. In this case, the last Truth-y value is 3.

   (d) `(or #f #f 0 #f (/ 1 0))`

   ```
   0
   ```

   Short-circuiting rules apply. This means that or returns the first Truth-y value or the last False-y value. In this case, the first Truth-y value is 0.

   (e) `(or #f #f (/ 1 0) 3 4)`

   ```
   Error: Zero Division
   ```

   Short-circuiting rules apply. This means that or returns the first Truth-y value or the last False-y value. In this case, (/ 1 0) evaluates in a Zero Division Error.

   (f) `(and (and) (or))`

   ```
   #f
   ```

   The special form or without any arguments evaluates to #f. The special form and without any arguments evaluates to #t. Also, short-circuiting rules apply. This means that and returns the first False-y value or the last Truth-y value. In this case, the first False-y value is #f.

2. What will Scheme output?

```
scm> (define c 2)

c

scm> (eval 'c)

2

scm> '(cons 1 nil)

(cons 1 nil)

scm> (eval '(cons 1 nil))

(1)

scm> (eval (list 'if '(even? c) 1 2))

1
```

Teaching Tips

- Quotation marks are easily one of the trickiest concepts in Scheme, so spend a lot of time making sure your students understand these problems thoroughly! It would help to do a quick mini lecture or review of quotation marks if your students need.
- In some cases it is easier to think of the single quotation mark as double quotes in Python that encompass a string, such as in "standalone" expressions like the third one. In such cases the exact "string" is returned
- Encourage students to make the connection between Scheme lists and Scheme expressions as often as they can. Being able to read Scheme expressions as lists will be very helpful in the future.
- Whenever there is an eval with a quote, you go down one level of evaluation and essentially pretend the quote is not there.
- Going off of the third point, it could be beneficial to model Scheme expressions on a pyramid of evaluation: exact string, evaluating the string, etc.

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)
```

```
(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.