INTRODUCTION TO SCHEME

COMPUTER SCIENCE MENTORS

November 2, 2020 - November 5, 2020

Scheme is a *functional* language, as opposed to Python, which is an *imperative* language; whereas a Python program is comprised of *statements* or "instructions" which do not evaluate to a value (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), a Scheme program is comprised solely of *expressions*, each of which simply evaluates to a value. Remember that the term evaluate means to find the value of something. A variable is evaluated by looking up the name in the current frame, and a function call expression is evaluated using the three steps listed below. The thing to notice is that different types of expressions have different rules for how to evaluate those expressions, and this goes for both Python and Scheme.

The four basic types of expressions in Scheme are literals (i.e., a value itself), call expressions (procedure calls), special forms (language features), and variables. A call expression or special form is denoted by a pair of parentheses and takes prefix notation, i.e., it is formed as so:

```
(operator operand_0, operand_1, ... , operand_n)
```

(Keep in mind each item in a call expression is also an expression)

Evaluation of a call expression progresses so:

1. Evaluate operator (returning a procedure)
2. Evaluate operands
3. Apply operator on operands

If Expression: The **if** keyword is similar to **if**/**else** statements in Python. It works as follows:

```
(if <predicate> <do if true> <do if false>)
```

This is similar to the following code in Python:

```
if <predicate>:
    <do if true>
else:
    <do if false>
```

Note that in Python, **if** is a statement whereas in Scheme, **if** is an expression and evaluates to a value like any other expression would. This means that in Scheme you could write something like this where the **if** expression can be placed as an operand in a function call expression:

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

Just like in Python, the `<do if false>` (in Python this would be the equivalent to the else clause) is optional. If there is no `<do if false>` and the `<predicate>` evaluates to false then the **if** expression evaluates to `undefined`.

Cond Expression: The `cond` expression comes in handy when you want to do a sequence of **if** statements. It works as follows:

```
(cond
    (predicate1 (action1))
    (predicate2 (action2))
    ...
    (else (final-action))
)
```

Just like the **if**/**else** expressions, Scheme will evaluate each predicate and execute the corresponding action only if the predicate is true. Like always, the **else** predicate is optional.

You can think of the `cond` expression as a cleaner way to write nested **if** statements in Scheme, which would look like the following:

```
(if <predicate1>
    <action1>
    (if <predicate2>
        <action2>
        (if ...
            ...
            final-action)))
```

Finally, `cond` is similar to the following code in Python:

```
if <predicate1>:
    <action1>
elif <predicate2>:
    <action2>
...
else:
    <final-action>
```

Define Expression: `define` does two things in Scheme. The first is that it defines variables using the following syntax:

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

The way this works is Scheme will evaluate `<expression>` and bind the value to `<name>` in the current environment. `<name>` must be a valid Scheme symbol (you can think of a symbol as an identifier or variable name).

`define` is also used to define functions using the following syntax (note that this is different from using `define` to create variables as there is an extra pair of parentheses around `<name> [param] ...`):

```
(define (<name> [param] ...) <body> ...)
```

Either way, after the `<name>` is bound to either a function or value, the define expression evaluates to the symbol `<name>`.

```
scm> (define x 3)
x
```

`links.cs61a.org/schemespec` will direct you to a page with all of the explanations and syntax descriptions of Scheme. Use it when filling out the WWSD section!

1 What Would Scheme Print?

1. What will Scheme output?

```
scm> (define pi 3.14)

scm> pi

scm> 'pi

scm> (+ 1 2)

scm> (+ 1 (* 3 4))

scm> (if 2 3 4)

scm> (if 0 3 4)

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

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

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

scm> (factorial 5)
```

2 Code Writing in Scheme

2. **Hailstone yet again** Define a program called `hailstone`, which takes in two numbers `seed` and `n`, and returns the `n`*th* hailstone number in the sequence starting at `seed`. Assume the hailstone sequence starting at `seed` is longer or equal to `n`. As a reminder, to get the next number in the sequence, if the number is even, divide by 2. Else, 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)










)
```

3 Scheme Lists

Unlike Python, all Scheme lists are linked lists. Recall a linked list is made up of Links that have a first and a rest, where the rest is another Link. Similarly, Scheme lists are made up of pairs with a first and a rest, where the rest is another pair.

Ways to make scheme lists:

- Cons
  Syntax: `(cons <car-elem> <cdr-elem>)`
  Takes in a pair of two elements; similar to how a python linked list has 2 elements as well- first and rest
- List
  Syntax: `(list <elem1> <elem2> ...)`
  Takes in an arbitrary number of elements/arguments, and constructs a list where each elem is the first of its own pair. Note how this differs from `cons` where you specify a first and rest rather than just specifying the first of each pair. All the arguments will be evaluated before being collected into the scheme list.
- `'` (aka single quote)
  Syntax: `'(<elem1> <elem2> ...)`
  Also takes in an arbitrary number of elements and construct a list out of the elements, but the arguments are not evaluated.

Ways to access list items:

- Car
  Syntax: `(car <pair>)`
  Gets you the first item of a pair
- Cdr
  Syntax: `(cdr <pair>)`
  Gets you the second item of a pair
- Cadr
  Syntax: `(cadr <pair>)`
  Gets you the `car` of the `cdr`
- Cddr
  Syntax: `(cddr <pair>)`
  Gets you the `cdr` of the `cdr`

You can make the following analogy:

`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)`

Draw box and pointers when appropriate. Ask your mentor if you're unsure what's going on. You aren't expected to understand this completely on your own.

3. What will Scheme output? Draw box-and-pointer diagrams to help determine this.

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

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

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

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

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


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

scm> a

scm> (car a)

scm> (cdr a)

scm> (cadr a)
```

How can we get the 3 out of a?

4 More Code Writing in Scheme

4. Implement `waldo`. `waldo` returns `#t` if the symbol waldo is in a list.

```
scm> (waldo '(1 4 waldo))
#t
scm> (waldo '())
#f
scm> (waldo '(1 4 9))
#f

(define (waldo lst)
```

5. **Extra challenge:** Define `waldo` so that it returns the index of the list where the symbol waldo was found (if waldo is not in the list, return `#f`).

```
scm> (waldo '(1 4 waldo))
2
scm> (waldo '())
#f
scm> (waldo '(1 4 9))
#f
```

```
(define (waldo lst)

)
```