INTRODUTION TO SCHEME

COMPUTER SCIENCE MENTORS 61A

March 31 – April 4, 2025

1 Scheme

1.1 Helpful Links:

- Scheme Specifications: Overview of the Scheme language with information on the types of values and the special forms in Scheme (https://cs61a.org/articles/scheme-spec/)
- Scheme Built-In Procedures: Description of basic built-in procedures in the 61A Scheme interpreter (https://cs61a.org/articles/scheme-builtins/)
- Scheme Staff Interpreter: Scheme interpreter where you can play around with Scheme (https://code.cs61a.org/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 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.

1.2 Atomics

Atomics are the simplest expressions. Some atomics, such as numbers, booleans, strings, (and the empty list (nil), which you will learn about later), are called "self-evaluating" because they evaluate to themselves:

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

Anything which is self-evaluating is automatically an atomic. **Symbols** are also atomic expressions; they evaluate to the values to which they are bound. For example, the symbol `+` evaluates to the addition **procedure** (function) `#[+]`, which is already built into Scheme.

Created by Jim Fang, Bryan Adolfo Ventura Benitez, Rajoshi Basu Worksheet feedback: links.jimfang.dev/feedback

Let's practice identifying atomic expressions in Scheme.

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

True or False: pi is an atomic expression.

True or False: - is an atomic expression.

True or False: // is an atomic expression.

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

1.3 Call Expressions

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

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

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 evaluated operator got from first step on evaluated operands got from second step.

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))

scm> (* 1 (+ 3 4))

scm> (/ 2)

scm> (- 2)

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

1.3.1 `eval` expression

`(`**`eval`** `<operand>)`

Another operator we can include in call expressions like `+`, `-`, etc. is the **`eval`** operator. Here, first **`eval`** is evaluated to the `#[eval]` procedure. Then we evaluate the operand and then we apply the `#[eval]` to the evaluated operand, which evaluates the evaluated operand again. For instance, if we had `(`**`eval`** `(+ 1 2))`, then in the final step we would evaluate 3 and return 3 as the value of the expression.

1.4 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.

1.4.1 `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."

1.4.2 `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)
```

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

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

```
scm> (cond ((< -5 -7) 3)
           (else 4))
```

1.4.3 `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 (which is always `#f`). If all of the values are true, return the last value. If no operands are provided, return `#t`.

1.4.4 `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 1 2)


scm> (or #f 1 2)


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


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

1.4.5 `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.

1.4.6 `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 lambda procedure that takes in the formal parameters `<op1> ... <opN>` and has the body `<body>`. Then, `define` binds this new lambda procedure 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. This shortcut is basically equivalent to this:

```
(define <name> (lambda (<op1> ... <opN>) <body>))
```

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
```

1.4.7 `begin` special form

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

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

1.4.8 `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.

1.4.9 `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*.

1.4.10 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 the Scheme Specification 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)`

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

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

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

   (e) `(and (and) (or))`

2. What will Scheme output?

```
scm> (define c 4)

scm> ((define (x) 1))

scm> (x)

scm> ((lambda (x y) (+ c)) 1 2)

scm> (eval 'c)

scm> '(cons 1 nil)

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

scm> (let ((a (+ 3 1)) (b 3)) (+ a b) (/ a b))
```

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)

    (if (_____________)

        _____________

        (if (_____________(_____________________________))

            (_____________

                (_____________________)

                (_____________________)
            )

            (_____________

                (+ _____ (* _____________))

                (_____________)
            )
        )
    )
)
```